hyperledger-fabricdocs Documentation

Release master

hyperledger

Apr 23, 2020

Contents

1 Introduction 3

2 What's new in Hyperledger Fabric v2.x 9

3 Release notes 13

4 Key Concepts 15

5 Getting Started 123

6 Developing Applications 141

7 Tutorials 201

8 Deploying a production network 335

9 Operations Guides 343

10 Upgrading to the latest release 419

11 Commands Reference 441

12 Architecture Reference 497

13 Frequently Asked Questions 521

14 Contributions Welcome! 525

15 Glossary 541

16 Releases 551

17 Still Have Questions? 553

18 Status 555

Note: Please make sure you are looking at the documentation that matches the version of the software you are using. See the version label at the top of the navigation panel on the left. You can change it using selector at the bottom of that navigation panel.

Enterprise grade permissioned distributed ledger platform that offers modularity and versatility for a broad set of industry use cases.

CHAPTER 1

Introduction

In general terms, a blockchain is an immutable transaction ledger, maintained within a distributed network of *peer nodes*. These nodes each maintain a copy of the ledger by applying transactions that have been validated by a *consensus protocol*, grouped into blocks that include a hash that bind each block to the preceding block.

The first and most widely recognized application of blockchain is the Bitcoin cryptocurrency, though others have followed in its footsteps. Ethereum, an alternative cryptocurrency, took a different approach, integrating many of the same characteristics as Bitcoin but adding *smart contracts* to create a platform for distributed applications. Bitcoin and Ethereum fall into a class of blockchain that we would classify as *public permissionless* blockchain technology. Basically, these are public networks, open to anyone, where participants interact anonymously.

As the popularity of Bitcoin, Ethereum and a few other derivative technologies grew, interest in applying the underlying technology of the blockchain, distributed ledger and distributed application platform to more innovative *enterprise* use cases also grew. However, many enterprise use cases require performance characteristics that the permissionless blockchain technologies are unable (presently) to deliver. In addition, in many use cases, the identity of the participants is a hard requirement, such as in the case of financial transactions where Know-Your-Customer (KYC) and Anti-Money Laundering (AML) regulations must be followed.

For enterprise use, we need to consider the following requirements:

- Participants must be identified/identifiable
- Networks need to be *permissioned*
- High transaction throughput performance
- Low latency of transaction confirmation
- Privacy and confidentiality of transactions and data pertaining to business transactions

While many early blockchain platforms are currently being *adapted* for enterprise use, Hyperledger Fabric has been *designed* for enterprise use from the outset. The following sections describe how Hyperledger Fabric (Fabric) differentiates itself from other blockchain platforms and describes some of the motivation for its architectural decisions.

1.1 Hyperledger Fabric

Hyperledger Fabric is an open source enterprise-grade permissioned distributed ledger technology (DLT) platform, designed for use in enterprise contexts, that delivers some key differentiating capabilities over other popular distributed ledger or blockchain platforms.

One key point of differentiation is that Hyperledger was established under the Linux Foundation, which itself has a long and very successful history of nurturing open source projects under **open governance** that grow strong sustaining communities and thriving ecosystems. Hyperledger is governed by a diverse technical steering committee, and the Hyperledger Fabric project by a diverse set of maintainers from multiple organizations. It has a development community that has grown to over 35 organizations and nearly 200 developers since its earliest commits.

Fabric has a highly **modular** and **configurable** architecture, enabling innovation, versatility and optimization for a broad range of industry use cases including banking, finance, insurance, healthcare, human resources, supply chain and even digital music delivery.

Fabric is the first distributed ledger platform to support **smart contracts authored in general-purpose programming languages** such as Java, Go and Node.js, rather than constrained domain-specific languages (DSL). This means that most enterprises already have the skill set needed to develop smart contracts, and no additional training to learn a new language or DSL is needed.

The Fabric platform is also **permissioned**, meaning that, unlike with a public permissionless network, the participants are known to each other, rather than anonymous and therefore fully untrusted. This means that while the participants may not *fully* trust one another (they may, for example, be competitors in the same industry), a network can be operated under a governance model that is built off of what trust *does* exist between participants, such as a legal agreement or framework for handling disputes.

One of the most important of the platform's differentiators is its support for **pluggable consensus protocols** that enable the platform to be more effectively customized to fit particular use cases and trust models. For instance, when deployed within a single enterprise, or operated by a trusted authority, fully byzantine fault tolerant consensus might be considered unnecessary and an excessive drag on performance and throughput. In situations such as that, a crash fault-tolerant (CFT) consensus protocol might be more than adequate whereas, in a multi-party, decentralized use case, a more traditional byzantine fault tolerant (BFT) consensus protocol might be required.

Fabric can leverage consensus protocols that **do not require a native cryptocurrency** to incent costly mining or to fuel smart contract execution. Avoidance of a cryptocurrency reduces some significant risk/attack vectors, and absence of cryptographic mining operations means that the platform can be deployed with roughly the same operational cost as any other distributed system.

The combination of these differentiating design features makes Fabric one of the **better performing platforms** available today both in terms of transaction processing and transaction confirmation latency, and it enables **privacy and confidentiality** of transactions and the smart contracts (what Fabric calls "chaincode") that implement them.

Let's explore these differentiating features in more detail.

1.2 Modularity

Hyperledger Fabric has been specifically architected to have a modular architecture. Whether it is pluggable consensus, pluggable identity management protocols such as LDAP or OpenID Connect, key management protocols or cryptographic libraries, the platform has been designed at its core to be configured to meet the diversity of enterprise use case requirements.

At a high level, Fabric is comprised of the following modular components:

- A pluggable *ordering service* establishes consensus on the order of transactions and then broadcasts blocks to peers.

- A pluggable *membership service provider* is responsible for associating entities in the network with cryptographic identities.
- An optional *peer-to-peer gossip service* disseminates the blocks output by ordering service to other peers.
- Smart contracts ("chaincode") run within a container environment (e.g. Docker) for isolation. They can be written in standard programming languages but do not have direct access to the ledger state.
- The ledger can be configured to support a variety of DBMSs.
- A pluggable endorsement and validation policy enforcement that can be independently configured per application.

There is fair agreement in the industry that there is no "one blockchain to rule them all". Hyperledger Fabric can be configured in multiple ways to satisfy the diverse solution requirements for multiple industry use cases.

1.3 Permissioned vs Permissionless Blockchains

In a permissionless blockchain, virtually anyone can participate, and every participant is anonymous. In such a context, there can be no trust other than that the state of the blockchain, prior to a certain depth, is immutable. In order to mitigate this absence of trust, permissionless blockchains typically employ a "mined" native cryptocurrency or transaction fees to provide economic incentive to offset the extraordinary costs of participating in a form of byzantine fault tolerant consensus based on "proof of work" (PoW).

Permissioned blockchains, on the other hand, operate a blockchain amongst a set of known, identified and often vetted participants operating under a governance model that yields a certain degree of trust. A permissioned blockchain provides a way to secure the interactions among a group of entities that have a common goal but which may not fully trust each other. By relying on the identities of the participants, a permissioned blockchain can use more traditional crash fault tolerant (CFT) or byzantine fault tolerant (BFT) consensus protocols that do not require costly mining.

Additionally, in such a permissioned context, the risk of a participant intentionally introducing malicious code through a smart contract is diminished. First, the participants are known to one another and all actions, whether submitting application transactions, modifying the configuration of the network or deploying a smart contract are recorded on the blockchain following an endorsement policy that was established for the network and relevant transaction type. Rather than being completely anonymous, the guilty party can be easily identified and the incident handled in accordance with the terms of the governance model.

1.4 Smart Contracts

A smart contract, or what Fabric calls "chaincode", functions as a trusted distributed application that gains its security/trust from the blockchain and the underlying consensus among the peers. It is the business logic of a blockchain application.

There are three key points that apply to smart contracts, especially when applied to a platform:

- many smart contracts run concurrently in the network,
- they may be deployed dynamically (in many cases by anyone), and
- application code should be treated as untrusted, potentially even malicious.

Most existing smart-contract capable blockchain platforms follow an **order-execute** architecture in which the consensus protocol:

- validates and orders transactions then propagates them to all peer nodes,
- each peer then executes the transactions sequentially.

The order-execute architecture can be found in virtually all existing blockchain systems, ranging from public/permissionless platforms such as Ethereum (with PoW-based consensus) to permissioned platforms such as Tendermint, Chain, and Quorum.

Smart contracts executing in a blockchain that operates with the order-execute architecture must be deterministic; otherwise, consensus might never be reached. To address the non-determinism issue, many platforms require that the smart contracts be written in a non-standard, or domain-specific language (such as Solidity) so that non-deterministic operations can be eliminated. This hinders wide-spread adoption because it requires developers writing smart contracts to learn a new language and may lead to programming errors.

Further, since all transactions are executed sequentially by all nodes, performance and scale is limited. The fact that the smart contract code executes on every node in the system demands that complex measures be taken to protect the overall system from potentially malicious contracts in order to ensure resiliency of the overall system.

1.5 A New Approach

Fabric introduces a new architecture for transactions that we call **execute-order-validate**. It addresses the resiliency, flexibility, scalability, performance and confidentiality challenges faced by the order-execute model by separating the transaction flow into three steps:

- *execute* a transaction and check its correctness, thereby endorsing it,
- *order* transactions via a (pluggable) consensus protocol, and
- *validate* transactions against an application-specific endorsement policy before committing them to the ledger

This design departs radically from the order-execute paradigm in that Fabric executes transactions before reaching final agreement on their order.

In Fabric, an application-specific endorsement policy specifies which peer nodes, or how many of them, need to vouch for the correct execution of a given smart contract. Thus, each transaction need only be executed (endorsed) by the subset of the peer nodes necessary to satisfy the transaction's endorsement policy. This allows for parallel execution increasing overall performance and scale of the system. This first phase also **eliminates any non-determinism**, as inconsistent results can be filtered out before ordering.

Because we have eliminated non-determinism, Fabric is the first blockchain technology that **enables use of standard programming languages**.

1.6 Privacy and Confidentiality

As we have discussed, in a public, permissionless blockchain network that leverages PoW for its consensus model, transactions are executed on every node. This means that neither can there be confidentiality of the contracts themselves, nor of the transaction data that they process. Every transaction, and the code that implements it, is visible to every node in the network. In this case, we have traded confidentiality of contract and data for byzantine fault tolerant consensus delivered by PoW.

This lack of confidentiality can be problematic for many business/enterprise use cases. For example, in a network of supply-chain partners, some consumers might be given preferred rates as a means of either solidifying a relationship, or promoting additional sales. If every participant can see every contract and transaction, it becomes impossible to maintain such business relationships in a completely transparent network — everyone will want the preferred rates!

As a second example, consider the securities industry, where a trader building a position (or disposing of one) would not want her competitors to know of this, or else they will seek to get in on the game, weakening the trader's gambit.

In order to address the lack of privacy and confidentiality for purposes of delivering on enterprise use case requirements, blockchain platforms have adopted a variety of approaches. All have their trade-offs.

Encrypting data is one approach to providing confidentiality; however, in a permissionless network leveraging PoW for its consensus, the encrypted data is sitting on every node. Given enough time and computational resource, the encryption could be broken. For many enterprise use cases, the risk that their information could become compromised is unacceptable.

Zero knowledge proofs (ZKP) are another area of research being explored to address this problem, the trade-off here being that, presently, computing a ZKP requires considerable time and computational resources. Hence, the trade-off in this case is performance for confidentiality.

In a permissioned context that can leverage alternate forms of consensus, one might explore approaches that restrict the distribution of confidential information exclusively to authorized nodes.

Hyperledger Fabric, being a permissioned platform, enables confidentiality through its channel architecture and private data feature. In channels, participants on a Fabric network establish a sub-network where every member has visibility to a particular set of transactions. Thus, only those nodes that participate in a channel have access to the smart contract (chaincode) and data transacted, preserving the privacy and confidentiality of both. Private data allows collections between members on a channel, allowing much of the same protection as channels without the maintenance overhead of creating and maintaining a separate channel.

1.7 Pluggable Consensus

The ordering of transactions is delegated to a modular component for consensus that is logically decoupled from the peers that execute transactions and maintain the ledger. Specifically, the ordering service. Since consensus is modular, its implementation can be tailored to the trust assumption of a particular deployment or solution. This modular architecture allows the platform to rely on well-established toolkits for CFT (crash fault-tolerant) or BFT (byzantine fault-tolerant) ordering.

Fabric currently offers a CFT ordering service implementation based on the `etcd` library of the Raft protocol. For information about currently available ordering services, check out our conceptual documentation about ordering.

Note also that these are not mutually exclusive. A Fabric network can have multiple ordering services supporting different applications or application requirements.

1.8 Performance and Scalability

Performance of a blockchain platform can be affected by many variables such as transaction size, block size, network size, as well as limits of the hardware, etc. The Hyperledger Fabric Performance and Scale working group currently works on a benchmarking framework called Hyperledger Caliper.

Several research papers have been published studying and testing the performance capabilities of Hyperledger Fabric. The latest scaled Fabric to 20,000 transactions per second.

1.9 Conclusion

Any serious evaluation of blockchain platforms should include Hyperledger Fabric in its short list.

Combined, the differentiating capabilities of Fabric make it a highly scalable system for permissioned blockchains supporting flexible trust assumptions that enable the platform to support a wide range of industry use cases ranging from government, to finance, to supply-chain logistics, to healthcare and so much more.

Hyperledger Fabric is the most active of the Hyperledger projects. The community building around the platform is growing steadily, and the innovation delivered with each successive release far out-paces any of the other enterprise blockchain platforms.

1.10 Acknowledgement

The preceding is derived from the peer reviewed "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains" - Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, Jason Yellick

CHAPTER 2

What's new in Hyperledger Fabric v2.x

The first Hyperledger Fabric major release since v1.0, Fabric v2.0 delivers important new features and changes for users and operators alike, including support for new application and privacy patterns, enhanced governance around smart contracts, and new options for operating nodes.

Each v2.x minor release builds on the v2.0 release with minor features, improvements, and bug fixes.

Let's take a look at some of the highlights of the Fabric v2.0 release…

2.1 Decentralized governance for smart contracts

Fabric v2.0 introduces decentralized governance for smart contracts, with a new process for installing a chaincode on your peers and starting it on a channel. The new Fabric chaincode lifecycle allows multiple organizations to come to agreement on the parameters of a chaincode, such as the chaincode endorsement policy, before it can be used to interact with the ledger. The new model offers several improvements over the previous lifecycle:

- **Multiple organizations must agree to the parameters of a chaincode** In the release 1.x versions of Fabric, one organization had the ability to set parameters of a chaincode (for instance the endorsement policy) for all other channel members, who only had the power to refuse to install the chaincode and therefore not take part in transactions invoking it. The new Fabric chaincode lifecycle is more flexible since it supports both centralized trust models (such as that of the previous lifecycle model) as well as decentralized models requiring a sufficient number of organizations to agree on an endorsement policy and other details before the chaincode becomes active on a channel.
- **More deliberate chaincode upgrade process** In the previous chaincode lifecycle, the upgrade transaction could be issued by a single organization, creating a risk for a channel member that had not yet installed the new chaincode. The new model allows for a chaincode to be upgraded only after a sufficient number of organizations have approved the upgrade.
- **Simpler endorsement policy and private data collection updates** Fabric lifecycle allows you to change an endorsement policy or private data collection configuration without having to repackage or reinstall the chaincode. Users can also take advantage of a new default endorsement policy that requires endorsement from a majority of organizations on the channel. This policy is updated automatically when organizations are added or removed from the channel.

- **Inspectable chaincode packages** The Fabric lifecycle packages chaincode in easily readable tar files. This makes it easier to inspect the chaincode package and coordinate installation across multiple organizations.
- **Start multiple chaincodes on a channel using one package** The previous lifecycle defined each chaincode on the channel using a name and version that was specified when the chaincode package was installed. You can now use a single chaincode package and deploy it multiple times with different names on the same channel or on different channels. For example, if you'd like to track different types of assets in their own 'copy' of the chaincode.
- **Chaincode packages do not need to be identical across channel members** Organizations can extend a chaincode for their own use case, for example to perform different validations in the interest of their organization. As long as the required number of organizations endorse chaincode transactions with matching results, the transaction will be validated and committed to the ledger. This also allows organizations to individually roll out minor fixes on their own schedules without requiring the entire network to proceed in lock-step.

2.1.1 Using the new chaincode lifecycle

For existing Fabric deployments, you can continue to use the prior chaincode lifecycle with Fabric v2.0. The new chaincode lifecycle will become effective only when the channel application capability is updated to v2.0. See the *Fabric chaincode lifecycle* concept topic for an overview of the new chaincode lifecycle.

2.2 New chaincode application patterns for collaboration and consensus

The same decentralized methods of coming to agreement that underpin the new chaincode lifecycle management can also be used in your own chaincode applications to ensure organizations consent to data transactions before they are committed to the ledger.

- **Automated checks** As mentioned above, organizations can add automated checks to chaincode functions to validate additional information before endorsing a transaction proposal.
- **Decentralized agreement** Human decisions can be modeled into a chaincode process that spans multiple transactions. The chaincode may require actors from various organizations to indicate their terms and conditions of agreement in a ledger transaction. Then, a final chaincode proposal can verify that the conditions from all the individual transactors are met, and "settle" the business transaction with finality across all channel members. For a concrete example of indicating terms and conditions in private, see the asset transfer scenario in the *Private data* documentation.

2.3 Private data enhancements

Fabric v2.0 also enables new patterns for working with and sharing private data, without the requirement of creating private data collections for all combinations of channel members that may want to transact. Specifically, instead of sharing private data within a collection of multiple members, you may want to share private data across collections, where each collection may include a single organization, or perhaps a single organization along with a regulator or auditor.

Several enhancements in Fabric v2.0 make these new private data patterns possible:

- **Sharing and verifying private data** When private data is shared with a channel member who is not a member of a collection, or shared with another private data collection that contains one or more channel members (by writing a key to that collection), the receiving parties can utilize the GetPrivateDataHash() chaincode API to verify that the private data matches the on-chain hashes that were created from private data in previous transactions.

- **Collection-level endorsement policies** Private data collections can now optionally be defined with an endorsement policy that overrides the chaincode-level endorsement policy for keys within the collection. This feature can be used to restrict which organizations can write data to a collection, and is what enables the new chaincode lifecycle and chaincode application patterns mentioned earlier. For example, you may have a chaincode endorsement policy that requires a majority of organizations to endorse, but for any given transaction, you may need two transacting organizations to individually endorse their agreement in their own private data collections.
- **Implicit per-organization collections** If you'd like to utilize per-organization private data patterns, you don't even need to define the collections when deploying chaincode in Fabric v2.0. Implicit organization-specific collections can be used without any upfront definition.

To learn more about the new private data patterns, see the *Private data* (conceptual documentation). For details about private data collection configuration and implicit collections, see the *Private Data* (reference documentation).

2.4 External chaincode launcher

The external chaincode launcher feature empowers operators to build and launch chaincode with the technology of their choice. Use of external builders and launchers is not required as the default behavior builds and runs chaincode in the same manner as prior releases using the Docker API.

- **Eliminate Docker daemon dependency** Prior releases of Fabric required peers to have access to a Docker daemon in order to build and launch chaincode - something that may not be desirable in production environments due to the privileges required by the peer process.
- **Alternatives to containers** Chaincode is no longer required to be run in Docker containers, and may be executed in the operator's choice of environment (including containers).
- **External builder executables** An operator can provide a set of external builder executables to override how the peer builds and launches chaincode.
- **Chaincode as an external service** Traditionally, chaincodes are launched by the peer, and then connect back to the peer. It is now possible to run chaincode as an external service, for example in a Kubernetes pod, which a peer can connect to and utilize for chaincode execution. See *Chaincode as an external service* for more information.

See *External Builders and Launchers* to learn more about the external chaincode launcher feature.

2.5 State database cache for improved performance on CouchDB

- When using external CouchDB state database, read delays during endorsement and validation phases have historically been a performance bottleneck.
- With Fabric v2.0, a new peer cache replaces many of these expensive lookups with fast local cache reads. The cache size can be configured by using the core.yaml property `cacheSize`.

2.6 Alpine-based docker images

Starting with v2.0, Hyperledger Fabric Docker images will use Alpine Linux, a security-oriented, lightweight Linux distribution. This means that Docker images are now much smaller, providing faster download and startup times, as well as taking up less disk space on host systems. Alpine Linux is designed from the ground up with security in mind, and the minimalist nature of the Alpine distribution greatly reduces the risk of security vulnerabilities.

2.7 Sample test network

The fabric-samples repository now includes a new Fabric test network. The test network is built to be a modular and user friendly sample Fabric network that makes it easy to test your applications and smart contracts. The network also supports the ability to deploy your network using Certificate Authorities, in addition to cryptogen.

For more information about this network, check out *Using the Fabric test network*.

2.8 Upgrading to Fabric v2.x

A major new release brings some additional upgrade considerations. Rest assured though, that rolling upgrades from v1.4.x to v2.x are supported, so that network components can be upgraded one at a time with no downtime.

The upgrade docs have been significantly expanded and reworked, and now have a standalone home in the documentation: *Upgrading to the latest release*. Here you'll find documentation on *Upgrading your components* and *Updating the capability level of a channel*, as well as a specific look at the considerations for upgrading to v2.0, *Considerations for getting to v2.x*.

CHAPTER 3

Release notes

The release notes provide more details for users moving to the new release. Specifically, take a look at the changes and deprecations that are being announced with the new Fabric v2.0 release, and the changes introduced in v2.1.

- Fabric v2.0.0 release notes.
- Fabric v2.0.1 release notes.
- Fabric v2.1.0 release notes.

CHAPTER 4

Key Concepts

4.1 Introduction

Hyperledger Fabric is a platform for distributed ledger solutions underpinned by a modular architecture delivering high degrees of confidentiality, resiliency, flexibility, and scalability. It is designed to support pluggable implementations of different components and accommodate the complexity and intricacies that exist across the economic ecosystem.

We recommend first-time users begin by going through the rest of the introduction below in order to gain familiarity with how blockchains work and with the specific features and components of Hyperledger Fabric.

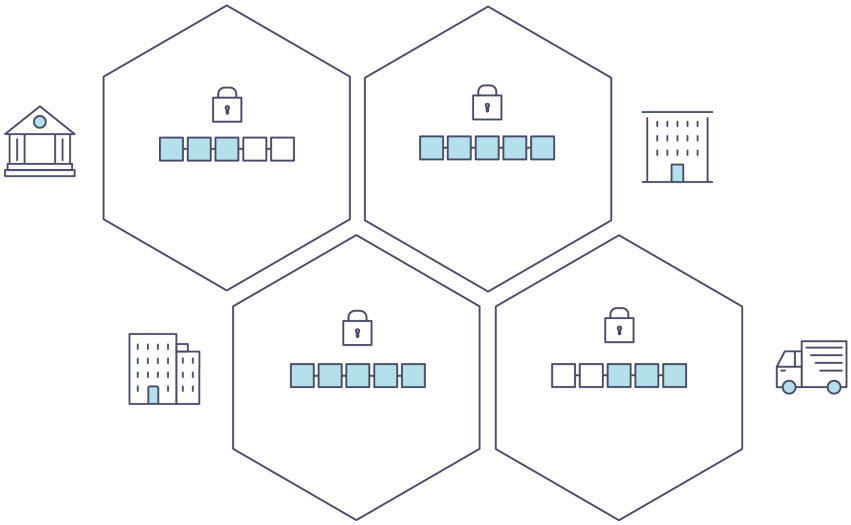
Once comfortable — or if you're already familiar with blockchain and Hyperledger Fabric — go to *Getting Started* and from there explore the demos, technical specifications, APIs, etc.

4.1.1 What is a Blockchain?

A Distributed Ledger

At the heart of a blockchain network is a distributed ledger that records all the transactions that take place on the network.

A blockchain ledger is often described as **decentralized** because it is replicated across many network participants, each of whom **collaborate** in its maintenance. We'll see that decentralization and collaboration are powerful attributes that mirror the way businesses exchange goods and services in the real world.



In addition to being decentralized and collaborative, the information recorded to a blockchain is append-only, using cryptographic techniques that guarantee that once a transaction has been added to the ledger it cannot be modified. This property of "immutability" makes it simple to determine the provenance of information because participants can be sure information has not been changed after the fact. It's why blockchains are sometimes described as **systems of proof**.

Smart Contracts

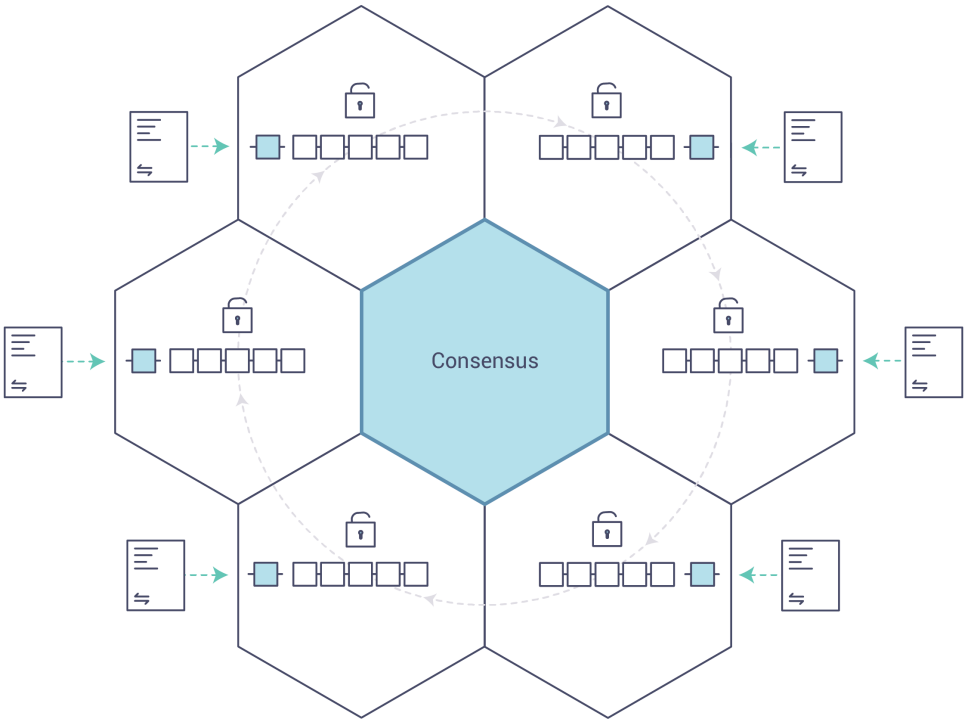
To support the consistent update of information — and to enable a whole host of ledger functions (transacting, querying, etc) — a blockchain network uses **smart contracts** to provide controlled access to the ledger.

Smart contracts are not only a key mechanism for encapsulating information and keeping it simple across the network, they can also be written to allow participants to execute certain aspects of transactions automatically.

A smart contract can, for example, be written to stipulate the cost of shipping an item where the shipping charge changes depending on how quickly the item arrives. With the terms agreed to by both parties and written to the ledger, the appropriate funds change hands automatically when the item is received.

Consensus

The process of keeping the ledger transactions synchronized across the network — to ensure that ledgers update only when transactions are approved by the appropriate participants, and that when ledgers do update, they update with the same transactions in the same order — is called **consensus**.



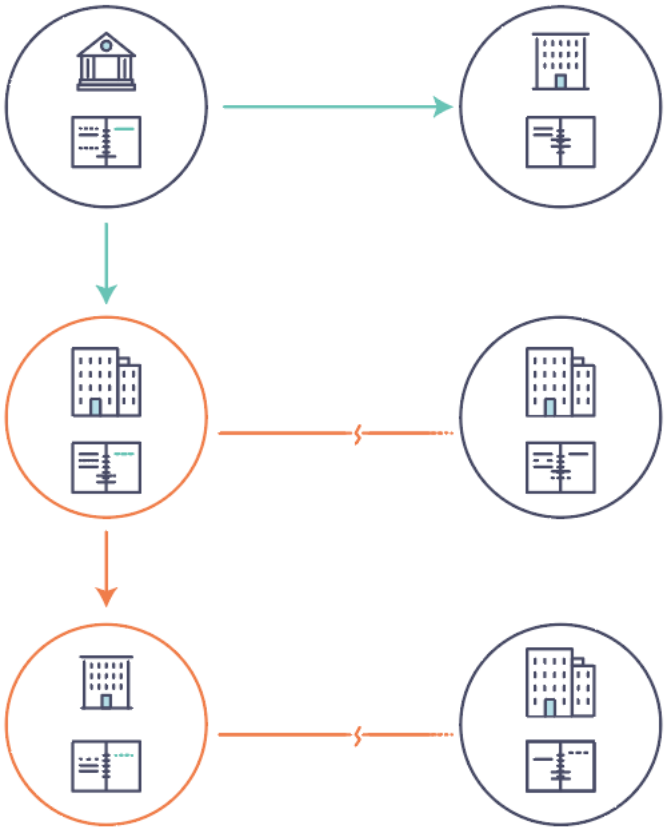
You'll learn a lot more about ledgers, smart contracts and consensus later. For now, it's enough to think of a blockchain as a shared, replicated transaction system which is updated via smart contracts and kept consistently synchronized through a collaborative process called consensus.

4.1.2 Why is a Blockchain useful?

Today's Systems of Record

The transactional networks of today are little more than slightly updated versions of networks that have existed since business records have been kept. The members of a **business network** transact with each other, but they maintain separate records of their transactions. And the things they're transacting — whether it's Flemish tapestries in the 16th century or the securities of today — must have their provenance established each time they're sold to ensure that the business selling an item possesses a chain of title verifying their ownership of it.

What you're left with is a business network that looks like this:



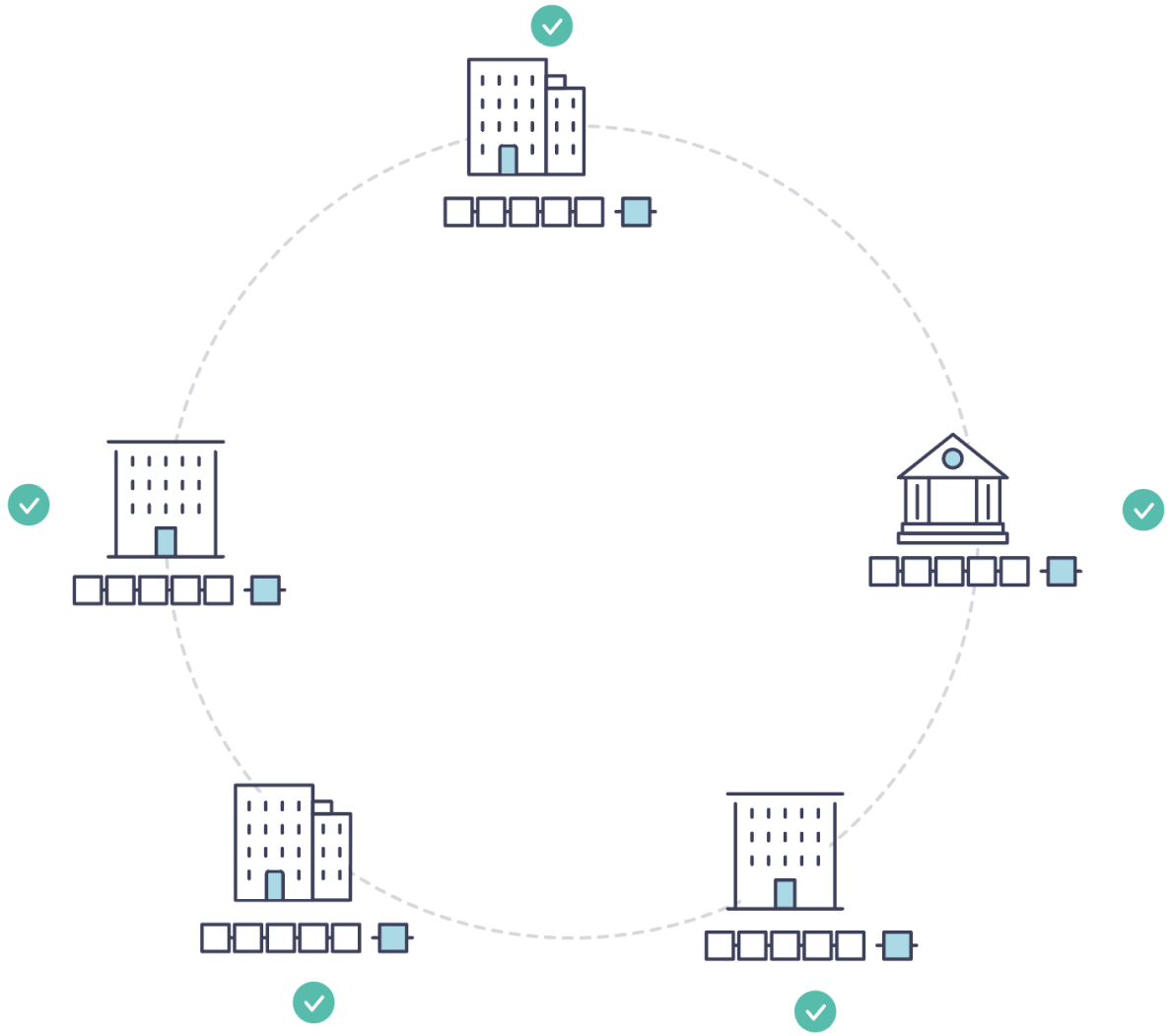
Modern technology has taken this process from stone tablets and paper folders to hard drives and cloud platforms, but the underlying structure is the same. Unified systems for managing the identity of network participants do not exist, establishing provenance is so laborious it takes days to clear securities transactions (the world volume of which is numbered in the many trillions of dollars), contracts must be signed and executed manually, and every database in the system contains unique information and therefore represents a single point of failure.

It's impossible with today's fractured approach to information and process sharing to build a system of record that spans a business network, even though the needs of visibility and trust are clear.

The Blockchain Difference

What if, instead of the rat's nest of inefficiencies represented by the "modern" system of transactions, business networks had standard methods for establishing identity on the network, executing transactions, and storing data? What if establishing the provenance of an asset could be determined by looking through a list of transactions that, once written, cannot be changed, and can therefore be trusted?

That business network would look more like this:



This is a blockchain network, wherein every participant has their own replicated copy of the ledger. In addition to ledger information being shared, the processes which update the ledger are also shared. Unlike today's systems, where a participant's **private** programs are used to update their **private** ledgers, a blockchain system has **shared** programs to update **shared** ledgers.

With the ability to coordinate their business network through a shared ledger, blockchain networks can reduce the time, cost, and risk associated with private information and processing while improving trust and visibility.

You now know what blockchain is and why it's useful. There are a lot of other details that are important, but they all relate to these fundamental ideas of the sharing of information and processes.

4.1.3 What is Hyperledger Fabric?

The Linux Foundation founded the Hyperledger project in 2015 to advance cross-industry blockchain technologies. Rather than declaring a single blockchain standard, it encourages a collaborative approach to developing blockchain technologies via a community process, with intellectual property rights that encourage open development and the adoption of key standards over time.

Hyperledger Fabric is one of the blockchain projects within Hyperledger. Like other blockchain technologies, it has a ledger, uses smart contracts, and is a system by which participants manage their transactions.

Where Hyperledger Fabric breaks from some other blockchain systems is that it is **private** and **permissioned**. Rather than an open permissionless system that allows unknown identities to participate in the network (requiring protocols like "proof of work" to validate transactions and secure the network), the members of a Hyperledger Fabric network enroll through a trusted **Membership Service Provider (MSP)**.

Hyperledger Fabric also offers several pluggable options. Ledger data can be stored in multiple formats, consensus mechanisms can be swapped in and out, and different MSPs are supported.

Hyperledger Fabric also offers the ability to create **channels**, allowing a group of participants to create a separate ledger of transactions. This is an especially important option for networks where some participants might be competitors and not want every transaction they make — a special price they're offering to some participants and not others, for example — known to every participant. If two participants form a channel, then those participants — and no others — have copies of the ledger for that channel.

Shared Ledger

Hyperledger Fabric has a ledger subsystem comprising two components: the **world state** and the **transaction log**. Each participant has a copy of the ledger to every Hyperledger Fabric network they belong to.

The world state component describes the state of the ledger at a given point in time. It's the database of the ledger. The transaction log component records all transactions which have resulted in the current value of the world state; it's the update history for the world state. The ledger, then, is a combination of the world state database and the transaction log history.

The ledger has a replaceable data store for the world state. By default, this is a LevelDB key-value store database. The transaction log does not need to be pluggable. It simply records the before and after values of the ledger database being used by the blockchain network.

Smart Contracts

Hyperledger Fabric smart contracts are written in **chaincode** and are invoked by an application external to the blockchain when that application needs to interact with the ledger. In most cases, chaincode interacts only with the database component of the ledger, the world state (querying it, for example), and not the transaction log.

Chaincode can be implemented in several programming languages. Currently, Go and Node are supported.

Privacy

Depending on the needs of a network, participants in a Business-to-Business (B2B) network might be extremely sensitive about how much information they share. For other networks, privacy will not be a top concern.

Hyperledger Fabric supports networks where privacy (using channels) is a key operational requirement as well as networks that are comparatively open.

Consensus

Transactions must be written to the ledger in the order in which they occur, even though they might be between different sets of participants within the network. For this to happen, the order of transactions must be established and a method for rejecting bad transactions that have been inserted into the ledger in error (or maliciously) must be put into place.

This is a thoroughly researched area of computer science, and there are many ways to achieve it, each with different trade-offs. For example, PBFT (Practical Byzantine Fault Tolerance) can provide a mechanism for file replicas to communicate with each other to keep each copy consistent, even in the event of corruption. Alternatively, in Bitcoin, ordering happens through a process called mining where competing computers race to solve a cryptographic puzzle which defines the order that all processes subsequently build upon.

Hyperledger Fabric has been designed to allow network starters to choose a consensus mechanism that best represents the relationships that exist between participants. As with privacy, there is a spectrum of needs; from networks that are highly structured in their relationships to those that are more peer-to-peer.

4.2 Hyperledger Fabric Model

This section outlines the key design features woven into Hyperledger Fabric that fulfill its promise of a comprehensive, yet customizable, enterprise blockchain solution:

- *Assets* — Asset definitions enable the exchange of almost anything with monetary value over the network, from whole foods to antique cars to currency futures.
- *Chaincode* — Chaincode execution is partitioned from transaction ordering, limiting the required levels of trust and verification across node types, and optimizing network scalability and performance.
- *Ledger Features* — The immutable, shared ledger encodes the entire transaction history for each channel, and includes SQL-like query capability for efficient auditing and dispute resolution.
- *Privacy* — Channels and private data collections enable private and confidential multi-lateral transactions that are usually required by competing businesses and regulated industries that exchange assets on a common network.
- *Security & Membership Services* — Permissioned membership provides a trusted blockchain network, where participants know that all transactions can be detected and traced by authorized regulators and auditors.
- *Consensus* — A unique approach to consensus enables the flexibility and scalability needed for the enterprise.

4.2.1 Assets

Assets can range from the tangible (real estate and hardware) to the intangible (contracts and intellectual property). Hyperledger Fabric provides the ability to modify assets using chaincode transactions.

Assets are represented in Hyperledger Fabric as a collection of key-value pairs, with state changes recorded as transactions on a *Channel* ledger. Assets can be represented in binary and/or JSON form.

4.2.2 Chaincode

Chaincode is software defining an asset or assets, and the transaction instructions for modifying the asset(s); in other words, it's the business logic. Chaincode enforces the rules for reading or altering key-value pairs or other state database information. Chaincode functions execute against the ledger's current state database and are initiated through a transaction proposal. Chaincode execution results in a set of key-value writes (write set) that can be submitted to the network and applied to the ledger on all peers.

4.2.3 Ledger Features

The ledger is the sequenced, tamper-resistant record of all state transitions in the fabric. State transitions are a result of chaincode invocations ('transactions') submitted by participating parties. Each transaction results in a set of asset key-value pairs that are committed to the ledger as creates, updates, or deletes.

The ledger is comprised of a blockchain ('chain') to store the immutable, sequenced record in blocks, as well as a state database to maintain current fabric state. There is one ledger per channel. Each peer maintains a copy of the ledger for each channel of which they are a member.

Some features of a Fabric ledger:

- Query and update ledger using key-based lookups, range queries, and composite key queries
- Read-only queries using a rich query language (if using CouchDB as state database)
- Read-only history queries — Query ledger history for a key, enabling data provenance scenarios

- Transactions consist of the versions of keys/values that were read in chaincode (read set) and keys/values that were written in chaincode (write set)
- Transactions contain signatures of every endorsing peer and are submitted to ordering service
- Transactions are ordered into blocks and are "delivered" from an ordering service to peers on a channel
- Peers validate transactions against endorsement policies and enforce the policies
- Prior to appending a block, a versioning check is performed to ensure that states for assets that were read have not changed since chaincode execution time
- There is immutability once a transaction is validated and committed
- A channel's ledger contains a configuration block defining policies, access control lists, and other pertinent information
- Channels contain *Membership Service Provider* instances allowing for crypto materials to be derived from different certificate authorities

See the ledger topic for a deeper dive on the databases, storage structure, and "query-ability."

4.2.4 Privacy

Hyperledger Fabric employs an immutable ledger on a per-channel basis, as well as chaincode that can manipulate and modify the current state of assets (i.e. update key-value pairs). A ledger exists in the scope of a channel — it can be shared across the entire network (assuming every participant is operating on one common channel) — or it can be privatized to include only a specific set of participants.

In the latter scenario, these participants would create a separate channel and thereby isolate/segregate their transactions and ledger. In order to solve scenarios that want to bridge the gap between total transparency and privacy, chaincode can be installed only on peers that need to access the asset states to perform reads and writes (in other words, if a chaincode is not installed on a peer, it will not be able to properly interface with the ledger).

When a subset of organizations on that channel need to keep their transaction data confidential, a private data collection (collection) is used to segregate this data in a private database, logically separate from the channel ledger, accessible only to the authorized subset of organizations.

Thus, channels keep transactions private from the broader network whereas collections keep data private between subsets of organizations on the channel.

To further obfuscate the data, values within chaincode can be encrypted (in part or in total) using common cryptographic algorithms such as AES before sending transactions to the ordering service and appending blocks to the ledger. Once encrypted data has been written to the ledger, it can be decrypted only by a user in possession of the corresponding key that was used to generate the cipher text.

See the *Private Data* topic for more details on how to achieve privacy on your blockchain network.

4.2.5 Security & Membership Services

Hyperledger Fabric underpins a transactional network where all participants have known identities. Public Key Infrastructure is used to generate cryptographic certificates which are tied to organizations, network components, and end users or client applications. As a result, data access control can be manipulated and governed on the broader network and on channel levels. This "permissioned" notion of Hyperledger Fabric, coupled with the existence and capabilities of channels, helps address scenarios where privacy and confidentiality are paramount concerns.

See the *Membership Service Providers (MSP)* topic to better understand cryptographic implementations, and the sign, verify, authenticate approach used in Hyperledger Fabric.

4.2.6 Consensus

In distributed ledger technology, consensus has recently become synonymous with a specific algorithm, within a single function. However, consensus encompasses more than simply agreeing upon the order of transactions, and this differentiation is highlighted in Hyperledger Fabric through its fundamental role in the entire transaction flow, from proposal and endorsement, to ordering, validation and commitment. In a nutshell, consensus is defined as the full-circle verification of the correctness of a set of transactions comprising a block.

Consensus is achieved ultimately when the order and results of a block's transactions have met the explicit policy criteria checks. These checks and balances take place during the lifecycle of a transaction, and include the usage of endorsement policies to dictate which specific members must endorse a certain transaction class, as well as system chaincodes to ensure that these policies are enforced and upheld. Prior to commitment, the peers will employ these system chaincodes to make sure that enough endorsements are present, and that they were derived from the appropriate entities. Moreover, a versioning check will take place during which the current state of the ledger is agreed or consented upon, before any blocks containing transactions are appended to the ledger. This final check provides protection against double spend operations and other threats that might compromise data integrity, and allows for functions to be executed against non-static variables.

In addition to the multitude of endorsement, validity and versioning checks that take place, there are also ongoing identity verifications happening in all directions of the transaction flow. Access control lists are implemented on hierarchical layers of the network (ordering service down to channels), and payloads are repeatedly signed, verified and authenticated as a transaction proposal passes through the different architectural components. To conclude, consensus is not merely limited to the agreed upon order of a batch of transactions; rather, it is an overarching characterization that is achieved as a byproduct of the ongoing verifications that take place during a transaction's journey from proposal to commitment.

Check out the *Transaction Flow* diagram for a visual representation of consensus.

4.3 Blockchain network

This topic will describe, **at a conceptual level**, how Hyperledger Fabric allows organizations to collaborate in the formation of blockchain networks. If you're an architect, administrator or developer, you can use this topic to get a solid understanding of the major structure and process components in a Hyperledger Fabric blockchain network. This topic will use a manageable worked example that introduces all of the major components in a blockchain network.

After reading this topic and understanding the concept of policies, you will have a solid understanding of the decisions that organizations need to make to establish the policies that control a deployed Hyperledger Fabric network. You'll also understand how organizations manage network evolution using declarative policies – a key feature of Hyperledger Fabric. In a nutshell, you'll understand the major technical components of Hyperledger Fabric and the decisions organizations need to make about them.

4.3.1 What is a blockchain network?

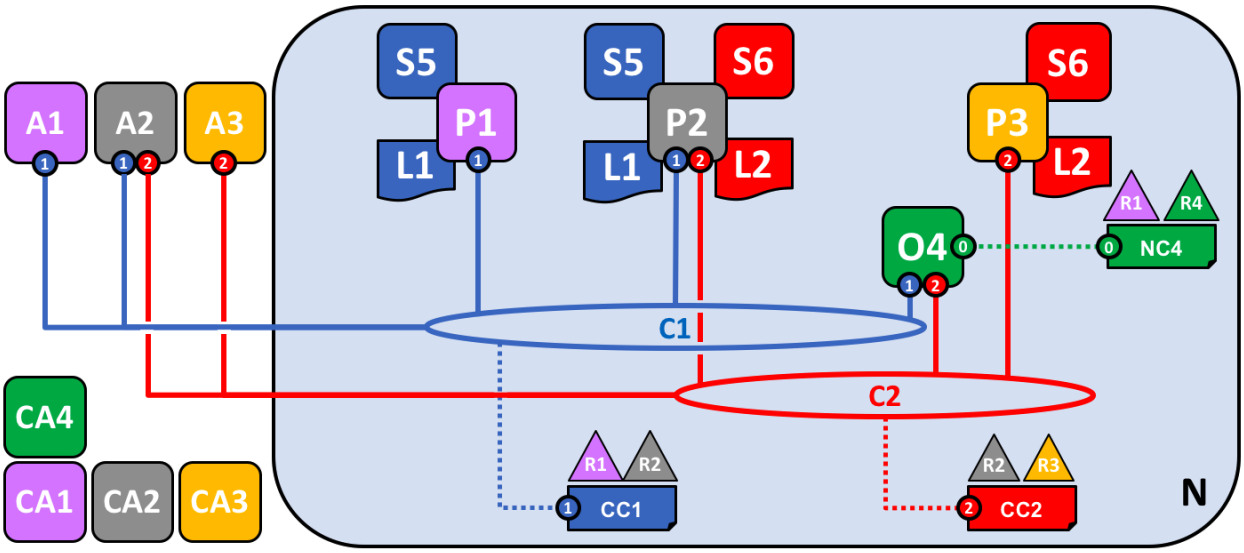
A blockchain network is a technical infrastructure that provides ledger and smart contract (chaincode) services to applications. Primarily, smart contracts are used to generate transactions which are subsequently distributed to every peer node in the network where they are immutably recorded on their copy of the ledger. The users of applications might be end users using client applications or blockchain network administrators.

In most cases, multiple organizations come together as a consortium to form the network and their permissions are determined by a set of policies that are agreed by the consortium when the network is originally configured. Moreover, network policies can change over time subject to the agreement of the organizations in the consortium, as we'll discover when we discuss the concept of *modification policy*.

4.3.2 The sample network

Before we start, let's show you what we're aiming at! Here's a diagram representing the **final state** of our sample network.

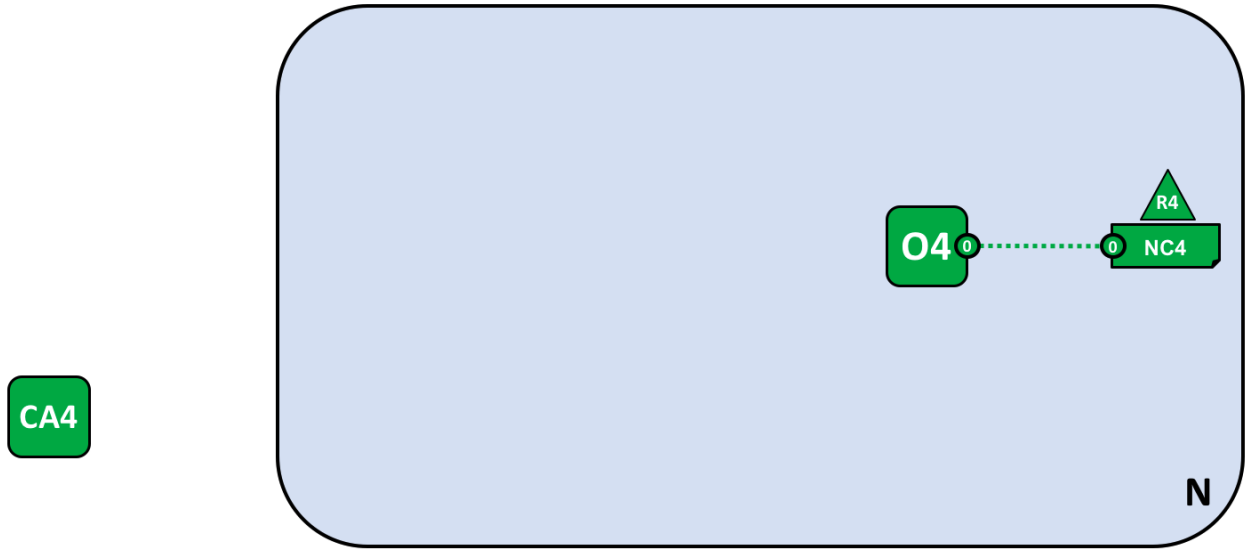
Don't worry that this might look complicated! As we go through this topic, we will build up the network piece by piece, so that you see how the organizations R1, R2, R3 and R4 contribute infrastructure to the network to help form it. This infrastructure implements the blockchain network, and it is governed by policies agreed by the organizations who form the network – for example, who can add new organizations. You'll discover how applications consume the ledger and smart contract services provided by the blockchain network.



Four organizations, R1, R2, R3 and R4 have jointly decided, and written into an agreement, that they will set up and exploit a Hyperledger Fabric network. R4 has been assigned to be the network initiator – it has been given the power to set up the initial version of the network. R4 has no intention to perform business transactions on the network. R1 and R2 have a need for a private communications within the overall network, as do R2 and R3. Organization R1 has a client application that can perform business transactions within channel C1. Organization R2 has a client application that can do similar work both in channel C1 and C2. Organization R3 has a client application that can do this on channel C2. Peer node P1 maintains a copy of the ledger L1 associated with C1. Peer node P2 maintains a copy of the ledger L1 associated with C1 and a copy of ledger L2 associated with C2. Peer node P3 maintains a copy of the ledger L2 associated with C2. The network is governed according to policy rules specified in network configuration NC4, the network is under the control of organizations R1 and R4. Channel C1 is governed according to the policy rules specified in channel configuration CC1; the channel is under the control of organizations R1 and R2. Channel C2 is governed according to the policy rules specified in channel configuration CC2; the channel is under the control of organizations R2 and R3. There is an ordering service O4 that services as a network administration point for N, and uses the system channel. The ordering service also supports application channels C1 and C2, for the purposes of transaction ordering into blocks for distribution. Each of the four organizations has a preferred Certificate Authority.

4.3.3 Creating the Network

Let's start at the beginning by creating the basis for the network:



The network is formed when an orderer is started. In our example network, N, the ordering service comprising a single node, O4, is configured according to a network configuration NC4, which gives administrative rights to organization R4. At the network level, Certificate Authority CA4 is used to dispense identities to the administrators and network nodes of the R4 organization.

We can see that the first thing that defines a **network, N,** is an **ordering service, O4**. It's helpful to think of the ordering service as the initial administration point for the network. As agreed beforehand, O4 is initially configured and started by an administrator in organization R4, and hosted in R4. The configuration NC4 contains the policies that describe the starting set of administrative capabilities for the network. Initially this is set to only give R4 rights over the network. This will change, as we'll see later, but for now R4 is the only member of the network.

Certificate Authorities

You can also see a Certificate Authority, CA4, which is used to issue certificates to administrators and network nodes. CA4 plays a key role in our network because it dispenses X.509 certificates that can be used to identify components as belonging to organization R4. Certificates issued by CAs can also be used to sign transactions to indicate that an organization endorses the transaction result – a precondition of it being accepted onto the ledger. Let's examine these two aspects of a CA in a little more detail.

Firstly, different components of the blockchain network use certificates to identify themselves to each other as being from a particular organization. That's why there is usually more than one CA supporting a blockchain network – different organizations often use different CAs. We're going to use four CAs in our network; one for each organization. Indeed, CAs are so important that Hyperledger Fabric provides you with a built-in one (called *Fabric-CA*) to help you get going, though in practice, organizations will choose to use their own CA.

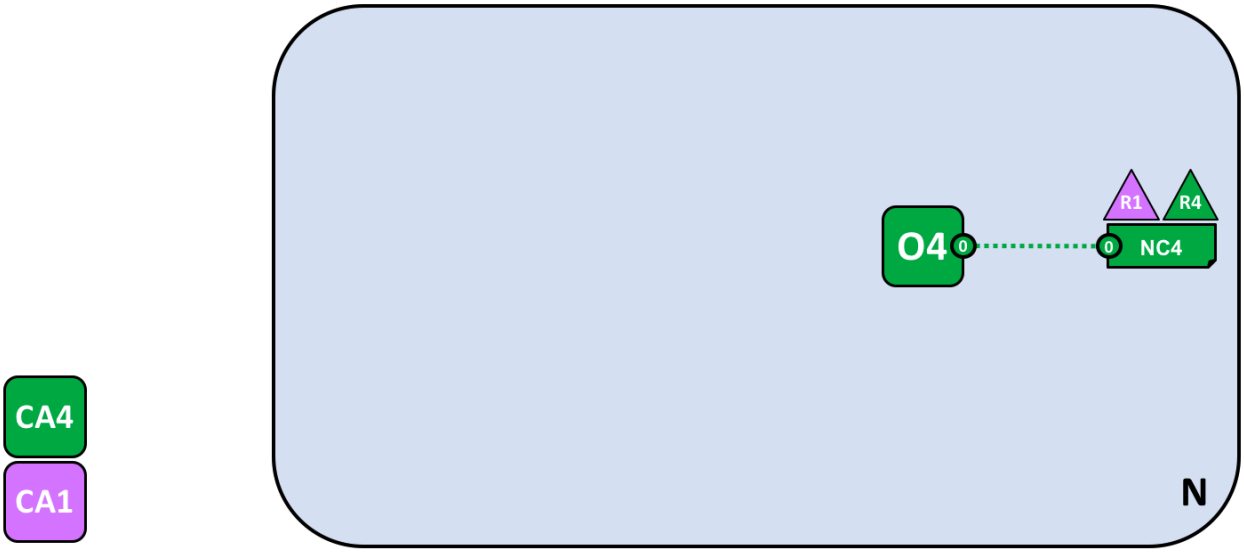
The mapping of certificates to member organizations is achieved by via a structure called a Membership Services Provider (MSP). Network configuration NC4 uses a named MSP to identify the properties of certificates dispensed by CA4 which associate certificate holders with organization R4. NC4 can then use this MSP name in policies to grant actors from R4 particular rights over network resources. An example of such a policy is to identify the administrators in R4 who can add new member organizations to the network. We don't show MSPs on these diagrams, as they would just clutter them up, but they are very important.

Secondly, we'll see later how certificates issued by CAs are at the heart of the transaction generation and validation process. Specifically, X.509 certificates are used in client application transaction proposals and smart contract transaction responses to digitally sign transactions. Subsequently the network nodes who host copies of the ledger verify that transaction signatures are valid before accepting transactions onto the ledger.

Let's recap the basic structure of our example blockchain network. There's a resource, the network N, accessed by a set of users defined by a Certificate Authority CA4, who have a set of rights over the resources in the network N as described by policies contained inside a network configuration NC4. All of this is made real when we configure and start the ordering service node O4.

4.3.4 Adding Network Administrators

NC4 was initially configured to only allow R4 users administrative rights over the network. In this next phase, we are going to allow organization R1 users to administer the network. Let's see how the network evolves:



Organization R4 updates the network configuration to make organization R1 an administrator too. After this point R1 and R4 have equal rights over the network configuration.

We see the addition of a new organization R1 as an administrator – R1 and R4 now have equal rights over the network. We can also see that certificate authority CA1 has been added – it can be used to identify users from the R1 organization. After this point, users from both R1 and R4 can administer the network.

Although the orderer node, O4, is running on R4's infrastructure, R1 has shared administrative rights over it, as long as it can gain network access. It means that R1 or R4 could update the network configuration NC4 to allow the R2 organization a subset of network operations. In this way, even though R4 is running the ordering service, and R1 has full administrative rights over it, R2 has limited rights to create new consortia.

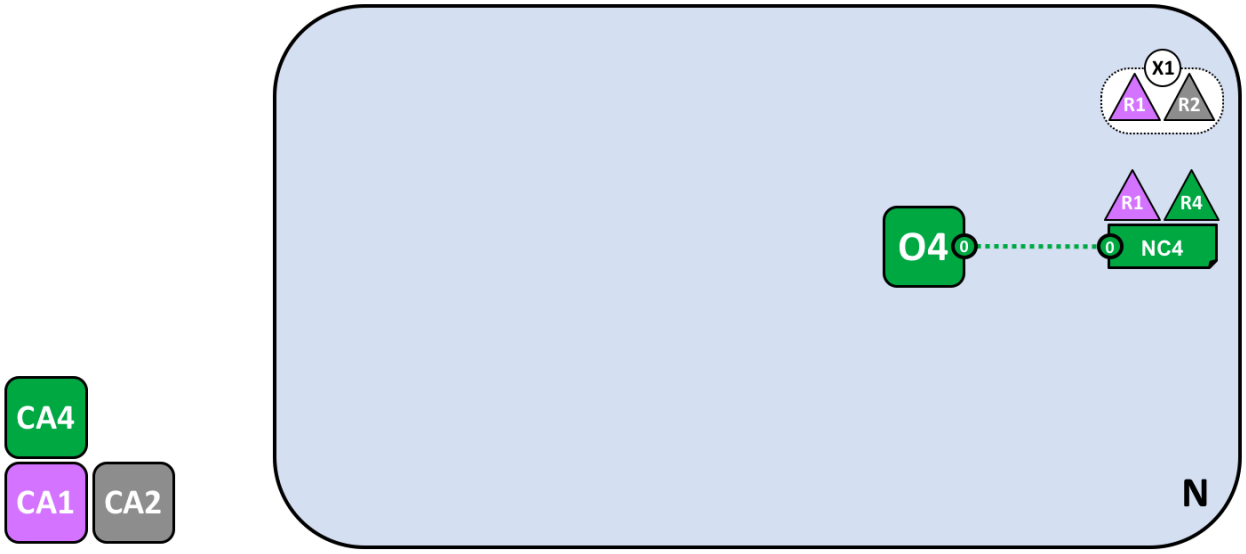
In its simplest form, the ordering service is a single node in the network, and that's what you can see in the example. Ordering services are usually multi-node, and can be configured to have different nodes in different organizations. For example, we might run O4 in R4 and connect it to O2, a separate orderer node in organization R1. In this way, we would have a multi-site, multi-organization administration structure.

We'll discuss the ordering service a little *later in this topic*, but for now just think of the ordering service as an administration point which provides different organizations controlled access to the network.

4.3.5 Defining a Consortium

Although the network can now be administered by R1 and R4, there is very little that can be done. The first thing we need to do is define a consortium. This word literally means "a group with a shared destiny", so it's an appropriate choice for a set of organizations in a blockchain network.

Let's see how a consortium is defined:



A network administrator defines a consortium X1 that contains two members, the organizations R1 and R2. This consortium definition is stored in the network configuration NC4, and will be used at the next stage of network development. CA1 and CA2 are the respective Certificate Authorities for these organizations.

Because of the way NC4 is configured, only R1 or R4 can create new consortia. This diagram shows the addition of a new consortium, X1, which defines R1 and R2 as its constituting organizations. We can also see that CA2 has been added to identify users from R2. Note that a consortium can have any number of organizational members – we have just shown two as it is the simplest configuration.

Why are consortia important? We can see that a consortium defines the set of organizations in the network who share a need to **transact** with one another – in this case R1 and R2. It really makes sense to group organizations together if they have a common goal, and that's exactly what's happening.

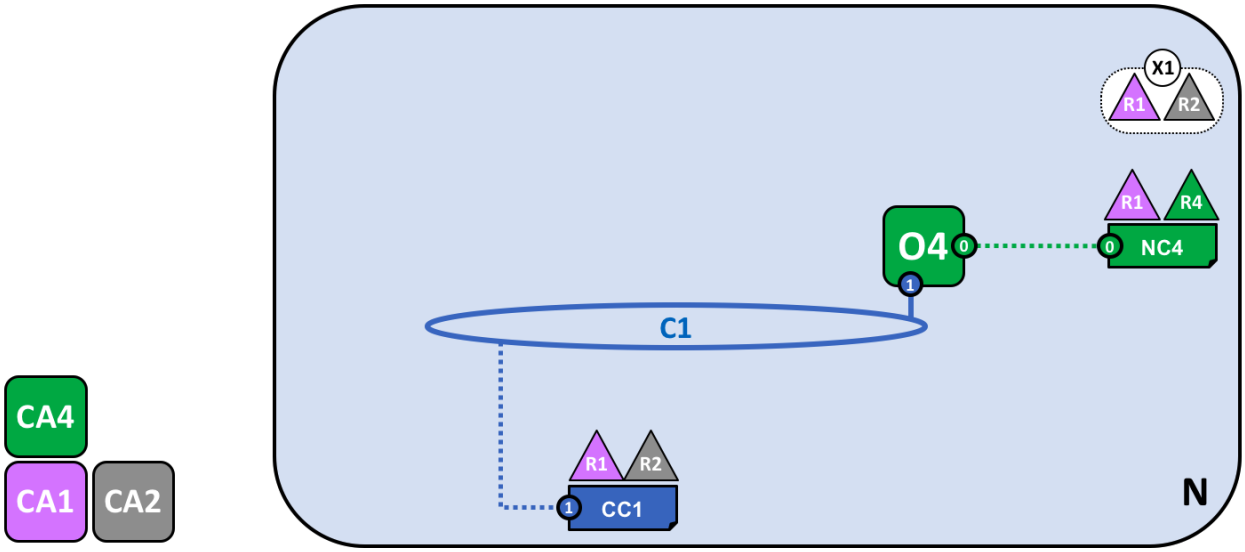
The network, although started by a single organization, is now controlled by a larger set of organizations. We could have started it this way, with R1, R2 and R4 having shared control, but this build up makes it easier to understand.

We're now going to use consortium X1 to create a really important part of a Hyperledger Fabric blockchain – **a channel**.

4.3.6 Creating a channel for a consortium

So let's create this key part of the Fabric blockchain network – **a channel**. A channel is a primary communications mechanism by which the members of a consortium can communicate with each other. There can be multiple channels in a network, but for now, we'll start with one.

Let's see how the first channel has been added to the network:



A channel C1 has been created for R1 and R2 using the consortium definition X1. The channel is governed by a channel configuration CC1, completely separate to the network configuration. CC1 is managed by R1 and R2 who have equal rights over C1. R4 has no rights in CC1 whatsoever.

The channel C1 provides a private communications mechanism for the consortium X1. We can see channel C1 has been connected to the ordering service O4 but that nothing else is attached to it. In the next stage of network development, we're going to connect components such as client applications and peer nodes. But at this point, a channel represents the **potential** for future connectivity.

Even though channel C1 is a part of the network N, it is quite distinguishable from it. Also notice that organizations R3 and R4 are not in this channel – it is for transaction processing between R1 and R2. In the previous step, we saw how R4 could grant R1 permission to create new consortia. It's helpful to mention that R4 **also** allowed R1 to create channels! In this diagram, it could have been organization R1 or R4 who created a channel C1. Again, note that a channel can have any number of organizations connected to it – we've shown two as it's the simplest configuration.

Again, notice how channel C1 has a completely separate configuration, CC1, to the network configuration NC4. CC1 contains the policies that govern the rights that R1 and R2 have over the channel C1 – and as we've seen, R3 and R4 have no permissions in this channel. R3 and R4 can only interact with C1 if they are added by R1 or R2 to the appropriate policy in the channel configuration CC1. An example is defining who can add a new organization to the channel. Specifically, note that R4 cannot add itself to the channel C1 – it must, and can only, be authorized by R1 or R2.

Why are channels so important? Channels are useful because they provide a mechanism for private communications and private data between the members of a consortium. Channels provide privacy from other channels, and from the network. Hyperledger Fabric is powerful in this regard, as it allows organizations to share infrastructure and keep it private at the same time. There's no contradiction here – different consortia within the network will have a need for different information and processes to be appropriately shared, and channels provide an efficient mechanism to do this. Channels provide an efficient sharing of infrastructure while maintaining data and communications privacy.

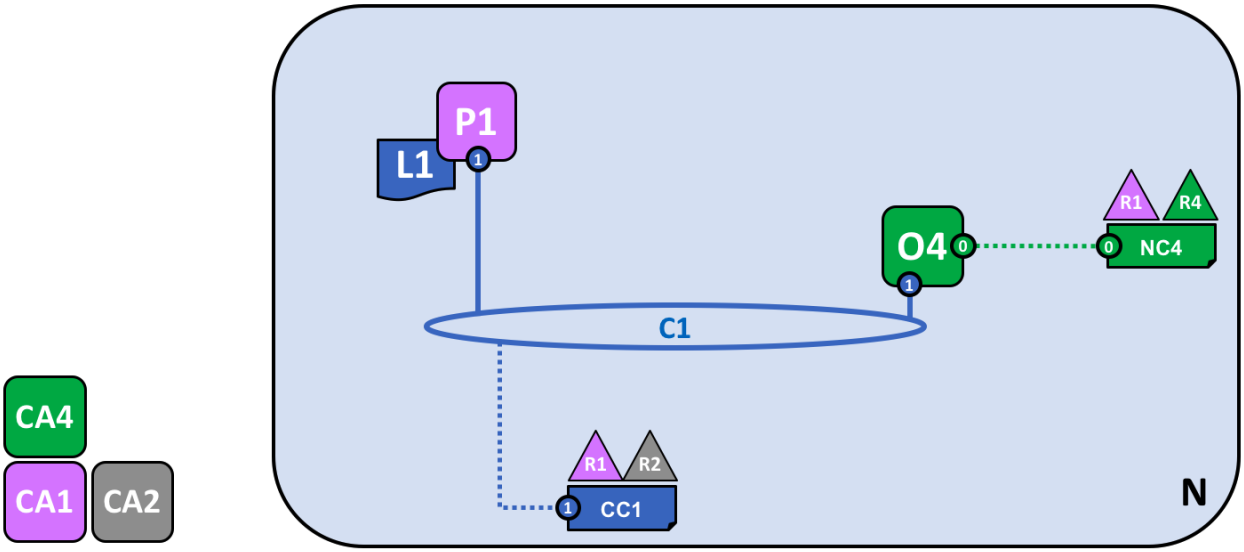
We can also see that once a channel has been created, it is in a very real sense "free from the network". It is only organizations that are explicitly specified in a channel configuration that have any control over it, from this time forward into the future. Likewise, any updates to network configuration NC4 from this time onwards will have no direct effect on channel configuration CC1; for example if consortia definition X1 is changed, it will not affect the members of channel C1. Channels are therefore useful because they allow private communications between the organizations constituting the channel. Moreover, the data in a channel is completely isolated from the rest of the network, including other channels.

As an aside, there is also a special **system channel** defined for use by the ordering service. It behaves in exactly the

same way as a regular channel, which are sometimes called **application channels** for this reason. We don't normally need to worry about this channel, but we'll discuss a little bit more about it *later in this topic*.

4.3.7 Peers and Ledgers

Let's now start to use the channel to connect the blockchain network and the organizational components together. In the next stage of network development, we can see that our network N has just acquired two new components, namely a peer node P1 and a ledger instance, L1.



A peer node P1 has joined the channel C1. P1 physically hosts a copy of the ledger L1. P1 and O4 can communicate with each other using channel C1.

Peer nodes are the network components where copies of the blockchain ledger are hosted! At last, we're starting to see some recognizable blockchain components! P1's purpose in the network is purely to host a copy of the ledger L1 for others to access. We can think of L1 as being **physically hosted** on P1, but **logically hosted** on the channel C1. We'll see this idea more clearly when we add more peers to the channel.

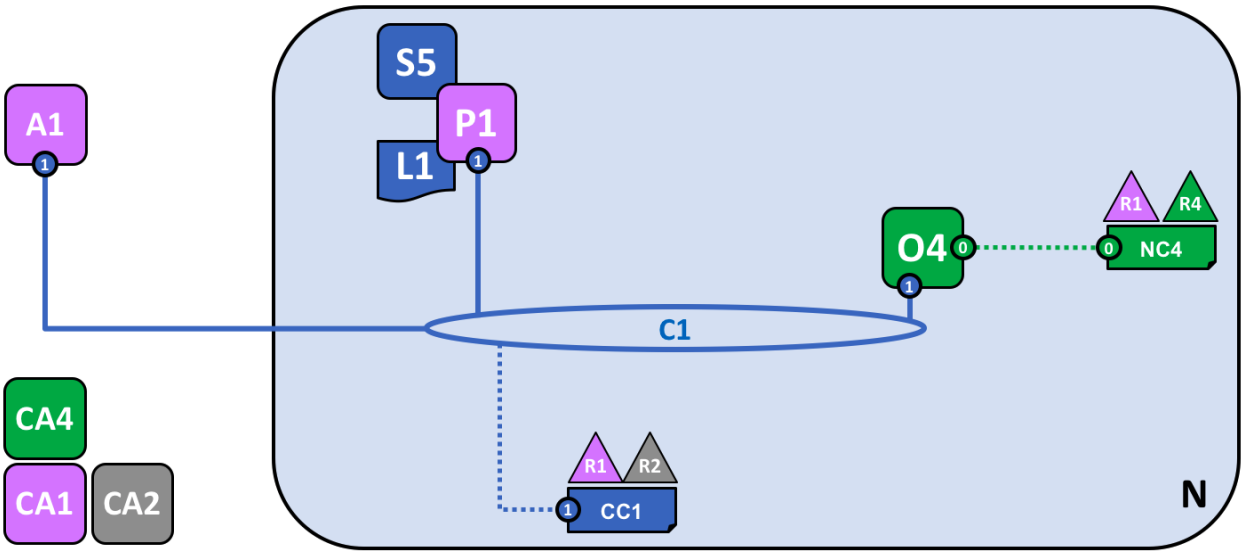
A key part of a P1's configuration is an X.509 identity issued by CA1 which associates P1 with organization R1. Once P1 is started, it can **join** channel C1 using the orderer O4. When O4 receives this join request, it uses the channel configuration CC1 to determine P1's permissions on this channel. For example, CC1 determines whether P1 can read and/or write information to the ledger L1.

Notice how peers are joined to channels by the organizations that own them, and though we've only added one peer, we'll see how there can be multiple peer nodes on multiple channels within the network. We'll see the different roles that peers can take on a little later.

4.3.8 Applications and Smart Contract chaincode

Now that the channel C1 has a ledger on it, we can start connecting client applications to consume some of the services provided by workhorse of the ledger, the peer!

Notice how the network has grown:



A smart contract S5 has been installed onto P1. Client application A1 in organization R1 can use S5 to access the ledger via peer node P1. A1, P1 and O4 are all joined to channel C1, i.e. they can all make use of the communication facilities provided by that channel.

In the next stage of network development, we can see that client application A1 can use channel C1 to connect to specific network resources – in this case A1 can connect to both peer node P1 and orderer node O4. Again, see how channels are central to the communication between network and organization components. Just like peers and orderers, a client application will have an identity that associates it with an organization. In our example, client application A1 is associated with organization R1; and although it is outside the Fabric blockchain network, it is connected to it via the channel C1.

It might now appear that A1 can access the ledger L1 directly via P1, but in fact, all access is managed via a special program called a smart contract chaincode, S5. Think of S5 as defining all the common access patterns to the ledger; S5 provides a well-defined set of ways by which the ledger L1 can be queried or updated. In short, client application A1 has to go through smart contract S5 to get to ledger L1!

Smart contracts can be created by application developers in each organization to implement a business process shared by the consortium members. Smart contracts are used to help generate transactions which can be subsequently distributed to every node in the network. We'll discuss this idea a little later; it'll be easier to understand when the network is bigger. For now, the important thing to understand is that to get to this point two operations must have been performed on the smart contract; it must have been **installed** on peers, and then **defined** on a channel.

Hyperledger Fabric users often use the terms **smart contract** and **chaincode** interchangeably. In general, a smart contract defines the **transaction logic** that controls the lifecycle of a business object contained in the world state. It is then packaged into a chaincode which is then deployed to a blockchain network. Think of smart contracts as governing transactions, whereas chaincode governs how smart contracts are packaged for deployment.

Installing a chaincode package

After a smart contract S5 has been developed, an administrator in organization R1 must create a chaincode package and install it onto peer node P1. This is a straightforward operation; once completed, P1 has full knowledge of S5. Specifically, P1 can see the **implementation** logic of S5 – the program code that it uses to access the ledger L1. We contrast this to the S5 **interface** which merely describes the inputs and outputs of S5, without regard to its implementation.

When an organization has multiple peers in a channel, it can choose the peers upon which it installs smart contracts; it does not need to install a smart contract on every peer.

Defining a chaincode

Although a chaincode is installed on the peers of individual organizations, it is governed and operated in the scope of a channel. Each organization needs to approve a **chaincode definition**, a set of parameters that establish how a chaincode will be used on a channel. An organization must approve a chaincode definition in order to use the installed smart contract to query the ledger and endorse transactions. In our example, which only has a single peer node P1, an administrator in organization R1 must approve a chaincode definition for S5.

A sufficient number of organizations need to approve a chaincode definition (A majority, by default) before the chaincode definition can be committed to the channel and used to interact with the channel ledger. Because the channel only has one member, the administrator of R1 can commit the chaincode definition of S5 to the channel C1. Once the definition has been committed, S5 can now be invoked by client application A1!

Note that although every component on the channel can now access S5, they are not able to see its program logic. This remains private to those nodes who have installed it; in our example that means P1. Conceptually this means that it's the smart contract **interface** that is defined and committed to a channel, in contrast to the smart contract **implementation** that is installed. To reinforce this idea; installing a smart contract shows how we think of it being **physically hosted** on a peer, whereas a smart contract that has been defined on a channel shows how we consider it **logically hosted** by the channel.

Endorsement policy

The most important piece of information supplied within the chaincode definition is the endorsement policy. It describes which organizations must approve transactions before they will be accepted by other organizations onto their copy of the ledger. In our sample network, transactions can only be accepted onto ledger L1 if R1 or R2 endorse them.

Committing the chaincode definition to the channel places the endorsement policy on the channel ledger; it enables it to be accessed by any member of the channel. You can read more about endorsement policies in the transaction flow topic.

Invoking a smart contract

Once a smart contract has been installed on a peer node and defined on a channel it can be invoked by a client application. Client applications do this by sending transaction proposals to peers owned by the organizations specified by the smart contract endorsement policy. The transaction proposal serves as input to the smart contract, which uses it to generate an endorsed transaction response, which is returned by the peer node to the client application.

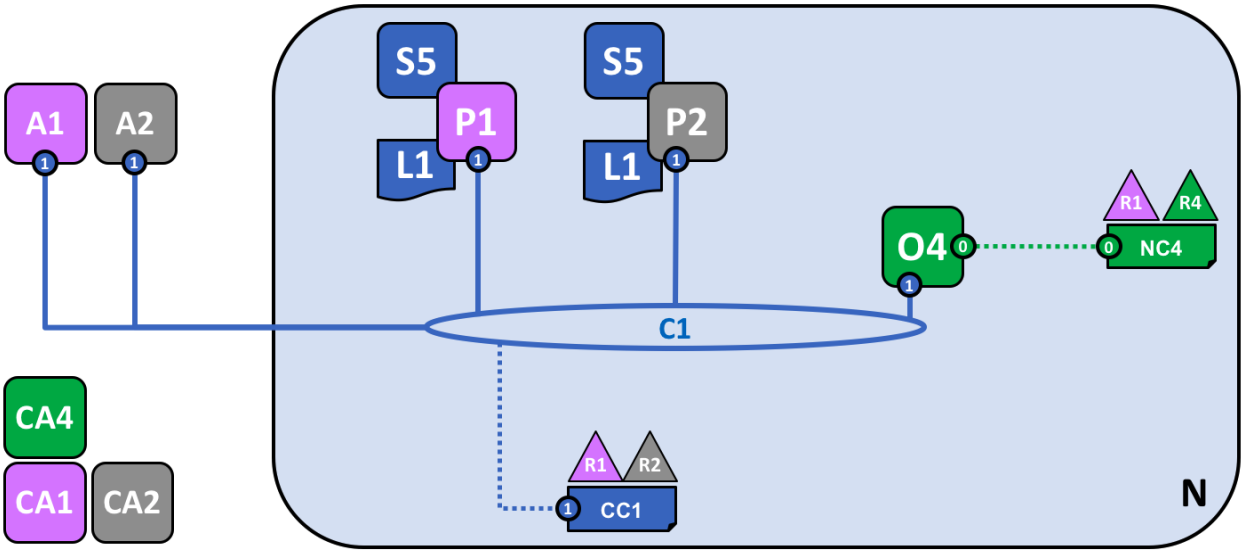
It's these transactions responses that are packaged together with the transaction proposal to form a fully endorsed transaction, which can be distributed to the entire network. We'll look at this in more detail later For now, it's enough to understand how applications invoke smart contracts to generate endorsed transactions.

By this stage in network development we can see that organization R1 is fully participating in the network. Its applications – starting with A1 – can access the ledger L1 via smart contract S5, to generate transactions that will be endorsed by R1, and therefore accepted onto the ledger because they conform to the endorsement policy.

4.3.9 Network completed

Recall that our objective was to create a channel for consortium X1 – organizations R1 and R2. This next phase of network development sees organization R2 add its infrastructure to the network.

Let's see how the network has evolved:



The network has grown through the addition of infrastructure from organization R2. Specifically, R2 has added peer node P2, which hosts a copy of ledger L1, and chaincode S5. R2 approves the same chaincode definition as R1. P2 has also joined channel C1, as has application A2. A2 and P2 are identified using certificates from CA2. All of this means that both applications A1 and A2 can invoke S5 on C1 either using peer node P1 or P2.

We can see that organization R2 has added a peer node, P2, on channel C1. P2 also hosts a copy of the ledger L1 and smart contract S5. We can see that R2 has also added client application A2 which can connect to the network via channel C1. To achieve this, an administrator in organization R2 has created peer node P2 and joined it to channel C1, in the same way as an administrator in R1. The administrator also has to approve the same chaincode definition as R1.

We have created our first operational network! At this stage in network development, we have a channel in which organizations R1 and R2 can fully transact with each other. Specifically, this means that applications A1 and A2 can generate transactions using smart contract S5 and ledger L1 on channel C1.

Generating and accepting transactions

In contrast to peer nodes, which always host a copy of the ledger, we see that there are two different kinds of peer nodes; those which host smart contracts and those which do not. In our network, every peer hosts a copy of the smart contract, but in larger networks, there will be many more peer nodes that do not host a copy of the smart contract. A peer can only *run* a smart contract if it is installed on it, but it can *know* about the interface of a smart contract by being connected to a channel.

You should not think of peer nodes which do not have smart contracts installed as being somehow inferior. It's more the case that peer nodes with smart contracts have a special power – to help **generate** transactions. Note that all peer nodes can **validate** and subsequently **accept** or **reject** transactions onto their copy of the ledger L1. However, only peer nodes with a smart contract installed can take part in the process of transaction **endorsement** which is central to the generation of valid transactions.

We don't need to worry about the exact details of how transactions are generated, distributed and accepted in this topic – it is sufficient to understand that we have a blockchain network where organizations R1 and R2 can share information and processes as ledger-captured transactions. We'll learn a lot more about transactions, ledgers, smart contracts in other topics.

Types of peers

In Hyperledger Fabric, while all peers are the same, they can assume multiple roles depending on how the network is configured. We now have enough understanding of a typical network topology to describe these roles.

- *Committing peer*. Every peer node in a channel is a committing peer. It receives blocks of generated transactions, which are subsequently validated before they are committed to the peer node's copy of the ledger as an append operation.
- *Endorsing peer*. Every peer with a smart contract *can* be an endorsing peer if it has a smart contract installed. However, to actually *be* an endorsing peer, the smart contract on the peer must be used by a client application to generate a digitally signed transaction response. The term *endorsing peer* is an explicit reference to this fact.

  An endorsement policy for a smart contract identifies the organizations whose peer should digitally sign a generated transaction before it can be accepted onto a committing peer's copy of the ledger.

These are the two major types of peer; there are two other roles a peer can adopt:

- *Leader peer*. When an organization has multiple peers in a channel, a leader peer is a node which takes responsibility for distributing transactions from the orderer to the other committing peers in the organization. A peer can choose to participate in static or dynamic leadership selection.

  It is helpful, therefore to think of two sets of peers from leadership perspective – those that have static leader selection, and those with dynamic leader selection. For the static set, zero or more peers can be configured as leaders. For the dynamic set, one peer will be elected leader by the set. Moreover, in the dynamic set, if a leader peer fails, then the remaining peers will re-elect a leader.

  It means that an organization's peers can have one or more leaders connected to the ordering service. This can help to improve resilience and scalability in large networks which process high volumes of transactions.
- *Anchor peer*. If a peer needs to communicate with a peer in another organization, then it can use one of the **anchor peers** defined in the channel configuration for that organization. An organization can have zero or more anchor peers defined for it, and an anchor peer can help with many different cross-organization communication scenarios.

Note that a peer can be a committing peer, endorsing peer, leader peer and anchor peer all at the same time! Only the anchor peer is optional – for all practical purposes there will always be a leader peer and at least one endorsing peer and at least one committing peer.

Adding organizations and peers to the channel

When R2 joins the channel, the organization must install smart contract S5 onto its peer node, P2. That's obvious – if applications A1 or A2 wish to use S5 on peer node P2 to generate transactions, it must first be present; installation is the mechanism by which this happens. At this point, peer node P2 has a physical copy of the smart contract and the ledger; like P1, it can both generate and accept transactions onto its copy of ledger L1.

R2 must approve the same chaincode definition as was approved by R1 in order to use smart contract S5. Because the chaincode definition has already been committed to the channel by organization R1, R2 can use the chaincode as soon as the organization approves the chaincode definition and installs the chaincode package. The commit transaction only needs to happen once. A new organization can use the chaincode as soon as they approve the chaincode parameters agreed to by other members of the channel. Because the approval of a chaincode definition occurs at the organization level, R2 can approve the chaincode definition once and join multiple peers to the channel with the chaincode package installed. However, if R2 wanted to change the chaincode definition, both R1 and R2 would need to approve a new definition for their organization, and then one of the organizations would need to commit the definition to the channel.

In our network, we can see that channel C1 connects two client applications, two peer nodes and an ordering service. Since there is only one channel, there is only one **logical** ledger with which these components interact. Peer nodes P1 and P2 have identical copies of ledger L1. Copies of smart contract S5 will usually be identically implemented using the same programming language, but if not, they must be semantically equivalent.

We can see that the careful addition of peers to the network can help support increased throughput, stability, and resilience. For example, more peers in a network will allow more applications to connect to it; and multiple peers in an organization will provide extra resilience in the case of planned or unplanned outages.

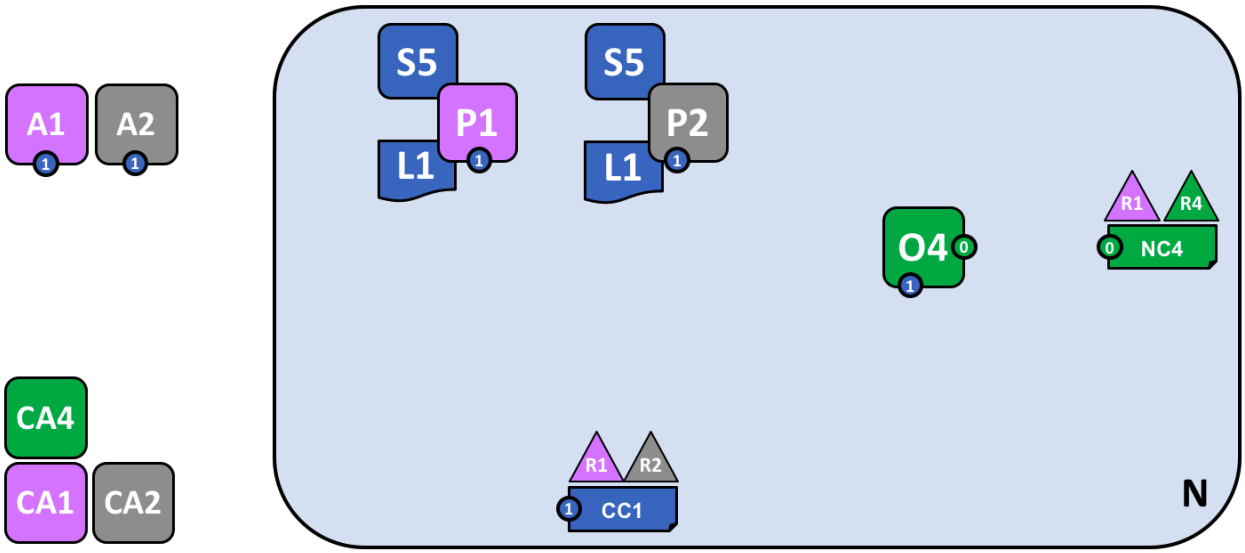
It all means that it is possible to configure sophisticated topologies which support a variety of operational goals – there is no theoretical limit to how big a network can get. Moreover, the technical mechanism by which peers within an individual organization efficiently discover and communicate with each other – the gossip protocol – will accommodate a large number of peer nodes in support of such topologies.

The careful use of network and channel policies allow even large networks to be well-governed. Organizations are free to add peer nodes to the network so long as they conform to the policies agreed by the network. Network and channel policies create the balance between autonomy and control which characterizes a de-centralized network.

4.3.10 Simplifying the visual vocabulary

We're now going to simplify the visual vocabulary used to represent our sample blockchain network. As the size of the network grows, the lines initially used to help us understand channels will become cumbersome. Imagine how complicated our diagram would be if we added another peer or client application, or another channel?

That's what we're going to do in a minute, so before we do, let's simplify the visual vocabulary. Here's a simplified representation of the network we've developed so far:



The diagram shows the facts relating to channel C1 in the network N as follows: Client applications A1 and A2 can use channel C1 for communication with peers P1 and P2, and orderer O4. Peer nodes P1 and P2 can use the communication services of channel C1. Ordering service O4 can make use of the communication services of channel C1. Channel configuration CC1 applies to channel C1.

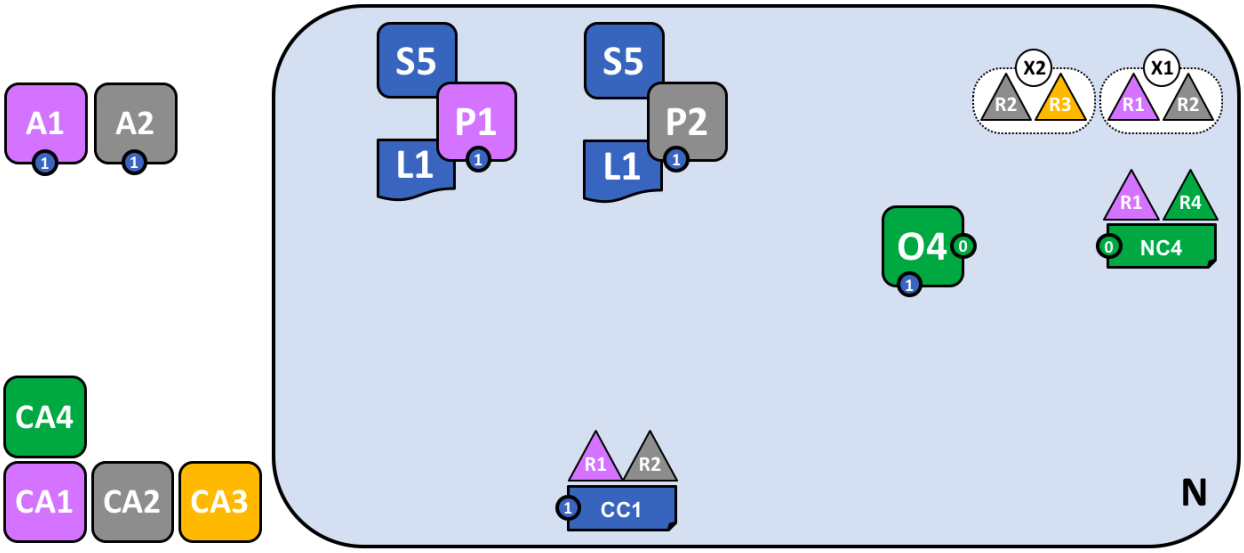
Note that the network diagram has been simplified by replacing channel lines with connection points, shown as blue circles which include the channel number. No information has been lost. This representation is more scalable because it eliminates crossing lines. This allows us to more clearly represent larger networks. We've achieved this simplification by focusing on the connection points between components and a channel, rather than the channel itself.

4.3.11 Adding another consortium definition

In this next phase of network development, we introduce organization R3. We're going to give organizations R2 and R3 a separate application channel which allows them to transact with each other. This application channel will be

completely separate to that previously defined, so that R2 and R3 transactions can be kept private to them.

Let's return to the network level and define a new consortium, X2, for R2 and R3:



A network administrator from organization R1 or R4 has added a new consortium definition, X2, which includes organizations R2 and R3. This will be used to define a new channel for X2.

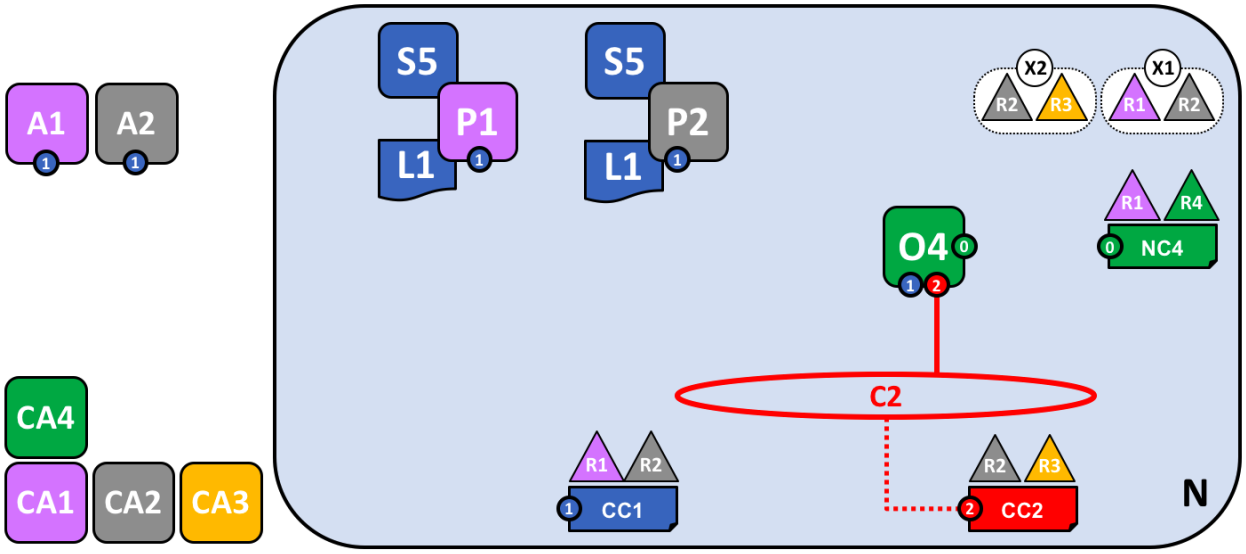
Notice that the network now has two consortia defined: X1 for organizations R1 and R2 and X2 for organizations R2 and R3. Consortium X2 has been introduced in order to be able to create a new channel for R2 and R3.

A new channel can only be created by those organizations specifically identified in the network configuration policy, NC4, as having the appropriate rights to do so, i.e. R1 or R4. This is an example of a policy which separates organizations that can manage resources at the network level versus those who can manage resources at the channel level. Seeing these policies at work helps us understand why Hyperledger Fabric has a sophisticated **tiered** policy structure.

In practice, consortium definition X2 has been added to the network configuration NC4. We discuss the exact mechanics of this operation elsewhere in the documentation.

4.3.12 Adding a new channel

Let's now use this new consortium definition, X2, to create a new channel, C2. To help reinforce your understanding of the simpler channel notation, we've used both visual styles – channel C1 is represented with blue circular end points, whereas channel C2 is represented with red connecting lines:



A new channel C2 has been created for R2 and R3 using consortium definition X2. The channel has a channel configuration CC2, completely separate to the network configuration NC4, and the channel configuration CC1. Channel C2 is managed by R2 and R3 who have equal rights over C2 as defined by a policy in CC2. R1 and R4 have no rights defined in CC2 whatsoever.

The channel C2 provides a private communications mechanism for the consortium X2. Again, notice how organizations united in a consortium are what form channels. The channel configuration CC2 now contains the policies that govern channel resources, assigning management rights to organizations R2 and R3 over channel C2. It is managed exclusively by R2 and R3; R1 and R4 have no power in channel C2. For example, channel configuration CC2 can subsequently be updated to add organizations to support network growth, but this can only be done by R2 or R3.

Note how the channel configurations CC1 and CC2 remain completely separate from each other, and completely separate from the network configuration, NC4. Again we're seeing the de-centralized nature of a Hyperledger Fabric network; once channel C2 has been created, it is managed by organizations R2 and R3 independently to other network elements. Channel policies always remain separate from each other and can only be changed by the organizations authorized to do so in the channel.

As the network and channels evolve, so will the network and channel configurations. There is a process by which this is accomplished in a controlled manner – involving configuration transactions which capture the change to these configurations. Every configuration change results in a new configuration block transaction being generated, and *later in this topic*, we'll see how these blocks are validated and accepted to create updated network and channel configurations respectively.

Network and channel configurations

Throughout our sample network, we see the importance of network and channel configurations. These configurations are important because they encapsulate the **policies** agreed by the network members, which provide a shared reference for controlling access to network resources. Network and channel configurations also contain **facts** about the network and channel composition, such as the name of consortia and its organizations.

For example, when the network is first formed using the ordering service node O4, its behaviour is governed by the network configuration NC4. The initial configuration of NC4 only contains policies that permit organization R4 to manage network resources. NC4 is subsequently updated to also allow R1 to manage network resources. Once this change is made, any administrator from organization R1 or R4 that connects to O4 will have network management rights because that is what the policy in the network configuration NC4 permits. Internally, each node in the ordering service records each channel in the network configuration, so that there is a record of each channel created, at the network level.

It means that although ordering service node O4 is the actor that created consortia X1 and X2 and channels C1 and C2, the **intelligence** of the network is contained in the network configuration NC4 that O4 is obeying. As long as O4 behaves as a good actor, and correctly implements the policies defined in NC4 whenever it is dealing with network resources, our network will behave as all organizations have agreed. In many ways NC4 can be considered more important than O4 because, ultimately, it controls network access.

The same principles apply for channel configurations with respect to peers. In our network, P1 and P2 are likewise good actors. When peer nodes P1 and P2 are interacting with client applications A1 or A2 they are each using the policies defined within channel configuration CC1 to control access to the channel C1 resources.

For example, if A1 wants to access the smart contract chaincode S5 on peer nodes P1 or P2, each peer node uses its copy of CC1 to determine the operations that A1 can perform. For example, A1 may be permitted to read or write data from the ledger L1 according to policies defined in CC1. We'll see later the same pattern for actors in channel and its channel configuration CC2. Again, we can see that while the peers and applications are critical actors in the network, their behaviour in a channel is dictated more by the channel configuration policy than any other factor.

Finally, it is helpful to understand how network and channel configurations are physically realized. We can see that network and channel configurations are logically singular – there is one for the network, and one for each channel. This is important; every component that accesses the network or the channel must have a shared understanding of the permissions granted to different organizations.

Even though there is logically a single configuration, it is actually replicated and kept consistent by every node that forms the network or channel. For example, in our network peer nodes P1 and P2 both have a copy of channel configuration CC1, and by the time the network is fully complete, peer nodes P2 and P3 will both have a copy of channel configuration CC2. Similarly ordering service node O4 has a copy of the network configuration, but in a *multi-node configuration*, every ordering service node will have its own copy of the network configuration.

Both network and channel configurations are kept consistent using the same blockchain technology that is used for user transactions – but for **configuration** transactions. To change a network or channel configuration, an administrator must submit a configuration transaction to change the network or channel configuration. It must be signed by the organizations identified in the appropriate policy as being responsible for configuration change. This policy is called the **mod_policy** and we'll *discuss it later*.

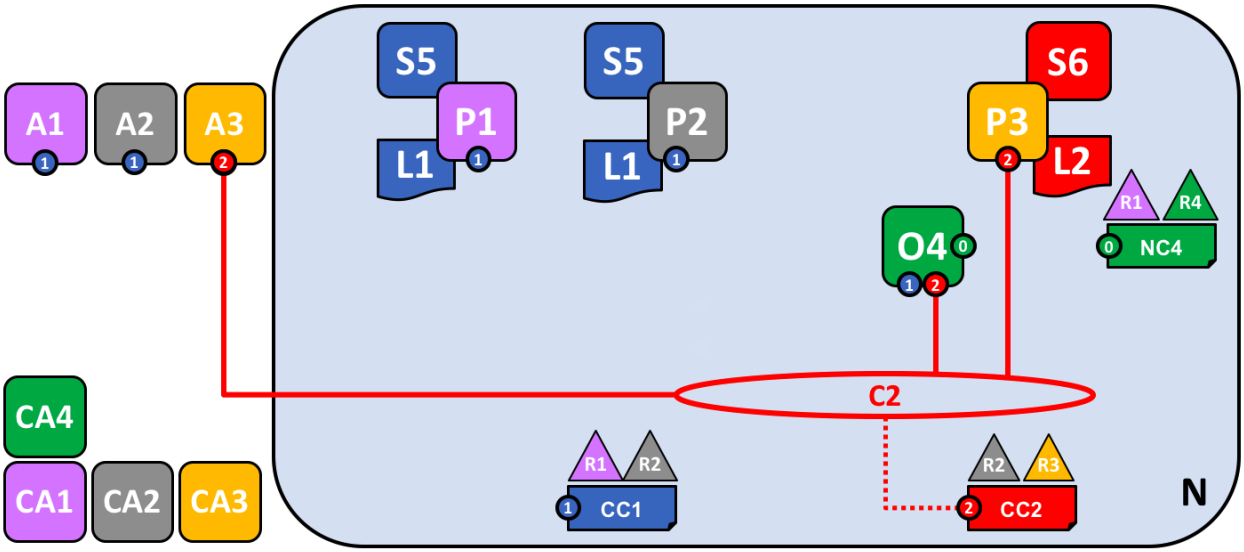
Indeed, the ordering service nodes operate a mini-blockchain, connected via the **system channel** we mentioned earlier. Using the system channel ordering service nodes distribute network configuration transactions. These transactions are used to co-operatively maintain a consistent copy of the network configuration at each ordering service node. In a similar way, peer nodes in an **application channel** can distribute channel configuration transactions. Likewise, these transactions are used to maintain a consistent copy of the channel configuration at each peer node.

This balance between objects that are logically singular, by being physically distributed is a common pattern in Hyperledger Fabric. Objects like network configurations, that are logically single, turn out to be physically replicated among a set of ordering services nodes for example. We also see it with channel configurations, ledgers, and to some extent smart contracts which are installed in multiple places but whose interfaces exist logically at the channel level. It's a pattern you see repeated time and again in Hyperledger Fabric, and enables Hyperledger Fabric to be both de-centralized and yet manageable at the same time.

4.3.13 Adding another peer

Now that organization R3 is able to fully participate in channel C2, let's add its infrastructure components to the channel. Rather than do this one component at a time, we're going to add a peer, its local copy of a ledger, a smart contract and a client application all at once!

Let's see the network with organization R3's components added:



The diagram shows the facts relating to channels C1 and C2 in the network N as follows: Client applications A1 and A2 can use channel C1 for communication with peers P1 and P2, and ordering service O4; client applications A3 can use channel C2 for communication with peer P3 and ordering service O4. Ordering service O4 can make use of the communication services of channels C1 and C2. Channel configuration CC1 applies to channel C1, CC2 applies to channel C2.

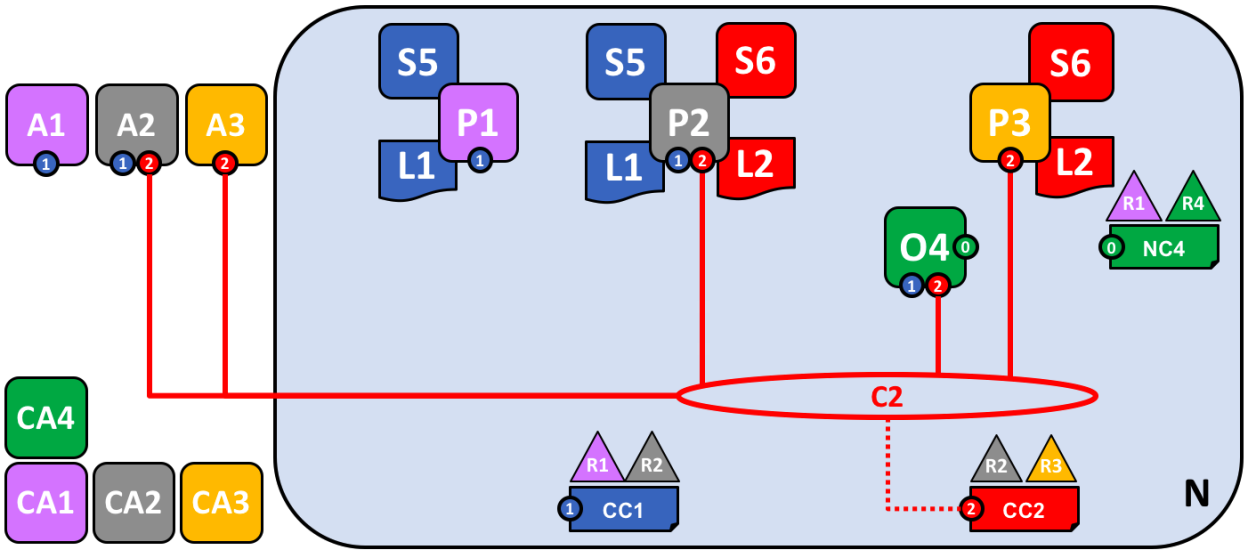
First of all, notice that because peer node P3 is connected to channel C2, it has a **different** ledger – L2 – to those peer nodes using channel C1. The ledger L2 is effectively scoped to channel C2. The ledger L1 is completely separate; it is scoped to channel C1. This makes sense – the purpose of the channel C2 is to provide private communications between the members of the consortium X2, and the ledger L2 is the private store for their transactions.

In a similar way, the smart contract S6, installed on peer node P3, and defined on channel C2, is used to provide controlled access to ledger L2. Application A3 can now use channel C2 to invoke the services provided by smart contract S6 to generate transactions that can be accepted onto every copy of the ledger L2 in the network.

At this point in time, we have a single network that has two completely separate channels defined within it. These channels provide independently managed facilities for organizations to transact with each other. Again, this is decentralization at work; we have a balance between control and autonomy. This is achieved through policies which are applied to channels which are controlled by, and affect, different organizations.

4.3.14 Joining a peer to multiple channels

In this final stage of network development, let's return our focus to organization R2. We can exploit the fact that R2 is a member of both consortia X1 and X2 by joining it to multiple channels:



The diagram shows the facts relating to channels C1 and C2 in the network N as follows: Client applications A1 can use channel C1 for communication with peers P1 and P2, and ordering service O4; client application A2 can use channel C1 for communication with peers P1 and P2 and channel C2 for communication with peers P2 and P3 and ordering service O4; client application A3 can use channel C2 for communication with peer P3 and P2 and ordering service O4. Ordering service O4 can make use of the communication services of channels C1 and C2. Channel configuration CC1 applies to channel C1, CC2 applies to channel C2.

We can see that R2 is a special organization in the network, because it is the only organization that is a member of two application channels! It is able to transact with organization R1 on channel C1, while at the same time it can also transact with organization R3 on a different channel, C2.

Notice how peer node P2 has smart contract S5 installed for channel C1 and smart contract S6 installed for channel C2. Peer node P2 is a full member of both channels at the same time via different smart contracts for different ledgers.

This is a very powerful concept – channels provide both a mechanism for the separation of organizations, and a mechanism for collaboration between organizations. All the while, this infrastructure is provided by, and shared between, a set of independent organizations.

It is also important to note that peer node P2's behaviour is controlled very differently depending upon the channel in which it is transacting. Specifically, the policies contained in channel configuration CC1 dictate the operations available to P2 when it is transacting in channel C1, whereas it is the policies in channel configuration CC2 that control P2's behaviour in channel C2.

Again, this is desirable – R2 and R1 agreed the rules for channel C1, whereas R2 and R3 agreed the rules for channel C2. These rules were captured in the respective channel policies – they can and must be used by every component in a channel to enforce correct behaviour, as agreed.

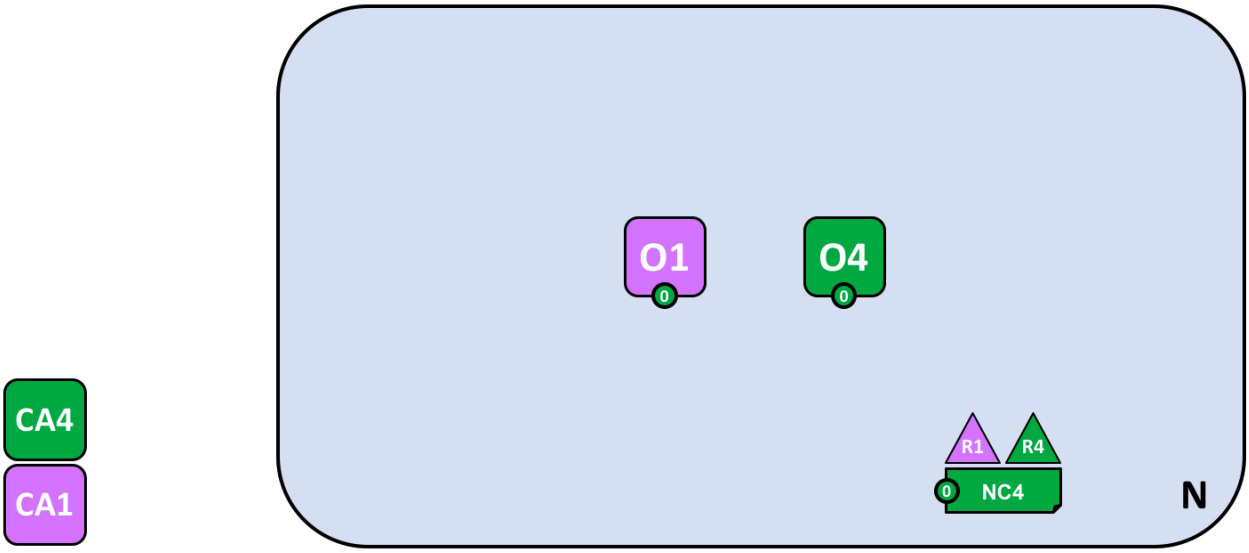
Similarly, we can see that client application A2 is now able to transact on channels C1 and C2. And likewise, it too will be governed by the policies in the appropriate channel configurations. As an aside, note that client application A2 and peer node P2 are using a mixed visual vocabulary – both lines and connections. You can see that they are equivalent; they are visual synonyms.

The ordering service

The observant reader may notice that the ordering service node appears to be a centralized component; it was used to create the network initially, and connects to every channel in the network. Even though we added R1 and R4 to the network configuration policy NC4 which controls the orderer, the node was running on R4's infrastructure. In a world of de-centralization, this looks wrong!

Don't worry! Our example network showed the simplest ordering service configuration to help you understand the idea of a network administration point. In fact, the ordering service can itself too be completely de-centralized! We mentioned earlier that an ordering service could be comprised of many individual nodes owned by different organizations, so let's see how that would be done in our sample network.

Let's have a look at a more realistic ordering service node configuration:



A multi-organization ordering service. The ordering service comprises ordering service nodes O1 and O4. O1 is provided by organization R1 and node O4 is provided by organization R4. The network configuration NC4 defines network resource permissions for actors from both organizations R1 and R4.

We can see that this ordering service completely de-centralized – it runs in organization R1 and it runs in organization R4. The network configuration policy, NC4, permits R1 and R4 equal rights over network resources. Client applications and peer nodes from organizations R1 and R4 can manage network resources by connecting to either node O1 or node O4, because both nodes behave the same way, as defined by the policies in network configuration NC4. In practice, actors from a particular organization *tend* to use infrastructure provided by their home organization, but that's certainly not always the case.

De-centralized transaction distribution

As well as being the management point for the network, the ordering service also provides another key facility – it is the distribution point for transactions. The ordering service is the component which gathers endorsed transactions from applications and orders them into transaction blocks, which are subsequently distributed to every peer node in the channel. At each of these committing peers, transactions are recorded, whether valid or invalid, and their local copy of the ledger updated appropriately.

Notice how the ordering service node O4 performs a very different role for the channel C1 than it does for the network N. When acting at the channel level, O4's role is to gather transactions and distribute blocks inside channel C1. It does this according to the policies defined in channel configuration CC1. In contrast, when acting at the network level, O4's role is to provide a management point for network resources according to the policies defined in network configuration NC4. Notice again how these roles are defined by different policies within the channel and network configurations respectively. This should reinforce to you the importance of declarative policy based configuration in Hyperledger Fabric. Policies both define, and are used to control, the agreed behaviours by each and every member of a consortium.

We can see that the ordering service, like the other components in Hyperledger Fabric, is a fully de-centralized component. Whether acting as a network management point, or as a distributor of blocks in a channel, its nodes can be

distributed as required throughout the multiple organizations in a network.

Changing policy

Throughout our exploration of the sample network, we've seen the importance of the policies to control the behaviour of the actors in the system. We've only discussed a few of the available policies, but there are many that can be declaratively defined to control every aspect of behaviour. These individual policies are discussed elsewhere in the documentation.

Most importantly of all, Hyperledger Fabric provides a uniquely powerful policy that allows network and channel administrators to manage policy change itself! The underlying philosophy is that policy change is a constant, whether it occurs within or between organizations, or whether it is imposed by external regulators. For example, new organizations may join a channel, or existing organizations may have their permissions increased or decreased. Let's investigate a little more how change policy is implemented in Hyperledger Fabric.

The key point of understanding is that policy change is managed by a policy within the policy itself. The **modification policy**, or **mod_policy** for short, is a first class policy within a network or channel configuration that manages change. Let's give two brief examples of how we've **already** used mod_policy to manage change in our network!

The first example was when the network was initially set up. At this time, only organization R4 was allowed to manage the network. In practice, this was achieved by making R4 the only organization defined in the network configuration NC4 with permissions to network resources. Moreover, the mod_policy for NC4 only mentioned organization R4 – only R4 was allowed to change this configuration.

We then evolved the network N to also allow organization R1 to administer the network. R4 did this by adding R1 to the policies for channel creation and consortium creation. Because of this change, R1 was able to define the consortia X1 and X2, and create the channels C1 and C2. R1 had equal administrative rights over the channel and consortium policies in the network configuration.

R4 however, could grant even more power over the network configuration to R1! R4 could add R1 to the mod_policy such that R1 would be able to manage change of the network policy too.

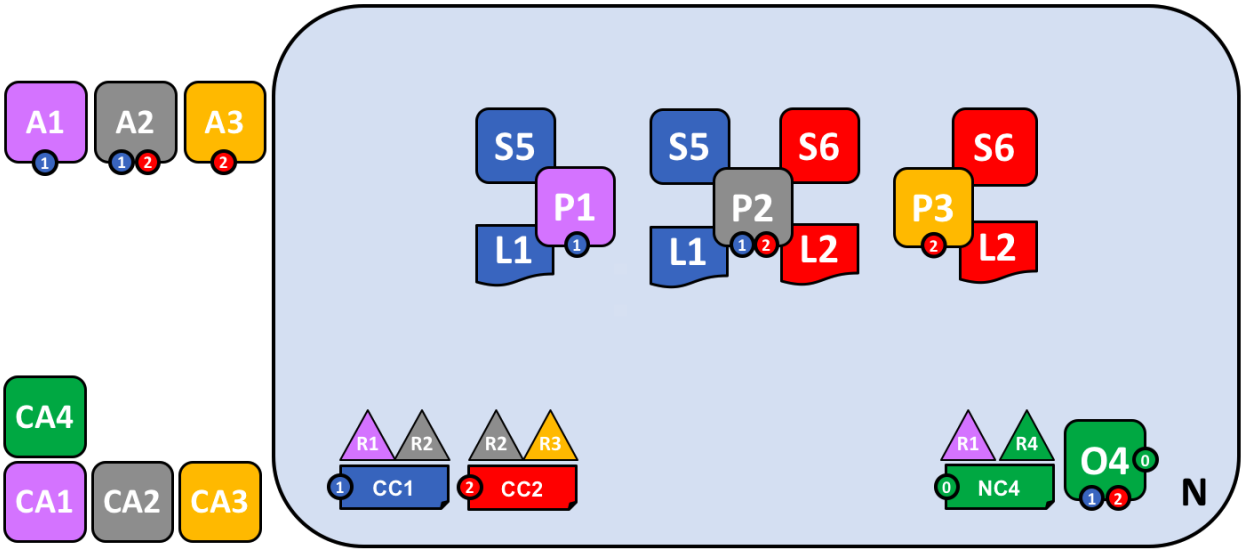
This second power is much more powerful than the first, because R1 now has **full control** over the network configuration NC4! This means that R1 can, in principle remove R4's management rights from the network. In practice, R4 would configure the mod_policy such that R4 would need to also approve the change, or that all organizations in the mod_policy would have to approve the change. There's lots of flexibility to make the mod_policy as sophisticated as it needs to be to support whatever change process is required.

This is mod_policy at work – it has allowed the graceful evolution of a basic configuration into a sophisticated one. All the time this has occurred with the agreement of all organization involved. The mod_policy behaves like every other policy inside a network or channel configuration; it defines a set of organizations that are allowed to change the mod_policy itself.

We've only scratched the surface of the power of policies and mod_policy in particular in this subsection. It is discussed at much more length in the policy topic, but for now let's return to our finished network!

4.3.15 Network fully formed

Let's recap what our network looks like using a consistent visual vocabulary. We've re-organized it slightly using our more compact visual syntax, because it better accommodates larger topologies:



In this diagram we see that the Fabric blockchain network consists of two application channels and one ordering channel. The organizations R1 and R4 are responsible for the ordering channel, R1 and R2 are responsible for the blue application channel while R2 and R3 are responsible for the red application channel. Client applications A1 is an element of organization R1, and CA1 is it's certificate authority. Note that peer P2 of organization R2 can use the communication facilities of the blue and the red application channel. Each application channel has its own channel configuration, in this case CC1 and CC2. The channel configuration of the system channel is part of the network configuration, NC4.

We're at the end of our conceptual journey to build a sample Hyperledger Fabric blockchain network. We've created a four organization network with two channels and three peer nodes, with two smart contracts and an ordering service. It is supported by four certificate authorities. It provides ledger and smart contract services to three client applications, who can interact with it via the two channels. Take a moment to look through the details of the network in the diagram, and feel free to read back through the topic to reinforce your knowledge, or go to a more detailed topic.

Summary of network components

Here's a quick summary of the network components we've discussed:

- Ledger. One per channel. Comprised of the Blockchain and the World state
- Smart contract (aka chaincode)
- Peer nodes
- Ordering service
- Channel
- Certificate Authority

4.3.16 Network summary

In this topic, we've seen how different organizations share their infrastructure to provide an integrated Hyperledger Fabric blockchain network. We've seen how the collective infrastructure can be organized into channels that provide private communications mechanisms that are independently managed. We've seen how actors such as client applications, administrators, peers and orderers are identified as being from different organizations by their use of certificates

from their respective certificate authorities. And in turn, we've seen the importance of policy to define the agreed permissions that these organizational actors have over network and channel resources.

4.4 Identity

4.4.1 What is an Identity?

The different actors in a blockchain network include peers, orderers, client applications, administrators and more. Each of these actors — active elements inside or outside a network able to consume services — has a digital identity encapsulated in an X.509 digital certificate. These identities really matter because they **determine the exact permissions over resources and access to information that actors have in a blockchain network.**

A digital identity furthermore has some additional attributes that Fabric uses to determine permissions, and it gives the union of an identity and the associated attributes a special name — **principal**. Principals are just like userIDs or groupIDs, but a little more flexible because they can include a wide range of properties of an actor's identity, such as the actor's organization, organizational unit, role or even the actor's specific identity. When we talk about principals, they are the properties which determine their permissions.

For an identity to be **verifiable**, it must come from a **trusted** authority. A membership service provider (MSP) is that trusted authority in Fabric. More specifically, an MSP is a component that defines the rules that govern the valid identities for this organization. The default MSP implementation in Fabric uses X.509 certificates as identities, adopting a traditional Public Key Infrastructure (PKI) hierarchical model (more on PKI later).

4.4.2 A Simple Scenario to Explain the Use of an Identity

Imagine that you visit a supermarket to buy some groceries. At the checkout you see a sign that says that only Visa, Mastercard and AMEX cards are accepted. If you try to pay with a different card — let's call it an "ImagineCard" — it doesn't matter whether the card is authentic and you have sufficient funds in your account. It will be not be accepted.

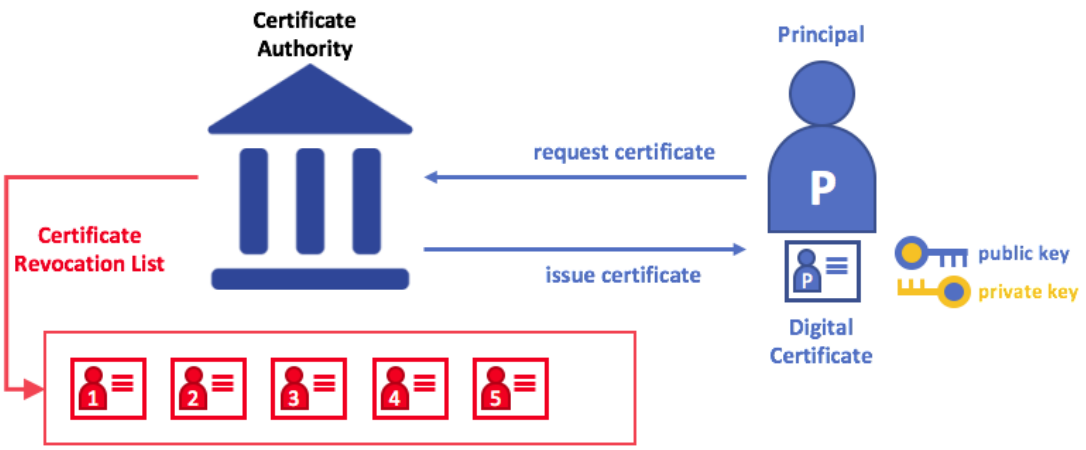
Having a valid credit card is not enough — it must also be accepted by the store! PKIs and MSPs work together in the same way — a PKI provides a list of identities, and an MSP says which of these are members of a given organization that participates in the network.

PKI certificate authorities and MSPs provide a similar combination of functionalities. A PKI is like a card provider — it dispenses many different types of verifiable identities. An MSP, on the other hand, is like the list of card providers accepted by the store, determining which identities are the trusted members (actors) of the store payment network. **MSPs turn verifiable identities into the members of a blockchain network**.

Let's drill into these concepts in a little more detail.

4.4.3 What are PKIs?

A public key infrastructure (PKI) is a collection of internet technologies that provides secure communications in a network. It's PKI that puts the **S** in **HTTPS** — and if you're reading this documentation on a web browser, you're probably using a PKI to make sure it comes from a verified source.



The elements of Public Key Infrastructure (PKI). A PKI is comprised of Certificate Authorities who issue digital certificates to parties (e.g., users of a service, service provider), who then use them to authenticate themselves in the messages they exchange in their environment. A CA's Certificate Revocation List (CRL) constitutes a reference for the certificates that are no longer valid. Revocation of a certificate can happen for a number of reasons. For example, a certificate may be revoked because the cryptographic private material associated to the certificate has been exposed.

Although a blockchain network is more than a communications network, it relies on the PKI standard to ensure secure communication between various network participants, and to ensure that messages posted on the blockchain are properly authenticated. It's therefore important to understand the basics of PKI and then why MSPs are so important.

There are four key elements to PKI:

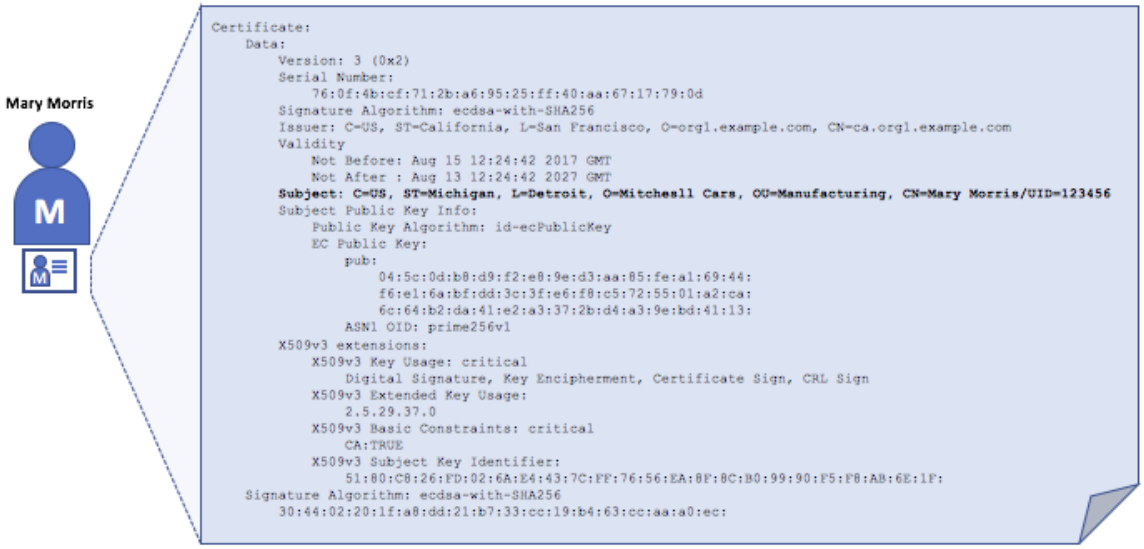
- **Digital Certificates**
- **Public and Private Keys**
- **Certificate Authorities**
- **Certificate Revocation Lists**

Let's quickly describe these PKI basics, and if you want to know more details, Wikipedia is a good place to start.

4.4.4 Digital Certificates

A digital certificate is a document which holds a set of attributes relating to the holder of the certificate. The most common type of certificate is the one compliant with the X.509 standard, which allows the encoding of a party's identifying details in its structure.

For example, Mary Morris in the Manufacturing Division of Mitchell Cars in Detroit, Michigan might have a digital certificate with a `SUBJECT` attribute of `C=US`, `ST=Michigan`, `L=Detroit`, `O=Mitchell Cars`, `OU=Manufacturing`, `CN=Mary Morris /UID=123456`. Mary's certificate is similar to her government identity card — it provides information about Mary which she can use to prove key facts about her. There are many other attributes in an X.509 certificate, but let's concentrate on just these for now.



A digital certificate describing a party called Mary Morris. Mary is the `SUBJECT` of the certificate, and the highlighted `SUBJECT` text shows key facts about Mary. The certificate also holds many more pieces of information, as you can see. Most importantly, Mary's public key is distributed within her certificate, whereas her private signing key is not. This signing key must be kept private.

What is important is that all of Mary's attributes can be recorded using a mathematical technique called cryptography (literally, "*secret writing*") so that tampering will invalidate the certificate. Cryptography allows Mary to present her certificate to others to prove her identity so long as the other party trusts the certificate issuer, known as a **Certificate Authority** (CA). As long as the CA keeps certain cryptographic information securely (meaning, its own **private signing key**), anyone reading the certificate can be sure that the information about Mary has not been tampered with — it will always have those particular attributes for Mary Morris. Think of Mary's X.509 certificate as a digital identity card that is impossible to change.

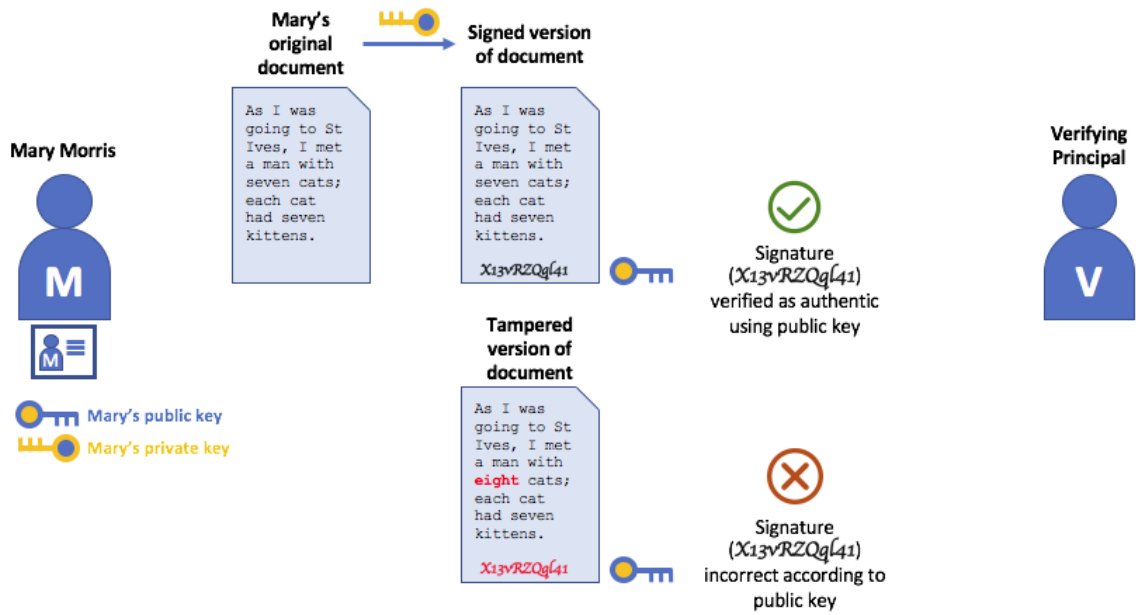
4.4.5 Authentication, Public keys, and Private Keys

Authentication and message integrity are important concepts in secure communications. Authentication requires that parties who exchange messages are assured of the identity that created a specific message. For a message to have "integrity" means that cannot have been modified during its transmission. For example, you might want to be sure you're communicating with the real Mary Morris rather than an impersonator. Or if Mary has sent you a message, you might want to be sure that it hasn't been tampered with by anyone else during transmission.

Traditional authentication mechanisms rely on **digital signatures** that, as the name suggests, allow a party to digitally **sign** its messages. Digital signatures also provide guarantees on the integrity of the signed message.

Technically speaking, digital signature mechanisms require each party to hold two cryptographically connected keys: a public key that is made widely available and acts as authentication anchor, and a private key that is used to produce **digital signatures** on messages. Recipients of digitally signed messages can verify the origin and integrity of a received message by checking that the attached signature is valid under the public key of the expected sender.

The unique relationship between a private key and the respective public key is the cryptographic magic that makes secure communications possible. The unique mathematical relationship between the keys is such that the private key can be used to produce a signature on a message that only the corresponding public key can match, and only on the same message.

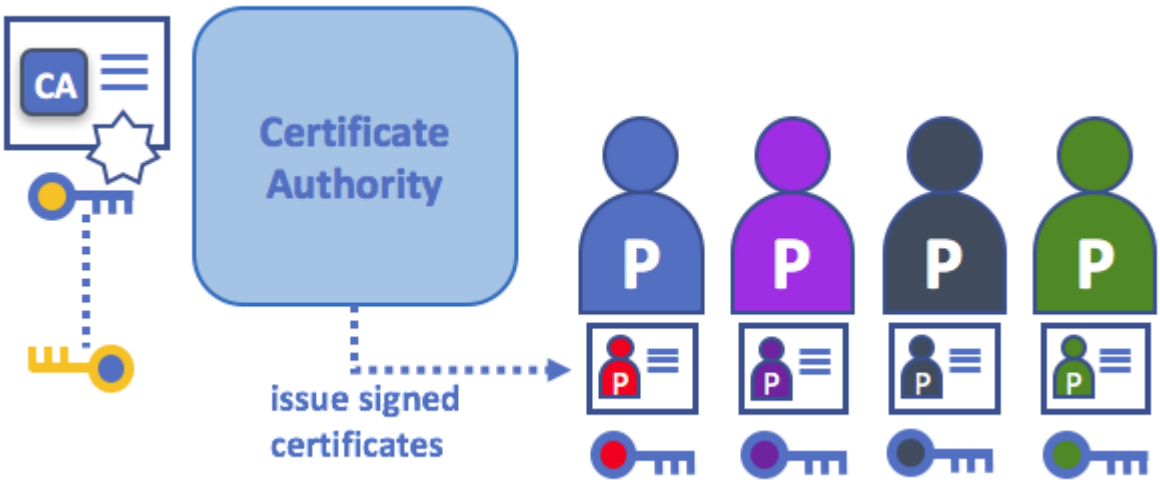


In the example above, Mary uses her private key to sign the message. The signature can be verified by anyone who sees the signed message using her public key.

4.4.6 Certificate Authorities

As you've seen, an actor or a node is able to participate in the blockchain network, via the means of a **digital identity** issued for it by an authority trusted by the system. In the most common case, digital identities (or simply **identities**) have the form of cryptographically validated digital certificates that comply with X.509 standard and are issued by a Certificate Authority (CA).

CAs are a common part of internet security protocols, and you've probably heard of some of the more popular ones: Symantec (originally Verisign), GeoTrust, DigiCert, GoDaddy, and Comodo, among others.



A Certificate Authority dispenses certificates to different actors. These certificates are digitally signed by the CA and

bind together the actor with the actor's public key (and optionally with a comprehensive list of properties). As a result, if one trusts the CA (and knows its public key), it can trust that the specific actor is bound to the public key included in the certificate, and owns the included attributes, by validating the CA's signature on the actor's certificate.

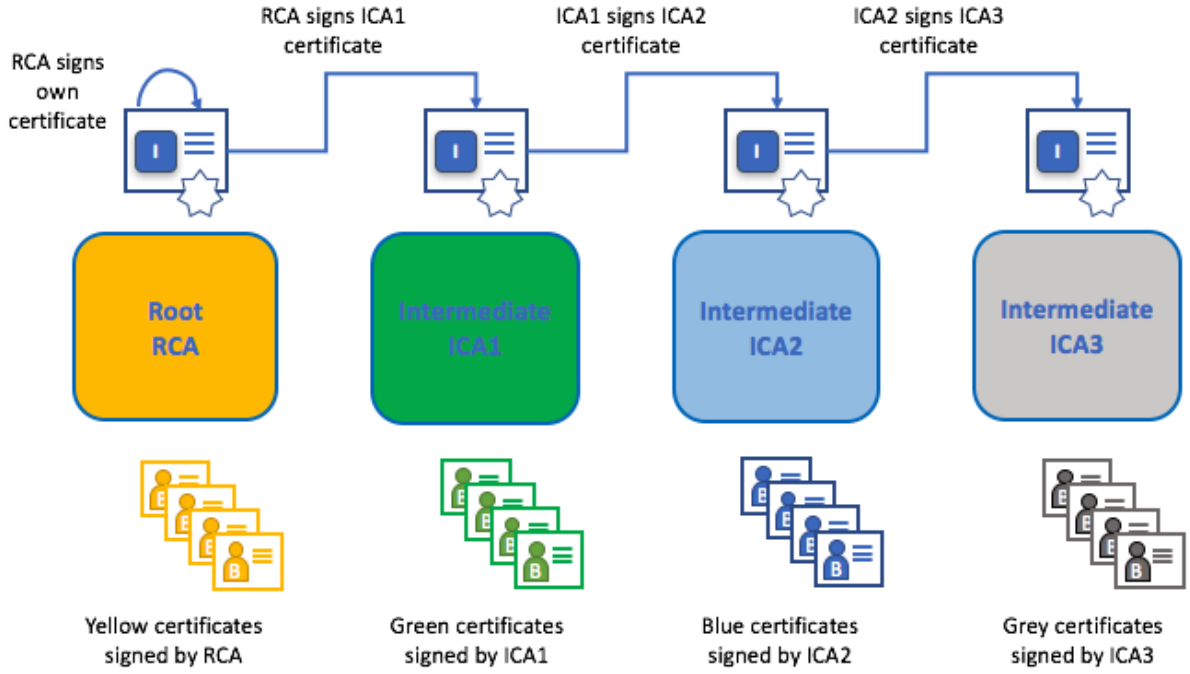
Certificates can be widely disseminated, as they do not include either the actors' nor the CA's private keys. As such they can be used as anchor of trusts for authenticating messages coming from different actors.

CAs also have a certificate, which they make widely available. This allows the consumers of identities issued by a given CA to verify them by checking that the certificate could only have been generated by the holder of the corresponding private key (the CA).

In a blockchain setting, every actor who wishes to interact with the network needs an identity. In this setting, you might say that **one or more CAs** can be used to **define the members of an organization's from a digital perspective**. It's the CA that provides the basis for an organization's actors to have a verifiable digital identity.

Root CAs, Intermediate CAs and Chains of Trust

CAs come in two flavors: **Root CAs** and **Intermediate CAs**. Because Root CAs (Symantec, Geotrust, etc) have to **securely distribute** hundreds of millions of certificates to internet users, it makes sense to spread this process out across what are called *Intermediate CAs*. These Intermediate CAs have their certificates issued by the root CA or another intermediate authority, allowing the establishment of a "chain of trust" for any certificate that is issued by any CA in the chain. This ability to track back to the Root CA not only allows the function of CAs to scale while still providing security — allowing organizations that consume certificates to use Intermediate CAs with confidence — it limits the exposure of the Root CA, which, if compromised, would endanger the entire chain of trust. If an Intermediate CA is compromised, on the other hand, there will be a much smaller exposure.



A chain of trust is established between a Root CA and a set of Intermediate CAs as long as the issuing CA for the certificate of each of these Intermediate CAs is either the Root CA itself or has a chain of trust to the Root CA.

Intermediate CAs provide a huge amount of flexibility when it comes to the issuance of certificates across multiple organizations, and that's very helpful in a permissioned blockchain system (like Fabric). For example, you'll see that different organizations may use different Root CAs, or the same Root CA with different Intermediate CAs — it really does depend on the needs of the network.

Fabric CA

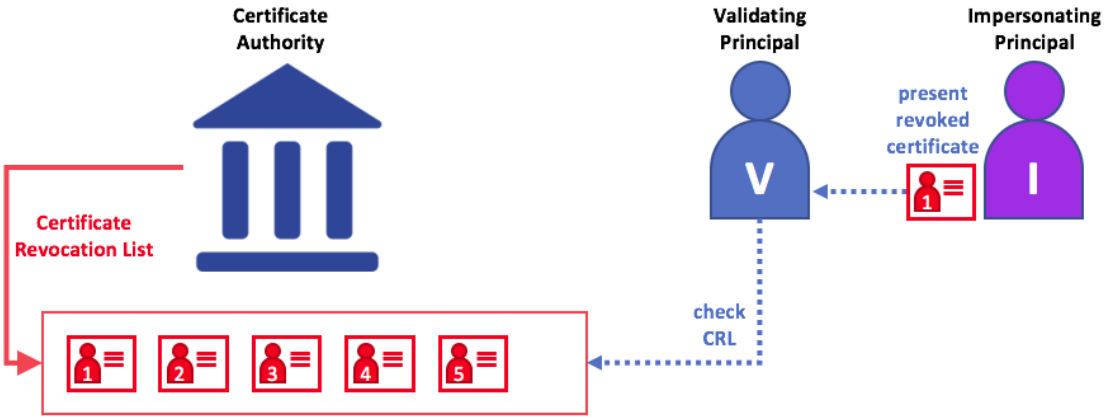
It's because CAs are so important that Fabric provides a built-in CA component to allow you to create CAs in the blockchain networks you form. This component — known as **Fabric CA** is a private root CA provider capable of managing digital identities of Fabric participants that have the form of X.509 certificates. Because Fabric CA is a custom CA targeting the Root CA needs of Fabric, it is inherently not capable of providing SSL certificates for general/automatic use in browsers. However, because **some** CA must be used to manage identity (even in a test environment), Fabric CA can be used to provide and manage certificates. It is also possible — and fully appropriate — to use a public/commercial root or intermediate CA to provide identification.

If you're interested, you can read a lot more about Fabric CA in the CA documentation section.

4.4.7 Certificate Revocation Lists

A Certificate Revocation List (CRL) is easy to understand — it's just a list of references to certificates that a CA knows to be revoked for one reason or another. If you recall the store scenario, a CRL would be like a list of stolen credit cards.

When a third party wants to verify another party's identity, it first checks the issuing CA's CRL to make sure that the certificate has not been revoked. A verifier doesn't have to check the CRL, but if they don't they run the risk of accepting a compromised identity.



Using a CRL to check that a certificate is still valid. If an impersonator tries to pass a compromised digital certificate to a validating party, it can be first checked against the issuing CA's CRL to make sure it's not listed as no longer valid.

Note that a certificate being revoked is very different from a certificate expiring. Revoked certificates have not expired — they are, by every other measure, a fully valid certificate. For more in-depth information about CRLs, click here.

Now that you've seen how a PKI can provide verifiable identities through a chain of trust, the next step is to see how these identities can be used to represent the trusted members of a blockchain network. That's where a Membership Service Provider (MSP) comes into play — **it identifies the parties who are the members of a given organization in the blockchain network**.

To learn more about membership, check out the conceptual documentation on MSPs.

4.5 Membership Service Provider (MSP)

4.5.1 Why do I need an MSP?

Because Fabric is a permissioned network, blockchain participants need a way to prove their identity to the rest of the network in order to transact on the network. If you've read through the documentation on Identity you've seen how a Public Key Infrastructure (PKI) can provide verifiable identities through a chain of trust. How is that chain of trust used by the blockchain network?

Certificate Authorities issue identities by generating a public and private key which forms a key-pair that can be used to prove identity. Because a private key can never be shared publicly, a mechanism is required to enable that proof which is where the MSP comes in. For example, a peer uses its private key to digitally sign, or endorse, a transaction. The MSP on the ordering service contains the peer's public key which is then used to verify that the signature attached to the transaction is valid. The private key is used to produce a signature on a transaction that only the corresponding public key, that is part of an MSP, can match. Thus, the MSP is the mechanism that allows that identity to be trusted and recognized by the rest of the network without ever revealing the member's private key.

Recall from the credit card scenario in the Identity topic that the Certificate Authority is like a card provider — it dispenses many different types of verifiable identities. An MSP, on the other hand, determines which credit card providers are accepted at the store. In this way, the MSP turns an identity (the credit card) into a role (the ability to buy things at the store).

This ability to turn verifiable identities into roles is fundamental to the way Fabric networks function, since it allows organizations, nodes, and channels the ability establish MSPs that determine who is allowed to do what at the organization, node, and channel level.

Identities are similar to your credit cards that are used to prove you can pay. The MSP is similar to the list of accepted credit cards.

Consider a consortium of banks that operate a blockchain network. Each bank operates peer and ordering nodes, and the peers endorse transactions submitted to the network. However, each bank would also have departments and account holders. The account holders would belong to each organization, but would not run nodes on the network. They would only interact with the system from their mobile or web application. So how does the network recognize and differentiate these identities? A CA was used to create the identities, but like the card example, those identities can't just be issued, they need to be recognized by the network. MSPs are used to define the organizations that are trusted by the network members. MSPs are also the mechanism that provide members with a set of roles and permissions within the network. Because the MSPs defining these organizations are known to the members of a network, they can then be used to validate that network entities that attempt to perform actions are allowed to.

Finally, consider if you want to join an *existing* network, you need a way to turn your identity into something that is recognized by the network. The MSP is the mechanism that enables you to participate on a permissioned blockchain

network. To transact on a Fabric network a member needs to:

1. Have an identity issued by a CA that is trusted by the network.
2. Become a member of an *organization* that is recognized and approved by the network members. The MSP is how the identity is linked to the membership of an organization. Membership is achieved by adding the member's public key (also known as certificate, signing cert, or signcert) to the organization's MSP.
3. Add the MSP to either a consortium on the network or a channel.
4. Ensure the MSP is included in the policy definitions on the network.

4.5.2 What is an MSP?

Despite its name, the Membership Service Provider does not actually provide anything. Rather, the implementation of the MSP requirement is a set of folders that are added to the configuration of the network and is used to define an organization both inwardly (organizations decide who its admins are) and outwardly (by allowing other organizations to validate that entities have the authority to do what they are attempting to do). Whereas Certificate Authorities generate the certificates that represent identities, the MSP contains a list of permissioned identities.

The MSP identifies which Root CAs and Intermediate CAs are accepted to define the members of a trust domain by listing the identities of their members, or by identifying which CAs are authorized to issue valid identities for their members.

But the power of an MSP goes beyond simply listing who is a network participant or member of a channel. It is the MSP that turns an identity into a **role** by identifying specific privileges an actor has on a node or channel. Note that when a user is registered with a Fabric CA, a role of admin, peer, client, orderer, or member must be associated with the user. For example, identities registered with the "peer" role should, naturally, be given to a peer. Similarly, identities registered with the "admin" role should be given to organization admins. We'll delve more into the significance of these roles later in the topic.

In addition, an MSP can allow for the identification of a list of identities that have been revoked — as discussed in the Identity documentation — but we will talk about how that process also extends to an MSP.

4.5.3 MSP domains

MSPs occur in two domains in a blockchain network:

- Locally on an actor's node (**local MSP**)
- In channel configuration (**channel MSP**)

The key difference between local and channel MSPs is not how they function – both turn identities into roles – but their **scope**. Each MSP lists roles and permissions at a particular level of administration.

Local MSPs

Local MSPs are defined for clients and for nodes (peers and orderers). Local MSPs define the permissions for a node (who are the peer admins who can operate the node, for example). The local MSPs of clients (the account holders in the banking scenario above), allow the user to authenticate itself in its transactions as a member of a channel (e.g. in chaincode transactions), or as the owner of a specific role into the system such as an organization admin, for example, in configuration transactions.

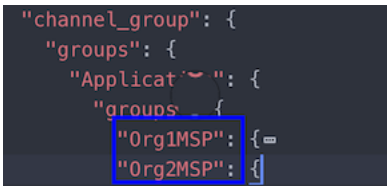
Every node must have a local MSP defined, as it defines who has administrative or participatory rights at that level (peer admins will not necessarily be channel admins, and vice versa). This allows for authenticating member messages outside the context of a channel and to define the permissions over a particular node (who has the ability to install chaincode on a peer, for example). Note that one or more nodes can be owned by an organization. An MSP

defines the organization admins. And the organization, the admin of the organization, the admin of the node, and the node itself should all have the same root of trust.

An orderer local MSP is also defined on the file system of the node and only applies to that node. Like peer nodes, orderers are also owned by a single organization and therefore have a single MSP to list the actors or nodes it trusts.

Channel MSPs

In contrast, **channel MSPs define administrative and participatory rights at the channel level**. Peers and ordering nodes on an application channel share the same view of channel MSPs, and will therefore be able to correctly authenticate the channel participants. This means that if an organization wishes to join the channel, an MSP incorporating the chain of trust for the organization's members would need to be included in the channel configuration. Otherwise transactions originating from this organization's identities will be rejected. Whereas local MSPs are represented as a folder structure on the file system, channel MSPs are described in a channel configuration.



Snippet from a channel config.json file that includes two organization MSPs.

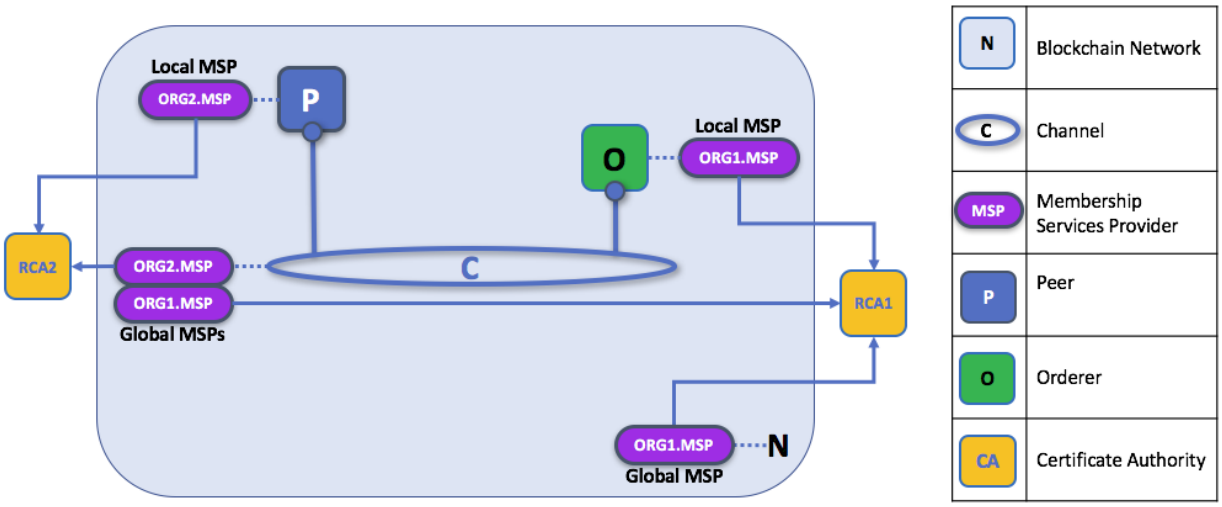
Channel MSPs identify who has authorities at a channel level. The channel MSP defines the *relationship* between the identities of channel members (which themselves are MSPs) and the enforcement of channel level policies. Channel MSPs contain the MSPs of the organizations of the channel members.

Every organization participating in a channel must have an MSP defined for it. In fact, it is recommended that there is a one-to-one mapping between organizations and MSPs. The MSP defines which members are empowered to act on behalf of the organization. This includes configuration of the MSP itself as well as approving administrative tasks that the organization has role, such as adding new members to a channel. If all network members were part of a single organization or MSP, data privacy is sacrificed. Multiple organizations facilitate privacy by segregating ledger data to only channel members. If more granularity is required within an organization, the organization can be further divided into organizational units (OUs) which we describe in more detail later in this topic.

The system channel MSP includes the MSPs of all the organizations that participate in an ordering service. An ordering service will likely include ordering nodes from multiple organizations and collectively these organizations run the ordering service, most importantly managing the consortium of organizations and the default policies that are inherited by the application channels.

Local MSPs are only defined on the file system of the node or user to which they apply. Therefore, physically and logically there is only one local MSP per node. However, as channel MSPs are available to all nodes in the channel, they are logically defined once in the channel configuration. However, **a channel MSP is also instantiated on the file system of every node in the channel and kept synchronized via consensus**. So while there is a copy of each channel MSP on the local file system of every node, logically a channel MSP resides on and is maintained by the channel or the network.

The following diagram illustrates how local and channel MSPs coexist on the network:



The MSPs for the peer and orderer are local, whereas the MSPs for a channel (including the network configuration channel, also known as the system channel) are global, shared across all participants of that channel. In this figure, the network system channel is administered by ORG1, but another application channel can be managed by ORG1 and ORG2. The peer is a member of and managed by ORG2, whereas ORG1 manages the orderer of the figure. ORG1 trusts identities from RCA1, whereas ORG2 trusts identities from RCA2. It is important to note that these are administration identities, reflecting who can administer these components. So while ORG1 administers the network, ORG2.MSP does exist in the network definition.

4.5.4 What role does an organization play in an MSP?

An **organization** is a logical managed group of members. This can be something as big as a multinational corporation or a small as a flower shop. What's most important about organizations (or **orgs**) is that they manage their members under a single MSP. The MSP allows an identity to be linked to an organization. Note that this is different from the organization concept defined in an X.509 certificate, which we mentioned above.

The exclusive relationship between an organization and its MSP makes it sensible to name the MSP after the organization, a convention you'll find adopted in most policy configurations. For example, organization `ORG1` would likely have an MSP called something like `ORG1-MSP`. In some cases an organization may require multiple membership groups — for example, where channels are used to perform very different business functions between organizations. In these cases it makes sense to have multiple MSPs and name them accordingly, e.g., `ORG2-MSP-NATIONAL` and `ORG2-MSP-GOVERNMENT`, reflecting the different membership roots of trust within `ORG2` in the `NATIONAL` sales channel compared to the `GOVERNMENT` regulatory channel.

Organizational Units (OUs) and MSPs

An organization can also be divided into multiple **organizational units**, each of which has a certain set of responsibilities, also referred to as `affiliations`. Think of an OU as a department inside an organization. For example, the `ORG1` organization might have both `ORG1.MANUFACTURING` and `ORG1.DISTRIBUTION` OUs to reflect these separate lines of business. When a CA issues X.509 certificates, the `OU` field in the certificate specifies the line of business to which the identity belongs. A benefit of using OUs like this is that these values can then be used in policy definitions in order to restrict access or in smart contracts for attribute-based access control. Otherwise, separate MSPs would need to be created for each organization.

Specifying OUs is optional. If OUs are not used, all of the identities that are part of an MSP — as identified by the Root CA and Intermediate CA folders — will be considered members of the organization.

Node OU Roles and MSPs

Additionally, there is a special kind of OU, sometimes referred to as a `Node OU`, that can be used to confer a role onto an identity. These Node OU roles are defined in the `$FABRIC_CFG_PATH/msp/config.yaml` file and contain a list of organizational units whose members are considered to be part of the organization represented by this MSP. This is particularly useful when you want to restrict the members of an organization to the ones holding an identity (signed by one of MSP designated CAs) with a specific Node OU role in it. For example, with node OU's you can implement a more granular endorsement policy that requires Org1 peers to endorse a transaction, rather than any member of Org1.

In order to use the Node OU roles, the "identity classification" feature must be enabled for the network. When using the folder-based MSP structure, this is accomplished by enabling "Node OUs" in the config.yaml file which resides in the root of the MSP folder:

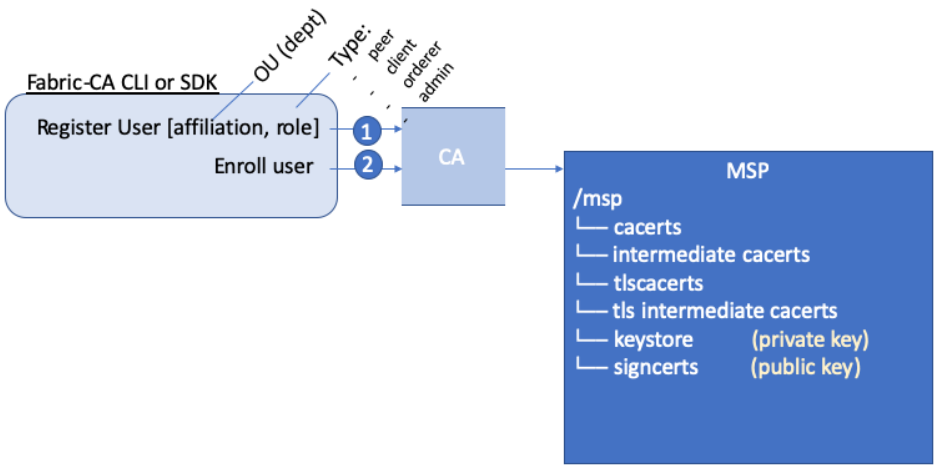
```
NodeOUs:
  Enable: true
  ClientOUIdentifier:
    Certificate: cacerts/ca.sampleorg-cert.pem
    OrganizationalUnitIdentifier: client
  PeerOUIdentifier:
    Certificate: cacerts/ca.sampleorg-cert.pem
    OrganizationalUnitIdentifier: peer
  AdminOUIdentifier:
    Certificate: cacerts/ca.sampleorg-cert.pem
    OrganizationalUnitIdentifier: admin
  OrdererOUIdentifier:
    Certificate: cacerts/ca.sampleorg-cert.pem
    OrganizationalUnitIdentifier: orderer
```

In the example above, there are 4 possible Node OU `ROLES` for the MSP:

- client
- peer
- admin
- orderer

This convention allows you to distinguish MSP roles by the OU present in the CommonName attribute of the X509 certificate. The example above says that any certificate issued by cacerts/ca.sampleorg-cert.pem in which OU=client will identified as a client, OU=peer as a peer, etc. Starting with Fabric v1.4.3, there is also an OU for the orderer and for admins. The new admins role means that you no longer have to explicitly place certs in the admincerts folder of the MSP directory. Rather, the `admin` role present in the user's signcert qualifies the identity as an admin user.

These Role and OU attributes are assigned to an identity when the Fabric CA or SDK is used to `register` a user with the CA. It is the subsequent `enroll` user command that generates the certificates in the users' `/msp` folder.



The resulting ROLE and OU attributes are visible inside the X.509 signing certificate located in the `/signcerts` folder. The `ROLE` attribute is identified as `hf.Type` and refers to an actor's role within its organization, (specifying, for example, that an actor is a `peer`). See the following snippet from a signing certificate shows how the Roles and OUs are represented in the certificate.

```
Certificate:
  Data
    Version: 3 (0x2)
    Serial Number:
      45:6a:4f:01:dc:fj:5d:b2:94:18:79:91:26:31:d8:0e:b0:9b:6b:88
  Signature Algorithm: ecdsa-with-SHA256
    Issuer: C=US, ST=New York, O=Hyperledger, OU=Fabric, CN=fabric-ca-server
    Validity
      Not Before: Nov 20 22:13:00 2019 GMT
      Not After : Nov 19 22:18:00 2020 GMT
    Subject: OU=peer, OU=ORG1, OU=DISTRIBUTION, CN=user1
             ROLE      ORGANIZATIONAL UNIT    ENROLL ID
            (Node OU)
    .
    .
    .
    X509v3 extensions:
      X509v3 Key Usage: critical
        Digital Signature
      X509v3 Basic Constraints: critical
        CA:FALSE
      X509v3 Subject Key Identifier:
        17:B0:9B:29:42:F6:44:E0:7D:02:C6:78:96:2D:97:14:7A:D7:FC:CA
      X509v3 Authority Key Identifier:
        keyid:DC:91:B7:85:A4:37:66:D0:D2:B7:62:A9:3F:59:83:D6:EB:01:E8:80
      1.2.3.4.5.6.7.8.1:
                    ORGANIZATIONAL UNIT          ENROLL ID        ROLE (Node OU)
        {"attrs":{"hf.Affiliation":"ORG1.DISTRIBUTION","hf.EnrollmentID":"user1","hf.Type":"peer"}}
```

Note: For Channel MSPs, just because an actor has the role of an administrator it doesn't mean that they can administer particular resources. The actual power a given identity has with respect to administering the system is determined by the *policies* that manage system resources. For example, a channel policy might specify that `ORG1-MANUFACTURING` administrators, meaning identities with a role of `admin` and a

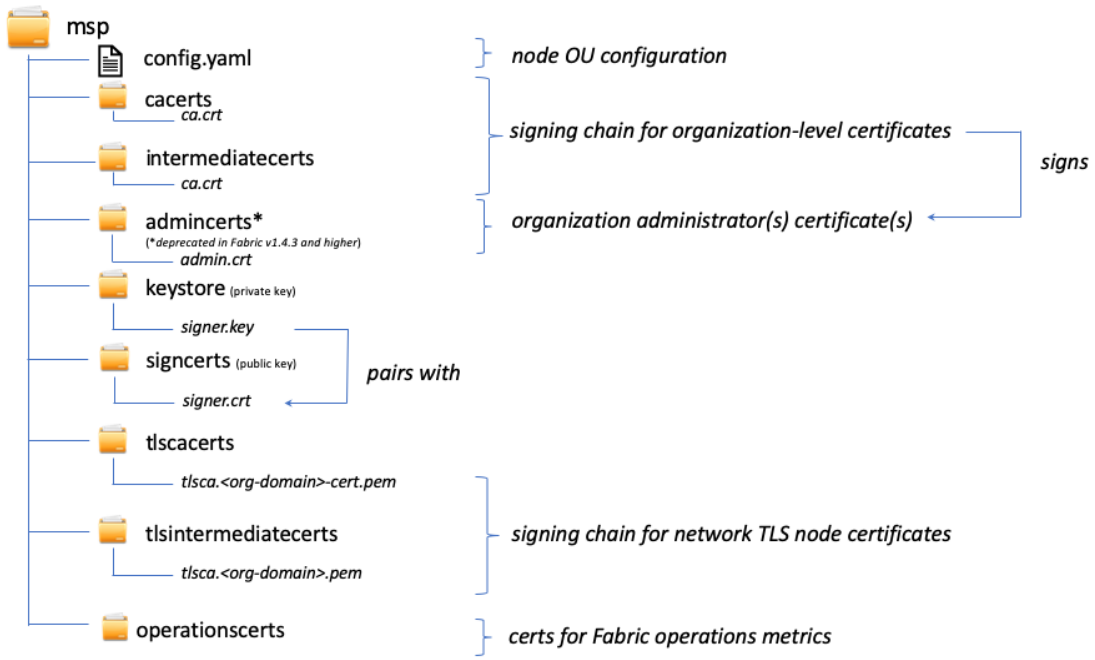
Node OU of `ORG1-MANUFACTURING`, have the rights to add new organizations to the channel, whereas the `ORG1-DISTRIBUTION` administrators have no such rights.

Finally, OUs could be used by different organizations in a consortium to distinguish each other. But in such cases, the different organizations have to use the same Root CAs and Intermediate CAs for their chain of trust, and assign the OU field to identify members of each organization. When every organization has the same CA or chain of trust, this makes the system more centralized than what might be desirable and therefore deserves careful consideration on a blockchain network.

4.5.5 MSP Structure

Let's explore the MSP elements that render the functionality we've described so far.

A local MSP folder contains the following sub-folders:



The figure above shows the subfolders in a local MSP on the file system

- **config.yaml:** Used to configure the identity classification feature in Fabric by enabling "Node OUs" and defining the accepted roles.
- **cacerts:** This folder contains a list of self-signed X.509 certificates of the Root CAs trusted by the organization represented by this MSP. There must be at least one Root CA certificate in this MSP folder.

  This is the most important folder because it identifies the CAs from which all other certificates must be derived to be considered members of the corresponding organization to form the chain of trust.
- **intermediatecerts:** This folder contains a list of X.509 certificates of the Intermediate CAs trusted by this organization. Each certificate must be signed by one of the Root CAs in the MSP or by any Intermediate CA whose issuing CA chain ultimately leads back to a trusted Root CA.

  An intermediate CA may represent a different subdivision of the organization (like `ORG1-MANUFACTURING` and `ORG1-DISTRIBUTION` do for `ORG1`), or the organization itself (as may be the case if a commercial CA is leveraged for the organization's identity management). In the latter case intermediate CAs can be used to represent organization subdivisions. Here you may find more information on best practices for MSP configuration.

Notice, that it is possible to have a functioning network that does not have an Intermediate CA, in which case this folder would be empty.

Like the Root CA folder, this folder defines the CAs from which certificates must be issued to be considered members of the organization.

- **admincerts (Deprecated from Fabric v1.4.3 and higher):** This folder contains a list of identities that define the actors who have the role of administrators for this organization. In general, there should be one or more X.509 certificates in this list.

  Note: Prior to Fabric v1.4.3, admins were defined by explicitly putting certs in the `admincerts` folder in the local MSP directory of your peer. **With Fabric v1.4.3 or higher, certificates in this folder are no longer required.** Instead, it is recommended that when the user is registered with the CA, that the `admin` role is used to designate the node administrator. Then, the identity is recognized as an `admin` by the Node OU role value in their signcert. As a reminder, in order to leverage the admin role, the "identity classification" feature must be enabled in the config.yaml above by setting "Node OUs" to `Enable: true`. We'll explore this more later.

  And as a reminder, for Channel MSPs, just because an actor has the role of an administrator it doesn't mean that they can administer particular resources. The actual power a given identity has with respect to administering the system is determined by the *policies* that manage system resources. For example, a channel policy might specify that `ORG1-MANUFACTURING` administrators have the rights to add new organizations to the channel, whereas the `ORG1-DISTRIBUTION` administrators have no such rights.

- **keystore: (private Key)** This folder is defined for the local MSP of a peer or orderer node (or in a client's local MSP), and contains the node's private key. This key is used to sign data — for example to sign a transaction proposal response, as part of the endorsement phase.

  This folder is mandatory for local MSPs, and must contain exactly one private key. Obviously, access to this folder must be limited only to the identities of users who have administrative responsibility on the peer.

  The **channel MSP** configuration does not include this folder, because channel MSPs solely aim to offer identity validation functionalities and not signing abilities.

  Note: If you are using a Hardware Security Module(HSM) for key management, this folder is empty because the private key is generated by and stored in the HSM.

- **signcert:** For a peer or orderer node (or in a client's local MSP) this folder contains the node's **signing key**. This key matches cryptographically the node's identity included in **Node Identity** folder and is used to sign data — for example to sign a transaction proposal response, as part of the endorsement phase.

  This folder is mandatory for local MSPs, and must contain exactly one public key. Obviously, access to this folder must be limited only to the identities of users who have administrative responsibility on the peer.

  Configuration of a **channel MSP** does not include this folder, as channel MSPs solely aim to offer identity validation functionalities and not signing abilities.

- **tlscacerts:** This folder contains a list of self-signed X.509 certificates of the Root CAs trusted by this organization **for secure communications between nodes using TLS**. An example of a TLS communication would be when a peer needs to connect to an orderer so that it can receive ledger updates.

  MSP TLS information relates to the nodes inside the network — the peers and the orderers, in other words, rather than the applications and administrations that consume the network.

  There must be at least one TLS Root CA certificate in this folder. For more information about TLS, see Securing Communication with Transport Layer Security (TLS).

- **tlsintermediatecacerts:** This folder contains a list intermediate CA certificates CAs trusted by the organization represented by this MSP **for secure communications between nodes using TLS**. This folder is specifically useful when commercial CAs are used for TLS certificates of an organization. Similar to membership intermediate CAs, specifying intermediate TLS CAs is optional.

- **operationscerts:** This folder contains the certificates required to communicate with the Fabric Operations Service API.

A channel MSP includes the following additional folder:

- **Revoked Certificates:** If the identity of an actor has been revoked, identifying information about the identity — not the identity itself — is held in this folder. For X.509-based identities, these identifiers are pairs of strings known as Subject Key Identifier (SKI) and Authority Access Identifier (AKI), and are checked whenever the certificate is being used to make sure the certificate has not been revoked.

  This list is conceptually the same as a CA's Certificate Revocation List (CRL), but it also relates to revocation of membership from the organization. As a result, the administrator of a channel MSP can quickly revoke an actor or node from an organization by advertising the updated CRL of the CA. This "list of lists" is optional. It will only become populated as certificates are revoked.

If you've read this doc as well as our doc on Identity, you should now have a pretty good grasp of how identities and MSPs work in Hyperledger Fabric. You've seen how a PKI and MSPs are used to identify the actors collaborating in a blockchain network. You've learned how certificates, public/private keys, and roots of trust work, in addition to how MSPs are physically and logically structured.

4.6 Policies

Audience: Architects, application and smart contract developers, administrators

In this topic, we'll cover:

- *What is a policy*
- *Why are policies needed*
- *How are policies implemented throughout Fabric*
- *Fabric policy domains*
- *How do you write a policy in Fabric*
- *Fabric chaincode lifecycle*
- *Overriding policy definitions*

4.6.1 What is a policy

At its most basic level, a policy is a set of rules that define the structure for how decisions are made and specific outcomes are reached. To that end, policies typically describe a **who** and a **what**, such as the access or rights that an individual has over an **asset**. We can see that policies are used throughout our daily lives to protect assets of value to us, from car rentals, health, our homes, and many more.

For example, an insurance policy defines the conditions, terms, limits, and expiration under which an insurance payout will be made. The policy is agreed to by the policy holder and the insurance company, and defines the rights and responsibilities of each party.

Whereas an insurance policy is put in place for risk management, in Hyperledger Fabric, policies are the mechanism for infrastructure management. Fabric policies represent how members come to agreement on accepting or rejecting changes to the network, a channel, or a smart contract. Policies are agreed to by the consortium members when a network is originally configured, but they can also be modified as the network evolves. For example, they describe the criteria for adding or removing members from a channel, change how blocks are formed, or specify the number of organizations required to endorse a smart contract. All of these actions are described by a policy which defines who can perform the action. Simply put, everything you want to do on a Fabric network is controlled by a policy.

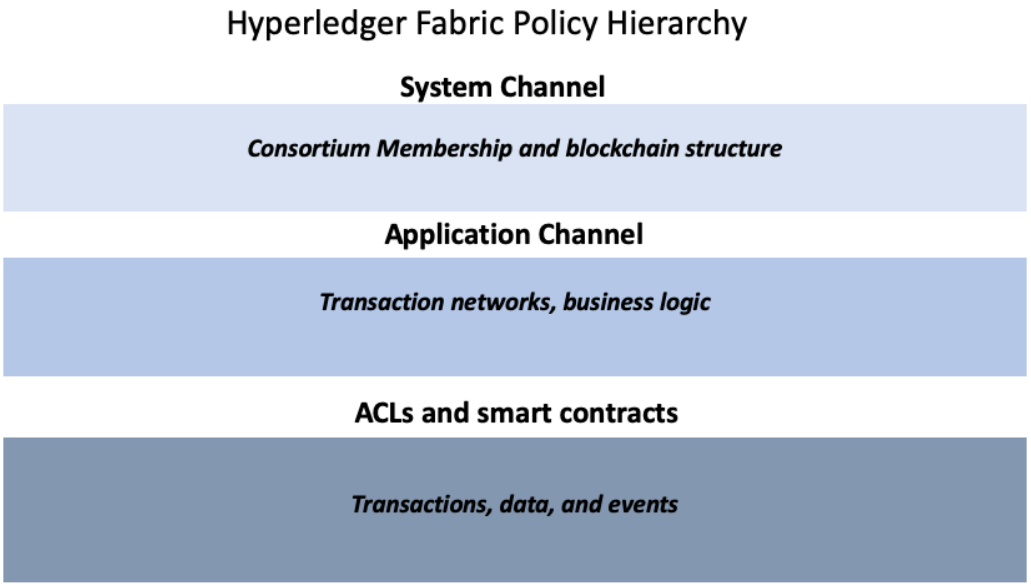
4.6.2 Why are policies needed

Policies are one of the things that make Hyperledger Fabric different from other blockchains like Ethereum or Bitcoin. In those systems, transactions can be generated and validated by any node in the network. The policies that govern the network are fixed at any point in time and can only be changed using the same process that governs the code. Because Fabric is a permissioned blockchain whose users are recognized by the underlying infrastructure, those users have the ability to decide on the governance of the network before it is launched, and change the governance of a running network.

Policies allow members to decide which organizations can access or update a Fabric network, and provide the mechanism to enforce those decisions. Policies contain the lists of organizations that have access to a given resource, such as a user or system chaincode. They also specify how many organizations need to agree on a proposal to update a resource, such as a channel or smart contracts. Once they are written, policies evaluate the collection of signatures attached to transactions and proposals and validate if the signatures fulfill the governance agreed to by the network.

4.6.3 How are policies implemented throughout Fabric

Policies are implemented at different levels of a Fabric network. Each policy domain governs different aspects of how a network operates.



A visual representation of the Fabric policy hierarchy.

System channel configuration

Every network begins with an ordering **system channel**. There must be exactly one ordering system channel for an ordering service, and it is the first channel to be created. The system channel also contains the organizations who are the members of the ordering service (ordering organizations) and those that are on the networks to transact (consortium organizations).

The policies in the ordering system channel configuration blocks govern the consensus used by the ordering service and define how new blocks are created. The system channel also governs which members of the consortium are allowed to create new channels.

Application channel configuration

Application *channels* are used to provide a private communication mechanism between organizations in the consortium.

The policies in an application channel govern the ability to add or remove members from the channel. Application channels also govern which organizations are required to approve a chaincode before the chaincode is defined and committed to a channel using the Fabric chaincode lifecyle. When an application channel is initially created, it inherits all the ordering service parameters from the orderer system channel by default. However, those parameters (and the policies governing them) can be customized in each channel.

Access control lists (ACLs)

Network administrators will be especially interested in the Fabric use of ACLs, which provide the ability to configure access to resources by associating those resources with existing policies. These "resources" could be functions on system chaincode (e.g., "GetBlockByNumber" on the "qscc" system chaincode) or other resources (e.g.,who can receive Block events). ACLs refer to policies defined in an application channel configuraton and extends them to control additional resources. The default set of Fabric ACLs is visible in the `configtx.yaml` file under the `Application: &ApplicationDefaults` section but they can and should be overridden in a production environment. The list of resources named in `configtx.yaml` is the complete set of all internal resources currently defined by Fabric.

In that file, ACLs are expressed using the following format:

```
# ACL policy for chaincode to chaincode invocation
peer/ChaincodeToChaincode: /Channel/Application/Readers
```

Where `peer/ChaincodeToChaincode` represents the resource being secured and `/Channel/Application/Readers` refers to the policy which must be satisfied for the associated transaction to be considered valid.

For a deeper dive into ACLS, refer to the topic in the Operations Guide on ACLs.

Smart contract endorsement policies

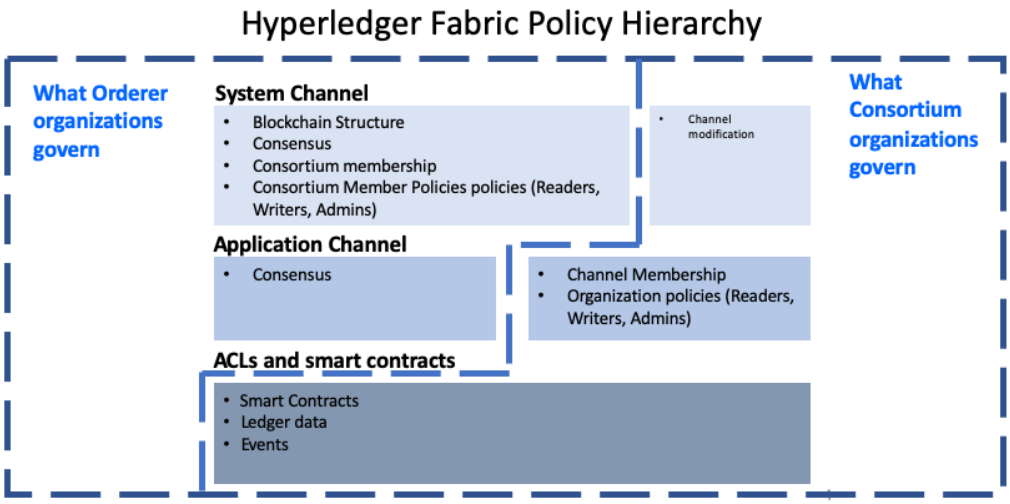
Every smart contract inside a chaincode package has an endorsement policy that specifies how many peers belonging to different channel members need to execute and validate a transaction against a given smart contract in order for the transaction to be considered valid. Hence, the endorsement policies define the organizations (through their peers) who must "endorse" (i.e., approve of) the execution of a proposal.

Modification policies

There is one last type of policy that is crucial to how policies work in Fabric, the `Modification policy`. Modification policies specify the group of identities required to sign (approve) any configuration *update*. It is the policy that defines how the policy is updated. Thus, each channel configuration element includes a reference to a policy which governs its modification.

4.6.4 The Fabric policy domains

While Fabric policies are flexible and can be configured to meet the needs of a network, the policy structure naturally leads to a division between the domains governed by either the Ordering Service organizations or the members of the consortium. In the following diagram you can see how the default policies implement control over the Fabric policy domains below.



A more detailed look at the policy domains governed by the Orderer organizations and consortium organizations.

A fully functional Fabric network can feature many organizations with different responsibilities. The domains provide the ability to extend different privileges and roles to different organizations by allowing the founders of the ordering service the ability to establish the initial rules and membership of the consortium. They also allow the organizations that join the consortium to create private application channels, govern their own business logic, and restrict access to the data that is put on the network.

The system channel configuration and a portion of each application channel configuration provides the ordering organizations control over which organizations are members of the consortium, how blocks are delivered to channels, and the consensus mechanism used by the nodes of the ordering service.

The system channel configuration provides members of the consortium the ability to create channels. Application channels and ACLs are the mechanism that consortium organizations use to add or remove members from a channel and restrict access to data and smart contracts on a channel.

4.6.5 How do you write a policy in Fabric

If you want to change anything in Fabric, the policy associated with the resource describes **who** needs to approve it, either with an explicit sign off from individuals, or an implicit sign off by a group. In the insurance domain, an explicit sign off could be a single member of the homeowners insurance agents group. And an implicit sign off would be analogous to requiring approval from a majority of the managerial members of the homeowners insurance group. This is particularly useful because the members of that group can change over time without requiring that the policy be updated. In Hyperledger Fabric, explicit sign offs in policies are expressed using the `Signature` syntax and implicit sign offs use the `ImplicitMeta` syntax.

Signature policies

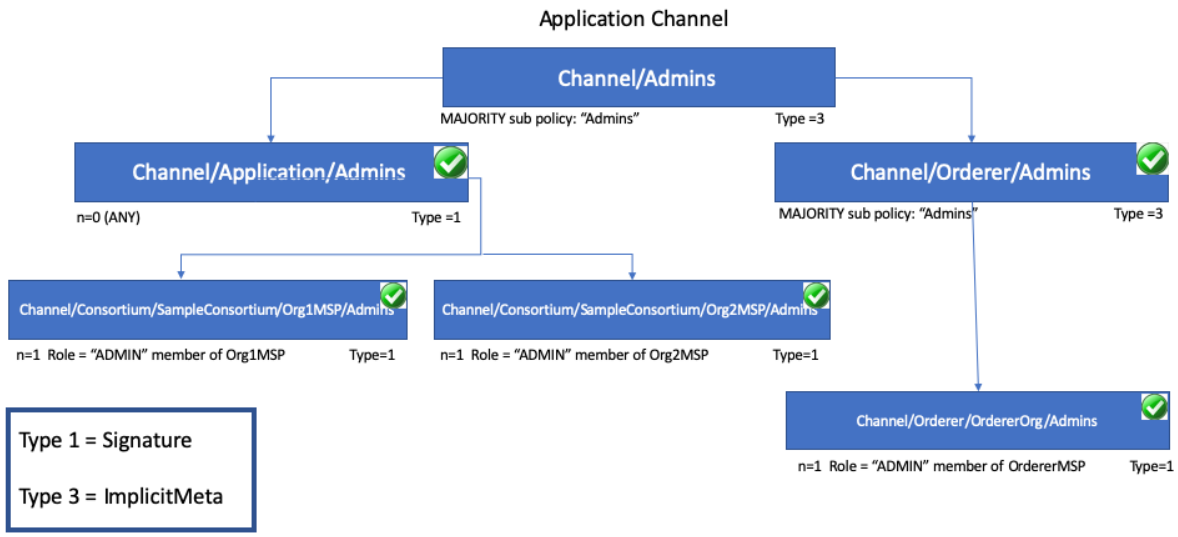
`Signature` policies define specific types of users who must sign in order for a policy to be satisfied such as `Org1.Peer OR Org2.Peer`. These policies are considered the most versatile because they allow for the construction of extremely specific rules like: "An admin of org A and 2 other admins, or 5 of 6 organization admins". The syntax supports arbitrary combinations of `AND`, `OR` and `NOutOf`. For example, a policy can be easily expressed by using

`AND (Org1, Org2)` which means that a signature from at least one member in Org1 AND one member in Org2 is required for the policy to be satisfied.

ImplicitMeta policies

`ImplicitMeta` policies are only valid in the context of channel configuration which is based on a tiered hierarchy of policies in a configuration tree. ImplicitMeta policies aggregate the result of policies deeper in the configuration tree that are ultimately defined by Signature policies. They are `Implicit` because they are constructed implicitly based on the current organizations in the channel configuration, and they are `Meta` because their evaluation is not against specific MSP principals, but rather against other sub-policies below them in the configuration tree.

The following diagram illustrates the tiered policy structure for an application channel and shows how the `ImplicitMeta` channel configuration admins policy, named `/Channel/Admins`, is resolved when the sub-policies named `Admins` below it in the configuration hierarchy are satisfied where each check mark represents that the conditions of the sub-policy were satisfied.



In order for the Channel/Admins policy to be satisfied, every sub-policy under it in the configuration hierarchy must be satisfied.

As you can see in the diagram above, `ImplicitMeta` policies, Type = 3, use a different syntax, `"<ANY|ALL|MAJORITY> <SubPolicyName>"`, for example:

```
`MAJORITY sub policy: Admins`
```

The diagram shows a sub-policy `Admins`, which refers to all the `Admins` policy below it in the configuration tree. You can create your own sub-policies and name them whatever you want and then define them in each of your organizations.

As mentioned above, a key benefit of an `ImplicitMeta` policy such as `MAJORITY Admins` is that when you add a new admin organization to the channel, you do not have to update the channel policy. Therefore `ImplicitMeta` policies are considered to be more flexible as the consortium members change. The consortium on the orderer can change as new members are added or an existing member leaves with the consortium members agreeing to the changes, but no policy updates are required. Recall that `ImplicitMeta` policies ultimately resolve the `Signature` sub-policies underneath them in the configuration tree as the diagram shows.

You can also define an application level implicit policy to operate across organizations, in a channel for example, and either require that ANY of them are satisfied, that ALL are satisfied, or that a MAJORITY are satisfied. This format lends itself to much better, more natural defaults, so that each organization can decide what it means for a valid endorsement.

Further granularity and control can be achieved if you include `NodeOUs` in your organization definition. Organization Units (OUs) are defined in the Fabric CA client configuration file and can be associated with an identity when it is created. In Fabric, `NodeOUs` provide a way to classify identities in a digital certificate hierarchy. For instance, an organization having specific `NodeOUs` enabled could require that a 'peer' sign for it to be a valid endorsement, whereas an organization without any might simply require that any member can sign.

4.6.6 An example: channel configuration policy

Understanding policies begins with examining the `configtx.yaml` where the channel policies are defined. We can use the `configtx.yaml` file in the Fabric test network to see examples of both policy syntax types. We are going to examine the configtx.yaml file used by the fabric-samples/test-network sample.

The first section of the file defines the organizations of the network. Inside each organization definition are the default policies for that organization, `Readers`, `Writers`, `Admins`, and `Endorsement`, although you can name your policies anything you want. Each policy has a `Type` which describes how the policy is expressed (`Signature` or `ImplicitMeta`) and a `Rule`.

The test network example below shows the Org1 organization definition in the system channel, where the policy `Type` is `Signature` and the endorsement policy rule is defined as `"OR('Org1MSP.peer')"`. This policy specifies that a peer that is a member of `Org1MSP` is required to sign. It is these signature policies that become the sub-policies that the ImplicitMeta policies point to.

Click here to see an example of an organization defined with signature policies

```
- &Org1
        # DefaultOrg defines the organization which is used in the sampleconfig
        # of the fabric.git development environment
        Name: Org1MSP

        # ID to load the MSP definition as
        ID: Org1MSP

        MSPDir: crypto-config/peerOrganizations/org1.example.com/msp

        # Policies defines the set of policies at this level of the config tree
        # For organization policies, their canonical path is usually
        #   /Channel/<Application|Orderer>/<OrgName>/<PolicyName>
        Policies:
            Readers:
                Type: Signature
                Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.client')"
            Writers:
                Type: Signature
                Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
            Admins:
                Type: Signature
                Rule: "OR('Org1MSP.admin')"
            Endorsement:
                Type: Signature
                Rule: "OR('Org1MSP.peer')"
```

The next example shows the `ImplicitMeta` policy type used in the `Application` section of the `configtx.yaml`. These set of policies lie on the `/Channel/Application/` path. If you use the default set of Fabric

ACLs, these policies define the behavior of many important features of application channels, such as who can query the channel ledger, invoke a chaincode, or update a channel config. These policies point to the sub-policies defined for each organization. The Org1 defined in the section above contains `Reader`, `Writer`, and `Admin` sub-policies that are evaluated by the `Reader`, `Writer`, and `Admin` `ImplicitMeta` policies in the `Application` section. Because the test network is built with the default policies, you can use the example Org1 to query the channel ledger, invoke a chaincode, and approve channel updates for any test network channel that you create.

Click here to see an example of ImplicitMeta policies

```
################################################################################
#
#   SECTION: Application
#
#   - This section defines the values to encode into a config transaction or
#   genesis block for application related parameters
#
################################################################################
Application: &ApplicationDefaults

    # Organizations is the list of orgs which are defined as participants on
    # the application side of the network
    Organizations:

    # Policies defines the set of policies at this level of the config tree
    # For Application policies, their canonical path is
    #   /Channel/Application/<PolicyName>
    Policies:
        Readers:
            Type: ImplicitMeta
            Rule: "ANY Readers"
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        Admins:
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"
        LifecycleEndorsement:
            Type: ImplicitMeta
            Rule: "MAJORITY Endorsement"
        Endorsement:
            Type: ImplicitMeta
            Rule: "MAJORITY Endorsement"
```

4.6.7 Fabric chaincode lifecycle

In the Fabric 2.0 release, a new chaincode lifecycle process was introduced, whereby a more democratic process is used to govern chaincode on the network. The new process allows multiple organizations to vote on how a chaincode will be operated before it can be used on a channel. This is significant because it is the combination of this new lifecycle process and the policies that are specified during that process that dictate the security across the network. More details on the flow are available in the Fabric chaincode lifecyle concept topic, but for purposes of this topic you should understand how policies are used in this flow. The new flow includes two steps where policies are specified: when chaincode is **approved** by organization members, and when it is **committed** to the channel.

The `Application` section of the `configtx.yaml` file includes the default chaincode lifecycle endorsement policy. In a production environment you would customize this definition for your own use case.

```
################################################################################
#
#   SECTION: Application
#
#   - This section defines the values to encode into a config transaction or
#   genesis block for application related parameters
#
################################################################################
Application: &ApplicationDefaults

    # Organizations is the list of orgs which are defined as participants on
    # the application side of the network
    Organizations:

    # Policies defines the set of policies at this level of the config tree
    # For Application policies, their canonical path is
    #   /Channel/Application/<PolicyName>
    Policies:
        Readers:
            Type: ImplicitMeta
            Rule: "ANY Readers"
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        Admins:
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"
        LifecycleEndorsement:
            Type: ImplicitMeta
            Rule: "MAJORITY Endorsement"
        Endorsement:
            Type: ImplicitMeta
            Rule: "MAJORITY Endorsement"
```

- The `LifecycleEndorsement` policy governs who needs to *approve a chaincode definition*.
- The `Endorsement` policy is the *default endorsement policy for a chaincode*. More on this below.

4.6.8 Chaincode endorsement policies

The endorsement policy is specified for a **chaincode** when it is approved and committed to the channel using the Fabric chaincode lifecycle (that is, one endorsement policy covers all of the state associated with a chaincode). The endorsement policy can be specified either by reference to an endorsement policy defined in the channel configuration or by explicitly specifying a Signature policy.

If an endorsement policy is not explicitly specified during the approval step, the default `Endorsement` policy `"MAJORITY Endorsement"` is used which means that a majority of the peers belonging to the different channel members (organizations) need to execute and validate a transaction against the chaincode in order for the transaction to be considered valid. This default policy allows organizations that join the channel to become automatically added to the chaincode endorsement policy. If you don't want to use the default endorsement policy, use the Signature policy format to specify a more complex endorsement policy (such as requiring that a chaincode be endorsed by one organization, and then one of the other organizations on the channel).

Signature policies also allow you to include `principals` which are simply a way of matching an identity to a role. Principals are just like user IDs or group IDs, but they are more versatile because they can include a wide range of properties of an actor's identity, such as the actor's organization, organizational unit, role or even the actor's specific identity. When we talk about principals, they are the properties which determine their permissions. Principals are

described as 'MSP.ROLE', where `MSP` represents the required MSP ID (the organization), and `ROLE` represents one of the four accepted roles: Member, Admin, Client, and Peer. A role is associated to an identity when a user enrolls with a CA. You can customize the list of roles available on your Fabric CA.

Some examples of valid principals are:

- 'Org0.Admin': an administrator of the Org0 MSP
- 'Org1.Member': a member of the Org1 MSP
- 'Org1.Client': a client of the Org1 MSP
- 'Org1.Peer': a peer of the Org1 MSP
- 'OrdererOrg.Orderer': an orderer in the OrdererOrg MSP

There are cases where it may be necessary for a particular state (a particular key-value pair, in other words) to have a different endorsement policy. This **state-based endorsement** allows the default chaincode-level endorsement policies to be overridden by a different policy for the specified keys.

For a deeper dive on how to write an endorsement policy refer to the topic on Endorsement policies in the Operations Guide.

Note: Policies work differently depending on which version of Fabric you are using:

- In Fabric releases prior to 2.0, chaincode endorsement policies can be updated during chaincode instantiation or by using the chaincode lifecycle commands. If not specified at instantiation time, the endorsement policy defaults to "any member of the organizations in the channel". For example, a channel with "Org1" and "Org2" would have a default endorsement policy of "OR('Org1.member', 'Org2.member')".
- Starting with Fabric 2.0, Fabric introduced a new chaincode lifecycle process that allows multiple organizations to agree on how a chaincode will be operated before it can be used on a channel. The new process requires that organizations agree to the parameters that define a chaincode, such as name, version, and the chaincode endorsement policy.

4.6.9 Overriding policy definitions

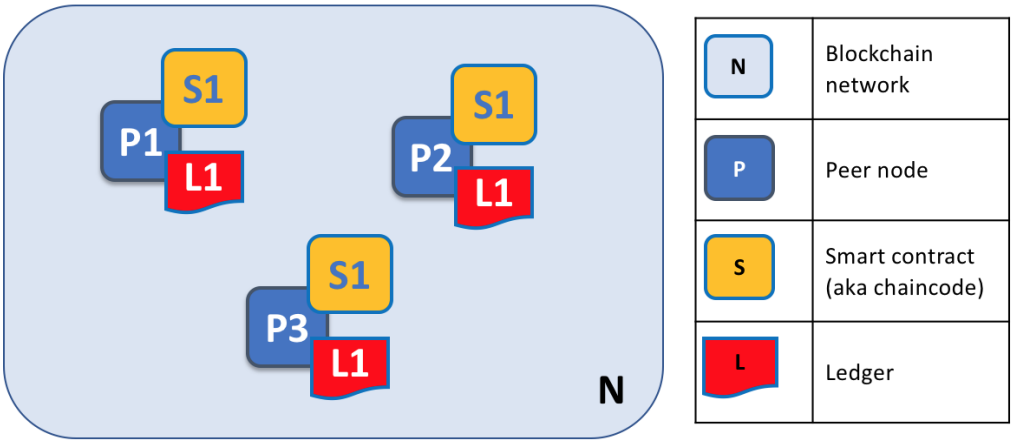
Hyperledger Fabric includes default policies which are useful for getting started, developing, and testing your blockchain, but they are meant to be customized in a production environment. You should be aware of the default policies in the `configtx.yaml` file. Channel configuration policies can be extended with arbitrary verbs, beyond the default `Readers, Writers, Admins` in `configtx.yaml`. The orderer system and application channels are overridden by issuing a config update when you override the default policies by editing the `configtx.yaml` for the orderer system channel or the `configtx.yaml` for a specific channel.

See the topic on Updating a channel configuration for more information.

4.7 Peers

A blockchain network is comprised primarily of a set of *peer nodes* (or, simply, *peers*). Peers are a fundamental element of the network because they host ledgers and smart contracts. Recall that a ledger immutably records all the transactions generated by smart contracts (which in Hyperledger Fabric are contained in a *chaincode*, more on this later). Smart contracts and ledgers are used to encapsulate the shared *processes* and shared *information* in a network, respectively. These aspects of a peer make them a good starting point to understand a Fabric network.

Other elements of the blockchain network are of course important: ledgers and smart contracts, orderers, policies, channels, applications, organizations, identities, and membership, and you can read more about them in their own dedicated sections. This section focusses on peers, and their relationship to those other elements in a Fabric network.



A blockchain network is comprised of peer nodes, each of which can hold copies of ledgers and copies of smart contracts. In this example, the network N consists of peers P1, P2 and P3, each of which maintain their own instance of the distributed ledger L1. P1, P2 and P3 use the same chaincode, S1, to access their copy of that distributed ledger.

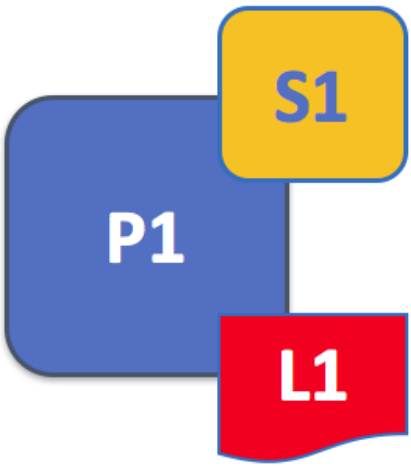
Peers can be created, started, stopped, reconfigured, and even deleted. They expose a set of APIs that enable administrators and applications to interact with the services that they provide. We'll learn more about these services in this section.

4.7.1 A word on terminology

Fabric implements **smart contracts** with a technology concept it calls **chaincode** — simply a piece of code that accesses the ledger, written in one of the supported programming languages. In this topic, we'll usually use the term **chaincode**, but feel free to read it as **smart contract** if you're more used to that term. It's the same thing! If you want to learn more about chaincode and smart contracts, check out our documentation on smart contracts and chaincode.

4.7.2 Ledgers and Chaincode

Let's look at a peer in a little more detail. We can see that it's the peer that hosts both the ledger and chaincode. More accurately, the peer actually hosts *instances* of the ledger, and *instances* of chaincode. Note that this provides a deliberate redundancy in a Fabric network — it avoids single points of failure. We'll learn more about the distributed and decentralized nature of a blockchain network later in this section.

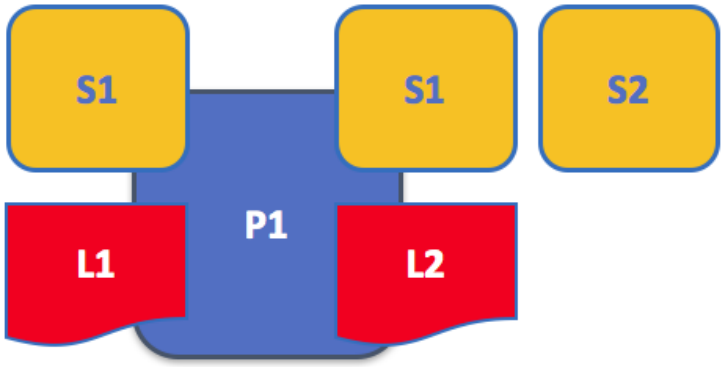


A peer hosts instances of ledgers and instances of chaincodes. In this example, P1 hosts an instance of ledger L1 and an instance of chaincode S1. There can be many ledgers and chaincodes hosted on an individual peer.

Because a peer is a *host* for ledgers and chaincodes, applications and administrators must interact with a peer if they want to access these resources. That's why peers are considered the most fundamental building blocks of a Fabric network. When a peer is first created, it has neither ledgers nor chaincodes. We'll see later how ledgers get created, and how chaincodes get installed, on peers.

Multiple Ledgers

A peer is able to host more than one ledger, which is helpful because it allows for a flexible system design. The simplest configuration is for a peer to manage a single ledger, but it's absolutely appropriate for a peer to host two or more ledgers when required.

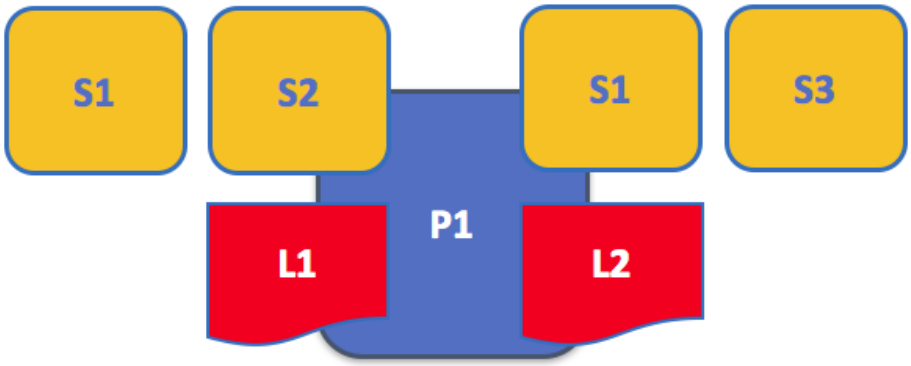


A peer hosting multiple ledgers. Peers host one or more ledgers, and each ledger has zero or more chaincodes that apply to them. In this example, we can see that the peer P1 hosts ledgers L1 and L2. Ledger L1 is accessed using chaincode S1. Ledger L2 on the other hand can be accessed using chaincodes S1 and S2.

Although it is perfectly possible for a peer to host a ledger instance without hosting any chaincodes which access that ledger, it's rare that peers are configured this way. The vast majority of peers will have at least one chaincode installed on it which can query or update the peer's ledger instances. It's worth mentioning in passing that, whether or not users have installed chaincodes for use by external applications, peers also have special **system chaincodes** that are always present. These are not discussed in detail in this topic.

Multiple Chaincodes

There isn't a fixed relationship between the number of ledgers a peer has and the number of chaincodes that can access that ledger. A peer might have many chaincodes and many ledgers available to it.



An example of a peer hosting multiple chaincodes. Each ledger can have many chaincodes which access it. In this example, we can see that peer P1 hosts ledgers L1 and L2, where L1 is accessed by chaincodes S1 and S2, and L2 is accessed by S1 and S3. We can see that S1 can access both L1 and L2.

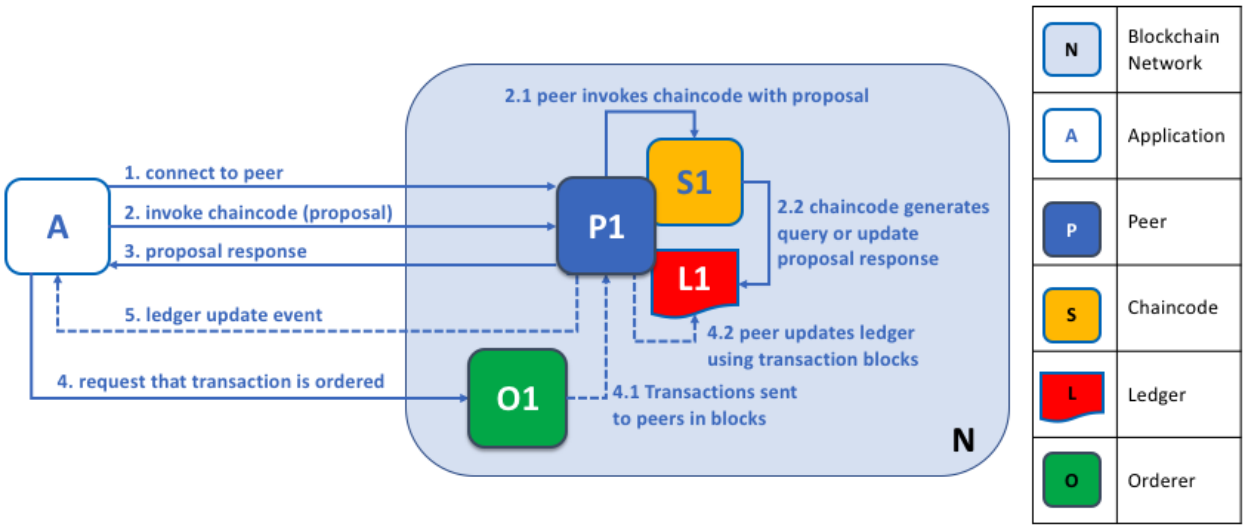
We'll see a little later why the concept of **channels** in Fabric is important when hosting multiple ledgers or multiple chaincodes on a peer.

4.7.3 Applications and Peers

We're now going to show how applications interact with peers to access the ledger. Ledger-query interactions involve a simple three-step dialogue between an application and a peer; ledger-update interactions are a little more involved, and require two extra steps. We've simplified these steps a little to help you get started with Fabric, but don't worry — what's most important to understand is the difference in application-peer interactions for ledger-query compared to ledger-update transaction styles.

Applications always connect to peers when they need to access ledgers and chaincodes. The Fabric Software Development Kit (SDK) makes this easy for programmers — its APIs enable applications to connect to peers, invoke chaincodes to generate transactions, submit transactions to the network that will get ordered, validated and committed to the distributed ledger, and receive events when this process is complete.

Through a peer connection, applications can execute chaincodes to query or update a ledger. The result of a ledger query transaction is returned immediately, whereas ledger updates involve a more complex interaction between applications, peers and orderers. Let's investigate this in a little more detail.



Peers, in conjunction with orderers, ensure that the ledger is kept up-to-date on every peer. In this example, application A connects to P1 and invokes chaincode S1 to query or update the ledger L1. P1 invokes S1 to generate a proposal response that contains a query result or a proposed ledger update. Application A receives the proposal response and, for queries, the process is now complete. For updates, A builds a transaction from all of the responses, which it sends to O1 for ordering. O1 collects transactions from across the network into blocks, and distributes these to all peers, including P1. P1 validates the transaction before committing to L1. Once L1 is updated, P1 generates an event, received by A, to signify completion.

A peer can return the results of a query to an application immediately since all of the information required to satisfy the query is in the peer's local copy of the ledger. Peers never consult with other peers in order to respond to a query from an application. Applications can, however, connect to one or more peers to issue a query; for example, to corroborate a result between multiple peers, or retrieve a more up-to-date result from a different peer if there's a suspicion that information might be out of date. In the diagram, you can see that ledger query is a simple three-step process.

An update transaction starts in the same way as a query transaction, but has two extra steps. Although ledger-updating applications also connect to peers to invoke a chaincode, unlike with ledger-querying applications, an individual peer cannot perform a ledger update at this time, because other peers must first agree to the change — a process called **consensus**. Therefore, peers return to the application a **proposed** update — one that this peer would apply subject to other peers' prior agreement. The first extra step — step four — requires that applications send an appropriate set of matching proposed updates to the entire network of peers as a transaction for commitment to their respective ledgers. This is achieved by the application by using an **orderer** to package transactions into blocks, and distributing them to the entire network of peers, where they can be verified before being applied to each peer's local copy of the ledger. As this whole ordering processing takes some time to complete (seconds), the application is notified asynchronously, as shown in step five.

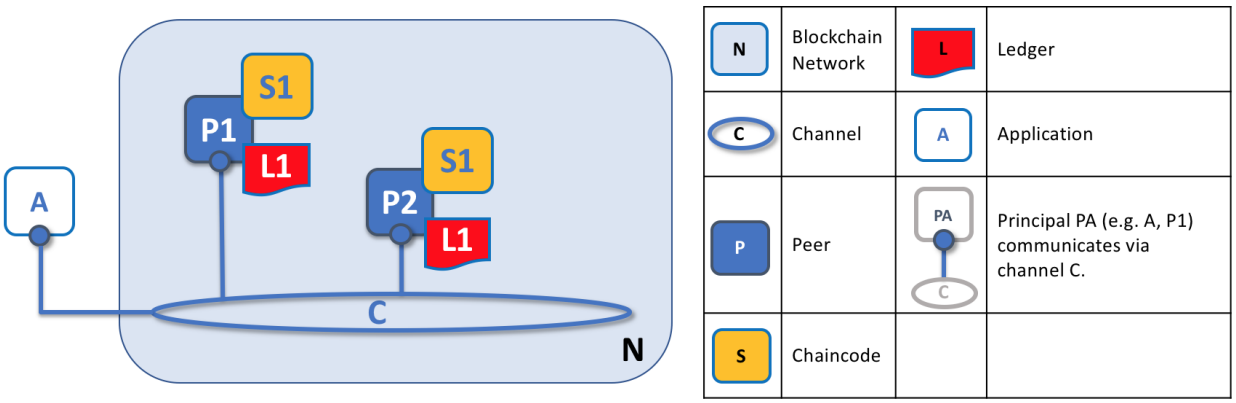
Later in this section, you'll learn more about the detailed nature of this ordering process — and for a really detailed look at this process see the Transaction Flow topic.

4.7.4 Peers and Channels

Although this section is about peers rather than channels, it's worth spending a little time understanding how peers interact with each other, and with applications, via *channels* — a mechanism by which a set of components within a blockchain network can communicate and transact *privately*.

These components are typically peer nodes, orderer nodes and applications and, by joining a channel, they agree to collaborate to collectively share and manage identical copies of the ledger associated with that channel. Conceptually, you can think of channels as being similar to groups of friends (though the members of a channel certainly don't need

to be friends!). A person might have several groups of friends, with each group having activities they do together. These groups might be totally separate (a group of work friends as compared to a group of hobby friends), or there can be some crossover between them. Nevertheless, each group is its own entity, with "rules" of a kind.



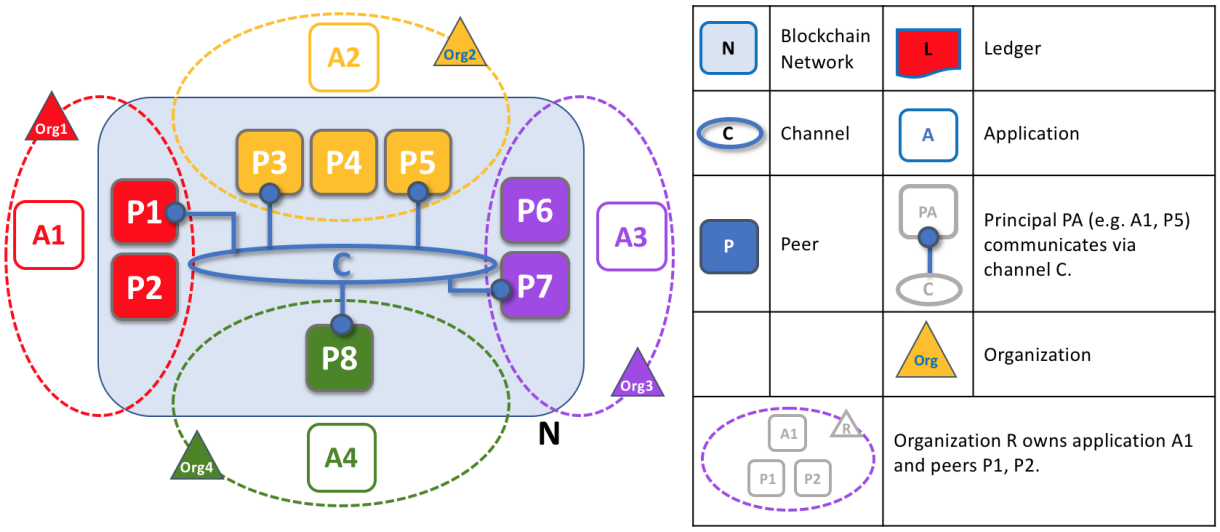
Channels allow a specific set of peers and applications to communicate with each other within a blockchain network. In this example, application A can communicate directly with peers P1 and P2 using channel C. You can think of the channel as a pathway for communications between particular applications and peers. (For simplicity, orderers are not shown in this diagram, but must be present in a functioning network.)

We see that channels don't exist in the same way that peers do — it's more appropriate to think of a channel as a logical structure that is formed by a collection of physical peers. *It is vital to understand this point — peers provide the control point for access to, and management of, channels.*

4.7.5 Peers and Organizations

Now that you understand peers and their relationship to ledgers, chaincodes and channels, you'll be able to see how multiple organizations come together to form a blockchain network.

Blockchain networks are administered by a collection of organizations rather than a single organization. Peers are central to how this kind of distributed network is built because they are owned by — and are the connection points to the network for — these organizations.



Peers in a blockchain network with multiple organizations. The blockchain network is built up from the peers owned and contributed by the different organizations. In this example, we see four organizations contributing eight peers to form a network. The channel C connects five of these peers in the network N — P1, P3, P5, P7 and P8. The other peers owned by these organizations have not been joined to this channel, but are typically joined to at least one other channel. Applications that have been developed by a particular organization will connect to their own organization's peers as well as those of different organizations. Again, for simplicity, an orderer node is not shown in this diagram.

It's really important that you can see what's happening in the formation of a blockchain network. *The network is both formed and managed by the multiple organizations who contribute resources to it.* Peers are the resources that we're discussing in this topic, but the resources an organization provides are more than just peers. There's a principle at work here — the network literally does not exist without organizations contributing their individual resources to the collective network. Moreover, the network grows and shrinks with the resources that are provided by these collaborating organizations.

You can see that (other than the ordering service) there are no centralized resources — in the *example above*, the network, **N**, would not exist if the organizations did not contribute their peers. This reflects the fact that the network does not exist in any meaningful sense unless and until organizations contribute the resources that form it. Moreover, the network does not depend on any individual organization — it will continue to exist as long as one organization remains, no matter which other organizations may come and go. This is at the heart of what it means for a network to be decentralized.

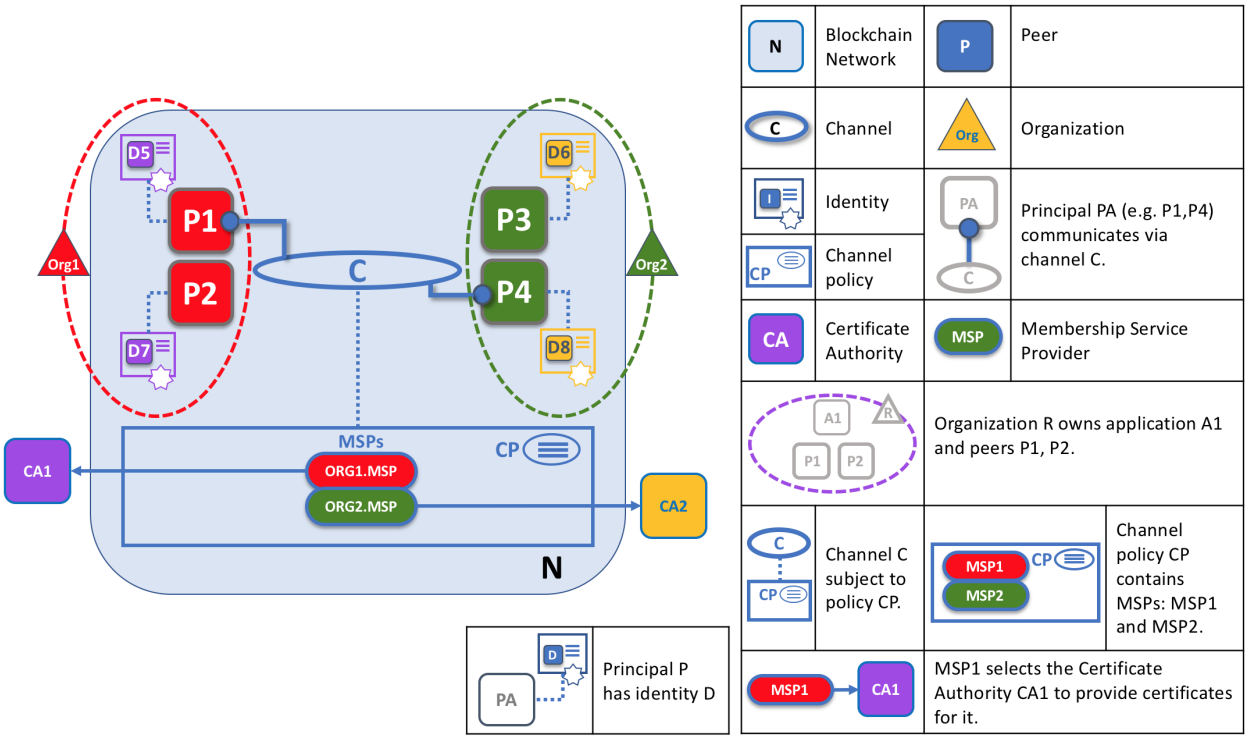
Applications in different organizations, as in the *example above*, may or may not be the same. That's because it's entirely up to an organization as to how its applications process their peers' copies of the ledger. This means that both application and presentation logic may vary from organization to organization even though their respective peers host exactly the same ledger data.

Applications connect either to peers in their organization, or peers in another organization, depending on the nature of the ledger interaction that's required. For ledger-query interactions, applications typically connect to their own organization's peers. For ledger-update interactions, we'll see later why applications need to connect to peers representing *every* organization that is required to endorse the ledger update.

4.7.6 Peers and Identity

Now that you've seen how peers from different organizations come together to form a blockchain network, it's worth spending a few moments understanding how peers get assigned to organizations by their administrators.

Peers have an identity assigned to them via a digital certificate from a particular certificate authority. You can read lots more about how X.509 digital certificates work elsewhere in this guide but, for now, think of a digital certificate as being like an ID card that provides lots of verifiable information about a peer. *Each and every peer in the network is assigned a digital certificate by an administrator from its owning organization*.



When a peer connects to a channel, its digital certificate identifies its owning organization via a channel MSP. In this example, P1 and P2 have identities issued by CA1. Channel C determines from a policy in its channel configuration that identities from CA1 should be associated with Org1 using ORG1.MSP. Similarly, P3 and P4 are identified by ORG2.MSP as being part of Org2.

Whenever a peer connects using a channel to a blockchain network, *a policy in the channel configuration uses the peer's identity to determine its rights.* The mapping of identity to organization is provided by a component called a *Membership Service Provider* (MSP) — it determines how a peer gets assigned to a specific role in a particular organization and accordingly gains appropriate access to blockchain resources. Moreover, a peer can be owned only by a single organization, and is therefore associated with a single MSP. We'll learn more about peer access control later in this section, and there's an entire section on MSPs and access control policies elsewhere in this guide. But for now, think of an MSP as providing linkage between an individual identity and a particular organizational role in a blockchain network.

To digress for a moment, peers as well as *everything that interacts with a blockchain network acquire their organizational identity from their digital certificate and an MSP*. Peers, applications, end users, administrators and orderers must have an identity and an associated MSP if they want to interact with a blockchain network. *We give a name to every entity that interacts with a blockchain network using an identity — a principal.* You can learn lots more about principals and organizations elsewhere in this guide, but for now you know more than enough to continue your understanding of peers!

Finally, note that it's not really important where the peer is physically located — it could reside in the cloud, or in a data centre owned by one of the organizations, or on a local machine — it's the digital certificate associated with it that identifies it as being owned by a particular organization. In our example above, P3 could be hosted in Org1's data center, but as long as the digital certificate associated with it is issued by CA2, then it's owned by Org2.

4.7.7 Peers and Orderers

We've seen that peers form the basis for a blockchain network, hosting ledgers and smart contracts which can be queried and updated by peer-connected applications. However, the mechanism by which applications and peers interact

with each other to ensure that every peer's ledger is kept consistent with each other is mediated by special nodes called *orderers*, and it's to these nodes we now turn our attention.

An update transaction is quite different from a query transaction because a single peer cannot, on its own, update the ledger — updating requires the consent of other peers in the network. A peer requires other peers in the network to approve a ledger update before it can be applied to a peer's local ledger. This process is called *consensus*, which takes much longer to complete than a simple query. But when all the peers required to approve the transaction do so, and the transaction is committed to the ledger, peers will notify their connected applications that the ledger has been updated. You're about to be shown a lot more detail about how peers and orderers manage the consensus process in this section.

Specifically, applications that want to update the ledger are involved in a 3-phase process, which ensures that all the peers in a blockchain network keep their ledgers consistent with each other.

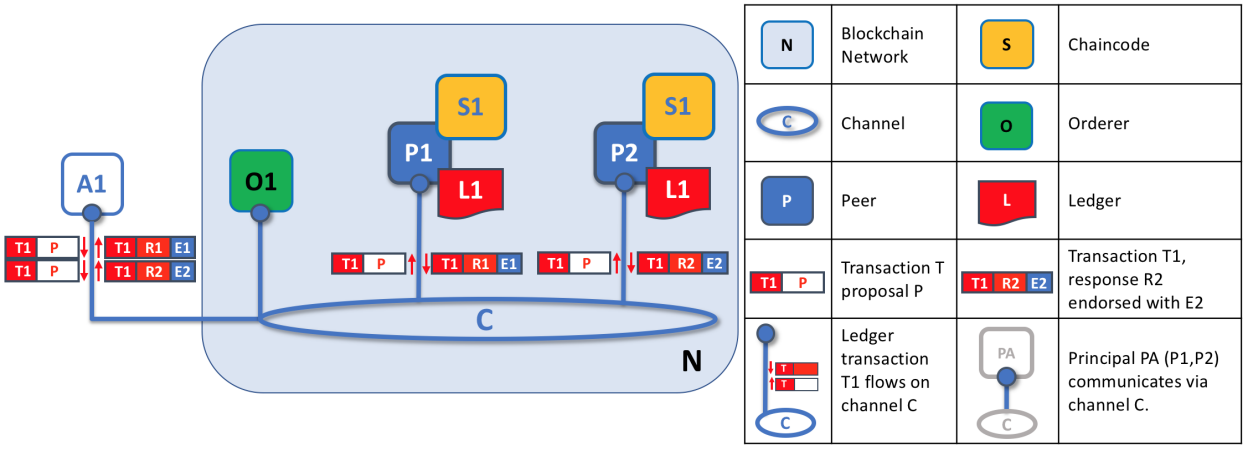
- In the first phase, applications work with a subset of *endorsing peers*, each of which provide an endorsement of the proposed ledger update to the application, but do not apply the proposed update to their copy of the ledger.
- In the second phase, these separate endorsements are collected together as transactions and packaged into blocks.
- In the third and final phase, these blocks are distributed back to every peer where each transaction is validated before being committed to that peer's copy of the ledger.

As you will see, orderer nodes are central to this process, so let's investigate in a little more detail how applications and peers use orderers to generate ledger updates that can be consistently applied to a distributed, replicated ledger.

Phase 1: Proposal

Phase 1 of the transaction workflow involves an interaction between an application and a set of peers — it does not involve orderers. Phase 1 is only concerned with an application asking different organizations' endorsing peers to agree to the results of the proposed chaincode invocation.

To start phase 1, applications generate a transaction proposal which they send to each of the required set of peers for endorsement. Each of these *endorsing peers* then independently executes a chaincode using the transaction proposal to generate a transaction proposal response. It does not apply this update to the ledger, but rather simply signs it and returns it to the application. Once the application has received a sufficient number of signed proposal responses, the first phase of the transaction flow is complete. Let's examine this phase in a little more detail.



Transaction proposals are independently executed by peers who return endorsed proposal responses. In this example, application A1 generates transaction T1 proposal P which it sends to both peer P1 and peer P2 on channel C. P1 executes S1 using transaction T1 proposal P generating transaction T1 response R1 which it endorses with E1. Independently, P2 executes S1 using transaction T1 proposal P generating transaction T1 response R2 which it endorses with E2. Application A1 receives two endorsed responses for transaction T1, namely E1 and E2.

Initially, a set of peers are chosen by the application to generate a set of proposed ledger updates. Which peers are chosen by the application? Well, that depends on the *endorsement policy* (defined for a chaincode), which defines the set of organizations that need to endorse a proposed ledger change before it can be accepted by the network. This is literally what it means to achieve consensus — every organization who matters must have endorsed the proposed ledger change *before* it will be accepted onto any peer's ledger.

A peer endorses a proposal response by adding its digital signature, and signing the entire payload using its private key. This endorsement can be subsequently used to prove that this organization's peer generated a particular response. In our example, if peer P1 is owned by organization Org1, endorsement E1 corresponds to a digital proof that "Transaction T1 response R1 on ledger L1 has been provided by Org1's peer P1!".

Phase 1 ends when the application receives signed proposal responses from sufficient peers. We note that different peers can return different and therefore inconsistent transaction responses to the application *for the same transaction proposal*. It might simply be that the result was generated at different times on different peers with ledgers at different states, in which case an application can simply request a more up-to-date proposal response. Less likely, but much more seriously, results might be different because the chaincode is *non-deterministic*. Non-determinism is the enemy of chaincodes and ledgers and if it occurs it indicates a serious problem with the proposed transaction, as inconsistent results cannot, obviously, be applied to ledgers. An individual peer cannot know that their transaction result is non-deterministic — transaction responses must be gathered together for comparison before non-determinism can be detected. (Strictly speaking, even this is not enough, but we defer this discussion to the transaction section, where non-determinism is discussed in detail.)

At the end of phase 1, the application is free to discard inconsistent transaction responses if it wishes to do so, effectively terminating the transaction workflow early. We'll see later that if an application tries to use an inconsistent set of transaction responses to update the ledger, it will be rejected.

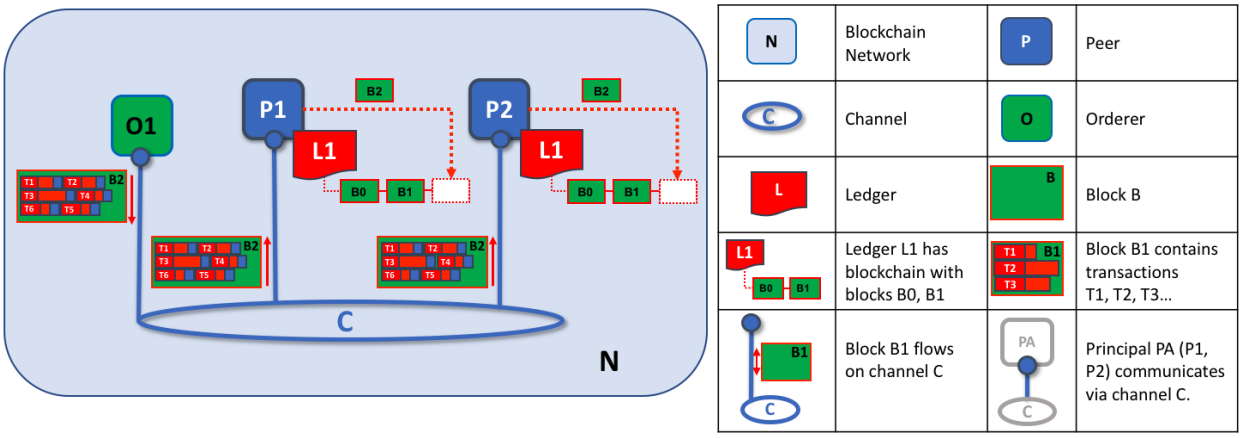
Phase 2: Ordering and packaging transactions into blocks

The second phase of the transaction workflow is the packaging phase. The orderer is pivotal to this process — it receives transactions containing endorsed transaction proposal responses from many applications, and orders the transactions into blocks. For more details about the ordering and packaging phase, check out our conceptual information about the ordering phase.

Phase 3: Validation and commit

At the end of phase 2, we see that orderers have been responsible for the simple but vital processes of collecting proposed transaction updates, ordering them, and packaging them into blocks, ready for distribution to the peers.

The final phase of the transaction workflow involves the distribution and subsequent validation of blocks from the orderer to the peers, where they can be committed to the ledger. Specifically, at each peer, every transaction within a block is validated to ensure that it has been consistently endorsed by all relevant organizations before it is committed to the ledger. Failed transactions are retained for audit, but are not committed to the ledger.



The second role of an orderer node is to distribute blocks to peers. In this example, orderer O1 distributes block B2 to peer P1 and peer P2. Peer P1 processes block B2, resulting in a new block being added to ledger L1 on P1. In parallel, peer P2 processes block B2, resulting in a new block being added to ledger L1 on P2. Once this process is complete, the ledger L1 has been consistently updated on peers P1 and P2, and each may inform connected applications that the transaction has been processed.

Phase 3 begins with the orderer distributing blocks to all peers connected to it. Peers are connected to orderers on channels such that when a new block is generated, all of the peers connected to the orderer will be sent a copy of the new block. Each peer will process this block independently, but in exactly the same way as every other peer on the channel. In this way, we'll see that the ledger can be kept consistent. It's also worth noting that not every peer needs to be connected to an orderer — peers can cascade blocks to other peers using the **gossip** protocol, who also can process them independently. But let's leave that discussion to another time!

Upon receipt of a block, a peer will process each transaction in the sequence in which it appears in the block. For every transaction, each peer will verify that the transaction has been endorsed by the required organizations according to the *endorsement policy* of the chaincode which generated the transaction. For example, some transactions may only need to be endorsed by a single organization, whereas others may require multiple endorsements before they are considered valid. This process of validation verifies that all relevant organizations have generated the same outcome or result. Also note that this validation is different than the endorsement check in phase 1, where it is the application that receives the response from endorsing peers and makes the decision to send the proposal transactions. In case the application violates the endorsement policy by sending wrong transactions, the peer is still able to reject the transaction in the validation process of phase 3.

If a transaction has been endorsed correctly, the peer will attempt to apply it to the ledger. To do this, a peer must perform a ledger consistency check to verify that the current state of the ledger is compatible with the state of the ledger when the proposed update was generated. This may not always be possible, even when the transaction has been fully endorsed. For example, another transaction may have updated the same asset in the ledger such that the transaction update is no longer valid and therefore can no longer be applied. In this way, the ledger is kept consistent across each peer in the channel because they each follow the same rules for validation.

After a peer has successfully validated each individual transaction, it updates the ledger. Failed transactions are not applied to the ledger, but they are retained for audit purposes, as are successful transactions. This means that peer blocks are almost exactly the same as the blocks received from the orderer, except for a valid or invalid indicator on each transaction in the block.

We also note that phase 3 does not require the running of chaincodes — this is done only during phase 1, and that's important. It means that chaincodes only have to be available on endorsing nodes, rather than throughout the blockchain network. This is often helpful as it keeps the logic of the chaincode confidential to endorsing organizations. This is in contrast to the output of the chaincodes (the transaction proposal responses) which are shared with every peer in the channel, whether or not they endorsed the transaction. This specialization of endorsing peers is designed to help scalability and confidentiality.

Finally, every time a block is committed to a peer's ledger, that peer generates an appropriate *event*. *Block events* include the full block content, while *block transaction events* include summary information only, such as whether each transaction in the block has been validated or invalidated. *Chaincode* events that the chaincode execution has produced can also be published at this time. Applications can register for these event types so that they can be notified when they occur. These notifications conclude the third and final phase of the transaction workflow.

In summary, phase 3 sees the blocks which are generated by the orderer consistently applied to the ledger. The strict ordering of transactions into blocks allows each peer to validate that transaction updates are consistently applied across the blockchain network.

Orderers and Consensus

This entire transaction workflow process is called *consensus* because all peers have reached agreement on the order and content of transactions, in a process that is mediated by orderers. Consensus is a multi-step process and applications are only notified of ledger updates when the process is complete — which may happen at slightly different times on different peers.

We will discuss orderers in a lot more detail in a future orderer topic, but for now, think of orderers as nodes which collect and distribute proposed ledger updates from applications for peers to validate and include on the ledger.

That's it! We've now finished our tour of peers and the other components that they relate to in Fabric. We've seen that peers are in many ways the most fundamental element — they form the network, host chaincodes and the ledger, handle transaction proposals and responses, and keep the ledger up-to-date by consistently applying transaction updates to it.

4.8 Ledger

Audience: Architects, Application and smart contract developers, administrators

A **ledger** is a key concept in Hyperledger Fabric; it stores important factual information about business objects; both the current value of the attributes of the objects, and the history of transactions that resulted in these current values.

In this topic, we're going to cover:

- *What is a Ledger?*
- *Storing facts about business objects*
- *A blockchain ledger*
- *The world state*
- *The blockchain data structure*
- *How blocks are stored in a blockchain*
- *Transactions*
- *World state database options*
- *The **Fabcar** example ledger*
- *Ledgers and namespaces*
- *Ledgers and channels*

4.8.1 What is a Ledger?

A ledger contains the current state of a business as a journal of transactions. The earliest European and Chinese ledgers date from almost 1000 years ago, and the Sumerians had stone ledgers 4000 years ago – but let's start with a more up-to-date example!

You're probably used to looking at your bank account. What's most important to you is the available balance – it's what you're able to spend at the current moment in time. If you want to see how your balance was derived, then you can look through the transaction credits and debits that determined it. This is a real life example of a ledger – a state (your bank balance), and a set of ordered transactions (credits and debits) that determine it. Hyperledger Fabric is motivated by these same two concerns – to present the current value of a set of ledger states, and to capture the history of the transactions that determined these states.

4.8.2 Ledgers, Facts, and States

A ledger doesn't literally store business objects – instead it stores **facts** about those objects. When we say "we store a business object in a ledger" what we really mean is that we're recording the facts about the current state of an object, and the facts about the history of transactions that led to the current state. In an increasingly digital world, it can feel like we're looking at an object, rather than facts about an object. In the case of a digital object, it's likely that it lives in an external datastore; the facts we store in the ledger allow us to identify its location along with other key information about it.

While the facts about the current state of a business object may change, the history of facts about it is **immutable**, it can be added to, but it cannot be retrospectively changed. We're going to see how thinking of a blockchain as an immutable history of facts about business objects is a simple yet powerful way to understand it.

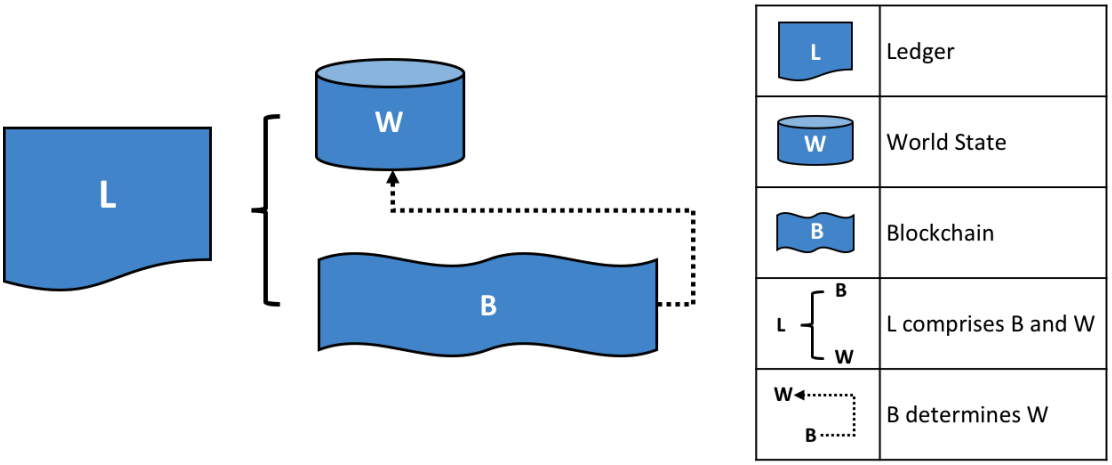
Let's now take a closer look at the Hyperledger Fabric ledger structure!

4.8.3 The Ledger

In Hyperledger Fabric, a ledger consists of two distinct, though related, parts – a world state and a blockchain. Each of these represents a set of facts about a set of business objects.

Firstly, there's a **world state** – a database that holds **current values** of a set of ledger states. The world state makes it easy for a program to directly access the current value of a state rather than having to calculate it by traversing the entire transaction log. Ledger states are, by default, expressed as **key-value** pairs, and we'll see later how Hyperledger Fabric provides flexibility in this regard. The world state can change frequently, as states can be created, updated and deleted.

Secondly, there's a **blockchain** – a transaction log that records all the changes that have resulted in the current the world state. Transactions are collected inside blocks that are appended to the blockchain – enabling you to understand the history of changes that have resulted in the current world state. The blockchain data structure is very different to the world state because once written, it cannot be modified; it is **immutable**.



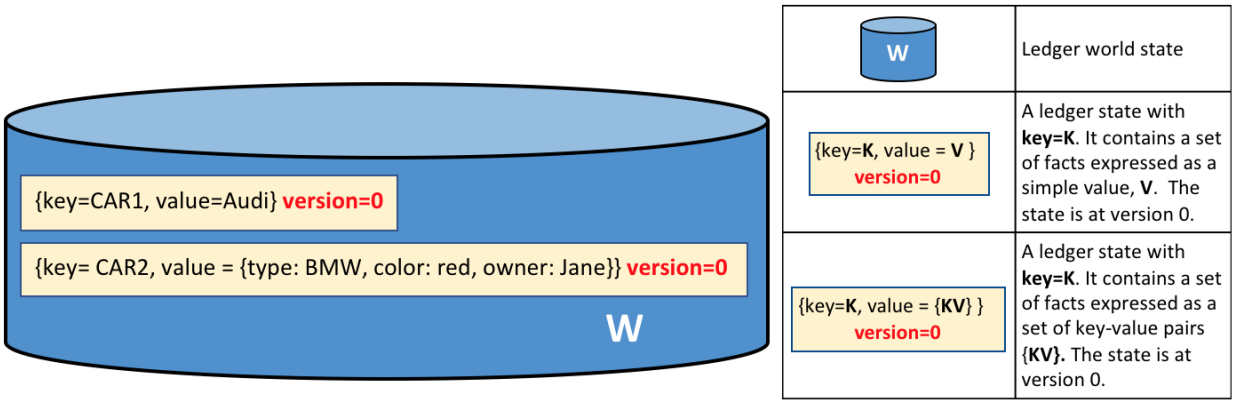
A Ledger L comprises blockchain B and world state W, where blockchain B determines world state W. We can also say that world state W is derived from blockchain B.

It's helpful to think of there being one **logical** ledger in a Hyperledger Fabric network. In reality, the network maintains multiple copies of a ledger – which are kept consistent with every other copy through a process called **consensus**. The term **Distributed Ledger Technology** (**DLT**) is often associated with this kind of ledger – one that is logically singular, but has many consistent copies distributed throughout a network.

Let's now examine the world state and blockchain data structures in more detail.

4.8.4 World State

The world state holds the current value of the attributes of a business object as a unique ledger state. That's useful because programs usually require the current value of an object; it would be cumbersome to traverse the entire blockchain to calculate an object's current value – you just get it directly from the world state.



A ledger world state containing two states. The first state is: key=CAR1 and value=Audi. The second state has a more complex value: key=CAR2 and value={model:BMW, color=red, owner=Jane}. Both states are at version 0.

A ledger state records a set of facts about a particular business object. Our example shows ledger states for two cars,

CAR1 and CAR2, each having a key and a value. An application program can invoke a smart contract which uses simple ledger APIs to **get**, **put** and **delete** states. Notice how a state value can be simple (Audi...) or compound (type:BMW...). The world state is often queried to retrieve objects with certain attributes, for example to find all red BMWs.

The world state is implemented as a database. This makes a lot of sense because a database provides a rich set of operators for the efficient storage and retrieval of states. We'll see later that Hyperledger Fabric can be configured to use different world state databases to address the needs of different types of state values and the access patterns required by applications, for example in complex queries.

Applications submit transactions which capture changes to the world state, and these transactions end up being committed to the ledger blockchain. Applications are insulated from the details of this consensus mechanism by the Hyperledger Fabric SDK; they merely invoke a smart contract, and are notified when the transaction has been included in the blockchain (whether valid or invalid). The key design point is that only transactions that are **signed** by the required set of **endorsing organizations** will result in an update to the world state. If a transaction is not signed by sufficient endorsers, it will not result in a change of world state. You can read more about how applications use smart contracts, and how to develop applications.

You'll also notice that a state has a version number, and in the diagram above, states CAR1 and CAR2 are at their starting versions, 0. The version number is for internal use by Hyperledger Fabric, and is incremented every time the state changes. The version is checked whenever the state is updated to make sure the current states matches the version at the time of endorsement. This ensures that the world state is changing as expected; that there has not been a concurrent update.

Finally, when a ledger is first created, the world state is empty. Because any transaction which represents a valid change to world state is recorded on the blockchain, it means that the world state can be re-generated from the blockchain at any time. This can be very convenient – for example, the world state is automatically generated when a peer is created. Moreover, if a peer fails abnormally, the world state can be regenerated on peer restart, before transactions are accepted.

4.8.5 Blockchain

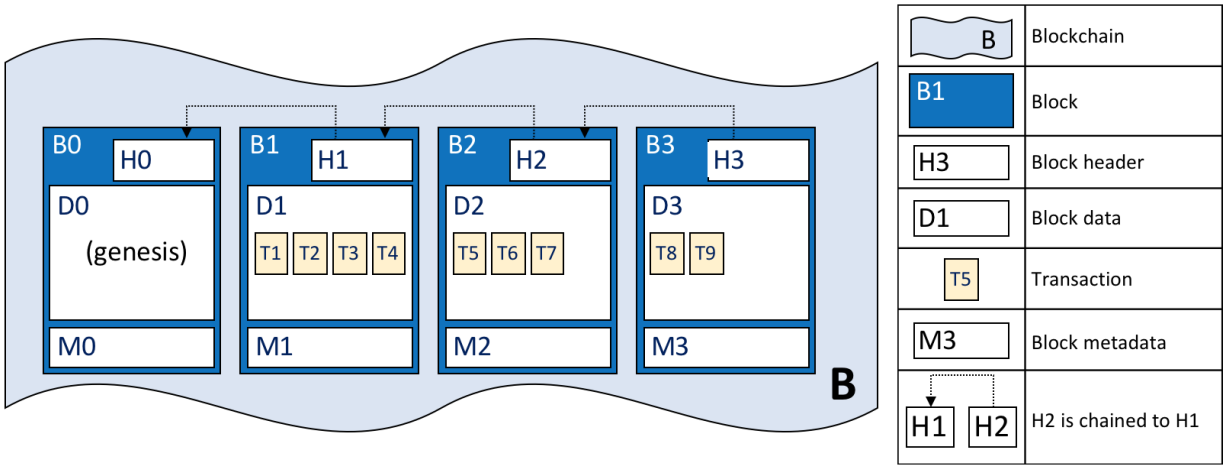
Let's now turn our attention from the world state to the blockchain. Whereas the world state contains a set of facts relating to the current state of a set of business objects, the blockchain is an historical record of the facts about how these objects arrived at their current states. The blockchain has recorded every previous version of each ledger state and how it has been changed.

The blockchain is structured as sequential log of interlinked blocks, where each block contains a sequence of transactions, each transaction representing a query or update to the world state. The exact mechanism by which transactions are ordered is discussed elsewhere; what's important is that block sequencing, as well as transaction sequencing within blocks, is established when blocks are first created by a Hyperledger Fabric component called the **ordering service**.

Each block's header includes a hash of the block's transactions, as well a hash of the prior block's header. In this way, all transactions on the ledger are sequenced and cryptographically linked together. This hashing and linking makes the ledger data very secure. Even if one node hosting the ledger was tampered with, it would not be able to convince all the other nodes that it has the 'correct' blockchain because the ledger is distributed throughout a network of independent nodes.

The blockchain is always implemented as a file, in contrast to the world state, which uses a database. This is a sensible design choice as the blockchain data structure is heavily biased towards a very small set of simple operations. Appending to the end of the blockchain is the primary operation, and query is currently a relatively infrequent operation.

Let's have a look at the structure of a blockchain in a little more detail.



A blockchain B containing blocks B0, B1, B2, B3. B0 is the first block in the blockchain, the genesis block.

In the above diagram, we can see that **block** B2 has a **block data** D2 which contains all its transactions: T5, T6, T7.

Most importantly, B2 has a **block header** H2, which contains a cryptographic **hash** of all the transactions in D2 as well as a hash of H1. In this way, blocks are inextricably and immutably linked to each other, which the term **blockchain** so neatly captures!

Finally, as you can see in the diagram, the first block in the blockchain is called the **genesis block**. It's the starting point for the ledger, though it does not contain any user transactions. Instead, it contains a configuration transaction containing the initial state of the network channel (not shown). We discuss the genesis block in more detail when we discuss the blockchain network and channels in the documentation.

4.8.6 Blocks

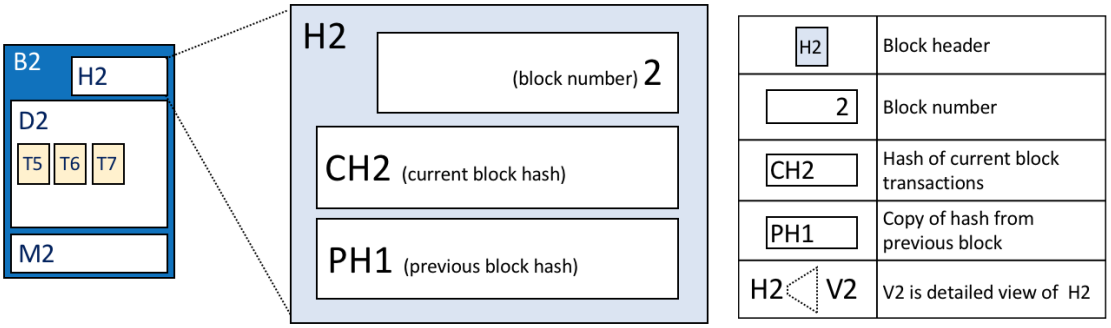
Let's have a closer look at the structure of a block. It consists of three sections

- **Block Header**

  This section comprises three fields, written when a block is created.

  - **Block number**: An integer starting at 0 (the genesis block), and increased by 1 for every new block appended to the blockchain.
  - **Current Block Hash**: The hash of all the transactions contained in the current block.
  - **Previous Block Header Hash**: The hash from the previous block header.

  These fields are internally derived by cryptographically hashing the block data. They ensure that each and every block is inextricably linked to its neighbour, leading to an immutable ledger.



Block header details. The header H2 of block B2 consists of block number 2, the hash CH2 of the current block data D2, and the hash of the prior block header H1.

- **Block Data**

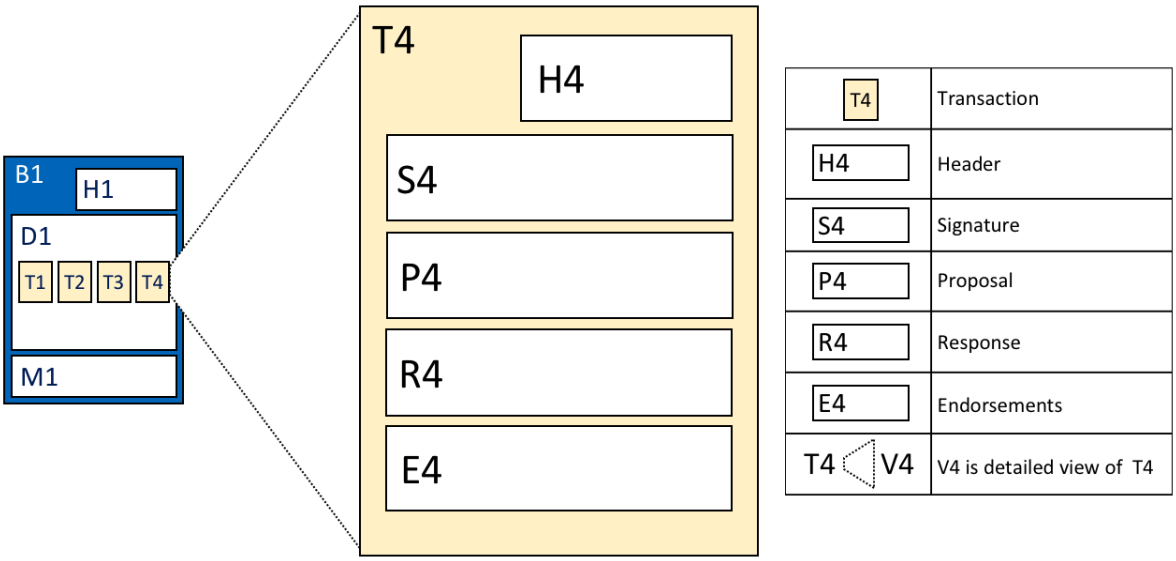
  This section contains a list of transactions arranged in order. It is written when the block is created by the ordering service. These transactions have a rich but straightforward structure, which we describe *later* in this topic.

- **Block Metadata**

  This section contains the certificate and signature of the block creator which is used to verify the block by network nodes. Subsequently, the block committer adds a valid/invalid indicator for every transaction into a bitmap that also resides in the block metadata, as well as a hash of the cumulative state updates up until and including that block, in order to detect a state fork. Unlike the block data and header fields, this section is not an input to the block hash computation.

4.8.7 Transactions

As we've seen, a transaction captures changes to the world state. Let's have a look at the detailed **blockdata** structure which contains the transactions in a block.



Transaction details. Transaction T4 in blockdata D1 of block B1 consists of transaction header, H4, a transaction signature, S4, a transaction proposal P4, a transaction response, R4, and a list of endorsements, E4.

In the above example, we can see the following fields:

- **Header**

  This section, illustrated by H4, captures some essential metadata about the transaction – for example, the name of the relevant chaincode, and its version.

- **Signature**

  This section, illustrated by S4, contains a cryptographic signature, created by the client application. This field is used to check that the transaction details have not been tampered with, as it requires the application's private key to generate it.

- **Proposal**

  This field, illustrated by P4, encodes the input parameters supplied by an application to the smart contract which creates the proposed ledger update. When the smart contract runs, this proposal provides a set of input parameters, which, in combination with the current world state, determines the new world state.

- **Response**

  This section, illustrated by R4, captures the before and after values of the world state, as a **Read Write set** (RW-set). It's the output of a smart contract, and if the transaction is successfully validated, it will be applied to the ledger to update the world state.

- **Endorsements**

  As shown in E4, this is a list of signed transaction responses from each required organization sufficient to satisfy the endorsement policy. You'll notice that, whereas only one transaction response is included in the transaction, there are multiple endorsements. That's because each endorsement effectively encodes its organization's particular transaction response – meaning that there's no need to include any transaction response that doesn't match sufficient endorsements as it will be rejected as invalid, and not update the world state.

That concludes the major fields of the transaction – there are others, but these are the essential ones that you need to understand to have a solid understanding of the ledger data structure.

4.8.8 World State database options

The world state is physically implemented as a database, to provide simple and efficient storage and retrieval of ledger states. As we've seen, ledger states can have simple or compound values, and to accommodate this, the world state database implementation can vary, allowing these values to be efficiently implemented. Options for the world state database currently include LevelDB and CouchDB.

LevelDB is the default and is particularly appropriate when ledger states are simple key-value pairs. A LevelDB database is co-located with the peer node – it is embedded within the same operating system process.

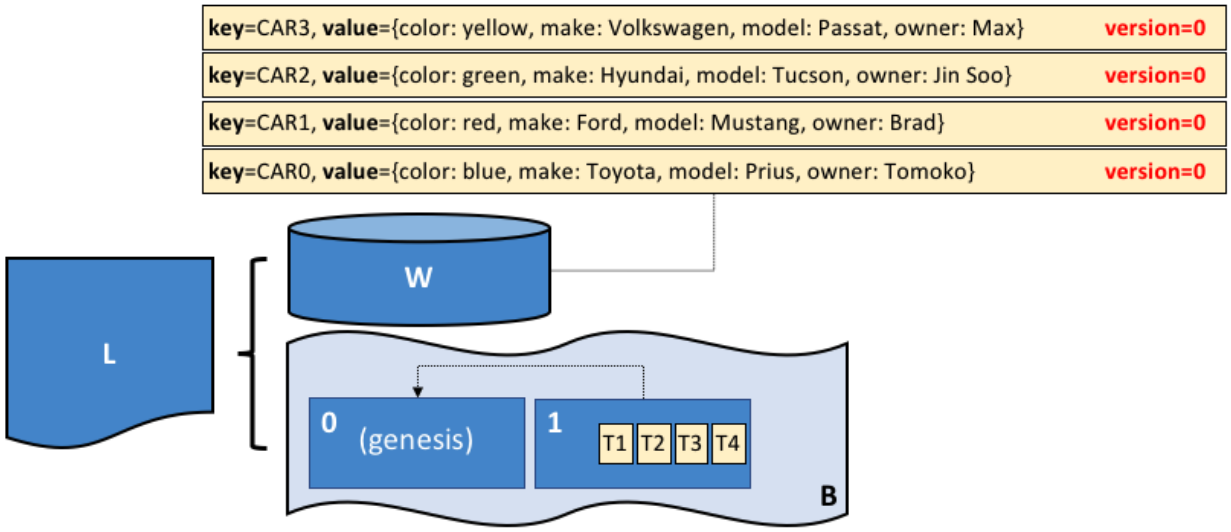
CouchDB is a particularly appropriate choice when ledger states are structured as JSON documents because CouchDB supports the rich queries and update of richer data types often found in business transactions. Implementation-wise, CouchDB runs in a separate operating system process, but there is still a 1:1 relation between a peer node and a CouchDB instance. All of this is invisible to a smart contract. See CouchDB as the StateDatabase for more information on CouchDB.

In LevelDB and CouchDB, we see an important aspect of Hyperledger Fabric – it is *pluggable*. The world state database could be a relational data store, or a graph store, or a temporal database. This provides great flexibility in the types of ledger states that can be efficiently accessed, allowing Hyperledger Fabric to address many different types of problems.

4.8.9 Example Ledger: fabcar

As we end this topic on the ledger, let's have a look at a sample ledger. If you've run the fabcar sample application, then you've created this ledger.

The fabcar sample app creates a set of 10 cars each with a unique identity; a different color, make, model and owner. Here's what the ledger looks like after the first four cars have been created.



The ledger, L, comprises a world state, W and a blockchain, B. W contains four states with keys: CAR0, CAR1, CAR2 and CAR3. B contains two blocks, 0 and 1. Block 1 contains four transactions: T1, T2, T3, T4.

We can see that the world state contains states that correspond to CAR0, CAR1, CAR2 and CAR3. CAR0 has a value which indicates that it is a blue Toyota Prius, currently owned by Tomoko, and we can see similar states and values for the other cars. Moreover, we can see that all car states are at version number 0, indicating that this is their starting version number – they have not been updated since they were created.

We can also see that the blockchain contains two blocks. Block 0 is the genesis block, though it does not contain any transactions that relate to cars. Block 1 however, contains transactions T1, T2, T3, T4 and these correspond to

transactions that created the initial states for CAR0 to CAR3 in the world state. We can see that block 1 is linked to block 0.

We have not shown the other fields in the blocks or transactions, specifically headers and hashes. If you're interested in the precise details of these, you will find a dedicated reference topic elsewhere in the documentation. It gives you a fully worked example of an entire block with its transactions in glorious detail – but for now, you have achieved a solid conceptual understanding of a Hyperledger Fabric ledger. Well done!

4.8.10 Namespaces

Even though we have presented the ledger as though it were a single world state and single blockchain, that's a little bit of an over-simplification. In reality, each chaincode has its own world state that is separate from all other chaincodes. World states are in a namespace so that only smart contracts within the same chaincode can access a given namespace.

A blockchain is not namespaced. It contains transactions from many different smart contract namespaces. You can read more about chaincode namespaces in this topic.

Let's now look at how the concept of a namespace is applied within a Hyperledger Fabric channel.

4.8.11 Channels

In Hyperledger Fabric, each channel has a completely separate ledger. This means a completely separate blockchain, and completely separate world states, including namespaces. It is possible for applications and smart contracts to communicate between channels so that ledger information can be accessed between them.

You can read more about how ledgers work with channels in this topic.

4.8.12 More information

See the Transaction Flow, Read-Write set semantics and CouchDB as the StateDatabase topics for a deeper dive on transaction flow, concurrency control, and the world state database.

4.9 The Ordering Service

Audience: Architects, ordering service admins, channel creators

This topic serves as a conceptual introduction to the concept of ordering, how orderers interact with peers, the role they play in a transaction flow, and an overview of the currently available implementations of the ordering service, with a particular focus on the recommended **Raft** ordering service implementation.

4.9.1 What is ordering?

Many distributed blockchains, such as Ethereum and Bitcoin, are not permissioned, which means that any node can participate in the consensus process, wherein transactions are ordered and bundled into blocks. Because of this fact, these systems rely on **probabilistic** consensus algorithms which eventually guarantee ledger consistency to a high degree of probability, but which are still vulnerable to divergent ledgers (also known as a ledger "fork"), where different participants in the network have a different view of the accepted order of transactions.

Hyperledger Fabric works differently. It features a node called an **orderer** (it's also known as an "ordering node") that does this transaction ordering, which along with other orderer nodes forms an **ordering service**. Because Fabric's design relies on **deterministic** consensus algorithms, any block validated by the peer is guaranteed to be final and correct. Ledgers cannot fork the way they do in many other distributed and permissionless blockchain networks.

In addition to promoting finality, separating the endorsement of chaincode execution (which happens at the peers) from ordering gives Fabric advantages in performance and scalability, eliminating bottlenecks which can occur when execution and ordering are performed by the same nodes.

4.9.2 Orderer nodes and channel configuration

In addition to their **ordering** role, orderers also maintain the list of organizations that are allowed to create channels. This list of organizations is known as the "consortium", and the list itself is kept in the configuration of the "orderer system channel" (also known as the "ordering system channel"). By default, this list, and the channel it lives on, can only be edited by the orderer admin. Note that it is possible for an ordering service to hold several of these lists, which makes the consortium a vehicle for Fabric multi-tenancy.

Orderers also enforce basic access control for channels, restricting who can read and write data to them, and who can configure them. Remember that who is authorized to modify a configuration element in a channel is subject to the policies that the relevant administrators set when they created the consortium or the channel. Configuration transactions are processed by the orderer, as it needs to know the current set of policies to execute its basic form of access control. In this case, the orderer processes the configuration update to make sure that the requestor has the proper administrative rights. If so, the orderer validates the update request against the existing configuration, generates a new configuration transaction, and packages it into a block that is relayed to all peers on the channel. The peers then processs the configuration transactions in order to verify that the modifications approved by the orderer do indeed satisfy the policies defined in the channel.

4.9.3 Orderer nodes and Identity

Everything that interacts with a blockchain network, including peers, applications, admins, and orderers, acquires their organizational identity from their digital certificate and their Membership Service Provider (MSP) definition.

For more information about identities and MSPs, check out our documentation on Identity and Membership.

Just like peers, ordering nodes belong to an organization. And similar to peers, a separate Certificate Authority (CA) should be used for each organization. Whether this CA will function as the root CA, or whether you choose to deploy a root CA and then intermediate CAs associated with that root CA, is up to you.

4.9.4 Orderers and the transaction flow

Phase one: Proposal

We've seen from our topic on Peers that they form the basis for a blockchain network, hosting ledgers, which can be queried and updated by applications through smart contracts.

Specifically, applications that want to update the ledger are involved in a process with three phases that ensures all of the peers in a blockchain network keep their ledgers consistent with each other.

In the first phase, a client application sends a transaction proposal to a subset of peers that will invoke a smart contract to produce a proposed ledger update and then endorse the results. The endorsing peers do not apply the proposed update to their copy of the ledger at this time. Instead, the endorsing peers return a proposal response to the client application. The endorsed transaction proposals will ultimately be ordered into blocks in phase two, and then distributed to all peers for final validation and commit in phase three.

For an in-depth look at the first phase, refer back to the Peers topic.

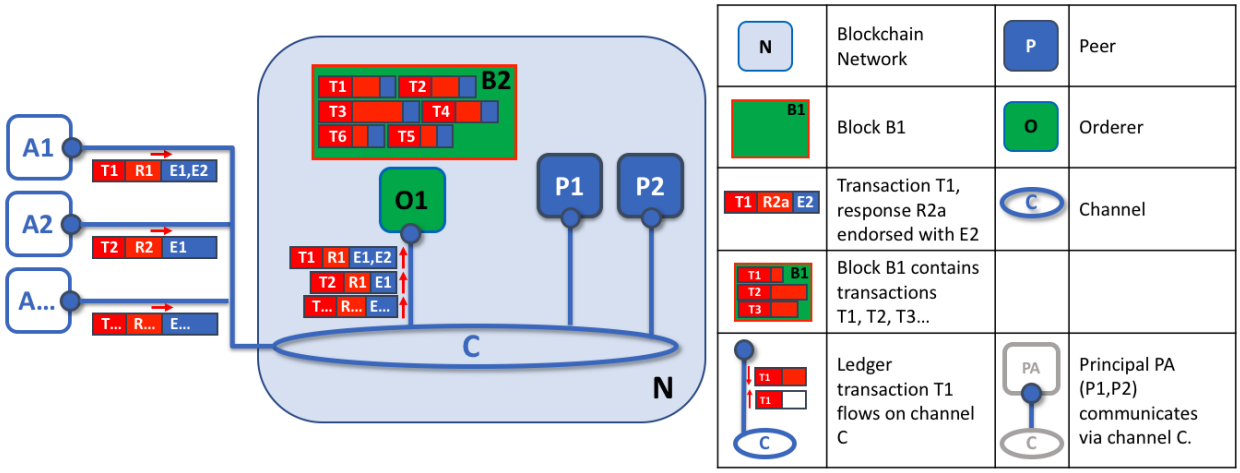
Phase two: Ordering and packaging transactions into blocks

After the completion of the first phase of a transaction, a client application has received an endorsed transaction proposal response from a set of peers. It's now time for the second phase of a transaction.

In this phase, application clients submit transactions containing endorsed transaction proposal responses to an ordering service node. The ordering service creates blocks of transactions which will ultimately be distributed to all peers on the channel for final validation and commit in phase three.

Ordering service nodes receive transactions from many different application clients concurrently. These ordering service nodes work together to collectively form the ordering service. Its job is to arrange batches of submitted transactions into a well-defined sequence and package them into *blocks*. These blocks will become the *blocks* of the blockchain!

The number of transactions in a block depends on channel configuration parameters related to the desired size and maximum elapsed duration for a block (`BatchSize` and `BatchTimeout` parameters, to be exact). The blocks are then saved to the orderer's ledger and distributed to all peers that have joined the channel. If a peer happens to be down at this time, or joins the channel later, it will receive the blocks after reconnecting to an ordering service node, or by gossiping with another peer. We'll see how this block is processed by peers in the third phase.



The first role of an ordering node is to package proposed ledger updates. In this example, application A1 sends a transaction T1 endorsed by E1 and E2 to the orderer O1. In parallel, Application A2 sends transaction T2 endorsed by E1 to the orderer O1. O1 packages transaction T1 from application A1 and transaction T2 from application A2 together with other transactions from other applications in the network into block B2. We can see that in B2, the transaction order is T1,T2,T3,T4,T6,T5 – which may not be the order in which these transactions arrived at the orderer! (This example shows a very simplified ordering service configuration with only one ordering node.)

It's worth noting that the sequencing of transactions in a block is not necessarily the same as the order received by the ordering service, since there can be multiple ordering service nodes that receive transactions at approximately the same time. What's important is that the ordering service puts the transactions into a strict order, and peers will use this order when validating and committing transactions.

This strict ordering of transactions within blocks makes Hyperledger Fabric a little different from other blockchains where the same transaction can be packaged into multiple different blocks that compete to form a chain. In Hyperledger Fabric, the blocks generated by the ordering service are **final**. Once a transaction has been written to a block, its position in the ledger is immutably assured. As we said earlier, Hyperledger Fabric's finality means that there are no **ledger forks** — validated transactions will never be reverted or dropped.

We can also see that, whereas peers execute smart contracts and process transactions, orderers most definitely do not. Every authorized transaction that arrives at an orderer is mechanically packaged in a block — the orderer makes no judgement as to the content of a transaction (except for channel configuration transactions, as mentioned earlier).

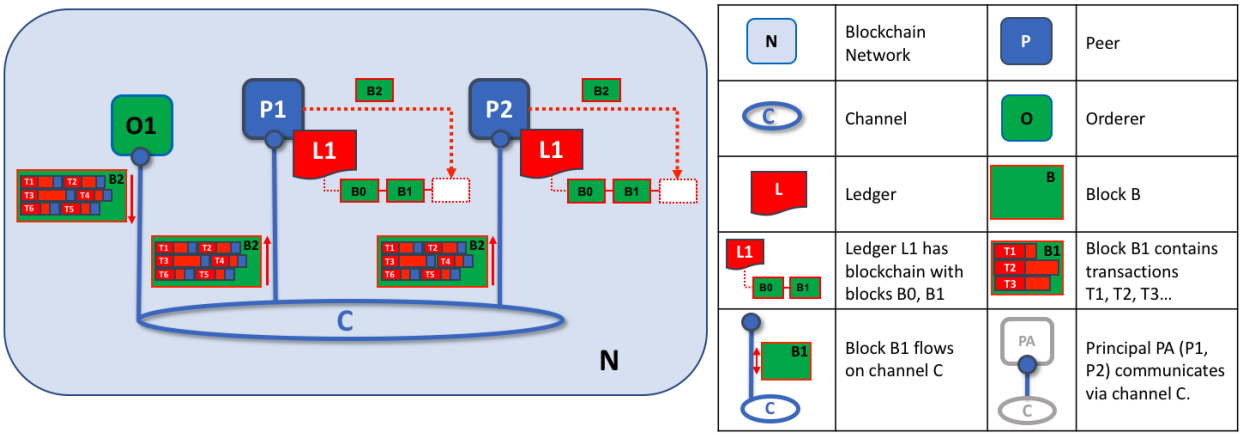
At the end of phase two, we see that orderers have been responsible for the simple but vital processes of collecting proposed transaction updates, ordering them, and packaging them into blocks, ready for distribution.

Phase three: Validation and commit

The third phase of the transaction workflow involves the distribution and subsequent validation of blocks from the orderer to the peers, where they can be committed to the ledger.

Phase 3 begins with the orderer distributing blocks to all peers connected to it. It's also worth noting that not every peer needs to be connected to an orderer — peers can cascade blocks to other peers using the **gossip** protocol.

Each peer will validate distributed blocks independently, but in a deterministic fashion, ensuring that ledgers remain consistent. Specifically, each peer in the channel will validate each transaction in the block to ensure it has been endorsed by the required organization's peers, that its endorsements match, and that it hasn't become invalidated by other recently committed transactions which may have been in-flight when the transaction was originally endorsed. Invalidated transactions are still retained in the immutable block created by the orderer, but they are marked as invalid by the peer and do not update the ledger's state.



The second role of an ordering node is to distribute blocks to peers. In this example, orderer O1 distributes block B2 to peer P1 and peer P2. Peer P1 processes block B2, resulting in a new block being added to ledger L1 on P1. In parallel, peer P2 processes block B2, resulting in a new block being added to ledger L1 on P2. Once this process is complete, the ledger L1 has been consistently updated on peers P1 and P2, and each may inform connected applications that the transaction has been processed.

In summary, phase three sees the blocks generated by the ordering service applied consistently to the ledger. The strict ordering of transactions into blocks allows each peer to validate that transaction updates are consistently applied across the blockchain network.

For a deeper look at phase 3, refer back to the Peers topic.

4.9.5 Ordering service implementations

While every ordering service currently available handles transactions and configuration updates the same way, there are nevertheless several different implementations for achieving consensus on the strict ordering of transactions between ordering service nodes.

For information about how to stand up an ordering node (regardless of the implementation the node will be used in), check out our documentation on standing up an ordering node.

- **Raft** (recommended)

New as of v1.4.1, Raft is a crash fault tolerant (CFT) ordering service based on an implementation of Raft protocol in `etcd`. Raft follows a "leader and follower" model, where a leader node is elected (per channel) and its decisions are replicated by the followers. Raft ordering services should be easier to set up and manage than Kafka-based ordering services, and their design allows different organizations to contribute nodes to a distributed ordering service.

- **Kafka** (deprecated in v2.0)

  Similar to Raft-based ordering, Apache Kafka is a CFT implementation that uses a "leader and follower" node configuration. Kafka utilizes a ZooKeeper ensemble for management purposes. The Kafka based ordering service has been available since Fabric v1.0, but many users may find the additional administrative overhead of managing a Kafka cluster intimidating or undesirable.

- **Solo** (deprecated in v2.0)

  The Solo implementation of the ordering service is intended for test only and consists only of a single ordering node. It has been deprecated and may be removed entirely in a future release. Existing users of Solo should move to a single node Raft network for equivalent function.

4.9.6 Raft

For information on how to configure a Raft ordering service, check out our documentation on configuring a Raft ordering service.

The go-to ordering service choice for production networks, the Fabric implementation of the established Raft protocol uses a "leader and follower" model, in which a leader is dynamically elected among the ordering nodes in a channel (this collection of nodes is known as the "consenter set"), and that leader replicates messages to the follower nodes. Because the system can sustain the loss of nodes, including leader nodes, as long as there is a majority of ordering nodes (what's known as a "quorum") remaining, Raft is said to be "crash fault tolerant" (CFT). In other words, if there are three nodes in a channel, it can withstand the loss of one node (leaving two remaining). If you have five nodes in a channel, you can lose two nodes (leaving three remaining nodes).

From the perspective of the service they provide to a network or a channel, Raft and the existing Kafka-based ordering service (which we'll talk about later) are similar. They're both CFT ordering services using the leader and follower design. If you are an application developer, smart contract developer, or peer administrator, you will not notice a functional difference between an ordering service based on Raft versus Kafka. However, there are a few major differences worth considering, especially if you intend to manage an ordering service:

- Raft is easier to set up. Although Kafka has many admirers, even those admirers will (usually) admit that deploying a Kafka cluster and its ZooKeeper ensemble can be tricky, requiring a high level of expertise in Kafka infrastructure and settings. Additionally, there are many more components to manage with Kafka than with Raft, which means that there are more places where things can go wrong. And Kafka has its own versions, which must be coordinated with your orderers. **With Raft, everything is embedded into your ordering node**.
- Kafka and Zookeeper are not designed to be run across large networks. While Kafka is CFT, it should be run in a tight group of hosts. This means that practically speaking you need to have one organization run the Kafka cluster. Given that, having ordering nodes run by different organizations when using Kafka (which Fabric supports) doesn't give you much in terms of decentralization because the nodes will all go to the same Kafka cluster which is under the control of a single organization. With Raft, each organization can have its own ordering nodes, participating in the ordering service, which leads to a more decentralized system.
- Raft is supported natively, which means that users are required to get the requisite images and learn how to use Kafka and ZooKeeper on their own. Likewise, support for Kafka-related issues is handled through Apache, the open-source developer of Kafka, not Hyperledger Fabric. The Fabric Raft implementation, on the other hand, has been developed and will be supported within the Fabric developer community and its support apparatus.
- Where Kafka uses a pool of servers (called "Kafka brokers") and the admin of the orderer organization specifies how many nodes they want to use on a particular channel, Raft allows the users to specify which ordering nodes

will be deployed to which channel. In this way, peer organizations can make sure that, if they also own an orderer, this node will be made a part of a ordering service of that channel, rather than trusting and depending on a central admin to manage the Kafka nodes.

- Raft is the first step toward Fabric's development of a byzantine fault tolerant (BFT) ordering service. As we'll see, some decisions in the development of Raft were driven by this. If you are interested in BFT, learning how to use Raft should ease the transition.

For all of these reasons, support for Kafka-based ordering service is being deprecated in Fabric v2.0.

Note: Similar to Solo and Kafka, a Raft ordering service can lose transactions after acknowledgement of receipt has been sent to a client. For example, if the leader crashes at approximately the same time as a follower provides acknowledgement of receipt. Therefore, application clients should listen on peers for transaction commit events regardless (to check for transaction validity), but extra care should be taken to ensure that the client also gracefully tolerates a timeout in which the transaction does not get committed in a configured timeframe. Depending on the application, it may be desirable to resubmit the transaction or collect a new set of endorsements upon such a timeout.

Raft concepts

While Raft offers many of the same features as Kafka — albeit in a simpler and easier-to-use package — it functions substantially different under the covers from Kafka and introduces a number of new concepts, or twists on existing concepts, to Fabric.

Log entry. The primary unit of work in a Raft ordering service is a "log entry", with the full sequence of such entries known as the "log". We consider the log consistent if a majority (a quorum, in other words) of members agree on the entries and their order, making the logs on the various orderers replicated.

Consenter set. The ordering nodes actively participating in the consensus mechanism for a given channel and receiving replicated logs for the channel. This can be all of the nodes available (either in a single cluster or in multiple clusters contributing to the system channel), or a subset of those nodes.

Finite-State Machine (FSM). Every ordering node in Raft has an FSM and collectively they're used to ensure that the sequence of logs in the various ordering nodes is deterministic (written in the same sequence).

Quorum. Describes the minimum number of consenters that need to affirm a proposal so that transactions can be ordered. For every consenter set, this is a **majority** of nodes. In a cluster with five nodes, three must be available for there to be a quorum. If a quorum of nodes is unavailable for any reason, the ordering service cluster becomes unavailable for both read and write operations on the channel, and no new logs can be committed.

Leader. This is not a new concept — Kafka also uses leaders, as we've said — but it's critical to understand that at any given time, a channel's consenter set elects a single node to be the leader (we'll describe how this happens in Raft later). The leader is responsible for ingesting new log entries, replicating them to follower ordering nodes, and managing when an entry is considered committed. This is not a special **type** of orderer. It is only a role that an orderer may have at certain times, and then not others, as circumstances determine.

Follower. Again, not a new concept, but what's critical to understand about followers is that the followers receive the logs from the leader and replicate them deterministically, ensuring that logs remain consistent. As we'll see in our section on leader election, the followers also receive "heartbeat" messages from the leader. In the event that the leader stops sending those message for a configurable amount of time, the followers will initiate a leader election and one of them will be elected the new leader.

Raft in a transaction flow

Every channel runs on a **separate** instance of the Raft protocol, which allows each instance to elect a different leader. This configuration also allows further decentralization of the service in use cases where clusters are made up of ordering nodes controlled by different organizations. While all Raft nodes must be part of the system channel, they do not necessarily have to be part of all application channels. Channel creators (and channel admins) have the ability to

pick a subset of the available orderers and to add or remove ordering nodes as needed (as long as only a single node is added or removed at a time).

While this configuration creates more overhead in the form of redundant heartbeat messages and goroutines, it lays necessary groundwork for BFT.

In Raft, transactions (in the form of proposals or configuration updates) are automatically routed by the ordering node that receives the transaction to the current leader of that channel. This means that peers and applications do not need to know who the leader node is at any particular time. Only the ordering nodes need to know.

When the orderer validation checks have been completed, the transactions are ordered, packaged into blocks, consented on, and distributed, as described in phase two of our transaction flow.

Architectural notes

How leader election works in Raft

Although the process of electing a leader happens within the orderer's internal processes, it's worth noting how the process works.

Raft nodes are always in one of three states: follower, candidate, or leader. All nodes initially start out as a **follower**. In this state, they can accept log entries from a leader (if one has been elected), or cast votes for leader. If no log entries or heartbeats are received for a set amount of time (for example, five seconds), nodes self-promote to the **candidate** state. In the candidate state, nodes request votes from other nodes. If a candidate receives a quorum of votes, then it is promoted to a **leader**. The leader must accept new log entries and replicate them to the followers.

For a visual representation of how the leader election process works, check out The Secret Lives of Data.

Snapshots

If an ordering node goes down, how does it get the logs it missed when it is restarted?

While it's possible to keep all logs indefinitely, in order to save disk space, Raft uses a process called "snapshotting", in which users can define how many bytes of data will be kept in the log. This amount of data will conform to a certain number of blocks (which depends on the amount of data in the blocks. Note that only full blocks are stored in a snapshot).

For example, let's say lagging replica `R1` was just reconnected to the network. Its latest block is `100`. Leader `L` is at block `196`, and is configured to snapshot at amount of data that in this case represents 20 blocks. `R1` would therefore receive block `180` from `L` and then make a `Deliver` request for blocks `101` to `180`. Blocks `180` to `196` would then be replicated to `R1` through the normal Raft protocol.

Kafka (deprecated in v2.0)

The other crash fault tolerant ordering service supported by Fabric is an adaptation of a Kafka distributed streaming platform for use as a cluster of ordering nodes. You can read more about Kafka at the Apache Kafka Web site, but at a high level, Kafka uses the same conceptual "leader and follower" configuration used by Raft, in which transactions (which Kafka calls "messages") are replicated from the leader node to the follower nodes. In the event the leader node goes down, one of the followers becomes the leader and ordering can continue, ensuring fault tolerance, just as with Raft.

The management of the Kafka cluster, including the coordination of tasks, cluster membership, access control, and controller election, among others, is handled by a ZooKeeper ensemble and its related APIs.

Kafka clusters and ZooKeeper ensembles are notoriously tricky to set up, so our documentation assumes a working knowledge of Kafka and ZooKeeper. If you decide to use Kafka without having this expertise, you should complete,

at a minimum, the first six steps of the Kafka Quickstart guide before experimenting with the Kafka-based ordering service. You can also consult this sample configuration file for a brief explanation of the sensible defaults for Kafka and ZooKeeper.

To learn how to bring up a a Kafka-based ordering service, check out our documentation on Kafka.

4.10 Smart Contracts and Chaincode

Audience: Architects, application and smart contract developers, administrators

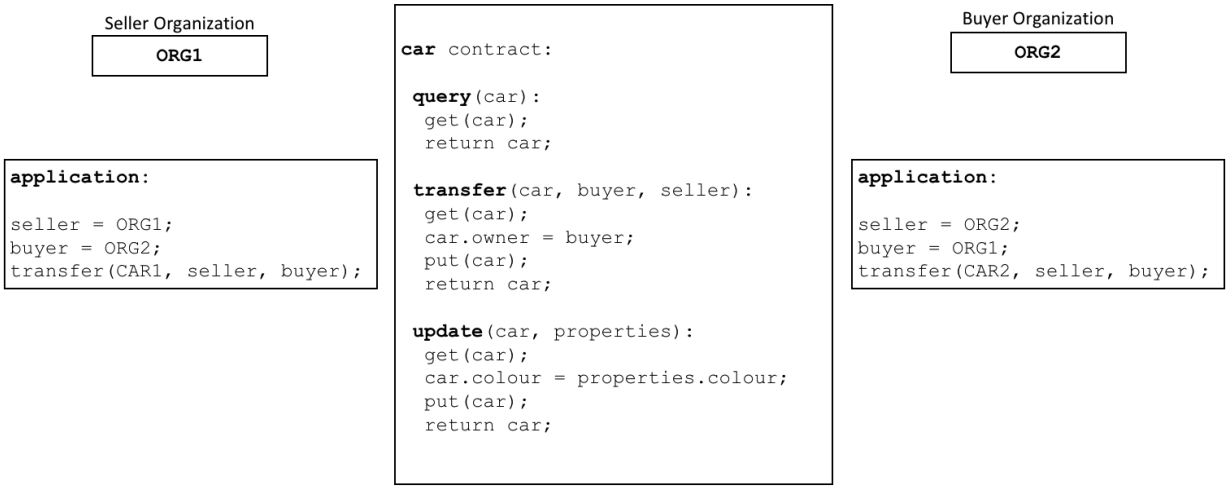
From an application developer's perspective, a **smart contract**, together with the ledger, form the heart of a Hyperledger Fabric blockchain system. Whereas a ledger holds facts about the current and historical state of a set of business objects, a **smart contract** defines the executable logic that generates new facts that are added to the ledger. A **chaincode** is typically used by administrators to group related smart contracts for deployment, but can also be used for low level system programming of Fabric. In this topic, we'll focus on why both **smart contracts** and **chaincode** exist, and how and when to use them.

In this topic, we'll cover:

- *What is a smart contract*
- *A note on terminology*
- *Smart contracts and the ledger*
- *How to develop a smart contract*
- *The importance of endorsement policies*
- *Valid transactions*
- *Channels and chaincode definitions*
- *Communicating between smart contracts*
- *What is system chaincode?*

4.10.1 Smart contract

Before businesses can transact with each other, they must define a common set of contracts covering common terms, data, rules, concept definitions, and processes. Taken together, these contracts lay out the **business model** that govern all of the interactions between transacting parties.



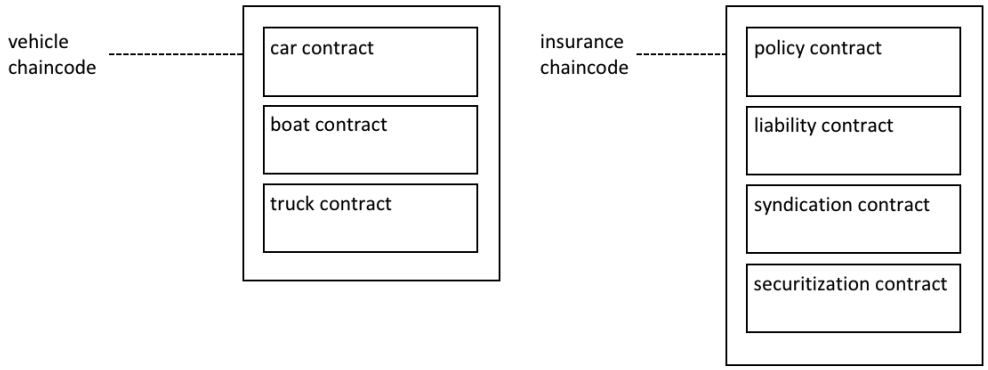
A smart contract defines the rules between different organizations in executable code. Applications invoke a smart contract to generate transactions that are recorded on the ledger.

Using a blockchain network, we can turn these contracts into executable programs – known in the industry as **smart contracts** – to open up a wide variety of new possibilities. That's because a smart contract can implement the governance rules for **any** type of business object, so that they can be automatically enforced when the smart contract is executed. For example, a smart contract might ensure that a new car delivery is made within a specified timeframe, or that funds are released according to prearranged terms, improving the flow of goods or capital respectively. Most importantly however, the execution of a smart contract is much more efficient than a manual human business process.

In the *diagram above*, we can see how two organizations, `ORG1` and `ORG2` have defined a `car` smart contract to `query`, `transfer` and `update` cars. Applications from these organizations invoke this smart contract to perform an agreed step in a business process, for example to transfer ownership of a specific car from `ORG1` to `ORG2`.

4.10.2 Terminology

Hyperledger Fabric users often use the terms **smart contract** and **chaincode** interchangeably. In general, a smart contract defines the **transaction logic** that controls the lifecycle of a business object contained in the world state. It is then packaged into a chaincode which is then deployed to a blockchain network. Think of smart contracts as governing transactions, whereas chaincode governs how smart contracts are packaged for deployment.



A smart contract is defined within a chaincode. Multiple smart contracts can be defined within the same chaincode. When a chaincode is deployed, all smart contracts within it are made available to applications.

In the diagram, we can see a `vehicle` chaincode that contains three smart contracts: `cars`, `boats` and `trucks`. We

can also see an `insurance` chaincode that contains four smart contracts: `policy`, `liability`, `syndication` and `securitization`. In both cases these contracts cover key aspects of the business process relating to vehicles and insurance. In this topic, we will use the `car` contract as an example. We can see that a smart contract is a domain specific program which relates to specific business processes, whereas a chaincode is a technical container of a group of related smart contracts.

4.10.3 Ledger

At the simplest level, a blockchain immutably records transactions which update states in a ledger. A smart contract programmatically accesses two distinct pieces of the ledger – a **blockchain**, which immutably records the history of all transactions, and a **world state** that holds a cache of the current value of these states, as it's the current value of an object that is usually required.

Smart contracts primarily **put**, **get** and **delete** states in the world state, and can also query the immutable blockchain record of transactions.

- A **get** typically represents a query to retrieve information about the current state of a business object.
- A **put** typically creates a new business object or modifies an existing one in the ledger world state.
- A **delete** typically represents the removal of a business object from the current state of the ledger, but not its history.

Smart contracts have many APIs available to them. Critically, in all cases, whether transactions create, read, update or delete business objects in the world state, the blockchain contains an immutable record of these changes.

4.10.4 Development

Smart contracts are the focus of application development, and as we've seen, one or more smart contracts can be defined within a single chaincode. Deploying a chaincode to a network makes all its smart contracts available to the organizations in that network. It means that only administrators need to worry about chaincode; everyone else can think in terms of smart contracts.

At the heart of a smart contract is a set of `transaction` definitions. For example, look at `fabcar.js`, where you can see a smart contract transaction that creates a new car:

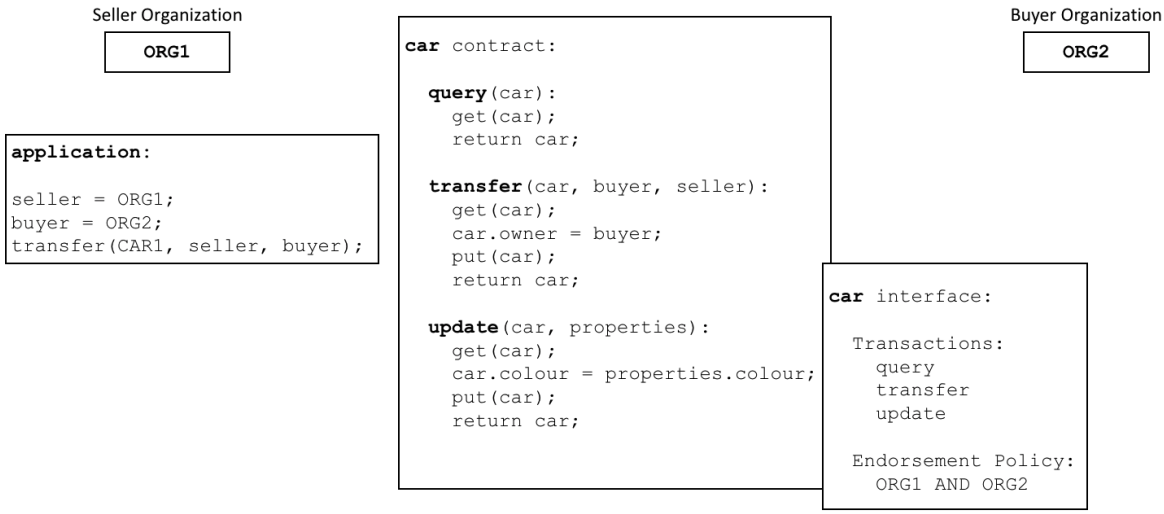
```
async createCar(ctx, carNumber, make, model, color, owner) {

    const car = {
        color,
        docType: 'car',
        make,
        model,
        owner,
    };

    await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
}
```

You can learn more about the **Fabcar** smart contract in the Writing your first application tutorial.

A smart contract can describe an almost infinite array of business use cases relating to immutability of data in multi-organizational decision making. The job of a smart contract developer is to take an existing business process that might govern financial prices or delivery conditions, and express it as a smart contract in a programming language such as JavaScript, Go, or Java. The legal and technical skills required to convert centuries of legal language into programming language is increasingly practiced by **smart contract auditors**. You can learn about how to design and develop a smart contract in the Developing applications topic.

4.10.5 Endorsement

Associated with every chaincode is an endorsement policy that applies to all of the smart contracts defined within it. An endorsement policy is very important; it indicates which organizations in a blockchain network must sign a transaction generated by a given smart contract in order for that transaction to be declared **valid**.



Every smart contract has an endorsement policy associated with it. This endorsement policy identifies which organizations must approve transactions generated by the smart contract before those transactions can be identified as valid.

An example endorsement policy might define that three of the four organizations participating in a blockchain network must sign a transaction before it is considered **valid**. All transactions, whether **valid** or **invalid** are added to a distributed ledger, but only **valid** transactions update the world state.

If an endorsement policy specifies that more than one organization must sign a transaction, then the smart contract must be executed by a sufficient set of organizations in order for a valid transaction to be generated. In the example *above*, a smart contract transaction to `transfer` a car would need to be executed and signed by both `ORG1` and `ORG2` for it to be valid.

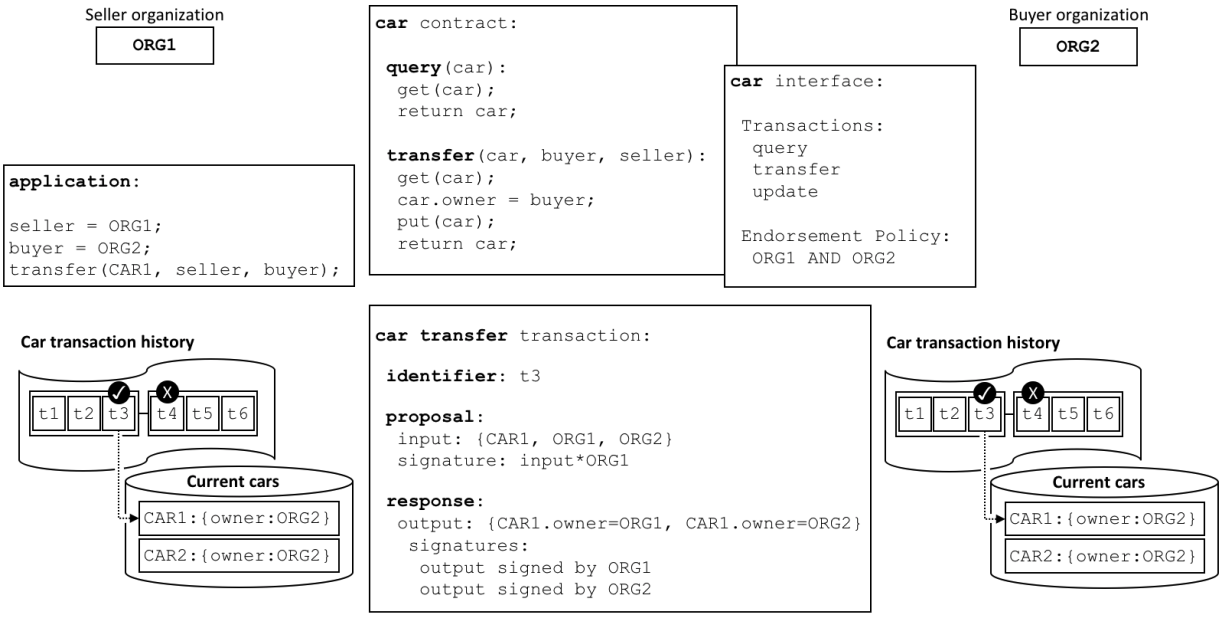
Endorsement policies are what make Hyperledger Fabric different to other blockchains like Ethereum or Bitcoin. In these systems valid transactions can be generated by any node in the network. Hyperledger Fabric more realistically models the real world; transactions must be validated by trusted organizations in a network. For example, a government organization must sign a valid `issueIdentity` transaction, or both the `buyer` and `seller` of a car must sign a `car` transfer transaction. Endorsement policies are designed to allow Hyperledger Fabric to better model these types of real-world interactions.

Finally, endorsement policies are just one example of policy in Hyperledger Fabric. Other policies can be defined to identify who can query or update the ledger, or add or remove participants from the network. In general, policies should be agreed in advance by the consortium of organizations in a blockchain network, although they are not set in stone. Indeed, policies themselves can define the rules by which they can be changed. And although an advanced topic, it is also possible to define custom endorsement policy rules over and above those provided by Fabric.

4.10.6 Valid transactions

When a smart contract executes, it runs on a peer node owned by an organization in the blockchain network. The contract takes a set of input parameters called the **transaction proposal** and uses them in combination with its program logic to read and write the ledger. Changes to the world state are captured as a **transaction proposal response** (or just **transaction response**) which contains a **read-write set** with both the states that have been read, and the new states

that are to be written if the transaction is valid. Notice that the world state **is not updated when the smart contract is executed**!



All transactions have an identifier, a proposal, and a response signed by a set of organizations. All transactions are recorded on the blockchain, whether valid or invalid, but only valid transactions contribute to the world state.

Examine the `car transfer` transaction. You can see a transaction `t3` for a car transfer between `ORG1` and `ORG2`. See how the transaction has input `{CAR1, ORG1, ORG2}` and output `{CAR1.owner=ORG1, CAR1.owner=ORG2}`, representing the change of owner from `ORG1` to `ORG2`. Notice how the input is signed by the application's organization `ORG1`, and the output is signed by *both* organizations identified by the endorsement policy, `ORG1` and `ORG2`. These signatures were generated by using each actor's private key, and mean that anyone in the network can verify that all actors in the network are in agreement about the transaction details.

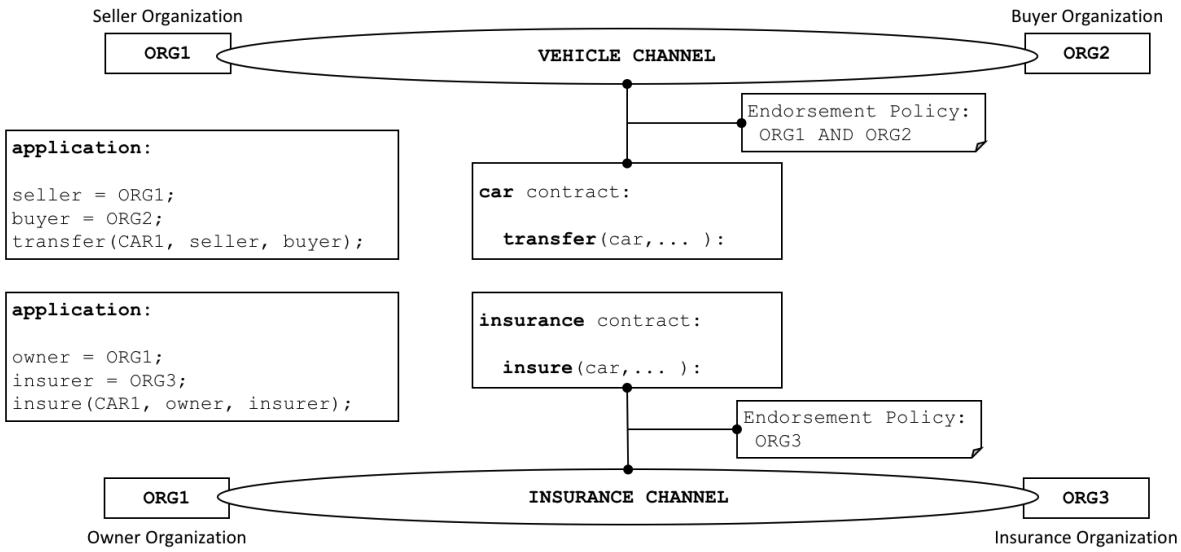
A transaction that is distributed to all peer nodes in the network is **validated** in two phases by each peer. Firstly, the transaction is checked to ensure it has been signed by sufficient organizations according to the endorsement policy. Secondly, it is checked to ensure that the current value of the world state matches the read set of the transaction when it was signed by the endorsing peer nodes; that there has been no intermediate update. If a transaction passes both these tests, it is marked as **valid**. All transactions are added to the blockchain history, whether **valid** or **invalid**, but only **valid** transactions result in an update to the world state.

In our example, `t3` is a valid transaction, so the owner of `CAR1` has been updated to `ORG2`. However, `t4` (not shown) is an invalid transaction, so while it was recorded in the ledger, the world state was not updated, and `CAR2` remains owned by `ORG2`.

Finally, to understand how to use a smart contract or chaincode with world state, read the chaincode namespace topic.

4.10.7 Channels

Hyperledger Fabric allows an organization to simultaneously participate in multiple, separate blockchain networks via **channels**. By joining multiple channels, an organization can participate in a so-called **network of networks**. Channels provide an efficient sharing of infrastructure while maintaining data and communications privacy. They are independent enough to help organizations separate their work traffic with different counterparties, but integrated enough to allow them to coordinate independent activities when necessary.



A channel provides a completely separate communication mechanism between a set of organizations. When a chaincode definition is committed to a channel, all the smart contracts within the chaincode are made available to the applications on that channel.

While the smart contract code is installed inside a chaincode package on an organizations peers, channel members can only execute a smart contract after the chaincode has been defined on a channel. The **chaincode definition** is a struct that contains the parameters that govern how a chaincode operates. These parameters include the chaincode name, version, and the endorsement policy. Each channel member agrees to the parameters of a chaincode by approving a chaincode definition for their organization. When a sufficient number of organizations (a majority by default) have approved to the same chaincode definition, the definition can be committed to the channel. The smart contracts inside the chaincode can then be executed by channel members, subject to the endorsement policy specified in the chaincode definition. The endorsement policy applies equally to all smart contracts defined within the same chaincode.

In the example *above*, a `car` contract is defined on the `VEHICLE` channel, and an `insurance` contract is defined on the `INSURANCE` channel. The chaincode definition of `car` specifies an endorsement policy that requires both `ORG1` and `ORG2` to sign transactions before they can be considered valid. The chaincode definition of the `insurance` contract specifies that only `ORG3` is required to endorse a transaction. `ORG1` participates in two networks, the `VEHICLE` channel and the `INSURANCE` network, and can coordinate activity with `ORG2` and `ORG3` across these two networks.

The chaincode definition provides a way for channel members to agree on the governance of a chaincode before they start using the smart contract to transact on the channel. Building on the example above, both `ORG1` and `ORG2` want to endorse transactions that invoke the `car` contract. Because the default policy requires that a majority of organizations approve a chaincode definition, both organizations need to approve an endorsement policy of `AND{ORG1,ORG2}`. Otherwise, `ORG1` and `ORG2` would approve different chaincode definitions and would be unable to commit the chaincode definition to the channel as a result. This process guarantees that a transaction from the `car` smart contract needs to be approved by both organizations.

4.10.8 Intercommunication

A Smart Contract can call other smart contracts both within the same channel and across different channels. It this way, they can read and write world state data to which they would not otherwise have access due to smart contract namespaces.

There are limitations to this inter-contract communication, which are described fully in the chaincode namespace topic.

4.10.9 System chaincode

The smart contracts defined within a chaincode encode the domain dependent rules for a business process agreed between a set of blockchain organizations. However, a chaincode can also define low-level program code which corresponds to domain independent *system* interactions, unrelated to these smart contracts for business processes.

The following are the different types of system chaincodes and their associated abbreviations:

- `_lifecycle` runs in all peers and manages the installation of chaincode on your peers, the approval of chaincode definitions for your organization, and the committing of chaincode definitions to channels. You can read more about how `_lifecycle` implements the Fabric chaincode lifecycle process.
- Lifecycle system chaincode (LSCC) manages the chaincode lifecycle for the 1.x releases of Fabric. This version of lifecycle required that chaincode be instantiated or upgraded on channels. You can still use LSCC to manage your chaincode if you have the channel application capability set to V1_4_x or below.
- **Configuration system chaincode (CSCC)** runs in all peers to handle changes to a channel configuration, such as a policy update. You can read more about this process in the following chaincode topic.
- **Query system chaincode (QSCC)** runs in all peers to provide ledger APIs which include block query, transaction query etc. You can read more about these ledger APIs in the transaction context topic.
- **Endorsement system chaincode (ESCC)** runs in endorsing peers to cryptographically sign a transaction response. You can read more about how the ESCC implements this process.
- **Validation system chaincode (VSCC)** validates a transaction, including checking endorsement policy and read-write set versioning. You can read more about the VSCC implements this process.

It is possible for low level Fabric developers and administrators to modify these system chaincodes for their own uses. However, the development and management of system chaincodes is a specialized activity, quite separate from the development of smart contracts, and is not normally necessary. Changes to system chaincodes must be handled with extreme care as they are fundamental to the correct functioning of a Hyperledger Fabric network. For example, if a system chaincode is not developed correctly, one peer node may update its copy of the world state or blockchain differently compared to another peer node. This lack of consensus is one form of a **ledger fork**, a very undesirable situation.

4.11 Fabric chaincode lifecycle

4.11.1 What is Chaincode?

Chaincode is a program, written in Go, Node.js, or Java that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it may be considered as a "smart contract". Ledger updates created by a chaincode are scoped exclusively to that chaincode and can't be accessed directly by another chaincode. However, within the same network, given the appropriate permission a chaincode may invoke another chaincode to access its state.

In this concept topic, we will explore chaincode through the eyes of a blockchain network operator rather than an application developer. Chaincode operators can use this topic as a guide to how to use the Fabric chainode lifecycle to deploy and manage chaincode on their network.

4.11.2 Deploying a chaincode

The Fabric chaincode lifecycle is a process that allows multiple organizations to agree on how a chaincode will be operated before it can be used on a channel. A network operator would use the Fabric lifecycle to perform the following tasks:

- *Install and define a chaincode*
- *Upgrade a chaincode*
- *Deployment Scenarios*
- *Migrate to the new Fabric lifecycle*

You can use the Fabric chaincode lifecycle by creating a new channel and setting the channel capabilities to V2_0. You will not be able to use the old lifecycle to install, instantiate, or update a chaincode on channels with V2_0 capabilities enabled. However, you can still invoke chaincode installed using the previous lifecycle model after you enable V2_0 capabilities. If you are upgrading from a v1.4.x network and need to edit your channel configurations to enable the new lifecycle, check out Enabling the new chaincode lifecycle.

4.11.3 Install and define a chaincode

Fabric chaincode lifecycle requires that organizations agree to the parameters that define a chaincode, such as name, version, and the chaincode endorsement policy. Channel members come to agreement using the following four steps. Not every organization on a channel needs to complete each step.

1. **Package the chaincode:** This step can be completed by one organization or by each organization.
2. **Install the chaincode on your peers:** Every organization that will use the chaincode to endorse a transaction or query the ledger needs to complete this step.
3. **Approve a chaincode definition for your organization:** Every organization that will use the chaincode needs to complete this step. The chaincode definition needs to be approved by a sufficient number of organizations to satisfy the channel's LifecycleEndorsment policy (a majority, by default) before the chaincode can be started on the channel.
4. **Commit the chaincode definition to the channel:** The commit transaction needs to be submitted by one organization once the required number of organizations on the channel have approved. The submitter first collects endorsements from enough peers of the organizations that have approved, and then submits the transaction to commit the chaincode definition.

This topic provides a detailed overview of the operations of the Fabric chaincode lifecycle rather than the specific commands. To learn more about how to use the Fabric lifecycle using the Peer CLI, see the Deploying a smart contract to a channel tutorial or the peer lifecycle command reference.

Step One: Packaging the smart contract

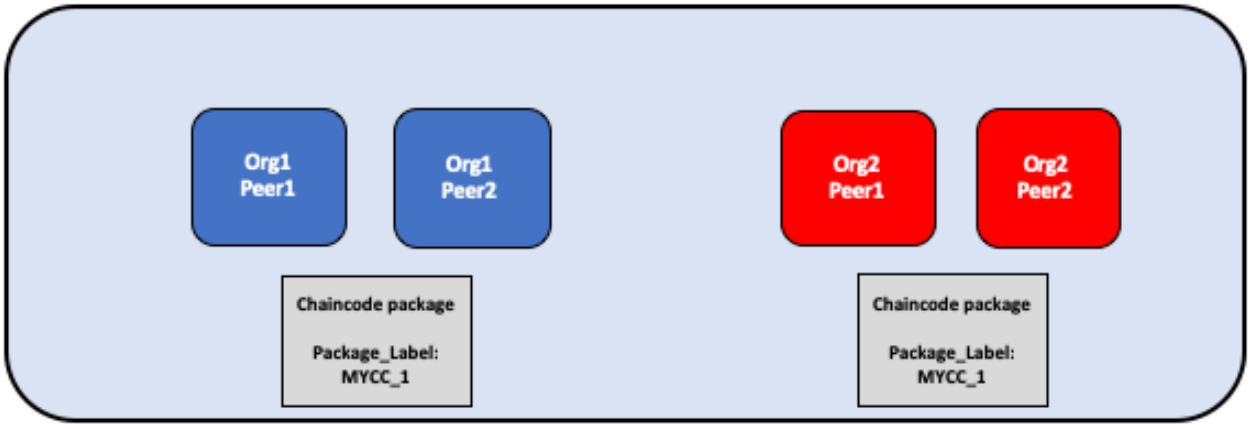
Chaincode needs to be packaged in a tar file before it can be installed on your peers. You can package a chaincode using the Fabric peer binaries, the Node Fabric SDK, or a third party tool such as GNU tar. When you create a chaincode package, you need to provide a chaincode package label to create a succinct and human readable description of the package.

If you use a third party tool to package the chaincode, the resulting file needs to be in the format below. The Fabric peer binaries and the Fabric SDKs will automatically create a file in this format.

- The chaincode needs to be packaged in a tar file, ending with a `.tar.gz` file extension.
- The tar file needs to contain two files (no directory): a metadata file "metadata.json" and another tar containing the chaincode files.

- "metadata.json" contains JSON that specifies the chaincode language, code path, and package label. You can see an example of a metadata file below:

```
{"Path":"fabric-samples/chaincode/fabcar/go","Type":"golang","Label":"fabcarv1"}
```

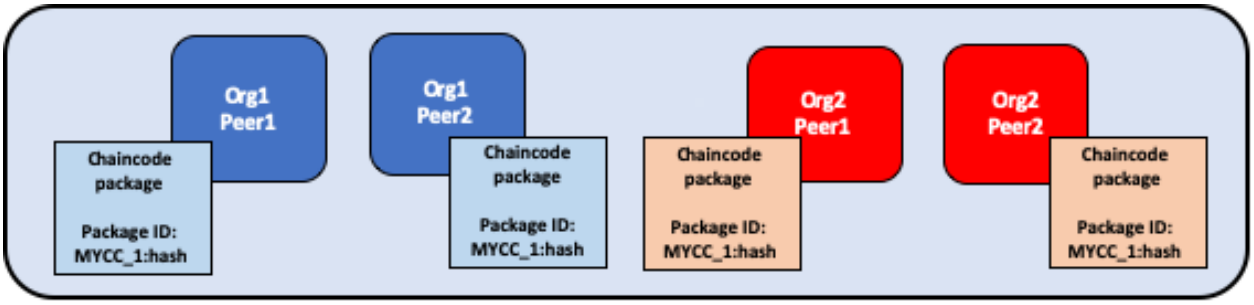


The chaincode is packaged separately by Org1 and Org2. Both organizations use MYCC_1 as their package label in order to identify the package using the name and version. It is not necessary for organizations to use the same package label.

Step Two: Install the chaincode on your peers

You need to install the chaincode package on every peer that will execute and endorse transactions. Whether using the CLI or an SDK, you need to complete this step using your **Peer Administrator**. Your peer will build the chaincode after the chaincode is installed, and return a build error if there is a problem with your chaincode. It is recommended that organizations only package a chaincode once, and then install the same package on every peer that belongs to their org. If a channel wants to ensure that each organization is running the same chaincode, one organization can package a chaincode and send it to other channel members out of band.

A successful install command will return a chaincode package identifier, which is the package label combined with a hash of the package. This package identifier is used to associate a chaincode package installed on your peers with a chaincode definition approved by your organization. **Save the identifier** for next step. You can also find the package identifier by querying the packages installed on your peer using the Peer CLI.



A peer administrator from Org1 and Org2 installs the chaincode package MYCC_1 on the peers joined to the channel. Installing the chaincode package builds the chaincode and creates a package identifier of MYCC_1:hash.

Step Three: Approve a chaincode definition for your organization

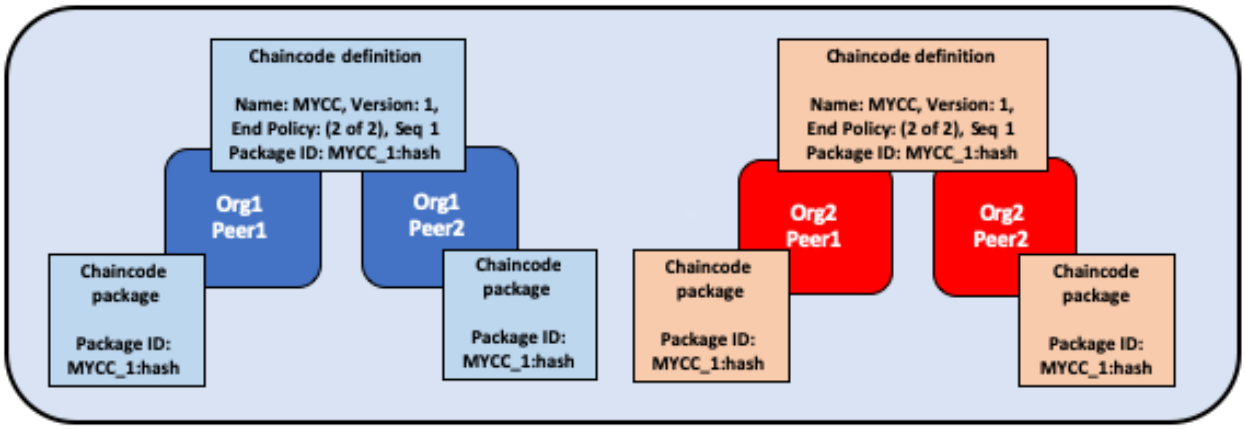
The chaincode is governed by a **chaincode definition**. When channel members approve a chaincode definition, the approval acts as a vote by an organization on the chaincode parameters it accepts. These approved organization

definitions allow channel members to agree on a chaincode before it can be used on a channel. The chaincode definition includes the following parameters, which need to be consistent across organizations:

- **Name:** The name that applications will use when invoking the chaincode.
- **Version:** A version number or value associated with a given chaincodes package. If you upgrade the chaincode binaries, you need to change your chaincode version as well.
- **Sequence:** The number of times the chaincode has been defined. This value is an integer, and is used to keep track of chaincode upgrades. For example, when you first install and approve a chaincode definition, the sequence number will be 1. When you next upgrade the chaincode, the sequence number will be incremented to 2.
- **Endorsement Policy:** Which organizations need to execute and validate the transaction output. The endorsement policy can be expressed as a string passed to the CLI, or it can reference a policy in the channel config. By default, the endorsement policy is set to `Channel/Application/Endorsement`, which defaults to require that a majority of organizations in the channel endorse a transaction.
- **Collection Configuration:** The path to a private data collection definition file associated with your chaincode. For more information about private data collections, see the Private Data architecture reference.
- **Initialization:** All chaincode need to contain an `Init` function that is used to initialize the chaincode. By default, this function is never executed. However, you can use the chaincode definition to request that the `Init` function be callable. If execution of `Init` is requested, fabric will ensure that `Init` is invoked before any other function and is only invoked once.
- **ESCC/VSCC Plugins:** The name of a custom endorsement or validation plugin to be used by this chaincode.

The chaincode definition also includes the **Package Identifier**. This is a required parameter for each organization that wants to use the chaincode. The package ID does not need to be the same for all organizations. An organization can approve a chaincode definition without installing a chaincode package or including the identifier in the definition.

Each channel member that wants to use the chaincode needs to approve a chaincode definition for their organization. This approval needs to be submitted to the ordering service, after which it is distributed to all peers. This approval needs to be submitted by your **Organization Administrator**. After the approval transaction has been successfully submitted, the approved definition is stored in a collection that is available to all the peers of your organization. As a result you only need to approve a chaincode for your organization once, even if you have multiple peers.



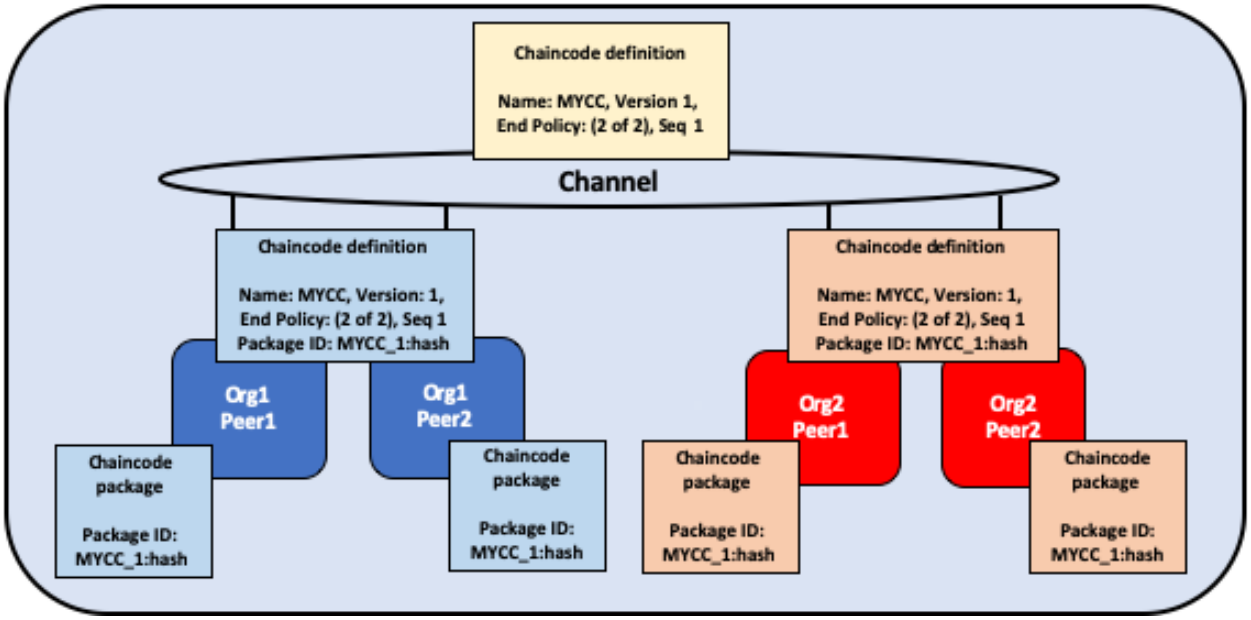
An organization administrator from Org1 and Org2 approve the chaincode definition of MYCC for their organization. The chaincode definition includes the chaincode name, version, and the endorsement policy, among other fields. Since both organizations will use the chaincode to endorse transactions, the approved definitions for both organizations need to include the packageID.

Step Four: Commit the chaincode definition to the channel

Once a sufficient number of channel members have approved a chaincode definition, one organization can commit the definition to the channel. You can use the `checkcommitreadiness` command to check whether committing the chaincode definition should be successful based on which channel members have approved a definition before committing it to the channel using the peer CLI. The commit transaction proposal is first sent to the peers of channel members, who query the chaincode definition approved for their organizations and endorse the definition if their organization has approved it. The transaction is then submitted to the ordering service, which then commits the chaincode definition to the channel. The commit definition transaction needs to be submitted as the **Organization Administrator**.

The number of organizations that need to approve a definition before it can be successfully committed to the channel is governed by the `Channel/Application/LifecycleEndorsement` policy. By default, this policy requires that a majority of organizations in the channel endorse the transaction. The LifecycleEndorsement policy is separate from the chaincode endorsement policy. For example, even if a chaincode endorsement policy only requires signatures from one or two organizations, a majority of channel members still need to approve the chaincode definition according to the default policy. When committing a channel definition, you need to target enough peer organizations in the channel to satisfy your LifecycleEndorsement policy. You can learn more about the Fabric chaincode lifecycle policies in the Policies concept topic.

You can also set the `Channel/Application/LifecycleEndorsement` policy to be a signature policy and explicitly specify the set of organizations on the channel that can approve a chaincode definition. This allows you to create a channel where a select number of organizations act as chaincode administrators and govern the business logic used by the channel. You can also use a signature policy if your channel has a large number Idemix organizations, which cannot approve chaincode definitions or endorse chaincode and may prevent the channel from reaching a majority as a result.

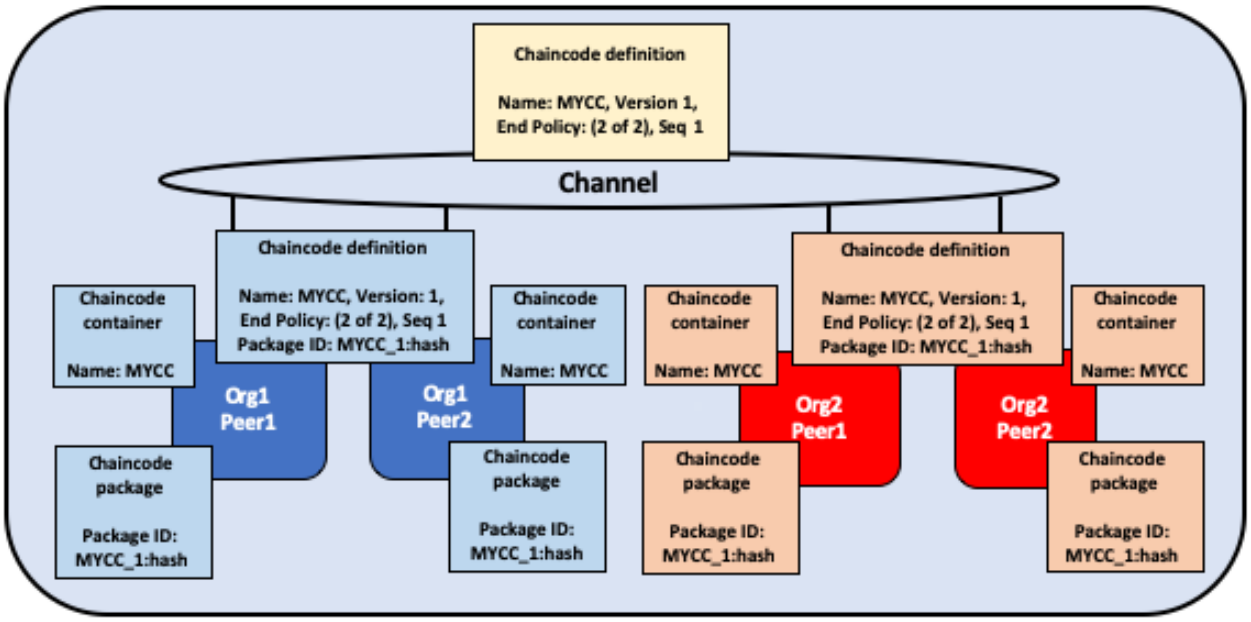


One organization administrator from Org1 or Org2 commits the chaincode definition to the channel. The definition on the channel does not include the packageID.

An organization can approve a chaincode definition without installing the chaincode package. If an organization does not need to use the chaincode, they can approve a chaincode definition without a package identifier to ensure that the Lifecycle Endorsement policy is satisfied.

After the chaincode definition has been committed to the channel, the chaincode container will launch on all of the peers where the chaincode has been installed, allowing channel members to start using the chaincode. It may take

a few minutes for the chaincode container to start. You can use the chaincode definition to require the invocation of the `Init` function to initialize the chaincode. If the invocation of the `Init` function is requested, the first invoke of the chaincode must be a call to the `Init` function. The invoke of the `Init` function is subject to the chaincode endorsement policy.

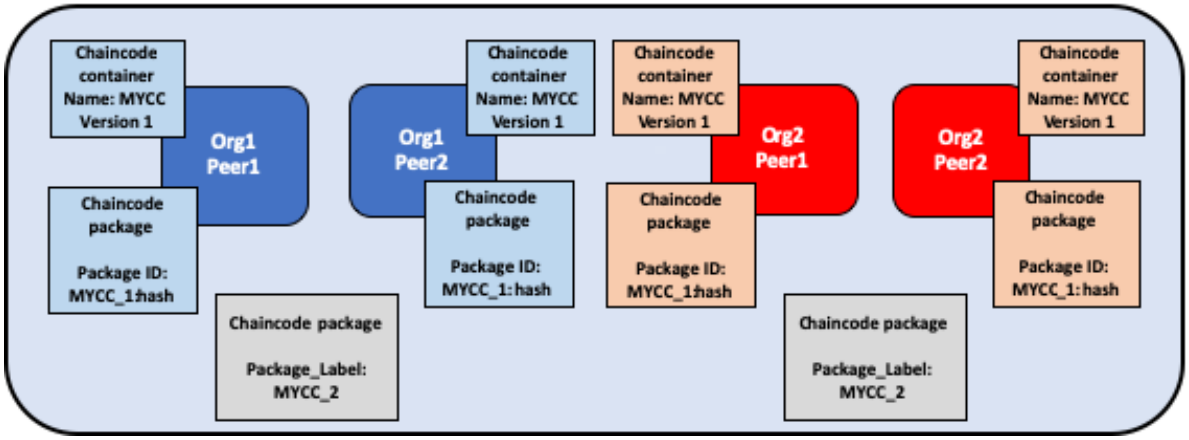


Once MYCC is defined on the channel, Org1 and Org2 can start using the chaincode. The first invoke of the chaincode on each peer starts the chaincode container on that peer.

4.11.4 Upgrade a chaincode

You can upgrade a chaincode using the same Fabric lifecycle process as you used to install and start the chainocode. You can upgrade the chaincode binaries, or only update the chaincode policies. Follow these steps to upgrade a chaincode:

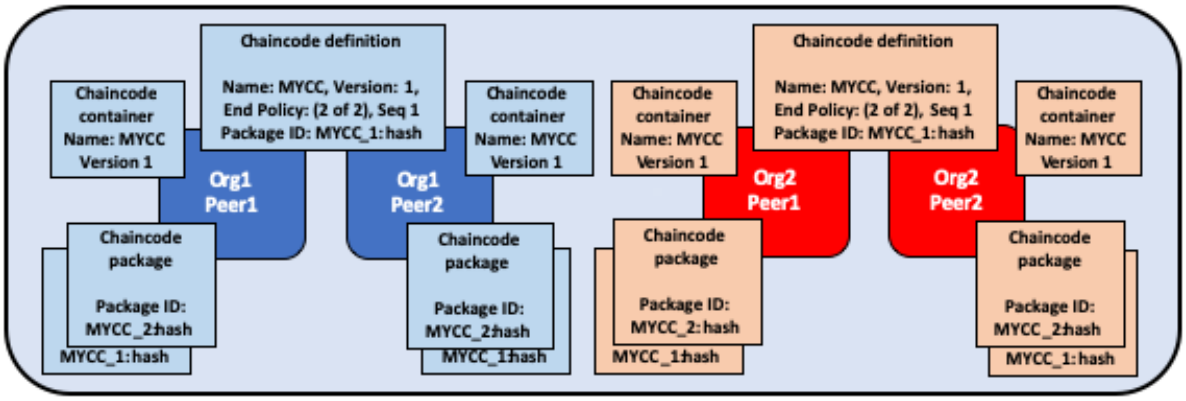
1. **Repackage the chaincode:** You only need to complete this step if you are upgrading the chaincode binaries.



Org1 and Org2 upgrade the chaincode binaries and repackage the chaincode. Both organizations use a different package label.

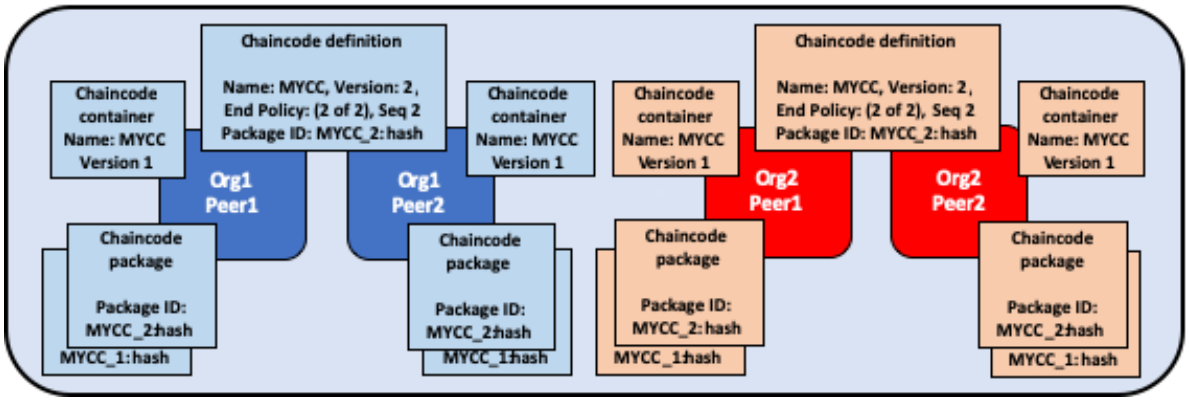
2. **Install the new chaincode package on your peers:** Once again, you only need to complete this step if you are upgrading the chaincode binaries. Installing the new chaincode package will generate a package ID, which you

will need to pass to the new chaincode definition. You also need to change the chaincode version, which is used by the lifecycle process to track if the chaincode binaries have been upgraded.



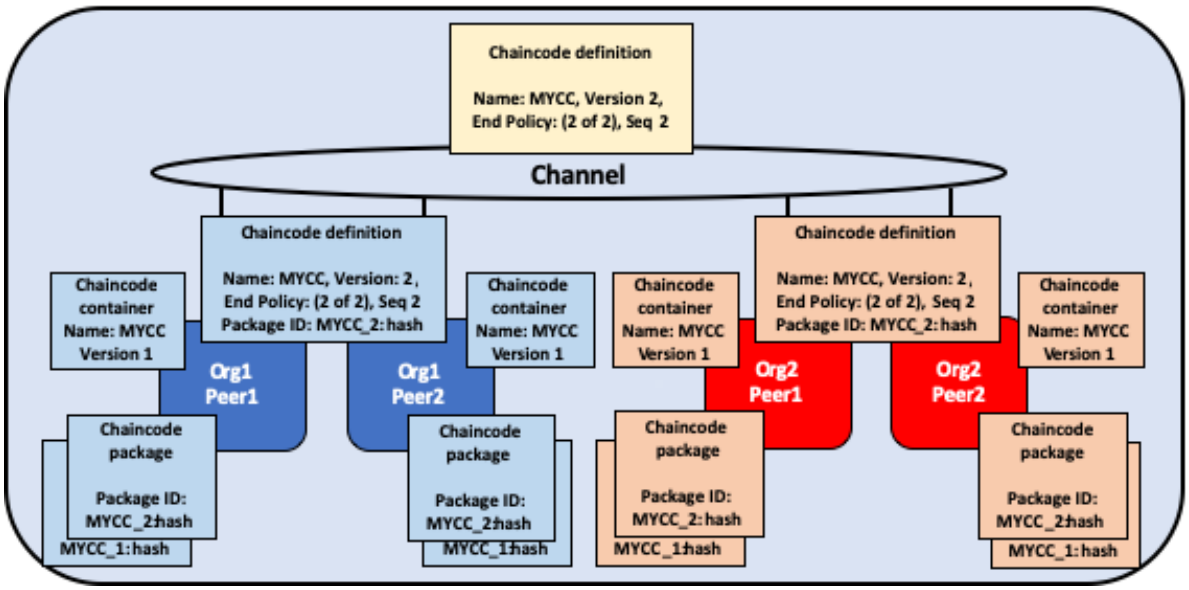
Org1 and Org2 install the new package on their peers. The installation creates a new packageID.

3. **Approve a new chaincode definition:** If you are upgrading the chaincode binaries, you need to update the chaincode version and the package ID in the chaincode definition. You can also update your chaincode endorsement policy without having to repackage your chaincode binaries. Channel members simply need to approve a definition with the new policy. The new definition needs to increment the **sequence** variable in the definition by one.



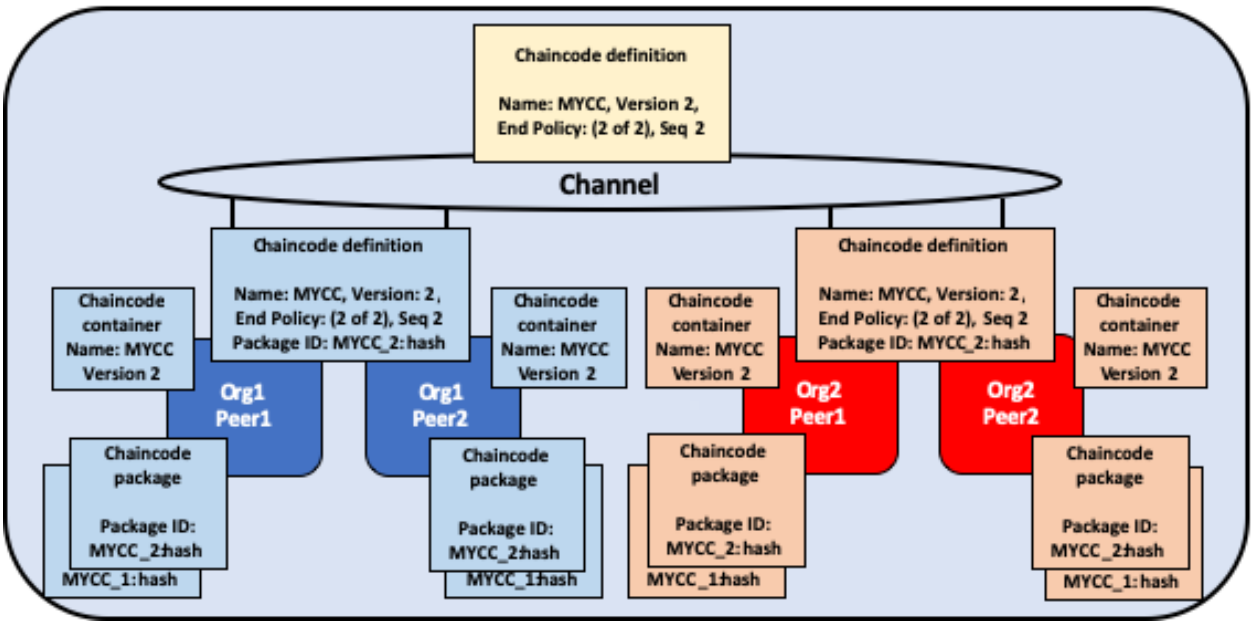
Organization administrators from Org1 and Org2 approve the new chaincode definition for their respective organizations. The new definition references the new packageID and changes the chaincode version. Since this is the first update of the chaincode, the sequence is incremented from one to two.

4. **Commit the definition to the channel:** When a sufficient number of channel members have approved the new chaincode definition, one organization can commit the new definition to upgrade the chaincode definition to the channel. There is no separate upgrade command as part of the lifecycle process.



An organization administrator from Org1 or Org2 commits the new chaincode definition to the channel.

After you commit the chaincode definition, a new chaincode container will launch with the code from the upgraded chaincode binaries. If you requested the execution of the `Init` function in the chaincode definition, you need to initialize the upgraded chaincode by invoking the `Init` function again after the new definition is successfully committed. If you updated the chaincode definition without changing the chaincode version, the chaincode container will remain the same and you do not need to invoke `Init` function.



Once the new definition has been committed to the channel, each peer will automatically start the new chaincode container.

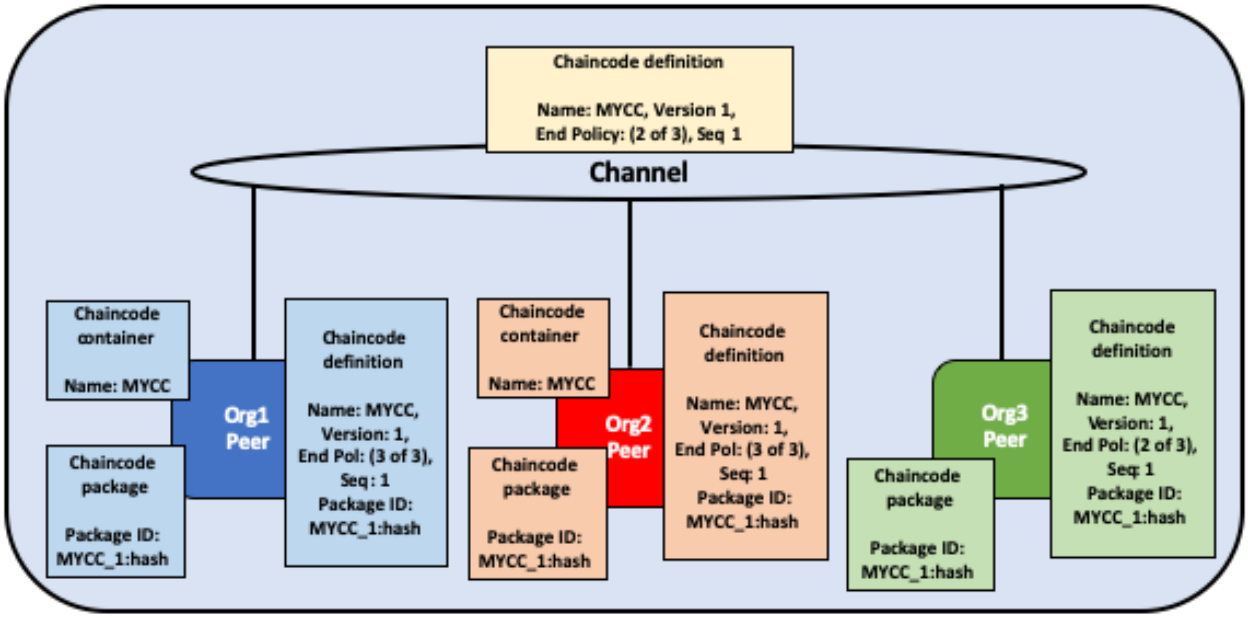
The Fabric chaincode lifecycle uses the **sequence** in the chaincode definition to keep track of upgrades. All channel members need to increment the sequence number by one and approve a new definition to upgrade the chaincode. The version parameter is used to track the chaincode binaries, and needs to be changed only when you upgrade the chaincode binaries.

4.11.5 Deployment scenarios

The following examples illustrate how you can use the Fabric chaincode lifecycle to manage channels and chaincode.

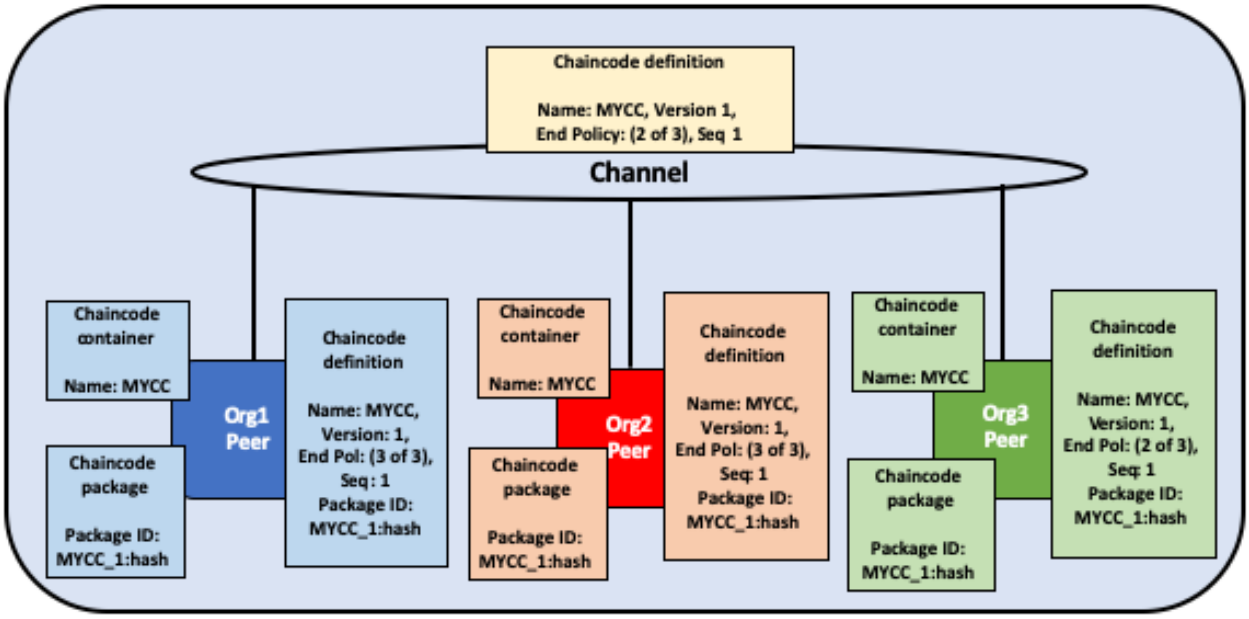
Joining a channel

A new organization can join a channel with a chaincode already defined, and start using the chaincode after installing the chaincode package and approving the chaincode definition that has already been committed to the channel.



Org3 joins the channel and approves the same chaincode definition that was previously committed to the channel by Org1 and Org2.

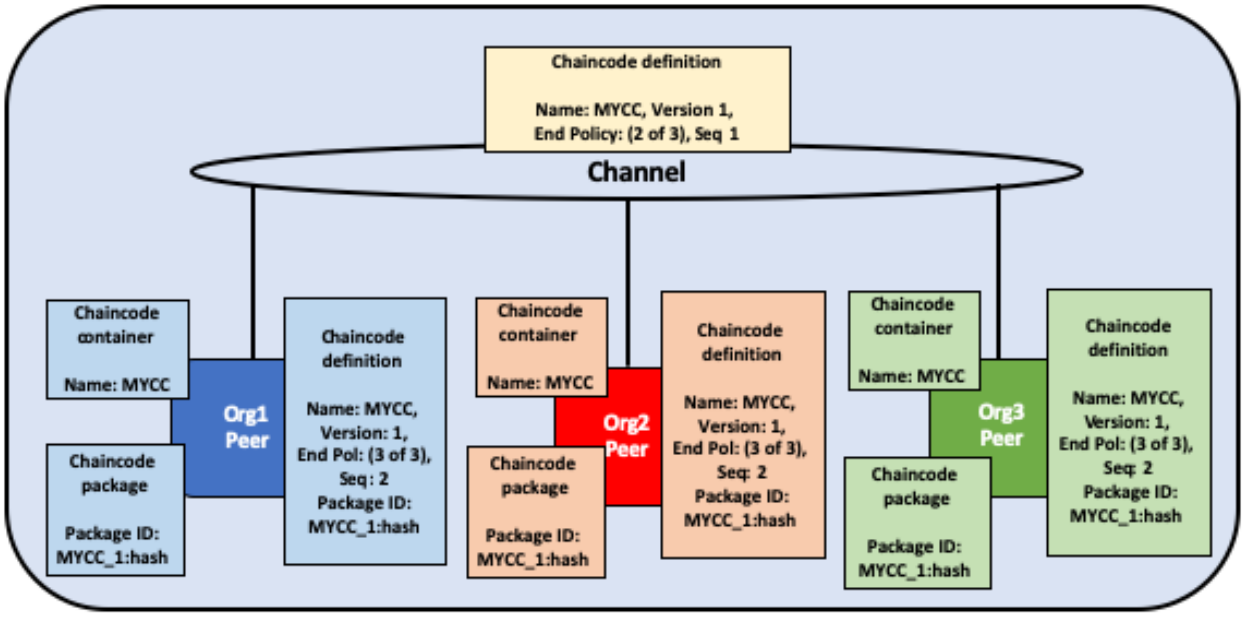
After approving the chaincode definition, the new organization can start using the chaincode after the package has been installed on their peers. The definition does not need to be committed again. If the endorsement policy is set the default policy that requires endorsements from a majority of channel members, then the endorsement policy will be updated automatically to include the new organization.



The chaincode container will start after the first invoke of the chaincode on the Org3 peer.

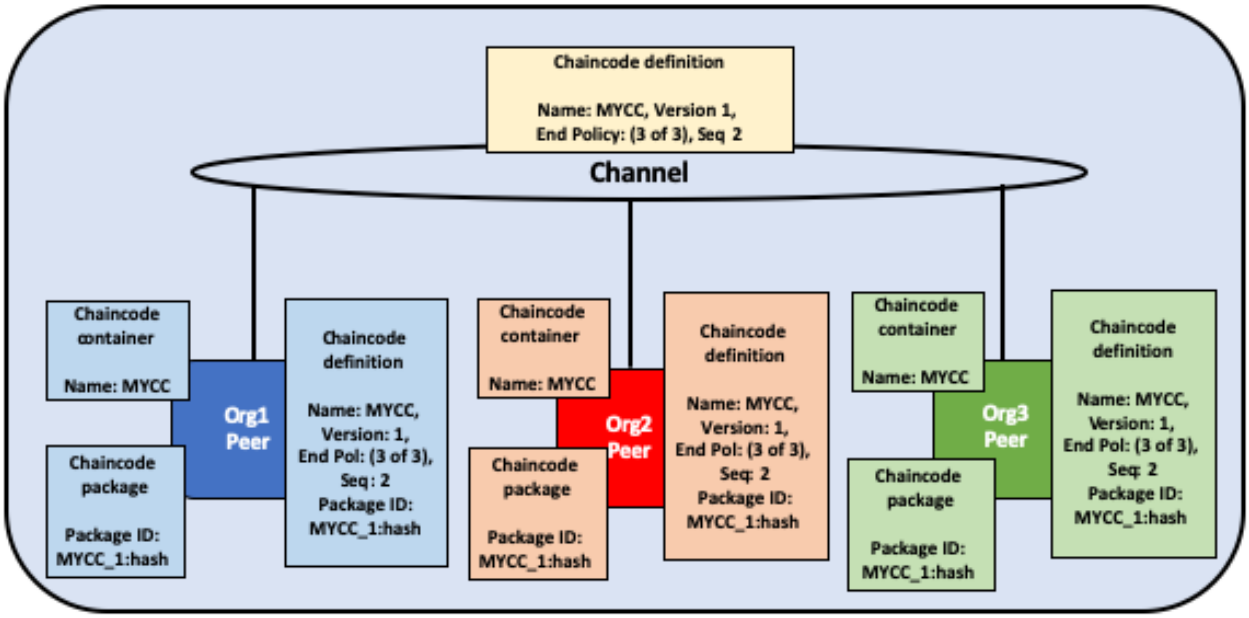
Updating an endorsement policy

You can use the chaincode definition to update an endorsement policy without having to repackage or re-install the chaincode. Channel members can approve a chaincode definition with a new endorsement policy and commit it to the channel.



Org1, Org2, and Org3 approve a new endorsement policy requiring that all three organizations endorse a transaction. They increment the definition sequence from one to two, but do not need to update the chaincode version.

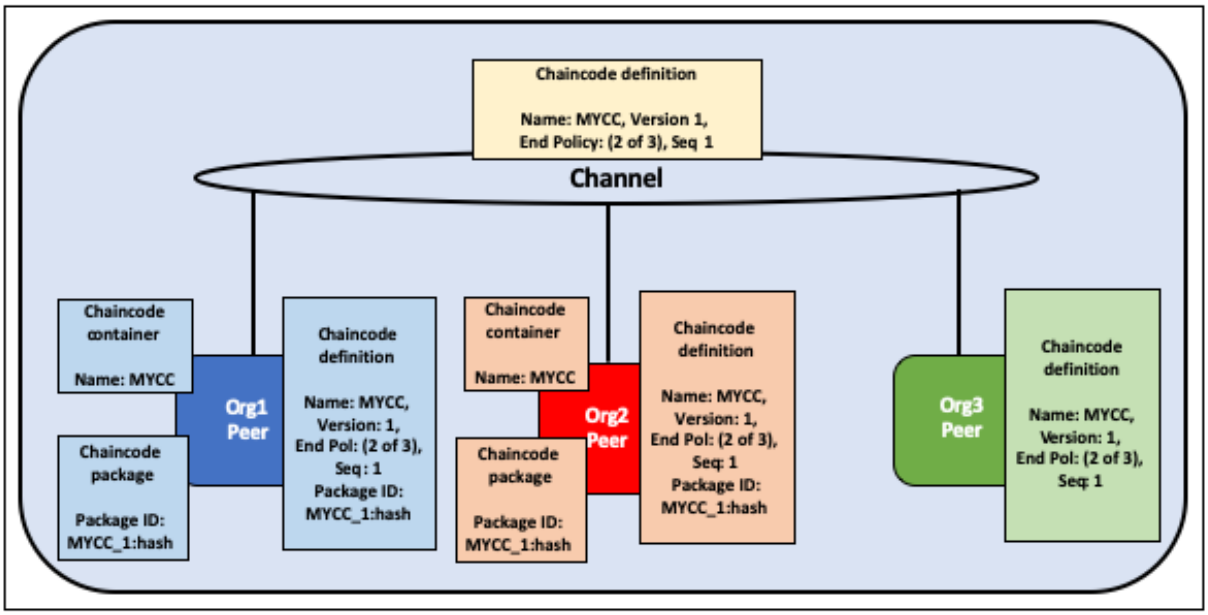
The new endorsement policy will take effect after the new definition is committed to the channel. Channel members do not have to restart the chaincode container by invoking the chaincode or executing the `Init` function in order to update the endorsement policy.



One organization commits the new chaincode definition to the channel to update the endorsement policy.

Approving a definition without installing the chaincode

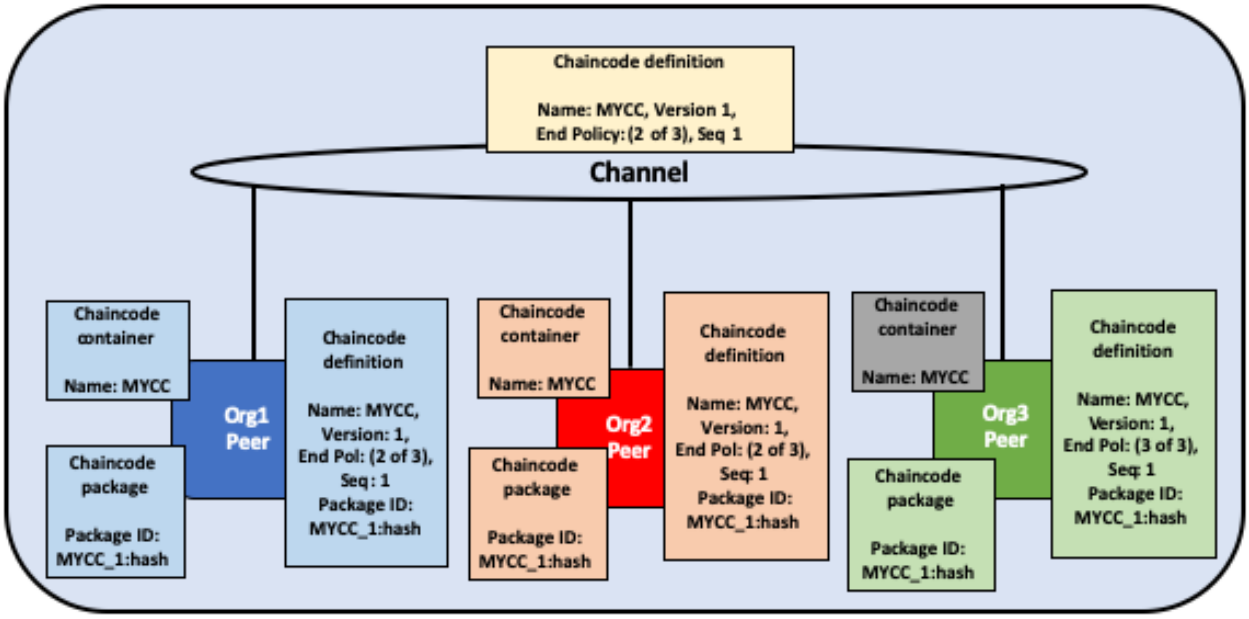
You can approve a chaincode definition without installing the chaincode package. This allows you to endorse a chaincode definition before it is committed to the channel, even if you do not want to use the chaincode to endorse transactions or query the ledger. You need to approve the same parameters as other members of the channel, but not need to include the packageID as part of the chaincode definition.



Org3 does not install the chaincode package. As a result, they do not need to provide a packageID as part of chaincode definition. However, Org3 can still endorse the definition of MYCC that has been committed to the channel.

One organization disagrees on the chaincode definition

An organization that does not approve a chaincode definition that has been committed to the channel cannot use the chaincode. Organizations that have either not approved a chaincode definition, or approved a different chaincode definition will not be able to execute the chaincode on their peers.

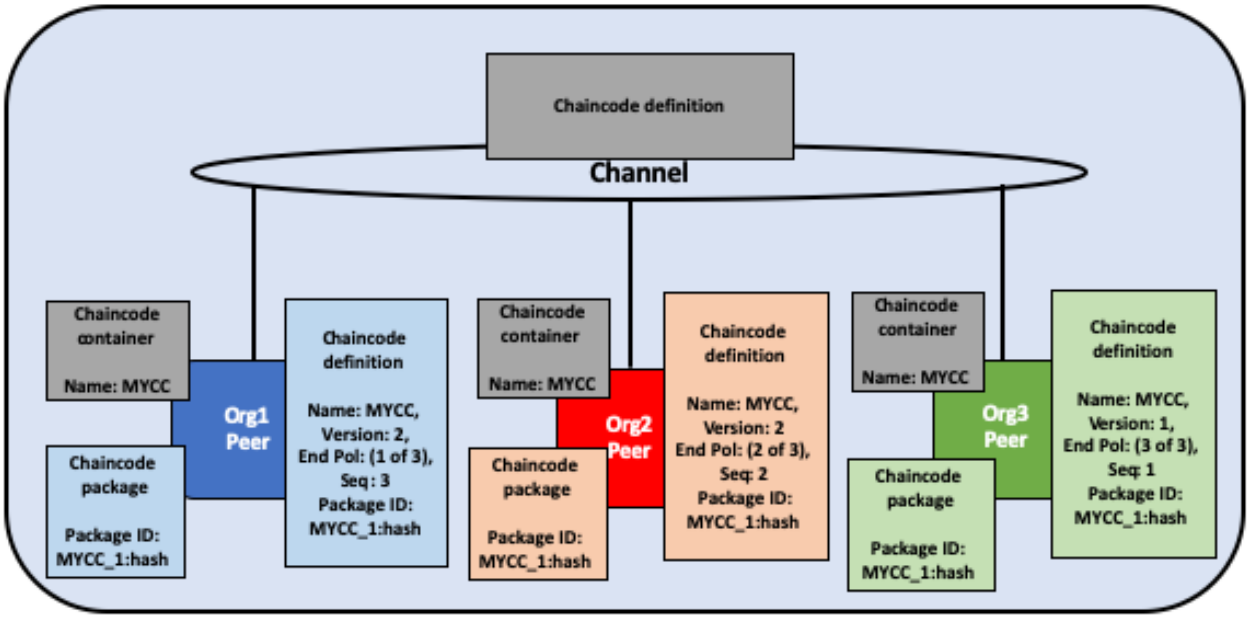


Org3 approves a chaincode definition with a different endorsement policy than Org1 and Org2. As a result, Org3 cannot use the MYCC chaincode on the channel. However, Org1 or Org2 can still get enough endorsements to commit the definition to the channel and use the chaincode. Transactions from the chaincode will still be added to the ledger and stored on the Org3 peer. However, the Org3 will not be able to endorse transactions.

An organization can approve a new chaincode definition with any sequence number or version. This allows you to approve the definition that has been committed to the channel and start using the chaincode. You can also approve a new chaincode definition in order to correct any mistakes made in the process of approving or packaging a chaincode.

The channel does not agree on a chaincode definition

If the organizations on a channel do not agree on a chaincode definition, the definition cannot be committed to the channel. None of the channel members will be able to use the chaincode.

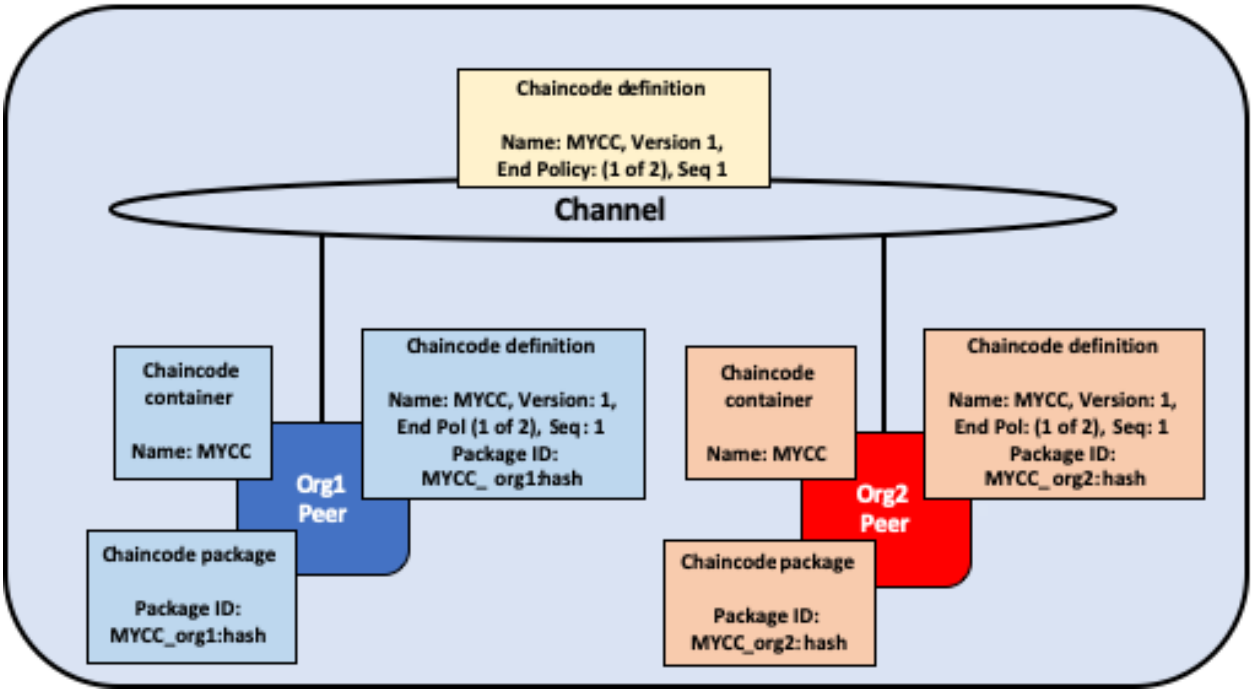


Org1, Org2, and Org3 all approve different chaincode definitions. As a result, no member of the channel can get enough endorsements to commit a chaincode definition to the channel. No channel member will be able to use the chaincode.

Organizations install different chaincode packages

Each organization can use a different packageID when they approve a chaincode definition. This allows channel members to install different chaincode binaries that use the same endorsement policy and read and write to data in the same chaincode namespace.

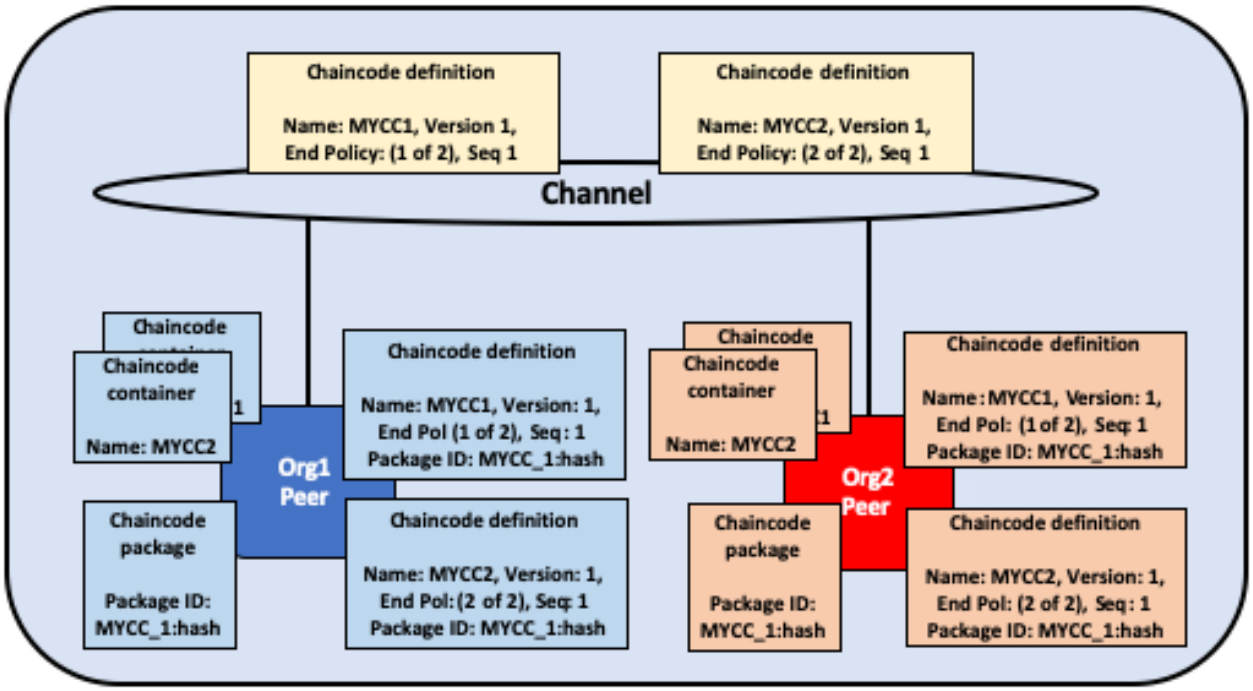
Organizations can use this capability to install smart contracts that contain business logic that is specific to their organization. Each organization's smart contract could contain additional validation that the organization requires before their peers endorse a transaction. Each organization can also write code that helps integrate the smart contract with data from their existing systems.



Org1 and Org2 each install versions of the MYCC chaincode containing business logic that is specific to their organization.

Creating multiple chaincodes using one package

You can use one chaincode package to create multiple chaincode instances on a channel by approving and committing multiple chaincode definitions. Each definition needs to specify a different chaincode name. This allows you to run multiple instances of a smart contract on a channel, but have the contract be subject to different endorsement policies.



Org1 and Org2 use the MYCC_1 chaincode package to approve and commit two different chaincode definitions. As a result, both peers have two chaincode containers running on their peers. MYCC1 has an endorsement policy of 1 out of 2, while MYCC2 has an endorsement policy of 2 out of 2.

4.11.6 Migrate to the new Fabric lifecycle

For information about migrating to the new lifecycle, check out Considerations for getting to v2.0.

If you need to update your channel configurations to enable the new lifecycle, check out Enabling the new chaincode lifecycle.

4.11.7 More information

You can watch video below to learn more about the motivation of the new Fabric chaincode lifecycle and how it is implemented.

4.12 Private data

4.12.1 What is private data?

In cases where a group of organizations on a channel need to keep data private from other organizations on that channel, they have the option to create a new channel comprising just the organizations who need access to the data. However, creating separate channels in each of these cases creates additional administrative overhead (maintaining chaincode versions, policies, MSPs, etc), and doesn't allow for use cases in which you want all channel participants to see a transaction while keeping a portion of the data private.

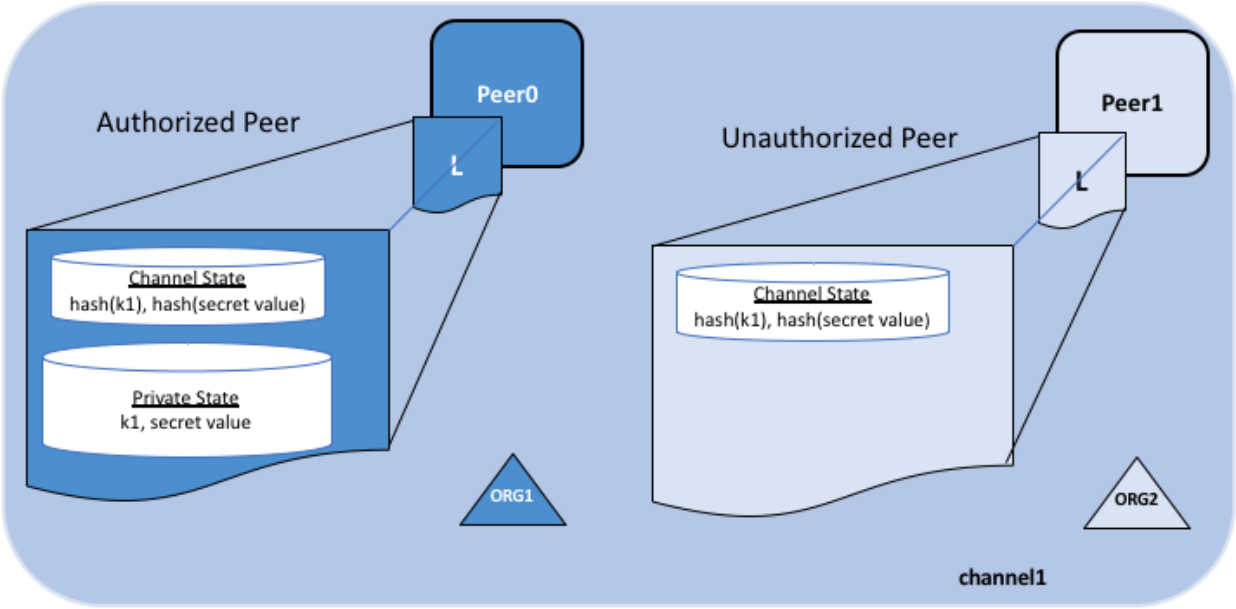
That's why Fabric offers the ability to create **private data collections**, which allow a defined subset of organizations on a channel the ability to endorse, commit, or query private data without having to create a separate channel.

4.12.2 What is a private data collection?

A collection is the combination of two elements:

1. **The actual private data**, sent peer-to-peer via gossip protocol to only the organization(s) authorized to see it. This data is stored in a private state database on the peers of authorized organizations, which can be accessed from chaincode on these authorized peers. The ordering service is not involved here and does not see the private data. Note that because gossip distributes the private data peer-to-peer across authorized organizations, it is required to set up anchor peers on the channel, and configure CORE_PEER_GOSSIP_EXTERNALENDPOINT on each peer, in order to bootstrap cross-organization communication.
2. **A hash of that data**, which is endorsed, ordered, and written to the ledgers of every peer on the channel. The hash serves as evidence of the transaction and is used for state validation and can be used for audit purposes.

The following diagram illustrates the ledger contents of a peer authorized to have private data and one which is not.



Collection members may decide to share the private data with other parties if they get into a dispute or if they want to transfer the asset to a third party. The third party can then compute the hash of the private data and see if it matches the state on the channel ledger, proving that the state existed between the collection members at a certain point in time.

In some cases, you may decide to have a set of collections each comprised of a single organization. For example an organization may record private data in their own collection, which could later be shared with other channel members and referenced in chaincode transactions. We'll see examples of this in the sharing private data topic below.

When to use a collection within a channel vs. a separate channel

- Use **channels** when entire transactions (and ledgers) must be kept confidential within a set of organizations that are members of the channel.
- Use **collections** when transactions (and ledgers) must be shared among a set of organizations, but when only a subset of those organizations should have access to some (or all) of the data within a transaction. Additionally, since private data is disseminated peer-to-peer rather than via blocks, use private data collections when transaction data must be kept confidential from ordering service nodes.

4.12.3 A use case to explain collections

Consider a group of five organizations on a channel who trade produce:

- **A Farmer** selling his goods abroad
- **A Distributor** moving goods abroad
- **A Shipper** moving goods between parties
- **A Wholesaler** purchasing goods from distributors
- **A Retailer** purchasing goods from shippers and wholesalers

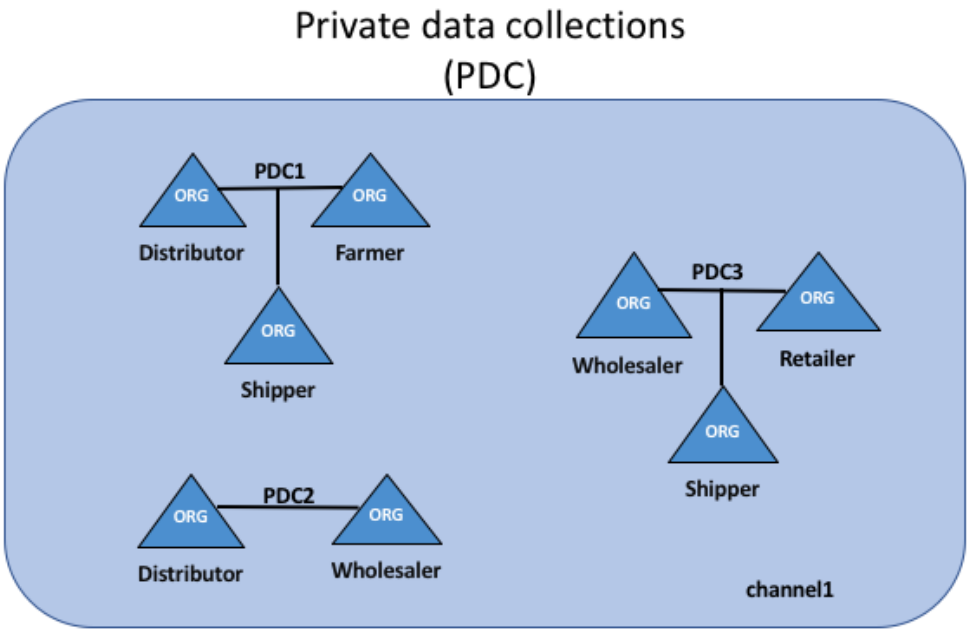
The **Distributor** might want to make private transactions with the **Farmer** and **Shipper** to keep the terms of the trades confidential from the **Wholesaler** and the **Retailer** (so as not to expose the markup they're charging).

The **Distributor** may also want to have a separate private data relationship with the **Wholesaler** because it charges them a lower price than it does the **Retailer**.

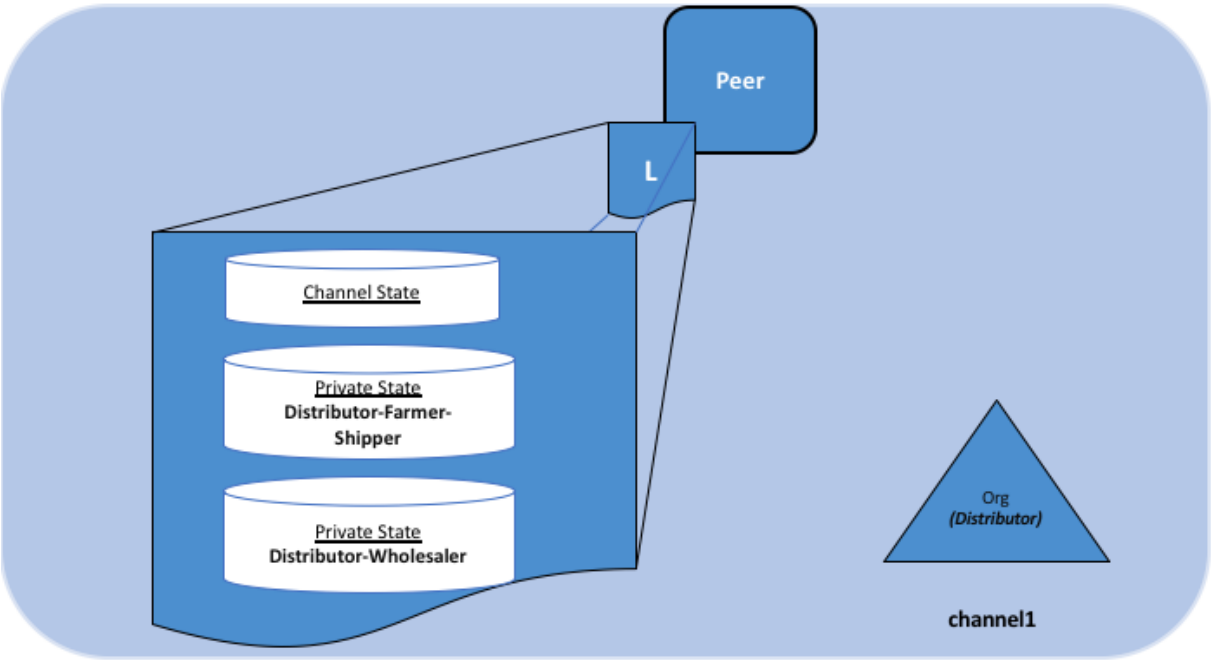
The **Wholesaler** may also want to have a private data relationship with the **Retailer** and the **Shipper**.

Rather than defining many small channels for each of these relationships, multiple private data collections **(PDC)** can be defined to share private data between:

1. PDC1: **Distributor**, **Farmer** and **Shipper**
2. PDC2: **Distributor** and **Wholesaler**
3. PDC3: **Wholesaler**, **Retailer** and **Shipper**



Using this example, peers owned by the **Distributor** will have multiple private databases inside their ledger which includes the private data from the **Distributor**, **Farmer** and **Shipper** relationship and the **Distributor** and **Wholesaler** relationship.



4.12.4 Transaction flow with private data

When private data collections are referenced in chaincode, the transaction flow is slightly different in order to protect the confidentiality of the private data as transactions are proposed, endorsed, and committed to the ledger.

For details on transaction flows that don't use private data refer to our documentation on transaction flow.

1. The client application submits a proposal request to invoke a chaincode function (reading or writing private data) to endorsing peers which are part of authorized organizations of the collection. The private data, or data used to generate private data in chaincode, is sent in a `transient` field of the proposal.
2. The endorsing peers simulate the transaction and store the private data in a `transient data store` (a temporary storage local to the peer). They distribute the private data, based on the collection policy, to authorized peers via gossip.
3. The endorsing peer sends the proposal response back to the client. The proposal response includes the endorsed read/write set, which includes public data, as well as a hash of any private data keys and values. *No private data is sent back to the client*. For more information on how endorsement works with private data, click here.
4. The client application submits the transaction (which includes the proposal response with the private data hashes) to the ordering service. The transactions with the private data hashes get included in blocks as normal. The block with the private data hashes is distributed to all the peers. In this way, all peers on the channel can validate transactions with the hashes of the private data in a consistent way, without knowing the actual private data.
5. At block commit time, authorized peers use the collection policy to determine if they are authorized to have access to the private data. If they do, they will first check their local `transient data store` to determine if they have already received the private data at chaincode endorsement time. If not, they will attempt to pull the private data from another authorized peer. Then they will validate the private data against the hashes in the public block and commit the transaction and the block. Upon validation/commit, the private data is moved to their copy of the private state database and private writeset storage. The private data is then deleted from the `transient data store`.

4.12.5 Sharing private data

In many scenarios private data keys/values in one collection may need to be shared with other channel members or with other private data collections, for example when you need to transact on private data with a channel member or group of channel members who were not included in the original private data collection. The receiving parties will typically want to verify the private data against the on-chain hashes as part of the transaction.

There are several aspects of private data collections that enable the sharing and verification of private data:

- First, you don't necessarily have to be a member of a collection to write to a key in a collection, as long as the endorsement policy is satisfied. Endorsement policy can be defined at the chaincode level, key level (using state-based endorsement), or collection level (starting in Fabric v2.0).
- Second, starting in v1.4.2 there is a chaincode API GetPrivateDataHash() that allows chaincode on non-member peers to read the hash value of a private key. This is an important feature as you will see later, because it allows chaincode to verify private data against the on-chain hashes that were created from private data in previous transactions.

This ability to share and verify private data should be considered when designing applications and the associated private data collections. While you can certainly create sets of multilateral private data collections to share data among various combinations of channel members, this approach may result in a large number of collections that need to be defined. Alternatively, consider using a smaller number of private data collections (e.g. one collection per organization, or one collection per pair of organizations), and then sharing private data with other channel members, or with other collections as the need arises. Starting in Fabric v2.0, implicit organization-specific collections are available for any chaincode to utilize, so that you don't even have to define these per-organization collections when deploying chaincode.

Private data sharing patterns

When modeling private data collections per organization, multiple patterns become available for sharing or transferring private data without the overhead of defining many multilateral collections. Here are some of the sharing patterns that could be leveraged in chaincode applications:

- **Use a corresponding public key for tracking public state** - You can optionally have a matching public key for tracking public state (e.g. asset properties, current ownership. etc), and for every organization that should have access to the asset's corresponding private data, you can create a private key/value in each organization's private data collection.
- **Chaincode access control** - You can implement access control in your chaincode, to specify which clients can query private data in a collection. For example, store an access control list for a private data collection key or range of keys, then in the chaincode get the client submitter's credentials (using GetCreator() chaincode API or CID library API GetID() or GetMSPID()), and verify they have access before returning the private data. Similarly you could require a client to pass a passphrase into chaincode, which must match a passphrase stored at the key level, in order to access the private data. Note, this pattern can also be used to restrict client access to public state data.
- **Sharing private data out of band** - As an off-chain option, you could share private data out of band with other organizations, and they can hash the key/value to verify it matches the on-chain hash by using GetPrivateDataHash() chaincode API. For example, an organization that wishes to purchase an asset from you may want to verify an asset's properties and that you are the legitimate owner by checking the on-chain hash, prior to agreeing to the purchase.
- **Sharing private data with other collections** - You could 'share' the private data on-chain with chaincode that creates a matching key/value in the other organization's private data collection. You'd pass the private data key/value to chaincode via transient field, and the chaincode could confirm a hash of the passed private data matches the on-chain hash from your collection using GetPrivateDataHash(), and then write the private data to the other organization's private data collection.

- **Transferring private data to other collections** - You could 'transfer' the private data with chaincode that deletes the private data key in your collection, and creates it in another organization's collection. Again, use the transient field to pass the private data upon chaincode invoke, and in the chaincode use GetPrivateDataHash() to confirm that the data exists in your private data collection, before deleting the key from your collection and creating the key in another organization's collection. To ensure that a transaction always deletes from one collection and adds to another collection, you may want to require endorsements from additional parties, such as a regulator or auditor.
- **Using private data for transaction approval** - If you want to get a counterparty's approval for a transaction before it is completed (e.g. an on-chain record that they agree to purchase an asset for a certain price), the chaincode can require them to 'pre-approve' the transaction, by either writing a private key to their private data collection or your collection, which the chaincode will then check using GetPrivateDataHash(). In fact, this is exactly the same mechanism that the built-in lifecycle system chaincode uses to ensure organizations agree to a chaincode definition before it is committed to a channel. Starting with Fabric v2.0, this pattern becomes more powerful with collection-level endorsement policies, to ensure that the chaincode is executed and endorsed on the collection owner's own trusted peer. Alternatively, a mutually agreed key with a key-level endorsement policy could be used, that is then updated with the pre-approval terms and endorsed on peers from the required organizations.
- **Keeping transactors private** - Variations of the prior pattern can also eliminate leaking the transactors for a given transaction. For example a buyer indicates agreement to buy on their own collection, then in a subsequent transaction seller references the buyer's private data in their own private data collection. The proof of transaction with hashed references is recorded on-chain, only the buyer and seller know that they are the transactors, but they can reveal the pre-images if a need-to-know arises, such as in a subsequent transaction with another party who could verify the hashes.

Coupled with the patterns above, it is worth noting that transactions with private data can be bound to the same conditions as regular channel state data, specifically:

- **Key level transaction access control** - You can include ownership credentials in a private data value, so that subsequent transactions can verify that the submitter has ownership privilege to share or transfer the data. In this case the chaincode would get the submitter's credentials (e.g. using GetCreator() chaincode API or CID library API GetID() or GetMSPID()), combine it with other private data that gets passed to the chaincode, hash it, and use GetPrivateDataHash() to verify that it matches the on-chain hash before proceeding with the transaction.
- **Key level endorsement policies** - And also as with normal channel state data, you can use state-based endorsement to specify which organizations must endorse transactions that share or transfer private data, using SetPrivateDataValidationParameter() chaincode API, for example to specify that only an owner's organization peer, custodian's organization peer, or other third party must endorse such transactions.

Example scenario: Asset transfer using private data collections

The private data sharing patterns mentioned above can be combined to enable powerful chaincode-based applications. For example, consider how an asset transfer scenario could be implemented using per-organization private data collections:

- An asset may be tracked by a UUID key in public chaincode state. Only the asset's ownership is recorded, nothing else is known about the asset.
- The chaincode will require that any transfer request must originate from the owning client, and the key is bound by state-based endorsement requiring that a peer from the owner's organization and a regulator's organization must endorse any transfer requests.
- The asset owner's private data collection contains the private details about the asset, keyed by a hash of the UUID. Other organizations and the ordering service will only see a hash of the asset details.
- Let's assume the regulator is a member of each collection as well, and therefore persists the private data, although this need not be the case.

A transaction to trade the asset would unfold as follows:

1. Off-chain, the owner and a potential buyer strike a deal to trade the asset for a certain price.
2. The seller provides proof of their ownership, by either passing the private details out of band, or by providing the buyer with credentials to query the private data on their node or the regulator's node.
3. Buyer verifies a hash of the private details matches the on-chain public hash.
4. The buyer invokes chaincode to record their bid details in their own private data collection. The chaincode is invoked on buyer's peer, and potentially on regulator's peer if required by the collection endorsement policy.
5. The current owner (seller) invokes chaincode to sell and transfer the asset, passing in the private details and bid information. The chaincode is invoked on peers of the seller, buyer, and regulator, in order to meet the endorsement policy of the public key, as well as the endorsement policies of the buyer and seller private data collections.
6. The chaincode verifies that the submitting client is the owner, verifies the private details against the hash in the seller's collection, and verifies the bid details against the hash in the buyer's collection. The chaincode then writes the proposed updates for the public key (setting ownership to the buyer, and setting endorsement policy to be the buying organization and regulator), writes the private details to the buyer's private data collection, and potentially deletes the private details from seller's collection. Prior to final endorsement, the endorsing peers ensure private data is disseminated to any other authorized peers of the seller and regulator.
7. The seller submits the transaction with the public data and private data hashes for ordering, and it is distributed to all channel peers in a block.
8. Each peer's block validation logic will consistently verify the endorsement policy was met (buyer, seller, regulator all endorsed), and verify that public and private state that was read in the chaincode has not been modified by any other transaction since chaincode execution.
9. All peers commit the transaction as valid since it passed validation checks. Buyer peers and regulator peers retrieve the private data from other authorized peers if they did not receive it at endorsement time, and persist the private data in their private data state database (assuming the private data matched the hashes from the transaction).
10. With the transaction completed, the asset has been transferred, and other channel members interested in the asset may query the history of the public key to understand its provenance, but will not have access to any private details unless an owner shares it on a need-to-know basis.

The basic asset transfer scenario could be extended for other considerations, for example the transfer chaincode could verify that a payment record is available to satisfy payment versus delivery requirements, or verify that a bank has submitted a letter of credit, prior to the execution of the transfer chaincode. And instead of transactors directly hosting peers, they could transact through custodian organizations who are running peers.

4.12.6 Purging private data

For very sensitive data, even the parties sharing the private data might want — or might be required by government regulations — to periodically "purge" the data on their peers, leaving behind a hash of the data on the blockchain to serve as immutable evidence of the private data.

In some of these cases, the private data only needs to exist on the peer's private database until it can be replicated into a database external to the peer's blockchain. The data might also only need to exist on the peers until a chaincode business process is done with it (trade settled, contract fulfilled, etc).

To support these use cases, private data can be purged if it has not been modified for a configurable number of blocks. Purged private data cannot be queried from chaincode, and is not available to other requesting peers.

4.12.7 How a private data collection is defined

For more details on collection definitions, and other low level information about private data and collections, refer to the private data reference topic.

4.13 Channel capabilities

Audience: Channel administrators, node administrators

Note: this is an advanced Fabric concept that is not necessary for new users or application developers to understand. However, as channels and networks mature, understanding and managing capabilities becomes vital. Furthermore, it is important to recognize that updating capabilities is a different, though often related, process to upgrading nodes. We'll describe this in detail in this topic.

Because Fabric is a distributed system that will usually involve multiple organizations, it is possible (and typical) that different versions of Fabric code will exist on different nodes within the network as well as on the channels in that network. Fabric allows this — it is not necessary for every peer and ordering node to be at the same version level. In fact, supporting different version levels is what enables rolling upgrades of Fabric nodes.

What **is** important is that networks and channels process things in the same way, creating deterministic results for things like channel configuration updates and chaincode invocations. Without deterministic results, one peer on a channel might invalidate a transaction while another peer may validate it.

To that end, Fabric defines levels of what are called "capabilities". These capabilities, which are defined in the configuration of each channel, ensure determinism by defining a level at which behaviors produce consistent results. As you'll see, these capabilities have versions which are closely related to node binary versions. Capabilities enable nodes running at different version levels to behave in a compatible and consistent way given the channel configuration at a specific block height. You will also see that capabilities exist in many parts of the configuration tree, defined along the lines of administration for particular tasks.

As you'll see, sometimes it is necessary to update your channel to a new capability level to enable a new feature.

4.13.1 Node versions and capability versions

If you're familiar with Hyperledger Fabric, you're aware that it follows a typical versioning pattern: v1.1, v1.2.1, v2.0, etc. These versions refer to releases and their related binary versions.

Capabilities follow the same versioning convention. There are v1.1 capabilities and v1.2 capabilities and 2.0 capabilities and so on. But it's important to note a few distinctions.

- **There is not necessarily a new capability level with each release**. The need to establish a new capability is determined on a case by case basis and relies chiefly on the backwards compatibility of new features and older binary versions. Adding Raft ordering services in v1.4.1, for example, did not change the way either transactions or ordering service functions were handled and thus did not require the establishment of any new capabilities. Private Data, on the other hand, could not be handled by peers before v1.2, requiring the establishment of a v1.2 capability level. Because not every release contains a new feature (or a bug fix) that changes the way transactions are processed, certain releases will not require any new capabilities (for example, v1.4) while others will only have new capabilities at particular levels (such as v1.2 and v1.3). We'll discuss the "levels" of capabilities and where they reside in the configuration tree later.
- **Nodes must be at least at the level of certain capabilities in a channel**. When a peer joins a channel, it reads all of the blocks in the ledger sequentially, starting with the genesis block of the channel and continuing through the transaction blocks and any subsequent configuration blocks. If a node, for example a peer, attempts to read a block containing an update to a capability it doesn't understand (for example, a v1.4.x peer trying to read a block containing a v2.0 application capability), **the peer will crash**. This crashing behavior is intentional, as

a v1.4.x peer should not attempt validate or commit any transactions past this point. Before joining a channel, **make sure the node is at least the Fabric version (binary) level of the capabilities specified in the channel config relevant to the node**. We'll discuss which capabilities are relevant to which nodes later. However, because no user wants their nodes to crash, it is strongly recommended to update all nodes to the required level (preferably, to the latest release) before attempting to update capabilities. This is in line with the default Fabric recommendation to **always** be at the latest binary and capability levels.

If users are unable to upgrade their binaries, then capabilities must be left at their lower levels. Lower level binaries and capabilities will still work together as they're meant to. However, keep in mind that it is a best practice to always update to new binaries even if a user chooses not to update their capabilities. Because capabilities themselves also include bug-fixes, it is always recommended to update capabilities once the network binaries support them.

4.13.2 Capability configuration groupings

As we discussed earlier, there is not a single capability level encompassing an entire channel. Rather, there are three capabilities, each representing an area of administration.

- **Orderer**: These capabilities govern tasks and processing exclusive to the ordering service. Because these capabilities do not involve processes that affect transactions or the peers, updating them falls solely to the ordering service admins (peers do not need to understand orderer capabilities and will therefore not crash no matter what the orderer capability is updated to). Note that these capabilities did not change between v1.1 and v1.4.2. However, as we'll see in the **channel** section, this does not mean that v1.1 ordering nodes will work on all channels with capability levels below v1.4.2.
- **Application**: These capabilities govern tasks and processing exclusive to the peers. Because ordering service admins have no role in deciding the nature of transactions between peer organizations, changing this capability level falls exclusively to peer organizations. For example, Private Data can only be enabled on a channel with the v1.2 (or higher) application group capability enabled. In the case of Private Data, this is the only capability that must be enabled, as nothing about the way Private Data works requires a change to channel administration or the way the ordering service processes transactions.
- **Channel**: This grouping encompasses tasks that are **jointly administered** by the peer organizations and the ordering service. For example, this is the capability that defines the level at which channel configuration updates, which are initiated by peer organizations and orchestrated by the ordering service, are processed. On a practical level, **this grouping defines the minimum level for all of the binaries in a channel, as both ordering nodes and peers must be at least at the binary level corresponding to this capability in order to process the capability**.

The **orderer** and **channel** capabilities of a channel are inherited by default from the ordering system channel, where modifying them are the exclusive purview of ordering service admins. As a result, peer organizations should inspect the genesis block of a channel prior to joining their peers to that channel. Although the channel capability is administered by the orderers in the orderer system channel (just as the consortium membership is), it is typical and expected that the ordering admins will coordinate with the consortium admins to ensure that the channel capability is only upgraded when the consortium is ready for it.

Because the ordering system channel does not define an **application** capability, this capability must be specified in the channel profile when creating the genesis block for the channel.

Take caution when specifying or modifying an application capability. Because the ordering service does not validate that the capability level exists, it will allow a channel to be created (or modified) to contain, for example, a v1.8 application capability even if no such capability exists. Any peer attempting to read a configuration block with this capability would, as we have shown, crash, and even if it was possible to modify the channel once again to a valid capability level, it would not matter, as no peer would be able to get past the block with the invalid v1.8 capability.

For a full look at the current valid orderer, application, and channel capabilities check out a sample `configtx.yaml` file, which lists them in the "Capabilities" section.

For more specific information about capabilities and where they reside in the channel configuration, check out defining capability requirements.

4.14 Use Cases

The Hyperledger Requirements WG is documenting a number of blockchain use cases and maintaining an inventory here.

CHAPTER 5

Getting Started

5.1 Prerequisites

Before we begin, if you haven't already done so, you may wish to check that you have all the prerequisites below installed on the platform(s) on which you'll be developing blockchain applications and/or operating Hyperledger Fabric.

5.1.1 Install Git

Download the latest version of git if it is not already installed, or if you have problems running the curl commands.

5.1.2 Install cURL

Download the latest version of the cURL tool if it is not already installed or if you get errors running the curl commands from the documentation.

Note: If you're on Windows please see the specific note on *Windows extras* below.

5.1.3 Docker and Docker Compose

You will need the following installed on the platform on which you will be operating, or developing on (or for), Hyperledger Fabric:

- MacOSX, *nix, or Windows 10: Docker Docker version 17.06.2-ce or greater is required.
- Older versions of Windows: Docker Toolbox - again, Docker version Docker 17.06.2-ce or greater is required.

You can check the version of Docker you have installed with the following command from a terminal prompt:

```
docker --version
```

Note: The following applies to linux systems running systemd.

Make sure the docker daemon is running.

```
sudo systemctl start docker
```

Optional: If you want the docker daemon to start when the system starts, use the following:

```
sudo systemctl enable docker
```

Add your user to the docker group.

```
sudo usermod -a -G docker <username>
```

Note: Installing Docker for Mac or Windows, or Docker Toolbox will also install Docker Compose. If you already had Docker installed, you should check that you have Docker Compose version 1.14.0 or greater installed. If not, we recommend that you install a more recent version of Docker.

You can check the version of Docker Compose you have installed with the following command from a terminal prompt:

```
docker-compose --version
```

5.1.4 Go Programming Language

Hyperledger Fabric uses the Go Programming Language for many of its components.

- Go version 1.14.x is required.

Given that we will be writing chaincode programs in Go, there are two environment variables you will need to set properly; you can make these settings permanent by placing them in the appropriate startup file, such as your personal `~/.bashrc` file if you are using the `bash` shell under Linux.

First, you must set the environment variable `GOPATH` to point at the Go workspace containing the downloaded Fabric code base, with something like:

```
export GOPATH=$HOME/go
```

Note: You **must** set the GOPATH variable

Even though, in Linux, Go's `GOPATH` variable can be a colon-separated list of directories, and will use a default value of `$HOME/go` if it is unset, the current Fabric build framework still requires you to set and export that variable, and it must contain **only** the single directory name for your Go workspace. (This restriction might be removed in a future release.)

Second, you should (again, in the appropriate startup file) extend your command search path to include the Go `bin` directory, such as the following example for `bash` under Linux:

```
export PATH=$PATH:$GOPATH/bin
```

While this directory may not exist in a new Go workspace installation, it is populated later by the Fabric build system with a small number of Go executables used by other parts of the build system. So even if you currently have no such directory yet, extend your shell search path as above.

5.1.5 Node.js Runtime and NPM

If you will be developing applications for Hyperledger Fabric leveraging the Hyperledger Fabric SDK for Node.js, version 8 is supported from 8.9.4 and higher. Node.js version 10 is supported from 10.15.3 and higher.

- Node.js download

Note: Installing Node.js will also install NPM, however it is recommended that you confirm the version of NPM installed. You can upgrade the `npm` tool with the following command:

```
npm install npm@5.6.0 -g
```

Python

Note: The following applies to Ubuntu 16.04 users only.

By default Ubuntu 16.04 comes with Python 3.5.1 installed as the `python3` binary. The Fabric Node.js SDK requires an iteration of Python 2.7 in order for `npm install` operations to complete successfully. Retrieve the 2.7 version with the following command:

```
sudo apt-get install python
```

Check your version(s):

```
python --version
```

5.1.6 Windows extras

If you are developing on Windows 7, you will want to work within the Docker Quickstart Terminal. However, by default it uses an old Git Bash and experience has shown this to be a poor development environment with limited functionality. It is suitable to run Docker based scenarios, such as *Getting Started*, but you will have difficulties with operations involving the `make` and `docker` commands.

Instead, it is recommended to use the MSYS2 environment and run make and docker from the MSYS2 command shell. To do so, install MSYS2 (along with the base developer toolchain and gcc packages using pacman) and launch Docker Toolbox from the MSYS2 shell with the following command:

```
/c/Program\ Files/Docker\ Toolbox/start.sh
```

Alternatively, you can change the Docker Quickstart Terminal command to use MSYS2 bash by changing the target of the Windows shortcut from:

```
"C:\Program Files\Git\bin\bash.exe" --login -i "C:\Program Files\Docker Toolbox\start.
↪sh"
```

to:

```
"C:\msys64\usr\bin\bash.exe" --login -i "C:\Program Files\Docker Toolbox\start.sh"
```

With the above change, you can now simply launch the Docker Quickstart Terminal and get a suitable environment.

On Windows 10 you should use the native Docker distribution and you may use the Windows PowerShell. However, for the `binaries` command to succeed you will still need to have the `uname` command available. You can get it as part of Git but beware that only the 64bit version is supported.

Before running any `git clone` commands, run the following commands:

```
git config --global core.autocrlf false
git config --global core.longpaths true
```

You can check the setting of these parameters with the following commands:

```
git config --get core.autocrlf
git config --get core.longpaths
```

These need to be `false` and `true` respectively.

The `curl` command that comes with Git and Docker Toolbox is old and does not handle properly the redirect used in *Getting Started*. Make sure you have and use a newer version which can be downloaded from the cURL downloads page

For Node.js you also need the necessary Visual Studio C++ Build Tools which are freely available and can be installed with the following command:

```
npm install --global windows-build-tools
```

See the NPM windows-build-tools page for more details.

Once this is done, you should also install the NPM GRPC module with the following command:

```
npm install --global grpc
```

Your environment should now be ready to go through the *Getting Started* samples and tutorials.

Note: If you have questions not addressed by this documentation, or run into issues with any of the tutorials, please visit the *Still Have Questions?* page for some tips on where to find additional help.

5.2 Install Samples, Binaries, and Docker Images

While we work on developing real installers for the Hyperledger Fabric binaries, we provide a script that will download and install samples and binaries to your system. We think that you'll find the sample applications installed useful to learn more about the capabilities and operations of Hyperledger Fabric.

Note: If you are running on **Windows** you will want to make use of the Docker Quickstart Terminal for the upcoming terminal commands. Please visit the *Prerequisites* if you haven't previously installed it.

If you are using Docker Toolbox on Windows 7 or macOS, you will need to use a location under `C:\Users` (Windows 7) or `/Users` (macOS) when installing and running the samples.

If you are using Docker for Mac, you will need to use a location under `/Users`, `/Volumes`, `/private`, or `/tmp`. To use a different location, please consult the Docker documentation for file sharing.

If you are using Docker for Windows, please consult the Docker documentation for shared drives and use a location under one of the shared drives.

Determine a location on your machine where you want to place the *fabric-samples* repository and enter that directory in a terminal window. The command that follows will perform the following steps:

1. If needed, clone the hyperledger/fabric-samples repository
2. Checkout the appropriate version tag
3. Install the Hyperledger Fabric platform-specific binaries and config files for the version specified into the /bin and /config directories of fabric-samples
4. Download the Hyperledger Fabric docker images for the version specified

Once you are ready, and in the directory into which you will install the Fabric Samples and binaries, go ahead and execute the command to pull down the binaries and images.

Note: If you want the latest production release, omit all version identifiers.

```
curl -sSL https://bit.ly/2ysbOFE | bash -s
```

Note: If you want a specific release, pass a version identifier for Fabric, Fabric-ca and thirdparty Docker images. The command below demonstrates how to download the latest production releases - **Fabric v2.1.0** and **Fabric CA v1.4.6**

```
curl -sSL https://bit.ly/2ysbOFE | bash -s -- <fabric_version> <fabric-ca_version>
↪<thirdparty_version>
curl -sSL https://bit.ly/2ysbOFE | bash -s -- 2.1.0 1.4.6 0.4.18
```

Note: If you get an error running the above curl command, you may have too old a version of curl that does not handle redirects or an unsupported environment.

Please visit the *Prerequisites* page for additional information on where to find the latest version of curl and get the right environment. Alternately, you can substitute the un-shortened URL: https://raw.githubusercontent.com/hyperledger/fabric/master/scripts/bootstrap.sh

The command above downloads and executes a bash script that will download and extract all of the platform-specific binaries you will need to set up your network and place them into the cloned repo you created above. It retrieves the following platform-specific binaries:

- `configtxgen`,
- `configtxlator`,
- `cryptogen`,
- `discover`,
- `idemixgen`
- `orderer`,
- `peer`,
- `fabric-ca-client`

and places them in the `bin` sub-directory of the current working directory.

You may want to add that to your PATH environment variable so that these can be picked up without fully qualifying the path to each binary. e.g.:

```
export PATH=<path to download location>/bin:$PATH
```

Finally, the script will download the Hyperledger Fabric docker images from Docker Hub into your local Docker registry and tag them as 'latest'.

The script lists out the Docker images installed upon conclusion.

Look at the names for each image; these are the components that will ultimately comprise our Hyperledger Fabric network. You will also notice that you have two instances of the same image ID - one tagged as "amd64-1.x.x" and one tagged as "latest". Prior to 1.2.0, the image being downloaded was determined by `uname -m` and showed as "x86_64-1.x.x".

Note: On different architectures, the x86_64/amd64 would be replaced with the string identifying your architecture.

Note: If you have questions not addressed by this documentation, or run into issues with any of the tutorials, please visit the *Still Have Questions?* page for some tips on where to find additional help.

5.3 Using the Fabric test network

After you have downloaded the Hyperledger Fabric Docker images and samples, you can deploy a test network by using scripts that are provided in the `fabric-samples` repository. You can use the test network to learn about Fabric by running nodes on your local machine. More experienced developers can use the network to test their smart contracts and applications. The network is meant to be used only as a tool for education and testing. It should not be used as a template for deploying a production network. The test network is being introduced in Fabric v2.0 as the long term replacement for the `first-network` sample.

The sample network deploys a Fabric network with Docker Compose. Because the nodes are isolated within a Docker Compose network, the test network is not configured to connect to other running fabric nodes.

Note: These instructions have been verified to work against the latest stable Docker images and the pre-compiled setup utilities within the supplied tar file. If you run these commands with images or tools from the current master branch, it is possible that you will encounter errors.

5.3.1 Before you begin

Before you can run the test network, you need to clone the `fabric-samples` repository and download the Fabric images. Make sure that that you have installed the Prerequisites and Installed the Samples, Binaries and Docker Images.

5.3.2 Bring up the test network

You can find the scripts to bring up the network in the `test-network` directory of the `fabric-samples` repository. Navigate to the test network directory by using the following command:

```
cd fabric-samples/test-network
```

In this directory, you can find an annotated script, `network.sh`, that stands up a Fabric network using the Docker images on your local machine. You can run `./network.sh -h` to print the script help text:

```
Usage:
  network.sh <Mode> [Flags]
    <Mode>
      - 'up' - bring up fabric orderer and peer nodes. No channel is created
      - 'up createChannel' - bring up fabric network with one channel
      - 'createChannel' - create and join a channel after the network is created
      - 'deployCC' - deploy the fabcar chaincode on the channel
      - 'down' - clear the network with docker-compose down
      - 'restart' - restart the network

    Flags:
    -ca <use CAs> -  create Certificate Authorities to generate the crypto material
    -c <channel name> - channel name to use (defaults to "mychannel")
    -s <dbtype> - the database backend to use: goleveldb (default) or couchdb
    -r <max retry> - CLI times out after certain number of attempts (defaults to 5)
    -d <delay> - delay duration in seconds (defaults to 3)
    -l <language> - the programming language of the chaincode to deploy: go (default),
↪ javascript, or java
    -v <version>  - chaincode version. Must be a round number, 1, 2, 3, etc
    -i <imagetag> - the tag to be used to launch the network (defaults to "latest")
    -verbose - verbose mode
  network.sh -h (print this message)

 Possible Mode and flags
  network.sh up -ca -c -r -d -s -i -verbose
  network.sh up createChannel -ca -c -r -d -s -i -verbose
  network.sh createChannel -c -r -d -verbose
  network.sh deployCC -l -v -r -d -verbose

 Taking all defaults:
    network.sh up

 Examples:
  network.sh up createChannel -ca -c mychannel -s couchdb -i 2.0.0
  network.sh createChannel -c channelName
  network.sh deployCC -l javascript
```

From inside the `test-network` directory, run the following command to remove any containers or artifacts from any previous runs:

```
./network.sh down
```

You can then bring up the network by issuing the following command. You will experience problems if you try to run the script from another directory:

```
./network.sh up
```

This command creates a Fabric network that consists of two peer nodes, one ordering node. No channel is created when you run `./network.sh up`, though we will get there in a *future step*. If the command completes successfully, you will see the logs of the nodes being created:

```
Creating network "net_test" with the default driver
Creating volume "net_orderer.example.com" with default driver
Creating volume "net_peer0.org1.example.com" with default driver
Creating volume "net_peer0.org2.example.com" with default driver
Creating orderer.example.com    ... done
Creating peer0.org2.example.com ... done
Creating peer0.org1.example.com ... done
CONTAINER ID        IMAGE                               COMMAND             CREATED
↪          STATUS                  PORTS                               NAMES
8d0c74b9d6af        hyperledger/fabric-orderer:latest   "orderer"           4 seconds␣
↪ago       Up Less than a second   0.0.0.0:7050->7050/tcp              orderer.
↪example.com
ea1cf82b5b99        hyperledger/fabric-peer:latest      "peer node start"   4 seconds␣
↪ago       Up Less than a second   0.0.0.0:7051->7051/tcp              peer0.org1.
↪example.com
cd8d9b23cb56        hyperledger/fabric-peer:latest      "peer node start"   4 seconds␣
↪ago       Up 1 second             7051/tcp, 0.0.0.0:9051->9051/tcp    peer0.org2.
↪example.com
```

If you don't get this result, jump down to *Troubleshooting* for help on what might have gone wrong. By default, the network uses the cryptogen tool to bring up the network. However, you can also *bring up the network with Certificate Authorities*.

The components of the test network

After your test network is deployed, you can take some time to examine its components. Run the following command to list all of Docker containers that are running on your machine. You should see the three nodes that were created by the `network.sh` script:

```
docker ps -a
```

Each node and user that interacts with a Fabric network needs to belong to an organization that is a network member. The group of organizations that are members of a Fabric network are often referred to as the consortium. The test network has two consortium members, Org1 and Org2. The network also includes one orderer organization that maintains the ordering service of the network.

Peers are the fundamental components of any Fabric network. Peers store the blockchain ledger and validate transactions before they are committed to the ledger. Peers run the smart contracts that contain the business logic that is used to manage the assets on the blockchain ledger.

Every peer in the network needs to belong to a member of the consortium. In the test network, each organization operates one peer each, `peer0.org1.example.com` and `peer0.org2.example.com`.

Every Fabric network also includes an ordering service. While peers validate transactions and add blocks of transactions to the blockchain ledger, they do not decide on the order of transactions or include them into new blocks. On a distributed network, peers may be running far away from each other and not have a common view of when a transaction was created. Coming to consensus on the order of transactions is a costly process that would create overhead for the peers.

An ordering service allows peers to focus on validating transactions and committing them to the ledger. After ordering nodes receive endorsed transactions from clients, they come to consensus on the order of transactions and then add them to blocks. The blocks are then distributed to peer nodes, which add the blocks the blockchain ledger. Ordering nodes also operate the system channel that defines the capabilities of a Fabric network, such as how blocks are made and which version of Fabric that nodes can use. The system channel defines which organizations are members of the consortium.

The sample network uses a single node Raft ordering service that is operated by the ordering organization. You can see the ordering node running on your machine as `orderer.example.com`. While the test network only uses a single node ordering service, a real network would have multiple ordering nodes, operated by one or multiple orderer organizations. The different ordering nodes would use the Raft consensus algorithm to come to agreement on the order of transactions across the network.

5.3.3 Creating a channel

Now that we have peer and orderer nodes running on our machine, we can use the script to create a Fabric channel for transactions between Org1 and Org2. Channels are a private layer of communication between specific network members. Channels can be used only by organizations that are invited to the channel, and are invisible to other members of the network. Each channel has a separate blockchain ledger. Organizations that have been invited "join" their peers to the channel to store the channel ledger and validate the transactions on the channel.

You can use the `network.sh` script to create a channel between Org1 and Org2 and join their peers to the channel. Run the following command to create a channel with the default name of `mychannel`:

```
./network.sh createChannel
```

If the command was successful, you can see the following message printed in your logs:

```
========= Channel successfully joined ===========
```

You can also use the channel flag to create a channel with custom name. As an example, the following command would create a channel named `channel1`:

```
./network.sh createChannel -c channel1
```

The channel flag also allows you to create multiple channels by specifying different channel names. After you create `mychannel` or `channel1`, you can use the command below to create a second channel named `channel2`:

```
./network.sh createChannel -c channel2
```

If you want to bring up the network and create a channel in a single step, you can use the `up` and `createChannel` modes together:

```
./network.sh up createChannel
```

5.3.4 Starting a chaincode on the channel

After you have created a channel, you can start using smart contracts to interact with the channel ledger. Smart contracts contain the business logic that governs assets on the blockchain ledger. Applications run by members of the network can invoke smart contracts to create assets on the ledger, as well as change and transfer those assets. Applications also query smart contracts to read data on the ledger.

To ensure that transactions are valid, transactions created using smart contracts typically need to be signed by multiple organizations to be committed to the channel ledger. Multiple signatures are integral to the trust model of Fabric. Requiring multiple endorsements for a transaction prevents one organization on a channel from tampering with the ledger on their peer or using business logic that was not agreed to. To sign a transaction, each organization needs to invoke and execute the smart contract on their peer, which then signs the output of the transaction. If the output is consistent and has been signed by enough organizations, the transaction can be committed to the ledger. The policy that specifies the set organizations on the channel that need to execute the smart contract is referred to as the endorsement policy, which is set for each chaincode as part of the chaincode definition.

In Fabric, smart contracts are deployed on the network in packages referred to as chaincode. A Chaincode is installed on the peers of an organization and then deployed to a channel, where it can then be used to endorse transactions and interact with the blockchain ledger. Before a chaincode can be deployed to a channel, the members of the channel need to agree on a chaincode definition that establishes chaincode governance. When the required number of organizations agree, the chaincode definition can be committed to the channel, and the chaincode is ready to be used.

After you have used the `network.sh` to create a channel, you can start a chaincode on the channel using the following command:

```
./network.sh deployCC
```

The `deployCC` subcommand will install the **fabcar** chaincode on `peer0.org1.example.com` and `peer0.org2.example.com` and then deploy the chaincode on the channel specified using the channel flag (or `mychannel` if no channel is specified). If you are deploying a chaincode for the first time, the script will install the chaincode dependencies. By default, The script installs the Go version of the fabcar chaincode. However, you can use the language flag, `-l`, to install the Java or javascript versions of the chaincode. You can find the Fabcar chaincode in the `chaincode` folder of the `fabric-samples` directory. This folder contains sample chaincode that are provided as examples and used by tutorials to highlight Fabric features.

After the **fabcar** chaincode definition has been committed to the channel, the script initializes the chaincode by invoking the `init` function and then invokes the chaincode to put an initial list of cars on the ledger. The script then queries the chaincode to verify the that the data was added. If the chaincode was installed, deployed, and invoked correctly, you should see the following list of cars printed in your logs:

```
[{"Key":"CAR0", "Record":{"make":"Toyota","model":"Prius","colour":"blue","owner":
↪"Tomoko"}},
{"Key":"CAR1", "Record":{"make":"Ford","model":"Mustang","colour":"red","owner":"Brad
↪"}},
{"Key":"CAR2", "Record":{"make":"Hyundai","model":"Tucson","colour":"green","owner":
↪"Jin Soo"}},
{"Key":"CAR3", "Record":{"make":"Volkswagen","model":"Passat","colour":"yellow","owner
↪":"Max"}},
{"Key":"CAR4", "Record":{"make":"Tesla","model":"S","colour":"black","owner":"Adriana
↪"}},
{"Key":"CAR5", "Record":{"make":"Peugeot","model":"205","colour":"purple","owner":
↪"Michel"}},
{"Key":"CAR6", "Record":{"make":"Chery","model":"S22L","colour":"white","owner":"Aarav
↪"}},
{"Key":"CAR7", "Record":{"make":"Fiat","model":"Punto","colour":"violet","owner":"Pari
↪"}},
{"Key":"CAR8", "Record":{"make":"Tata","model":"Nano","colour":"indigo","owner":
↪"Valeria"}},
{"Key":"CAR9", "Record":{"make":"Holden","model":"Barina","colour":"brown","owner":
↪"Shotaro"}}]
===================== Query successful on peer0.org1 on channel 'mychannel'␣
↪=====================
```

5.3.5 Interacting with the network

After you bring up the test network, you can use the `peer` CLI to interact with your network. The `peer` CLI allows you to invoke deployed smart contracts, update channels, or install and deploy new smart contracts from the CLI.

Make sure that you are operating from the `test-network` directory. If you followed the instructions to install the Samples, Binaries and Docker Images, You can find the `peer` binaries in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

```
export PATH=${PWD}/../bin:${PWD}:$PATH
```

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

You can now set the environment variables that allow you to operate the `peer` CLI as Org1:

```
# Environment variables for Org1

export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

The `CORE_PEER_TLS_ROOTCERT_FILE` and `CORE_PEER_MSPCONFIGPATH` environment variables point to the Org1 crypto material in the `organizations` folder.

If you used `./network.sh deployCC` to install and start the fabcar chaincode, you can now query the ledger from your CLI. Run the following command to get the list of cars that were added to your channel ledger:

```
peer chaincode query -C mychannel -n fabcar -c '{"Args":["queryAllCars"]}'
```

If the command is successful, you can see the same list of cars that were printed in the logs when you ran the script:

```
[{"Key":"CAR0", "Record":{"make":"Toyota","model":"Prius","colour":"blue","owner":
↪"Tomoko"}},{"Key":"CAR1", "Record":{"make":"Ford","model":"Mustang","colour":"red",
↪"owner":"Brad"}},{"Key":"CAR2", "Record":{"make":"Hyundai","model":"Tucson","colour
↪":"green","owner":"Jin Soo"}},{"Key":"CAR3", "Record":{"make":"Volkswagen","model":
↪"Passat","colour":"yellow","owner":"Max"}},{"Key":"CAR4", "Record":{"make":"Tesla",
↪"model":"S","colour":"black","owner":"Adriana"}},{"Key":"CAR5", "Record":{"make":
↪"Peugeot","model":"205","colour":"purple","owner":"Michel"}},{"Key":"CAR6", "Record
↪":{"make":"Chery","model":"S22L","colour":"white","owner":"Aarav"}},{"Key":"CAR7",
↪"Record":{"make":"Fiat","model":"Punto","colour":"violet","owner":"Pari"}},{"Key":
↪"CAR8", "Record":{"make":"Tata","model":"Nano","colour":"indigo","owner":"Valeria"}}
↪,{"Key":"CAR9", "Record":{"make":"Holden","model":"Barina","colour":"brown","owner":
↪"Shotaro"}}]
```

Chaincodes are invoked when a network member wants to transfer or change an asset on the ledger. Use the following command to change the owner of a car on the ledger by invoking the fabcar chaincode:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls true --cafile ${PWD}/organizations/ordererOrganizations/example.com/
↪orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel␣
↪-n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"function":"changeCarOwner","Args":["CAR9","Dave"]}'
```

If the command is successful, you should see the following response:

```
2019-12-04 17:38:21.048 EST [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001␣
↪Chaincode invoke successful. result: status:200
```

Because the endorsement policy for the fabcar chaincode requires the transaction to be signed by Org1 and Org2, the chaincode invoke command needs to target both `peer0.org1.example.com` and `peer0.org2.example.com` using the `--peerAddresses` flag. Because TLS is enabled for the network, the command also needs to reference the TLS certificate for each peer using the `--tlsRootCertFiles` flag.

After we invoke the chaincode, we can use another query to see how the invoke changed the assets on the blockchain ledger. Since we already queried the Org1 peer, we can take this opportunity to query the chaincode running on the Org2 peer. Set the following environment variables to operate as Org2:

```
# Environment variables for Org2

export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

You can now query the fabcar chaincode running on `peer0.org2.example.com`:

```
peer chaincode query -C mychannel -n fabcar -c '{"Args":["queryAllCars"]}'
```

The result will show that `"CAR9"` was transferred to Dave:

```
[{"Key":"CAR0", "Record":{"make":"Toyota","model":"Prius","colour":"blue","owner":
↪"Tomoko"}},{"Key":"CAR1", "Record":{"make":"Ford","model":"Mustang","colour":"red",
↪"owner":"Brad"}},{"Key":"CAR2", "Record":{"make":"Hyundai","model":"Tucson","colour
↪":"green","owner":"Jin Soo"}},{"Key":"CAR3", "Record":{"make":"Volkswagen","model":
↪"Passat","colour":"yellow","owner":"Max"}},{"Key":"CAR4", "Record":{"make":"Tesla",
↪"model":"S","colour":"black","owner":"Adriana"}},{"Key":"CAR5", "Record":{"make":
↪"Peugeot","model":"205","colour":"purple","owner":"Michel"}},{"Key":"CAR6", "Record
↪":{"make":"Chery","model":"S22L","colour":"white","owner":"Aarav"}},{"Key":"CAR7",
↪"Record":{"make":"Fiat","model":"Punto","colour":"violet","owner":"Pari"}},{"Key":
↪"CAR8", "Record":{"make":"Tata","model":"Nano","colour":"indigo","owner":"Valeria"}}
↪,{"Key":"CAR9", "Record":{"make":"Holden","model":"Barina","colour":"brown","owner":
↪"Dave"}}]
```

5.3.6 Bring down the network

When you are finished using the test network, you can bring down the network with the following command:

```
./network.sh down
```

The command will stop and remove the node and chaincode containers, delete the organization crypto material, and remove the chaincode images from your Docker Registry. The command also removes the channel artifacts and docker volumes from previous runs, allowing you to run `./network.sh up` again if you encountered any problems.

5.3.7 Bring up the network with Certificate Authorities

Hyperledger Fabric uses public key infrastructure (PKI) to verify the actions of all network participants. Every node, network administrator, and user submitting transactions needs to have a public certificate and private key to verify their identity. These identities need to have a valid root of trust, establishing that the certificates were issued by an organization that is a member of the network. The `network.sh` script creates all of the cryptographic material that is required to deploy and operate the network before it creates the peer and ordering nodes.

By default, the script uses the cryptogen tool to create the certificates and keys. The tool is provided for development and testing, and can quickly create the required crypto material for Fabric organizations with a valid root of trust. When you run `./network.sh up`, you can see the cryptogen tool creating the certificates and keys for Org1, Org2, and the Orderer Org.

```
creating Org1, Org2, and ordering service organization with crypto from 'cryptogen'

/Usr/fabric-samples/test-network/../bin/cryptogen

##########################################################
##### Generate certificates using cryptogen tool #########
##########################################################

##########################################################
############ Create Org1 Identities ######################
##########################################################
+ cryptogen generate --config=./organizations/cryptogen/crypto-config-org1.yaml --
↪output=organizations
org1.example.com
+ res=0
+ set +x
##########################################################
############ Create Org2 Identities ######################
##########################################################
+ cryptogen generate --config=./organizations/cryptogen/crypto-config-org2.yaml --
↪output=organizations
org2.example.com
+ res=0
+ set +x
##########################################################
############ Create Orderer Org Identities ###############
##########################################################
+ cryptogen generate --config=./organizations/cryptogen/crypto-config-orderer.yaml --
↪output=organizations
+ res=0
+ set +x
```

However, the test network script also provides the option to bring up the network using Certificate Authorities (CAs). In a production network, each organization operates a CA (or multiple intermediate CAs) that creates the identities that belong to their organization. All of the identities created by a CA run by the organization share the same root of trust. Although it takes more time than using cryptogen, bringing up the test network using CAs provides an introduction to how a network is deployed in production. Deploying CAs also allows you to enroll client identities with the Fabric SDKs and create a certificate and private key for your applications.

If you would like to bring up a network using Fabric CAs, first run the following command to bring down any running networks:

```
./network.sh down
```

You can then bring up the network with the CA flag:

```
./network.sh up -ca
```

After you issue the command, you can see the script bringing up three CAs, one for each organization in the network.

```
##########################################################
##### Generate certificates using Fabric CA's ############
```

(continues on next page)

(continued from previous page)

```
############################################################
Creating network "net_default" with the default driver
Creating ca_org2    ... done
Creating ca_org1    ... done
Creating ca_orderer ... done
```

It is worth taking time to examine the logs generated by the `./network.sh` script after the CAs have been deployed. The test network uses the Fabric CA client to register node and user identities with the CA of each organization. The script then uses the enroll command to generate an MSP folder for each identity. The MSP folder contains the certificate and private key for each identity, and establishes the identity's role and membership in the organization that operated the CA. You can use the following command to examine the MSP folder of the Org1 admin user:

```
tree organizations/peerOrganizations/org1.example.com/users/Admin/@org1.example.com/
```

The command will reveal the MSP folder structure and configuration file:

```
organizations/peerOrganizations/org1.example.com/users/Admin@org1.example.com/
└── msp
    ├── IssuerPublicKey
    ├── IssuerRevocationPublicKey
    ├── cacerts
    │   └── localhost-7054-ca-org1.pem
    ├── config.yaml
    ├── keystore
    │   └── 58e81e6f1ee8930df46841bf88c22a08ae53c1332319854608539ee78ed2fd65_sk
    ├── signcerts
    │   └── cert.pem
    └── user
```

You can learn more about the contents of MSP folders by going to the Membership Service Provider concept topic.

Both cryptogen and the Fabric CAs generate the cryptographic material for each organization in the `organizations` folder. You can find the commands that are used to set up the network in the `registerEnroll.sh` script in the `organizations/fabric-ca` directory. To learn more about how you would use the Fabric CA to deploy a Fabric network, visit the Fabric CA operations guide. You can learn more about how Fabric uses PKI by visiting the identity and membership concept topics.

5.3.8 What's happening behind the scenes?

If you are interested in learning more about the sample network, you can investigate the files and scripts in the `test-network` directory. The steps below provide a guided tour of what happens when you issue the command of `./network.sh up`.

- `./network.sh` creates the certificates and keys for two peer organizations and the orderer organization. By default, the script uses the cryptogen tool using the configuration files located in the `organizations/cryptogen` folder. If you use the `-ca` flag to create Certificate Authorities, the script uses Fabric CA server configuration files and `registerEnroll.sh` script located in the `organizations/fabric-ca` folder. Both cryptogen and the Fabric CAs create the crypto material and MSP folders for all three organizations in the `organizations` folder.
- The script uses configtxgen tool to create the system channel genesis block. Configtxgen consumes the `TwoOrgsOrdererGenesis` channel profile in the `configtx/configtx.yaml` file to create the genesis block. The block is stored in the `system-genesis-block` folder.
- Once the organization crypto material and the system channel genesis block have been generated, the `network.sh` can bring up the nodes of the network. The script uses the `docker-compose-test-net.`

yaml file in the docker folder to create the peer and orderer nodes. The docker folder also contains the docker-compose-e2e.yaml file that brings up the nodes of the network alongside three Fabric CAs. This file is meant to be used to run end-to-end tests by the Fabric SDK. Refer to the Node SDK repo for details on running these tests.

- If you use the createChannel subcommand, ./network.sh runs the createChannel.sh script in the scripts folder to create a channel using the supplied channel name. The script uses the configtx.yaml file to create the channel creation transaction, as well as two anchor peer update transactions. The script uses the peer cli to create the channel, join peer0.org1.example.com and peer0.org2.example.com to the channel, and make both of the peers anchor peers.
- If you issue the deployCC command, ./network.sh runs the deployCC.sh script to install the **fabcar** chaincode on both peers and then define then chaincode on the channel. Once the chaincode definition is committed to the channel, the peer cli initializes the chainocde using the Init and invokes the chaincode to put initial data on the ledger.

5.3.9 Troubleshooting

If you have any problems with the tutorial, review the following:

- You should always start your network fresh. You can use the following command to remove the artifacts, crypto material, containers, volumes, and chaincode images from previous runs:

```
./network.sh down
```

  You **will** see errors if you do not remove old containers, images, and volumes.

- If you see Docker errors, first check your Docker version (Prerequisites), and then try restarting your Docker process. Problems with Docker are oftentimes not immediately recognizable. For example, you may see errors that are the result of your node not being able to access the crypto material mounted within a container.

  If problems persist, you can remove your images and start from scratch:

```
docker rm -f $(docker ps -aq)
docker rmi -f $(docker images -q)
```

- If you see errors on your create, approve, commit, invoke or query commands, make sure you have properly updated the channel name and chaincode name. There are placeholder values in the supplied sample commands.
- If you see the error below:

```
Error: Error endorsing chaincode: rpc error: code = 2 desc = Error installing␣
↪chaincode code mycc:1.0(chaincode /var/hyperledger/production/chaincodes/mycc.1.
↪0 exits)
```

  You likely have chaincode images (e.g. dev-peer1.org2.example.com-fabcar-1.0 or dev-peer0.org1.example.com-fabcar-1.0) from prior runs. Remove them and try again.

```
docker rmi -f $(docker images | grep peer[0-9]-peer[0-9] | awk '{print $3}')
```

- If you see the below error:

```
[configtx/tool/localconfig] Load -> CRIT 002 Error reading configuration:␣
↪Unsupported Config Type ""
panic: Error reading configuration: Unsupported Config Type ""
```

Then you did not set the `FABRIC_CFG_PATH` environment variable properly. The configtxgen tool needs this variable in order to locate the configtx.yaml. Go back and execute an `export FABRIC_CFG_PATH=$PWD/configtx/configtx.yaml`, then recreate your channel artifacts.

- If you see an error stating that you still have "active endpoints", then prune your Docker networks. This will wipe your previous networks and start you with a fresh environment:

```
docker network prune
```

You will see the following message:

```
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N]
```

Select `y`.

- If you see an error similar to the following:

```
/bin/bash: ./scripts/createChannel.sh: /bin/bash^M: bad interpreter: No such file␣
↪or directory
```

Ensure that the file in question (**createChannel.sh** in this example) is encoded in the Unix format. This was most likely caused by not setting `core.autocrlf` to `false` in your Git configuration (see Windows extras). There are several ways of fixing this. If you have access to the vim editor for instance, open the file:

```
vim ./fabric-samples/test-network/scripts/createChannel.sh
```

Then change its format by executing the following vim command:

```
:set ff=unix
```

- If your orderer exits upon creation or if you see that the create channel command fails due to an inability to connect to your ordering service, use the `docker logs` command to read the logs from the ordering node. You may see the following message:

```
PANI 007 [channel system-channel] config requires unsupported orderer␣
↪capabilities: Orderer capability V2_0 is required but not supported: Orderer␣
↪capability V2_0 is required but not supported
```

This occurs when you are trying to run the network using Fabric version 1.4.x docker images. The test network needs to run using Fabric version 2.x.

If you continue to see errors, share your logs on the **fabric-questions** channel on Hyperledger Rocket Chat or on StackOverflow.

Before we begin, if you haven't already done so, you may wish to check that you have all the *Prerequisites* installed on the platform(s) on which you'll be developing blockchain applications and/or operating Hyperledger Fabric.

Once you have the prerequisites installed, you are ready to download and install HyperLedger Fabric. While we work on developing real installers for the Fabric binaries, we provide a script that will *Install Samples, Binaries, and Docker Images* to your system. The script also will download the Docker images to your local registry.

After you have downloaded the Fabric Samples and Docker images to your local machine, you can get started working with Fabric with the *Using the Fabric test network* tutorial.

5.4 Hyperledger Fabric smart contract (chaincode) APIs

Hyperledger Fabric offers a number of APIs to support developing smart contracts (chaincode) in various programming languages. Smart contract APIs are available for Go, Node.js, and Java:

- Go contract-api.
- Node.js contract API and Node.js contract API documentation.
- Java contract API and Java contract API documentation.

5.5 Hyperledger Fabric application SDKs

Hyperledger Fabric offers a number of SDKs to support developing applications in various programming languages. SDKs are available for Node.js and Java:

- Node.js SDK and Node.js SDK documentation.
- Java SDK and Java SDK documentation.

In addition, there are two more application SDKs that have not yet been officially released (for Python and Go), but they are still available for downloading and testing:

- Python SDK.
- Go SDK.

Currently, Node.js and Java support the new application programming model delivered in Hyperledger Fabric v1.4. Support for Go is planned to be delivered in a later release.

5.6 Hyperledger Fabric CA

Hyperledger Fabric provides an optional certificate authority service that you may choose to use to generate the certificates and key material to configure and manage identity in your blockchain network. However, any CA that can generate ECDSA certificates may be used.

CHAPTER 6

Developing Applications

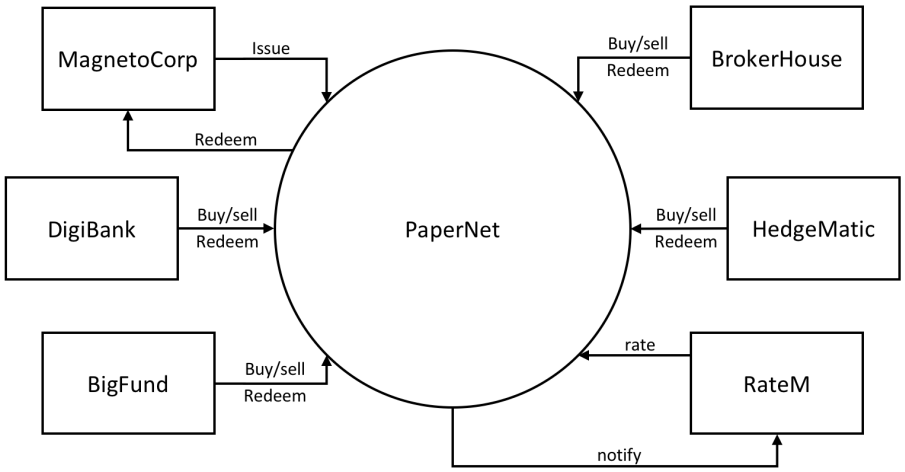
6.1 The scenario

Audience: Architects, Application and smart contract developers, Business professionals

In this topic, we're going to describe a business scenario involving six organizations who use PaperNet, a commercial paper network built on Hyperledger Fabric, to issue, buy and redeem commercial paper. We're going to use the scenario to outline requirements for the development of commercial paper applications and smart contracts used by the participant organizations.

6.1.1 PaperNet network

PaperNet is a commercial paper network that allows suitably authorized participants to issue, trade, redeem and rate commercial paper.



The PaperNet commercial paper network. Six organizations currently use PaperNet network to issue, buy, sell, redeem and rate commercial paper. MagentoCorp issues and redeems commercial paper. DigiBank, BigFund, BrokerHouse and HedgeMatic all trade commercial paper with each other. RateM provides various measures of risk for commercial paper.

Let's see how MagnetoCorp uses PaperNet and commercial paper to help its business.

6.1.2 Introducing the actors

MagnetoCorp is a well-respected company that makes self-driving electric vehicles. In early April 2020, MagnetoCorp won a large order to manufacture 10,000 Model D cars for Daintree, a new entrant in the personal transport market. Although the order represents a significant win for MagnetoCorp, Daintree will not have to pay for the vehicles until they start to be delivered on November 1, six months after the deal was formally agreed between MagnetoCorp and Daintree.

To manufacture the vehicles, MagnetoCorp will need to hire 1000 workers for at least 6 months. This puts a short term strain on its finances – it will require an extra 5M USD each month to pay these new employees. **Commercial paper** is designed to help MagnetoCorp overcome its short term financing needs – to meet payroll every month based on the expectation that it will be cash rich when Daintree starts to pay for its new Model D cars.

At the end of May, MagnetoCorp needs 5M USD to meet payroll for the extra workers it hired on May 1. To do this, it issues a commercial paper with a face value of 5M USD with a maturity date 6 months in the future – when it expects to see cash flow from Daintree. DigiBank thinks that MagnetoCorp is creditworthy, and therefore doesn't require much of a premium above the central bank base rate of 2%, which would value 4.95M USD today at 5M USD in 6 months time. It therefore purchases the MagnetoCorp 6 month commercial paper for 4.94M USD – a slight discount compared to the 4.95M USD it is worth. DigiBank fully expects that it will be able to redeem 5M USD from MagnetoCorp in 6 months time, making it a profit of 10K USD for bearing the increased risk associated with this commercial paper. This extra 10K means it receives a 2.4% return on investment – significantly better than the risk free return of 2%.

At the end of June, when MagnetoCorp issues a new commercial paper for 5M USD to meet June's payroll, it is purchased by BigFund for 4.94M USD. That's because the commercial conditions are roughly the same in June as they are in May, resulting in BigFund valuing MagnetoCorp commercial paper at the same price that DigiBank did in May.

Each subsequent month, MagnetoCorp can issue new commercial paper to meet its payroll obligations, and these may be purchased by DigiBank, or any other participant in the PaperNet commercial paper network – BigFund, HedgeMatic or BrokerHouse. These organizations may pay more or less for the commercial paper depending on two factors – the

central bank base rate, and the risk associated with MagnetoCorp. This latter figure depends on a variety of factors such as the production of Model D cars, and the creditworthiness of MagnetoCorp as assessed by RateM, a ratings agency.

The organizations in PaperNet have different roles, MagnetoCorp issues paper, DigiBank, BigFund, HedgeMatic and BrokerHouse trade paper and RateM rates paper. Organizations of the same role, such as DigiBank, Bigfund, HedgeMatic and BrokerHouse are competitors. Organizations of different roles are not necessarily competitors, yet might still have opposing business interest, for example MagentoCorp will desire a high rating for its papers to sell them at a high price, while DigiBank would benefit from a low rating, such that it can buy them at a low price. As can be seen, even a seemingly simple network such as PaperNet can have complex trust relationships. A blockchain can help establish trust among organizations that are competitors or have opposing business interests that might lead to disputes. Fabric in particular has the means to capture even fine-grained trust relationships.

Let's pause the MagnetoCorp story for a moment, and develop the client applications and smart contracts that PaperNet uses to issue, buy, sell and redeem commercial paper as well as capture the trust relationships between the organizations. We'll come back to the role of the rating agency, RateM, a little later.

6.2 Analysis

Audience: Architects, Application and smart contract developers, Business professionals

Let's analyze commercial paper in a little more detail. PaperNet participants such as MagnetoCorp and DigiBank use commercial paper transactions to achieve their business objectives – let's examine the structure of a commercial paper and the transactions that affect it over time. We will also consider which organizations in PaperNet need to sign off on a transaction based on the trust relationships among the organizations in the network. Later we'll focus on how money flows between buyers and sellers; for now, let's focus on the first paper issued by MagnetoCorp.

6.2.1 Commercial paper lifecycle

A paper 00001 is issued by MagnetoCorp on May 31. Spend a few moments looking at the first **state** of this paper, with its different properties and values:

```
Issuer = MagnetoCorp
Paper = 00001
Owner = MagnetoCorp
Issue date = 31 May 2020
Maturity = 30 November 2020
Face value = 5M USD
Current state = issued
```

This paper state is a result of the **issue** transaction and it brings MagnetoCorp's first commercial paper into existence! Notice how this paper has a 5M USD face value for redemption later in the year. See how the `Issuer` and `Owner` are the same when paper 00001 is issued. Notice that this paper could be uniquely identified as `MagnetoCorp00001` – a composition of the `Issuer` and `Paper` properties. Finally, see how the property `Current state = issued` quickly identifies the stage of MagnetoCorp paper 00001 in its lifecycle.

Shortly after issuance, the paper is bought by DigiBank. Spend a few moments looking at how the same commercial paper has changed as a result of this **buy** transaction:

```
Issuer = MagnetoCorp
Paper = 00001
Owner = DigiBank
Issue date = 31 May 2020
Maturity date = 30 November 2020
```

(continues on next page)

(continued from previous page)

```
Face value = 5M USD
Current state = trading
```

The most significant change is that of `Owner` – see how the paper initially owned by `MagnetoCorp` is now owned by `DigiBank`. We could imagine how the paper might be subsequently sold to BrokerHouse or HedgeMatic, and the corresponding change to `Owner`. Note how `Current state` allow us to easily identify that the paper is now `trading`.

After 6 months, if DigiBank still holds the commercial paper, it can redeem it with MagnetoCorp:

```
Issuer = MagnetoCorp
Paper = 00001
Owner = MagnetoCorp
Issue date = 31 May 2020
Maturity date = 30 November 2020
Face value = 5M USD
Current state = redeemed
```

This final **redeem** transaction has ended the commercial paper's lifecycle – it can be considered closed. It is often mandatory to keep a record of redeemed commercial papers, and the `redeemed` state allows us to quickly identify these. The value of `Owner` of a paper can be used to perform access control on the **redeem** transaction, by comparing the `Owner` against the identity of the transaction creator. Fabric supports this through the `getCreator()` chaincode API. If Go is used as a chaincode language, the client identity chaincode library can be used to retrieve additional attributes of the transaction creator.

6.2.2 Transactions

We've seen that paper 00001's lifecycle is relatively straightforward – it moves between `issued`, `trading` and `redeemed` as a result of an **issue**, **buy**, or **redeem** transaction.

These three transactions are initiated by MagnetoCorp and DigiBank (twice), and drive the state changes of paper 00001. Let's have a look at the transactions that affect this paper in a little more detail:

Issue

Examine the first transaction initiated by MagnetoCorp:

```
Txn = issue
Issuer = MagnetoCorp
Paper = 00001
Issue time = 31 May 2020 09:00:00 EST
Maturity date = 30 November 2020
Face value = 5M USD
```

See how the **issue** transaction has a structure with properties and values. This transaction structure is different to, but closely matches, the structure of paper 00001. That's because they are different things – paper 00001 reflects a state of PaperNet that is a result of the **issue** transaction. It's the logic behind the **issue** transaction (which we cannot see) that takes these properties and creates this paper. Because the transaction **creates** the paper, it means there's a very close relationship between these structures.

The only organization that is involved in the **issue** transaction is MagnetoCorp. Naturally, MagnetoCorp needs to sign off on the transaction. In general, the issuer of a paper is required to sign off on a transaction that issues a new paper.

Buy

Next, examine the **buy** transaction which transfers ownership of paper 00001 from MagnetoCorp to DigiBank:

```
Txn = buy
Issuer = MagnetoCorp
Paper = 00001
Current owner = MagnetoCorp
New owner = DigiBank
Purchase time = 31 May 2020 10:00:00 EST
Price = 4.94M USD
```

See how the **buy** transaction has fewer properties that end up in this paper. That's because this transaction only **modifies** this paper. It's only `New owner = DigiBank` that changes as a result of this transaction; everything else is the same. That's OK – the most important thing about the **buy** transaction is the change of ownership, and indeed in this transaction, there's an acknowledgement of the current owner of the paper, MagnetoCorp.

You might ask why the `Purchase time` and `Price` properties are not captured in paper 00001? This comes back to the difference between the transaction and the paper. The 4.94 M USD price tag is actually a property of the transaction, rather than a property of this paper. Spend a little time thinking about this difference; it is not as obvious as it seems. We're going to see later that the ledger will record both pieces of information – the history of all transactions that affect this paper, as well its latest state. Being clear on this separation of information is really important.

It's also worth remembering that paper 00001 may be bought and sold many times. Although we're skipping ahead a little in our scenario, let's examine what transactions we **might** see if paper 00001 changes ownership.

If we have a purchase by BigFund:

```
Txn = buy
Issuer = MagnetoCorp
Paper = 00001
Current owner = DigiBank
New owner = BigFund
Purchase time = 2 June 2020 12:20:00 EST
Price = 4.93M USD
```

Followed by a subsequent purchase by HedgeMatic:

```
Txn = buy
Issuer = MagnetoCorp
Paper = 00001
Current owner = BigFund
New owner = HedgeMatic
Purchase time = 3 June 2020 15:59:00 EST
Price = 4.90M USD
```

See how the paper owners changes, and how in our example, the price changes. Can you think of a reason why the price of MagnetoCorp commercial paper might be falling?

Intuitively, a **buy** transaction demands that both the selling as well as the buying organization need to sign off on such a transaction such that there is proof of the mutual agreement among the two parties that are part of the deal.

Redeem

The **redeem** transaction for paper 00001 represents the end of its lifecycle. In our relatively simple example, HedgeMatic initiates the transaction which transfers the commercial paper back to MagnetoCorp:

```
Txn = redeem
Issuer = MagnetoCorp
Paper = 00001
Current owner = HedgeMatic
Redeem time = 30 Nov 2020 12:00:00 EST
```

Again, notice how the **redeem** transaction has very few properties; all of the changes to paper 00001 can be calculated data by the redeem transaction logic: the `Issuer` will become the new owner, and the `Current state` will change to `redeemed`. The `Current owner` property is specified in our example, so that it can be checked against the current holder of the paper.

From a trust perspective, the same reasoning of the **buy** transaction also applies to the **redeem** instruction: both organizations involved in the transaction are required to sign off on it.

6.2.3 The Ledger

In this topic, we've seen how transactions and the resultant paper states are the two most important concepts in PaperNet. Indeed, we'll see these two fundamental elements in any Hyperledger Fabric distributed ledger – a world state, that contains the current value of all objects, and a blockchain that records the history of all transactions that resulted in the current world state.

The required sign-offs on transactions are enforced through rules, which are evaluated before appending a transaction to the ledger. Only if the required signatures are present, Fabric will accept a transaction as valid.

You're now in a great place translate these ideas into a smart contract. Don't worry if your programming is a little rusty, we'll provide tips and pointers to understand the program code. Mastering the commercial paper smart contract is the first big step towards designing your own application. Or, if you're a business analyst who's comfortable with a little programming, don't be afraid to keep dig a little deeper!

6.3 Process and Data Design

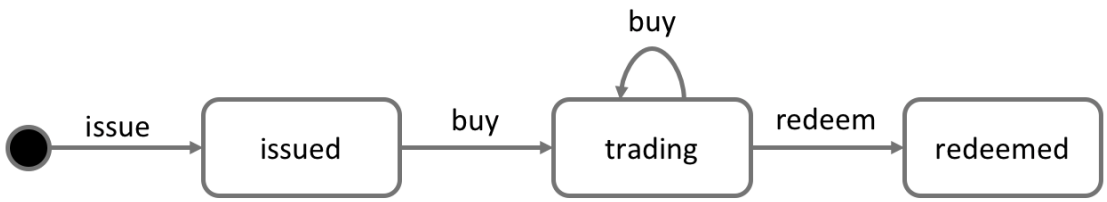
Audience: Architects, Application and smart contract developers, Business professionals

This topic shows you how to design the commercial paper processes and their related data structures in PaperNet. Our analysis highlighted that modelling PaperNet using states and transactions provided a precise way to understand what's happening. We're now going to elaborate on these two strongly related concepts to help us subsequently design the smart contracts and applications of PaperNet.

6.3.1 Lifecycle

As we've seen, there are two important concepts that concern us when dealing with commercial paper; **states** and **transactions**. Indeed, this is true for *all* blockchain use cases; there are conceptual objects of value, modeled as states, whose lifecycle transitions are described by transactions. An effective analysis of states and transactions is an essential starting point for a successful implementation.

We can represent the life cycle of a commercial paper using a state transition diagram:



*The state transition diagram for commercial paper. Commercial papers transition between **issued**, **trading** and **redeemed** states by means of the **issue**, **buy** and **redeem** transactions.*

See how the state diagram describes how commercial papers change over time, and how specific transactions govern the life cycle transitions. In Hyperledger Fabric, smart contracts implement transaction logic that transition commercial papers between their different states. Commercial paper states are actually held in the ledger world state; so let's take a closer look at them.

6.3.2 Ledger state

Recall the structure of a commercial paper:

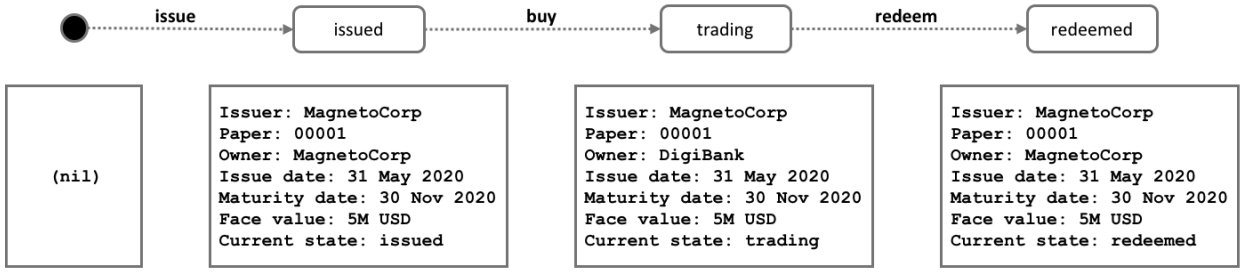
```
Issuer: MagnetoCorp
Paper: 00001
Owner: DigiBank
Issue date: 31 May 2020
Maturity date: 30 Nov 2020
Face value: 5M USD
Current state: trading
```

A commercial paper can be represented as a set of properties, each with a value. Typically, some combination of these properties will provide a unique key for each paper.

See how a commercial paper `Paper` property has value `00001`, and the `Face value` property has value `5M USD`. Most importantly, the `Current state` property indicates whether the commercial paper is `issued`,`trading` or `redeemed`. In combination, the full set of properties make up the **state** of a commercial paper. Moreover, the entire collection of these individual commercial paper states constitutes the ledger world state.

All ledger state share this form; each has a set of properties, each with a different value. This *multi-property* aspect of states is a powerful feature – it allows us to think of a Fabric state as a vector rather than a simple scalar. We then represent facts about whole objects as individual states, which subsequently undergo transitions controlled by transaction logic. A Fabric state is implemented as a key/value pair, in which the value encodes the object properties in a format that captures the object's multiple properties, typically JSON. The ledger database can support advanced query operations against these properties, which is very helpful for sophisticated object retrieval.

See how MagnetoCorp's paper `00001` is represented as a state vector that transitions according to different transaction stimuli:



A commercial paper state is brought into existence and transitions as a result of different transactions. Hyperledger Fabric states have multiple properties, making them vectors rather than scalars.

Notice how each individual paper starts with the empty state, which is technically a `nil` state for the paper, as it doesn't exist! See how paper `00001` is brought into existence by the **issue** transaction, and how it is subsequently updated as a result of the **buy** and **redeem** transactions.

Notice how each state is self-describing; each property has a name and a value. Although all our commercial papers currently have the same properties, this need not be the case for all time, as Hyperledger Fabric supports different states having different properties. This allows the same ledger world state to contain different forms of the same asset as well as different types of asset. It also makes it possible to update a state's structure; imagine a new regulation that requires an additional data field. Flexible state properties support the fundamental requirement of data evolution over time.

6.3.3 State keys

In most practical applications, a state will have a combination of properties that uniquely identify it in a given context – it's **key**. The key for a PaperNet commercial paper is formed by a concatenation of the `Issuer` and `paper` properties; so for MagnetoCorp's first paper, it's `MagnetoCorp00001`.

A state key allows us to uniquely identify a paper; it is created as a result of the **issue** transaction and subsequently updated by **buy** and **redeem**. Hyperledger Fabric requires each state in a ledger to have a unique key.

When a unique key is not available from the available set of properties, an application-determined unique key is specified as an input to the transaction that creates the state. This unique key is usually with some form of UUID, which although less readable, is a standard practice. What's important is that every individual state object in a ledger must have a unique key.

Note: You should avoid using U+0000 (nil byte) in keys.

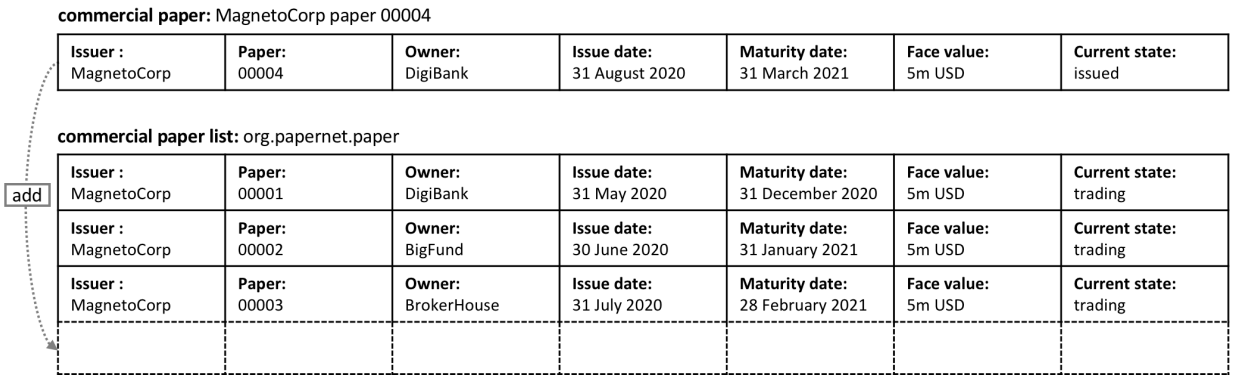
6.3.4 Multiple states

As we've seen, commercial papers in PaperNet are stored as state vectors in a ledger. It's a reasonable requirement to be able to query different commercial papers from the ledger; for example: find all the papers issued by MagnetoCorp, or: find all the papers issued by MagnetoCorp in the `redeemed` state.

To make these kinds of search tasks possible, it's helpful to group all related papers together in a logical list. The PaperNet design incorporates the idea of a commercial paper list – a logical container which is updated whenever commercial papers are issued or otherwise changed.

Logical representation

It's helpful to think of all PaperNet commercial papers being in a single list of commercial papers:

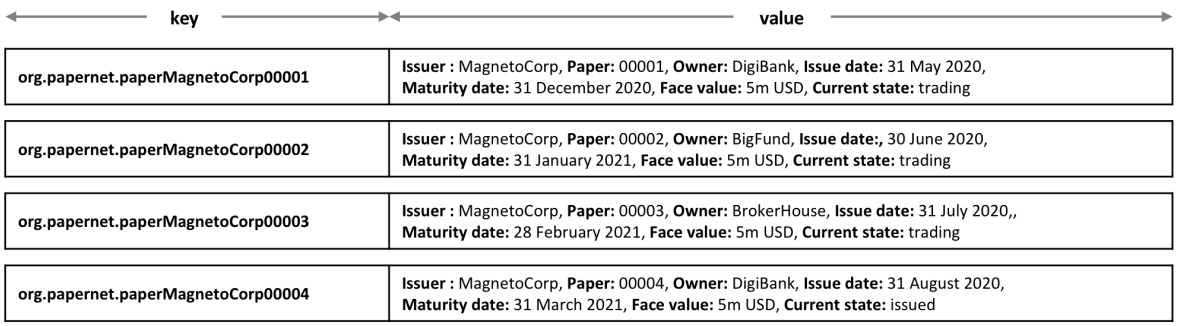


MagnetoCorp's newly created commercial paper 00004 is added to the list of existing commercial papers.

New papers can be added to the list as a result of an **issue** transaction, and papers already in the list can be updated with **buy** or **redeem** transactions. See how the list has a descriptive name: `org.papernet.papers`; it's a really good idea to use this kind of DNS name because well-chosen names will make your blockchain designs intuitive to other people. This idea applies equally well to smart contract names.

Physical representation

While it's correct to think of a single list of papers in PaperNet – `org.papernet.papers` – lists are best implemented as a set of individual Fabric states, whose composite key associates the state with its list. In this way, each state's composite key is both unique and supports effective list query.



Representing a list of PaperNet commercial papers as a set of distinct Hyperledger Fabric states

Notice how each paper in the list is represented by a vector state, with a unique **composite** key formed by the concatenation of `org.papernet.paper`, `Issuer` and `Paper` properties. This structure is helpful for two reasons:

- It allows us to examine any state vector in the ledger to determine which list it's in, without reference to a separate list. It's analogous to looking at set of sports fans, and identifying which team they support by the colour of the shirt they are wearing. The sports fans self-declare their allegiance; we don't need a list of fans.
- Hyperlegder Fabric internally uses a concurrency control mechanism to update a ledger, such that keeping papers in separate state vectors vastly reduces the opportunity for shared-state collisions. Such collisions require transaction re-submission, complicate application design, and decrease performance.

This second point is actually a key take-away for Hyperledger Fabric; the physical design of state vectors is **very important** to optimum performance and behaviour. Keep your states separate!

6.3.5 Trust relationships

We have discussed how the different roles in a network, such as issuer, trader or rating agencies as well as different business interests determine who needs to sign off on a transaction. In Fabric, these rules are captured by so-called **endorsement policies**. The rules can be set on a chaincode granularity, as well as for individual state keys.

This means that in PaperNet, we can set one rule for the whole namespace that determines which organizations can issue new papers. Later, rules can be set and updated for individual papers to capture the trust relationships of buy and redeem transactions.

In the next topic, we will show you how to combine these design concepts to implement the PaperNet commercial paper smart contract, and then an application in exploits it!

6.4 Smart Contract Processing

Audience: Architects, Application and smart contract developers

At the heart of a blockchain network is a smart contract. In PaperNet, the code in the commercial paper smart contract defines the valid states for commercial paper, and the transaction logic that transition a paper from one state to another. In this topic, we're going to show you how to implement a real world smart contract that governs the process of issuing, buying and redeeming commercial paper.

We're going to cover:

- *What is a smart contract and why it's important*
- *How to define a smart contract*
- *How to define a transaction*
- *How to implement a transaction*
- *How to represent a business object in a smart contract*
- *How to store and retrieve an object in the ledger*

If you'd like, you can download the sample and even run it locally. It is written in JavaScript and Java, but the logic is quite language independent, so you'll easily be able to see what's going on! (The sample will become available for Go as well.)

6.4.1 Smart Contract

A smart contract defines the different states of a business object and governs the processes that move the object between these different states. Smart contracts are important because they allow architects and smart contract developers to define the key business processes and data that are shared across the different organizations collaborating in a blockchain network.

In the PaperNet network, the smart contract is shared by the different network participants, such as MagnetoCorp and DigiBank. The same version of the smart contract must be used by all applications connected to the network so that they jointly implement the same shared business processes and data.

6.4.2 Implementation Languages

There are two runtimes that are supported, the Java Virtual Machine and Node.js. This gives the opportunity to use one of JavaScript, TypeScript, Java or any other language that can run on one of these supported runtimes.

In Java and TypeScript, annotations or decorators are used to provide information about the smart contract and it's structure. This allows for a richer development experience — for example, author information or return types can be enforced. Within JavaScript, conventions must be followed, therefore, there are limitations around what can be determined automatically.

Examples here are given in both JavaScript and Java.

6.4.3 Contract class

A copy of the PaperNet commercial paper smart contract is contained in a single file. View it with your browser, or open it in your favorite editor if you've downloaded it.

- `papercontract.js` - JavaScript version
- `CommercialPaperContract.java` - Java version

You may notice from the file path that this is MagnetoCorp's copy of the smart contract. MagnetoCorp and DigiBank must agree on the version of the smart contract that they are going to use. For now, it doesn't matter which organization's copy you use, they are all the same.

Spend a few moments looking at the overall structure of the smart contract; notice that it's quite short! Towards the top of the file, you'll see that there's a definition for the commercial paper smart contract:

JavaScript

```
class CommercialPaperContract extends Contract {...}
```

Java

```
@Contract(...)
@Default
public class CommercialPaperContract implements ContractInterface {...}
```

The `CommercialPaperContract` class contains the transaction definitions for commercial paper – **issue**, **buy** and **redeem**. It's these transactions that bring commercial papers into existence and move them through their lifecycle. We'll examine these *transactions* soon, but for now notice for JavaScript, that the `CommericalPaperContract` extends the Hyperledger Fabric `Contract` class.

With Java, the class must be decorated with the `@Contract(...)` annotation. This provides the opportunity to supply additional information about the contract, such as license and author. The `@Default()` annotation indicates that this contract class is the default contract class. Being able to mark a contract class as the default contract class is useful in some smart contracts which have multiple contract classes.

If you are using a TypeScript implementation, there are similar `@Contract(...)` annotations that fulfill the same purpose as in Java.

For more information on the available annotations, consult the available API documentation:

- API documentation for Java smart contracts
- API documentation for Node.js smart contracts

These classes, annotations, and the `Context` class, were brought into scope earlier:

JavaScript

```
const { Contract, Context } = require('fabric-contract-api');
```

Java

```
import org.hyperledger.fabric.contract.Context;
import org.hyperledger.fabric.contract.ContractInterface;
import org.hyperledger.fabric.contract.annotation.Contact;
import org.hyperledger.fabric.contract.annotation.Contract;
import org.hyperledger.fabric.contract.annotation.Default;
import org.hyperledger.fabric.contract.annotation.Info;
import org.hyperledger.fabric.contract.annotation.License;
import org.hyperledger.fabric.contract.annotation.Transaction;
```

Our commercial paper contract will use built-in features of these classes, such as automatic method invocation, a per-transaction context, transaction handlers, and class-shared state.

Notice also how the JavaScript class constructor uses its superclass to initialize itself with an explicit contract name:

```
constructor() {
    super('org.papernet.commercialpaper');
}
```

With the Java class, the constructor is blank as the explicit contract name can be specified in the `@Contract()` annotation. If it's absent, then the name of the class is used.

Most importantly, `org.papernet.commercialpaper` is very descriptive – this smart contract is the agreed definition of commercial paper for all PaperNet organizations.

Usually there will only be one smart contract per file – contracts tend to have different lifecycles, which makes it sensible to separate them. However, in some cases, multiple smart contracts might provide syntactic help for applications, e.g. `EuroBond`, `DollarBond`, `YenBond`, but essentially provide the same function. In such cases, smart contracts and transactions can be disambiguated.

6.4.4 Transaction definition

Within the class, locate the **issue** method.

JavaScript

```
async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {...
↪}
```

Java

```
@Transaction
public CommercialPaper issue(CommercialPaperContext ctx,
                             String issuer,
                             String paperNumber,
                             String issueDateTime,
                             String maturityDateTime,
                             int faceValue) {...}
```

The Java annotation `@Transaction` is used to mark this method as a transaction definition; TypeScript has an equivalent annotation.

This function is given control whenever this contract is called to `issue` a commercial paper. Recall how commercial paper 00001 was created with the following transaction:

```
Txn = issue
Issuer = MagnetoCorp
Paper = 00001
Issue time = 31 May 2020 09:00:00 EST
Maturity date = 30 November 2020
Face value = 5M USD
```

We've changed the variable names for programming style, but see how these properties map almost directly to the `issue` method variables.

The `issue` method is automatically given control by the contract whenever an application makes a request to issue a commercial paper. The transaction property values are made available to the method via the corresponding variables. See how an application submits a transaction using the Hyperledger Fabric SDK in the application topic, using a sample application program.

You might have noticed an extra variable in the **issue** definition – `ctx`. It's called the **transaction context**, and it's always first. By default, it maintains both per-contract and per-transaction information relevant to *transaction logic*. For example, it would contain MagnetoCorp's specified transaction identifier, a MagnetoCorp issuing user's digital certificate, as well as access to the ledger API.

See how the smart contract extends the default transaction context by implementing its own `createContext()` method rather than accepting the default implementation:

JavaScript

```
createContext() {
  return new CommercialPaperContext()
}
```

Java

```
@Override
public Context createContext(ChaincodeStub stub) {
     return new CommercialPaperContext(stub);
}
```

This extended context adds a custom property `paperList` to the defaults:

JavaScript

```
class CommercialPaperContext extends Context {

  constructor() {
    super();
    // All papers are held in a list of papers
    this.paperList = new PaperList(this);
}
```

Java

```
class CommercialPaperContext extends Context {
    public CommercialPaperContext(ChaincodeStub stub) {
        super(stub);
        this.paperList = new PaperList(this);
    }
    public PaperList paperList;
}
```

We'll soon see how `ctx.paperList` can be subsequently used to help store and retrieve all PaperNet commercial papers.

To solidify your understanding of the structure of a smart contract transaction, locate the **buy** and **redeem** transaction definitions, and see if you can see how they map to their corresponding commercial paper transactions.

The **buy** transaction:

```
Txn = buy
Issuer = MagnetoCorp
Paper = 00001
Current owner = MagnetoCorp
New owner = DigiBank
Purchase time = 31 May 2020 10:00:00 EST
Price = 4.94M USD
```

JavaScript

```
async buy(ctx, issuer, paperNumber, currentOwner, newOwner, price, purchaseTime) {...}
```

Java

```
@Transaction
public CommercialPaper buy(CommercialPaperContext ctx,
                           String issuer,
                           String paperNumber,
                           String currentOwner,
                           String newOwner,
                           int price,
                           String purchaseDateTime) {...}
```

The **redeem** transaction:

```
Txn = redeem
Issuer = MagnetoCorp
Paper = 00001
Redeemer = DigiBank
Redeem time = 31 Dec 2020 12:00:00 EST
```

JavaScript

```
async redeem(ctx, issuer, paperNumber, redeemingOwner, redeemDateTime) {...}
```

Java

```
@Transaction
public CommercialPaper redeem(CommercialPaperContext ctx,
                              String issuer,
                              String paperNumber,
                              String redeemingOwner,
                              String redeemDateTime) {...}
```

In both cases, observe the 1:1 correspondence between the commercial paper transaction and the smart contract method definition.

All of the JavaScript functions use the `async` and `await` keywords which allow JavaScript functions to be treated as if they were synchronous function calls.

6.4.5 Transaction logic

Now that you've seen how contracts are structured and transactions are defined, let's focus on the logic within the smart contract.

Recall the first **issue** transaction:

```
Txn = issue
Issuer = MagnetoCorp
Paper = 00001
Issue time = 31 May 2020 09:00:00 EST
Maturity date = 30 November 2020
Face value = 5M USD
```

It results in the **issue** method being passed control:

JavaScript

```
async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {

   // create an instance of the paper
  let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime,␣
↪maturityDateTime, faceValue);

  // Smart contract, rather than paper, moves paper into ISSUED state
  paper.setIssued();

  // Newly issued paper is owned by the issuer
  paper.setOwner(issuer);

  // Add the paper to the list of all similar commercial papers in the ledger world␣
↪state
  await ctx.paperList.addPaper(paper);

  // Must return a serialized paper to caller of smart contract
  return paper.toBuffer();
}
```

Java

```
@Transaction
public CommercialPaper issue(CommercialPaperContext ctx,
                             String issuer,
                             String paperNumber,
                             String issueDateTime,
                             String maturityDateTime,
                             int faceValue) {

    System.out.println(ctx);

    // create an instance of the paper
    CommercialPaper paper = CommercialPaper.createInstance(issuer, paperNumber,␣
↪issueDateTime, maturityDateTime,
            faceValue,issuer,"");

    // Smart contract, rather than paper, moves paper into ISSUED state
    paper.setIssued();
```

(continues on next page)

(continued from previous page)

```
    // Newly issued paper is owned by the issuer
    paper.setOwner(issuer);

    System.out.println(paper);
    // Add the paper to the list of all similar commercial papers in the ledger
    // world state
    ctx.paperList.addPaper(paper);

    // Must return a serialized paper to caller of smart contract
    return paper;
}
```

The logic is simple: take the transaction input variables, create a new commercial paper `paper`, add it to the list of all commercial papers using `paperList`, and return the new commercial paper (serialized as a buffer) as the transaction response.

See how `paperList` is retrieved from the transaction context to provide access to the list of commercial papers. `issue()`, `buy()` and `redeem()` continually re-access `ctx.paperList` to keep the list of commercial papers up-to-date.

The logic for the **buy** transaction is a little more elaborate:

JavaScript

```
async buy(ctx, issuer, paperNumber, currentOwner, newOwner, price, purchaseDateTime) {

  // Retrieve the current paper using key fields provided
  let paperKey = CommercialPaper.makeKey([issuer, paperNumber]);
  let paper = await ctx.paperList.getPaper(paperKey);

  // Validate current owner
  if (paper.getOwner() !== currentOwner) {
      throw new Error('Paper ' + issuer + paperNumber + ' is not owned by ' +␣
↪currentOwner);
  }

  // First buy moves state from ISSUED to TRADING
  if (paper.isIssued()) {
      paper.setTrading();
  }

  // Check paper is not already REDEEMED
  if (paper.isTrading()) {
      paper.setOwner(newOwner);
  } else {
      throw new Error('Paper ' + issuer + paperNumber + ' is not trading. Current␣
↪state = ' +paper.getCurrentState());
  }

  // Update the paper
  await ctx.paperList.updatePaper(paper);
  return paper.toBuffer();
}
```

Java

```
@Transaction
public CommercialPaper buy(CommercialPaperContext ctx,
                           String issuer,
                           String paperNumber,
                           String currentOwner,
                           String newOwner,
                           int price,
                           String purchaseDateTime) {

    // Retrieve the current paper using key fields provided
    String paperKey = State.makeKey(new String[] { paperNumber });
    CommercialPaper paper = ctx.paperList.getPaper(paperKey);

    // Validate current owner
    if (!paper.getOwner().equals(currentOwner)) {
        throw new RuntimeException("Paper " + issuer + paperNumber + " is not owned␣
↪by " + currentOwner);
    }

    // First buy moves state from ISSUED to TRADING
    if (paper.isIssued()) {
        paper.setTrading();
    }

    // Check paper is not already REDEEMED
    if (paper.isTrading()) {
        paper.setOwner(newOwner);
    } else {
        throw new RuntimeException(
                "Paper " + issuer + paperNumber + " is not trading. Current state = "␣
↪+ paper.getState());
    }

    // Update the paper
    ctx.paperList.updatePaper(paper);
    return paper;
}
```

See how the transaction checks `currentOwner` and that `paper` is `TRADING` before changing the owner with `paper.setOwner(newOwner)`. The basic flow is simple though – check some pre-conditions, set the new owner, update the commercial paper on the ledger, and return the updated commercial paper (serialized as a buffer) as the transaction response.

Why don't you see if you can understand the logic for the **redeem** transaction?

6.4.6 Representing an object

We've seen how to define and implement the **issue**, **buy** and **redeem** transactions using the `CommercialPaper` and `PaperList` classes. Let's end this topic by seeing how these classes work.

Locate the `CommercialPaper` class:

JavaScript In the paper.js file:

```
class CommercialPaper extends State {...}
```

Java In the CommercialPaper.java file:

```
@DataType()
public class CommercialPaper extends State {...}
```

This class contains the in-memory representation of a commercial paper state. See how the `createInstance` method initializes a new commercial paper with the provided parameters:

JavaScript

```
static createInstance(issuer, paperNumber, issueDateTime, maturityDateTime,␣
↪faceValue) {
  return new CommercialPaper({ issuer, paperNumber, issueDateTime, maturityDateTime,␣
↪faceValue });
}
```

Java

```
public static CommercialPaper createInstance(String issuer, String paperNumber,␣
↪String issueDateTime,
        String maturityDateTime, int faceValue, String owner, String state) {
    return new CommercialPaper().setIssuer(issuer).setPaperNumber(paperNumber).
↪setMaturityDateTime(maturityDateTime)
            .setFaceValue(faceValue).setKey().setIssueDateTime(issueDateTime).
↪setOwner(owner).setState(state);
}
```

Recall how this class was used by the **issue** transaction:

JavaScript

```
let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime,␣
↪maturityDateTime, faceValue);
```

Java

```
CommercialPaper paper = CommercialPaper.createInstance(issuer, paperNumber,␣
↪issueDateTime, maturityDateTime,
        faceValue,issuer,"");
```

See how every time the issue transaction is called, a new in-memory instance of a commercial paper is created containing the transaction data.

A few important points to note:

- This is an in-memory representation; we'll see *later* how it appears on the ledger.
- The `CommercialPaper` class extends the `State` class. `State` is an application-defined class which creates a common abstraction for a state. All states have a business object class which they represent, a composite key, can be serialized and de-serialized, and so on. `State` helps our code be more legible when we are storing more than one business object type on the ledger. Examine the `State` class in the `state.js` file.
- A paper computes its own key when it is created – this key will be used when the ledger is accessed. The key is formed from a combination of `issuer` and `paperNumber`.

```
constructor(obj) {
  super(CommercialPaper.getClass(), [obj.issuer, obj.paperNumber]);
  Object.assign(this, obj);
}
```

- A paper is moved to the `ISSUED` state by the transaction, not by the paper class. That's because it's the smart contract that governs the lifecycle state of the paper. For example, an `import` transaction might create a new set of papers immediately in the `TRADING` state.

The rest of the `CommercialPaper` class contains simple helper methods:

```
getOwner() {
    return this.owner;
}
```

Recall how methods like this were used by the smart contract to move the commercial paper through its lifecycle. For example, in the **redeem** transaction we saw:

```
if (paper.getOwner() === redeemingOwner) {
  paper.setOwner(paper.getIssuer());
  paper.setRedeemed();
}
```

6.4.7 Access the ledger

Now locate the `PaperList` class in the `paperlist.js` file:

```
class PaperList extends StateList {
```

This utility class is used to manage all PaperNet commercial papers in Hyperledger Fabric state database. The PaperList data structures are described in more detail in the architecture topic.

Like the `CommercialPaper` class, this class extends an application-defined `StateList` class which creates a common abstraction for a list of states – in this case, all the commercial papers in PaperNet.

The `addPaper()` method is a simple veneer over the `StateList.addState()` method:

```
async addPaper(paper) {
  return this.addState(paper);
}
```

You can see in the `StateList.js` file how the `StateList` class uses the Fabric API `putState()` to write the commercial paper as state data in the ledger:

```
async addState(state) {
  let key = this.ctx.stub.createCompositeKey(this.name, state.getSplitKey());
  let data = State.serialize(state);
  await this.ctx.stub.putState(key, data);
}
```

Every piece of state data in a ledger requires these two fundamental elements:

- **Key**: `key` is formed with `createCompositeKey()` using a fixed name and the key of `state`. The name was assigned when the `PaperList` object was constructed, and `state.getSplitKey()` determines each state's unique key.
- **Data**: `data` is simply the serialized form of the commercial paper state, created using the `State.serialize()` utility method. The `State` class serializes and deserializes data using JSON, and the State's business object class as required, in our case `CommercialPaper`, again set when the `PaperList` object was constructed.

Notice how a `StateList` doesn't store anything about an individual state or the total list of states – it delegates all of that to the Fabric state database. This is an important design pattern – it reduces the opportunity for ledger MVCC collisions in Hyperledger Fabric.

The StateList `getState()` and `updateState()` methods work in similar ways:

```
async getState(key) {
  let ledgerKey = this.ctx.stub.createCompositeKey(this.name, State.splitKey(key));
  let data = await this.ctx.stub.getState(ledgerKey);
  let state = State.deserialize(data, this.supportedClasses);
  return state;
}
```

```
async updateState(state) {
  let key = this.ctx.stub.createCompositeKey(this.name, state.getSplitKey());
  let data = State.serialize(state);
  await this.ctx.stub.putState(key, data);
}
```

See how they use the Fabric APIs `putState()`, `getState()` and `createCompositeKey()` to access the ledger. We'll expand this smart contract later to list all commercial papers in paperNet – what might the method look like to implement this ledger retrieval?

That's it! In this topic you've understood how to implement the smart contract for PaperNet. You can move to the next sub topic to see how an application calls the smart contract using the Fabric SDK.

6.5 Application

Audience: Architects, Application and smart contract developers

An application can interact with a blockchain network by submitting transactions to a ledger or querying ledger content. This topic covers the mechanics of how an application does this; in our scenario, organizations access PaperNet using applications which invoke **issue**, **buy** and **redeem** transactions defined in a commercial paper smart contract. Even though MagnetoCorp's application to issue a commercial paper is basic, it covers all the major points of understanding.

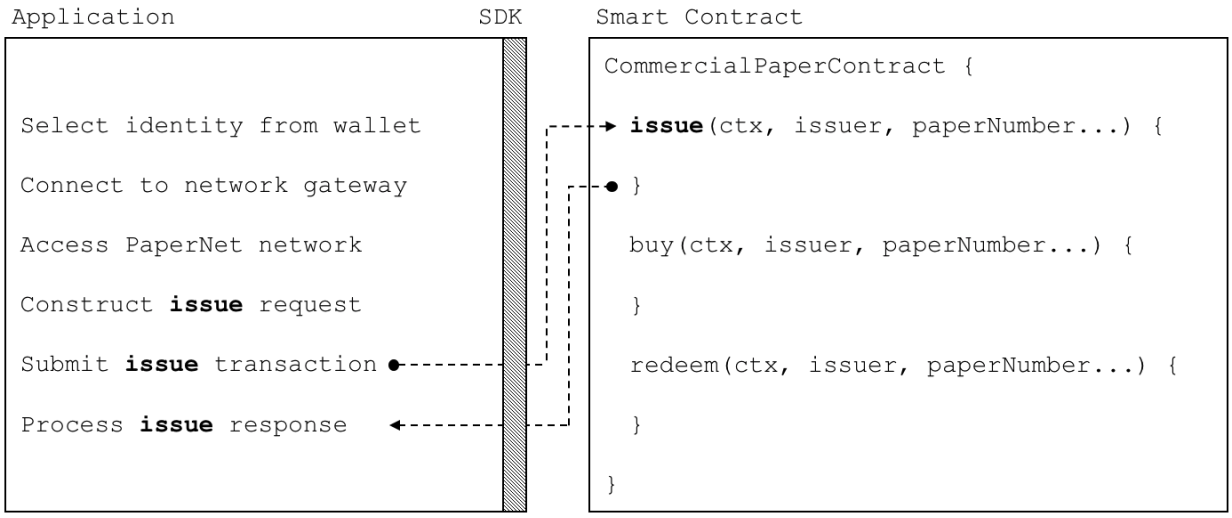
In this topic, we're going to cover:

- *The application flow to invoke a smart contract*
- *How an application uses a wallet and identity*
- *How an application connects using a gateway*
- *How to access a particular network*
- *How to construct a transaction request*
- *How to submit a transaction*
- *How to process a transaction response*

To help your understanding, we'll make reference to the commercial paper sample application provided with Hyperledger Fabric. You can download it and run it locally. It is written in both JavaScript and Java, but the logic is quite language independent, so you'll easily be able to see what's going on! (The sample will become available for Go as well.)

6.5.1 Basic Flow

An application interacts with a blockchain network using the Fabric SDK. Here's a simplified diagram of how an application invokes a commercial paper smart contract:



A PaperNet application invokes the commercial paper smart contract to submit an issue transaction request.

An application has to follow six basic steps to submit a transaction:

- Select an identity from a wallet
- Connect to a gateway
- Access the desired network
- Construct a transaction request for a smart contract
- Submit the transaction to the network
- Process the response

You're going to see how a typical application performs these six steps using the Fabric SDK. You'll find the application code in the `issue.js` file. View it in your browser, or open it in your favourite editor if you've downloaded it. Spend a few moments looking at the overall structure of the application; even with comments and spacing, it's only 100 lines of code!

6.5.2 Wallet

Towards the top of `issue.js`, you'll see two Fabric classes are brought into scope:

```
const { Wallets, Gateway } = require('fabric-network');
```

You can read about the `fabric-network` classes in the node SDK documentation, but for now, let's see how they are used to connect MagnetoCorp's application to PaperNet. The application uses the Fabric **Wallet** class as follows:

```
const wallet = await Wallets.newFileSystemWallet('../identity/user/isabella/wallet');
```

See how `wallet` locates a wallet in the local filesystem. The identity retrieved from the wallet is clearly for a user called Isabella, who is using the `issue` application. The wallet holds a set of identities – X.509 digital certificates –

which can be used to access PaperNet or any other Fabric network. If you run the tutorial, and look in this directory, you'll see the identity credentials for Isabella.

Think of a wallet holding the digital equivalents of your government ID, driving license or ATM card. The X.509 digital certificates within it will associate the holder with a organization, thereby entitling them to rights in a network channel. For example, `Isabella` might be an administrator in MagnetoCorp, and this could give her more privileges than a different user – `Balaji` from DigiBank. Moreover, a smart contract can retrieve this identity during smart contract processing using the transaction context.

Note also that wallets don't hold any form of cash or tokens – they hold identities.

6.5.3 Gateway

The second key class is a Fabric **Gateway**. Most importantly, a gateway identifies one or more peers that provide access to a network – in our case, PaperNet. See how `issue.js` connects to its gateway:

```
await gateway.connect(connectionProfile, connectionOptions);
```

`gateway.connect()` has two important parameters:

- **connectionProfile**: the file system location of a connection profile that identifies a set of peers as a gateway to PaperNet
- **connectionOptions**: a set of options used to control how `issue.js` interacts with PaperNet

See how the client application uses a gateway to insulate itself from the network topology, which might change. The gateway takes care of sending the transaction proposal to the right peer nodes in the network using the connection profile and connection options.

Spend a few moments examining the connection profile `./gateway/connectionProfile.yaml`. It uses YAML, making it easy to read.

It was loaded and converted into a JSON object:

```
let connectionProfile = yaml.safeLoad(file.readFileSync('./gateway/connectionProfile.
↪yaml', 'utf8'));
```

Right now, we're only interested in the `channels:` and `peers:` sections of the profile: (We've modified the details slightly to better explain what's happening.)

```
channels:
  papernet:
    peers:
      peer1.magnetocorp.com:
        endorsingPeer: true
        eventSource: true

      peer2.digibank.com:
        endorsingPeer: true
        eventSource: true

peers:
  peer1.magnetocorp.com:
    url: grpcs://localhost:7051
    grpcOptions:
      ssl-target-name-override: peer1.magnetocorp.com
      request-timeout: 120
    tlsCACerts:
```

(continues on next page)

(continued from previous page)

```
      path: certificates/magnetocorp/magnetocorp.com-cert.pem

  peer2.digibank.com:
    url: grpcs://localhost:8051
    grpcOptions:
      ssl-target-name-override: peer1.digibank.com
    tlsCACerts:
      path: certificates/digibank/digibank.com-cert.pem
```

See how `channel:` identifies the `PaperNet:` network channel, and two of its peers. MagnetoCorp has `peer1.magenetocorp.com` and DigiBank has `peer2.digibank.com`, and both have the role of endorsing peers. Link to these peers via the `peers:` key, which contains details about how to connect to them, including their respective network addresses.

The connection profile contains a lot of information – not just peers – but network channels, network orderers, organizations, and CAs, so don't worry if you don't understand all of it!

Let's now turn our attention to the `connectionOptions` object:

```
let connectionOptions = {
    identity: userName,
    wallet: wallet,
    discovery: { enabled:true, asLocalhost: true }
};
```

See how it specifies that identity, `userName`, and wallet, `wallet`, should be used to connect to a gateway. These were assigned values earlier in the code.

There are other connection options which an application could use to instruct the SDK to act intelligently on its behalf. For example:

```
let connectionOptions = {
  identity: userName,
  wallet: wallet,
  eventHandlerOptions: {
    commitTimeout: 100,
    strategy: EventStrategies.MSPID_SCOPE_ANYFORTX
  },
}
```

Here, `commitTimeout` tells the SDK to wait 100 seconds to hear whether a transaction has been committed. And `strategy: EventStrategies.MSPID_SCOPE_ANYFORTX` specifies that the SDK can notify an application after a single MagnetoCorp peer has confirmed the transaction, in contrast to `strategy: EventStrategies.NETWORK_SCOPE_ALLFORTX` which requires that all peers from MagnetoCorp and DigiBank to confirm the transaction.

If you'd like to, read more about how connection options allow applications to specify goal-oriented behaviour without having to worry about how it is achieved.

6.5.4 Network channel

The peers defined in the gateway `connectionProfile.yaml` provide `issue.js` with access to PaperNet. Because these peers can be joined to multiple network channels, the gateway actually provides the application with access to multiple network channels!

See how the application selects a particular channel:

```
const network = await gateway.getNetwork('PaperNet');
```

From this point onwards, `network` will provide access to PaperNet. Moreover, if the application wanted to access another network, `BondNet`, at the same time, it is easy:

```
const network2 = await gateway.getNetwork('BondNet');
```

Now our application has access to a second network, `BondNet`, simultaneously with `PaperNet`!

We can see here a powerful feature of Hyperledger Fabric – applications can participate in a **network of networks**, by connecting to multiple gateway peers, each of which is joined to multiple network channels. Applications will have different rights in different channels according to their wallet identity provided in `gateway.connect()`.

6.5.5 Construct request

The application is now ready to **issue** a commercial paper. To do this, it's going to use `CommercialPaperContract` and again, its fairly straightforward to access this smart contract:

```
const contract = await network.getContract('papercontract', 'org.papernet.
↪commercialpaper');
```

Note how the application provides a name – `papercontract` – and an explicit contract name: `org.papernet.commercialpaper`! We see how a contract name picks out one contract from the `papercontract.js` chaincode file that contains many contracts. In PaperNet, `papercontract.js` was installed and deployed to the channel with the name `papercontract`, and if you're interested, read how to deploy a chaincode containing multiple smart contracts.

If our application simultaneously required access to another contract in PaperNet or BondNet this would be easy:

```
const euroContract = await network.getContract('EuroCommercialPaperContract');

const bondContract = await network2.getContract('BondContract');
```

In these examples, note how we didn't use a qualifying contract name – we have only one smart contract per file, and `getContract()` will use the first contract it finds.

Recall the transaction MagnetoCorp uses to issue its first commercial paper:

```
Txn = issue
Issuer = MagnetoCorp
Paper = 00001
Issue time = 31 May 2020 09:00:00 EST
Maturity date = 30 November 2020
Face value = 5M USD
```

Let's now submit this transaction to PaperNet!

6.5.6 Submit transaction

Submitting a transaction is a single method call to the SDK:

```
const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001
↪', '2020-05-31', '2020-11-30', '5000000');
```

See how the `submitTransaction()` parameters match those of the transaction request. It's these values that will be passed to the `issue()` method in the smart contract, and used to create a new commercial paper. Recall its signature:

```
async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {...
↪}
```

It might appear that a smart contract receives control shortly after the application issues `submitTransaction()`, but that's not the case. Under the covers, the SDK uses the `connectionOptions` and `connectionProfile` details to send the transaction proposal to the right peers in the network, where it can get the required endorsements. But the application doesn't need to worry about any of this – it just issues `submitTransaction` and the SDK takes care of it all!

Note that the `submitTransaction` API includes a process for listening for transaction commits. Listening for commits is required because without it, you will not know whether your transaction has successfully been orderered, validated, and committed to the ledger.

Let's now turn our attention to how the application handles the response!

6.5.7 Process response

Recall from `papercontract.js` how the **issue** transaction returns a commercial paper response:

```
return paper.toBuffer();
```

You'll notice a slight quirk – the new `paper` needs to be converted to a buffer before it is returned to the application. Notice how `issue.js` uses the class method `CommercialPaper.fromBuffer()` to rehydrate the response buffer as a commercial paper:

```
let paper = CommercialPaper.fromBuffer(issueResponse);
```

This allows `paper` to be used in a natural way in a descriptive completion message:

```
console.log(`${paper.issuer} commercial paper : ${paper.paperNumber} successfully␣
↪issued for value ${paper.faceValue}`);
```

See how the same `paper` class has been used in both the application and smart contract – if you structure your code like this, it'll really help readability and reuse.

As with the transaction proposal, it might appear that the application receives control soon after the smart contract completes, but that's not the case. Under the covers, the SDK manages the entire consensus process, and notifies the application when it is complete according to the `strategy` connectionOption. If you're interested in what the SDK does under the covers, read the detailed transaction flow.

That's it! In this topic you've understood how to call a smart contract from a sample application by examining how MagnetoCorp's application issues a new commercial paper in PaperNet. Now examine the key ledger and smart contract data structures are designed by in the architecture topic behind them.

6.6 Application design elements

This section elaborates the key features for client application and smart contract development found in Hyperledger Fabric. A solid understanding of the features will help you design and implement efficient and effective solutions.

6.6.1 Contract names

Audience: Architects, application and smart contract developers, administrators

A chaincode is a generic container for deploying code to a Hyperledger Fabric blockchain network. One or more related smart contracts are defined within a chaincode. Every smart contract has a name that uniquely identifies it within a chaincode. Applications access a particular smart contract within a chaincode using its contract name.

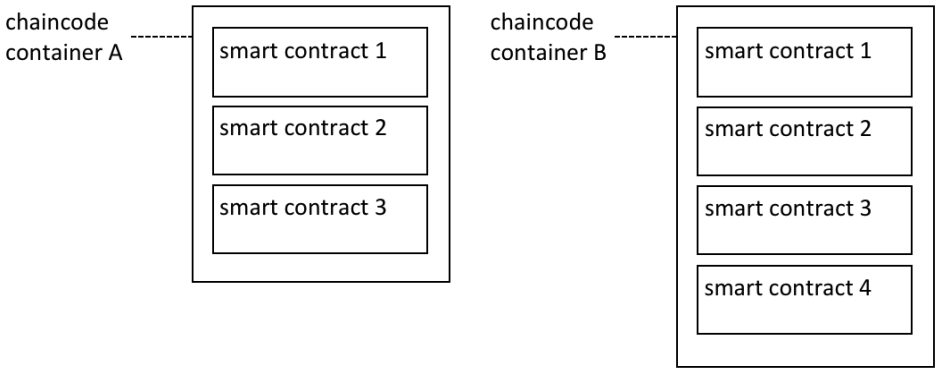
In this topic, we're going to cover:

- *How a chaincode contains multiple smart contracts*
- *How to assign a smart contract name*
- *How to use a smart contract from an application*
- *The default smart contract*

Chaincode

In the Developing Applications topic, we can see how the Fabric SDKs provide high level programming abstractions which help application and smart contract developers to focus on their business problem, rather than the low level details of how to interact with a Fabric network.

Smart contracts are one example of a high level programming abstraction, and it is possible to define smart contracts within in a chaincode container. When a chaincode is installed on your peer and deployed to a channel, all the smart contracts within it are made available to your applications.



Multiple smart contracts can be defined within a chaincode. Each is uniquely identified by their name within a chaincode.

In the diagram *above*, chaincode A has three smart contracts defined within it, whereas chaincode B has four smart contracts. See how the chaincode name is used to fully qualify a particular smart contract.

The ledger structure is defined by a set of deployed smart contracts. That's because the ledger contains facts about the business objects of interest to the network (such as commercial paper within PaperNet), and these business objects are moved through their lifecycle (e.g. issue, buy, redeem) by the transaction functions defined within a smart contract.

In most cases, a chaincode will only have one smart contract defined within it. However, it can make sense to keep related smart contracts together in a single chaincode. For example, commercial papers denominated in different currencies might have contracts `EuroPaperContract`, `DollarPaperContract`, `YenPaperContract` which might need to be kept synchronized with each other in the channel to which they are deployed.

Name

Each smart contract within a chaincode is uniquely identified by its contract name. A smart contract can explicitly assign this name when the class is constructed, or let the `Contract` class implicitly assign a default name.

Examine the `papercontract.js` chaincode file:

```
class CommercialPaperContract extends Contract {

    constructor() {
        // Unique name when multiple contracts per chaincode file
        super('org.papernet.commercialpaper');
    }
```

See how the `CommercialPaperContract` constructor specifies the contract name as `org.papernet.commercialpaper`. The result is that within the `papercontract` chaincode, this smart contract is now associated with the contract name `org.papernet.commercialpaper`.

If an explicit contract name is not specified, then a default name is assigned – the name of the class. In our example, the default contract name would be `CommercialPaperContract`.

Choose your names carefully. It's not just that each smart contract must have a unique name; a well-chosen name is illuminating. Specifically, using an explicit DNS-style naming convention is recommended to help organize clear and meaningful names; `org.papernet.commercialpaper` conveys that the PaperNet network has defined a standard commercial paper smart contract.

Contract names are also helpful to disambiguate different smart contract transaction functions with the same name in a given chaincode. This happens when smart contracts are closely related; their transaction names will tend to be the same. We can see that a transaction is uniquely defined within a channel by the combination of its chaincode and smart contract name.

Contract names must be unique within a chaincode file. Some code editors will detect multiple definitions of the same class name before deployment. Regardless the chaincode will return an error if multiple classes with the same contract name are explicitly or implicitly specified.

Application

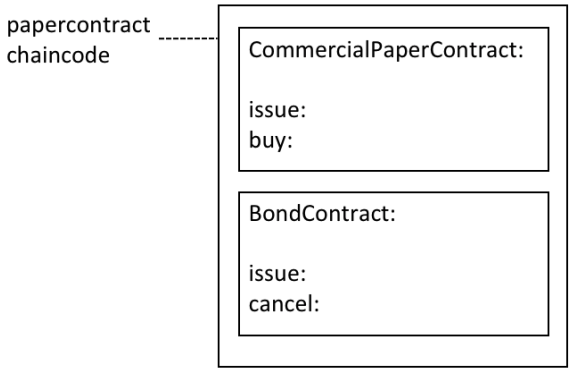
Once a chaincode has been installed on a peer and deployed to a channel, the smart contracts in it are accessible to an application:

```
const network = await gateway.getNetwork(`papernet`);

const contract = await network.getContract('papercontract', 'org.papernet.
↪commercialpaper');

const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001
↪', '2020-05-31', '2020-11-30', '5000000');
```

See how the application accesses the smart contract with the `network.getContract()` method. The `papercontract` chaincode name `org.papernet.commercialpaper` returns a `contract` reference which can be used to submit transactions to issue commercial paper with the `contract.submitTransaction()` API.

Default contract

The first smart contract defined in a chaincode is called the *default* smart contract. A default is helpful because a chaincode will usually have one smart contract defined within it; a default allows the application to access those transactions directly – without specifying a contract name.



A default smart contract is the first contract defined in a chaincode.

In this diagram, `CommercialPaperContract` is the default smart contract. Even though we have two smart contracts, the default smart contract makes our *previous* example easier to write:

```
const network = await gateway.getNetwork(`papernet`);

const contract = await network.getContract('papercontract');

const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001
↪', '2020-05-31', '2020-11-30', '5000000');
```

This works because the default smart contract in `papercontract` is `CommercialPaperContract` and it has an `issue` transaction. Note that the `issue` transaction in `BondContract` can only be invoked by explicitly addressing it. Likewise, even though the `cancel` transaction is unique, because `BondContract` is *not* the default smart contract, it must also be explicitly addressed.

In most cases, a chaincode will only contain a single smart contract, so careful naming of the chaincode can reduce the need for developers to care about chaincode as a concept. In the example code *above* it feels like `papercontract` is a smart contract.

In summary, contract names are a straightforward mechanism to identify individual smart contracts within a given chaincode. Contract names make it easy for applications to find a particular smart contract and use it to access the ledger.

6.6.2 Chaincode namespace

Audience: Architects, application and smart contract developers, administrators

A chaincode namespace allows it to keep its world state separate from other chaincodes. Specifically, smart contracts in the same chaincode share direct access to the same world state, whereas smart contracts in different chaincodes cannot directly access each other's world state. If a smart contract needs to access another chaincode world state, it can do this by performing a chaincode-to-chaincode invocation. Finally, a blockchain can contain transactions which relate to different world states.

In this topic, we're going to cover:

- *The importance of namespaces*
- *What is a chaincode namespace*
- *Channels and namespaces*
- *How to use chaincode namespaces*

- *How to access world states across smart contracts*
- *Design considerations for chaincode namespaces*

Motivation

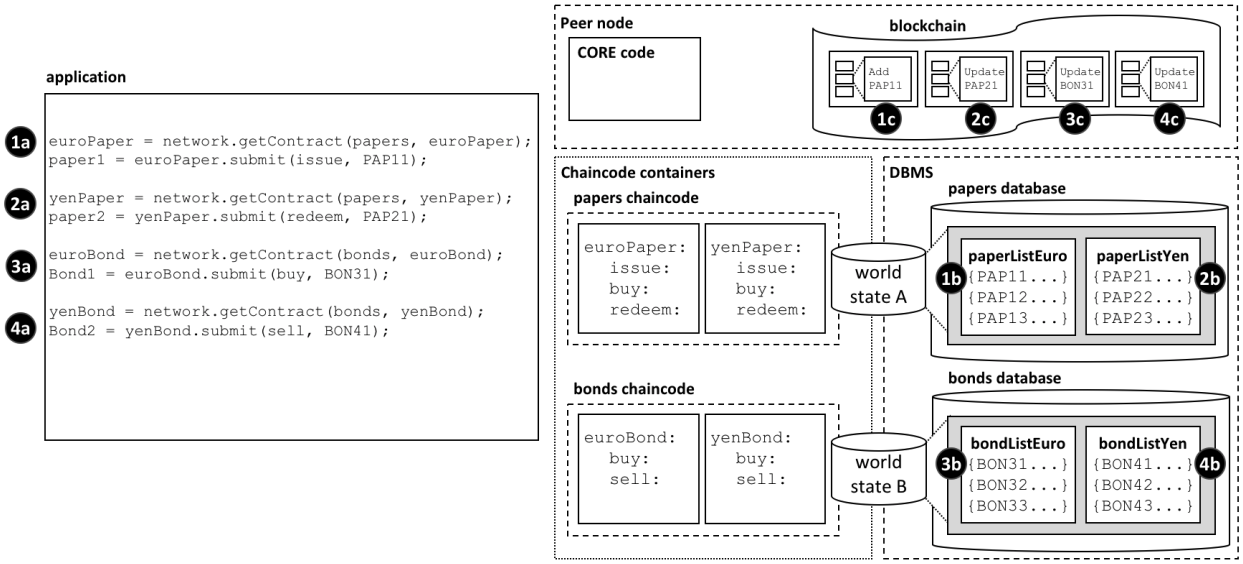
A namespace is a common concept. We understand that *Park Street, New York* and *Park Street, Seattle* are different streets even though they have the same name. The city forms a **namespace** for Park Street, simultaneously providing freedom and clarity.

It's the same in a computer system. Namespaces allow different users to program and operate different parts of a shared system, without getting in each other's way. Many programming languages have namespaces so that programs can freely assign unique identifiers, such as variable names, without worrying about other programs doing the same. We'll see that Hyperledger Fabric uses namespaces to help smart contracts keep their ledger world state separate from other smart contracts.

Scenario

Let's examine how the ledger world state organizes facts about business objects that are important to the organizations in a channel using the diagram below. Whether these objects are commercial papers, bonds, or vehicle registrations, and wherever they are in their lifecycle, they are maintained as states within the ledger world state database. A smart contract manages these business objects by interacting with the ledger (world state and blockchain), and in most cases this will involve it querying or updating the ledger world state.

It's vitally important to understand that the ledger world state is partitioned according to the chaincode of the smart contract that accesses it, and this partitioning, or *namespacing* is an important design consideration for architects, administrators and programmers.



The ledger world state is separated into different namespaces according to the chaincode that accesses it. Within a given channel, smart contracts in the same chaincode share the same world state, and smart contracts in different chaincodes cannot directly access each other's world state. Likewise, a blockchain can contain transactions that relate to different chaincode world states.

In our example, we can see four smart contracts defined in two different chaincodes, each of which is in their own chaincode container. The `euroPaper` and `yenPaper` smart contracts are defined in the `papers` chaincode. The situation is similar for the `euroBond` and `yenBond` smart contracts – they are defined in the `bonds` chaincode. This design helps application programmers understand whether they are working with commercial papers or bonds priced

in Euros or Yen, and because the rules for each financial product don't really change for different currencies, it makes sense to manage their deployment in the same chaincode.

The *diagram* also shows the consequences of this deployment choice. The database management system (DBMS) creates different world state databases for the `papers` and `bonds` chaincodes and the smart contracts contained within them. `World state A` and `world state B` are each held within distinct databases; the data are isolated from each other such that a single world state query (for example) cannot access both world states. The world state is said to be *namespaced* according to its chaincode.

See how `world state A` contains two lists of commercial papers `paperListEuro` and `paperListYen`. The states `PAP11` and `PAP21` are instances of each paper managed by the `euroPaper` and `yenPaper` smart contracts respectively. Because they share the same chaincode namespace, their keys (`PAPxyz`) must be unique within the namespace of the `papers` chaincode, a little like a street name is unique within a town. Notice how it would be possible to write a smart contract in the `papers` chaincode that performed an aggregate calculation over all the commercial papers – whether priced in Euros or Yen – because they share the same namespace. The situation is similar for bonds – they are held within `world state B` which maps to a separate `bonds` database, and their keys must be unique.

Just as importantly, namespaces mean that `euroPaper` and `yenPaper` cannot directly access `world state B`, and that `euroBond` and `yenBond` cannot directly access `world state A`. This isolation is helpful, as commercial papers and bonds are very distinct financial instruments; they have different attributes and are subject to different rules. It also means that `papers` and `bonds` could have the same keys, because they are in different namespaces. This is helpful; it provides a significant degree of freedom for naming. Use this freedom to name different business objects meaningfully.

Most importantly, we can see that a blockchain is associated with the peer operating in a particular channel, and that it contains transactions that affect both `world state A` and `world state B`. That's because the blockchain is the most fundamental data structure contained in a peer. The set of world states can always be recreated from this blockchain, because they are the cumulative results of the blockchain's transactions. A world state helps simplify smart contracts and improve their efficiency, as they usually only require the current value of a state. Keeping world states separate via namespaces helps smart contracts isolate their logic from other smart contracts, rather than having to worry about transactions that correspond to different world states. For example, a `bonds` contract does not need to worry about `paper` transactions, because it cannot see their resultant world state.

It's also worth noticing that the peer, chaincode containers and DBMS all are logically different processes. The peer and all its chaincode containers are always in physically separate operating system processes, but the DBMS can be configured to be embedded or separate, depending on its type. For LevelDB, the DBMS is wholly contained within the peer, but for CouchDB, it is a separate operating system process.

It's important to remember that namespace choices in this example are the result of a business requirement to share commercial papers in different currencies but isolate them separate from bonds. Think about how the namespace structure would be modified to meet a business requirement to keep every financial asset class separate, or share all commercial papers and bonds?

Channels

If a peer is joined to multiple channels, then a new blockchain is created and managed for each channel. Moreover, every time a chaincode is deployed to a new channel, a new world state database is created for it. It means that the channel also forms a kind of namespace alongside that of the chaincode for the world state.

However, the same peer and chaincode container processes can be simultaneously joined to multiple channels – unlike blockchains, and world state databases, these processes do not increase with the number of channels joined.

For example, if you deployed the `papers` and `bonds` chaincode to a new channel, there would a totally separate blockchain created, and two new world state databases created. However, the peer and chaincode containers would not increase; each would just be connected to multiple channels.

Usage

Let's use our commercial paper *example* to show how an application uses a smart contract with namespaces. It's worth noting that an application communicates with the peer, and the peer routes the request to the appropriate chaincode container which then accesses the DBMS. This routing is done by the peer **core** component shown in the diagram.

Here's the code for an application that uses both commercial papers and bonds, priced in Euros and Yen. The code is fairly self-explanatory:

```
const euroPaper = network.getContract(papers, euroPaper);
paper1 = euroPaper.submit(issue, PAP11);

const yenPaper = network.getContract(papers, yenPaper);
paper2 = yenPaper.submit(redeem, PAP21);

const euroBond = network.getContract(bonds, euroBond);
bond1 = euroBond.submit(buy, BON31);

const yenBond = network.getContract(bonds, yenBond);
bond2 = yenBond.submit(sell, BON41);
```

See how the application:

- Accesses the `euroPaper` and `yenPaper` contracts using the `getContract()` API specifying the `papers` chaincode. See interaction points **1a** and **2a**.
- Accesses the `euroBond` and `yenBond` contracts using the `getContract()` API specifying the `bonds` chaincode. See interaction points **3a** and **4a**.
- Submits an `issue` transaction to the network for commercial paper `PAP11` using the `euroPaper` contract. See interaction point **1a**. This results in the creation of a commercial paper represented by state `PAP11` in `world state A`; interaction point **1b**. This operation is captured as a transaction in the blockchain at interaction point **1c**.
- Submits a `redeem` transaction to the network for commercial paper `PAP21` using the `yenPaper` contract. See interaction point **2a**. This results in the creation of a commercial paper represented by state `PAP21` in `world state A`; interaction point **2b**. This operation is captured as a transaction in the blockchain at interaction point **2c**.
- Submits a `buy` transaction to the network for bond `BON31` using the `euroBond` contract. See interaction point **3a**. This results in the creation of a bond represented by state `BON31` in `world state B`; interaction point **3b**. This operation is captured as a transaction in the blockchain at interaction point **3c**.
- Submits a `sell` transaction to the network for bond `BON41` using the `yenBond` contract. See interaction point **4a**. This results in the creation of a bond represented by state `BON41` in `world state B`; interaction point **4b**. This operation is captured as a transaction in the blockchain at interaction point **4c**.

See how smart contracts interact with the world state:

- `euroPaper` and `yenPaper` contracts can directly access `world state A`, but cannot directly access `world state B`. `World state A` is physically held in the `papers` database in the database management system (DBMS) corresponding to the `papers` chaincode.
- `euroBond` and `yenBond` contracts can directly access `world state B`, but cannot directly access `world state A`. `World state B` is physically held in the `bonds` database in the database management system (DBMS) corresponding to the `bonds` chaincode.

See how the blockchain captures transactions for all world states:

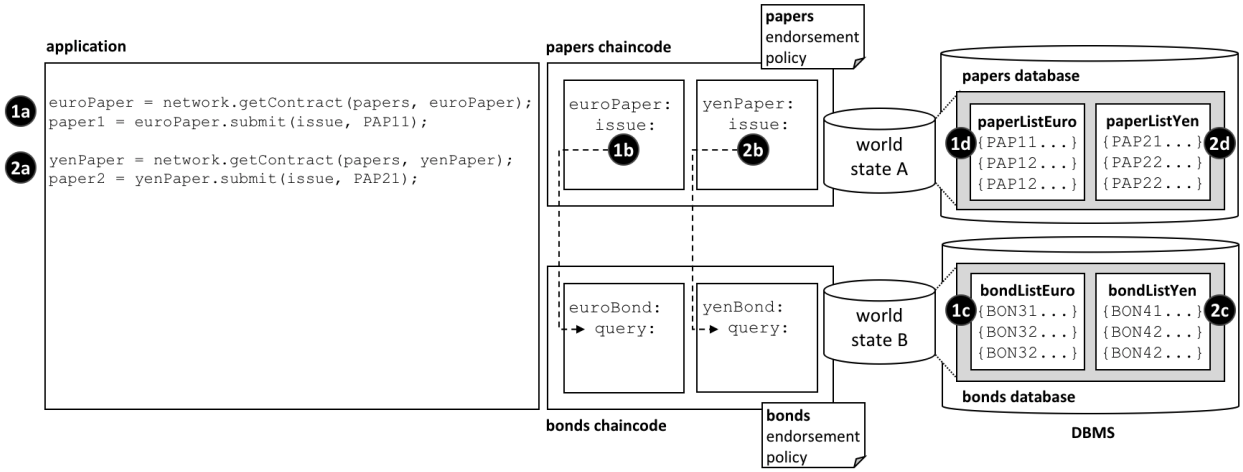
- Interactions **1c** and **2c** correspond to transactions create and update commercial papers `PAP11` and `PAP21` respectively. These are both contained within `world state A`.

- Interactions **3c** and **4c** correspond to transactions both update bonds `BON31` and `BON41`. These are both contained within `world state B`.
- If `world state A` or `world state B` were destroyed for any reason, they could be recreated by replaying all the transactions in the blockchain.

Cross chaincode access

As we saw in our example *scenario*, `euroPaper` and `yenPaper` cannot directly access `world state B`. That's because we have designed our chaincodes and smart contracts so that these chaincodes and world states are kept separately from each other. However, let's imagine that `euroPaper` needs to access `world state B`.

Why might this happen? Imagine that when a commercial paper was issued, the smart contract wanted to price the paper according to the current return on bonds with a similar maturity date. In this case it will be necessary for the `euroPaper` contract to be able to query the price of bonds in `world state B`. Look at the following diagram to see how we might structure this interaction.



How chaincodes and smart contracts can indirectly access another world state – via its chaincode.

Notice how:

- the application submits an `issue` transaction in the `euroPaper` smart contract to issue `PAP11`. See interaction **1a**.
- the `issue` transaction in the `euroPaper` smart contract calls the `query` transaction in the `euroBond` smart contract. See interaction point **1b**.
- the `query`in `euroBond` can retrieve information from `world state B`. See interaction point **1c**.
- when control returns to the `issue` transaction, it can use the information in the response to price the paper and update `world state A` with information. See interaction point **1d**.
- the flow of control for issuing commercial paper priced in Yen is the same. See interaction points **2a**, **2b**, **2c** and **2d**.

Control is passed between chaincode using the `invokeChaincode()` API.

This API passes control from one chaincode to another chaincode.

Although we have only discussed query transactions in the example, it is possible to invoke a smart contract which will update the called chaincode's world state. See the *considerations* below.

Considerations

- In general, each chaincode will have a single smart contract in it.
- Multiple smart contracts should only be deployed in the same chaincode if they are very closely related. Usually, this is only necessary if they share the same world state.
- Chaincode namespaces provide isolation between different world states. In general it makes sense to isolate unrelated data from each other. Note that you cannot choose the chaincode namespace; it is assigned by Hyperledger Fabric, and maps directly to the name of the chaincode.
- For chaincode to chaincode interactions using the `invokeChaincode()` API, both chaincodes must be installed on the same peer.
  - For interactions that only require the called chaincode's world state to be queried, the invocation can be in a different channel to the caller's chaincode.
  - For interactions that require the called chaincode's world state to be updated, the invocation must be in the same channel as the caller's chaincode.

6.6.3 Transaction context

Audience: Architects, application and smart contract developers

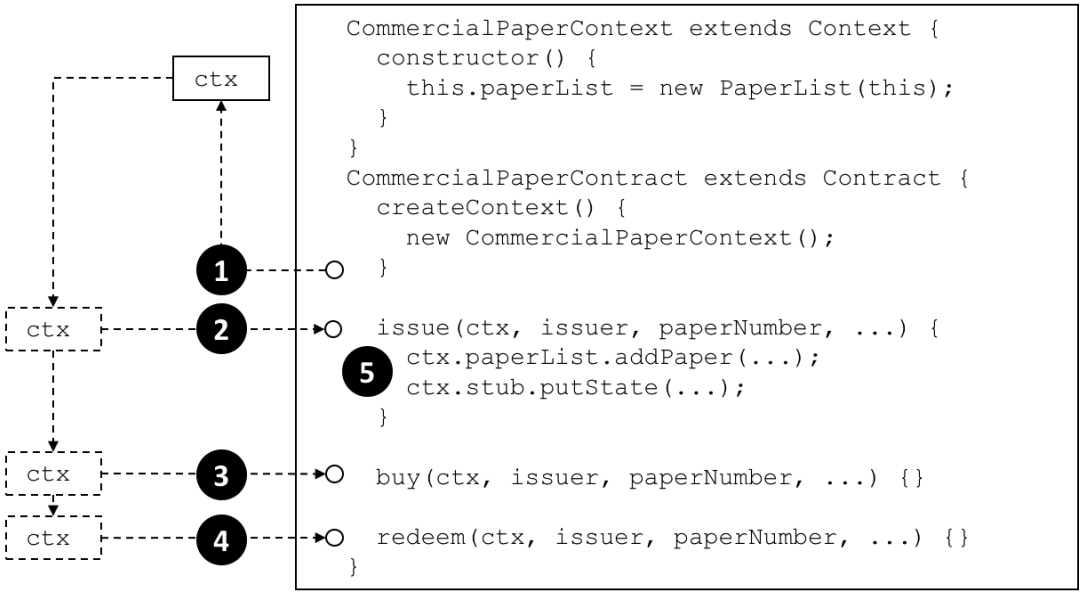
A transaction context performs two functions. Firstly, it allows a developer to define and maintain user variables across transaction invocations within a smart contract. Secondly, it provides access to a wide range of Fabric APIs that allow smart contract developers to perform operations relating to detailed transaction processing. These range from querying or updating the ledger, both the immutable blockchain and the modifiable world state, to retrieving the transaction-submitting application's digital identity.

A transaction context is created when a smart contract is deployed to a channel and made available to every subsequent transaction invocation. A transaction context helps smart contract developers write programs that are powerful, efficient and easy to reason about.

- *Why a transaction context is important*
- *How to use a transaction context*
- *What's in a transaction context*
- *Using a context* `stub`
- *Using a context* `clientIdentity`

Scenario

In the commercial paper sample, papercontract initially defines the name of the list of commercial papers for which it's responsible. Each transaction subsequently refers to this list; the issue transaction adds new papers to it, the buy transaction changes its owner, and the redeem transaction marks it as complete. This is a common pattern; when writing a smart contract it's often helpful to initialize and recall particular variables in sequential transactions.



A smart contract transaction context allows smart contracts to define and maintain user variables across transaction invocations. Refer to the text for a detailed explanation.

Programming

When a smart contract is constructed, a developer can optionally override the built-in `Context` class `createContext` method to create a custom context:

```
createContext() {
    new CommercialPaperContext();
}
```

In our example, the `CommercialPaperContext` is specialized for `CommercialPaperContract`. See how the custom context, addressed through `this`, adds the specific variable `PaperList` to itself:

```
CommercialPaperContext extends Context {
    constructor () {
        this.paperList = new PaperList(this);
    }
}
```

When the createContext() method returns at point **(1)** in the diagram *above*, a custom context `ctx` has been created which contains `paperList` as one of its variables.

Subsequently, whenever a smart contract transaction such as issue, buy or redeem is called, this context will be passed to it. See how at points **(2)**, **(3)** and **(4)** the same commercial paper context is passed into the transaction method using the `ctx` variable.

See how the context is then used at point **(5)**:

```
ctx.paperList.addPaper(...);
ctx.stub.putState(...);
```

Notice how `paperList` created in `CommercialPaperContext` is available to the issue transaction. See how `paperList` is similarly used by the **redeem** and **buy** transactions; `ctx` makes the smart contracts efficient and easy to reason about.

You can also see that there's another element in the context – `ctx.stub` – which was not explictly added by `CommercialPaperContext`. That's because `stub` and other variables are part of the built-in context. Let's now examine the structure of this built-in context, these implicit variables and how to use them.

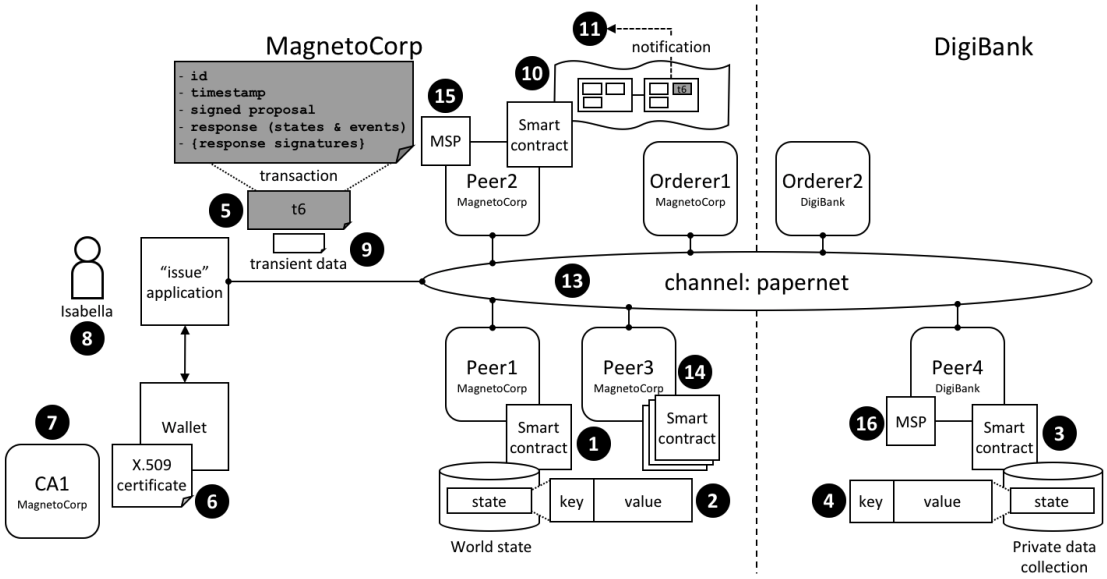
Structure

As we've seen from the *example*, a transaction context can contain any number of user variables such as `paperList`.

The transaction context also contains two built-in elements that provide access to a wide range of Fabric functionality ranging from the client application that submitted the transaction to ledger access.

- `ctx.stub` is used to access APIs that provide a broad range of transaction processing operations from `putState()` and `getState()` to access the ledger, to `getTxID()` to retrieve the current transaction ID.
- `ctx.clientIdentity` is used to get information about the identity of the user who submitted the transaction.

We'll use the following diagram to show you what a smart contract can do using the `stub` and `clientIdentity` using the APIs available to it:



A smart contract can access a range of functionality in a smart contract via the transaction context `stub` and `clientIdentity`. Refer to the text for a detailed explanation.

Stub

The APIs in the stub fall into the following categories:

- **World state data APIs**. See interaction point **(1)**. These APIs enable smart contracts to get, put and delete state corresponding to individual objects from the world state, using their key:
  - getState()
  - putState()
  - deleteState()

  These basic APIs are complemented by query APIs which enable contracts to retrieve a set of states, rather than an individual state. See interaction point **(2)**. The set is either defined by a range of key values, using full or

partial keys, or a query according to values in the underlying world state database. For large queries, the result sets can be paginated to reduce storage requirements:

  - getStateByRange()
  - getStateByRangeWithPagination()
  - getStateByPartialCompositeKey()
  - getStateByPartialCompositeKeyWithPagination()
  - getQueryResult()
  - getQueryResultWithPagination()

- **Private data APIs**. See interaction point **(3)**. These APIs enable smart contracts to interact with a private data collection. They are analogous to the APIs for world state interactions, but for private data. There are APIs to get, put and delete a private data state by its key:

  - getPrivateData()
  - putPrivateData()
  - deletePrivateData()

  This set is complemented by set of APIs to query private data **(4)**. These APIs allow smart contracts to retrieve a set of states from a private data collection, according to a range of key values, either full or partial keys, or a query according to values in the underlying world state database. There are currently no pagination APIs for private data collections.

  - getPrivateDataByRange()
  - getPrivateDataByPartialCompositeKey()
  - getPrivateDataQueryResult()

- **Transaction APIs**. See interaction point **(5)**. These APIs are used by a smart contract to retrieve details about the current transaction proposal being processed by the smart contract. This includes the transaction identifier and the time when the transaction proposal was created.

  - getTxID() returns the identifier of the current transaction proposal **(5)**.
  - getTxTimestamp() returns the timestamp when the current transaction proposal was created by the application **(5)**.
  - getCreator() returns the raw identity (X.509 or otherwise) of the creator of transaction proposal. If this is an X.509 certificate then it is often more appropriate to use `ctx.ClientIdentity`.
  - getSignedProposal() returns a signed copy of the current transaction proposal being processed by the smart contract.
  - getBinding() is used to prevent transactions being maliciously or accidentally replayed using a nonce. (For practical purposes, a nonce is a random number generated by the client application and incorporated in a cryptographic hash.) For example, this API could be used by a smart contract at **(1)** to detect a replay of the transaction **(5)**.
  - getTransient() allows a smart contract to access the transient data an application passes to a smart contract. See interaction points **(9)** and **(10)**. Transient data is private to the application-smart contract interaction. It is not recorded on the ledger and is often used in conjunction with private data collections **(3)**.

- **Key APIs** are used by smart contracts to manipulate state key in the world state or a private data collection. See interaction points **2** and **4**.

  The simplest of these APIs allows smart contracts to form and split composite keys from their individual components. Slightly more advanced are the `ValidationParameter()` APIs which get and set the state based

endorsement policies for world state **(2)** and private data **(4)**. Finally, `getHistoryForKey()` retrieves the history for a state by returning the set of stored values, including the transaction identifiers that performed the state update, allowing the transactions to be read from the blockchain **(10)**.

  - createCompositeKey()
  - splitCompositeKey()
  - setStateValidationParameter()
  - getStateValidationParameter()
  - getPrivateDataValidationParameter()
  - setPrivateDataValidationParameter()
  - getHistoryForKey()

- **Event APIs** are used to manage event processing in a smart contract.
  - setEvent()

```
  Smart contracts use this API to add user events to a transaction response.
  See interaction point **(5)**. These events are ultimately recorded on the
  blockchain and sent to listening applications at interaction point
  **(11)**.

<br>
```

- **Utility APIs** are a collection of useful APIs that don't easily fit in a pre-defined category, so we've grouped them together! They include retrieving the current channel name and passing control to a different chaincode on the same peer.
  - getChannelID()

    See interaction point **(13)**. A smart contract running on any peer can use this API to determined on which channel the application invoked the smart contract.
  - invokeChaincode()

    See interaction point **(14)**. Peer3 owned by MagnetoCorp has multiple smart contracts installed on it. These smart contracts are able to call each other using this API. The smart contracts must be collocated; it is not possible to call a smart contract on a different peer.

  Some of these utility APIs are only used if you're using low-level chaincode, rather than smart contracts. These APIs are primarily for the detailed manipulation of chaincode input; the smart contract `Contract` class does all of this parameter marshalling automatically for developers.

  - getFunctionAndParameters()
  - getStringArgs()
  - getArgs()

ClientIdentity

In most cases, the application submitting a transaction will be using an X.509 certificate. In the *example*, an X.509 certificate **(6)** issued by `CA1` **(7)** is being used by `Isabella` **(8)** in her application to sign the proposal in transaction `t6` **(5)**.

`ClientIdentity` takes the information returned by `getCreator()` and puts a set of X.509 utility APIs on top of it to make it easier to use for this common use case.

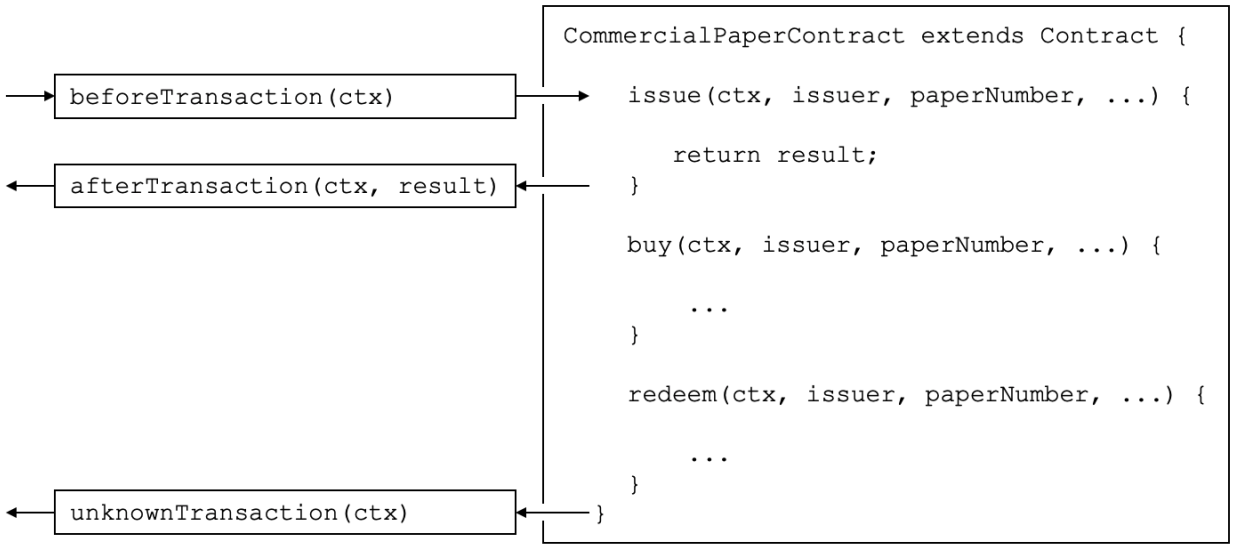
- getX509Certificate() returns the full X.509 certificate of the transaction submitter, including all its attributes and their values. See interaction point **(6)**.
- getAttributeValue() returns the value of a particular X.509 attribute, for example, the organizational unit `OU`, or distinguished name `DN`. See interaction point **(6)**.
- assertAttributeValue() returns `TRUE` if the specified attribute of the X.509 attribute has a specified value. See interaction point **(6)**.
- getID() returns the unique identity of the transaction submitter, according to their distinguished name and the issuing CA's distinguished name. The format is `x509::{subject DN}::{issuer DN}`. See interaction point **(6)**.
- getMSPID() returns the channel MSP of the transaction submitter. This allows a smart contract to make processing decisions based on the submitter's organizational identity. See interaction point **(15)** or **(16)**.

6.6.4 Transaction handlers

Audience: Architects, Application and smart contract developers

Transaction handlers allow smart contract developers to define common processing at key points during the interaction between an application and a smart contract. Transaction handlers are optional but, if defined, they will receive control before or after every transaction in a smart contract is invoked. There is also a specific handler which receives control when a request is made to invoke a transaction not defined in a smart contract.

Here's an example of transaction handlers for the commercial paper smart contract sample:



*Before, After and Unknown transaction handlers. In this example, `beforeTransaction()` is called before the **issue**, **buy** and **redeem** transactions. `afterTransaction()` is called after the **issue**, **buy** and **redeem** transactions. `unknownTransaction()` is only called if a request is made to invoke a transaction not defined in the smart contract. (The diagram is simplified by not repeating `beforeTransaction` and `afterTransaction` boxes for each transaction.)*

Types of handler

There are three types of transaction handlers which cover different aspects of the interaction between an application and a smart contract:

- **Before handler**: is called before every smart contract transaction is invoked. The handler will usually modify the transaction context to be used by the transaction. The handler has access to the full range of Fabric APIs; for example, it can issue `getState()` and `putState()`.
- **After handler**: is called after every smart contract transaction is invoked. The handler will usually perform post-processing common to all transactions, and also has full access to the Fabric APIs.
- **Unknown handler**: is called if an attempt is made to invoke a transaction that is not defined in a smart contract. Typically, the handler will record the failure for subsequent processing by an administrator. The handler has full access to the Fabric APIs.

Defining a transaction handler is optional; a smart contract will perform correctly without handlers being defined. A smart contract can define at most one handler of each type.

Defining a handler

Transaction handlers are added to the smart contract as methods with well defined names. Here's an example which adds a handler of each type:

```
CommercialPaperContract extends Contract {

    ...

    async beforeTransaction(ctx) {
        // Write the transaction ID as an informational to the console
        console.info(ctx.stub.getTxID());
    };

    async afterTransaction(ctx, result) {
        // This handler interacts with the ledger
        ctx.stub.cpList.putState(...);
    };

    async unknownTransaction(ctx) {
        // This handler throws an exception
        throw new Error('Unknown transaction function');
    };

}
```

The form of a transaction handler definition is the similar for all handler types, but notice how the `afterTransaction(ctx, result)` also receives any result returned by the transaction. The API documentation shows you the exact form of these handlers.

Handler processing

Once a handler has been added to the smart contract, it will be invoked during transaction processing. During processing, the handler receives `ctx`, the transaction context, performs some processing, and returns control as it completes. Processing continues as follows:

- **Before handler**: If the handler completes successfully, the transaction is called with the updated context. If the handler throws an exception, then the transaction is not called and the smart contract fails with the exception error message.
- **After handler**: If the handler completes successfully, then the smart contract completes as determined by the invoked transaction. If the handler throws an exception, then the transaction fails with the exception error message.

- **Unknown handler**: The handler should complete by throwing an exception with the required error message. If an **Unknown handler** is not specified, or an exception is not thrown by it, there is sensible default processing; the smart contract will fail with an **unknown transaction** error message.

If the handler requires access to the function and parameters, then it is easy to do this:

```
async beforeTransaction(ctx) {
    // Retrieve details of the transaction
    let txnDetails = ctx.stub.getFunctionAndParameters();

    console.info(`Calling function: ${txnDetails.fcn} `);
    console.info(util.format(`Function arguments : %j ${stub.getArgs()} ``);
}
```

See how this handler uses the utility API `getFunctionAndParameters` via the transaction context.

Multiple handlers

It is only possible to define at most one handler of each type for a smart contract. If a smart contract needs to invoke multiple functions during before, after or unknown handling, it should coordinate this from within the appropriate function.

6.6.5 Endorsement policies

Audience: Architects, Application and smart contract developers

Endorsement policies define the smallest set of organizations that are required to endorse a transaction in order for it to be valid. To endorse, an organization's endorsing peer needs to run the smart contract associated with the transaction and sign its outcome. When the ordering service sends the transaction to the committing peers, they will each individually check whether the endorsements in the transaction fulfill the endorsement policy. If this is not the case, the transaction is invalidated and it will have no effect on world state.

Endorsement policies work at two different granularities: they can be set for an entire namespace, as well as for individual state keys. They are formulated using basic logic expressions such as `AND` and `OR`. For example, in PaperNet this could be used as follows: the endorsement policy for a paper that has been sold from MagnetoCorp to DigiBank could be set to `AND(MagnetoCorp.peer, DigiBank.peer)`, requiring any changes to this paper to be endorsed by both MagnetoCorp and DigiBank.

6.6.6 Connection Profile

Audience: Architects, application and smart contract developers

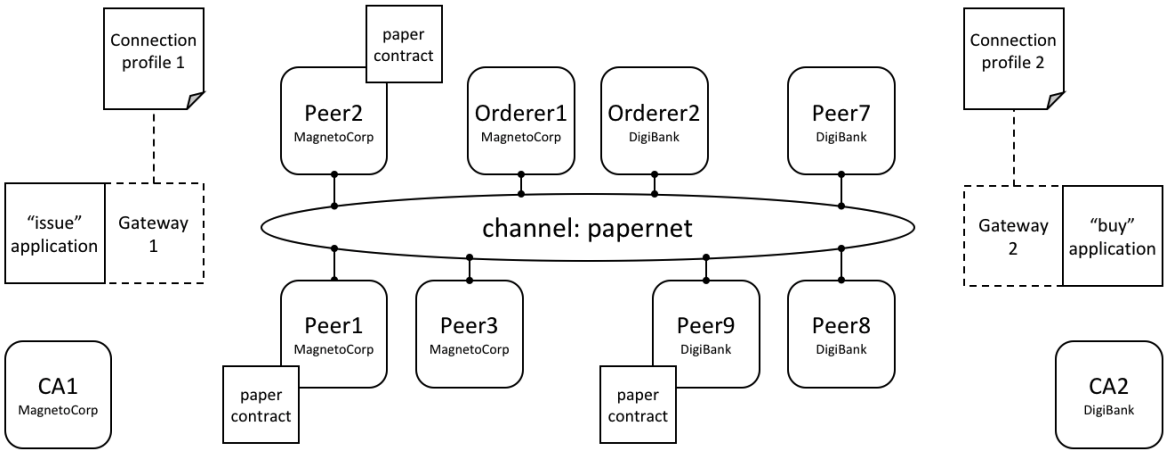
A connection profile describes a set of components, including peers, orderers and certificate authorities in a Hyperledger Fabric blockchain network. It also contains channel and organization information relating to these components. A connection profile is primarily used by an application to configure a gateway that handles all network interactions, allowing it to focus on business logic. A connection profile is normally created by an administrator who understands the network topology.

In this topic, we're going to cover:

- *Why connection profiles are important*
- *How applications use a connection profile*
- *How to define a connection profile*

Scenario

A connection profile is used to configure a gateway. Gateways are important for many reasons, the primary being to simplify an application's interaction with a network channel.



Two applications, issue and buy, use gateways 1&2 configured with connection profiles 1&2. Each profile describes a different subset of MagnetoCorp and DigiBank network components. Each connection profile must contain sufficient information for a gateway to interact with the network on behalf of the issue and buy applications. See the text for a detailed explanation.

A connection profile contains a description of a network view, expressed in a technical syntax, which can either be JSON or YAML. In this topic, we use the YAML representation, as it's easier for you to read. Static gateways need more information than dynamic gateways because the latter can use service discovery to dynamically augment the information in a connection profile.

A connection profile should not be an exhaustive description of a network channel; it just needs to contain enough information sufficient for a gateway that's using it. In the network above, connection profile 1 needs to contain at least the endorsing organizations and peers for the `issue` transaction, as well as identifying the peers that will notify the gateway when the transaction has been committed to the ledger.

It's easiest to think of a connection profile as describing a *view* of the network. It could be a comprehensive view, but that's unrealistic for a few reasons:

- Peers, orderers, certificate authorities, channels, and organizations are added and removed according to demand.
- Components can start and stop, or fail unexpectedly (e.g. power outage).
- A gateway doesn't need a view of the whole network, only what's necessary to successfully handle transaction submission or event notification for example.
- Service Discovery can augment the information in a connection profile. Specifically, dynamic gateways can be configured with minimal Fabric topology information; the rest can be discovered.

A static connection profile is normally created by an administrator who understands the network topology in detail. That's because a static profile can contain quite a lot of information, and an administrator needs to capture this in the corresponding connection profile. In contrast, dynamic profiles minimize the amount of definition required and therefore can be a better choice for developers who want to get going quickly, or administrators who want to create a more responsive gateway. Connection profiles are created in either the YAML or JSON format using an editor of choice.

Usage

We'll see how to define a connection profile in a moment; let's first see how it is used by a sample MagnetoCorp `issue` application:

```
const yaml = require('js-yaml');
const { Gateway } = require('fabric-network');

const connectionProfile = yaml.safeLoad(fs.readFileSync('../gateway/paperNet.yaml',
↪'utf8'));

const gateway = new Gateway();

await gateway.connect(connectionProfile, connectionOptions);
```

After loading some required classes, see how the `paperNet.yaml` gateway file is loaded from the file system, converted to a JSON object using the `yaml.safeLoad()` method, and used to configure a gateway using its `connect()` method.

By configuring a gateway with this connection profile, the issue application is providing the gateway with the relevant network topology it should use to process transactions. That's because the connection profile contains sufficient information about the PaperNet channels, organizations, peers, orderers and CAs to ensure transactions can be successfully processed.

It's good practice for a connection profile to define more than one peer for any given organization – it prevents a single point of failure. This practice also applies to dynamic gateways; to provide more than one starting point for service discovery.

A DigiBank `buy` application would typically configure its gateway with a similar connection profile, but with some important differences. Some elements will be the same, such as the channel; some elements will overlap, such as the endorsing peers. Other elements will be completely different, such as notification peers or certificate authorities for example.

The `connectionOptions` passed to a gateway complement the connection profile. They allow an application to declare how it would like the gateway to use the connection profile. They are interpreted by the SDK to control interaction patterns with network components, for example to select which identity to connect with, or which peers to use for event notifications. Read about the list of available connection options and when to use them.

Structure

To help you understand the structure of a connection profile, we're going to step through an example for the network shown *above*. Its connection profile is based on the PaperNet commercial paper sample, and stored in the GitHub repository. For convenience, we've reproduced it *below*. You will find it helpful to display it in another browser window as you now read about it:

- Line 9: `name: "papernet.magnetocorp.profile.sample"`

  This is the name of the connection profile. Try to use DNS style names; they are a very easy way to convey meaning.

- Line 16: `x-type: "hlfv1"`

  Users can add their own `x-` properties that are "application-specific" – just like with HTTP headers. They are provided primarily for future use.

- Line 20: `description: "Sample connection profile for documentation topic"`

  A short description of the connection profile. Try to make this helpful for the reader who might be seeing this for the first time!

- Line 25: `version: "1.0"`

  The schema version for this connection profile. Currently only version 1.0 is supported, and it is not envisioned that this schema will change frequently.

- Line 32: `channels:`

  This is the first really important line. `channels:` identifies that what follows are *all* the channels that this connection profile describes. However, it is good practice to keep different channels in different connection profiles, especially if they are used independently of each other.

- Line 36: `papernet:`

  Details of `papernet`, the first channel in this connection profile, will follow.

- Line 41: `orderers:`

  Details of all the orderers for `papernet` follow. You can see in line 45 that the orderer for this channel is `orderer1.magnetocorp.example.com`. This is just a logical name; later in the connection profile (lines 134 - 147), there will be details of how to connect to this orderer. Notice that `orderer2.digibank.example.com` is not in this list; it makes sense that applications use their own organization's orderers, rather than those from a different organization.

- Line 49: `peers:`

  Details of all the peers for `papernet` will follow.

  You can see three peers listed from MagnetoCorp: `peer1.magnetocorp.example.com`, `peer2.magnetocorp.example.com` and `peer3.magnetocorp.example.com`. It's not necessary to list all the peers in MagnetoCorp, as has been done here. You can see only one peer listed from DigiBank: `peer9.digibank.example.com`; including this peer starts to imply that the endorsement policy requires MagnetoCorp and DigiBank to endorse transactions, as we'll now confirm. It's good practice to have multiple peers to avoid single points of failure.

  Underneath each peer you can see four non-exclusive roles: **endorsingPeer**, **chaincodeQuery**, **ledgerQuery** and **eventSource**. See how `peer1` and `peer2` can perform all roles as they host `papercontract`. Contrast to `peer3`, which can only be used for notifications, or ledger queries that access the blockchain component of the ledger rather than the world state, and hence do not need to have smart contracts installed. Notice how `peer9` should not be used for anything other than endorsement, because those roles are better served by MagnetoCorp peers.

  Again, see how the peers are described according to their logical names and their roles. Later in the profile, we'll see the physical information for these peers.

- Line 97: `organizations:`

  Details of all the organizations will follow, for all channels. Note that these organizations are for all channels, even though `papernet` is currently the only one listed. That's because organizations can be in multiple channels, and channels can have multiple organizations. Moreover, some application operations relate to organizations rather than channels. For example, an application can request notification from one or all peers within its organization, or all organizations within the network – using connection options. For this, there needs to be an organization to peer mapping, and this section provides it.

- Line 101: `MagnetoCorp:`

  All peers that are considered part of MagnetoCorp are listed: `peer1`, `peer2` and `peer3`. Likewise for Certificate Authorities. Again, note the logical name usages, the same as the `channels:` section; physical information will follow later in the profile.

- Line 121: `DigiBank:`

  Only `peer9` is listed as part of DigiBank, and no Certificate Authorities. That's because these other peers and the DigiBank CA are not relevant for users of this connection profile.

- Line 134: `orderers:`

  The physical information for orderers is now listed. As this connection profile only mentioned one orderer for `papernet`, you see `orderer1.magnetocorp.example.com` details listed. These include its IP address and port, and gRPC options that can override the defaults used when communicating with the orderer, if necessary. As with `peers:`, for high availability, specifying more than one orderer is a good idea.

- Line 152: `peers:`

  The physical information for all previous peers is now listed. This connection profile has three peers for MagnetoCorp: `peer1`, `peer2`, and `peer3`; for DigiBank, a single peer `peer9` has its information listed. For each peer, as with orderers, their IP address and port is listed, together with gRPC options that can override the defaults used when communicating with a particular peer, if necessary.

- Line 194: `certificateAuthorities:`

  The physical information for certificate authorities is now listed. The connection profile has a single CA listed for MagnetoCorp, `ca1-magnetocorp`, and its physical information follows. As well as IP details, the registrar information allows this CA to be used for Certificate Signing Requests (CSR). These are used to request new certificates for locally generated public/private key pairs.

Now you've understood a connection profile for MagnetoCorp, you might like to look at a corresponding profile for DigiBank. Locate where the profile is the same as MagnetoCorp's, see where it's similar, and finally where it's different. Think about why these differences make sense for DigiBank applications.

That's everything you need to know about connection profiles. In summary, a connection profile defines sufficient channels, organizations, peers, orderers and certificate authorities for an application to configure a gateway. The gateway allows the application to focus on business logic rather than the details of the network topology.

Sample

This file is reproduced inline from the GitHub commercial paper sample.

```
1: ---
2: #
3: # [Required]. A connection profile contains information about a set of network
4: # components. It is typically used to configure gateway, allowing applications
5: # interact with a network channel without worrying about the underlying
6: # topology. A connection profile is normally created by an administrator who
7: # understands this topology.
8: #
9: name: "papernet.magnetocorp.profile.sample"
10: #
11: # [Optional]. Analogous to HTTP, properties with an "x-" prefix are deemed
12: # "application-specific", and ignored by the gateway. For example, property
13: # "x-type" with value "hlfv1" was originally used to identify a connection
14: # profile for Fabric 1.x rather than 0.x.
15: #
16: x-type: "hlfv1"
17: #
18: # [Required]. A short description of the connection profile
19: #
20: description: "Sample connection profile for documentation topic"
21: #
22: # [Required]. Connection profile schema version. Used by the gateway to
23: # interpret these data.
24: #
25: version: "1.0"
```

(continues on next page)

(continued from previous page)

```
#
# [Optional]. A logical description of each network channel; its peer and
# orderer names and their roles within the channel. The physical details of
# these components (e.g. peer IP addresses) will be specified later in the
# profile; we focus first on the logical, and then the physical.
#
channels:
  #
  # [Optional]. papernet is the only channel in this connection profile
  #
  papernet:
    #
    # [Optional]. Channel orderers for PaperNet. Details of how to connect to
    # them is specified later, under the physical "orderers:" section
    #
    orderers:
    #
    # [Required]. Orderer logical name
    #
      - orderer1.magnetocorp.example.com
    #
    # [Optional]. Peers and their roles
    #
    peers:
    #
    # [Required]. Peer logical name
    #
      peer1.magnetocorp.example.com:
        #
        # [Optional]. Is this an endorsing peer? (It must have chaincode
        # installed.) Default: true
        #
        endorsingPeer: true
        #
        # [Optional]. Is this peer used for query? (It must have chaincode
        # installed.) Default: true
        #
        chaincodeQuery: true
        #
        # [Optional]. Is this peer used for non-chaincode queries? All peers
        # support these types of queries, which include queryBlock(),
        # queryTransaction(), etc. Default: true
        #
        ledgerQuery: true
        #
        # [Optional]. Is this peer used as an event hub? All peers can produce
        # events. Default: true
        #
        eventSource: true
      #
      peer2.magnetocorp.example.com:
        endorsingPeer: true
        chaincodeQuery: true
        ledgerQuery: true
        eventSource: true
      #
      peer3.magnetocorp.example.com:
```

(continues on next page)

(continued from previous page)

```
83:         endorsingPeer: false
84:         chaincodeQuery: false
85:         ledgerQuery: true
86:         eventSource: true
87:       #
88:       peer9.digibank.example.com:
89:         endorsingPeer: true
90:         chaincodeQuery: false
91:         ledgerQuery: false
92:         eventSource: false
93: #
94: # [Required]. List of organizations for all channels. At least one organization
95: # is required.
96: #
97: organizations:
98:    #
99:    # [Required]. Organizational information for MagnetoCorp
100:   #
101:   MagnetoCorp:
102:     #
103:     # [Required]. The MSPID used to identify MagnetoCorp
104:     #
105:     mspid: MagnetoCorpMSP
106:     #
107:     # [Required]. The MagnetoCorp peers
108:     #
109:     peers:
110:       - peer1.magnetocorp.example.com
111:       - peer2.magnetocorp.example.com
112:       - peer3.magnetocorp.example.com
113:     #
114:     # [Optional]. Fabric-CA Certificate Authorities.
115:     #
116:     certificateAuthorities:
117:       - ca-magnetocorp
118:   #
119:   # [Optional]. Organizational information for DigiBank
120:   #
121:   DigiBank:
122:     #
123:     # [Required]. The MSPID used to identify DigiBank
124:     #
125:     mspid: DigiBankMSP
126:     #
127:     # [Required]. The DigiBank peers
128:     #
129:     peers:
130:       - peer9.digibank.example.com
131: #
132: # [Optional]. Orderer physical information, by orderer name
133: #
134: orderers:
135:   #
136:   # [Required]. Name of MagnetoCorp orderer
137:   #
138:   orderer1.magnetocorp.example.com:
139:     #
```

(continues on next page)

(continued from previous page)

```
    # [Required]. This orderer's IP address
    #
    url: grpc://localhost:7050
    #
    # [Optional]. gRPC connection properties used for communication
    #
    grpcOptions:
      ssl-target-name-override: orderer1.magnetocorp.example.com
#
# [Required]. Peer physical information, by peer name. At least one peer is
# required.
#
peers:
  #
  # [Required]. First MagetoCorp peer physical properties
  #
  peer1.magnetocorp.example.com:
    #
    # [Required]. Peer's IP address
    #
    url: grpc://localhost:7151
    #
    # [Optional]. gRPC connection properties used for communication
    #
    grpcOptions:
      ssl-target-name-override: peer1.magnetocorp.example.com
      request-timeout: 120001
  #
  # [Optional]. Other MagnetoCorp peers
  #
  peer2.magnetocorp.example.com:
    url: grpc://localhost:7251
    grpcOptions:
      ssl-target-name-override: peer2.magnetocorp.example.com
      request-timeout: 120001
  #
  peer3.magnetocorp.example.com:
    url: grpc://localhost:7351
    grpcOptions:
      ssl-target-name-override: peer3.magnetocorp.example.com
      request-timeout: 120001
  #
  # [Required]. Digibank peer physical properties
  #
  peer9.digibank.example.com:
    url: grpc://localhost:7951
    grpcOptions:
      ssl-target-name-override: peer9.digibank.example.com
      request-timeout: 120001
#
# [Optional]. Fabric-CA Certificate Authority physical information, by name.
# This information can be used to (e.g.) enroll new users. Communication is via
# REST, hence options relate to HTTP rather than gRPC.
#
certificateAuthorities:
  #
  # [Required]. MagnetoCorp CA
```

(continues on next page)

(continued from previous page)

```
  #
  ca1-magnetocorp:
    #
    # [Required]. CA IP address
    #
    url: http://localhost:7054
    #
    # [Optioanl]. HTTP connection properties used for communication
    #
    httpOptions:
      verify: false
    #
    # [Optional]. Fabric-CA supports Certificate Signing Requests (CSRs). A
    # registrar is needed to enroll new users.
    #
    registrar:
      - enrollId: admin
        enrollSecret: adminpw
    #
    # [Optional]. The name of the CA.
    #
    caName: ca-magnetocorp
```

6.6.7 Connection Options

Audience: Architects, administrators, application and smart contract developers

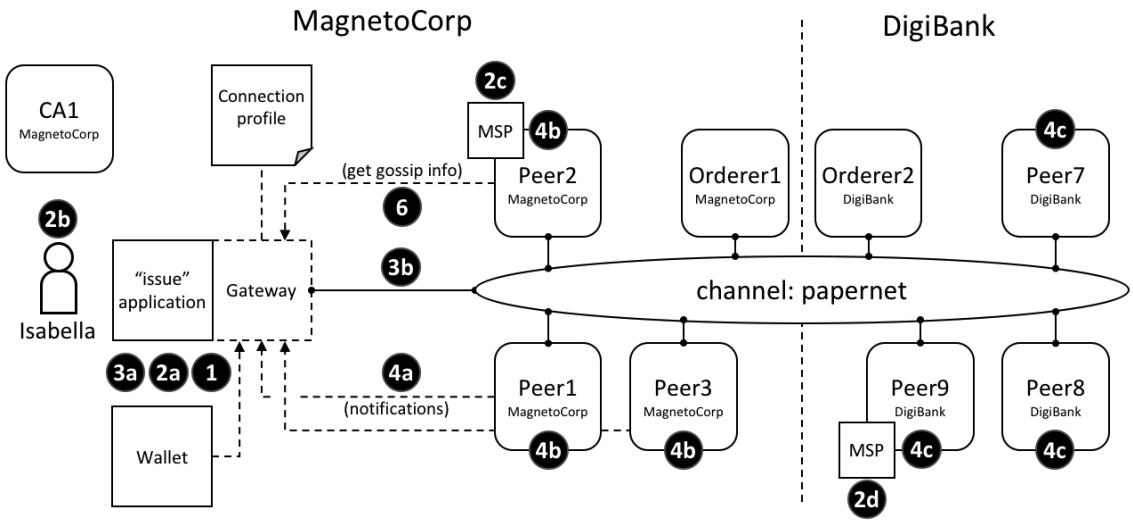
Connection options are used in conjunction with a connection profile to control *precisely* how a gateway interacts with a network. Using a gateway allows an application to focus on business logic rather than network topology.

In this topic, we're going to cover:

- *Why connection options are important*
- *How an application uses connection options*
- *What each connection option does*
- *When to use a particular connection option*

Scenario

A connection option specifies a particular aspect of a gateway's behaviour. Gateways are important for many reasons, the primary being to allow an application to focus on business logic and smart contracts, while it manages interactions with the many components of a network.



The different interaction points where connection options control behaviour. These options are explained fully in the text.

One example of a connection option might be to specify that the gateway used by the `issue` application should use identity `Isabella` to submit transactions to the `papernet` network. Another might be that a gateway should wait for all three nodes from MagnetoCorp to confirm a transaction has been committed returning control. Connection options allow applications to specify the precise behaviour of a gateway's interaction with the network. Without a gateway, applications need to do a lot more work; gateways save you time, make your application more readable, and less error prone.

Usage

We'll describe the *full set* of connection options available to an application in a moment; let's first see how they are specified by the sample MagnetoCorp `issue` application:

```
const userName = 'User1@org1.example.com';
const wallet = new FileSystemWallet('../identity/user/isabella/wallet');

const connectionOptions = {
  identity: userName,
  wallet: wallet,
  eventHandlerOptions: {
    commitTimeout: 100,
    strategy: EventStrategies.MSPID_SCOPE_ANYFORTX
    }
  };

await gateway.connect(connectionProfile, connectionOptions);
```

See how the `identity` and `wallet` options are simple properties of the `connectionOptions` object. They have values `userName` and `wallet` respectively, which were set earlier in the code. Contrast these options with the `eventHandlerOptions` option which is an object in its own right. It has two properties: `commitTimeout: 100` (measured in seconds) and `strategy: EventStrategies.MSPID_SCOPE_ANYFORTX`.

See how `connectionOptions` is passed to a gateway as a complement to `connectionProfile`; the network is identified by the connection profile and the options specify precisely how the gateway should interact with it. Let's now look at the available options.

Options

Here's a list of the available options and what they do.

- `wallet` identifies the wallet that will be used by the gateway on behalf of the application. See interaction **1**; the wallet is specified by the application, but it's actually the gateway that retrieves identities from it.

  A wallet must be specified; the most important decision is the type of wallet to use, whether that's file system, in-memory, HSM or database.

- `identity` is the user identity that the application will use from `wallet`. See interaction **2a**; the user identity is specified by the application and represents the user of the application, Isabella, **2b**. The identity is actually retrieved by the gateway.

  In our example, Isabella's identity will be used by different MSPs (**2c**, **2d**) to identify her as being from MagnetoCorp, and having a particular role within it. These two facts will correspondingly determine her permission over resources, such as being able to read and write the ledger, for example.

  A user identity must be specified. As you can see, this identity is fundamental to the idea that Hyperledger Fabric is a *permissioned* network – all actors have an identity, including applications, peers and orderers, which determines their control over resources. You can read more about this idea in the membership services topic.

- `clientTlsIdentity` is the identity that is retrieved from a wallet (**3a**) and used for secure communications (**3b**) between the gateway and different channel components, such as peers and orderers.

  Note that this identity is different to the user identity. Even though `clientTlsIdentity` is important for secure communications, it is not as foundational as the user identity because its scope does not extend beyond secure network communications.

  `clientTlsIdentity` is optional. You are advised to set it in production environments. You should always use a different `clientTlsIdentity` to `identity` because these identities have very different meanings and lifecycles. For example, if your `clientTlsIdentity` was compromised, then so would your `identity`; it's more secure to keep them separate.

- `eventHandlerOptions.commitTimeout` is optional. It specifies, in seconds, the maximum amount of time the gateway should wait for a transaction to be committed by any peer (**4a**) before returning control to the application. The set of peers to use for notification is determined by the `eventHandlerOptions.strategy` option. If a commitTimeout is not specified, the gateway will use a timeout of 300 seconds.

- `eventHandlerOptions.strategy` is optional. It identifies the set of peers that a gateway should use to listen for notification that a transaction has been committed. For example, whether to listen for a single peer, or all peers, from its organization. It can take one of the following values:

  - `EventStrategies.MSPID_SCOPE_ANYFORTX` Listen for **any** peer within the user's organization. In our example, see interaction points **4b**; any of peer 1, peer 2 or peer 3 from MagnetoCorp can notify the gateway.

  - `EventStrategies.MSPID_SCOPE_ALLFORTX` **This is the default value**. Listen for **all** peers within the user's organization. In our example peer, see interaction point **4b**. All peers from MagnetoCorp must all have notified the gateway; peer 1, peer 2 and peer 3. Peers are only counted if they are known/discovered and available; peers that are stopped or have failed are not included.

  - `EventStrategies.NETWORK_SCOPE_ANYFORTX` Listen for **any** peer within the entire network channel. In our example, see interaction points **4b** and **4c**; any of peer 1-3 from MagnetoCorp or peer 7-9 of DigiBank can notify the gateway.

  - `EventStrategies.NETWORK_SCOPE_ALLFORTX` Listen for **all** peers within the entire network channel. In our example, see interaction points **4b** and **4c**. All peers from MagnetoCorp and DigiBank must notify the gateway; peers 1-3 and peers 7-9. Peers are only counted if they are known/discovered and available; peers that are stopped or have failed are not included.

  - <PluginEventHandlerFunction> The name of a user-defined event handler. This allows a user to define their own logic for event handling. See how to define a plugin event handler, and examine a sample handler.

    A user-defined event handler is only necessary if you have very specific event handling requirements; in general, one of the built-in event strategies will be sufficient. An example of a user-defined event handler might be to wait for more than half the peers in an organization to confirm a transaction has been committed.

    If you do specify a user-defined event handler, it does not affect your application logic; it is quite separate from it. The handler is called by the SDK during processing; it decides when to call it, and uses its results to select which peers to use for event notification. The application receives control when the SDK has finished its processing.

    If a user-defined event handler is not specified then the default values for `EventStrategies` are used.

- `discovery.enabled` is optional and has possible values `true` or `false`. The default is `true`. It determines whether the gateway uses service discovery to augment the network topology specified in the connection profile. See interaction point **6**; peer's gossip information used by the gateway.

  This value will be overridden by the `INITIALIIZE-WITH-DISCOVERY` environment variable, which can be set to `true` or `false`.

- `discovery.asLocalhost` is optional and has possible values `true` or `false`. The default is `true`. It determines whether IP addresses found during service discovery are translated from the docker network to the local host.

  Typically developers will write applications that use docker containers for their network components such as peers, orderers and CAs, but that do not run in docker containers themselves. This is why `true` is the default; in production environments, applications will likely run in docker containers in the same manner as network components and therefore address translation is not required. In this case, applications should either explicitly specify `false` or use the environment variable override.

  This value will be overridden by the `DISCOVERY-AS-LOCALHOST` environment variable, which can be set to `true` or `false`.

Considerations

The following list of considerations is helpful when deciding how to choose connection options.

- `eventHandlerOptions.commitTimeout` and `eventHandlerOptions.strategy` work together. For example, `commitTimeout: 100` and `strategy: EventStrategies.MSPID_SCOPE_ANYFORTX` means that the gateway will wait for up to 100 seconds for *any* peer to confirm a transaction has been committed. In contrast, specifying `strategy: EventStrategies.NETWORK_SCOPE_ALLFORTX` means that the gateway will wait up to 100 seconds for *all* peers in *all* organizations.

- The default value of `eventHandlerOptions.strategy: EventStrategies.MSPID_SCOPE_ALLFORTX` will wait for all peers in the application's organization to commit the transaction. This is a good default because applications can be sure that all their peers have an up-to-date copy of the ledger, minimizing concurrency issues

  However, as the number of peers in an organization grows, it becomes a little unnecessary to wait for all peers, in which case using a pluggable event handler can provide a more efficient strategy. For example the same set of peers could be used to submit transactions and listen for notifications, on the safe assumption that consensus will keep all ledgers synchronized.

- Service discovery requires `clientTlsIdentity` to be set. That's because the peers exchanging information with an application need to be confident that they are exchanging information with entities they trust. If `clientTlsIdentity` is not set, then `discovery` will not be obeyed, regardless of whether or not it is set.
- Although applications can set connection options when they connect to the gateway, it can be necessary for these options to be overridden by an administrator. That's because options relate to network interactions, which can vary over time. For example, an administrator trying to understand the effect of using service discovery on network performance.

  A good approach is to define application overrides in a configuration file which is read by the application when it configures its connection to the gateway.

  Because the discovery options `enabled` and `asLocalHost` are most frequently required to be overridden by administrators, the environment variables `INITIALIIZE-WITH-DISCOVERY` and `DISCOVERY-AS-LOCALHOST` are provided for convenience. The administrator should set these in the production runtime environment of the application, which will most likely be a docker container.

6.6.8 Wallet

Audience: Architects, application and smart contract developers

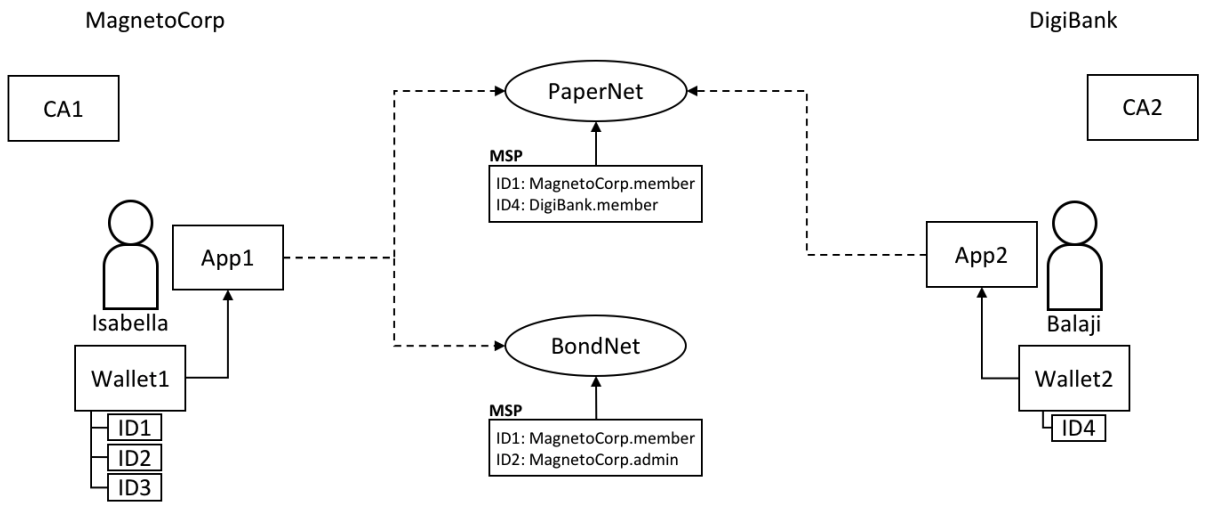
A wallet contains a set of user identities. An application run by a user selects one of these identities when it connects to a channel. Access rights to channel resources, such as the ledger, are determined using this identity in combination with an MSP.

In this topic, we're going to cover:

- *Why wallets are important*
- *How wallets are organized*
- *Different types of wallet*
- *Wallet operations*

Scenario

When an application connects to a network channel such as PaperNet, it selects a user identity to do so, for example `ID1`. The channel MSPs associate `ID1` with a role within a particular organization, and this role will ultimately determine the application's rights over channel resources. For example, `ID1` might identify a user as a member of the MagnetoCorp organization who can read and write to the ledger, whereas `ID2` might identify an administrator in MagnetoCorp who can add a new organization to a consortium.



Two users, Isabella and Balaji have wallets containing different identities they can use to connect to different network channels, PaperNet and BondNet.

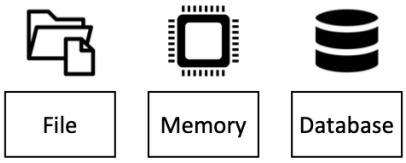
Consider the example of two users; Isabella from MagnetoCorp and Balaji from DigiBank. Isabella is going to use App 1 to invoke a smart contract in PaperNet and a different smart contract in BondNet. Similarly, Balaji is going to use App 2 to invoke smart contracts, but only in PaperNet. (It's very easy for applications to access multiple networks and multiple smart contracts within them.)

See how:

- MagnetoCorp uses CA1 to issue identities and DigiBank uses CA2 to issue identities. These identities are stored in user wallets.
- Balaji's wallet holds a single identity, `ID4` issued by CA2. Isabella's wallet has many identities, `ID1`, `ID2` and `ID3`, issued by CA1. Wallets can hold multiple identities for a single user, and each identity can be issued by a different CA.
- Both Isabella and Balaji connect to PaperNet, and its MSPs determine that Isabella is a member of the MagnetoCorp organization, and Balaji is a member of the DigiBank organization, because of the respective CAs that issued their identities. (It is possible for an organization to use multiple CAs, and for a single CA to support multiple organizations.)
- Isabella can use `ID1` to connect to both PaperNet and BondNet. In both cases, when Isabella uses this identity, she is recognized as a member of MangetoCorp.
- Isabella can use `ID2` to connect to BondNet, in which case she is identified as an administrator of MagnetoCorp. This gives Isabella two very different privileges: `ID1` identifies her as a simple member of MagnetoCorp who can read and write to the BondNet ledger, whereas `ID2` identities her as a MagnetoCorp administrator who can add a new organization to BondNet.
- Balaji cannot connect to BondNet with `ID4`. If he tried to connect, `ID4` would not be recognized as belonging to DigiBank because CA2 is not known to BondNet's MSP.

Types

There are different types of wallets according to where they store their identities:



The three different types of wallet storage: File system, In-memory and CouchDB.

- **File system**: This is the most common place to store wallets; file systems are pervasive, easy to understand, and can be network mounted. They are a good default choice for wallets.
- **In-memory**: A wallet in application storage. Use this type of wallet when your application is running in a constrained environment without access to a file system; typically a web browser. It's worth remembering that this type of wallet is volatile; identities will be lost after the application ends normally or crashes.
- **CouchDB**: A wallet stored in CouchDB. This is the rarest form of wallet storage, but for those users who want to use the database back-up and restore mechanisms, CouchDB wallets can provide a useful option to simplify disaster recovery.

Use factory functions provided by the `Wallets` class to create wallets.

Hardware Security Module

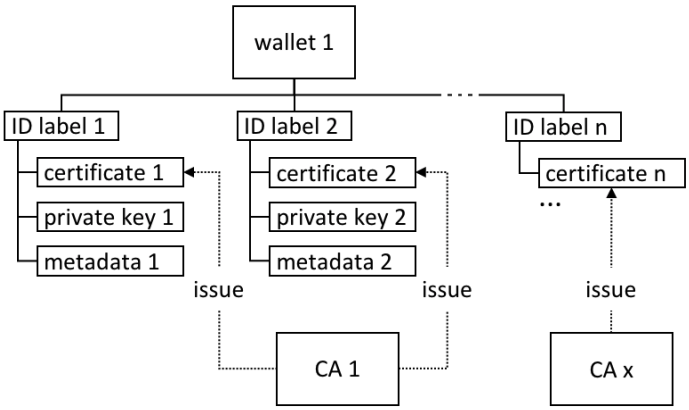
A Hardware Security Module (HSM) is an ultra-secure, tamper-proof device that stores digital identity information, particularly private keys. HSMs can be locally attached to your computer or network accessible. Most HSMs provide the ability to perform on-board encryption with private keys, such that the private keys never leave the HSM.

An HSM can be used with any of the wallet types. In this case the certificate for an identity will be stored in the wallet and the private key will be stored in the HSM.

To enable the use of HSM-managed identities, an `IdentityProvider` must be configured with the HSM connection information and registered with the wallet. For further details, refer to the Using wallets to manage identities tutorial.

Structure

A single wallet can hold multiple identities, each issued by a particular Certificate Authority. Each identity has a standard structure comprising a descriptive label, an X.509 certificate containing a public key, a private key, and some Fabric-specific metadata. Different *wallet types* map this structure appropriately to their storage mechanism.



A Fabric wallet can hold multiple identities with certificates issued by a different Certificate Authority. Identities comprise certificate, private key and Fabric metadata.

There's a couple of key class methods that make it easy to manage wallets and identities:

```
const identity: X509Identity = {
    credentials: {
        certificate: certificatePEM,
        privateKey: privateKeyPEM,
    },
    mspId: 'Org1MSP',
    type: 'X.509',
};
await wallet.put(identityLabel, identity);
```

See how an `identity` is created that has metadata `Org1MSP`, a `certificate` and a `privateKey`. See how `wallet.put()` adds this identity to the wallet with a particular `identityLabel`.

The `Gateway` class only requires the `mspId` and `type` metadata to be set for an identity – `Org1MSP` and `X.509` in the above example. It *currently* uses the MSP ID value to identify particular peers from a connection profile, for example when a specific notification strategy is requested. In the DigiBank gateway file `networkConnection.yaml`, see how `Org1MSP` notifications will be associated with `peer0.org1.example.com`:

```
organizations:
  Org1:
    mspid: Org1MSP

    peers:
      - peer0.org1.example.com
```

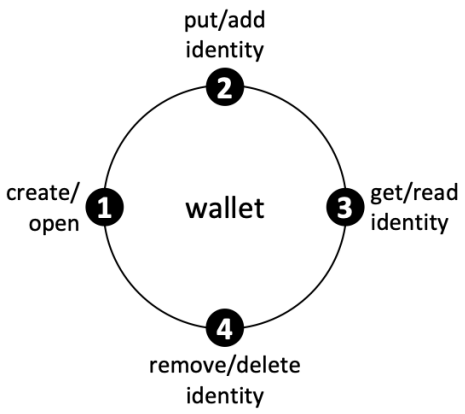
You really don't need to worry about the internal structure of the different wallet types, but if you're interested, navigate to a user identity folder in the commercial paper sample:

```
magnetocorp/identity/user/isabella/
                                    wallet/
                                          User1@org1.example.com.id
```

You can examine these files, but as discussed, it's easier to use the SDK to manipulate these data.

Operations

The different wallet types all implement a common Wallet interface which provides a standard set of APIs to manage identities. It means that applications can be made independent of the underlying wallet storage mechanism; for example, File system and HSM wallets are handled in a very similar way.



Wallets follow a lifecycle: they can be created or opened, and identities can be read, added and deleted.

An application can use a wallet according to a simple lifecycle. Wallets can be opened or created, and subsequently identities can be added, updated, read and deleted. Spend a little time on the different `Wallet` methods in the JSDoc to see how they work; the commercial paper tutorial provides a nice example in `addToWallet.js`:

```
const wallet = await Wallets.newFileSystemWallet('../identity/user/isabella/wallet');

const cert = fs.readFileSync(path.join(credPath, '.../User1@org1.example.com-cert.pem
↪')).toString();
const key = fs.readFileSync(path.join(credPath, '.../_sk')).toString();

const identityLabel = 'User1@org1.example.com';
const identity = {
    credentials: {
        certificate: cert,
        privateKey: key,
    },
    mspId: 'Org1MSP',
    type: 'X.509',
};

await wallet.put(identityLabel, identity);
```

Notice how:

- When the program is first run, a wallet is created on the local file system at `.../isabella/wallet`.
- a certificate `cert` and private `key` are loaded from the file system.
- a new X.509 identity is created with `cert`, `key` and `Org1MSP`.
- the new identity is added to the wallet with `wallet.put()` with a label `User1@org1.example.com`.

That's everything you need to know about wallets. You've seen how they hold identities that are used by applications on behalf of users to access Fabric network resources. There are different types of wallets available depending on your application and security needs, and a simple set of APIs to help applications manage wallets and the identities within them.

6.6.9 Gateway

Audience: Architects, application and smart contract developers

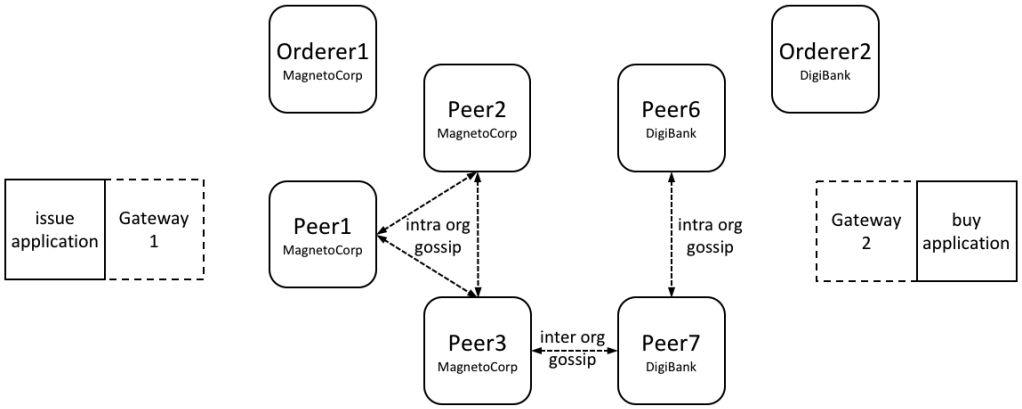
A gateway manages the network interactions on behalf of an application, allowing it to focus on business logic. Applications connect to a gateway and then all subsequent interactions are managed using that gateway's configuration.

In this topic, we're going to cover:

- *Why gateways are important*
- *How applications use a gateway*
- *How to define a static gateway*
- *How to define a dynamic gateway for service discovery*
- *Using multiple gateways*

Scenario

A Hyperledger Fabric network channel can constantly change. The peer, orderer and CA components, contributed by the different organizations in the network, will come and go. Reasons for this include increased or reduced business demand, and both planned and unplanned outages. A gateway relieves an application of this burden, allowing it to focus on the business problem it is trying to solve.



A MagnetoCorp and DigiBank applications (issue and buy) delegate their respective network interactions to their gateways. Each gateway understands the network channel topology comprising the multiple peers and orderers of two organizations MagnetoCorp and DigiBank, leaving applications to focus on business logic. Peers can talk to each other both within and across organizations using the gossip protocol.

A gateway can be used by an application in two different ways:

- **Static**: The gateway configuration is *completely* defined in a connection profile. All the peers, orderers and CAs available to an application are statically defined in the connection profile used to configure the gateway. For peers, this includes their role as an endorsing peer or event notification hub, for example. You can read more about these roles in the connection profile topic.

  The SDK will use this static topology, in conjunction with gateway connection options, to manage the transaction submission and notification processes. The connection profile must contain enough of the network topology to allow a gateway to interact with the network on behalf of the application; this includes the network channels, organizations, orderers, peers and their roles.

- **Dynamic**: The gateway configuration is minimally defined in a connection profile. Typically, one or two peers from the application's organization are specified, and they use service discovery to discover the available network topology. This includes peers, orderers, channels, deployed smart contracts and their endorsement policies. (In production environments, a gateway configuration should specify at least two peers for availability.)

  The SDK will use all of the static and discovered topology information, in conjunction with gateway connection options, to manage the transaction submission and notification processes. As part of this, it will also intelligently

use the discovered topology; for example, it will *calculate* the minimum required endorsing peers using the discovered endorsement policy for the smart contract.

You might ask yourself whether a static or dynamic gateway is better? The trade-off is between predictability and responsiveness. Static networks will always behave the same way, as they perceive the network as unchanging. In this sense they are predictable – they will always use the same peers and orderers if they are available. Dynamic networks are more responsive as they understand how the network changes – they can use newly added peers and orderers, which brings extra resilience and scalability, at potentially some cost in predictability. In general it's fine to use dynamic networks, and indeed this the default mode for gateways.

Note that the *same* connection profile can be used statically or dynamically. Clearly, if a profile is going to be used statically, it needs to be comprehensive, whereas dynamic usage requires only sparse population.

Both styles of gateway are transparent to the application; the application program design does not change whether static or dynamic gateways are used. This also means that some applications may use service discovery, while others may not. In general using dynamic discovery means less definition and more intelligence by the SDK; it is the default.

Connect

When an application connects to a gateway, two options are provided. These are used in subsequent SDK processing:

```
await gateway.connect(connectionProfile, connectionOptions);
```

- **Connection profile**: `connectionProfile` is the gateway configuration that will be used for transaction processing by the SDK, whether statically or dynamically. It can be specified in YAML or JSON, though it must be converted to a JSON object when passed to the gateway:

  ```
  let connectionProfile = yaml.safeLoad(fs.readFileSync('../gateway/paperNet.yaml',
  ↪'utf8'));
  ```

  Read more about connection profiles and how to configure them.

- **Connection options**: `connectionOptions` allow an application to declare rather than implement desired transaction processing behaviour. Connection options are interpreted by the SDK to control interaction patterns with network components, for example to select which identity to connect with, or which peers to use for event notifications. These options significantly reduce application complexity without compromising functionality. This is possible because the SDK has implemented much of the low level logic that would otherwise be required by applications; connection options control this logic flow.

  Read about the list of available connection options and when to use them.

Static

Static gateways define a fixed view of a network. In the MagnetoCorp *scenario*, a gateway might identify a single peer from MagnetoCorp, a single peer from DigiBank, and a MagentoCorp orderer. Alternatively, a gateway might define *all* peers and orderers from MagnetCorp and DigiBank. In both cases, a gateway must define a view of the network sufficient to get commercial paper transactions endorsed and distributed.

Applications can use a gateway statically by explicitly specifying the connect option `discovery: { enabled:false }` on the `gateway.connect()` API. Alternatively, the environment variable setting `FABRIC_SDK_DISCOVERY=false` will always override the application choice.

Examine the connection profile used by the MagnetoCorp issue application. See how all the peers, orderers and even CAs are specified in this file, including their roles.

It's worth bearing in mind that a static gateway represents a view of a network at a *moment in time*. As networks change, it may be important to reflect this in a change to the gateway file. Applications will automatically pick up these changes when they re-load the gateway file.

Dynamic

Dynamic gateways define a small, fixed *starting point* for a network. In the MagnetoCorp *scenario*, a dynamic gateway might identify just a single peer from MagnetoCorp; everything else will be discovered! (To provide resiliency, it might be better to define two such bootstrap peers.)

If service discovery is selected by an application, the topology defined in the gateway file is augmented with that produced by this process. Service discovery starts with the gateway definition, and finds all the connected peers and orderers within the MagnetoCorp organization using the gossip protocol. If anchor peers have been defined for a channel, then service discovery will use the gossip protocol across organizations to discover components within the connected organization. This process will also discover smart contracts installed on peers and their endorsement policies defined at a channel level. As with static gateways, the discovered network must be sufficient to get commercial paper transactions endorsed and distributed.

Dynamic gateways are the default setting for Fabric applications. They can be explicitly specified using the connect option `discovery: { enabled:true }` on the `gateway.connect()` API. Alternatively, the environment variable setting `FABRIC_SDK_DISCOVERY=true` will always override the application choice.

A dynamic gateway represents an up-to-date view of a network. As networks change, service discovery will ensure that the network view is an accurate reflection of the topology visible to the application. Applications will automatically pick up these changes; they do not even need to re-load the gateway file.

Multiple gateways

Finally, it is straightforward for an application to define multiple gateways, both for the same or different networks. Moreover, applications can use the name gateway both statically and dynamically.

It can be helpful to have multiple gateways. Here are a few reasons:

- Handling requests on behalf of different users.
- Connecting to different networks simultaneously.
- Testing a network configuration, by simultaneously comparing its behaviour with an existing configuration.

This topic covers how to develop a client application and smart contract to solve a business problem using Hyperledger Fabric. In a real world **Commercial Paper** scenario, involving multiple organizations, you'll learn about all the concepts and tasks required to accomplish this goal. We assume that the blockchain network is already available.

The topic is designed for multiple audiences:

- Solution and application architect
- Client application developer
- Smart contract developer
- Business professional

You can choose to read the topic in order, or you can select individual sections as appropriate. Individual topic sections are marked according to reader relevance, so whether you're looking for business or technical information it'll be clear when a topic is for you.

The topic follows a typical software development lifecycle. It starts with business requirements, and then covers all the major technical activities required to develop an application and smart contract to meet these requirements.

If you'd prefer, you can try out the commercial paper scenario immediately, following an abbreviated explanation, by running the commercial paper tutorial. You can return to this topic when you need fuller explanations of the concepts introduced in the tutorial.

CHAPTER 7

Tutorials

Application developers can use the Fabric tutorials to get started building their own solutions. Start working with Fabric by deploying the test network on your local machine. You can then use the steps provided by the *Deploying a smart contract to a channel* tutorial to deploy and test your smart contracts. The *Writing Your First Application* tutorial provides an introduction to how to use the APIs provided by the Fabric SDKs to invoke smart contracts from your client applications. For an in depth overview of how Fabric applications and smart contracts work together, you can visit the *Developing Applications* topic.

The *Deploying a smart contract to a channel* tutorial can also be used by network operators to learn how to use the Fabric chaincode lifecycle to manage smart contracts deployed on a running network. Both network operators and application developers can use the tutorials on Private data and CouchDB to explore important Fabric features. When you are ready to deploy Hyperledger Fabric in production, see the guide for *Deploying a production network*.

There are two tutorials for updating a channel: *Updating a channel configuration* and *Updating the capability level of a channel*, while *Upgrading your components* shows how to upgrade components like peers, ordering nodes, SDKs, and more.

Finally, we provide an introduction to how to write a basic smart contract, *Chaincode for Developers*.

Note: If you have questions not addressed by this documentation, or run into issues with any of the tutorials, please visit the *Still Have Questions?* page for some tips on where to find additional help.

7.1 Deploying a smart contract to a channel

End users interact with the blockchain ledger by invoking smart contracts. In Hyperledger Fabric, smart contracts are deployed in packages referred to as chaincode. Organizations that want to validate transactions or query the ledger need to install a chaincode on their peers. After a chaincode has been installed on the peers joined to a channel, channel members can deploy the chaincode to the channel and use the smart contracts in the chaincode to create or update assets on the channel ledger.

A chaincode is deployed to a channel using a process known as the Fabric chaincode lifecycle. The Fabric chaincode lifecycle allows multiple organizations to agree how a chaincode will be operated before it can be used to create

transactions. For example, while an endorsement policy specifies which organizations need to execute a chaincode to validate a transaction, channel members need to use the Fabric chaincode lifecycle to agree on the chaincode endorsement policy. For a more in depth overview about how to deploy and manage a chaincode on a channel, see Fabric chaincode lifecyle.

You can use this tutorial to learn how to use the peer lifecycle chaincode commands to deploy a chaincode to a channel of the Fabric test network. Once you have an understanding of the commands, you can use the steps in this tutorial to deploy your own chaincode to the test network, or to deploy chaincode to a production network. In this tutorial, you will deploy the Fabcar chaincode that is used by the Writing your first application tutorial.

Note: These instructions use the Fabric chaincode lifecycle introduced in the v2.0 release. If you would like to use the previous lifecycle to install and instantiate a chaincode, visit the v1.4 version of the Fabric documentation.

7.1.1 Start the network

We will start by deploying an instance of the Fabric test network. Before you begin, make sure that that you have installed the Prerequisites and Installed the Samples, Binaries and Docker Images. Use the following command to navigate to the test network directory within your local clone of the `fabric-samples` repository:

```
cd fabric-samples/test-network
```

For the sake of this tutorial, we want to operate from a known initial state. The following command will kill any active or stale docker containers and remove previously generated artifacts.

```
./network.sh down
```

You can then use the following command to start the test network:

```
./network.sh up createChannel
```

The `createChannel` command creates a channel named `mychannel` with two channel members, Org1 and Org2. The command also joins a peer that belongs to each organization to the channel. If the network and the channel are created successfully, you can see the following message printed in the logs:

```
========= Channel successfully joined ===========
```

We can now use the Peer CLI to deploy the Fabcar chaincode to the channel using the following steps:

- *Step one: Package the smart contract*
- *Step two: Install the chaincode package*
- *Step three: Approve a chaincode definition*
- *Step four: Committing the chaincode definition to the channel*

7.1.2 Setup Logspout (optional)

This step is not required, but is extremely useful for troubleshooting chaincode. To monitor the logs of the smart contract, an administrator can view the aggregated output from a set of Docker containers using the `logspout` tool. The tool collects the output streams from different Docker containers into one place, making it easy to see what's happening from a single window. This can help administrators debug problems when they install smart contracts or developers when they invoke smart contracts. Because some containers are created purely for the purposes of starting a smart contract and only exist for a short time, it is helpful to collect all of the logs from your network.

A script to install and configure Logspout, `monitordocker.sh`, is already included in the `commercial-paper` sample in the Fabric samples. We will use the same script in this tutorial as well. The Logspout tool will continuously

stream logs to your terminal, so you will need to use a new terminal window. Open a new terminal and navigate to the `test-network` directory.

```
cd fabric-samples/test-network
```

You can run the `monitordocker.sh` script from any directory. For ease of use, we will copy the `monitordocker.sh` script from the `commercial-paper` sample to your working directory

```
cp ../commercial-paper/organization/digibank/configuration/cli/monitordocker.sh .
# if you're not sure where it is
find . -name monitordocker.sh
```

You can then start Logspout by running the following command:

```
./monitordocker.sh net_test
```

You should see output similar to the following:

```
Starting monitoring on all containers on the network net_basic
Unable to find image 'gliderlabs/logspout:latest' locally
latest: Pulling from gliderlabs/logspout
4fe2ade4980c: Pull complete
decca452f519: Pull complete
ad60f6b6c009: Pull complete
Digest: sha256:374e06b17b004bddc5445525796b5f7adb8234d64c5c5d663095fccafb6e4c26
Status: Downloaded newer image for gliderlabs/logspout:latest
1f99d130f15cf01706eda3e1f040496ec885036d485cb6bcc0da4a567ad84361
```

You will not see any logs at first, but this will change when we deploy our chaincode. It can be helpful to make this terminal window wide and the font small.

7.1.3 Package the smart contract

We need to package the chaincode before it can be installed on our peers. The steps are different if you want to install a smart contract written in *Go*, *Java*, or *JavaScript*.

Go

Before we package the chaincode, we need to install the chaincode dependences. Navigate to the folder that contains the Go version of the Fabcar chaincode.

```
cd fabric-samples/chaincode/fabcar/go
```

The sample uses a Go module to install the chaincode dependencies. The dependencies are listed in a `go.mod` file in the `fabcar/go` directory. You should take a moment to examine this file.

```
$ cat go.mod
module github.com/hyperledger/fabric-samples/chaincode/fabcar/go

go 1.13

require github.com/hyperledger/fabric-contract-api-go v0.0.0-20191118113407-
↪4c6ff12b4f96
```

The `go.mod` file imports the Fabric contract API into the smart contract package. You can open `fabcar.go` in a text editor to see how the contract API is used to define the `SmartContract` type at the beginning of the smart contract:

```
// SmartContract provides functions for managing a car
type SmartContract struct {
    contractapi.Contract
}
```

The `SmartContract` type is then used to create the transaction context for the functions defined within the smart contract that read and write data to the blockchain ledger.

```
// CreateCar adds a new car to the world state with given details
func (s *SmartContract) CreateCar(ctx contractapi.TransactionContextInterface,␣
↪carNumber string, make string, model string, colour string, owner string) error {
    car := Car{
        Make:   make,
        Model:  model,
        Colour: colour,
        Owner:  owner,
    }

    carAsBytes, _ := json.Marshal(car)

    return ctx.GetStub().PutState(carNumber, carAsBytes)
}
```

You can learn more about the Go contract API by visiting the API documentation and the smart contract processing topic.

To install the smart contract dependencies, run the following command from the `fabcar/go` directory.

```
GO111MODULE=on go mod vendor
```

If the command is successful, the go packages will be installed inside a `vendor` folder.

Now that we that have our dependences, we can create the chaincode package. Navigate back to our working directory in the `test-network` folder so that we can package the chaincode together with our other network artifacts.

```
cd ../../../test-network
```

You can use the `peer` CLI to create a chaincode package in the required format. The `peer` binaries are located in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

```
export PATH=${PWD}/../bin:${PWD}:$PATH
```

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

To confirm that you are able to use the `peer` CLI, check the version of the binaries. The binaries need to be version `2.0.0` or later to run this tutorial.

```
peer version
```

You also need to set the `CORE_PEER_MSPCONFIGPATH` to the location of the MSP folder of an admin or client user. Run the following command to establish the Org1 admin as the identity we're using.

```
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
```

You can now create the chaincode package using the peer lifecycle chaincode package command:

```
peer lifecycle chaincode package fabcar.tar.gz --path ../chaincode/fabcar/go/ --lang␣
↪golang --label fabcar_1
```

This command will create a package named `fabcar.tar.gz` in your current directory. The `--lang` flag is used to specify the chaincode language and the `--path` flag provides the location of your smart contract code. The `--label` flag is used to specify a chaincode label that will identity your chaincode after it is installed. It is recommended that your label include the chaincode name and version.

Now that we created the chaincode package, we can *install the chaincode* on the peers of the test network.

JavaScript

Before we package the chaincode, we need to install the chaincode dependences. Navigate to the folder that contains the JavaScript version of the Fabcar chaincode.

```
cd fabric-samples/chaincode/fabcar/javascript
```

The dependencies are listed in the `package.json` file in the `fabcar/javascript` directory. You should take a moment to examine this file. You can find the dependences section displayed below:

```
"dependencies": {
        "fabric-contract-api": "^2.0.0",
        "fabric-shim": "^2.0.0"
```

The `package.json` file imports the Fabric contract class into the smart contract package. You can open `lib/fabcar.js` in a text editor to see the contract class imported into the smart contract and used to create the FabCar class.

```
const { Contract } = require('fabric-contract-api');

class FabCar extends Contract {
    ...
}
```

The `FabCar` class provides the transaction context for the functions defined within the smart contract that read and write data to the blockchain ledger.

```
async createCar(ctx, carNumber, make, model, color, owner) {
    console.info('============= START : Create Car ===========');

    const car = {
        color,
        docType: 'car',
        make,
        model,
        owner,
    };

    await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
    console.info('============= END : Create Car ===========');
}
```

You can learn more about the JavaScript contract API by visiting the API documentation and the smart contract processing topic.

To install the smart contract dependencies, run the following command from the `fabcar/javascript` directory.

```
npm install
```

If the command is successful, the JavaScript packages will be installed inside a `npm_modules` folder.

Now that we that have our dependences, we can create the chaincode package. Navigate back to our working directory in the `test-network` folder so that we can package the chaincode together with our other network artifacts.

```
cd ../../../test-network
```

You can use the `peer` CLI to create a chaincode package in the required format. The `peer` binaries are located in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

```
export PATH=${PWD}/../bin:${PWD}:$PATH
```

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

To confirm that you are able to use the `peer` CLI, check the version of the binaries. The binaries need to be version `2.0.0` or later to run this tutorial.

```
peer version
```

You also need to set the `CORE_PEER_MSPCONFIGPATH` to the location of the MSP folder of an admin or client user. Run the following command to establish the Org1 admin as the identity we're using.

```
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
```

You can now create the chaincode package using the peer lifecycle chaincode package command:

```
peer lifecycle chaincode package fabcar.tar.gz --path ../chaincode/fabcar/javascript/␣
↪--lang node --label fabcar_1
```

This command will create a package named `fabcar.tar.gz` in your current directory. The `--lang` flag is used to specify the chaincode language and the `--path` flag provides the location of your smart contract code. The `--label` flag is used to specify a chaincode label that will identity your chaincode after it is installed. It is recommended that your label include the chaincode name and version.

Now that we created the chaincode package, we can *install the chaincode* on the peers of the test network.

Java

Before we package the chaincode, we need to install the chaincode dependences. Navigate to the folder that contains the Java version of the Fabcar chaincode.

```
cd fabric-samples/chaincode/fabcar/java
```

The sample uses Gradle to install the chaincode dependencies. The dependencies are listed in the `build.gradle` file in the `fabcar/java` directory. You should take a moment to examine this file. You can find the dependences section displayed below:

```
dependencies {
    compileOnly 'org.hyperledger.fabric-chaincode-java:fabric-chaincode-shim:2.0.+'
    implementation 'com.owlike:genson:1.5'
    testImplementation 'org.hyperledger.fabric-chaincode-java:fabric-chaincode-shim:2.
↪0.+'
    testImplementation 'org.junit.jupiter:junit-jupiter:5.4.2'
    testImplementation 'org.assertj:assertj-core:3.11.1'
    testImplementation 'org.mockito:mockito-core:2.+'
}
```

The `build.gradle` file imports the Java chaincode shim into the smart contract package, which includes the contract class. You can find Fabcar smart contract in the `src` directory. You can navigate to the `FabCar.java` file and open it in a text editor to see how the contract class is used to create the transaction context for the functions defined that read and write data to the blockchain ledger.

You can learn more about the Java chaincode shim and the contract class by visiting the Java chaincode documentation and the smart contract processing topic.

To install the smart contract dependencies, run the following command from the `fabcar/java` directory.

```
./gradlew installDist
```

If the command is successful, you will be able to find the built smart contract in the `build` folder.

Now that we have installed the dependences and built the smart contract, we can create the chaincode package. Navigate back to our working directory in the `test-network` folder so that we can package the chaincode together with our other network artifacts.

```
cd ../../../test-network
```

You can use the `peer` CLI to create a chaincode package in the required format. The `peer` binaries are located in the `bin` folder of the `fabric-samples` repository. Use the following command to add those binaries to your CLI Path:

```
export PATH=${PWD}/../bin:${PWD}:$PATH
```

You also need to set the `FABRIC_CFG_PATH` to point to the `core.yaml` file in the `fabric-samples` repository:

```
export FABRIC_CFG_PATH=$PWD/../config/
```

To confirm that you are able to use the `peer` CLI, check the version of the binaries. The binaries need to be version `2.0.0` or later to run this tutorial.

```
peer version
```

You also need to set the `CORE_PEER_MSPCONFIGPATH` to the location of the MSP folder of an admin or client user. Run the following command to establish the Org1 admin as the identity we're using.

```
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
```

You can now create the chaincode package using the peer lifecycle chaincode package command:

```
peer lifecycle chaincode package fabcar.tar.gz --path ../chaincode/fabcar/java/build/
↪install/fabcar --lang java --label fabcar_1
```

This command will create a package named `fabcar.tar.gz` in your current directory. The `--lang` flag is used to specify the chaincode language and the `--path` flag provides the location of your smart contract code. The `--label`

flag is used to specify a chaincode label that will identity your chaincode after it is installed. It is recommended that your label include the chaincode name and version.

Now that we created the chaincode package, we can *install the chaincode* on the peers of the test network.

7.1.4 Install the chaincode package

After we package the Fabcar smart contract, we can install the chaincode on our peers. The chaincode needs to be installed on every peer that will endorse a transaction. Because we are going to set the endorsement policy to require endorsements from both Org1 and Org2, we need to install the chaincode on the peers operated by both organizations:

- peer0.org1.example.com
- peer0.org2.example.com

Let's install the chaincode on the Org1 peer first. Set the following environment variables to operate the `peer` CLI as the Org1 admin user. The `CORE_PEER_ADDRESS` will be set to point to the Org1 peer, `peer0.org1.example.com`.

```
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

Issue the peer lifecycle chaincode install command to install the chaincode on the peer:

```
peer lifecycle chaincode install fabcar.tar.gz
```

If the command is successful, the peer will generate and return the package identifier. This package ID will be used to approve the chaincode in the next step. You should see output similar to the following:

```
2020-02-12 11:40:02.923 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪001 Installed remotely: response:<status:200 payload:"\nIfabcar_
↪1:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3\022\010fabcar_1"␣
↪>
2020-02-12 11:40:02.925 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪002 Chaincode code package identifier: fabcar_
↪1:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3
```

We can now install the chaincode on the Org2 peer. Set the following environment variables to operate as the Org2 admin and target target the Org2 peer, `peer0.org2.example.com`.

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

Issue the following command to install the chaincode:

```
peer lifecycle chaincode install fabcar.tar.gz
```

The chaincode is built by the peer when the chaicode is installed. The install command will return any build errors from the chaincode if there is a problem with the smart contract code.

7.1.5 Approve a chaincode definition

After you install the chaincode package, you need to approve a chaincode definition for your organization. The definition includes the important parameters of chaincode governance such as the name, version, and the chaincode endorsement policy.

The set of channel members who need to approve a chaincode before it can be deployed is governed by the `Application/Channel/lifeycleEndorsement` policy. By default, this policy requires that a majority of channel members need to approve a chaincode before it can used on a channel. Because we have only two organizations on the channel, and a majority of 2 is 2, we need approve a chaincode definition of Fabcar as Org1 and Org2.

If an organization has installed the chaincode on their peer, they need to include the packageID in the chaincode definition approved by their organization. The package ID is used to associate the chaincode installed on a peer with an approved chaincode definition, and allows an organization to use the chaincode to endorse transactions. You can find the package ID of a chaincode by using the peer lifecycle chaincode queryinstalled command to query your peer.

```
peer lifecycle chaincode queryinstalled
```

The package ID is the combination of the chaincode label and a hash of the chaincode binaries. Every peer will generate the same package ID. You should see output similar to the following:

```
Installed chaincodes on peer:
Package ID: fabcar_1:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3,
↪ Label: fabcar_1
```

We are going to use the package ID when we approve the chaincode, so let's go ahead and save it as an environment variable. Paste the package ID returned by `peer lifecycle chaincode queryinstalled` into the command below. The package ID may not be the same for all users, so you need to complete this step using the package ID returned from your command window.

```
CC_PACKAGE_ID=fabcar_
↪1:69de748301770f6ef64b42aa6bb6cb291df20aa39542c3ef94008615704007f3
```

Because the environment variables have been set to operate the `peer` CLI as the Org2 admin, we can approve the chaincode definition of Fabcar as Org2. Chaincode is approved at the organization level, so the command only needs to target one peer. The approval is distributed to the other peers within the organization using gossip. Approve the chaincode definition using the peer lifecycle chaincode approveformyorg command:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name fabcar -
↪-version 1.0 --init-required --package-id $CC_PACKAGE_ID --sequence 1 --tls true --
↪cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.
↪example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

The command above uses the `--package-id` flag to include the package identifier in the chaincode definition. The `--sequence` parameter is an integer that keeps track of the number of times a chaincode has been defined or updated. Because the chaincode is being deployed to the channel for the first time, the sequence number is 1. When the Fabcar chaincode is upgraded, the sequence number will be incremented to 2. The command also uses the `--init-required` flag to request that Init function be invoked to initialize the chaincode, before other functions can be used to interact with the ledger. By default, the Init function does not need to be executed. You do not need to request that the Init function be invoked to initialize your chaincode. Because the Fabcar chaincode uses the Fabric

contract API, we can initialize the chaincode by calling any function in the chaincode and the Init function will be called in the background.

We could have provided a `--signature-policy` or `--channel-config-policy` argument to the `approveformyorg` command to specify a chaincode endorsement policy. The endorsement policy specifies how many peers belonging to different channel members need to validate a transaction against a given chaincode. Because we did not set a policy, the definition of Fabcar will use the default endorsement policy, which requires that a transaction be endorsed by a majority of channel members present when the transaction is submitted. This implies that if new organizations are added or removed from the channel, the endorsement policy is updated automatically to require more or fewer endorsements. In this tutorial, the default policy will require a majority of 2 out of 2 and transactions will need to be endorsed by a peer from Org1 and Org2. If you want to specify a custom endorsement policy, you can use the Endorsement Policies operations guide to learn about the policy syntax.

You need to approve a chaincode definition with an identity that has an admin role. As a result, the `CORE_PEER_MSPCONFIGPATH` variable needs to point to the MSP folder that contains an admin identity. You cannot approve a chaincode definition with a client user. The approval needs to be submitted to the ordering service, which will validate the admin signature and then distribute the approval to your peers.

We still need to approve the chaincode definition as Org1. Set the following environment variables to operate as the Org1 admin:

```
CORE_PEER_LOCALMSPID="Org1MSP"
CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.com/users/
↪Admin@org1.example.com/msp
CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.example.com/
↪peers/peer0.org1.example.com/tls/ca.crt
CORE_PEER_ADDRESS=localhost:7051
```

You can now approve the chaincode definition as Org1.

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name fabcar -
↪-version 1.0 --init-required --package-id $CC_PACKAGE_ID --sequence 1 --tls true --
↪cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.
↪example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

We now have the majority we need to deploy the Fabcar the chaincode to the channel. While only a majority of organizations need to approve a chaincode definition (with the default policies), all organizations need to approve a chaincode definition to start the chaincode on their peers. If you commit the definition before a channel member has approved the chaincode, the organization will not be able to endorse transactions. As a result, it is recommended that all channel members approve a chaincode before committing the chaincode definition.

7.1.6 Committing the chaincode definition to the channel

After a sufficient number of organizations have approved a chaincode definition, one organization can commit the chaincode definition to the channel. If a majority of channel members have approved the definition, the commit transaction will be successful and the parameters agreed to in the chaincode definition will be implemented on the channel.

You can use the peer lifecycle chaincode checkcommitreadiness command to check whether channel members have approved the same chaincode definition. The flags used for the `checkcommitreadiness` command are identical to the flags used to approve a chaincode for your organization. However, you do not need to include the `--package-id` flag.

```
peer lifecycle chaincode checkcommitreadiness --channelID mychannel --name fabcar --
↪version 1.0 --init-required --sequence 1 --tls true --cafile ${PWD}/organizations/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪example.com-cert.pem --output json
```

(continues on next page)

(continued from previous page)

The command will produce a JSON map that displays if a channel member has approved the parameters that were specified in the `checkcommitreadiness` command:

```
    {
            "Approvals": {
                    "Org1MSP": true,
                    "Org2MSP": true
            }
    }
```

Since both organizations that are members of the channel have approved the same parameters, the chaincode definition is ready to be committed to the channel. You can use the peer lifecycle chaincode commit command to commit the chaincode definition to the channel. The commit command also needs to be submitted by an organization admin.

```
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride␣
↪orderer.example.com --channelID mychannel --name fabcar --version 1.0 --sequence 1 -
↪-init-required --tls true --cafile ${PWD}/organizations/ordererOrganizations/
↪example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -
↪-peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

The transaction above uses the `--peerAddresses` flag to target `peer0.org1.example.com` from Org1 and `peer0.org2.example.com` from Org2. The `commit` transaction is submitted to the peers joined to the channel to query the chaincode definition that was approved by the organization that operates the peer. The command needs to target the peers from a sufficient number of organizations to satisfy the policy for deploying a chaincode. Because the approval is distributed within each organization, you can target any peer that belongs to a channel member.

The chaincode definition endorsements by channel members are submitted to the ordering service to be added to a block and distributed to the channel. The peers on the channel then validate whether a sufficient number of organizations have approved the chaincode definition. The `peer lifecycle chaincode commit` command will wait for the validations from the peer before returning a response.

You can use the peer lifecycle chaincode querycommitted command to confirm that the chaincode definition has been committed to the channel.

```
peer lifecycle chaincode querycommitted --channelID mychannel --name fabcar --cafile $
↪{PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/
↪msp/tlscacerts/tlsca.example.com-cert.pem
```

If the chaincode was successful committed to the channel, the `querycommitted` command will return the sequence and version of the chaincode definition:

```
Committed chaincode definition for chaincode 'fabcar' on channel 'mychannel':
Version: 1, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc,␣
↪Approvals: [Org1MSP: true, Org2MSP: true]
```

7.1.7 Invoking the chaincode

After the chaincode definition has been committed to a channel, the chaincode will start on the peers joined to the channel where the chaincode was installed. The Fabcar chaincode is now ready to be invoked by client applications. Use the following command create an initial set of cars on the ledger. Note that the invoke command needs target a sufficient number of peers to meet chaincode endorsement policy.

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls true --cafile ${PWD}/organizations/ordererOrganizations/example.com/
↪orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel␣
↪-n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt --isInit␣
↪-c '{"function":"InitLedger","Args":[]}'
```

If the command is successful, you should be able to a response similar to the following:

```
2020-02-12 18:22:20.576 EST [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001␣
↪Chaincode invoke successful. result: status:200
```

Because we included the `--init-required` flag in the chaincode definition, the first transaction needs to initialize the chaincode by passing the `--isInit` flag to the invoke command. Because the Fabcar smart contract uses Fabric contract API, the first transaction can target any function in the chaincode. If we were using the low level APIs provided by the Fabric Chaincode Shim API, the transaction would need to target the Init function. The first invoke of the chaincode, whether of Init or another function, is subject to the chaincode endorsement policy.

We can use a query function to read the set of cars that were created by the chaincode:

```
peer chaincode query -C mychannel -n fabcar -c '{"Args":["queryAllCars"]}'
```

The response to the query should be the following list of cars:

```
[{"Key":"CAR0","Record":{"make":"Toyota","model":"Prius","colour":"blue","owner":
↪"Tomoko"}},
{"Key":"CAR1","Record":{"make":"Ford","model":"Mustang","colour":"red","owner":"Brad"}
↪},
{"Key":"CAR2","Record":{"make":"Hyundai","model":"Tucson","colour":"green","owner":
↪"Jin Soo"}},
{"Key":"CAR3","Record":{"make":"Volkswagen","model":"Passat","colour":"yellow","owner
↪":"Max"}},
{"Key":"CAR4","Record":{"make":"Tesla","model":"S","colour":"black","owner":"Adriana"}
↪},
{"Key":"CAR5","Record":{"make":"Peugeot","model":"205","colour":"purple","owner":
↪"Michel"}},
{"Key":"CAR6","Record":{"make":"Chery","model":"S22L","colour":"white","owner":"Aarav
↪"}},
{"Key":"CAR7","Record":{"make":"Fiat","model":"Punto","colour":"violet","owner":"Pari
↪"}},
{"Key":"CAR8","Record":{"make":"Tata","model":"Nano","colour":"indigo","owner":
↪"Valeria"}},
{"Key":"CAR9","Record":{"make":"Holden","model":"Barina","colour":"brown","owner":
↪"Shotaro"}}]
```

7.1.8 Clean up

When you are finished using the chaincode, you can also use the following commands to remove the Logspout tool.

```
docker stop logspout
docker rm logspout
```

You can then bring down the test network by issuing the following command from the `test-network` directory:

```
./network.sh down
```

7.1.9 Next steps

After you write your smart contract and deploy it to a channel, you can use the APIs provided by the Fabric SDKs to invoke the smart contracts from a client application. This allows end users to interact with the assets on the blockchain ledger. To get started with the Fabric SDKs, see the Writing Your first application tutorial.

7.2 Writing Your First Application

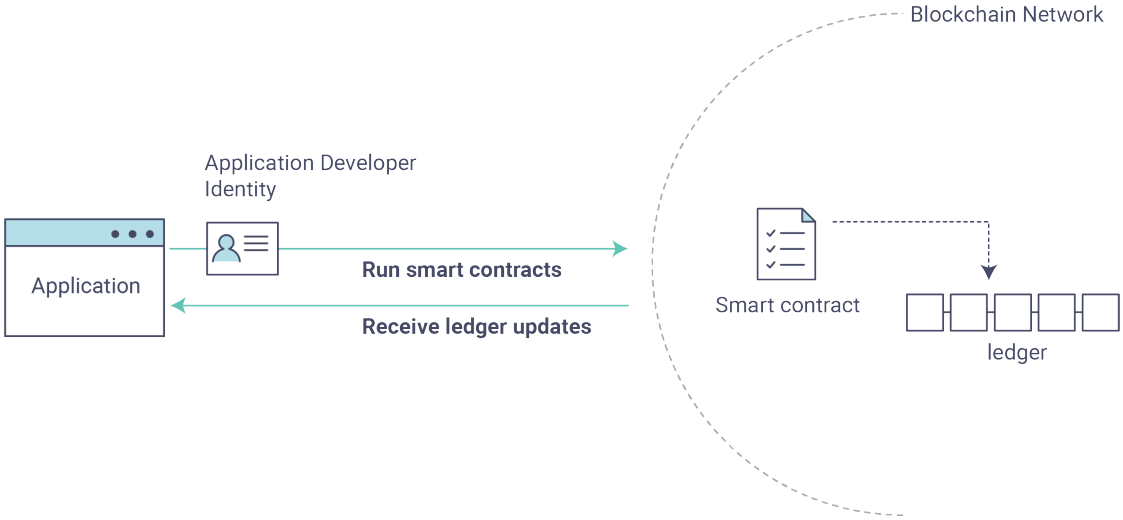
Note: If you're not yet familiar with the fundamental architecture of a Fabric network, you may want to visit the *Key Concepts* section prior to continuing.

It is also worth noting that this tutorial serves as an introduction to Fabric applications and uses simple smart contracts and applications. For a more in-depth look at Fabric applications and smart contracts, check out our *Developing Applications* section or the *Commercial paper tutorial*.

This tutorial provides an introduction to how Fabric applications interact with deployed blockchain networks. The tutorial uses sample programs built using the Fabric SDKs – described in detail in the *Application* topic – to invoke a smart contract which queries and updates the ledger with the smart contract API – described in detail in *Smart Contract Processing*. We will also use our sample programs and a deployed Certificate Authority to generate the X.509 certificates that an application needs to interact with a permissioned blockchain. The sample applications and the smart contract they invoke are collectively known as FabCar.

We'll go through three principle steps:

1. Setting up a development environment. Our application needs a network to interact with, so we'll deploy a basic network for our smart contracts and application.



2. Explore a sample smart contract. We'll inspect the sample Fabcar smart contract to learn about the transactions within them, and how they are used by applications to query and update the ledger.

3. Interact with the smart contract with a sample application. Our application will use the FabCar smart contract to query and update car assets on the ledger. We'll get into the code of the apps and the transactions they create, including querying a car, querying a range of cars, and creating a new car.

After completing this tutorial you should have a basic understanding of how Fabric applications and smart contracts work together to manage data on the distributed ledger of a blockchain network.

7.2.1 Set up the blockchain network

If you've already run through *Using the Fabric test network* tutorial and have a network up and running, this tutorial will bring down your running network network before bringing up a new one.

If you are using Mac OS and running Mojave, you will need to install Xcode.

Launch the network

Note: This tutorial demonstrates the JavaScript versions of the FabCar smart contract and application, but the `fabric-samples` repo also contains Go, Java and TypeScript versions of this sample. To try the Go, Java or TypeScript versions, change the `javascript` argument for `./startFabric.sh` below to either `go`, `java` or `typescript` and follow the instructions written to the terminal.

Navigate to the `fabcar` subdirectory within your local clone of the `fabric-samples` repo.

```
cd fabric-samples/fabcar
```

Launch your network using the `startFabric.sh` shell script.

```
./startFabric.sh javascript
```

This command will deploy the Fabric test network with two peers and an ordering service. Instead of using the cryptogen tool, we will bring up the test network using Certificate Authorities. We will use one of these CAs to create the certificates and keys that will be used by our applications in a future step. The `startFabric.sh` script will also deploy and initialize the JavaScript version of the FabCar smart contract on the channel `mychannel`, and then invoke the smart contract to put initial data on the ledger.

Install the application

From the `fabcar` directory inside `fabric-samples`, navigate to the `javascript` folder.

```
cd javascript
```

This directory contains sample programs that were developed using the Fabric SDK for Node.js. Run the following command to install the application dependencies. It will take about a minute to complete:

```
npm install
```

This process is installing the key application dependencies defined in `package.json`. The most important of which is the `fabric-network` class; it enables an application to use identities, wallets, and gateways to connect to channels, submit transactions, and wait for notifications. This tutorial also uses the `fabric-ca-client` class to enroll users with their respective certificate authorities, generating a valid identity which is then used by `fabric-network` class methods.

Once `npm install` completes, everything is in place to run the application. Let's take a look at the sample JavaScript application files we will be using in this tutorial:

```
ls
```

You should see the following:

```
enrollAdmin.js  node_modules       package.json  registerUser.js
invoke.js       package-lock.json  query.js      wallet
```

There are files for other program languages, for example in the `fabcar/java` directory. You can read these once you've used the JavaScript example – the principles are the same.

If you are using Mac OS and running Mojave, you will need to install Xcode.

7.2.2 Enrolling the admin user

Note: The following two sections involve communication with the Certificate Authority. You may find it useful to stream the CA logs when running the upcoming programs by opening a new terminal shell and running `docker logs -f ca_org1`.

When we created the network, an admin user — literally called `admin` — was created as the **registrar** for the certificate authority (CA). Our first step is to generate the private key, public key, and X.509 certificate for `admin` using the `enroll.js` program. This process uses a **Certificate Signing Request** (CSR) — the private and public key are first generated locally and the public key is then sent to the CA which returns an encoded certificate for use by the application. These credentials are then stored in the wallet, allowing us to act as an administrator for the CA.

Let's enroll user `admin`:

```
node enrollAdmin.js
```

This command stores the CA administrator's credentials in the `wallet` directory. You can find administrator's certificate and private key in the `wallet/admin.id` file.

7.2.3 Register and enroll an application user

Our `admin` is used to work with the CA. Now that we have the administrator's credentials in a wallet, we can create a new application user which will be used to interact with the blockchain. Run the following command to register and enroll a new user named `appUser`:

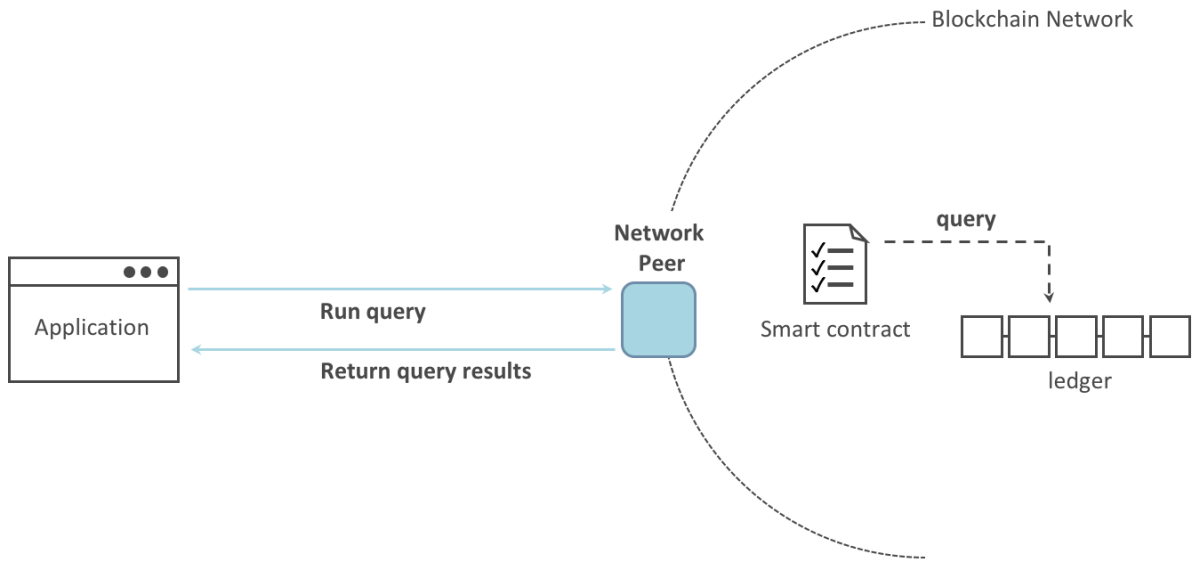
```
node registerUser.js
```

Similar to the admin enrollment, this program uses a CSR to enroll `appUser` and store its credentials alongside those of `admin` in the wallet. We now have identities for two separate users — `admin` and `appUser` — that can be used by our application.

7.2.4 Querying the ledger

Each peer in a blockchain network hosts a copy of the ledger. An application program can view the most recent data from the ledger using read only invocations of a smart contract running on your peers called a query.

Here is a simplified representation of how a query works:



The most common queries involve the current values of data in the ledger – its world state. The world state is represented as a set of key-value pairs, and applications can query data for a single key or multiple keys. Moreover, you can use complex queries to read the data on the ledger when you use CouchDB as your state database and model your data in JSON. This can be very helpful when looking for all assets that match certain keywords with particular values; all cars with a particular owner, for example.

First, let's run our `query.js` program to return a listing of all the cars on the ledger. This program uses our second identity – `appUser` – to access the ledger:

```
node query.js
```

The output should look like this:

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been evaluated, result is:
[{"Key":"CAR0","Record":{"color":"blue","docType":"car","make":"Toyota","model":"Prius
↪","owner":"Tomoko"}},
{"Key":"CAR1","Record":{"color":"red","docType":"car","make":"Ford","model":"Mustang",
↪"owner":"Brad"}},
{"Key":"CAR2","Record":{"color":"green","docType":"car","make":"Hyundai","model":
↪"Tucson","owner":"Jin Soo"}},
{"Key":"CAR3","Record":{"color":"yellow","docType":"car","make":"Volkswagen","model":
↪"Passat","owner":"Max"}},
{"Key":"CAR4","Record":{"color":"black","docType":"car","make":"Tesla","model":"S",
↪"owner":"Adriana"}},
{"Key":"CAR5","Record":{"color":"purple","docType":"car","make":"Peugeot","model":"205
↪","owner":"Michel"}},
{"Key":"CAR6","Record":{"color":"white","docType":"car","make":"Chery","model":"S22L",
↪"owner":"Aarav"}},
{"Key":"CAR7","Record":{"color":"violet","docType":"car","make":"Fiat","model":"Punto
↪","owner":"Pari"}},
{"Key":"CAR8","Record":{"color":"indigo","docType":"car","make":"Tata","model":"Nano",
↪"owner":"Valeria"}},
{"Key":"CAR9","Record":{"color":"brown","docType":"car","make":"Holden","model":
↪"Barina","owner":"Shotaro"}}]
```

Let's take a closer look at how *query.js* program uses the APIs provided by the Fabric Node SDK to interact with our Fabric network. Use an editor (e.g. atom or visual studio) to open `query.js`.

The application starts by bringing in scope two key classes from the `fabric-network` module; `Wallets` and `Gateway`. These classes will be used to locate the `appUser` identity in the wallet, and use it to connect to the network:

```
const { Gateway, Wallets } = require('fabric-network');
```

First, the program uses the Wallet class to get our application user from our file system.

```
const identity = await wallet.get('appUser');
```

Once the program has an identity, it uses the Gateway class to connect to our network.

```
const gateway = new Gateway();
await gateway.connect(ccpPath, { wallet, identity: 'appUser', discovery: { enabled:␣
↪true, asLocalhost: true } });
```

`ccpPath` describes the path to the connection profile that our application will use to connect to our network. The connection profile was loaded from inside the `fabric-samples/test network` directory and parsed as a JSON file:

```
const ccpPath = path.resolve(__dirname, '..', '..', 'test-network','organizations',
↪'peerOrganizations','org1.example.com', 'connection-org1.json');
```

If you'd like to understand more about the structure of a connection profile, and how it defines the network, check out the connection profile topic.

A network can be divided into multiple channels, and the next important line of code connects the application to a particular channel within the network, `mychannel`, where our smart contract was deployed:

```
const network = await gateway.getNetwork('mychannel');
```

Within this channel, we can access the FabCar smart contract to interact with the ledger:

```
const contract = network.getContract('fabcar');
```

Within FabCar there are many different **transactions**, and our application initially uses the `queryAllCars` transaction to access the ledger world state data:

```
const result = await contract.evaluateTransaction('queryAllCars');
```

The `evaluateTransaction` method represents one of the simplest interactions with a smart contract in blockchain network. It simply picks a peer defined in the connection profile and sends the request to it, where it is evaluated. The smart contract queries all the cars on the peer's copy of the ledger and returns the result to the application. This interaction does not result in an update the ledger.

7.2.5 The FabCar smart contract

Let's take a look at the transactions within the FabCar smart contract. Open a new terminal and navigate to the JavaScript version of the FabCar Smart contract inside the `fabric-samples` repository:

```
cd fabric-samples/chaincode/fabcar/javascript/lib
```

Open the `fabcar.js` file in a text editor editor.

See how our smart contract is defined using the `Contract` class:

```
class FabCar extends Contract {...
```

Within this class structure, you'll see that we have the following transactions defined: `initLedger`, `queryCar`, `queryAllCars`, `createCar`, and `changeCarOwner`. For example:

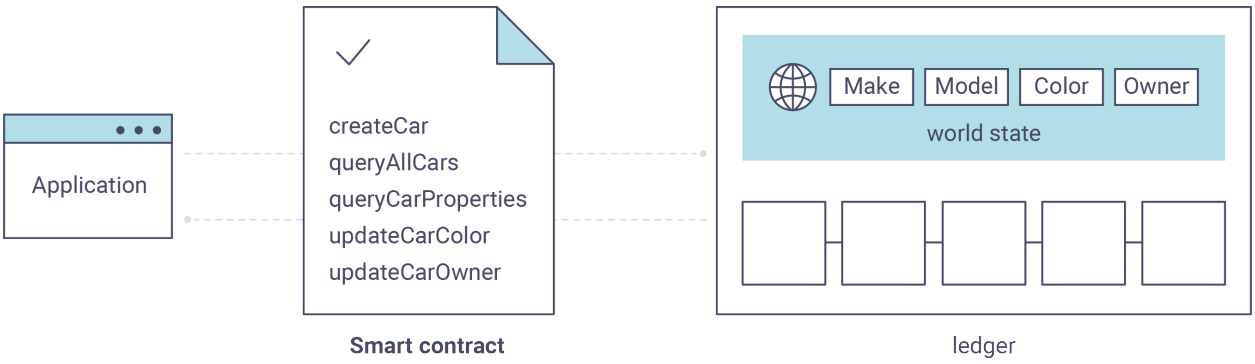
```
async queryCar(ctx, carNumber) {...}
async queryAllCars(ctx) {...}
```

Let's take a closer look at the `queryAllCars` transaction to see how it interacts with the ledger.

```
async queryAllCars(ctx) {

  const startKey = 'CAR0';
  const endKey = 'CAR999';

  const iterator = await ctx.stub.getStateByRange(startKey, endKey);
```

This code defines the range of cars that `queryAllCars` will retrieve from the ledger. Every car between `CAR0` and `CAR999` – 1,000 cars in all, assuming every key has been tagged properly – will be returned by the query. The remainder of the code iterates through the query results and packages them into JSON for the application.

Below is a representation of how an application would call different transactions in a smart contract. Each transaction uses a broad set of APIs such as `getStateByRange` to interact with the ledger. You can read more about these APIs in detail.



We can see our `queryAllCars` transaction, and another called `createCar`. We will use this later in the tutorial to update the ledger, and add a new block to the blockchain.

But first, go back to the `query` program and change the `evaluateTransaction` request to query `CAR4`. The `query` program should now look like this:

```
const result = await contract.evaluateTransaction('queryCar', 'CAR4');
```

Save the program and navigate back to your `fabcar/javascript` directory. Now run the `query` program again:

```
node query.js
```

You should see the following:

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been evaluated, result is:
{"color":"black","docType":"car","make":"Tesla","model":"S","owner":"Adriana"}
```

If you go back and look at the result from when the transaction was `queryAllCars`, you can see that `CAR4` was Adriana's black Tesla model S, which is the result that was returned here.

We can use the `queryCar` transaction to query against any car, using its key (e.g. `CAR0`) and get whatever make, model, color, and owner correspond to that car.

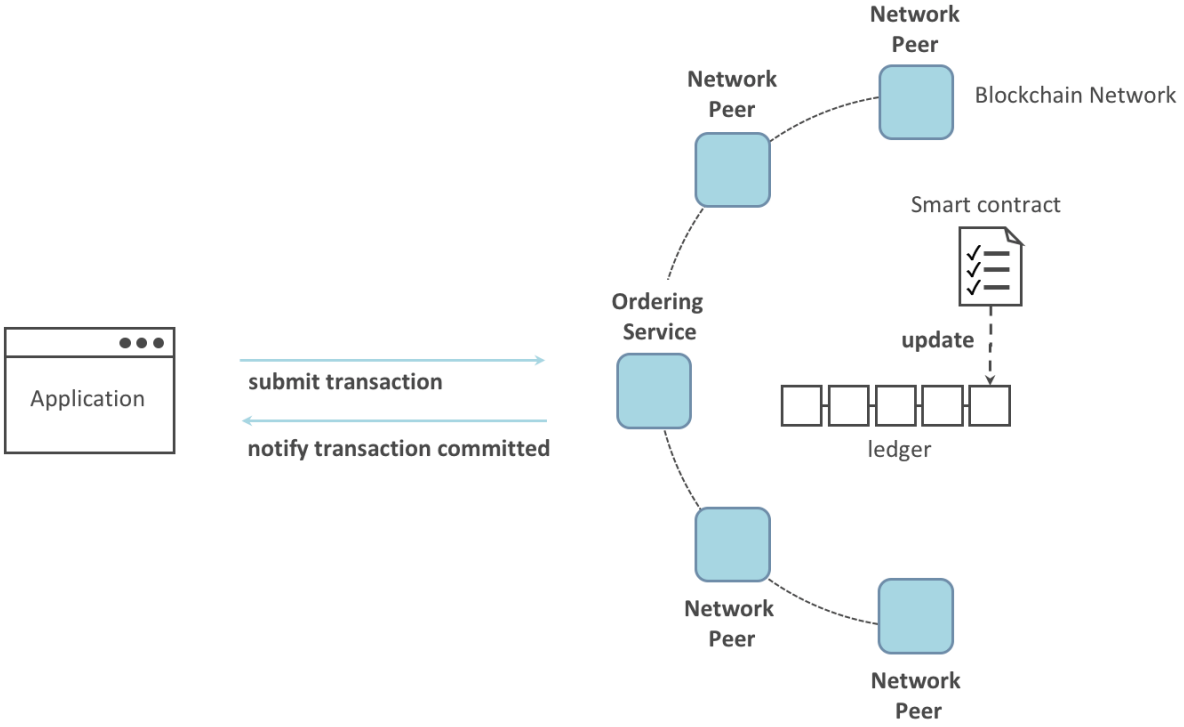
Great. At this point you should be comfortable with the basic query transactions in the smart contract and the handful of parameters in the query program.

Time to update the ledger...

7.2.6 Updating the ledger

Now that we've done a few ledger queries and added a bit of code, we're ready to update the ledger. There are a lot of potential updates we could make, but let's start by creating a **new** car.

From an application perspective, updating the ledger is simple. An application submits a transaction to the blockchain network, and when it has been validated and committed, the application receives a notification that the transaction has been successful. Under the covers this involves the process of **consensus** whereby the different components of the blockchain network work together to ensure that every proposed update to the ledger is valid and performed in an agreed and consistent order.



Above, you can see the major components that make this process work. As well as the multiple peers which each host a copy of the ledger, and optionally a copy of the smart contract, the network also contains an ordering service. The ordering service coordinates transactions for a network; it creates blocks containing transactions in a well-defined sequence originating from all the different applications connected to the network.

Our first update to the ledger will create a new car. We have a separate program called `invoke.js` that we will use to make updates to the ledger. Just as with queries, use an editor to open the program and navigate to the code block where we construct our transaction and submit it to the network:

```
await contract.submitTransaction('createCar', 'CAR12', 'Honda', 'Accord', 'Black',
↪'Tom');
```

See how the applications calls the smart contract transaction `createCar` to create a black Honda Accord with an owner named Tom. We use `CAR12` as the identifying key here, just to show that we don't need to use sequential keys.

Save it and run the program:

```
node invoke.js
```

If the invoke is successful, you will see output like this:

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been submitted
```

Notice how the `invoke` application interacted with the blockchain network using the `submitTransaction` API, rather than `evaluateTransaction`.

```
await contract.submitTransaction('createCar', 'CAR12', 'Honda', 'Accord', 'Black',
↪'Tom');
```

`submitTransaction` is much more sophisticated than `evaluateTransaction`. Rather than interacting with a single peer, the SDK will send the `submitTransaction` proposal to every required organization's peer in the blockchain network. Each of these peers will execute the requested smart contract using this proposal, to generate a transaction response which it signs and returns to the SDK. The SDK collects all the signed transaction responses into a single transaction, which it then sends to the orderer. The orderer collects and sequences transactions from every application into a block of transactions. It then distributes these blocks to every peer in the network, where every transaction is validated and committed. Finally, the SDK is notified, allowing it to return control to the application.

Note: `submitTransaction` also includes a listener that checks to make sure the transaction has been validated and committed to the ledger. Applications should either utilize a commit listener, or leverage an API like `submitTransaction` that does this for you. Without doing this, your transaction may not have been successfully ordered, validated, and committed to the ledger.

`submitTransaction` does all this for the application! The process by which the application, smart contract, peers and ordering service work together to keep the ledger consistent across the network is called consensus, and it is explained in detail in this section.

To see that this transaction has been written to the ledger, go back to `query.js` and change the argument from `CAR4` to `CAR12`.

In other words, change this:

```
const result = await contract.evaluateTransaction('queryCar', 'CAR4');
```

To this:

```
const result = await contract.evaluateTransaction('queryCar', 'CAR12');
```

Save once again, then query:

```
node query.js
```

Which should return this:

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been evaluated, result is:
{"color":"Black","docType":"car","make":"Honda","model":"Accord","owner":"Tom"}
```

Congratulations. You've created a car and verified that its recorded on the ledger!

So now that we've done that, let's say that Tom is feeling generous and he wants to give his Honda Accord to someone named Dave.

To do this, go back to `invoke.js` and change the smart contract transaction from `createCar` to `changeCarOwner` with a corresponding change in input arguments:

```
await contract.submitTransaction('changeCarOwner', 'CAR12', 'Dave');
```

The first argument — `CAR12` — identifies the car that will be changing owners. The second argument — `Dave` — defines the new owner of the car.

Save and execute the program again:

```
node invoke.js
```

Now let's query the ledger again and ensure that Dave is now associated with the `CAR12` key:

```
node query.js
```

It should return this result:

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been evaluated, result is:
{"color":"Black","docType":"car","make":"Honda","model":"Accord","owner":"Dave"}
```

The ownership of `CAR12` has been changed from Tom to Dave.

Note: In a real world application the smart contract would likely have some access control logic. For example, only certain authorized users may create new cars, and only the car owner may transfer the car to somebody else.

7.2.7 Clean up

When you are finished using the FabCar sample, you can bring down the test network using `networkDown.sh` script.

```
./networkDown.sh
```

This command will bring down the CAs, peers, and ordering node of the network that we created. It will also remove the `admin` and `appUser` crypto material stored in the `wallet` directory. Note that all of the data on the ledger will be lost. If you want to go through the tutorial again, you will start from a clean initial state.

7.2.8 Summary

Now that we've done a few queries and a few updates, you should have a pretty good sense of how applications interact with a blockchain network using a smart contract to query or update the ledger. You've seen the basics of the roles smart contracts, APIs, and the SDK play in queries and updates and you should have a feel for how different kinds of applications could be used to perform other business tasks and operations.

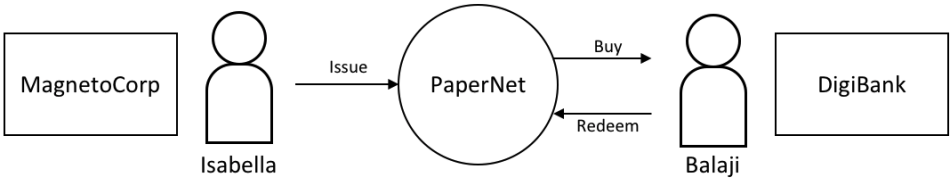
7.2.9 Additional resources

As we said in the introduction, we have a whole section on *Developing Applications* that includes in-depth information on smart contracts, process and data design, a tutorial using a more in-depth Commercial Paper tutorial and a large amount of other material relating to the development of applications.

7.3 Commercial paper tutorial

Audience: Architects, application and smart contract developers, administrators

This tutorial will show you how to install and use a commercial paper sample application and smart contract. It is a task-oriented topic, so it emphasizes procedures above concepts. When you'd like to understand the concepts in more detail, you can read the Developing Applications topic.



In this tutorial two organizations, MagnetoCorp and DigiBank, trade commercial paper with each other using PaperNet, a Hyperledger Fabric blockchain network.

Once you've set up the test network, you'll act as Isabella, an employee of MagnetoCorp, who will issue a commercial paper on its behalf. You'll then switch roles to take the role of Balaji, an employee of DigiBank, who will buy this commercial paper, hold it for a period of time, and then redeem it with MagnetoCorp for a small profit.

You'll act as an developer, end user, and administrator, each in different organizations, performing the following steps designed to help you understand what it's like to collaborate as two different organizations working independently, but according to mutually agreed rules in a Hyperledger Fabric network.

- *Set up machine* and *download samples*
- *Create the network*
- *Examine the commercial paper smart contract*
- *Deploy the smart contract to the channel* by approving the chaincode definition as MagnetoCorp and Digibank.
- Understand the structure of a MagnetoCorp *application*, including its *dependencies*
- Configure and use a *wallet and identities*
- Run a MagnetoCorp application to *issue a commercial paper*
- Understand how DigiBank uses the smart contract in their *applications*
- As Digibank, run applications that *buy* and *redeem* commercial paper

This tutorial has been tested on MacOS and Ubuntu, and should work on other Linux distributions. A Windows version is under development.

7.3.1 Prerequisites

Before you start, you must install some prerequisite technology required by the tutorial. We've kept these to a minimum so that you can get going quickly.

You **must** have the following technologies installed:

- **Node** version 8.9.0, or higher. Node is a JavaScript runtime that you can use to run applications and smart contracts. You are recommended to use the LTS (Long Term Support) version of node. Install node here.
- **Docker** version 18.06, or higher. Docker help developers and administrators create standard environments for building and running applications and smart contracts. Hyperledger Fabric is provided as a set of Docker images, and the PaperNet smart contract will run in a Docker container. Install Docker here.

You **will** find it helpful to install the following technologies:

- A source code editor, such as **Visual Studio Code** version 1.28, or higher. VS Code will help you develop and test your application and smart contract. Install VS Code here.

  Many excellent code editors are available including Atom, Sublime Text and Brackets.

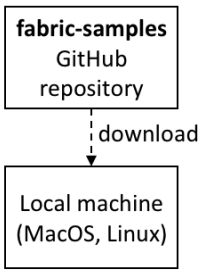
You **may** find it helpful to install the following technologies as you become more experienced with application and smart contract development. There's no requirement to install these when you first run the tutorial:

- **Node Version Manager**. NVM helps you easily switch between different versions of node – it can be really helpful if you're working on multiple projects at the same time. Install NVM here.

7.3.2 Download samples

The commercial paper tutorial is one of the samples in the `fabric-samples` directory. Before you begin this tutorial, ensure that you have followed the instructions to install the Fabric Prerequisites and Download the Samples, Binaries and Docker Images. When you are finished, you will have cloned the `fabric-samples` repository that contains the tutorial scripts, smart contract, and application files.

`https://github.com/hyperledger/fabric-samples`



Download the `fabric-samples` GitHub repository to your local machine.

After downloading, feel free to examine the directory structure of `fabric-samples`:

```
$ cd fabric-samples
$ ls

CODEOWNERS             basic-network              first-network
CODE_OF_CONDUCT.md     chaincode                  high-throughput
CONTRIBUTING.md        chaincode-docker-devmode   interest_rate_swaps
Jenkinsfile            ci                         off_chain_data
LICENSE                ci.properties              scripts
MAINTAINERS.md         commercial-paper           test-network
```

(continues on next page)

(continued from previous page)

```
README.md              docs
SECURITY.md            fabcar
```

Notice the `commercial-paper` directory – that's where our sample is located!

You've now completed the first stage of the tutorial! As you proceed, you'll open multiple command windows for different users and components. For example:

- To show peer, orderer and CA log output from your network.
- To approve the chaincode as an administrator from MagnetoCorp and as an administrator from DigiBank.
- To run applications on behalf of Isabella and Balaji, who will use the smart contract to trade commercial paper with each other.

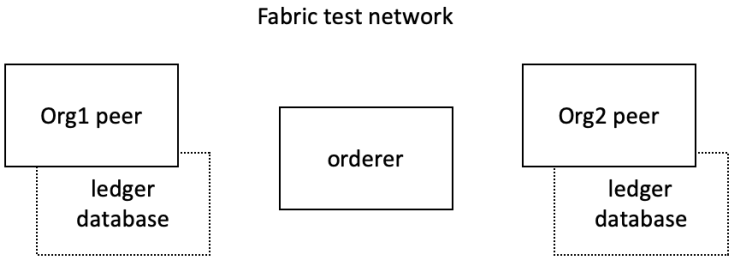
We'll make it clear when you should run a command from particular command window; for example:

```
(isabella)$ ls
```

indicates that you should run the `ls` command from Isabella's window.

7.3.3 Create the network

This tutorial will deploy a smart contract using the Fabric test network. The test network consists of two peer organizations and an ordering organization. The two peer organizations operate one peer each, while the ordering organization operates a single node raft ordering service. We will also use the test network to create a single channel named `mychannel` that both peer organizations will be members of.



The Fabric test network is comprised of two peer organizations, Org1 and Org2, each with one peer and its ledger database, an ordering node. Each of these components runs as a Docker container.

The two peers, the peer ledgers, the orderer and the CA each run in the their own Docker container. In production environments, organizations typically use existing CAs that are shared with other systems; they're not dedicated to the Fabric network.

The two organizations of the test network allow us to interact with a blockchain ledger as two organizations that operate separate peers. In this tutorial, we will operate Org1 of the test network as DigiBank and Org2 as MagnetoCorp.

You can start the test network and create the channel with a script provided in the commercial paper directory. Change to the `commercial-paper` directory in the `fabric-samples`:

```
cd fabric-samples/commercial-paper
```

Then use the script to start the test network:

```
./network-starter.sh
```

If the command is successful, you will see the test network being created in your logs. You can use the `docker ps` command to see the Fabric nodes running on your local machine:

```
$ docker ps

CONTAINER ID        IMAGE                               COMMAND                  ␣
↪CREATED             STATUS              PORTS                                       ␣
↪ NAMES
321cc489b10f        hyperledger/fabric-peer:latest      "peer node start"        2␣
↪minutes ago        Up 2 minutes        0.0.0.0:7051->7051/tcp                      ␣
↪peer0.org1.example.com
ad668671f95f        hyperledger/fabric-peer:latest      "peer node start"        2␣
↪minutes ago        Up 2 minutes        7051/tcp, 0.0.0.0:9051->9051/tcp            ␣
↪peer0.org2.example.com
caadbe4d8592        hyperledger/fabric-couchdb          "tini -- /docker-ent..."   2␣
↪minutes ago        Up 2 minutes        4369/tcp, 9100/tcp, 0.0.0.0:7984->5984/tcp  ␣
↪couchdb1
ebabe52903b8        hyperledger/fabric-couchdb          "tini -- /docker-ent..."   2␣
↪minutes ago        Up 2 minutes        4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp  ␣
↪couchdb0
7c72711c6e18        hyperledger/fabric-orderer:latest   "orderer"                2␣
↪minutes ago        Up 2 minutes        0.0.0.0:7050->7050/tcp                      ␣
↪orderer.example.com
```

See if you can map these containers to the nodes of the test network (you may need to horizontally scroll to locate the information):

- The Org1 peer, `peer0.org1.example.com`, is running in container `321cc489b10f`
- The Org2 peer, `peer0.org2.example.com`, is running in container `ad668671f95f`
- The CouchDB database for the Org1 peer, `couchdb0`, is running in container `ebabe52903b8`
- The CouchDB database for the Org2 peer, `couchdb1`, is running in container `caadbe4d8592`
- The ordering node `orderer.example.com` is running in container `7c72711c6e18`

These containers all form a Docker network called `net_test`. You can view the network with the `docker network` command:

```
$ docker network inspect net_test

    {
          "Name": "net_test",
          "Id": "b77b99d29e37677fac48b7ecd78383bdebf09ebdd6b00e87e3d9444252b1ce31",
          "Created": "2020-01-30T23:04:39.6157465Z",
          "Containers": {
              "321cc489b10ff46554d0b215da307d38daf35b68bbea635ae0ae3176c3ae0945": {
                  "Name": "peer0.org1.example.com",
                  "IPv4Address": "192.168.224.5/20",
              },
              "7c72711c6e18caf7bff4cf78c27efc9ef3b2359a749c926c8aba1beacfdb0211": {
                  "Name": "orderer.example.com",
                  "IPv4Address": "192.168.224.4/20",
              },
              "ad668671f95f351f0119320198e1d1e19ebbb0d75766c6c8b9bb7bd36ba506af": {
                  "Name": "peer0.org2.example.com",
```

(continues on next page)

(continued from previous page)

```
                "IPv4Address": "192.168.224.6/20",
            },
            "caadbe4d8592aa558fe14d07a424a9e04365620ede1143b6ce5902ce038c0851": {
                "Name": "couchdb1",
                "IPv4Address": "192.168.224.2/20",
            },
            "ebabe52903b8597d016dbc0d0ca4373ef75162d3400efbe6416975abafd08a8f": {
                "Name": "couchdb0",
                "IPv4Address": "192.168.224.3/20",
            }
        },
        "Labels": {}
    }
]
```

See how the five containers use different IP addresses, while being part of a single Docker network. (We've abbreviated the output for clarity.)

Because we are operating the test network as DigiBank and MagnetoCorp, `peer0.org1.example.com` will belong to the DigiBank organization while `peer0.org2.example.com` will be operated by MagnetoCorp. Now that the test network is up and running, we can refer our network as PaperNet from this point forward.

To recap: you've downloaded the Hyperledger Fabric samples repository from GitHub and you've got the test network running on your local machine. Let's now start to play the role of MagnetoCorp, who wish to trade commercial paper.

7.3.4 Monitor the network as MagnetoCorp

The commercial paper tutorial allows you to act as two organizations by providing two separate folders for DigiBank and MagnetoCorp. The two folders contain the smart contracts and application files for each organization. Because the two organizations have different roles in the trading of the commercial paper, the application files are different for each organization. Open a new window in the `fabric-samples` repository and use the following command to change into the MagnetoCorp directory:

```
cd commercial-paper/organization/magnetocorp
```

The first thing we are going to do as MagnetoCorp is monitor the components of PaperNet. An administrator can view the aggregated output from a set of Docker containers using the `logspout` tool. The tool collects the different output streams into one place, making it easy to see what's happening from a single window. This can be really helpful for administrators when installing smart contracts or for developers when invoking smart contracts, for example.

In the MagnetoCorp directory, run the following command to run the `monitordocker.sh` script and start the `logspout` tool for the containers associated with PaperNet running on `net_test`:

```
(magnetocorp admin)$ ./configuration/cli/monitordocker.sh net_test
...
latest: Pulling from gliderlabs/logspout
4fe2ade4980c: Pull complete
decca452f519: Pull complete
(...)
Starting monitoring on all containers on the network net_test
b7f3586e5d0233de5a454df369b8eadab0613886fc9877529587345fc01a3582
```

Note that you can pass a port number to the above command if the default port in `monitordocker.sh` is already in use.

```
(magnetocorp admin)$ ./monitordocker.sh net_test <port_number>
```

This window will now show output from the Docker containers for the remainder of the tutorial, so go ahead and open another command window. The next thing we will do is examine the smart contract that MagnetoCorp will use to issue to the commercial paper.

7.3.5 Examine the commercial paper smart contract

`issue`, `buy` and `redeem` are the three functions at the heart of the commercial paper smart contract. It is used by applications to submit transactions which correspondingly issue, buy and redeem commercial paper on the ledger. Our next task is to examine this smart contract.

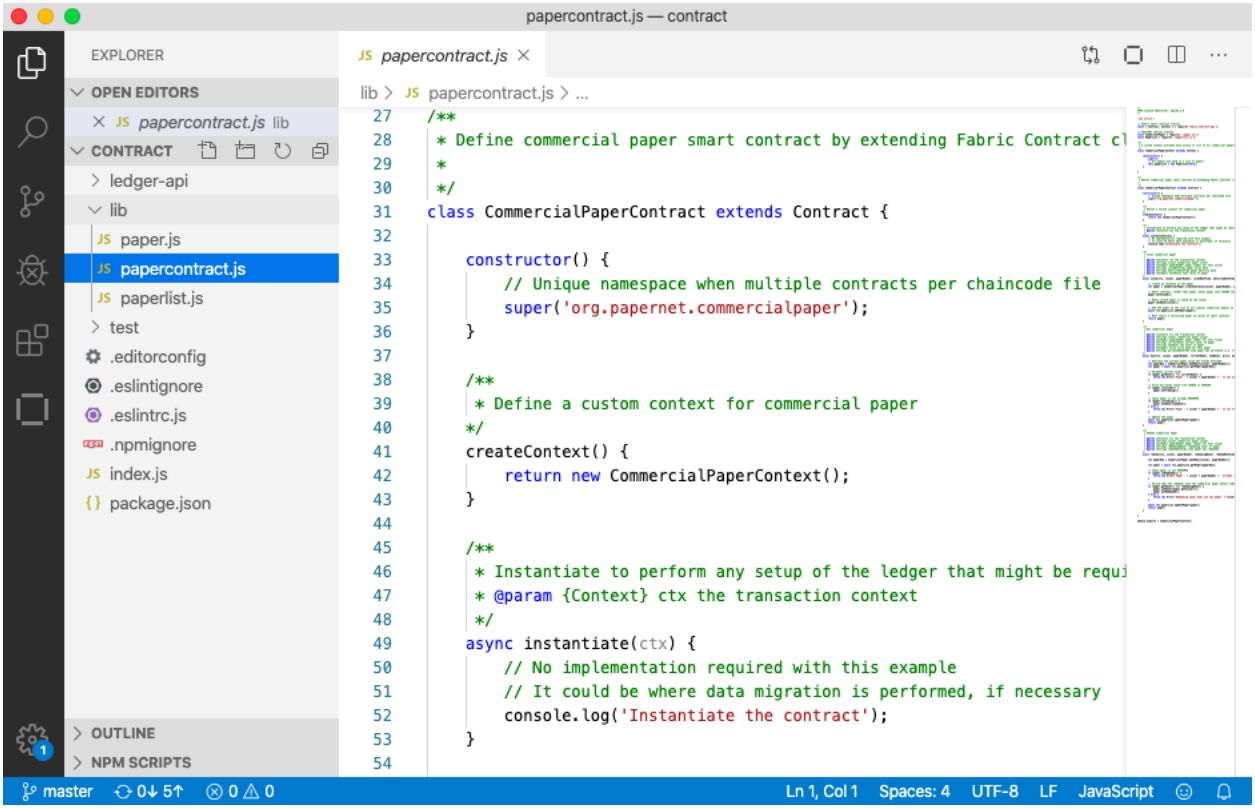
Open a new terminal in the `fabric-samples` directory and change into the MagnetoCorp folder to act as the MagnetoCorp developer.

```
cd commercial-paper/organization/magnetocorp
```

You can then view the smart contract in the `contract` directory using your chosen editor (VS Code in this tutorial):

```
(magnetocorp developer)$ code contract
```

In the `lib` directory of the folder, you'll see `papercontract.js` file – this contains the commercial paper smart contract!



An example code editor displaying the commercial paper smart contract in `papercontract.js`

`papercontract.js` is a JavaScript program designed to run in the Node.js environment. Note the following key program lines:

- `const { Contract, Context } = require('fabric-contract-api');`

This statement brings into scope two key Hyperledger Fabric classes that will be used extensively by the smart contract – `Contract` and `Context`. You can learn more about these classes in the `fabric-shim` JSDOCS.

- `class CommercialPaperContract extends Contract {`

  This defines the smart contract class `CommercialPaperContract` based on the built-in Fabric `Contract` class. The methods which implement the key transactions to `issue`, `buy` and `redeem` commercial paper are defined within this class.

- `async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime...) {`

  This method defines the commercial paper `issue` transaction for PaperNet. The parameters that are passed to this method will be used to create the new commercial paper.

  Locate and examine the `buy` and `redeem` transactions within the smart contract.

- `let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime...);`

  Within the `issue` transaction, this statement creates a new commercial paper in memory using the `CommercialPaper` class with the supplied transaction inputs. Examine the `buy` and `redeem` transactions to see how they similarly use this class.

- `await ctx.paperList.addPaper(paper);`

  This statement adds the new commercial paper to the ledger using `ctx.paperList`, an instance of a `PaperList` class that was created when the smart contract context `CommercialPaperContext` was initialized. Again, examine the `buy` and `redeem` methods to see how they use this class.

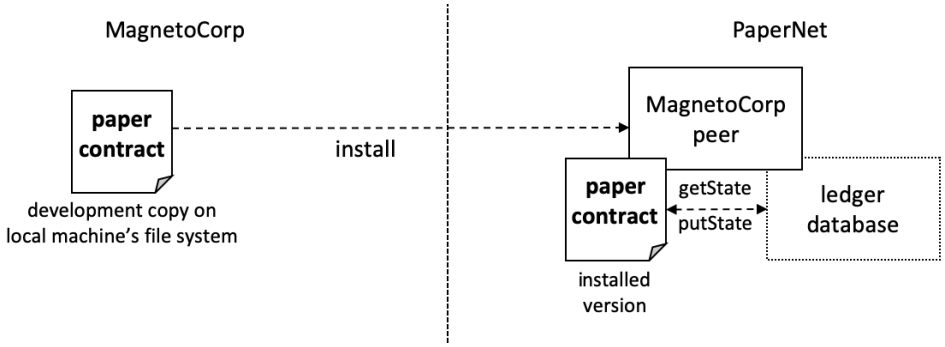
- `return paper;`

  This statement returns a binary buffer as response from the `issue` transaction for processing by the caller of the smart contract.

Feel free to examine other files in the `contract` directory to understand how the smart contract works, and read in detail how `papercontract.js` is designed in the smart contract topic.

7.3.6 Deploy the smart contract to the channel

Before `papercontract` can be invoked by applications, it must be installed onto the appropriate peer nodes of the test network and then defined on the channel using the Fabric chaincode lifecycle. The Fabric chaincode lifecycle allows multiple organizations to agree to the parameters of a chaincode before the chainocde is deployed to a channel. As a result, we need to install and approve the chaincode as administrators of both MagnetoCorp and DigiBank.



A MagnetoCorp administrator installs a copy of the `papercontract` onto a MagnetoCorp peer.

Smart contracts are the focus of application development, and are contained within a Hyperledger Fabric artifact called chaincode. One or more smart contracts can be defined within a single chaincode, and installing a chaincode will allow

them to be consumed by the different organizations in PaperNet. It means that only administrators need to worry about chaincode; everyone else can think in terms of smart contracts.

Install and approve the smart contract as MagnetoCorp

We will first install and approve the smart contract as the MagnetoCorp admin. Make sure that you are operating from the `magnetocorp` folder, or navigate back to that folder using the following command:

```
cd commercial-paper/organization/magnetocorp
```

A MagnetoCorp administrator can interact with PaperNet using the `peer` CLI. However, the administrator needs to set certain environment variables in their command window to use the correct set of `peer` binaries, send commands to the address of the MagnetoCorp peer, and sign requests with the correct crypto material.

You can use a script provided by the sample to set the environment variables in your command window. Run the following command in the `magnetocorp` directory:

```
source magnetocorp.sh
```

You will see the full list of environment variables printed in your window. We can now use this command window to interact with PaperNet as the MagnetoCorp administrator.

The first step is to install the `papercontract` smart contract. The smart contract can be packaged into a chaincode using the `peer lifecycle chaincode package` command. In the MagnetoCorp administrator's command window, run the following command to create the chaincode package:

```
(magnetocorp admin)$ peer lifecycle chaincode package cp.tar.gz --lang node --path ./
↪contract --label cp_0
```

The MagnetoCorp admin can now install the chaincode on the MagnetoCorp peer using the `peer lifecycle chaincode install` command:

```
(magnetocorp admin)$ peer lifecycle chaincode install cp.tar.gz
```

If the command is successful, you will see messages similar to the following printed in your terminal:

```
2020-01-30 18:32:33.762 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪001 Installed remotely: response:<status:200 payload:"\nEcp_
↪0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c\022\004cp_0" >
2020-01-30 18:32:33.762 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪002 Chaincode code package identifier: cp_
↪0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

Because the MagnetoCorp admin has set `CORE_PEER_ADDRESS=localhost:9051` to target its commands to `peer0.org2.example.com`, the `INFO 001 Installed remotely...` indicates that `papercontract` has been successfully installed on this peer.

After we install the smart contract, we need to approve the chaincode definition for `papercontract` as MagnetoCorp. The first step is to find the packageID of the chaincode we installed on our peer. We can query the packageID using the `peer lifecycle chaincode queryinstalled` command:

```
peer lifecycle chaincode queryinstalled
```

The command will return the same package identifier as the install command. You should see output similar to the following:

```
Installed chaincodes on peer:
Package ID: cp_0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c,␣
↪Label: cp_0
```

We will need the package ID in the next step, so we will save it as an environment variable. The package ID may not be the same for all users, so you need to complete this step using the package ID returned from your command window.

```
export PACKAGE_ID=cp_
↪0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

The admin can now approve the chaincode definition for MagnetoCorp using the `peer lifecycle chaincode approveformyorg` command:

```
(magnetocorp admin)$ peer lifecycle chaincode approveformyorg --orderer␣
↪localhost:7050 --ordererTLSHostnameOverride orderer.example.com --channelID␣
↪mychannel --name papercontract -v 0 --package-id $PACKAGE_ID --sequence 1 --tls --
↪cafile $ORDERER_CA
```

One of the most important chaincode parameters that channel members need to agree to using the chaincode definition is the chaincode endorsement policy. The endorsement policy describes the set of organizations that must endorse (execute and sign) a transaction before it can be determined to be valid. By approving the `papercontract` chaincode without the `--policy` flag, the MagnetoCorp admin agrees to using the default endorsement policy, which requires a majority of organizations on the channel to endorse a transaction. All transactions, whether valid or invalid, will be recorded on the ledger blockchain, but only valid transactions will update the world state.

Install and approve the smart contract as DigiBank

By default, the Fabric Chaincode lifecycle requires a majority of organizations on the channel to successfully commit the chaincode definition to the channel. This implies that we need to approve the `papernet` chaincode as both MagnetoCorp and DigiBank to get the required majority of 2 out of 2. Open a new terminal window in the `fabric-samples` and navigate to the older that contains the DigiBank smart contract and application files:

```
(digibank admin)$ cd commercial-paper/organization/digibank/
```

Use the script in the DigiBank folder to set the environment variables that will allow you to act as the DigiBank admin:

```
source digibank.sh
```

We can now install and approve `papercontract` as the DigiBank. Run the following command to package the chaincode:

```
(digibank admin)$ peer lifecycle chaincode package cp.tar.gz --lang node --path ./
↪contract --label cp_0
```

The admin can now install the chaincode on the DigiBank peer:

```
(digibank admin)$ peer lifecycle chaincode install cp.tar.gz
```

We then need to query and save the packageID of the chaincode that was just installed:

```
(digibank admin)$ peer lifecycle chaincode queryinstalled
```

Save the package ID as an environment variable. Complete this step using the package ID returned from your console.

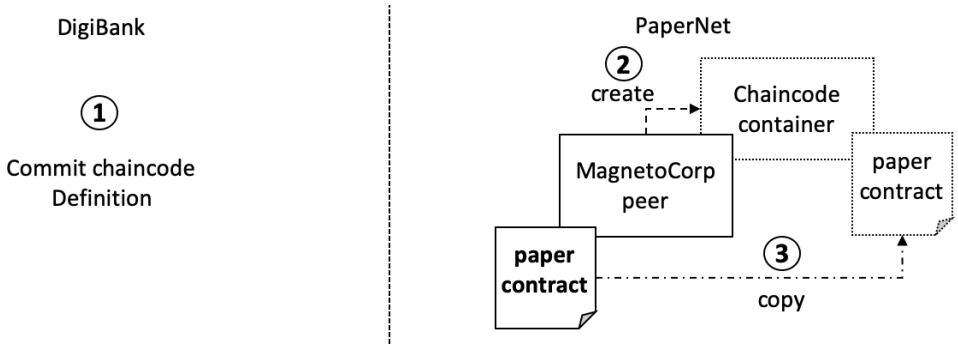
```
export PACKAGE_ID=cp_
↪0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

The Digibank admin can now approve the chaincode definition of `papercontract`:

```
(digibank admin)$ peer lifecycle chaincode approveformyorg --orderer localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name␣
↪papercontract -v 0 --package-id $PACKAGE_ID --sequence 1 --tls --cafile $ORDERER_CA
```

Commit the chaincode definition to the channel

Now that DigiBank and MagnetoCorp have both approved the `papernet` chaincode, we have the majority we need (2 out of 2) to commit the chaincode definition to the channel. Once the chaincode is successfully defined on the channel, the `CommercialPaper` smart contract inside the `papercontract` chaincode can be invoked by client applications on the channel. Since either organization can commit the chaincode to the channel, we will continue operating as the DigiBank admin:



After the DigiBank administrator commits the definition of the `papercontract` chaincode to the channel, a new Docker chaincode container will be created to run `papercontract` on both PaperNet peers

The DigiBank administrator uses the `peer lifecycle chaincode commit` command to commit the chaincode definition of `papercontract` to `mychannel`:

```
(digibank admin)$ peer lifecycle chaincode commit -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --peerAddresses localhost:7051 --
↪tlsRootCertFiles ${PEER0_ORG1_CA} --peerAddresses localhost:9051 --tlsRootCertFiles
↪${PEER0_ORG2_CA} --channelID mychannel --name papercontract -v 0 --sequence 1 --tls␣
↪--cafile $ORDERER_CA --waitForEvent
```

The chaincode container will start after the chaincode definition has been committed to the channel. You can use the `docker ps` command to see `papercontract` container starting on both peers.

```
(digibank admin)$ docker ps

CONTAINER ID        IMAGE
↪                                                                                  ␣
↪               COMMAND                  CREATED             STATUS              ␣
↪PORTS                                       NAMES
d4ba9dc9c55f        dev-peer0.org1.example.com-cp_0-
↪ebef35e7f1f25eea1dcc6fcad5019477cd7f434c6a5dcaf4e81744e282903535-
↪05cf67c20543ee1c24cf7dfe74abce99785374db15b3bc1de2da372700c25608   "docker-
↪entrypoint.s..."   30 seconds ago      Up 28 seconds                              ␣
↪                    dev-peer0.org1.example.com-cp_0-
↪ebef35e7f1f25eea1dcc6fcad5019477cd7f434c6a5dcaf4e81744e282903535
```

(continues on next page)

(continued from previous page)

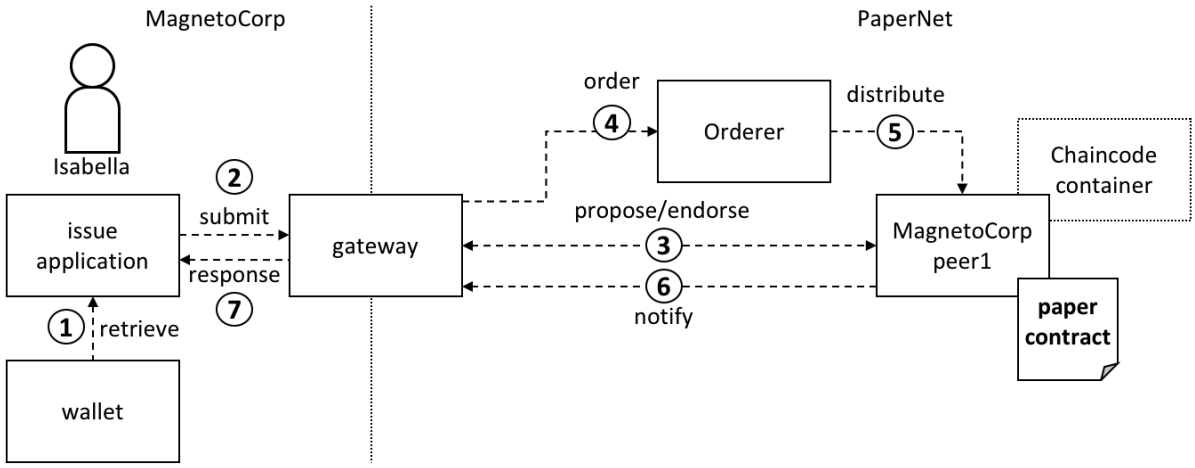
```
a944c0f8b6d6        dev-peer0.org2.example.com-cp_0-
↪1487670371e56d107b5e980ce7f66172c89251ab21d484c7f988c02912ddeaec-
↪1a147b6fd2a8bd2ae12db824fad8d08a811c30cc70bc5b6bc49a2cbebc2e71ee   "docker-
↪entrypoint.s..."   31 seconds ago      Up 28 seconds
↪                    dev-peer0.org2.example.com-cp_0-
↪1487670371e56d107b5e980ce7f66172c89251ab21d484c7f988c02912ddeaec
```

Notice that the containers are named to indicate the peer that started it, and the fact that it's running `papercontract` version `0`.

Now that we have deployed the `papercontract` chaincode to the channel, we can use the MagnetoCorp application to issue the commercial paper. Let's take a moment to examine the application structure.

7.3.7 Application structure

The smart contract contained in `papercontract` is called by MagnetoCorp's application `issue.js`. Isabella uses this application to submit a transaction to the ledger which issues commercial paper `00001`. Let's quickly examine how the `issue` application works.



A gateway allows an application to focus on transaction generation, submission and response. It coordinates transaction proposal, ordering and notification processing between the different network components.

Because the `issue` application submits transactions on behalf of Isabella, it starts by retrieving Isabella's X.509 certificate from her wallet, which might be stored on the local file system or a Hardware Security Module HSM. The `issue` application is then able to utilize the gateway to submit transactions on the channel. The Hyperledger Fabric SDK provides a gateway abstraction so that applications can focus on application logic while delegating network interaction to the gateway. Gateways and wallets make it straightforward to write Hyperledger Fabric applications.

So let's examine the `issue` application that Isabella is going to use. Open a separate terminal window for her, and in `fabric-samples` locate the MagnetoCorp `/application` folder:

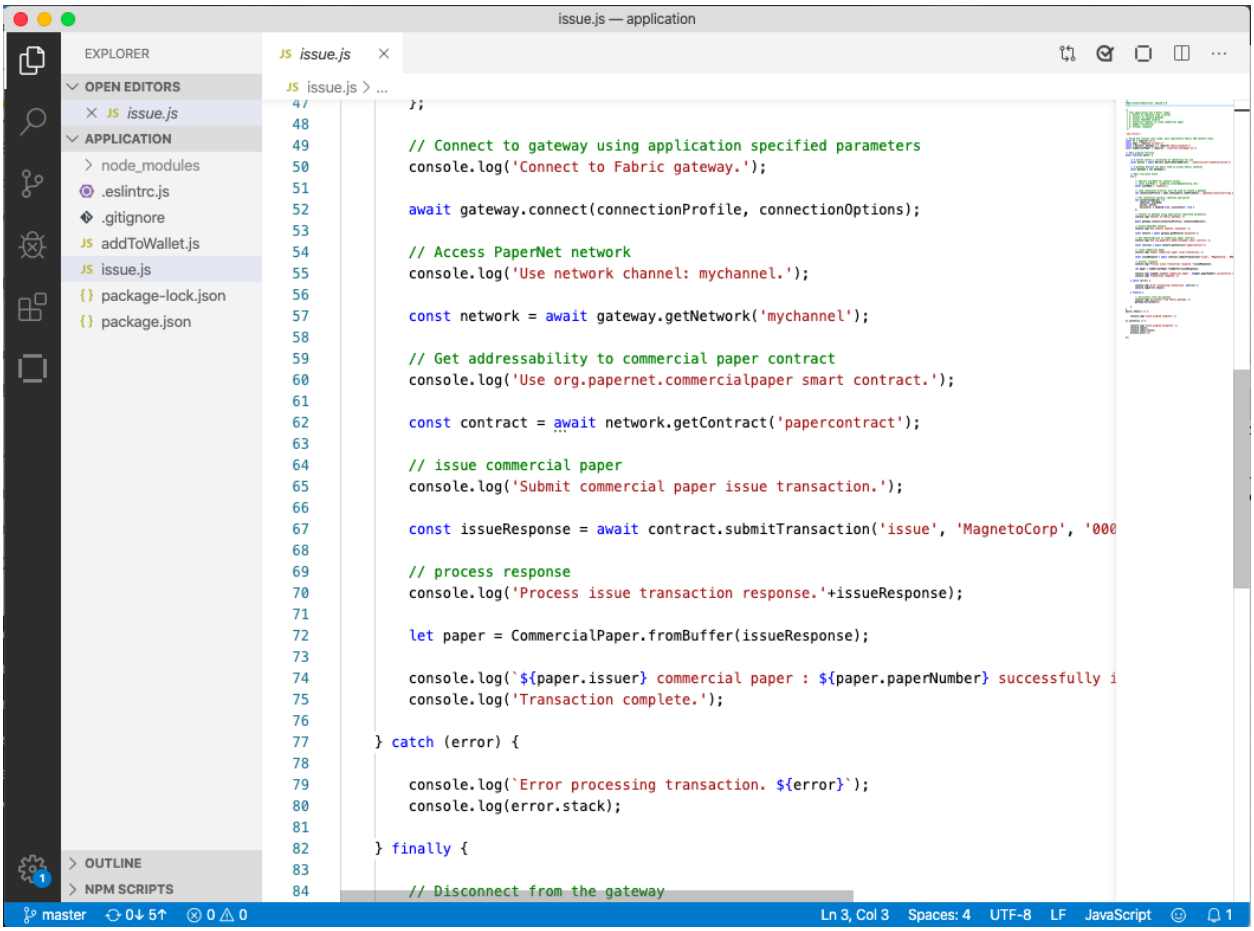
```
(isabella)$ cd commercial-paper/organization/magnetocorp/application/
(isabella)$ ls

addToWallet.js      issue.js        package.json
```

`addToWallet.js` is the program that Isabella is going to use to load her identity into her wallet, and `issue.js` will use this identity to create commercial paper `00001` on behalf of MagnetoCorp by invoking `papercontract`.

Change to the directory that contains MagnetoCorp's copy of the application `issue.js`, and use your code editor to examine it:

```
(isabella)$ cd commercial-paper/organization/magnetocorp/application
(isabella)$ code issue.js
```

Examine this directory; it contains the issue application and all its dependencies.



A code editor displaying the contents of the commercial paper application directory.

Note the following key program lines in `issue.js`:

- `const { Wallets, Gateway } = require('fabric-network');`

  This statement brings two key Hyperledger Fabric SDK classes into scope – `Wallet` and `Gateway`.M

- `const wallet = await Wallets.newFileSystemWallet('../identity/user/isabella/wallet');`

  This statement identifies that the application will use `isabella` wallet when it connects to the blockchain network channel. Because Isabella's X.509 certificate is in the local file system, the application creates a new `FileSystemWallet`. The application will select a particular identity within `isabella` wallet.

- `await gateway.connect(connectionProfile, connectionOptions);`

  This line of code connects to the network using the gateway identified by `connectionProfile`, using the identity referred to in `ConnectionOptions`.

  See how `../gateway/networkConnection.yaml` and `User1@org1.example.com` are used for these values respectively.

- `const network = await gateway.getNetwork('mychannel');`

  This connects the application to the network channel `mychannel`, where the `papercontract` was previously instantiated.

- `const contract = await network.getContract('papercontract');`

  This statement gives the application access to the `papercontract` chaincode. Once an application has issued getContract, it can submit to any smart contract transaction implemented within the chaincode.

- `const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001', ...);`

  This line of code submits the a transaction to the network using the `issue` transaction defined within the smart contract. `MagnetoCorp`, `00001...` are the values to be used by the `issue` transaction to create a new commercial paper.

- `let paper = CommercialPaper.fromBuffer(issueResponse);`

  This statement processes the response from the `issue` transaction. The response needs to deserialized from a buffer into `paper`, a `CommercialPaper` object which can interpreted correctly by the application.

Feel free to examine other files in the `/application` directory to understand how `issue.js` works, and read in detail how it is implemented in the application topic.

7.3.8 Application dependencies

The `issue.js` application is written in JavaScript and designed to run in the Node.js environment that acts as a client to the PaperNet network. As is common practice, MagnetoCorp's application is built on many external node packages — to improve quality and speed of development. Consider how `issue.js` includes the `js-yaml` package to process the YAML gateway connection profile, or the `fabric-network` package to access the `Gateway` and `Wallet` classes:

```
const yaml = require('js-yaml');
const { Wallets, Gateway } = require('fabric-network');
```

These packages have to be downloaded from npm to the local file system using the `npm install` command. By convention, packages must be installed into an application-relative `/node_modules` directory for use at runtime.

Examine the `package.json` file to see how `issue.js` identifies the packages to download and their exact versions:

```
  "dependencies": {
    "fabric-network": "~1.4.0",
    "fabric-client": "~1.4.0",
    "js-yaml": "^3.12.0"
  },
```

npm versioning is very powerful; you can read more about it here.

Let's install these packages with the `npm install` command – this may take up to a minute to complete:

```
(isabella)$ cd commercial-paper/organization/magnetocorp/application/
(isabella)$ npm install

(           ) extract:lodash: sill extract ansi-styles@3.2.1
(...)
added 738 packages in 46.701s
```

See how this command has updated the directory:

```
(isabella)$ ls

addToWallet.js      node_modules            package.json
issue.js            package-lock.json
```

Examine the `node_modules` directory to see the packages that have been installed. There are lots, because `js-yaml` and `fabric-network` are themselves built on other npm packages! Helpfully, the `package-lock.json` file identifies the exact versions installed, which can prove invaluable if you want to exactly reproduce environments; to test, diagnose problems or deliver proven applications for example.

7.3.9 Wallet

Isabella is almost ready to run `issue.js` to issue MagnetoCorp commercial paper `00001`; there's just one remaining task to perform! As `issue.js` acts on behalf of Isabella, and therefore MagnetoCorp, it will use identity from her wallet that reflects these facts. We now need to perform this one-time activity of adding appropriate X.509 credentials to her wallet.

In Isabella's terminal window, run the `addToWallet.js` program to add identity information to her wallet:

```
(isabella)$ node addToWallet.js

done
```

`addToWallet.js` is a simple file-copying program which you can examine at your leisure. It moves an identity from the test network sample to Isabella's wallet. Let's focus on the result of this program — the contents of the wallet which will be used to submit transactions to PaperNet:

```
(isabella)$ ls ../identity/user/isabella/wallet/

isabella.id
```

Isabella can store multiple identities in her wallet, though in our example, she only uses one. The `wallet` folder contains an `isabella.id` file that provides the information that Isabella needs to connect to the network. Other identities used by Isabella would have their own file. You can open this file to see the identity information that `issue.js` will use on behalf of Isabella inside a JSON file. The output has been formatted for clarity.

```
(isabella)$  cat ../identity/user/isabella/wallet/*

{
  "credentials": {
    "certificate": "-----BEGIN CERTIFICATE-----
↪\nMIICKTCCAdCgAwIBAgIQWKwvLG+sqeO3LwwQK6avZDAKBggqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMBEGA1UECBMKQ2F
↪53dbo00wSzAOBgNVHQ8BAf8EBAMCB4AwDAYDVR0TAQH/
↪BAIwADArBgNV\nHSMEJDAigCDOCdm4irsZFU3D6Hak4+84QRg1N43iwg8w1V6DRhgLyDAKBggqhkjO\nPQQDAgNHADBEAiBhzK
↪mRtUdaJagIgIgYpbZ\nXf0CSiTXIWOJIsswN4Jp+ZxkJfFVmXndqKqz+VM=\n-----END CERTIFICATE---
↪--\n",
    "privateKey": "-----BEGIN PRIVATE KEY-----
↪\nMIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQggs55vQg2oXi8gNi8\nNidE8Fy5zenohArDq3FGJD8cKU2hRAN
↪53db\n-----END PRIVATE KEY-----\n"
  },
  "mspId": "Org2MSP",
  "type": "X.509",
  "version": 1
}
```

In the file you can notice the following:

- a `"privateKey":` used to sign transactions on Isabella's behalf, but not distributed outside of her immediate control.
- a `"certificate":` which contains Isabella's public key and other X.509 attributes added by the Certificate Authority at certificate creation. This certificate is distributed to the network so that different actors at different times can cryptographically verify information created by Isabella's private key.

You can Learn more about certificates here. In practice, the certificate file also contains some Fabric-specific metadata such as Isabella's organization and role – read more in the wallet topic.

7.3.10 Issue application

Isabella can now use `issue.js` to submit a transaction that will issue MagnetoCorp commercial paper `00001`:

```
(isabella)$ node issue.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper issue transaction.
Process issue transaction response.{"class":"org.papernet.commercialpaper","key":"\
↪"MagnetoCorp\":\"00001\"","currentState":1,"issuer":"MagnetoCorp","paperNumber":
↪"00001","issueDateTime":"2020-05-31","maturityDateTime":"2020-11-30","faceValue":
↪"5000000","owner":"MagnetoCorp"}
MagnetoCorp commercial paper : 00001 successfully issued for value 5000000
Transaction complete.
Disconnect from Fabric gateway.
Issue program complete.
```

The `node` command initializes a Node.js environment, and runs `issue.js`. We can see from the program output that MagnetoCorp commercial paper 00001 was issued with a face value of 5M USD.

As you've seen, to achieve this, the application invokes the `issue` transaction defined in the `CommercialPaper` smart contract within `papercontract.js`. This had been installed and instantiated in the network by the MagnetoCorp administrator. It's the smart contract which interacts with the ledger via the Fabric APIs, most notably `putState()` and `getState()`, to represent the new commercial paper as a vector state within the world state. We'll see how this vector state is subsequently manipulated by the `buy` and `redeem` transactions also defined within the smart contract.

All the time, the underlying Fabric SDK handles the transaction endorsement, ordering and notification process, making the application's logic straightforward; the SDK uses a gateway to abstract away network details and connectionOptions to declare more advanced processing strategies such as transaction retry.

Let's now follow the lifecycle of MagnetoCorp 00001 by switching our emphasis to an employee of DigiBank, Balaji, who will buy the commercial paper using a DigiBank application.

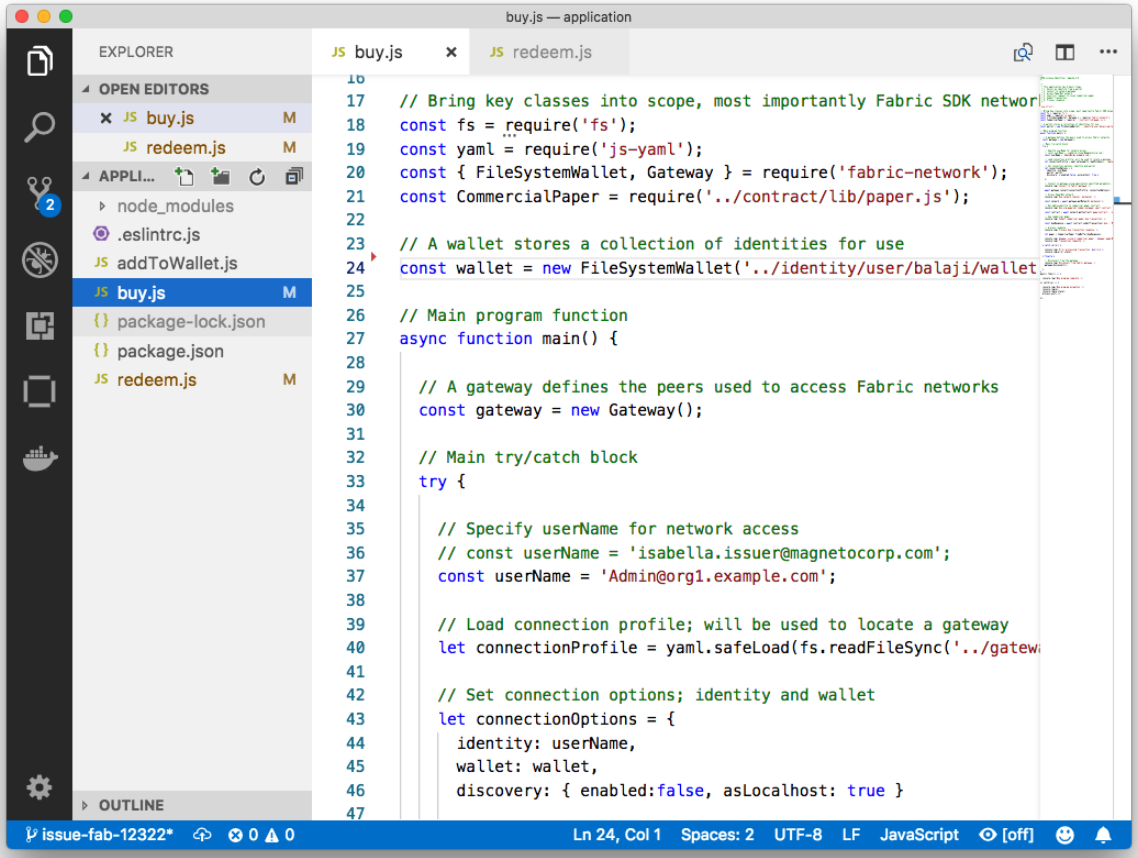
7.3.11 Digibank applications

Balaji uses DigiBank's `buy` application to submit a transaction to the ledger which transfers ownership of commercial paper `00001` from MagnetoCorp to DigiBank. The `CommercialPaper` smart contract is the same as that used by MagnetoCorp's application, however the transaction is different this time – it's `buy` rather than `issue`. Let's examine how DigiBank's application works.

Open a separate terminal window for Balaji. In `fabric-samples`, change to the DigiBank application directory that contains the application, `buy.js`, and open it with your editor:

```
(balaji)$ cd commercial-paper/organization/digibank/application/
(balaji)$ code buy.js
```

As you can see, this directory contains both the `buy` and `redeem` applications that will be used by Balaji.



DigiBank's commercial paper directory containing the `buy.js` and `redeem.js` applications.

DigiBank's `buy.js` application is very similar in structure to MagnetoCorp's `issue.js` with two important differences:

- **Identity**: the user is a DigiBank user `Balaji` rather than MagnetoCorp's `Isabella`

  ```
  const wallet = await Wallets.newFileSystemWallet('../identity/user/balaji/wallet
  ↪');
  ```

  See how the application uses the `balaji` wallet when it connects to the PaperNet network channel. `buy.js` selects a particular identity within `balaji` wallet.

- **Transaction**: the invoked transaction is `buy` rather than `issue`

  ```
  const buyResponse = await contract.submitTransaction('buy', 'MagnetoCorp', '00001
  ↪', ...);
  ```

  A `buy` transaction is submitted with the values `MagnetoCorp`, `00001`, `...`, that are used by the `CommercialPaper` smart contract class to transfer ownership of commercial paper `00001` to DigiBank.

Feel free to examine other files in the `application` directory to understand how the application works, and read in detail how `buy.js` is implemented in the application topic.

7.3.12 Run as DigiBank

The DigiBank applications which buy and redeem commercial paper have a very similar structure to MagnetoCorp's issue application. Therefore, let's install their dependencies and set up Balaji's wallet so that he can use these applications to buy and redeem commercial paper.

Like MagnetoCorp, Digibank must the install the required application packages using the `npm install` command, and again, this make take a short time to complete.

In the DigiBank administrator window, install the application dependencies:

```
(digibank admin)$ cd commercial-paper/organization/digibank/application/
(digibank admin)$ npm install

(            ) extract:lodash: sill extract ansi-styles@3.2.1
(...)
added 738 packages in 46.701s
```

In Balaji's command window, run the `addToWallet.js` program to add the identity to his wallet:

```
(balaji)$ node addToWallet.js

done
```

The `addToWallet.js` program has added identity information for `balaji`, to his wallet, which will be used by `buy.js` and `redeem.js` to submit transactions to `PaperNet`.

Like Isabella, Balaji can store multiple identities in his wallet, though in our example, he only uses one. His corresponding id file at `digibank/identity/user/balaji/wallet/balaji.id` is very similar Isabella's — feel free to examine it.

7.3.13 Buy application

Balaji can now use `buy.js` to submit a transaction that will transfer ownership of MagnetoCorp commercial paper 00001 to DigiBank.

Run the `buy` application in Balaji's window:

```
(balaji)$ node buy.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper buy transaction.
Process buy transaction response.
MagnetoCorp commercial paper : 00001 successfully purchased by DigiBank
Transaction complete.
Disconnect from Fabric gateway.
Buy program complete.
```

You can see the program output that MagnetoCorp commercial paper 00001 was successfully purchased by Balaji on behalf of DigiBank. `buy.js` invoked the `buy` transaction defined in the `CommercialPaper` smart contract which updated commercial paper `00001` within the world state using the `putState()` and `getState()` Fabric APIs. As you've seen, the application logic to buy and issue commercial paper is very similar, as is the smart contract logic.

7.3.14 Redeem application

The final transaction in the lifecycle of commercial paper 00001 is for DigiBank to redeem it with MagnetoCorp. Balaji uses `redeem.js` to submit a transaction to perform the redeem logic within the smart contract.

Run the `redeem` transaction in Balaji's window:

```
(balaji)$ node redeem.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper redeem transaction.
Process redeem transaction response.
MagnetoCorp commercial paper : 00001 successfully redeemed with MagnetoCorp
Transaction complete.
Disconnect from Fabric gateway.
Redeem program complete.
```

Again, see how the commercial paper 00001 was successfully redeemed when `redeem.js` invoked the `redeem` transaction defined in `CommercialPaper`. Again, it updated commercial paper `00001` within the world state to reflect that the ownership returned to MagnetoCorp, the issuer of the paper.

7.3.15 Clean up

When you are finished using the Commercial Paper tutorial, you can use a script to clean up your environment. Use a command window to navigate back to the root directory of the commercial paper sample:

```
cd fabric-samples/commercial-paper
```

You can then bring down the network with the following command:

```
./network-clean.sh
```

This command will bring down the peers, CouchDB containers, and ordering node of the network, in addition to the logspout tool. It will also remove the identities that we created for Isabella and Balaji. Note that all of the data on the ledger will be lost. If you want to go through the tutorial again, you will start from a clean initial state.

7.3.16 Further reading

To understand how applications and smart contracts shown in this tutorial work in more detail, you'll find it helpful to read Developing Applications. This topic will give you a fuller explanation of the commercial paper scenario, the PaperNet business network, its actors, and how the applications and smart contracts they use work in detail.

Also feel free to use this sample to start creating your own applications and smart contracts!

7.4 Using Private Data in Fabric

This tutorial will demonstrate the use of collections to provide storage and retrieval of private data on the blockchain network for authorized peers of organizations.

The information in this tutorial assumes knowledge of private data stores and their use cases. For more information, check out *Private data*.

Note: These instructions use the new Fabric chaincode lifecycle introduced in the Fabric v2.0 release. If you would like to use the previous lifecycle model to use private data with chaincode, visit the v1.4 version of the Using Private Data in Fabric tutorial.

The tutorial will take you through the following steps to practice defining, configuring and using private data with Fabric:

1. *Build a collection definition JSON file*
2. *Read and Write private data using chaincode APIs*
3. *Install and define a chaincode with a collection*
4. *Store private data*
5. *Query the private data as an authorized peer*
6. *Query the private data as an unauthorized peer*
7. *Purge Private Data*
8. *Using indexes with private data*
9. *Additional resources*

This tutorial will deploy the marbles private data sample to the Fabric test network to demonstrate how to create, deploy, and use a collection of private data. You should have completed the task *Install Samples, Binaries, and Docker Images*.

7.4.1 Build a collection definition JSON file

The first step in privatizing data on a channel is to build a collection definition which defines access to the private data.

The collection definition describes who can persist data, how many peers the data is distributed to, how many peers are required to disseminate the private data, and how long the private data is persisted in the private database. Later, we will demonstrate how chaincode APIs `PutPrivateData` and `GetPrivateData` are used to map the collection to the private data being secured.

A collection definition is composed of the following properties:

- `name`: Name of the collection.
- `policy`: Defines the organization peers allowed to persist the collection data.
- `requiredPeerCount`: Number of peers required to disseminate the private data as a condition of the endorsement of the chaincode
- `maxPeerCount`: For data redundancy purposes, the number of other peers that the current endorsing peer will attempt to distribute the data to. If an endorsing peer goes down, these other peers are available at commit time if there are requests to pull the private data.
- `blockToLive`: For very sensitive information such as pricing or personal information, this value represents how long the data should live on the private database in terms of blocks. The data will live for this specified number of blocks on the private database and after that it will get purged, making this data obsolete from the network. To keep private data indefinitely, that is, to never purge private data, set the `blockToLive` property to `0`.
- `memberOnlyRead`: a value of `true` indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed read access to private data.

To illustrate usage of private data, the marbles private data example contains two private data collection definitions: `collectionMarbles` and `collectionMarblePrivateDetails`. The `policy` property in the `collectionMarbles` definition allows all members of the channel (Org1 and Org2) to have the private data in a private database. The `collectionMarblesPrivateDetails` collection allows only members of Org1 to have the private data in their private database.

For more information on building a policy definition refer to the *Endorsement policies* topic.

```
// collections_config.json

[
  {
        "name": "collectionMarbles",
        "policy": "OR('Org1MSP.member', 'Org2MSP.member')",
        "requiredPeerCount": 0,
        "maxPeerCount": 3,
        "blockToLive":1000000,
        "memberOnlyRead": true
  },

  {
        "name": "collectionMarblePrivateDetails",
        "policy": "OR('Org1MSP.member')",
        "requiredPeerCount": 0,
        "maxPeerCount": 3,
        "blockToLive":3,
        "memberOnlyRead": true
  }
]
```

The data to be secured by these policies is mapped in chaincode and will be shown later in the tutorial.

This collection definition file is deployed when the chaincode definition is committed to the channel using the peer lifecycle chaincode commit command. More details on this process are provided in Section 3 below.

7.4.2 Read and Write private data using chaincode APIs

The next step in understanding how to privatize data on a channel is to build the data definition in the chaincode. The marbles private data sample divides the private data into two separate data definitions according to how the data will be accessed.

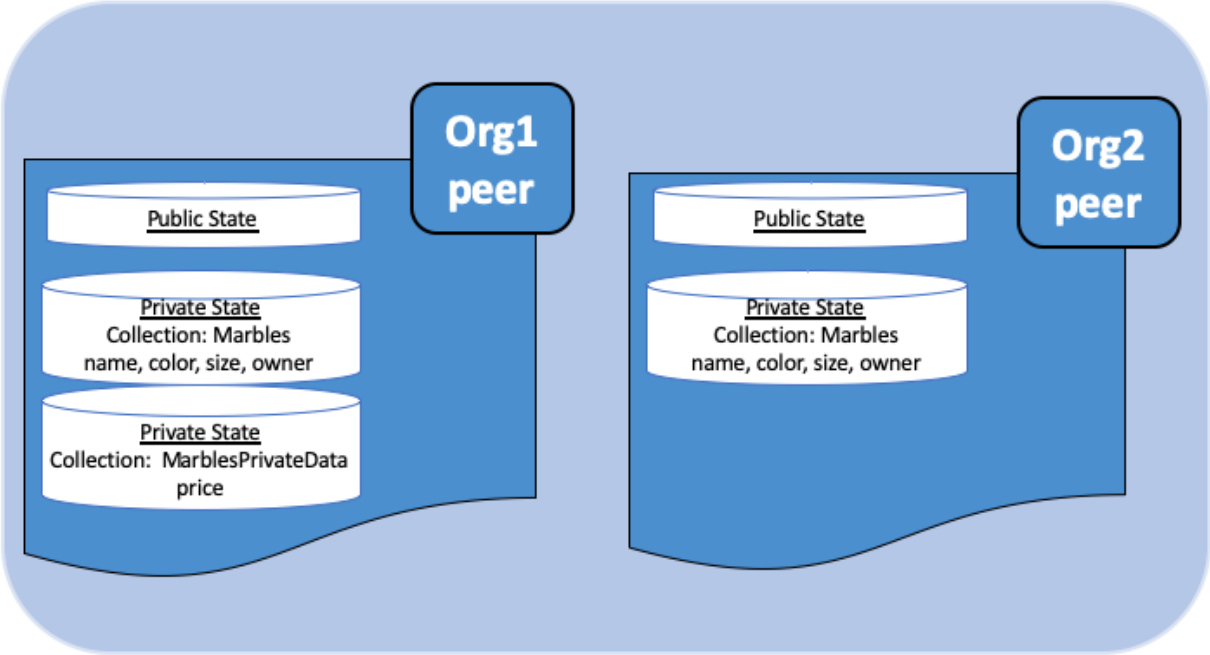
```
// Peers in Org1 and Org2 will have this private data in a side database
type marble struct {
  ObjectType string `json:"docType"`
  Name       string `json:"name"`
  Color      string `json:"color"`
  Size       int    `json:"size"`
  Owner      string `json:"owner"`
}

// Only peers in Org1 will have this private data in a side database
type marblePrivateDetails struct {
  ObjectType string `json:"docType"`
  Name       string `json:"name"`
  Price      int    `json:"price"`
}
```

Specifically access to the private data will be restricted as follows:

- `name, color, size, and owner` will be visible to all members of the channel (Org1 and Org2)
- `price` only visible to members of Org1

Thus two different sets of private data are defined in the marbles private data sample. The mapping of this data to the collection policy which restricts its access is controlled by chaincode APIs. Specifically, reading and writing private data using a collection definition is performed by calling `GetPrivateData()` and `PutPrivateData()`, which can be found here.

The following diagram illustrates the private data model used by the marbles private data sample.



Reading collection data

Use the chaincode API `GetPrivateData()` to query private data in the database. `GetPrivateData()` takes two arguments, the **collection name** and the data key. Recall the collection `collectionMarbles` allows members of Org1 and Org2 to have the private data in a side database, and the collection `collectionMarblePrivateDetails` allows only members of Org1 to have the private data in a side database. For implementation details refer to the following two marbles private data functions:

- **readMarble** for querying the values of the `name, color, size and owner` attributes
- **readMarblePrivateDetails** for querying the values of the `price` attribute

When we issue the database queries using the peer commands later in this tutorial, we will call these two functions.

Writing private data

Use the chaincode API `PutPrivateData()` to store the private data into the private database. The API also requires the name of the collection. Since the marbles private data sample includes two different collections, it is called twice in the chaincode:

1. Write the private data `name, color, size and owner` using the collection named `collectionMarbles`.

2. Write the private data `price` using the collection named `collectionMarblePrivateDetails`.

For example, in the following snippet of the `initMarble` function, `PutPrivateData()` is called twice, once for each set of private data.

```
// ==== Create marble object, marshal to JSON, and save to state ====
      marble := &marble{
              ObjectType: "marble",
              Name:       marbleInput.Name,
              Color:      marbleInput.Color,
              Size:       marbleInput.Size,
              Owner:      marbleInput.Owner,
      }
      marbleJSONasBytes, err := json.Marshal(marble)
      if err != nil {
              return shim.Error(err.Error())
      }

      // === Save marble to state ===
      err = stub.PutPrivateData("collectionMarbles", marbleInput.Name,␣
↪marbleJSONasBytes)
      if err != nil {
              return shim.Error(err.Error())
      }

      // ==== Create marble private details object with price, marshal to JSON, and␣
↪save to state ====
      marblePrivateDetails := &marblePrivateDetails{
              ObjectType: "marblePrivateDetails",
              Name:       marbleInput.Name,
              Price:      marbleInput.Price,
      }
      marblePrivateDetailsBytes, err := json.Marshal(marblePrivateDetails)
      if err != nil {
              return shim.Error(err.Error())
      }
      err = stub.PutPrivateData("collectionMarblePrivateDetails", marbleInput.Name,␣
↪marblePrivateDetailsBytes)
      if err != nil {
              return shim.Error(err.Error())
      }
```

To summarize, the policy definition above for our `collection.json` allows all peers in Org1 and Org2 to store and transact with the marbles private data `name, color, size, owner` in their private database. But only peers in Org1 can store and transact with the `price` private data in its private database.

As an additional data privacy benefit, since a collection is being used, only the private data hashes go through orderer, not the private data itself, keeping private data confidential from orderer.

7.4.3 Start the network

Now we are ready to step through some commands which demonstrate how to use private data.

Try it yourself

Before installing, defining, and using the marbles private data chaincode below, we need to start the Fabric test network. For the sake of this tutorial, we want to operate from a known initial state. The following command will kill any active

or stale Docker containers and remove previously generated artifacts. Therefore let's run the following command to clean up any previous environments:

```
cd fabric-samples/test-network
./network.sh down
```

If you have not run through the tutorial before, you will need to vendor the chaincode dependencies before we can deploy it to the network. Run the following commands:

```
cd ../chaincode/marbles02_private/go
GO111MODULE=on go mod vendor
cd ../../../test-network
```

If you've already run through this tutorial, you'll also want to delete the underlying Docker containers for the marbles private data chaincode. Let's run the following commands to clean up previous environments:

```
docker rm -f $(docker ps -a | awk '($2 ~ /dev-peer.*.marblesp.*/) {print $1}')
docker rmi -f $(docker images | awk '($1 ~ /dev-peer.*.marblesp.*/) {print $3}')
```

From the `test-network` directory, you can use the following command to start up the Fabric test network with CouchDB:

```
./network.sh up createChannel -s couchdb
```

This command will deploy a Fabric network consisting of a single channel named `mychannel` with two organizations (each maintaining one peer node) and an ordering service while using CouchDB as the state database. Either LevelDB or CouchDB may be used with collections. CouchDB was chosen to demonstrate how to use indexes with private data.

Note: For collections to work, it is important to have cross organizational gossip configured correctly. Refer to our documentation on *Gossip data dissemination protocol*, paying particular attention to the section on "anchor peers". Our tutorial does not focus on gossip given it is already configured in the test network, but when configuring a channel, the gossip anchors peers are critical to configure for collections to work properly.

7.4.4 Install and define a chaincode with a collection

Client applications interact with the blockchain ledger through chaincode. Therefore we need to install a chaincode on every peer that will execute and endorse our transactions. However, before we can interact with our chaincode, the members of the channel need to agree on a chaincode definition that establishes chaincode governance, including the private data collection configuration. We are going to package, install, and then define the chaincode on the channel using *peer lifecycle chaincode*.

The chaincode needs to be packaged before it can be installed on our peers. We can use the peer lifecycle chaincode package command to package the marbles chaincode.

The test network includes two organizations, Org1 and Org2, with one peer each. Therefore, the chaincode package has to be installed on two peers:

- peer0.org1.example.com
- peer0.org2.example.com

After the chaincode is packaged, we can use the peer lifecycle chaincode install command to install the Marbles chaincode on each peer.

Try it yourself

Assuming you have started the test network, copy and paste the following environment variables in your CLI to interact with the network and operate as the Org1 admin. Make sure that you are in the *test-network* directory.

```
export PATH=${PWD}/../bin:${PWD}:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

1. Use the following command to package the marbles private data chaincode.

```
peer lifecycle chaincode package marblesp.tar.gz --path ../chaincode/marbles02_
↪private/go/ --lang golang --label marblespv1
```

This command will create a chaincode package named marblesp.tar.gz.

2. Use the following command to install the chaincode package onto the peer `peer0.org1.example.com`.

```
peer lifecycle chaincode install marblesp.tar.gz
```

A successful install command will return the chaincode identifier, similar to the response below:

```
2019-04-22 19:09:04.336 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪001 Installed remotely: response:<status:200 payload:
↪"\nKmarblespv1:57f5353b2568b79cb5384b5a8458519a47186efc4fcadb98280f5eae6d59c1cd\022\nmarblespv1
↪" >
2019-04-22 19:09:04.336 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪002 Chaincode code package identifier:␣
↪marblespv1:57f5353b2568b79cb5384b5a8458519a47186efc4fcadb98280f5eae6d59c1cd
```

3. Now use the CLI as the Org2 admin. Copy and paste the following block of commands as a group and run them all at once:

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

4. Run the following command to install the chaincode on the Org2 peer:

```
peer lifecycle chaincode install marblesp.tar.gz
```

Approve the chaincode definition

Each channel member that wants to use the chaincode needs to approve a chaincode definition for their organization. Since both organizations are going to use the chaincode in this tutorial, we need to approve the chaincode definition for both Org1 and Org2 using the peer lifecycle chaincode approveformyorg command. The chaincode definition also includes the private data collection definition that accompanies the `marbles02_private` sample. We will provide the path to the collections JSON file using the `--collections-config` flag.

Try it yourself

Run the following commands from the `test-network` directory to approve a definition for Org1 and Org2.

1. Use the following command to query your peer for the package ID of the installed chaincode.

```
peer lifecycle chaincode queryinstalled
```

The command will return the same package identifier as the install command. You should see output similar to the following:

```
Installed chaincodes on peer:
Package ID:␣
↪marblespv1:f8c8e06bfc27771028c4bbc3564341887881e29b92a844c66c30bac0ff83966e, Label:␣
↪marblespv1
```

2. Declare the package ID as an environment variable. Paste the package ID of marblespv1 returned by the `peer lifecycle chaincode queryinstalled` into the command below. The package ID may not be the same for all users, so you need to complete this step using the package ID returned from your console.

```
export CC_PACKAGE_
↪ID=marblespv1:f8c8e06bfc27771028c4bbc3564341887881e29b92a844c66c30bac0ff83966e
```

3. Make sure we are running the CLI as Org1. Copy and paste the following block of commands as a group into the peer container and run them all at once:

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

4. Use the following command to approve a definition of the marbles private data chaincode for Org1. This command includes a path to the collection definition file.

```
export ORDERER_CA=${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name␣
↪marblesp --version 1.0 --collections-config ../chaincode/marbles02_private/
↪collections_config.json --signature-policy "OR('Org1MSP.member','Org2MSP.member')" -
↪-init-required --package-id $CC_PACKAGE_ID --sequence 1 --tls true --cafile
↪$ORDERER_CA
```

When the command completes successfully you should see something similar to:

```
2020-01-03 17:26:55.022 EST [chaincodeCmd] ClientWait -> INFO 001 txid␣
↪[06c9e86ca68422661e09c15b8e6c23004710ea280efda4bf54d501e655bafa9b] committed with␣
↪status (VALID) at
```

5. Now use the CLI to switch to Org2. Copy and paste the following block of commands as a group into the peer container and run them all at once.

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

6. You can now approve the chaincode definition for Org2:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name␣
↪marblesp --version 1.0 --collections-config ../chaincode/marbles02_private/
↪collections_config.json --signature-policy "OR('Org1MSP.member','Org2MSP.member')" -
↪-init-required --package-id $CC_PACKAGE_ID --sequence 1 --tls true --cafile
↪$ORDERER_CA
```

Commit the chaincode definition

Once a sufficient number of organizations (in this case, a majority) have approved a chaincode definition, one organization can commit the definition to the channel.

Use the peer lifecycle chaincode commit command to commit the chaincode definition. This command will also deploy the collection definition to the channel.

We are ready to use the chaincode after the chaincode definition has been committed to the channel. Because the marbles private data chaincode contains an initiation function, we need to use the peer chaincode invoke command to invoke `Init()` before we can use other functions in the chaincode.

Try it yourself

1. Run the following commands to commit the definition of the marbles private data chaincode to the channel `mychannel`.

```
export ORDERER_CA=${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
export ORG1_CA=${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.
↪org1.example.com/tls/ca.crt
export ORG2_CA=${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.
↪org2.example.com/tls/ca.crt
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride␣
↪orderer.example.com --channelID mychannel --name marblesp --version 1.0 --sequence␣
↪1 --collections-config ../chaincode/marbles02_private/collections_config.json --
↪signature-policy "OR('Org1MSP.member','Org2MSP.member')" --init-required --tls true␣
↪--cafile $ORDERER_CA --peerAddresses localhost:7051 --tlsRootCertFiles $ORG1_CA --
↪peerAddresses localhost:9051 --tlsRootCertFiles $ORG2_CA
```

When the commit transaction completes successfully you should see something similar to:

```
2020-01-06 16:24:46.104 EST [chaincodeCmd] ClientWait -> INFO 001 txid␣
↪[4a0d0f5da43eb64f7cbfd72ea8a8df18c328fb250cb346077d91166d86d62d46] committed with␣
↪status (VALID) at localhost:9051
2020-01-06 16:24:46.184 EST [chaincodeCmd] ClientWait -> INFO 002 txid␣
↪[4a0d0f5da43eb64f7cbfd72ea8a8df18c328fb250cb346077d91166d86d62d46] committed with␣
↪status (VALID) at localhost:7051
```

2. Use the following command to invoke the `Init` function to initialize the chaincode:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --channelID mychannel --name marblesp --isInit --tls true --cafile $ORDERER_CA -
↪-peerAddresses localhost:7051 --tlsRootCertFiles $ORG1_CA -c '{"Args":["Init"]}'
```

7.4.5 Store private data

Acting as a member of Org1, who is authorized to transact with all of the private data in the marbles private data sample, switch back to an Org1 peer and submit a request to add a marble:

Try it yourself

Copy and paste the following set of commands into your CLI in the *test-network* directory:

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

Invoke the marbles `initMarble` function which creates a marble with private data — name `marble1` owned by `tom` with a color `blue`, size `35` and price of `99`. Recall that private data **price** will be stored separately from the private data **name, owner, color, size**. For this reason, the `initMarble` function calls the `PutPrivateData()` API twice to persist the private data, once for each collection. Also note that the private data is passed using the `--transient` flag. Inputs passed as transient data will not be persisted in the transaction in order to keep the data private. Transient data is passed as binary data and therefore when using CLI it must be base64 encoded. We use an environment variable to capture the base64 encoded value, and use `tr` command to strip off the problematic newline characters that linux base64 command adds.

```
export MARBLE=$(echo -n "{\"name\":\"marble1\",\"color\":\"blue\",\"size\":35,\"owner\
↪":\"tom\",\"price\":99}" | base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪marblesp -c '{"Args":["initMarble"]}' --transient "{\"marble\":\"$MARBLE\"}"
```

You should see results similar to:

```
[chaincodeCmd] chaincodeInvokeOrQuery->INFO 001 Chaincode invoke successful. result:␣
↪status:200
```

7.4.6 Query the private data as an authorized peer

Our collection definition allows all members of Org1 and Org2 to have the `name, color, size, owner` private data in their side database, but only peers in Org1 can have the `price` private data in their side database. As an authorized peer in Org1, we will query both sets of private data.

The first `query` command calls the `readMarble` function which passes `collectionMarbles` as an argument.

```
// ============================================
// readMarble - read a marble from chaincode state
// ============================================

func (t *SimpleChaincode) readMarble(stub shim.ChaincodeStubInterface, args []string)␣
↪pb.Response {
    var name, jsonResp string
    var err error
    if len(args) != 1 {
            return shim.Error("Incorrect number of arguments. Expecting name of the␣
↪marble to query")
```

(continues on next page)

(continued from previous page)

```
    }

    name = args[0]
    valAsbytes, err := stub.GetPrivateData("collectionMarbles", name) //get the␣
↪marble from chaincode state

    if err != nil {
            jsonResp = "{\"Error\":\"Failed to get state for " + name + "\"}"
            return shim.Error(jsonResp)
    } else if valAsbytes == nil {
            jsonResp = "{\"Error\":\"Marble does not exist: " + name + "\"}"
            return shim.Error(jsonResp)
    }

    return shim.Success(valAsbytes)
}
```

The second `query` command calls the `readMarblePrivateDetails` function which passes `collectionMarblePrivateDetails` as an argument.

```
// ===============================================
// readMarblePrivateDetails - read a marble private details from chaincode state
// ===============================================

func (t *SimpleChaincode) readMarblePrivateDetails(stub shim.ChaincodeStubInterface,␣
↪args []string) pb.Response {
    var name, jsonResp string
    var err error

    if len(args) != 1 {
            return shim.Error("Incorrect number of arguments. Expecting name of the␣
↪marble to query")
    }

    name = args[0]
    valAsbytes, err := stub.GetPrivateData("collectionMarblePrivateDetails", name) //
↪get the marble private details from chaincode state

    if err != nil {
            jsonResp = "{\"Error\":\"Failed to get private details for " + name + ":
↪" + err.Error() + "\"}"
            return shim.Error(jsonResp)
    } else if valAsbytes == nil {
            jsonResp = "{\"Error\":\"Marble private details does not exist: " + name␣
↪+ "\"}"
            return shim.Error(jsonResp)
    }
    return shim.Success(valAsbytes)
}
```

Now *Try it yourself*

Query for the `name, color, size and owner` private data of `marble1` as a member of Org1. Note that since queries do not get recorded on the ledger, there is no need to pass the marble name as a transient input.

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarble","marble1"]}'
```

You should see the following result:

```
{"color":"blue","docType":"marble","name":"marble1","owner":"tom","size":35}
```

Query for the `price` private data of `marble1` as a member of Org1.

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails",
↪"marble1"]}'
```

You should see the following result:

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

7.4.7 Query the private data as an unauthorized peer

Now we will switch to a member of Org2. Org2 has the marbles private data `name, color, size, owner` in its side database, but does not store the marbles `price` data. We will query for both sets of private data.

Switch to a peer in Org2

Run the following commands to operate as the Org2 admin and query the Org2 peer.

Try it yourself

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

Query private data Org2 is authorized to

Peers in Org2 should have the first set of marbles private data (`name, color, size and owner`) in their side database and can access it using the `readMarble()` function which is called with the `collectionMarbles` argument.

Try it yourself

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarble","marble1"]}'
```

You should see something similar to the following result:

```
{"docType":"marble","name":"marble1","color":"blue","size":35,"owner":"tom"}
```

Query private data Org2 is not authorized to

Peers in Org2 do not have the marbles `price` private data in their side database. When they try to query for this data, they get back a hash of the key matching the public state but will not have the private state.

Try it yourself

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails",
↪"marble1"]}'
```

You should see a result similar to:

```
Error: endorsement failure during query. response: status:500
message:"{\"Error\":\"Failed to get private details for marble1:
GET_STATE failed: transaction ID:␣
↪d9c437d862de66755076aeebe79e7727791981606ae1cb685642c93f102b03e5:
tx creator does not have read access permission on privatedata in␣
↪chaincodeName:marblesp collectionName: collectionMarblePrivateDetails\"}"
```

Members of Org2 will only be able to see the public hash of the private data.

7.4.8 Purge Private Data

For use cases where private data only needs to be on the ledger until it can be replicated into an off-chain database, it is possible to "purge" the data after a certain set number of blocks, leaving behind only hash of the data that serves as immutable evidence of the transaction.

There may be private data including personal or confidential information, such as the pricing data in our example, that the transacting parties don't want disclosed to other organizations on the channel. Thus, it has a limited lifespan, and can be purged after existing unchanged on the blockchain for a designated number of blocks using the `blockToLive` property in the collection definition.

Our `collectionMarblePrivateDetails` definition has a `blockToLive` property value of three meaning this data will live on the side database for three blocks and then after that it will get purged. Tying all of the pieces together, recall this collection definition `collectionMarblePrivateDetails` is associated with the `price` private data in the `initMarble()` function when it calls the `PutPrivateData()` API and passes the `collectionMarblePrivateDetails` as an argument.

We will step through adding blocks to the chain, and then watch the price information get purged by issuing four new transactions (Create a new marble, followed by three marble transfers) which adds four new blocks to the chain. After the fourth transaction (third marble transfer), we will verify that the price private data is purged.

Try it yourself

Switch back to Org1 using the following commands. Copy and paste the following code block and run it inside your peer container:

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

Open a new terminal window and view the private data logs for this peer by running the following command. Note the highest block number.

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

Back in the peer container, query for the **marble1** price data by running the following command. (A Query does not create a new transaction on the ledger since no data is transacted).

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails",
↪"marble1"]}'
```

You should see results similar to:

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

The `price` data is still in the private data ledger.

Create a new **marble2** by issuing the following command. This transaction creates a new block on the chain.

```
export MARBLE=$(echo -n "{\"name\":\"marble2\",\"color\":\"blue\",\"size\":35,\"owner\
↪":\"tom\",\"price\":99}" | base64 | tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪marblesp -c '{"Args":["initMarble"]}' --transient "{\"marble\":\"$MARBLE\"}"
```

Switch back to the Terminal window and view the private data logs for this peer again. You should see the block height increase by 1.

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

Back in the peer container, query for the **marble1** price data again by running the following command:

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails",
↪"marble1"]}'
```

The private data has not been purged, therefore the results are unchanged from previous query:

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

Transfer marble2 to "joe" by running the following command. This transaction will add a second new block on the chain.

```
export MARBLE_OWNER=$(echo -n "{\"name\":\"marble2\",\"owner\":\"joe\"}" | base64 |␣
↪tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪marblesp -c '{"Args":["transferMarble"]}' --transient "{\"marble_owner\":\"$MARBLE_
↪OWNER\"}"
```

Switch back to the Terminal window and view the private data logs for this peer again. You should see the block height increase by 1.

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

Back in the peer container, query for the marble1 price data by running the following command:

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails",
↪"marble1"]}'
```

You should still be able to see the price private data.

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

Transfer marble2 to "tom" by running the following command. This transaction will create a third new block on the chain.

```
export MARBLE_OWNER=$(echo -n "{\"name\":\"marble2\",\"owner\":\"tom\"}" | base64 |␣
↪tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪marblesp -c '{"Args":["transferMarble"]}' --transient "{\"marble_owner\":\"$MARBLE_
↪OWNER\"}"
```

Switch back to the Terminal window and view the private data logs for this peer again. You should see the block height increase by 1.

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

Back in the peer container, query for the marble1 price data by running the following command:

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails",
↪"marble1"]}'
```

You should still be able to see the price data.

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

Finally, transfer marble2 to "jerry" by running the following command. This transaction will create a fourth new block on the chain. The `price` private data should be purged after this transaction.

```
export MARBLE_OWNER=$(echo -n "{\"name\":\"marble2\",\"owner\":\"jerry\"}" | base64 |␣
↪tr -d \\n)
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪marblesp -c '{"Args":["transferMarble"]}' --transient "{\"marble_owner\":\"$MARBLE_
↪OWNER\"}"
```

Switch back to the Terminal window and view the private data logs for this peer again. You should see the block height increase by 1.

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

Back in the peer container, query for the marble1 price data by running the following command:

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails",
↪"marble1"]}'
```

Because the price data has been purged, you should no longer be able to see it. You should see something similar to:

```
Error: endorsement failure during query. response: status:500
message:"{\"Error\":\"Marble private details does not exist: marble1\"}"
```

7.4.9 Using indexes with private data

Indexes can also be applied to private data collections, by packaging indexes in the `META-INF/statedb/couchdb/collections/<collection_name>/indexes` directory alongside the chaincode. An example index is available here .

For deployment of chaincode to production environments, it is recommended to define any indexes alongside chaincode so that the chaincode and supporting indexes are deployed automatically as a unit, once the chaincode has been

installed on a peer and instantiated on a channel. The associated indexes are automatically deployed upon chaincode instantiation on the channel when the `--collections-config` flag is specified pointing to the location of the collection JSON file.

7.4.10 Additional resources

For additional private data education, a video tutorial has been created.

Note: The video uses the previous lifecycle model to install private data collections with chaincode.

7.5 Using CouchDB

This tutorial will describe the steps required to use the CouchDB as the state database with Hyperledger Fabric. By now, you should be familiar with Fabric concepts and have explored some of the samples and tutorials.

Note: These instructions use the new Fabric chaincode lifecycle introduced in the Fabric v2.0 release. If you would like to use the previous lifecycle model to use indexes with chaincode, visit the v1.4 version of the Using CouchDB.

The tutorial will take you through the following steps:

1. *Enable CouchDB in Hyperledger Fabric*
2. *Create an index*
3. *Add the index to your chaincode folder*
4. *Install and define the Chaincode*
5. *Query the CouchDB State Database*
6. *Use best practices for queries and indexes*
7. *Query the CouchDB State Database With Pagination*
8. *Update an Index*
9. *Delete an Index*

For a deeper dive into CouchDB refer to *CouchDB as the State Database* and for more information on the Fabric ledger refer to the Ledger topic. Follow the tutorial below for details on how to leverage CouchDB in your blockchain network.

Throughout this tutorial, we will use the Marbles sample as our use case to demonstrate how to use CouchDB with Fabric and will deploy Marbles to the Fabric test network. You should have completed the task *Install Samples, Binaries, and Docker Images*.

7.5.1 Why CouchDB?

Fabric supports two types of peer databases. LevelDB is the default state database embedded in the peer node. LevelDB stores chaincode data as simple key-value pairs and only supports key, key range, and composite key queries. CouchDB is an optional, alternate state database that allows you to model data on the ledger as JSON and issue rich queries against data values rather than the keys. CouchDB also allows you to deploy indexes with your chaincode to make queries more efficient and enable you to query large datasets.

In order to leverage the benefits of CouchDB, namely content-based JSON queries, your data must be modeled in JSON format. You must decide whether to use LevelDB or CouchDB before setting up your network. Switching a peer from using LevelDB to CouchDB is not supported due to data compatibility issues. All peers on the network must use the same database type. If you have a mix of JSON and binary data values, you can still use CouchDB, however the binary values can only be queried based on key, key range, and composite key queries.

7.5.2 Enable CouchDB in Hyperledger Fabric

CouchDB runs as a separate database process alongside the peer. There are additional considerations in terms of setup, management, and operations. A Docker image of CouchDB is available and we recommend that it be run on the same server as the peer. You will need to setup one CouchDB container per peer and update each peer container by changing the configuration found in `core.yaml` to point to the CouchDB container. The `core.yaml` file must be located in the directory specified by the environment variable FABRIC_CFG_PATH:

- For Docker deployments, `core.yaml` is pre-configured and located in the peer container `FABRIC_CFG_PATH` folder. However, when using Docker environments, you typically pass environment variables by editing the `docker-compose-couch.yaml` to override the core.yaml
- For native binary deployments, `core.yaml` is included with the release artifact distribution.

Edit the `stateDatabase` section of `core.yaml`. Specify `CouchDB` as the `stateDatabase` and fill in the associated `couchDBConfig` properties. For more information, see CouchDB configuration.

7.5.3 Create an index

Why are indexes important?

Indexes allow a database to be queried without having to examine every row with every query, making them run faster and more efficiently. Normally, indexes are built for frequently occurring query criteria allowing the data to be queried more efficiently. To leverage the major benefit of CouchDB – the ability to perform rich queries against JSON data – indexes are not required, but they are strongly recommended for performance. Also, if sorting is required in a query, CouchDB requires an index of the sorted fields.

Note: Rich queries that do not have an index will work but may throw a warning in the CouchDB log that the index was not found. However, if a rich query includes a sort specification, then an index on that field is required; otherwise, the query will fail and an error will be thrown.

To demonstrate building an index, we will use the data from the Marbles sample. In this example, the Marbles data structure is defined as:

```
type marble struct {
        ObjectType string `json:"docType"` //docType is used to distinguish the␣
↪various types of objects in state database
        Name       string `json:"name"`    //the field tags are needed to keep case␣
↪from bouncing around
        Color      string `json:"color"`
        Size       int    `json:"size"`
        Owner      string `json:"owner"`
}
```

In this structure, the attributes (`docType`, `name`, `color`, `size`, `owner`) define the ledger data associated with the asset. The attribute `docType` is a pattern used in the chaincode to differentiate different data types that may need to be queried separately. When using CouchDB, it recommended to include this `docType` attribute to distinguish each

type of document in the chaincode namespace. (Each chaincode is represented as its own CouchDB database, that is, each chaincode has its own namespace for keys.)

With respect to the Marbles data structure, `docType` is used to identify that this document/asset is a marble asset. Potentially there could be other documents/assets in the chaincode database. The documents in the database are searchable against all of these attribute values.

When defining an index for use in chaincode queries, each one must be defined in its own text file with the extension **.json* and the index definition must be formatted in the CouchDB index JSON format.

To define an index, three pieces of information are required:

- *fields*: these are the frequently queried fields
- *name*: name of the index
- *type*: always json in this context

For example, a simple index named `foo-index` for a field named `foo`.

```
{
    "index": {
        "fields": ["foo"]
    },
    "name" : "foo-index",
    "type" : "json"
}
```

Optionally the design document attribute `ddoc` can be specified on the index definition. A design document is CouchDB construct designed to contain indexes. Indexes can be grouped into design documents for efficiency but CouchDB recommends one index per design document.

Tip: When defining an index it is a good practice to include the `ddoc` attribute and value along with the index name. It is important to include this attribute to ensure that you can update the index later if needed. Also it gives you the ability to explicitly specify which index to use on a query.

Here is another example of an index definition from the Marbles sample with the index name `indexOwner` using multiple fields `docType` and `owner` and includes the `ddoc` attribute:

```
{
  "index":{
      "fields":["docType","owner"] // Names of the fields to be queried
  },
  "ddoc":"indexOwnerDoc", // (optional) Name of the design document in which the␣
↪index will be created.
  "name":"indexOwner",
  "type":"json"
}
```

In the example above, if the design document `indexOwnerDoc` does not already exist, it is automatically created when the index is deployed. An index can be constructed with one or more attributes specified in the list of fields and any combination of attributes can be specified. An attribute can exist in multiple indexes for the same docType. In the following example, `index1` only includes the attribute `owner`, `index2` includes the attributes `owner and color` and `index3` includes the attributes `owner, color and size`. Also, notice each index definition has its own `ddoc` value, following the CouchDB recommended practice.

```
{
  "index":{
```

(continues on next page)

(continued from previous page)

```
      "fields":["owner"] // Names of the fields to be queried
  },
  "ddoc":"index1Doc", // (optional) Name of the design document in which the index␣
↪will be created.
  "name":"index1",
  "type":"json"
}

{
  "index":{
      "fields":["owner", "color"] // Names of the fields to be queried
  },
  "ddoc":"index2Doc", // (optional) Name of the design document in which the index␣
↪will be created.
  "name":"index2",
  "type":"json"
}

{
  "index":{
      "fields":["owner", "color", "size"] // Names of the fields to be queried
  },
  "ddoc":"index3Doc", // (optional) Name of the design document in which the index␣
↪will be created.
  "name":"index3",
  "type":"json"
}
```

In general, you should model index fields to match the fields that will be used in query filters and sorts. For more details on building an index in JSON format refer to the CouchDB documentation.

A final word on indexing, Fabric takes care of indexing the documents in the database using a pattern called `index warming`. CouchDB does not typically index new or updated documents until the next query. Fabric ensures that indexes stay 'warm' by requesting an index update after every block of data is committed. This ensures queries are fast because they do not have to index documents before running the query. This process keeps the index current and refreshed every time new records are added to the state database.

7.5.4 Add the index to your chaincode folder

Once you finalize an index, you need to package it with your chaincode for deployment by placing it in the appropriate metadata folder. You can install the chaincode using the *peer lifecycle chaincode* command. The JSON index files must be located under the path `META-INF/statedb/couchdb/indexes` which is located inside the directory where the chaincode resides.

The Marbles sample below illustrates how the index is packaged with the chaincode.

This sample includes one index named indexOwnerDoc:

```
{"index":{"fields":["docType","owner"]},"ddoc":"indexOwnerDoc", "name":"indexOwner",
↪"type":"json"}
```

Start the network

Try it yourself

We will bring up the Fabric test network and use it to deploy the marbles chainocde. Use the following command to navigate to the *test-network* directory in the Fabric samples:

```
cd fabric-samples/test-network
```

For this tutorial, we want to operate from a known initial state. The following command will kill any active or stale Docker containers and remove previously generated artifacts:

```
./network.sh down
```

If you have not run through the tutorial before, you will need to vendor the chaincode dependencies before we can deploy it to the network. Run the following commands:

```
cd ../chaincode/marbles02/go
GO111MODULE=on go mod vendor
cd ../../../test-network
```

From the *test-network* directory, deploy the test network with CouchDB with the following command:

```
./network.sh up createChannel -s couchdb
```

This will create two fabric peer nodes that use CouchDB as the state database. It will also create one ordering node and a single channel named `mychannel`.

7.5.5 Install and define the Chaincode

Client applications interact with the blockchain ledger through chaincode. Therefore we need to install a chaincode on every peer that will execute and endorse our transactions. However, before we can interact with our chaincode, the members of the channel need to agree on a chaincode definition that establishes chaincode governance. In the previous section, we demonstrated how to add the index to the chaincode folder so that the index is deployed with the chaincode.

The chaincode needs to be packaged before it can be installed on our peers. We can use the peer lifecycle chaincode package command to package the marbles chaincode.

Try it yourself

1. After you start the test network, copy and paste the following environment variables in your CLI to interact with the network as the Org1 admin. Make sure that you are in the *test-network* directory.

```
export PATH=${PWD}/../bin:${PWD}:$PATH
export FABRIC_CFG_PATH=${PWD}/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

2. Use the following command to package the marbles chaincode:

```
peer lifecycle chaincode package marbles.tar.gz --path ../chaincode/marbles02/go --
↪lang golang --label marbles_1
```

This command will create a chaincode package named marbles.tar.gz.

3. Use the following command to install the chaincode package onto the peer `peer0.org1.example.com`:

```
peer lifecycle chaincode install marbles.tar.gz
```

A successful install command will return the chaincode identifier, similar to the response below:

```
2019-04-22 18:47:38.312 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪001 Installed remotely: response:<status:200 payload:"\nJmarbles_
↪1:0907c1f3d3574afca69946e1b6132691d58c2f5c5703df7fc3b692861e92ecd3\022\tmarbles_1" >
2019-04-22 18:47:38.312 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪002 Chaincode code package identifier: marbles_
↪1:0907c1f3d3574afca69946e1b6132691d58c2f5c5703df7fc3b692861e92ecd3
```

After installing the chaincode on `peer0.org1.example.com`, we need to approve a chaincode definition for Org1.

4. Use the following command to query your peer for the package ID of the installed chaincode.

```
peer lifecycle chaincode queryinstalled
```

The command will return the same package identifier as the install command. You should see output similar to the following:

```
Installed chaincodes on peer:
Package ID: marbles_
↪1:60ec9430b221140a45b96b4927d1c3af736c1451f8d432e2a869bdbf417f9787, Label: marbles_1
```

5. Declare the package ID as an environment variable. Paste the package ID of marbles_1 returned by the `peer lifecycle chaincode queryinstalled` command into the command below. The package ID may not be the same for all users, so you need to complete this step using the package ID returned from your console.

```
export CC_PACKAGE_ID=marbles_
↪1:60ec9430b221140a45b96b4927d1c3af736c1451f8d432e2a869bdbf417f9787
```

6. Use the following command to approve a definition of the marbles chaincode for Org1.

```
export ORDERER_CA=${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name marbles␣
↪--version 1.0 --signature-policy "OR('Org1MSP.member','Org2MSP.member')" --init-
↪required --package-id $CC_PACKAGE_ID --sequence 1 --tls true --cafile $ORDERER_CA
```

When the command completes successfully you should see something similar to :

```
2020-01-07 16:24:20.886 EST [chaincodeCmd] ClientWait -> INFO 001 txid␣
↪[560cb830efa1272c85d2f41a473483a25f3b12715d55e22a69d55abc46581415] committed with␣
↪status (VALID) at
```

We need a majority of organizations to approve a chaincode definition before it can be committed to the channel. This implies that we need Org2 to approve the chaincode definition as well. Because we do not need Org2 to endorse the chaincode and did not install the package on any Org2 peers, we do not need to provide a packageID as part of the chaincode definition.

7. Use the CLI to operate as the Org2 admin. Copy and paste the following block of commands as a group into the peer container and run them all at once.

```
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.
↪com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=localhost:9051
```

8. Use the following command to approve the chaincode definition for Org2:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name marbles␣
↪--version 1.0 --signature-policy "OR('Org1MSP.member','Org2MSP.member')" --init-
↪required --sequence 1 --tls true --cafile $ORDERER_CA
```

9. We can now use the peer lifecycle chaincode commit command to commit the chaincode definition to the channel:

```
export ORDERER_CA=${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
export ORG1_CA=${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.
↪org1.example.com/tls/ca.crt
export ORG2_CA=${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.
↪org2.example.com/tls/ca.crt
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride␣
↪orderer.example.com --channelID mychannel --name marbles --version 1.0 --sequence 1␣
↪--signature-policy "OR('Org1MSP.member','Org2MSP.member')" --init-required --tls␣
↪true --cafile $ORDERER_CA --peerAddresses localhost:7051 --tlsRootCertFiles $ORG1_
↪CA --peerAddresses localhost:9051 --tlsRootCertFiles $ORG2_CA
```

When the commit transaction completes successfully you should see something similar to:

```
2019-04-22 18:57:34.274 UTC [chaincodeCmd] ClientWait -> INFO 001 txid␣
↪[3da8b0bb8e128b5e1b6e4884359b5583dff823fce2624f975c69df6bce614614] committed with␣
↪status (VALID) at peer0.org2.example.com:9051
2019-04-22 18:57:34.709 UTC [chaincodeCmd] ClientWait -> INFO 002 txid␣
↪[3da8b0bb8e128b5e1b6e4884359b5583dff823fce2624f975c69df6bce614614] committed with␣
↪status (VALID) at peer0.org1.example.com:7051
```

10. Because the marbles chaincode contains an initialization function, we need to use the peer chaincode invoke command to invoke `Init()` before we can use other functions in the chaincode:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --channelID mychannel --name marbles --isInit --tls true --cafile $ORDERER_CA --
↪peerAddresses localhost:7051 --tlsRootCertFiles $ORG1_CA -c '{"Args":["Init"]}'
```

Verify index was deployed

Indexes will be deployed to each peer's CouchDB state database once the chaincode has been installed on the peer and deployed to the channel. You can verify that the CouchDB index was created successfully by examining the peer log in the Docker container.

Try it yourself

To view the logs in the peer Docker container, open a new Terminal window and run the following command to grep for message confirmation that the index was created.

```
docker logs peer0.org1.example.com  2>&1 | grep "CouchDB index"
```

You should see a result that looks like the following:

```
[couchdb] CreateIndex -> INFO 0be Created CouchDB index [indexOwner] in state␣
↪database [mychannel_marbles] using design document [_design/indexOwnerDoc]
```

Note: If you installed Marbles on a different peer than `peer0.org1.example.com`, you may need to replace it with the name of a different peer where Marbles was installed.

7.5.6 Query the CouchDB State Database

Now that the index has been defined in the JSON file and deployed alongside the chaincode, chaincode functions can execute JSON queries against the CouchDB state database, and thereby peer commands can invoke the chaincode functions.

Specifying an index name on a query is optional. If not specified, and an index already exists for the fields being queried, the existing index will be automatically used.

Tip: It is a good practice to explicitly include an index name on a query using the `use_index` keyword. Without it, CouchDB may pick a less optimal index. Also CouchDB may not use an index at all and you may not realize it, at the low volumes during testing. Only upon higher volumes you may realize slow performance because CouchDB is not using an index and you assumed it was.

Build the query in chaincode

You can perform complex rich queries against the data on the ledger using queries defined within your chaincode. The marbles02 sample includes two rich query functions:

- **queryMarbles** –

  Example of an **ad hoc rich query**. This is a query where a (selector) string can be passed into the function. This query would be useful to client applications that need to dynamically build their own selectors at runtime. For more information on selectors refer to CouchDB selector syntax.

- **queryMarblesByOwner** –

  Example of a **parameterized query** where the query logic is baked into the chaincode. In this case the function accepts a single argument, the marble owner. It then queries the state database for JSON documents matching the docType of "marble" and the owner id using the JSON query syntax.

Run the query using the peer command

In absence of a client application, we can use the peer command to test the queries defined in the chaincode. We will customize the peer chaincode query command to use the Marbles index `indexOwner` and query for all marbles owned by "tom" using the `queryMarbles` function.

Try it yourself

Before querying the database, we should add some data. Run the following command as Org1 to create a marble owned by "tom":

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪marbles -c '{"Args":["initMarble","marble1","blue","35","tom"]}'
```

After an index has been deployed when the chaincode is initialized, it will automatically be utilized by chaincode queries. CouchDB can determine which index to use based on the fields being queried. If an index exists for the query criteria it will be used. However the recommended approach is to specify the `use_index` keyword on the query. The peer command below is an example of how to specify the index explicitly in the selector syntax by including the `use_index` keyword:

```
// Rich Query with index name explicitly specified:
peer chaincode query -C mychannel -n marbles -c '{"Args":["queryMarbles", "{\
↪"selector\":{\"docType\":\"marble\",\"owner\":\"tom\"}, \"use_index\":[\"_design/
↪indexOwnerDoc\", \"indexOwner\"]}"]}'
```

Delving into the query command above, there are three arguments of interest:

- `queryMarbles`

  Name of the function in the Marbles chaincode. Notice a shim `shim.ChaincodeStubInterface` is used to access and modify the ledger. The `getQueryResultForQueryString()` passes the queryString to the shim API `getQueryResult()`.

```
func (t *SimpleChaincode) queryMarbles(stub shim.ChaincodeStubInterface, args␣
↪[]string) pb.Response {

        //   0
        // "queryString"
         if len(args) < 1 {
                 return shim.Error("Incorrect number of arguments. Expecting 1")
         }

         queryString := args[0]

         queryResults, err := getQueryResultForQueryString(stub, queryString)
         if err != nil {
               return shim.Error(err.Error())
         }
         return shim.Success(queryResults)
}
```

- `{"selector":{"docType":"marble","owner":"tom"}`

  This is an example of an **ad hoc selector** string which finds all documents of type `marble` where the `owner` attribute has a value of `tom`.

- `"use_index":["_design/indexOwnerDoc", "indexOwner"]`

  Specifies both the design doc name `indexOwnerDoc` and index name `indexOwner`. In this example the selector query explicitly includes the index name, specified by using the `use_index` keyword. Recalling the index definition above *Create an index*, it contains a design doc, `"ddoc":"indexOwnerDoc"`. With CouchDB, if you plan to explicitly include the index name on the query, then the index definition must include the `ddoc` value, so it can be referenced with the `use_index` keyword.

The query runs successfully and the index is leveraged with the following results:

```
Query Result: [{"Key":"marble1", "Record":{"color":"blue","docType":"marble","name":
↪"marble1","owner":"tom","size":35}}]
```

7.5.7 Use best practices for queries and indexes

Queries that use indexes will complete faster, without having to scan the full database in couchDB. Understanding indexes will allow you to write your queries for better performance and help your application handle larger amounts of data or blocks on your network.

It is also important to plan the indexes you install with your chaincode. You should install only a few indexes per chaincode that support most of your queries. Adding too many indexes, or using an excessive number of fields in an index, will degrade the performance of your network. This is because the indexes are updated after each block is committed. The more indexes need to be updated through "index warming", the longer it will take for transactions to complete.

The examples in this section will help demonstrate how queries use indexes and what type of queries will have the best performance. Remember the following when writing your queries:

- All fields in the index must also be in the selector or sort sections of your query for the index to be used.
- More complex queries will have a lower performance and will be less likely to use an index.
- You should try to avoid operators that will result in a full table scan or a full index scan such as `$or`, `$in` and `$regex`.

In the previous section of this tutorial, you issued the following query against the marbles chaincode:

```
// Example one: query fully supported by the index
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles", "{\
↪"selector\":{\"docType\":\"marble\",\"owner\":\"tom\"}, \"use_index\":[\
↪"indexOwnerDoc\", \"indexOwner\"]}"]}'
```

The marbles chaincode was installed with the `indexOwnerDoc` index:

```
{"index":{"fields":["docType","owner"]},"ddoc":"indexOwnerDoc", "name":"indexOwner",
↪"type":"json"}
```

Notice that both the fields in the query, `docType` and `owner`, are included in the index, making it a fully supported query. As a result this query will be able to use the data in the index, without having to search the full database. Fully supported queries such as this one will return faster than other queries from your chaincode.

If you add extra fields to the query above, it will still use the index. However, the query will additionally have to scan the indexed data for the extra fields, resulting in a longer response time. As an example, the query below will still use the index, but will take a longer time to return than the previous example.

```
// Example two: query fully supported by the index with additional data
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles", "{\
↪"selector\":{\"docType\":\"marble\",\"owner\":\"tom\",\"color\":\"red\"}, \"use_
↪index\":[\"/indexOwnerDoc\", \"indexOwner\"]}"]}'
```

A query that does not include all fields in the index will have to scan the full database instead. For example, the query below searches for the owner, without specifying the the type of item owned. Since the ownerIndexDoc contains both the `owner` and `docType` fields, this query will not be able to use the index.

```
// Example three: query not supported by the index
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles", "{\
↪"selector\":{\"owner\":\"tom\"}, \"use_index\":[\"indexOwnerDoc\", \"indexOwner\"]}
↪"]}'
```

In general, more complex queries will have a longer response time, and have a lower chance of being supported by an index. Operators such as `$or`, `$in`, and `$regex` will often cause the query to scan the full index or not use the index at all.

As an example, the query below contains an `$or` term that will search for every marble and every item owned by tom.

```
// Example four: query with $or supported by the index
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles", "{\
↪"selector\":{"\$or\":[{\"docType\:\"marble\"},{\"owner\":\"tom\"}]}, \"use_index\
↪":[\"indexOwnerDoc\", \"indexOwner\"]}"]}'
```

This query will still use the index because it searches for fields that are included in `indexOwnerDoc`. However, the `$or` condition in the query requires a scan of all the items in the index, resulting in a longer response time.

Below is an example of a complex query that is not supported by the index.

```
// Example five: Query with $or not supported by the index
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles", "{\
↪"selector\":{"\$or\":[{\"docType\":\"marble\",\"owner\":\"tom\"},{"\color\":
↪"\yellow\"}]}, \"use_index\":[\"indexOwnerDoc\", \"indexOwner\"]}"]}'
```

The query searches for all marbles owned by tom or any other items that are yellow. This query will not use the index because it will need to search the entire table to meet the `$or` condition. Depending the amount of data on your ledger, this query will take a long time to respond or may timeout.

While it is important to follow best practices with your queries, using indexes is not a solution for collecting large amounts of data. The blockchain data structure is optimized to validate and confirm transactions and is not suited for data analytics or reporting. If you want to build a dashboard as part of your application or analyze the data from your network, the best practice is to query an off chain database that replicates the data from your peers. This will allow you to understand the data on the blockchain without degrading the performance of your network or disrupting transactions.

You can use block or chaincode events from your application to write transaction data to an off-chain database or analytics engine. For each block received, the block listener application would iterate through the block transactions and build a data store using the key/value writes from each valid transaction's `rwset`. The *Peer channel-based event services* provide replayable events to ensure the integrity of downstream data stores. For an example of how you can use an event listener to write data to an external database, visit the Off chain data sample in the Fabric Samples.

7.5.8 Query the CouchDB State Database With Pagination

When large result sets are returned by CouchDB queries, a set of APIs is available which can be called by chaincode to paginate the list of results. Pagination provides a mechanism to partition the result set by specifying a `pagesize` and a start point – a `bookmark` which indicates where to begin the result set. The client application iteratively invokes the chaincode that executes the query until no more results are returned. For more information refer to this topic on pagination with CouchDB.

We will use the Marbles sample function `queryMarblesWithPagination` to demonstrate how pagination can be implemented in chaincode and the client application.

- **queryMarblesWithPagination** –

  Example of an **ad hoc rich query with pagination**. This is a query where a (selector) string can be passed into the function similar to the above example. In this case, a `pageSize` is also included with the query as well as a `bookmark`.

In order to demonstrate pagination, more data is required. This example assumes that you have already added marble1 from above. Run the following commands in the peer container to create four more marbles owned by "tom", to create a total of five marbles owned by "tom":

Try it yourself

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.
↪com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile  ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪marbles -c '{"Args":["initMarble","marble2","yellow","35","tom"]}'
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile  ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪marbles -c '{"Args":["initMarble","marble3","green","20","tom"]}'
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile  ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪marbles -c '{"Args":["initMarble","marble4","purple","20","tom"]}'
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls --cafile  ${PWD}/organizations/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n␣
↪marbles -c '{"Args":["initMarble","marble5","blue","40","tom"]}'
```

In addition to the arguments for the query in the previous example, queryMarblesWithPagination adds `pagesize` and `bookmark`. `PageSize` specifies the number of records to return per query. The `bookmark` is an "anchor" telling couchDB where to begin the page. (Each page of results returns a unique bookmark.)

- `queryMarblesWithPagination`

  Name of the function in the Marbles chaincode. Notice a shim `shim.ChaincodeStubInterface` is used to access and modify the ledger. The `getQueryResultForQueryStringWithPagination()` passes the queryString along with the pagesize and bookmark to the shim API `GetQueryResultWithPagination()`.

```
func (t *SimpleChaincode) queryMarblesWithPagination(stub shim.ChaincodeStubInterface,
↪ args []string) pb.Response {

      //   0
      // "queryString"
      if len(args) < 3 {
              return shim.Error("Incorrect number of arguments. Expecting 3")
      }

      queryString := args[0]
      //return type of ParseInt is int64
      pageSize, err := strconv.ParseInt(args[1], 10, 32)
      if err != nil {
              return shim.Error(err.Error())
      }
      bookmark := args[2]

      queryResults, err := getQueryResultForQueryStringWithPagination(stub,␣
↪queryString, int32(pageSize), bookmark)
      if err != nil {
              return shim.Error(err.Error())
      }
      return shim.Success(queryResults)
}
```

The following example is a peer command which calls queryMarblesWithPagination with a pageSize of `3` and no bookmark specified.

Tip: When no bookmark is specified, the query starts with the "first" page of records.

Try it yourself

```
// Rich Query with index name explicitly specified and a page size of 3:
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":[
↪"queryMarblesWithPagination", "{\"selector\":{\"docType\":\"marble\",\"owner\":\
↪"tom\"}, \"use_index\":[\"_design/indexOwnerDoc\", \"indexOwner\"]}","3",""]}'
```

The following response is received (carriage returns added for clarity), three of the five marbles are returned because the `pagsize` was set to `3`:

```
[{"Key":"marble1", "Record":{"color":"blue","docType":"marble","name":"marble1","owner
↪":"tom","size":35}},
 {"Key":"marble2", "Record":{"color":"yellow","docType":"marble","name":"marble2",
↪"owner":"tom","size":35}},
 {"Key":"marble3", "Record":{"color":"green","docType":"marble","name":"marble3",
↪"owner":"tom","size":20}}]
```

(continues on next page)

(continued from previous page)

```
[{"ResponseMetadata":{"RecordsCount":"3",
"Bookmark":
↪"g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqz5yYWJeWkGoOkOWDSOSANIFk2iCyIyVySn5uVBQAGEhRz
↪"}}]
```

Note: Bookmarks are uniquely generated by CouchDB for each query and represent a placeholder in the result set. Pass the returned bookmark on the subsequent iteration of the query to retrieve the next set of results.

The following is a peer command to call queryMarblesWithPagination with a pageSize of `3`. Notice this time, the query includes the bookmark returned from the previous query.

Try it yourself

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":[
↪"queryMarblesWithPagination", "{\"selector\":{\"docType\":\"marble\",\"owner\":\
↪"tom\"}, \"use_index\":[\"_design/indexOwnerDoc\", \"indexOwner\"]}","3",
↪"g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqz5yYWJeWkGoOkOWDSOSANIFk2iCyIyVySn5uVBQAGEhRz
↪"]}'
```

The following response is received (carriage returns added for clarity). The last two records are retrieved:

```
[{"Key":"marble4", "Record":{"color":"purple","docType":"marble","name":"marble4",
↪"owner":"tom","size":20}},
 {"Key":"marble5", "Record":{"color":"blue","docType":"marble","name":"marble5","owner
↪":"tom","size":40}}]
[{"ResponseMetadata":{"RecordsCount":"2",
"Bookmark":
↪"g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqz5yYWJeWkmoKkOWDSOSANIFk2iCyIyVySn5uVBQAGYhR1
↪"}}]
```

The final command is a peer command to call queryMarblesWithPagination with a pageSize of `3` and with the bookmark from the previous query.

Try it yourself

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":[
↪"queryMarblesWithPagination", "{\"selector\":{\"docType\":\"marble\",\"owner\":\
↪"tom\"}, \"use_index\":[\"_design/indexOwnerDoc\", \"indexOwner\"]}","3",
↪"g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqz5yYWJeWkmoKkOWDSOSANIFk2iCyIyVySn5uVBQAGYhR1
↪"]}'
```

The following response is received (carriage returns added for clarity). No records are returned, indicating that all pages have been retrieved:

```
[]
[{"ResponseMetadata":{"RecordsCount":"0",
"Bookmark":
↪"g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqz5yYWJeWkmoKkOWDSOSANIFk2iCyIyVySn5uVBQAGYhR1
↪"}}]
```

For an example of how a client application can iterate over the result sets using pagination, search for the `getQueryResultForQueryStringWithPagination` function in the Marbles sample.

7.5.9 Update an Index

It may be necessary to update an index over time. The same index may exist in subsequent versions of the chaincode that gets installed. In order for an index to be updated, the original index definition must have included the design document `ddoc` attribute and an index name. To update an index definition, use the same index name but alter the index definition. Simply edit the index JSON file and add or remove fields from the index. Fabric only supports the index type JSON. Changing the index type is not supported. The updated index definition gets redeployed to the peer's state database when the chaincode definition is committed to the channel. Changes to the index name or `ddoc` attributes will result in a new index being created and the original index remains unchanged in CouchDB until it is removed.

Note: If the state database has a significant volume of data, it will take some time for the index to be re-built, during which time chaincode invokes that issue queries may fail or timeout.

Iterating on your index definition

If you have access to your peer's CouchDB state database in a development environment, you can iteratively test various indexes in support of your chaincode queries. Any changes to chaincode though would require redeployment. Use the CouchDB Fauxton interface or a command line curl utility to create and update indexes.

Note: The Fauxton interface is a web UI for the creation, update, and deployment of indexes to CouchDB. If you want to try out this interface, there is an example of the format of the Fauxton version of the index in Marbles sample. If you have deployed the test network with CouchDB, the Fauxton interface can be loaded by opening a browser and navigating to `http://localhost:5984/_utils`.

Alternatively, if you prefer not use the Fauxton UI, the following is an example of a curl command which can be used to create the index on the database `mychannel_marbles`:

```
// Index for docType, owner.
// Example curl command line to define index in the CouchDB channel_chaincode database
 curl -i -X POST -H "Content-Type: application/json" -d
        "{\"index\":{\"fields\":[\"docType\",\"owner\"]},
          \"name\":\"indexOwner\",
          \"ddoc\":\"indexOwnerDoc\",
          \"type\":\"json\"}" http://hostname:port/mychannel_marbles/_index
```

Note: If you are using the test network configured with CouchDB, replace hostname:port with `localhost:5984`.

7.5.10 Delete an Index

Index deletion is not managed by Fabric tooling. If you need to delete an index, manually issue a curl command against the database or delete it using the Fauxton interface.

The format of the curl command to delete an index would be:

```
curl -X DELETE http://localhost:5984/{database_name}/_index/{design_doc}/json/{index_
↪name} -H  "accept: */*" -H  "Host: localhost:5984"
```

To delete the index used in this tutorial, the curl command would be:

```
curl -X DELETE http://localhost:5984/mychannel_marbles/_index/indexOwnerDoc/json/
↪indexOwner -H  "accept: */*" -H  "Host: localhost:5984"
```

7.6 Adding an Org to a Channel

Note: Ensure that you have downloaded the appropriate images and binaries as outlined in *Install Samples, Binaries, and Docker Images* and *Prerequisites* that conform to the version of this documentation (which can be found at the bottom of the table of contents to the left).

This tutorial extends the Fabric test network by adding a new organization – Org3 – to an application channel.

While we will focus on adding a new organization to the channel, you can use a similar process to make other channel configuration updates (updating modification policies or altering batch size, for example). To learn more about the process and possibilities of channel config updates in general, check out *Updating a channel configuration*). It's also worth noting that channel configuration updates like the one demonstrated here will usually be the responsibility of an organization admin (rather than a chaincode or application developer).

7.6.1 Setup the Environment

We will be operating from the root of the `test-network` subdirectory within your local clone of `fabric-samples`. Change into that directory now.

```
cd fabric-samples/test-network
```

First, use the `network.sh` script to tidy up. This command will kill any active or stale Docker containers and remove previously generated artifacts. It is by no means **necessary** to bring down a Fabric network in order to perform channel configuration update tasks. However, for the sake of this tutorial, we want to operate from a known initial state. Therefore let's run the following command to clean up any previous environments:

```
./network.sh down
```

You can now use the script to bring up the test network with one channel named `mychannel`:

```
./network.sh up createChannel
```

If the command was successful, you can see the following message printed in your logs:

```
========= Channel successfully joined ===========
```

Now that you have a clean version of the test network running on your machine, we can start the process of adding a new org to the channel we created. First, we are going use a script to add Org3 to the channel to confirm that the process works. Then, we will go through the step by step process of adding Org3 by updating the channel configuration.

7.6.2 Bring Org3 into the Channel with the Script

You should be in the `test-network` directory. To use the script, simply issue the following commands:

```
cd addOrg3
./addOrg3.sh up
```

The output here is well worth reading. You'll see the Org3 crypto material being generated, the Org3 organization definition being created, and then the channel configuration being updated, signed, and then submitted to the channel.

If everything goes well, you'll get this message:

```
========= Finished adding Org3 to your test network! =========
```

Now that we have confirmed we can add Org3 to our channel, we can go through the steps to update the channel configuration that the script completed behind the scenes.

7.6.3 Bring Org3 into the Channel Manually

If you just used the `addOrg3.sh` script, you'll need to bring your network down. The following command will bring down all running components and remove the crypto material for all organizations:

```
./addOrg3.sh down
```

After the network is brought down, bring it back up again:

```
cd ..
./network.sh up createChannel
```

This will bring your network back to the same state it was in before you executed the `addOrg3.sh` script.

Now we're ready to add Org3 to the channel manually. As a first step, we'll need to generate Org3's crypto material.

7.6.4 Generate the Org3 Crypto Material

In another terminal, change into the `addOrg3` subdirectory from `test-network`.

```
cd addOrg3
```

First, we are going to create the certificates and keys for the Org3 peer, along with an application and admin user. Because we are updating an example channel, we are going to use the cryptogen tool instead of using a Certificate Authority. The following command uses cryptogen to read the `org3-crypto.yaml` file and generate the Org3 crypto material in a new `org3.example.com` folder:

```
../../bin/cryptogen generate --config=org3-crypto.yaml --output="../organizations"
```

You can find the generated Org3 crypto material alongside the certificates and keys for Org1 and Org2 in the `test-network/organizations/peerOrganizations` directory.

Once we have created the Org3 crypto material, we can use the configtxgen tool to print out the Org3 organization definition. We will preface the command by telling the tool to look in the current directory for the `configtx.yaml` file that it needs to ingest.

```
export FABRIC_CFG_PATH=$PWD
../../bin/configtxgen -printOrg Org3MSP > ../organizations/peerOrganizations/org3.
↪example.com/org3.json
```

The above command creates a JSON file – `org3.json` – and writes it to the `test-network/organizations/peerOrganizations/org3.example.com` folder. The organization definition contains the policy definitions for Org3, as well as three important certificates encoded in base64 format:

- a CA root cert, used to establish the organizations root of trust
- a TLS root cert, used by the gossip protocol to identify Org3 for block dissemination and service discovery

- The admin user certificate (which will be needed to act as the admin of Org3 later on)

We will add Org3 to the channel by appending this organization definition to the channel configuration.

7.6.5 Bring up Org3 components

After we have created the Org3 certificate material, we can now bring up the Org3 peer. From the `addOrg3` directory, issue the following command:

```
docker-compose -f docker/docker-compose-org3.yaml up -d
```

If the command is successful, you will see the the creation of the Org3 peer and an instance of the Fabric tools container named Org3CLI:

```
Creating peer0.org3.example.com ... done
Creating Org3cli                ... done
```

This Docker Compose file has been configured to bridge across our initial network, so that the Org3 peer and Org3CLI resolve with the existing peers and ordering node of the test network. We will use the Org3CLI container to communicate with the network and issue the peer commands that will add Org3 to the channel.

7.6.6 Prepare the CLI Environment

The update process makes use of the configuration translator tool – configtxlator. This tool provides a stateless REST API independent of the SDK. Additionally it provides a CLI tool that can be used to simplify configuration tasks in Fabric networks. The tool allows for the easy conversion between different equivalent data representations/formats (in this case, between protobufs and JSON). Additionally, the tool can compute a configuration update transaction based on the differences between two channel configurations.

Use the following command to exec into the Org3CLI container:

```
docker exec -it Org3cli bash
```

This container has been mounted with the `organizations` folder, giving us access to the crypto material and TLS certificates for all organizations and the Orderer Org. We can use environment variables to operate the Org3CLI container as the admin of Org1, Org2, or Org3. First, we need to set the environment variables for the orderer TLS certificate and the channel name:

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/organizations/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪example.com-cert.pem
export CHANNEL_NAME=mychannel
```

Check to make sure the variables have been properly set:

```
echo $ORDERER_CA && echo $CHANNEL_NAME
```

Note: If for any reason you need to restart the Org3CLI container, you will also need to re-export the two environment variables – `ORDERER_CA` and `CHANNEL_NAME`.

7.6.7 Fetch the Configuration

Now we have the Org3CLI container with our two key environment variables – `ORDERER_CA` and `CHANNEL_NAME` exported. Let's go fetch the most recent config block for the channel – `mychannel`.

The reason why we have to pull the latest version of the config is because channel config elements are versioned. Versioning is important for several reasons. It prevents config changes from being repeated or replayed (for instance, reverting to a channel config with old CRLs would represent a security risk). Also it helps ensure concurrency (if you want to remove an Org from your channel, for example, after a new Org has been added, versioning will help prevent you from removing both Orgs, instead of just the Org you want to remove).

Because Org3 is not yet a member of the channel, we need to operate as the admin of another organization to fetch the channel config. Because Org1 is a member of the channel, the Org1 admin has permission to fetch the channel config from the ordering service. Issue the following commands to operate as the Org1 admin.

```
# you can issue all of these commands at once

export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/
↪ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪organizations/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
```

We can now issue the command to fetch the latest config block:

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CHANNEL_
↪NAME --tls --cafile $ORDERER_CA
```

This command saves the binary protobuf channel configuration block to `config_block.pb`. Note that the choice of name and file extension is arbitrary. However, following a convention which identifies both the type of object being represented and its encoding (protobuf or JSON) is recommended.

When you issued the `peer channel fetch` command, the following output is displayed in your logs:

```
2017-11-07 17:17:57.383 UTC [channelCmd] readBlock -> DEBU 011 Received block: 2
```

This is telling us that the most recent configuration block for `mychannel` is actually block 2, **NOT** the genesis block. By default, the `peer channel fetch config` command returns the most **recent** configuration block for the targeted channel, which in this case is the third block. This is because the test network script, `network.sh`, defined anchor peers for our two organizations – `Org1` and `Org2` – in two separate channel update transactions. As a result, we have the following configuration sequence:

- block 0: genesis block
- block 1: Org1 anchor peer update
- block 2: Org2 anchor peer update

7.6.8 Convert the Configuration to JSON and Trim It Down

Now we will make use of the `configtxlator` tool to decode this channel configuration block into JSON format (which can be read and modified by humans). We also must strip away all of the headers, metadata, creator signatures, and so on that are irrelevant to the change we want to make. We accomplish this by means of the `jq` tool:

```
configtxlator proto_decode --input config_block.pb --type common.Block | jq .data.
↪data[0].payload.data.config > config.json
```

This command leaves us with a trimmed down JSON object – `config.json` – which will serve as the baseline for our config update.

Take a moment to open this file inside your text editor of choice (or in your browser). Even after you're done with this tutorial, it will be worth studying it as it reveals the underlying configuration structure and the other kind of channel updates that can be made. We discuss them in more detail in *Updating a channel configuration*.

7.6.9 Add the Org3 Crypto Material

Note: The steps you've taken up to this point will be nearly identical no matter what kind of config update you're trying to make. We've chosen to add an org with this tutorial because it's one of the most complex channel configuration updates you can attempt.

We'll use the `jq` tool once more to append the Org3 configuration definition – `org3.json` – to the channel's application groups field, and name the output – `modified_config.json`.

```
jq -s '.[0] * {"channel_group":{"groups":{"Application":{"groups": {"Org3MSP":.[1]}}}}
↪}' config.json ./organizations/peerOrganizations/org3.example.com/org3.json >␣
↪modified_config.json
```

Now, within the Org3CLI container we have two JSON files of interest – `config.json` and `modified_config.json`. The initial file contains only Org1 and Org2 material, whereas the "modified" file contains all three Orgs. At this point it's simply a matter of re-encoding these two JSON files and calculating the delta.

First, translate `config.json` back into a protobuf called `config.pb`:

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
```

Next, encode `modified_config.json` to `modified_config.pb`:

```
configtxlator proto_encode --input modified_config.json --type common.Config --output␣
↪modified_config.pb
```

Now use `configtxlator` to calculate the delta between these two config protobufs. This command will output a new protobuf binary named `org3_update.pb`:

```
configtxlator compute_update --channel_id $CHANNEL_NAME --original config.pb --
↪updated modified_config.pb --output org3_update.pb
```

This new proto – `org3_update.pb` – contains the Org3 definitions and high level pointers to the Org1 and Org2 material. We are able to forgo the extensive MSP material and modification policy information for Org1 and Org2 because this data is already present within the channel's genesis block. As such, we only need the delta between the two configurations.

Before submitting the channel update, we need to perform a few final steps. First, let's decode this object into editable JSON format and call it `org3_update.json`:

```
configtxlator proto_decode --input org3_update.pb --type common.ConfigUpdate | jq . >␣
↪org3_update.json
```

Now, we have a decoded update file – `org3_update.json` – that we need to wrap in an envelope message. This step will give us back the header field that we stripped away earlier. We'll name this file `org3_update_in_envelope.json`:

```
echo '{"payload":{"header":{"channel_header":{"channel_id":"'$CHANNEL_NAME'", "type
↪":2}},"data":{"config_update":'$(cat org3_update.json)'}}}' | jq . > org3_update_in_
↪envelope.json
```

Using our properly formed JSON – `org3_update_in_envelope.json` – we will leverage the `configtxlator` tool one last time and convert it into the fully fledged protobuf format that Fabric requires. We'll name our final update object `org3_update_in_envelope.pb`:

```
configtxlator proto_encode --input org3_update_in_envelope.json --type common.
↪Envelope --output org3_update_in_envelope.pb
```

7.6.10 Sign and Submit the Config Update

Almost done!

We now have a protobuf binary – `org3_update_in_envelope.pb` – within the Org3CLI container. However, we need signatures from the requisite Admin users before the config can be written to the ledger. The modification policy (mod_policy) for our channel Application group is set to the default of "MAJORITY", which means that we need a majority of existing org admins to sign it. Because we have only two orgs – Org1 and Org2 – and the majority of two is two, we need both of them to sign. Without both signatures, the ordering service will reject the transaction for failing to fulfill the policy.

First, let's sign this update proto as Org1. Remember that we exported the necessary environment variables to operate the Org3CLI container as the Org1 admin. As a result, the following `peer channel signconfigtx` command will sign the update as Org1.

```
peer channel signconfigtx -f org3_update_in_envelope.pb
```

The final step is to switch the container's identity to reflect the Org2 Admin user. We do this by exporting four environment variables specific to the Org2 MSP.

Note: Switching between organizations to sign a config transaction (or to do anything else) is not reflective of a real-world Fabric operation. A single container would never be mounted with an entire network's crypto material. Rather, the config update would need to be securely passed out-of-band to an Org2 Admin for inspection and approval.

Export the Org2 environment variables:

```
# you can issue all of these commands at once

export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/
↪ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪organizations/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
export CORE_PEER_ADDRESS=peer0.org2.example.com:9051
```

Lastly, we will issue the `peer channel update` command. The Org2 Admin signature will be attached to this call so there is no need to manually sign the protobuf a second time:

Note: The upcoming update call to the ordering service will undergo a series of systematic signature and policy checks. As such you may find it useful to stream and inspect the ordering node's logs. You can issue a `docker logs -f orderer.example.com` command from a terminal outside the Org3CLI container to display them.

Send the update call:

```
peer channel update -f org3_update_in_envelope.pb -c $CHANNEL_NAME -o orderer.example.
↪com:7050 --tls --cafile $ORDERER_CA
```

You should see a message similar to the following if your update has been submitted successfully:

```
2020-01-09 21:30:45.791 UTC [channelCmd] update -> INFO 002 Successfully submitted␣
↪channel update
```

The successful channel update call returns a new block – block 3 – to all of the peers on the channel. If you remember, blocks 0-2 are the initial channel configurations. Block 3 serves as the most recent channel configuration with Org3 now defined on the channel.

You can inspect the logs for `peer0.org1.example.com` by navigating to a terminal outside the Org3CLI container and issuing the following command:

```
docker logs -f peer0.org1.example.com
```

7.6.11 Join Org3 to the Channel

At this point, the channel configuration has been updated to include our new organization – Org3 – meaning that peers attached to it can now join `mychannel`.

Inside the Org3CLI container, export the following environment variables to operate as the Org3 Admin:

```
# you can issue all of these commands at once

export CORE_PEER_LOCALMSPID="Org3MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪organizations/peerOrganizations/org3.example.com/peers/peer0.org3.example.com/tls/
↪ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪organizations/peerOrganizations/org3.example.com/users/Admin@org3.example.com/msp
export CORE_PEER_ADDRESS=peer0.org3.example.com:11051
```

Now let's send a call to the ordering service asking for the genesis block of `mychannel`. As a result of the successful channel update, the ordering service will verify that Org3 can pull the genesis block and join the channel. If Org3 had not been successfully appended to the channel config, the ordering service would reject this request.

Note: Again, you may find it useful to stream the ordering node's logs to reveal the sign/verify logic and policy checks.

Use the `peer channel fetch` command to retrieve this block:

```
peer channel fetch 0 mychannel.block -o orderer.example.com:7050 -c $CHANNEL_NAME --
↪tls --cafile $ORDERER_CA
```

Notice, that we are passing a `0` to indicate that we want the first block on the channel's ledger; the genesis block. If we simply passed the `peer channel fetch config` command, then we would have received block 3 – the updated config with Org3 defined. However, we can't begin our ledger with a downstream block – we must start with block 0.

If successful, the command returned the genesis block to a file named `mychannel.block`. We can now use this block to join the peer to the channel. Issue the `peer channel join` command and pass in the genesis block to join the Org3 peer to the channel:

```
peer channel join -b mychannel.block
```

7.6.12 Configuring Leader Election

Note: This section is included as a general reference for understanding the leader election settings when adding organizations to a network after the initial channel configuration has completed. This sample defaults to dynamic leader election, which is set for all peers in the network.

Newly joining peers are bootstrapped with the genesis block, which does not contain information about the organization that is being added in the channel configuration update. Therefore new peers are not able to utilize gossip as they cannot verify blocks forwarded by other peers from their own organization until they get the configuration transaction which added the organization to the channel. Newly added peers must therefore have one of the following configurations so that they receive blocks from the ordering service:

1. To utilize static leader mode, configure the peer to be an organization leader:

```
CORE_PEER_GOSSIP_USELEADERELECTION=false
CORE_PEER_GOSSIP_ORGLEADER=true
```

Note: This configuration must be the same for all new peers added to the channel.

2. To utilize dynamic leader election, configure the peer to use leader election:

```
CORE_PEER_GOSSIP_USELEADERELECTION=true
CORE_PEER_GOSSIP_ORGLEADER=false
```

Note: Because peers of the newly added organization won't be able to form membership view, this option will be similar to the static configuration, as each peer will start proclaiming itself to be a leader. However, once they get updated with the configuration transaction that adds the organization to the channel, there will be only one active leader for the organization. Therefore, it is recommended to leverage this option if you eventually want the organization's peers to utilize leader election.

7.6.13 Install, define, and invoke chaincode

We can confirm that Org3 is a member of `mychannel` by installing and invoking a chaincode on the channel. If the existing channel members have already committed a chaincode definition to the channel, a new organization can start using the chaincode by approving the chaincode definition.

Note: These instructions use the Fabric chaincode lifecycle introduced in the v2.0 release. If you would like to use the previous lifecycle to install and instantiate a chaincode, visit the v1.4 version of the Adding an org to a channel tutorial.

Before we install a chaincode as Org3, we can use the `./network.sh` script to deploy the Fabcar chaincode on the channel. Open a new terminal outside the Org3CLI container and navigate to the `test-network` directory. You can then use use the `test-network` script to deploy the Fabcar chaincode:

```
cd fabric-samples/test-network
./network.sh deployCC
```

The script will install the Fabcar chaincode on the Org1 and Org2 peers, approve the chaincode definition for Org1 and Org2, and then commit the chaincode definition to the channel. Once the chaincode definition has been committed to the channel, the Fabcar chaincode is initialized and invoked to put initial data on the ledger. The commands below assume that we are still using the channel `mychannel`.

After the chaincode has been to deployed we can use the following steps to use invoke Fabcar chaincode as Org3. These steps can be completed from the `test-network` directory, without having to exec into Org3CLI container. Copy and paste the following environment variables in your terminal in order to interact with the network as the Org3 admin:

```
export PATH=${PWD}/../bin:${PWD}:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org3MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org3.
↪example.com/peers/peer0.org3.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org3.example.
↪com/users/Admin@org3.example.com/msp
export CORE_PEER_ADDRESS=localhost:11051
```

The first step is to package the Fabcar chaincode:

```
peer lifecycle chaincode package fabcar.tar.gz --path ../chaincode/fabcar/go/ --lang␣
↪golang --label fabcar_1
```

This command will create a chaincode package named `fabcar.tar.gz`, which we can install on the Org3 peer. Modify the command accordingly if the channel is running a chaincode written in Java or Node.js. Issue the following command to install the chaincode package `peer0.org3.example.com`:

```
peer lifecycle chaincode install fabcar.tar.gz
```

The next step is to approve the chaincode definition of Fabcar as Org3. Org3 needs to approve the same definition that Org1 and Org2 approved and committed to the channel. In order to invoke the chaincode, Org3 needs to include the package identifier in the chaincode definition. You can find the package identifier by querying your peer:

```
peer lifecycle chaincode queryinstalled
```

You should see output similar to the following:

```
Get installed chaincodes on peer:
Package ID: fabcar_1:25f28c212da84a8eca44d14cf12549d8f7b674a0d8288245561246fa90f7ab03,
↪ Label: fabcar_1
```

We are going to need the package ID in a future command, so lets go ahead and save it as an environment variable. Paste the package ID returned by the `peer lifecycle chaincode queryinstalled` command into the command below. The package ID may not be the same for all users, so you need to complete this step using the package ID returned from your console.

```
export CC_PACKAGE_ID=fabcar_
↪1:25f28c212da84a8eca44d14cf12549d8f7b674a0d8288245561246fa90f7ab03
```

Use the following command to approve a definition of the Fabcar chaincode for Org3:

```
# use the --package-id flag to provide the package identifier
# use the --init-required flag to request the ``Init`` function be invoked to␣
↪initialize the chaincode
peer lifecycle chaincode approveformyorg -o localhost:7050 --
↪ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name fabcar -
↪-version 1 --init-required --package-id $CC_PACKAGE_ID --sequence 1 --tls true --
↪cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.
↪example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

You can use the `peer lifecycle chaincode querycommitted` command to check if the chaincode definition you have approved has already been committed to the channel.

```
# use the --name flag to select the chaincode whose definition you want to query
peer lifecycle chaincode querycommitted --channelID mychannel --name fabcar --cafile $
↪{PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/
↪msp/tlscacerts/tlsca.example.com-cert.pem
```

A successful command will return information about the committed definition:

```
Committed chaincode definition for chaincode 'fabcar' on channel 'mychannel':
Version: 1, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc,␣
↪Approvals: [Org1MSP: true, Org2MSP: true, Org3MSP: true]
```

Org3 can use the Fabcar chaincode after it approves the chaincode definition that was committed to the channel. The chaincode definition uses the default endorsement policy, which requires a majority of organizations on the channel endorse a transaction. This implies that if an organization is added to or removed from the channel, the endorsement policy will be updated automatically. We previously needed endorsements from Org1 and Org2 (2 out of 2). Now we need endorsements from two organizations out of Org1, Org2, and Org3 (2 out of 3).

You can query the chaincode to ensure that it has started on the Org3 peer. Note that you may need to wait for the chaincode container to start.

```
peer chaincode query -C mychannel -n fabcar -c '{"Args":["queryAllCars"]}'
```

You should see the initial list of cars that were added to the ledger as a response.

Now, invoke the chaincode to add a new car to the ledger. In the command below, we target a peer in Org1 and Org3 to collect a sufficient number of endorsements.

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.
↪com --tls true --cafile ${PWD}/organizations/ordererOrganizations/example.com/
↪orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel␣
↪-n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses localhost:11051 --tlsRootCertFiles ${PWD}/organizations/
↪peerOrganizations/org3.example.com/peers/peer0.org3.example.com/tls/ca.crt -c '{
↪"function":"createCar","Args":["CAR11","Honda","Accord","Black","Tom"]}'
```

We can query again to see the new car, "CAR11" on the our the ledger:

```
peer chaincode query -C mychannel -n fabcar -c '{"Args":["queryCar","CAR11"]}'
```

7.6.14 Conclusion

The channel configuration update process is indeed quite involved, but there is a logical method to the various steps. The endgame is to form a delta transaction object represented in protobuf binary format and then acquire the requisite

number of admin signatures such that the channel configuration update transaction fulfills the channel's modification policy.

The `configtxlator` and `jq` tools, along with the `peer channel` commands, provide us with the functionality to accomplish this task.

7.6.15 Updating the Channel Config to include an Org3 Anchor Peer (Optional)

The Org3 peers were able to establish gossip connection to the Org1 and Org2 peers since Org1 and Org2 had anchor peers defined in the channel configuration. Likewise newly added organizations like Org3 should also define their anchor peers in the channel configuration so that any new peers from other organizations can directly discover an Org3 peer. In this section, we will make a channel configuration update to define an Org3 anchor peer. The process will be similar to the previous configuration update, therefore we'll go faster this time.

If you don't have it open, exec back into the Org3CLI container:

```
docker exec -it Org3cli bash
```

Export the $ORDERER_CA and $CHANNEL_NAME variables if they are not already set:

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/organizations/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪example.com-cert.pem
export CHANNEL_NAME=mychannel
```

As before, we will fetch the latest channel configuration to get started. Inside the Org3CLI container, fetch the most recent config block for the channel, using the `peer channel fetch` command.

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CHANNEL_
↪NAME --tls --cafile $ORDERER_CA
```

After fetching the config block we will want to convert it into JSON format. To do this we will use the configtxlator tool, as done previously when adding Org3 to the channel. When converting it we need to remove all the headers, metadata, and signatures that are not required to update Org3 to include an anchor peer by using the jq tool. This information will be reincorporated later before we proceed to update the channel configuration.

```
configtxlator proto_decode --input config_block.pb --type common.Block | jq .data.
↪data[0].payload.data.config > config.json
```

The `config.json` is the now trimmed JSON representing the latest channel configuration that we will update.

Using the jq tool again, we will update the configuration JSON with the Org3 anchor peer we want to add.

```
jq '.channel_group.groups.Application.groups.Org3MSP.values += {"AnchorPeers":{"mod_
↪policy": "Admins","value":{"anchor_peers": [{"host": "peer0.org3.example.com","port
↪": 11051}]},"version": "0"}}' config.json > modified_anchor_config.json
```

We now have two JSON files, one for the current channel configuration, `config.json`, and one for the desired channel configuration `modified_anchor_config.json`. Next we convert each of these back into protobuf format and calculate the delta between the two.

Translate `config.json` back into protobuf format as `config.pb`

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
```

Translate the `modified_anchor_config.json` into protobuf format as `modified_anchor_config.pb`

```
configtxlator proto_encode --input modified_anchor_config.json --type common.Config --
↪output modified_anchor_config.pb
```

Calculate the delta between the two protobuf formatted configurations.

```
configtxlator compute_update --channel_id $CHANNEL_NAME --original config.pb --
↪updated modified_anchor_config.pb --output anchor_update.pb
```

Now that we have the desired update to the channel we must wrap it in an envelope message so that it can be properly read. To do this we must first convert the protobuf back into a JSON that can be wrapped.

We will use the configtxlator command again to convert `anchor_update.pb` into `anchor_update.json`

```
configtxlator proto_decode --input anchor_update.pb --type common.ConfigUpdate | jq .␣
↪> anchor_update.json
```

Next we will wrap the update in an envelope message, restoring the previously stripped away header, outputting it to `anchor_update_in_envelope.json`

```
echo '{"payload":{"header":{"channel_header":{"channel_id":"'$CHANNEL_NAME'", "type
↪":2}},"data":{"config_update":'$(cat anchor_update.json)'}}}' | jq . > anchor_
↪update_in_envelope.json
```

Now that we have reincorporated the envelope we need to convert it to a protobuf so it can be properly signed and submitted to the orderer for the update.

```
configtxlator proto_encode --input anchor_update_in_envelope.json --type common.
↪Envelope --output anchor_update_in_envelope.pb
```

Now that the update has been properly formatted it is time to sign off and submit it. Since this is only an update to Org3 we only need to have Org3 sign off on the update. Run the following commands to make sure that we are operating as the Org3 admin:

```
# you can issue all of these commands at once

export CORE_PEER_LOCALMSPID="Org3MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪organizations/peerOrganizations/org3.example.com/peers/peer0.org3.example.com/tls/
↪ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪organizations/peerOrganizations/org3.example.com/users/Admin@org3.example.com/msp
export CORE_PEER_ADDRESS=peer0.org3.example.com:11051
```

We can now just use the `peer channel update` command to sign the update as the Org3 admin before submitting it to the orderer.

```
peer channel update -f anchor_update_in_envelope.pb -c $CHANNEL_NAME -o orderer.
↪example.com:7050 --tls --cafile $ORDERER_CA
```

The orderer receives the config update request and cuts a block with the updated configuration. As peers receive the block, they will process the configuration updates.

Inspect the logs for one of the peers. While processing the configuration transaction from the new block, you will see gossip re-establish connections using the new anchor peer for Org3. This is proof that the configuration update has been successfully applied!

```
docker logs -f peer0.org1.example.com
```

```
2019-06-12 17:08:57.924 UTC [gossip.gossip] learnAnchorPeers -> INFO 89a Learning␣
↪about the configured anchor peers of Org1MSP for channel mychannel : [{peer0.org1.
↪example.com 7051}]
2019-06-12 17:08:57.926 UTC [gossip.gossip] learnAnchorPeers -> INFO 89b Learning␣
↪about the configured anchor peers of Org2MSP for channel mychannel : [{peer0.org2.
↪example.com 9051}]
2019-06-12 17:08:57.926 UTC [gossip.gossip] learnAnchorPeers -> INFO 89c Learning␣
↪about the configured anchor peers of Org3MSP for channel mychannel : [{peer0.org3.
↪example.com 11051}]
```

Congratulations, you have now made two configuration updates — one to add Org3 to the channel, and a second to define an anchor peer for Org3.

7.7 Updating a channel configuration

Audience: network administrators, node administrators

7.7.1 What is a channel configuration?

Like many complex systems, Hyperledger Fabric networks are comprised of both **structure** and a number related of **processes**.

- **Structure**: encompassing users (like admins), organizations, peers, ordering nodes, CAs, smart contracts, and applications.
- **Process**: the way these structures interact. Most important of these are Policies, the rules that govern which users can do what, and under what conditions.

Information identifying the structure of blockchain networks and the processes governing how structures interact are contained in **channel configurations**. These configurations are collectively decided upon by the members of channels and are contained in blocks that are committed to the ledger of a channel. Channel configurations can be built using a tool called `configtxgen`, which uses a `configtx.yaml` file as its input. You can look at a sample `configtx.yaml` file here.

Because configurations are contained in blocks (the first of these is known as the genesis block with the latest representing the current configuration of the channel), the process for updating a channel configuration (changing the structure by adding members, for example, or processes by modifying channel policies) is known as a **configuration update transaction**.

In production networks, these configuration update transactions will normally be proposed by a single channel admin after an out of band discussion, just as the initial configuration of the channel will be decided on out of band by the initial members of the channel.

In this topic, we'll:

- Show a full sample configuration of an application channel.
- Discuss many of the channel parameters that can be edited.
- Show the process for updating a channel configuration, including the commands necessary to pull, translate, and scope a configuration into something that humans can read.
- Discuss the methods that can be used to edit a channel configuration.
- Show the process used to reformat a configuration and get the signatures necessary for it to be approved.

7.7.2 Channel parameters that can be updated

Channels are highly configurable, but not infinitely so. Once certain things about a channel (for example, the name of the channel) have been specified, they cannot be changed. And changing one of the parameters we'll talk about in this topic requires satisfying the relevant policy as specified in the channel configuration.

In this section, we'll look a sample channel configuration and show the configuration parameters that can be updated.

Sample channel configuration

To see what the configuration file of an application channel looks like after it has been pulled and scoped, click **Click here to see the config** below. For ease of readability, it might be helpful to put this config into a viewer that supports JSON folding, like atom or Visual Studio.

Note: for simplicity, we are only showing an application channel configuration here. The configuration of the orderer system channel is very similar, but not identical, to the configuration of an application channel. However, the same basic rules and structure apply, as do the commands to pull and edit a configuration, as you can see in our topic on Updating the capability level of a channel.

Click here to see the config. Note that this is the configuration of an application channel, not the orderer system channel.

```
{
  "channel_group": {
    "groups": {
      "Application": {
        "groups": {
          "Org1MSP": {
            "groups": {},
            "mod_policy": "Admins",
            "policies": {
              "Admins": {
                "mod_policy": "Admins",
                "policy": {
                  "type": 1,
                  "value": {
                    "identities": [
                      {
                        "principal": {
                          "msp_identifier": "Org1MSP",
                          "role": "ADMIN"
                        },
                        "principal_classification": "ROLE"
                      }
                    ],
                    "rule": {
                      "n_out_of": {
                        "n": 1,
                        "rules": [
                          {
                            "signed_by": 0
                          }
                        ]
                      }
                    },
                    "version": 0
                  }
```

(continues on next page)

(continued from previous page)

```
                },
                "version": "0"
              },
              "Endorsement": {
                "mod_policy": "Admins",
                "policy": {
                  "type": 1,
                  "value": {
                    "identities": [
                      {
                        "principal": {
                          "msp_identifier": "Org1MSP",
                          "role": "PEER"
                        },
                        "principal_classification": "ROLE"
                      }
                    ],
                    "rule": {
                      "n_out_of": {
                        "n": 1,
                        "rules": [
                          {
                            "signed_by": 0
                          }
                        ]
                      }
                    },
                    "version": 0
                  }
                },
                "version": "0"
              },
              "Readers": {
                "mod_policy": "Admins",
                "policy": {
                  "type": 1,
                  "value": {
                    "identities": [
                      {
                        "principal": {
                          "msp_identifier": "Org1MSP",
                          "role": "ADMIN"
                        },
                        "principal_classification": "ROLE"
                      },
                      {
                        "principal": {
                          "msp_identifier": "Org1MSP",
                          "role": "PEER"
                        },
                        "principal_classification": "ROLE"
                      },
                      {
                        "principal": {
                          "msp_identifier": "Org1MSP",
                          "role": "CLIENT"
                        },
```

(continues on next page)

(continued from previous page)

```
                "principal_classification": "ROLE"
              }
            ],
            "rule": {
              "n_out_of": {
                "n": 1,
                "rules": [
                  {
                    "signed_by": 0
                  },
                  {
                    "signed_by": 1
                  },
                  {
                    "signed_by": 2
                  }
                ]
              }
            },
            "version": 0
          }
        },
        "version": "0"
      },
      "Writers": {
        "mod_policy": "Admins",
        "policy": {
          "type": 1,
          "value": {
            "identities": [
              {
                "principal": {
                  "msp_identifier": "Org1MSP",
                  "role": "ADMIN"
                },
                "principal_classification": "ROLE"
              },
              {
                "principal": {
                  "msp_identifier": "Org1MSP",
                  "role": "CLIENT"
                },
                "principal_classification": "ROLE"
              }
            ],
            "rule": {
              "n_out_of": {
                "n": 1,
                "rules": [
                  {
                    "signed_by": 0
                  },
                  {
                    "signed_by": 1
                  }
                ]
              }
```

(continues on next page)

(continued from previous page)


```
                    },
                    "version": 0
                  }
                },
                "version": "0"
              }
            },
            "values": {
              "AnchorPeers": {
                "mod_policy": "Admins",
                "value": {
                  "anchor_peers": [
                    {
                      "host": "peer0.org1.example.com",
                      "port": 7051
                    }
                  ]
                },
                "version": "0"
              },
              "MSP": {
                "mod_policy": "Admins",
                "value": {
                  "config": {
                    "admins": [],
                    "crypto_config": {
                      "identity_identifier_hash_function": "SHA256",
                      "signature_hash_family": "SHA2"
                    },
                    "fabric_node_ous": {
                      "admin_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ0lVYWVSeWNkQytlR1lUTUNyWTg2UFVXUEdzQU
↪",
                        "organizational_unit_identifier": "admin"
                      },
                      "client_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ0lVYWVSeWNkQytlR1lUTUNyWTg2UFVXUEdzQU
↪",
                        "organizational_unit_identifier": "client"
                      },
                      "enable": true,
                      "orderer_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ0lVYWVSeWNkQytlR1lUTUNyWTg2UFVXUEdzQU
↪",
                        "organizational_unit_identifier": "orderer"
                      },
                      "peer_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ0lVYWVSeWNkQytlR1lUTUNyWTg2UFVXUEdzQU
↪",
                        "organizational_unit_identifier": "peer"
                      }
                    },
                    "intermediate_certs": [],
```


(continues on next page)

(continued from previous page)


```
                    "name": "Org1MSP",
                    "organizational_unit_identifiers": [],
                    "revocation_list": [],
                    "root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ0lVYWVSeWNkQytlR1lUTUNyWTg2UFVXUEdzQU
↪"
                    ],
                    "signing_identity": null,
                    "tls_intermediate_certs": [],
                    "tls_root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNKekNDQWMyZ0F3SUJBZ0lVYWVSeWNkQytlR1lUTUNyWTg2UFVXUEdzQU
↪"
                    ]
                  },
                  "type": 0
                },
                "version": "0"
              }
            },
            "version": "1"
          },
          "Org2MSP": {
            "groups": {},
            "mod_policy": "Admins",
            "policies": {
              "Admins": {
                "mod_policy": "Admins",
                "policy": {
                  "type": 1,
                  "value": {
                    "identities": [
                      {
                        "principal": {
                          "msp_identifier": "Org2MSP",
                          "role": "ADMIN"
                        },
                        "principal_classification": "ROLE"
                      }
                    ],
                    "rule": {
                      "n_out_of": {
                        "n": 1,
                        "rules": [
                          {
                            "signed_by": 0
                          }
                        ]
                      }
                    },
                    "version": 0
                  }
                },
                "version": "0"
              },
              "Endorsement": {
```


(continues on next page)

(continued from previous page)

```
                "mod_policy": "Admins",
                "policy": {
                  "type": 1,
                  "value": {
                    "identities": [
                      {
                        "principal": {
                          "msp_identifier": "Org2MSP",
                          "role": "PEER"
                        },
                        "principal_classification": "ROLE"
                      }
                    ],
                    "rule": {
                      "n_out_of": {
                        "n": 1,
                        "rules": [
                          {
                            "signed_by": 0
                          }
                        ]
                      }
                    },
                    "version": 0
                  }
                },
                "version": "0"
              },
              "Readers": {
                "mod_policy": "Admins",
                "policy": {
                  "type": 1,
                  "value": {
                    "identities": [
                      {
                        "principal": {
                          "msp_identifier": "Org2MSP",
                          "role": "ADMIN"
                        },
                        "principal_classification": "ROLE"
                      },
                      {
                        "principal": {
                          "msp_identifier": "Org2MSP",
                          "role": "PEER"
                        },
                        "principal_classification": "ROLE"
                      },
                      {
                        "principal": {
                          "msp_identifier": "Org2MSP",
                          "role": "CLIENT"
                        },
                        "principal_classification": "ROLE"
                      }
                    ],
                    "rule": {
```

(continues on next page)

(continued from previous page)

```
                    "n_out_of": {
                      "n": 1,
                      "rules": [
                        {
                          "signed_by": 0
                        },
                        {
                          "signed_by": 1
                        },
                        {
                          "signed_by": 2
                        }
                      ]
                    }
                  },
                  "version": 0
                }
              },
              "version": "0"
            },
            "Writers": {
              "mod_policy": "Admins",
              "policy": {
                "type": 1,
                "value": {
                  "identities": [
                    {
                      "principal": {
                        "msp_identifier": "Org2MSP",
                        "role": "ADMIN"
                      },
                      "principal_classification": "ROLE"
                    },
                    {
                      "principal": {
                        "msp_identifier": "Org2MSP",
                        "role": "CLIENT"
                      },
                      "principal_classification": "ROLE"
                    }
                  ],
                  "rule": {
                    "n_out_of": {
                      "n": 1,
                      "rules": [
                        {
                          "signed_by": 0
                        },
                        {
                          "signed_by": 1
                        }
                      ]
                    }
                  },
                  "version": 0
                }
              },
```

(continues on next page)

(continued from previous page)


```
                "version": "0"
              }
            },
            "values": {
              "AnchorPeers": {
                "mod_policy": "Admins",
                "value": {
                  "anchor_peers": [
                    {
                      "host": "peer0.org2.example.com",
                      "port": 9051
                    }
                  ]
                },
                "version": "0"
              },
              "MSP": {
                "mod_policy": "Admins",
                "value": {
                  "config": {
                    "admins": [],
                    "crypto_config": {
                      "identity_identifier_hash_function": "SHA256",
                      "signature_hash_family": "SHA2"
                    },
                    "fabric_node_ous": {
                      "admin_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNIakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS
↪",
                        "organizational_unit_identifier": "admin"
                      },
                      "client_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNIakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS
↪",
                        "organizational_unit_identifier": "client"
                      },
                      "enable": true,
                      "orderer_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNIakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS
↪",
                        "organizational_unit_identifier": "orderer"
                      },
                      "peer_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNIakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS
↪",
                        "organizational_unit_identifier": "peer"
                      }
                    },
                    "intermediate_certs": [],
                    "name": "Org2MSP",
                    "organizational_unit_identifiers": [],
                    "revocation_list": [],
                    "root_certs": [
```


(continues on next page)

(continued from previous page)

```
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNIakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS
↪"
                    ],
                    "signing_identity": null,
                    "tls_intermediate_certs": [],
                    "tls_root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNIakNDQWNXZ0F3SUJBZ0lVQVFkb1B0S0E0bEk2a0RrMituYzk5NzNhS
↪"
                    ]
                  },
                  "type": 0
                },
                "version": "0"
              }
            },
            "version": "1"
          }
        },
        "mod_policy": "Admins",
        "policies": {
          "Admins": {
            "mod_policy": "Admins",
            "policy": {
              "type": 3,
              "value": {
                "rule": "MAJORITY",
                "sub_policy": "Admins"
              }
            },
            "version": "0"
          },
          "Endorsement": {
            "mod_policy": "Admins",
            "policy": {
              "type": 3,
              "value": {
                "rule": "MAJORITY",
                "sub_policy": "Endorsement"
              }
            },
            "version": "0"
          },
          "LifecycleEndorsement": {
            "mod_policy": "Admins",
            "policy": {
              "type": 3,
              "value": {
                "rule": "MAJORITY",
                "sub_policy": "Endorsement"
              }
            },
            "version": "0"
          },
          "Readers": {
            "mod_policy": "Admins",
```

(continues on next page)

(continued from previous page)

```
            "policy": {
              "type": 3,
              "value": {
                "rule": "ANY",
                "sub_policy": "Readers"
              }
            },
            "version": "0"
          },
          "Writers": {
            "mod_policy": "Admins",
            "policy": {
              "type": 3,
              "value": {
                "rule": "ANY",
                "sub_policy": "Writers"
              }
            },
            "version": "0"
          }
        },
        "values": {
          "Capabilities": {
            "mod_policy": "Admins",
            "value": {
              "capabilities": {
                "V2_0": {}
              }
            },
            "version": "0"
          }
        },
        "version": "1"
      },
      "Orderer": {
        "groups": {
          "OrdererOrg": {
            "groups": {},
            "mod_policy": "Admins",
            "policies": {
              "Admins": {
                "mod_policy": "Admins",
                "policy": {
                  "type": 1,
                  "value": {
                    "identities": [
                      {
                        "principal": {
                          "msp_identifier": "OrdererMSP",
                          "role": "ADMIN"
                        },
                        "principal_classification": "ROLE"
                      }
                    ],
                    "rule": {
                      "n_out_of": {
                        "n": 1,
```

(continues on next page)

(continued from previous page)

```
                    "rules": [
                      {
                        "signed_by": 0
                      }
                    ]
                  }
                },
                "version": 0
              }
            },
            "version": "0"
          },
          "Readers": {
            "mod_policy": "Admins",
            "policy": {
              "type": 1,
              "value": {
                "identities": [
                  {
                    "principal": {
                      "msp_identifier": "OrdererMSP",
                      "role": "MEMBER"
                    },
                    "principal_classification": "ROLE"
                  }
                ],
                "rule": {
                  "n_out_of": {
                    "n": 1,
                    "rules": [
                      {
                        "signed_by": 0
                      }
                    ]
                  }
                },
                "version": 0
              }
            },
            "version": "0"
          },
          "Writers": {
            "mod_policy": "Admins",
            "policy": {
              "type": 1,
              "value": {
                "identities": [
                  {
                    "principal": {
                      "msp_identifier": "OrdererMSP",
                      "role": "MEMBER"
                    },
                    "principal_classification": "ROLE"
                  }
                ],
                "rule": {
                  "n_out_of": {
```

(continues on next page)

(continued from previous page)


```
                        "n": 1,
                        "rules": [
                          {
                            "signed_by": 0
                          }
                        ]
                      }
                    },
                    "version": 0
                  }
                },
                "version": "0"
              }
            },
            "values": {
              "MSP": {
                "mod_policy": "Admins",
                "value": {
                  "config": {
                    "admins": [],
                    "crypto_config": {
                      "identity_identifier_hash_function": "SHA256",
                      "signature_hash_family": "SHA2"
                    },
                    "fabric_node_ous": {
                      "admin_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDekNDQWJHZ0F3SUJBZ0lVUkgyT0tlV1loaStFMkFHZ3IwWUdlVTRUOW
↪",
                        "organizational_unit_identifier": "admin"
                      },
                      "client_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDekNDQWJHZ0F3SUJBZ0lVUkgyT0tlV1loaStFMkFHZ3IwWUdlVTRUOW
↪",
                        "organizational_unit_identifier": "client"
                      },
                      "enable": true,
                      "orderer_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDekNDQWJHZ0F3SUJBZ0lVUkgyT0tlV1loaStFMkFHZ3IwWUdlVTRUOW
↪",
                        "organizational_unit_identifier": "orderer"
                      },
                      "peer_ou_identifier": {
                        "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDekNDQWJHZ0F3SUJBZ0lVUkgyT0tlV1loaStFMkFHZ3IwWUdlVTRUOW
↪",
                        "organizational_unit_identifier": "peer"
                      }
                    },
                    "intermediate_certs": [],
                    "name": "OrdererMSP",
                    "organizational_unit_identifiers": [],
                    "revocation_list": [],
                    "root_certs": [
```


(continues on next page)


```
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDekNDQWJHZ0F3SUJBZ0lVUkgyT0tlV1loaStFMkFHZ3IwWUdlVTRUOW
↪"
```

(continued from previous page)


```
                    ],
                    "signing_identity": null,
                    "tls_intermediate_certs": [],
                    "tls_root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDekNDQWJHZ0F3SUJBZ0lVUkgyT0tlV1loaStFMkFHZ3IwWUdlVTRUOW
↪"
                    ]
                  },
                  "type": 0
                },
                "version": "0"
              }
            },
            "version": "0"
          }
        },
        "mod_policy": "Admins",
        "policies": {
          "Admins": {
            "mod_policy": "Admins",
            "policy": {
              "type": 3,
              "value": {
                "rule": "MAJORITY",
                "sub_policy": "Admins"
              }
            },
            "version": "0"
          },
          "BlockValidation": {
            "mod_policy": "Admins",
            "policy": {
              "type": 3,
              "value": {
                "rule": "ANY",
                "sub_policy": "Writers"
              }
            },
            "version": "0"
          },
          "Readers": {
            "mod_policy": "Admins",
            "policy": {
              "type": 3,
              "value": {
                "rule": "ANY",
                "sub_policy": "Readers"
              }
            },
            "version": "0"
          },
          "Writers": {
            "mod_policy": "Admins",
            "policy": {
              "type": 3,
              "value": {
```


(continues on next page)

(continued from previous page)


```
                "rule": "ANY",
                "sub_policy": "Writers"
              }
            },
            "version": "0"
          }
        },
        "values": {
          "BatchSize": {
            "mod_policy": "Admins",
            "value": {
              "absolute_max_bytes": 103809024,
              "max_message_count": 10,
              "preferred_max_bytes": 524288
            },
            "version": "0"
          },
          "BatchTimeout": {
            "mod_policy": "Admins",
            "value": {
              "timeout": "2s"
            },
            "version": "0"
          },
          "Capabilities": {
            "mod_policy": "Admins",
            "value": {
              "capabilities": {
                "V2_0": {}
              }
            },
            "version": "0"
          },
          "ChannelRestrictions": {
            "mod_policy": "Admins",
            "value": null,
            "version": "0"
          },
          "ConsensusType": {
            "mod_policy": "Admins",
            "value": {
              "metadata": {
                "consenters": [
                  {
                    "client_tls_cert":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN3akNDQW1pZ0F3SUJBZ0lVZG9JbmpzaW5vVnZua0llbE5WUU8wbDRMbE
↪",
                    "host": "orderer.example.com",
                    "port": 7050,
                    "server_tls_cert":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN3akNDQW1pZ0F3SUJBZ0lVZG9JbmpzaW5vVnZua0llbE5WUU8wbDRMbE
↪"
                  }
                ],
                "options": {
                  "election_tick": 10,
                  "heartbeat_tick": 1,
```


(continues on next page)

(continued from previous page)


```
                "max_inflight_blocks": 5,
                "snapshot_interval_size": 16777216,
                "tick_interval": "500ms"
              }
            },
            "state": "STATE_NORMAL",
            "type": "etcdraft"
          },
          "version": "0"
        }
      },
      "version": "0"
    }
  },
  "mod_policy": "Admins",
  "policies": {
    "Admins": {
      "mod_policy": "Admins",
      "policy": {
        "type": 3,
        "value": {
          "rule": "MAJORITY",
          "sub_policy": "Admins"
        }
      },
      "version": "0"
    },
    "Readers": {
      "mod_policy": "Admins",
      "policy": {
        "type": 3,
        "value": {
          "rule": "ANY",
          "sub_policy": "Readers"
        }
      },
      "version": "0"
    },
    "Writers": {
      "mod_policy": "Admins",
      "policy": {
        "type": 3,
        "value": {
          "rule": "ANY",
          "sub_policy": "Writers"
        }
      },
      "version": "0"
    }
  },
  "values": {
    "BlockDataHashingStructure": {
      "mod_policy": "Admins",
      "value": {
        "width": 4294967295
      },
      "version": "0"
```


(continues on next page)

(continued from previous page)

```
      },
      "Capabilities": {
        "mod_policy": "Admins",
        "value": {
          "capabilities": {
            "V2_0": {}
          }
        },
        "version": "0"
      },
      "Consortium": {
        "mod_policy": "Admins",
        "value": {
          "name": "SampleConsortium"
        },
        "version": "0"
      },
      "HashingAlgorithm": {
        "mod_policy": "Admins",
        "value": {
          "name": "SHA256"
        },
        "version": "0"
      },
      "OrdererAddresses": {
        "mod_policy": "/Channel/Orderer/Admins",
        "value": {
          "addresses": [
            "orderer.example.com:7050"
          ]
        },
        "version": "0"
      }
    },
    "version": "0"
  },
  "sequence": "3"
}
```

A config might look intimidating in this form, but once you study it you'll see that it has a logical structure.

For example, let's take a look at the config with a few of the tabs closed.

Note that this is the configuration of an application channel, not the orderer system channel.

```
{
  "channel_group": {
    "groups": {
      "Application": {
        "groups": {
          "Org1MSP": {…
          "Org2MSP": {…
        },
        "mod_policy": "Admins",
        "policies": {
          "Admins": {…
          "Readers": {…
          "Writers": {…
        },
        "values": {
          "Capabilities": {…
        },
        "version": "1"
      },
      "Orderer": {
        "groups": {
          "OrdererOrg": {…
        },
        "mod_policy": "Admins",
        "policies": {
          "Admins": {…
          "BlockValidation": {…
          "Readers": {…
          "Writers": {…
        },
        "values": {
          "BatchSize": {…
          "BatchTimeout": {…
          "Capabilities": {…
```

The structure of the config should now be more obvious. You can see the config groupings: `Channel`, `Application`, and `Orderer`, and the configuration parameters related to each config grouping (we'll talk more about these in the next section), but also where the MSPs representing organizations are. Note that the `Channel` config grouping is below the `Orderer` group config values.

More about these parameters

In this section, we'll take a deeper look at the configurable values in the context of where they sit in the configuration.

First, there are config parameters that occur in multiple parts of the configuration:

- **Policies**. Policies are not just a configuration value (which can be updated as defined in a `mod_policy`), they define the circumstances under which all parameters can be changed. For more information, check out Policies.
- **Capabilities**. Ensures that networks and channels process things in the same way, creating deterministic results for things like channel configuration updates and chaincode invocations. Without deterministic results, one peer on a channel might invalidate a transaction while another peer may validate it. For more information, check out Capabilities.

`Channel/Application`

Governs the configuration parameters unique to application channels (for example, adding or removing channel members). By default, changing these parameters requires the signature of a majority of the application organization admins.

- **Add orgs to a channel**. To add an organization to a channel, their MSP and other organization parameters must be generated and added here (under `Channel/Application/groups`).
- **Organization-related parameters**. Any parameters specific to an organization, (identifying an anchor peer, for example, or the certificates of org admins), can be changed. Note that changing these values will by default not require the majority of application organization admins but only an admin of the organization itself.

`Channel/Orderer`

Governs configuration parameters unique to the ordering service or the orderer system channel, requires a majority of the ordering organizations' admins (by default there is only one ordering organization, though more can be added, for example when multiple organizations contribute nodes to the ordering service).

- **Batch size**. These parameters dictate the number and size of transactions in a block. No block will appear larger than `absolute_max_bytes` large or with more than `max_message_count` transactions inside the block. If it is possible to construct a block under `preferred_max_bytes`, then a block will be cut prematurely, and transactions larger than this size will appear in their own block.
- **Batch timeout**. The amount of time to wait after the first transaction arrives for additional transactions before cutting a block. Decreasing this value will improve latency, but decreasing it too much may decrease throughput by not allowing the block to fill to its maximum capacity.
- **Block validation**. This policy specifies the signature requirements for a block to be considered valid. By default, it requires a signature from some member of the ordering org.
- **Consensus type**. To enable the migration of Kafka based ordering services to Raft based ordering services, it is possible to change the consensus type of a channel. For more information, check out Migrating from Kafka to Raft.
- **Raft ordering service parameters**. For a look at the parameters unique to a Raft ordering service, check out Raft configuration.

- **Kafka brokers** (where applicable). When `ConsensusType` is set to `kafka`, the `brokers` list enumerates some subset (or preferably all) of the Kafka brokers for the orderer to initially connect to at startup.

`Channel`

Governs configuration parameters that both the peer orgs and the ordering service orgs need to consent to, requires both the agreement of a majority of application organization admins and orderer organization admins.

- **Orderer addresses**. A list of addresses where clients may invoke the orderer `Broadcast` and `Deliver` functions. The peer randomly chooses among these addresses and fails over between them for retrieving blocks.
- **Hashing structure**. The block data is an array of byte arrays. The hash of the block data is computed as a Merkle tree. This value specifies the width of that Merkle tree. For the time being, this value is fixed to `4294967295` which corresponds to a simple flat hash of the concatenation of the block data bytes.
- **Hashing algorithm**. The algorithm used for computing the hash values encoded into the blocks of the blockchain. In particular, this affects the data hash, and the previous block hash fields of the block. Note, this field currently only has one valid value (`SHA256`) and should not be changed.

System channel configuration parameters

Certain configuration values are unique to the orderer system channel.

- **Channel creation policy.** Defines the policy value which will be set as the mod_policy for the Application group of new channels for the consortium it is defined in. The signature set attached to the channel creation request will be checked against the instantiation of this policy in the new channel to ensure that the channel creation is authorized. Note that this config value is only set in the orderer system channel.
- **Channel restrictions.** Only editable in the orderer system channel. The total number of channels the orderer is willing to allocate may be specified as `max_count`. This is primarily useful in pre-production environments with weak consortium `ChannelCreation` policies.

7.7.3 Editing a config

Updating a channel configuration is a three step operation that's conceptually simple:

1. Get the latest channel config
2. Create a modified channel config
3. Create a config update transaction

However, as you'll see, this conceptual simplicity is wrapped in a somewhat convoluted process. As a result, some users might choose to script the process of pulling, translating, and scoping a config update. Users also have the option of how to modify the channel configuration itself, either manually or by using a tool like `jq`.

We have two tutorials that deal specifically with editing a channel configuration to achieve a specific end:

- Adding an Org to a Channel: shows the process for adding an additional organization to an existing channel.
- Updating channel capabilities: shows how to update channel capabilities.

In this topic, we'll show the process of editing a channel configuration independent of the end goal of the configuration update.

Set environment variables for your config update

Before you attempt to use the sample commands, make sure to export the following environment variables, which will depend on the way you have structured your deployment. Note that the channel name, `CH_NAME` will have to be set for every channel being updated, as channel configuration updates only apply to the configuration of the channel being updated (with the exception of the ordering system channel, whose configuration is copied into the configuration of application channels by default).

- `CH_NAME`: the name of the channel being updated.
- `TLS_ROOT_CA`: the path to the root CA cert of the TLS CA of the organization proposing the update.
- `CORE_PEER_LOCALMSPID`: the name of your MSP.
- `CORE_PEER_MSPCONFIGPATH`: the absolute path to the MSP of your organization.
- `ORDERER_CONTAINER`: the name of an ordering node container. Note that when targeting the ordering service, you can target any active node in the ordering service. Your requests will be forwarded to the leader automatically.

Note: this topic will provide default names for the various JSON and protobuf files being pulled and modified (`config_block.pb`, `config_block.json`, etc). You are free to use whatever names you want. However, be aware that unless you go back and erase these files at the end of each config update, you will have to select different when making an additional update.

Step 1: Pull and translate the config

The first step in updating a channel configuration is getting the latest config block. This is a three step process. First, we'll pull the channel configuration in protobuf format, creating a file called `config_block.pb`.

Make sure you are in the peer container.

Now issue:

```
peer channel fetch config config_block.pb -o $ORDERER_CONTAINER -c $CH_NAME --tls --
↪cafile $TLS_ROOT_CA
```

Next, we'll convert the protobuf version of the channel config into a JSON version called `config_block.json` (JSON files are easier for humans to read and understand):

```
configtxlator proto_decode --input config_block.pb --type common.Block --output␣
↪config_block.json
```

Finally, we'll scope out all of the unnecessary metadata from the config, which makes it easier to read. You are free to call this file whatever you want, but in this example we'll call it `config.json`.

```
jq .data.data[0].payload.data.config config_block.json > config.json
```

Now let's make a copy of `config.json` called `modified_config.json`. **Do not edit `config.json` directly**, as we will be using it to compute the difference between `config.json` and `modified_config.json` in a later step.

```
cp config.json modified_config.json
```

Step 2: Modify the config

At this point, you have two options of how you want to modify the config.

1. Open `modified_config.json` using the text editor of your choice and make edits. Online tutorials exist that describe how to copy a file from a container that does not have an editor, edit it, and add it back to the container.
2. Use `jq` to apply edits to the config.

Whether you choose to edit the config manually or using `jq` depends on your use case. Because `jq` is concise and scriptable (an advantage when the same configuration update will be made to multiple channels), it's the recommend method for performing a channel update. For an example on how `jq` can be used, check out Updating channel capabilities, which shows multiple `jq` commands leveraging a capabilities config file called `capabilities.json`. If you are updating something other than the capabilities in your channel, you will have to modify your `jq` command and JSON file accordingly.

For more information about the content and structure of a channel configuration, check out our *sample channel config* above.

Step 3: Re-encode and submit the config

Whether you make your config updates manually or using a tool like `jq`, you now have to run the process you ran to pull and scope the config in reverse, along with a step to calculate the difference between the old config and the new one, before submitting the config update to the other administrators on the channel to be approved.

First, we'll turn our `config.json` file back to protobuf format, creating a file called `config.pb`. Then we'll do the same with our `modified_config.json` file. Afterwards, we'll compute the difference between the two files, creating a file called `config_update.pb`.

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb

configtxlator proto_encode --input modified_config.json --type common.Config --output␣
↪modified_config.pb

configtxlator compute_update --channel_id $CH_NAME --original config.pb --updated␣
↪modified_config.pb --output config_update.pb
```

Now that we have calculated the difference between the old config and the new one, we can apply the changes to the config.

```
configtxlator proto_decode --input config_update.pb --type common.ConfigUpdate --
↪output config_update.json

echo '{"payload":{"header":{"channel_header":{"channel_id":"'$CH_NAME'", "type":2}},
↪"data":{"config_update":'$(cat config_update.json)'}}}' | jq . > config_update_in_
↪envelope.json

configtxlator proto_encode --input config_update_in_envelope.json --type common.
↪Envelope --output config_update_in_envelope.pb
```

Submit the config update transaction:

```
peer channel update -f config_update_in_envelope.pb -c $CH_NAME -o $ORDERER_CONTAINER␣
↪--tls true --cafile $TLS_ROOT_CA
```

Our config update transaction represents the difference between the original config and the modified one, but the ordering service will translate this into a full channel config.

7.7.4 Get the Necessary Signatures

Once you've successfully generated the new configuration protobuf file, it will need to satisfy the relevant policy for whatever it is you're trying to change, typically (though not always) by requiring signatures from other organizations.

Note: you may be able to script the signature collection, dependent on your application. In general, you may always collect more signatures than are required.

The actual process of getting these signatures will depend on how you've set up your system, but there are two main implementations. Currently, the Fabric command line defaults to a "pass it along" system. That is, the Admin of the Org proposing a config update sends the update to someone else (another Admin, typically) who needs to sign it. This Admin signs it (or doesn't) and passes it along to the next Admin, and so on, until there are enough signatures for the config to be submitted.

This has the virtue of simplicity — when there are enough signatures, the last admin can simply submit the config transaction (in Fabric, the `peer channel update` command includes a signature by default). However, this process will only be practical in smaller channels, since the "pass it along" method can be time consuming.

The other option is to submit the update to every Admin on a channel and wait for enough signatures to come back. These signatures can then be stitched together and submitted. This makes life a bit more difficult for the Admin who created the config update (forcing them to deal with a file per signer) but is the recommended workflow for users which are developing Fabric management applications.

Once the config has been added to the ledger, it will be a best practice to pull it and convert it to JSON to check to make sure everything was added correctly. This will also serve as a useful copy of the latest config.

7.8 Chaincode for Developers

7.8.1 What is Chaincode?

Chaincode is a program, written in Go, Node.js, or Java that implements a prescribed interface. Chaincode runs in a seperate process from the peer and initializes and manages the ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it similar to a "smart contract". A chaincode can be invoked to update or query the ledger in a proposal transaction. Given the appropriate permission, a chaincode may invoke another chaincode, either in the same channel or in different channels, to access its state. Note that, if the called chaincode is on a different channel from the calling chaincode, only read query is allowed. That is, the called chaincode on a different channel is only a `Query`, which does not participate in state validation checks in subsequent commit phase.

In the following sections, we will explore chaincode through the eyes of an application developer. We'll present a simple chaincode sample application and walk through the purpose of each method in the Chaincode Shim API. If you are a network operator who is deploying a chaincode to running network, visit the *Deploying a smart contract to a channel* tutorial and the *Fabric chaincode lifecycle* concept topic.

7.8.2 Chaincode API

Every chaincode program must implement the `Chaincode` interface whose methods are called in response to received transactions. You can find the reference documentation of the Chaincode Shim API for different languages below:

- Go
- Node.js
- Java

In each language, the `Invoke` method is called by clients to submit transaction proposals. This method allows you to use the chaincode to read and write data on the channel ledger.

You also need to include an `Init` method that will serve as the initialization function for your chaincode. This method will be called in order to initialize the chaincode when it is started or upgraded. By default, this function is never executed. However, you can use the chaincode definition to request that the `Init` function be executed. If execution of `Init` is requested, fabric will ensure that `Init` is invoked before any other function and is only invoked once. This option provides you additional control over which users can initialize the chaincode and the ability to add initial data to the ledger. If you are using the peer CLI to approve the chaincode definition, use the `--init-required` flag to request the execution of the `Init` function. Then call the `Init` function by using the *peer chaincode invoke* command and passing the `--isInit` flag. For more information, see *Fabric chaincode lifecycle*.

The other interface in the chaincode "shim" APIs is the `ChaincodeStubInterface`:

- Go
- Node.js
- Java

which is used to access and modify the ledger, and to make invocations between chaincodes.

In this tutorial using Go chaincode, we will demonstrate the use of these APIs by implementing a simple chaincode application that manages simple "assets".

7.8.3 Simple Asset Chaincode

Our application is a basic sample chaincode to create assets (key-value pairs) on the ledger.

Choosing a Location for the Code

If you haven't been doing programming in Go, you may want to make sure that you have *Go Programming Language* installed and your system properly configured.

Now, you will want to create a directory for your chaincode application as a child directory of `$GOPATH/src/`.

To keep things simple, let's use the following command:

```
mkdir -p $GOPATH/src/sacc && cd $GOPATH/src/sacc
```

Now, let's create the source file that we'll fill in with code:

```
touch sacc.go
```

Housekeeping

First, let's start with some housekeeping. As with every chaincode, it implements the Chaincode interface in particular, `Init` and `Invoke` functions. So, let's add the Go import statements for the necessary dependencies for our chaincode. We'll import the chaincode shim package and the peer protobuf package. Next, let's add a struct `SimpleAsset` as a receiver for Chaincode shim functions.

```
package main

import (
    "fmt"
```

(continues on next page)

(continued from previous page)

```
    "github.com/hyperledger/fabric-chaincode-go/shim"
    "github.com/hyperledger/fabric-protos-go/peer"
)

// SimpleAsset implements a simple chaincode to manage an asset
type SimpleAsset struct {
}
```

Initializing the Chaincode

Next, we'll implement the `Init` function.

```
// Init is called during chaincode instantiation to initialize any data.
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {

}
```

Note: Note that chaincode upgrade also calls this function. When writing a chaincode that will upgrade an existing one, make sure to modify the `Init` function appropriately. In particular, provide an empty "Init" method if there's no "migration" or nothing to be initialized as part of the upgrade.

Next, we'll retrieve the arguments to the `Init` call using the ChaincodeStubInterface.GetStringArgs function and check for validity. In our case, we are expecting a key-value pair.

```
// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data, so be careful to avoid a scenario where you
// inadvertently clobber your ledger's data!
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
  // Get the args from the transaction proposal
  args := stub.GetStringArgs()
  if len(args) != 2 {
    return shim.Error("Incorrect arguments. Expecting a key and a value")
  }
}
```

Next, now that we have established that the call is valid, we'll store the initial state in the ledger. To do this, we will call ChaincodeStubInterface.PutState with the key and value passed in as the arguments. Assuming all went well, return a peer.Response object that indicates the initialization was a success.

```
// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data, so be careful to avoid a scenario where you
// inadvertently clobber your ledger's data!
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
  // Get the args from the transaction proposal
  args := stub.GetStringArgs()
  if len(args) != 2 {
    return shim.Error("Incorrect arguments. Expecting a key and a value")
  }

  // Set up any variables or assets here by calling stub.PutState()
```

(continues on next page)

(continued from previous page)

```
  // We store the key and the value on the ledger
  err := stub.PutState(args[0], []byte(args[1]))
  if err != nil {
    return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
  }
  return shim.Success(nil)
}
```

Invoking the Chaincode

First, let's add the `Invoke` function's signature.

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The 'set'
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {

}
```

As with the `Init` function above, we need to extract the arguments from the `ChaincodeStubInterface`. The `Invoke` function's arguments will be the name of the chaincode application function to invoke. In our case, our application will simply have two functions: `set` and `get`, that allow the value of an asset to be set or its current state to be retrieved. We first call ChaincodeStubInterface.GetFunctionAndParameters to extract the function name and the parameters to that chaincode application function.

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

}
```

Next, we'll validate the function name as being either `set` or `get`, and invoke those chaincode application functions, returning an appropriate response via the `shim.Success` or `shim.Error` functions that will serialize the response into a gRPC protobuf message.

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
            result, err = set(stub, args)
    } else {
            result, err = get(stub, args)
    }
    if err != nil {
            return shim.Error(err.Error())
```

(continues on next page)

(continued from previous page)

```
    }

    // Return the result as success payload
    return shim.Success([]byte(result))
}
```

Implementing the Chaincode Application

As noted, our chaincode application implements two functions that can be invoked via the `Invoke` function. Let's implement those functions now. Note that as we mentioned above, to access the ledger's state, we will leverage the ChaincodeStubInterface.PutState and ChaincodeStubInterface.GetState functions of the chaincode shim API.

```
// Set stores the asset (both key and value) on the ledger. If the key exists,
// it will override the value with the new one
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }

    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
            return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

// Get returns the value of the specified asset key
func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
            return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0],␣
↪err)
    }
    if value == nil {
            return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}
```

Pulling it All Together

Finally, we need to add the `main` function, which will call the shim.Start function. Here's the whole chaincode program source.

```
package main

import (
    "fmt"
```

(continues on next page)

(continued from previous page)

```
    "github.com/hyperledger/fabric-chaincode-go/shim"
    "github.com/hyperledger/fabric-protos-go/peer"
)

// SimpleAsset implements a simple chaincode to manage an asset
type SimpleAsset struct {
}

// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data.
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Get the args from the transaction proposal
    args := stub.GetStringArgs()
    if len(args) != 2 {
            return shim.Error("Incorrect arguments. Expecting a key and a value")
    }

    // Set up any variables or assets here by calling stub.PutState()

    // We store the key and the value on the ledger
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
            return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
    }
    return shim.Success(nil)
}

// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
            result, err = set(stub, args)
    } else { // assume 'get' even if fn is nil
            result, err = get(stub, args)
    }
    if err != nil {
            return shim.Error(err.Error())
    }

    // Return the result as success payload
    return shim.Success([]byte(result))
}

// Set stores the asset (both key and value) on the ledger. If the key exists,
// it will override the value with the new one
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }
```

(continues on next page)

(continued from previous page)

```
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
            return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

// Get returns the value of the specified asset key
func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
            return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0],␣
↪err)
    }
    if value == nil {
            return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}

// main function starts up the chaincode in the container during instantiate
func main() {
    if err := shim.Start(new(SimpleAsset)); err != nil {
            fmt.Printf("Error starting SimpleAsset chaincode: %s", err)
    }
}
```

7.8.4 Chaincode access control

Chaincode can utilize the client (submitter) certificate for access control decisions by calling the GetCreator() function. Additionally the Go shim provides extension APIs that extract client identity from the submitter's certificate that can be used for access control decisions, whether that is based on client identity itself, or the org identity, or on a client identity attribute.

For example an asset that is represented as a key/value may include the client's identity as part of the value (for example as a JSON attribute indicating that asset owner), and only this client may be authorized to make updates to the key/value in the future. The client identity library extension APIs can be used within chaincode to retrieve this submitter information to make such access control decisions.

See the client identity (CID) library documentation for more details.

To add the client identity shim extension to your chaincode as a dependency, see *Managing external dependencies for chaincode written in Go*.

7.8.5 Managing external dependencies for chaincode written in Go

Your Go chaincode requires packages (like the chaincode shim) that are not part of the Go standard library. These packages must be included in your chaincode package.

There are many tools available for managing (or "vendoring") these dependencies. The following demonstrates how to use `govendor`:

```
govendor init
govendor add +external  // Add all external package, or
govendor add github.com/external/pkg // Add specific external package
```

This imports the external dependencies into a local `vendor` directory. If you are vendoring the Fabric shim or shim extensions, clone the Fabric repository to your $GOPATH/src/github.com/hyperledger directory, before executing the govendor commands.

Once dependencies are vendored in your chaincode directory, `peer chaincode package` and `peer chaincode install` operations will then include code associated with the dependencies into the chaincode package.

7.9 Building Your First Network

Note: The Build your first network (BYFN) tutorial has been deprecated. If you are getting started with Hyperledger Fabric and would like to deploy a basic network, see *Using the Fabric test network*. If you are deploying Fabric in production, see the guide for *Deploying a production network*.

The build your first network (BYFN) scenario provisions a sample Hyperledger Fabric network consisting of two organizations, each maintaining two peer nodes. It also will deploy a Raft ordering service by default.

7.9.1 Install prerequisites

Before we begin, if you haven't already done so, you may wish to check that you have all the *Prerequisites* installed on the platform(s) on which you'll be developing blockchain applications and/or operating Hyperledger Fabric.

You will also need to *Install Samples, Binaries, and Docker Images*. You will notice that there are a number of samples included in the `fabric-samples` repository. We will be using the `first-network` sample. Let's open that sub-directory now.

```
cd fabric-samples/first-network
```

Note: The supplied commands in this documentation **MUST** be run from your `first-network` sub-directory of the `fabric-samples` repository clone. If you elect to run the commands from a different location, the various provided scripts will be unable to find the binaries.

7.9.2 Want to run it now?

We provide a fully annotated script — `byfn.sh` — that leverages these Docker images to quickly bootstrap a Hyperledger Fabric network that by default is comprised of four peers representing two different organizations, and a Raft ordering service. It will also launch a container to run a scripted execution that will join peers to a channel, deploy a chaincode and drive execution of transactions against the deployed chaincode.

Here's the help text for the `byfn.sh` script:

```
Usage:
byfn.sh <mode> [-c <channel name>] [-t <timeout>] [-d <delay>] [-f <docker-compose-
↪file>] [-s <dbtype>] [-l <language>] [-o <consensus-type>] [-i <imagetag>] [-v]"
```

(continues on next page)

(continued from previous page)

```
  <mode> - one of 'up', 'down', 'restart', 'generate' or 'upgrade'"
    - 'up' - bring up the network with docker-compose up"
    - 'down' - clear the network with docker-compose down"
    - 'restart' - restart the network"
    - 'generate' - generate required certificates and genesis block"
    - 'upgrade'  - upgrade the network from version 1.3.x to 1.4.0"
  -c <channel name> - channel name to use (defaults to \"mychannel\")"
  -t <timeout> - CLI timeout duration in seconds (defaults to 10)"
  -d <delay> - delay duration in seconds (defaults to 3)"
  -f <docker-compose-file> - specify which docker-compose file use (defaults to␣
↪docker-compose-cli.yaml)"
  -s <dbtype> - the database backend to use: goleveldb (default) or couchdb"
  -l <language> - the chaincode language: golang (default), javascript, or java"
  -a - launch certificate authorities (no certificate authorities are launched by␣
↪default)
  -n - do not deploy chaincode (abstore chaincode is deployed by default)
  -i <imagetag> - the tag to be used to launch the network (defaults to \"latest\")"
  -v - verbose mode"
byfn.sh -h (print this message)"

Typically, one would first generate the required certificates and
genesis block, then bring up the network. e.g.:"

  byfn.sh generate -c mychannel"
  byfn.sh up -c mychannel -s couchdb"
  byfn.sh up -c mychannel -s couchdb -i 1.4.0"
  byfn.sh up -l javascript"
  byfn.sh down -c mychannel"
  byfn.sh upgrade -c mychannel"

Taking all defaults:"
      byfn.sh generate"
      byfn.sh up"
      byfn.sh down"
```

If you choose not to supply a flag, the script will use default values.

Generate Network Artifacts

Ready to give it a go? Okay then! Execute the following command:

```
./byfn.sh generate
```

You will see a brief description as to what will occur, along with a yes/no command line prompt. Respond with a `y` or hit the return key to execute the described action.

```
Generating certs and genesis block for channel 'mychannel' with CLI timeout of '10'␣
↪seconds and CLI delay of '3' seconds
Continue? [Y/n] y
proceeding ...
/Users/xxx/dev/fabric-samples/bin/cryptogen

##########################################################
##### Generate certificates using cryptogen tool #########
##########################################################
```

(continues on next page)

(continued from previous page)

```
org1.example.com
2017-06-12 21:01:37.334 EDT [bccsp] GetDefault -> WARN 001 Before using BCCSP, please␣
↪call InitFactories(). Falling back to bootBCCSP.
...

/Users/xxx/dev/fabric-samples/bin/configtxgen
##########################################################
#########  Generating Orderer Genesis block ##############
##########################################################
2017-06-12 21:01:37.558 EDT [common/configtx/tool] main -> INFO 001 Loading␣
↪configuration
2017-06-12 21:01:37.562 EDT [msp] getMspConfig -> INFO 002 intermediate certs folder␣
↪not found at [/Users/xxx/dev/byfn/crypto-config/ordererOrganizations/example.com/
↪msp/intermediatecerts]. Skipping.: [stat /Users/xxx/dev/byfn/crypto-config/
↪ordererOrganizations/example.com/msp/intermediatecerts: no such file or directory]
...
2017-06-12 21:01:37.588 EDT [common/configtx/tool] doOutputBlock -> INFO 00b␣
↪Generating genesis block
2017-06-12 21:01:37.590 EDT [common/configtx/tool] doOutputBlock -> INFO 00c Writing␣
↪genesis block

#################################################################
### Generating channel configuration transaction 'channel.tx' ###
#################################################################
2017-06-12 21:01:37.634 EDT [common/configtx/tool] main -> INFO 001 Loading␣
↪configuration
2017-06-12 21:01:37.644 EDT [common/configtx/tool] doOutputChannelCreateTx -> INFO␣
↪002 Generating new channel configtx
2017-06-12 21:01:37.645 EDT [common/configtx/tool] doOutputChannelCreateTx -> INFO␣
↪003 Writing new channel tx

#################################################################
#######    Generating anchor peer update for Org1MSP   ##########
#################################################################
2017-06-12 21:01:37.674 EDT [common/configtx/tool] main -> INFO 001 Loading␣
↪configuration
2017-06-12 21:01:37.678 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO␣
↪002 Generating anchor peer update
2017-06-12 21:01:37.679 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO␣
↪003 Writing anchor peer update

#################################################################
#######    Generating anchor peer update for Org2MSP   ##########
#################################################################
2017-06-12 21:01:37.700 EDT [common/configtx/tool] main -> INFO 001 Loading␣
↪configuration
2017-06-12 21:01:37.704 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO␣
↪002 Generating anchor peer update
2017-06-12 21:01:37.704 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO␣
↪003 Writing anchor peer update
```

This first step generates all of the certificates and keys for our various network entities, the `genesis block` used to bootstrap the ordering service, and a collection of configuration transactions required to configure a *Channel*.

Bring Up the Network

Next, you can bring the network up with one of the following commands:

```
./byfn.sh up
```

The above command will compile Go chaincode images and spin up the corresponding containers. Go is the default chaincode language, however there is also support for Node.js and Java chaincode. If you'd like to run through this tutorial with node chaincode, pass the following command instead:

```
# we use the -l flag to specify the chaincode language
# forgoing the -l flag will default to "golang"

./byfn.sh up -l javascript
```

Note: For more information on the Node.js shim, please refer to its documentation.

Note: For more information on the Java shim, please refer to its documentation.

o make the sample run with Java chaincode, you have to specify `-l java` as follows:

```
./byfn.sh up -l java
```

Note: Do not run both of these commands. Only one language can be tried unless you bring down and recreate the network between.

You will be prompted as to whether you wish to continue or abort. Respond with a `y` or hit the return key:

```
Starting for channel 'mychannel' with CLI timeout of '10' seconds and CLI delay of '3
↪' seconds
Continue? [Y/n]
proceeding ...
Creating network "net_byfn" with the default driver
Creating peer0.org1.example.com
Creating peer1.org1.example.com
Creating peer0.org2.example.com
Creating orderer.example.com
Creating peer1.org2.example.com
Creating cli



 ____    _____      _      ____    _____
/ ___|  |_   _|    / \    |  _ \  |_   _|
\___ \    | |     / _ \   | |_) |   | |
 ___) |   | |    / ___ \  |  _ <    | |
|____/    |_|   /_/   \_\ |_| \_\   |_|

Channel name : mychannel
Creating channel...
```

The logs will continue from there. This will launch all of the containers, and then drive a complete end-to-end application scenario. Upon successful completion, it should report the following in your terminal window:

```
Query Result: 90
2017-05-16 17:08:15.158 UTC [main] main -> INFO 008 Exiting.....
===================== Query successful on peer1.org2 on channel 'mychannel'␣
↪=====================

===================== All GOOD, BYFN execution completed =====================


 _____   _   _   ____
| ____| | \ | | |  _ \
|  _|   |  \| | | | | |
| |___  | |\  | | |_| |
|_____| |_| \_| |____/
```

You can scroll through these logs to see the various transactions. If you don't get this result, then jump down to the *Troubleshooting* section and let's see whether we can help you discover what went wrong.

Bring Down the Network

Finally, let's bring it all down so we can explore the network setup one step at a time. The following will kill your containers, remove the crypto material and four artifacts, and delete the chaincode images from your Docker Registry:

```
./byfn.sh down
```

Once again, you will be prompted to continue, respond with a `y` or hit the return key:

```
Stopping with channel 'mychannel' and CLI timeout of '10'
Continue? [Y/n] y
proceeding ...
WARNING: The CHANNEL_NAME variable is not set. Defaulting to a blank string.
WARNING: The TIMEOUT variable is not set. Defaulting to a blank string.
Removing network net_byfn
468aaa6201ed
...
Untagged: dev-peer1.org2.example.com-mycc-1.0:latest
Deleted: sha256:ed3230614e64e1c83e510c0c282e982d2b06d148b1c498bbdcc429e2b2531e91
...
```

If you'd like to learn more about the underlying tooling and bootstrap mechanics, continue reading. In these next sections we'll walk through the various steps and requirements to build a fully-functional Hyperledger Fabric network.

Note: The manual steps outlined below assume that the `FABRIC_LOGGING_SPEC` in the `cli` container is set to `DEBUG`. You can set this by modifying the `docker-compose-cli.yaml` file in the `first-network` directory. e.g.

```
cli:
  container_name: cli
  image: hyperledger/fabric-tools:$IMAGE_TAG
  tty: true
  stdin_open: true
  environment:
    - GOPATH=/opt/gopath
    - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
    - FABRIC_LOGGING_SPEC=DEBUG
    #- FABRIC_LOGGING_SPEC=INFO
```

7.9.3 Crypto Generator

We will use the `cryptogen` tool to generate the cryptographic material (x509 certs and signing keys) for our various network entities. These certificates are representative of identities, and they allow for sign/verify authentication to take place as our entities communicate and transact.

How does it work?

Cryptogen consumes a file — `crypto-config.yaml` — that contains the network topology and allows us to generate a set of certificates and keys for both the Organizations and the components that belong to those Organizations. Each Organization is provisioned a unique root certificate (`ca-cert`) that binds specific components (peers and orderers) to that Org. By assigning each Organization a unique CA certificate, we are mimicking a typical network where a participating *Member* would use its own Certificate Authority. Transactions and communications within Hyperledger Fabric are signed by an entity's private key (`keystore`), and then verified by means of a public key (`signcerts`).

You will notice a `count` variable within this file. We use this to specify the number of peers per Organization; in our case there are two peers per Org. We won't delve into the minutiae of x.509 certificates and public key infrastructure right now. If you're interested, you can peruse these topics on your own time.

After we run the `cryptogen` tool, the generated certificates and keys will be saved to a folder titled `crypto-config`. Note that the `crypto-config.yaml` file lists five orderers as being tied to the orderer organization. While the `cryptogen` tool will create certificates for all five of these orderers. These orderers will be used in a etcdraft ordering service implementation and be used to create the system channel and `mychannel`.

7.9.4 Configuration Transaction Generator

The `configtxgen` tool is used to create four configuration artifacts:

- orderer `genesis block`,
- channel `configuration transaction`,
- and two `anchor peer transactions` - one for each Peer Org.

Please see *configtxgen* for a complete description of this tool's functionality.

The orderer block is the *Genesis Block* for the ordering service, and the channel configuration transaction file is broadcast to the orderer at *Channel* creation time. The anchor peer transactions, as the name might suggest, specify each Org's *Anchor Peer* on this channel.

How does it work?

Configtxgen consumes a file - `configtx.yaml` - that contains the definitions for the sample network. There are three members - one Orderer Org (`OrdererOrg`) and two Peer Orgs (`Org1` & `Org2`) each managing and maintaining two peer nodes. This file also specifies a consortium - `SampleConsortium` - consisting of our two Peer Orgs. Pay specific attention to the "Profiles" section at the bottom of this file. You will notice that we have several unique profiles. A few are worth noting:

- `SampleMultiNodeEtcdRaft`: generates the genesis block for a Raft ordering service. Only used if you issue the `-o` flag and specify `etcdraft`.
- `TwoOrgsChannel`: generates the genesis block for our channel, `mychannel`.

These headers are important, as we will pass them in as arguments when we create our artifacts.

Note: Notice that our `SampleConsortium` is defined in the system-level profile and then referenced by our channel-level profile. Channels exist within the purview of a consortium, and all consortia must be defined in the scope of the network at large.

This file also contains two additional specifications that are worth noting. Firstly, we specify the anchor peers for each Peer Org (`peer0.org1.example.com` & `peer0.org2.example.com`). Secondly, we point to the location of the MSP directory for each member, in turn allowing us to store the root certificates for each Org in the orderer genesis block. This is a critical concept. Now any network entity communicating with the ordering service can have its digital signature verified.

7.9.5 Run the tools

You can manually generate the certificates/keys and the various configuration artifacts using the `configtxgen` and `cryptogen` commands. Alternately, you could try to adapt the byfn.sh script to accomplish your objectives.

Manually generate the artifacts

You can refer to the `generateCerts` function in the byfn.sh script for the commands necessary to generate the certificates that will be used for your network configuration as defined in the `crypto-config.yaml` file. However, for the sake of convenience, we will also provide a reference here.

First let's run the `cryptogen` tool. Our binary is in the `bin` directory, so we need to provide the relative path to where the tool resides.

```
../bin/cryptogen generate --config=./crypto-config.yaml
```

You should see the following in your terminal:

```
org1.example.com
org2.example.com
```

The certs and keys (i.e. the MSP material) will be output into a directory - `crypto-config` - at the root of the `first-network` directory.

Next, we need to tell the `configtxgen` tool where to look for the `configtx.yaml` file that it needs to ingest. We will tell it look in our present working directory:

```
export FABRIC_CFG_PATH=$PWD
```

Then, we'll invoke the `configtxgen` tool to create the orderer genesis block:

```
../bin/configtxgen -profile SampleMultiNodeEtcdRaft -channelID byfn-sys-channel -
↪outputBlock ./channel-artifacts/genesis.block
```

Note: The orderer genesis block and the subsequent artifacts we are about to create will be output into the `channel-artifacts` directory at the root of the `first-network` directory. The *channelID* in the above command is the name of the system channel.

Create a Channel Configuration Transaction

Next, we need to create the channel transaction artifact. Be sure to replace `$CHANNEL_NAME` or set `CHANNEL_NAME` as an environment variable that can be used throughout these instructions:

```
# The channel.tx artifact contains the definitions for our sample channel

export CHANNEL_NAME=mychannel  && ../bin/configtxgen -profile TwoOrgsChannel -
↪outputCreateChannelTx ./channel-artifacts/channel.tx -channelID $CHANNEL_NAME
```

Note that the `TwoOrgsChannel` profile will use the ordering service configuration you specified when creating the genesis block for the network.

You should see an output similar to the following in your terminal:

```
2017-10-26 19:24:05.324 EDT [common/tools/configtxgen] main -> INFO 001 Loading␣
↪configuration
2017-10-26 19:24:05.329 EDT [common/tools/configtxgen] doOutputChannelCreateTx ->␣
↪INFO 002 Generating new channel configtx
2017-10-26 19:24:05.329 EDT [common/tools/configtxgen] doOutputChannelCreateTx ->␣
↪INFO 003 Writing new channel tx
```

Next, we will define the anchor peer for Org1 on the channel that we are constructing. Again, be sure to replace `$CHANNEL_NAME` or set the environment variable for the following commands. The terminal output will mimic that of the channel transaction artifact:

```
../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-
↪artifacts/Org1MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org1MSP
```

Now, we will define the anchor peer for Org2 on the same channel:

```
../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-
↪artifacts/Org2MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org2MSP
```

7.9.6 Start the network

Note: If you ran the `byfn.sh` example above previously, be sure that you have brought down the test network before you proceed (see *Bring Down the Network*).

We will leverage a script to spin up our network. The docker-compose file references the images that we have previously downloaded, and bootstraps the orderer with our previously generated `genesis.block`.

We want to go through the commands manually in order to expose the syntax and functionality of each call.

First let's start our network:

```
docker-compose -f docker-compose-cli.yaml -f docker-compose-etcdraft2.yaml up -d
```

If you want to see the realtime logs for your network, then do not supply the `-d` flag. If you let the logs stream, then you will need to open a second terminal to execute the CLI calls.

Create & Join Channel

Recall that we created the channel configuration transaction using the `configtxgen` tool in the *Create a Channel Configuration Transaction* section, above. You can repeat that process to create additional channel configuration transactions, using the same or different profiles in the `configtx.yaml` that you pass to the `configtxgen` tool. Then you can repeat the process defined in this section to establish those other channels in your network.

We will enter the CLI container using the `docker exec` command:

```
docker exec -it cli bash
```

If successful you should see the following:

```
bash-5.0#
```

For the following CLI commands against `peer0.org1.example.com` to work, we need to preface our commands with the four environment variables given below. These variables for `peer0.org1.example.com` are baked into the CLI container, therefore we can operate without passing them. **HOWEVER**, if you want to send calls to other peers or the orderer, keep the CLI container defaults targeting `peer0.org1.example.com`, but override the environment variables as seen in the example below when you make any CLI calls:

```
# Environment variables for PEER0

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
CORE_PEER_LOCALMSPID="Org1MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

Next, we are going to pass in the generated channel configuration transaction artifact that we created in the *Create a Channel Configuration Transaction* section (we called it `channel.tx`) to the orderer as part of the create channel request.

We specify our channel name with the `-c` flag and our channel configuration transaction with the `-f` flag. In this case it is `channel.tx`, however you can mount your own configuration transaction with a different name. Once again we will set the `CHANNEL_NAME` environment variable within our CLI container so that we don't have to explicitly pass this argument. Channel names must be all lower case, less than 250 characters long and match the regular expression `[a-z][a-z0-9.-]*`.

```
export CHANNEL_NAME=mychannel

# the channel.tx file is mounted in the channel-artifacts directory within your CLI␣
↪container
# as a result, we pass the full path for the file
# we also pass the path for the orderer ca-cert in order to verify the TLS handshake
# be sure to export or replace the $CHANNEL_NAME variable appropriately

peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-
↪artifacts/channel.tx --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/
↪peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem
```

Note: Notice the `--cafile` that we pass as part of this command. It is the local path to the orderer's root cert, allowing us to verify the TLS handshake.

This command returns a genesis block - `<CHANNEL_NAME.block>` - which we will use to join the channel. It contains the configuration information specified in `channel.tx` If you have not made any modifications to the default channel name, then the command will return you a proto titled `mychannel.block`.

Note: You will remain in the CLI container for the remainder of these manual commands. You must also remember to preface all commands with the corresponding environment variables when targeting a peer other than `peer0.org1.example.com`.

Now let's join `peer0.org1.example.com` to the channel.

```
# By default, this joins ``peer0.org1.example.com`` only
# the <CHANNEL_NAME.block> was returned by the previous command
# if you have not modified the channel name, you will join with mychannel.block
# if you have created a different channel name, then pass in the appropriately named␣
↪block

 peer channel join -b mychannel.block
```

You can make other peers join the channel as necessary by making appropriate changes in the four environment variables we used in the *Create & Join Channel* section, above.

Rather than join every peer, we will simply join `peer0.org2.example.com` so that we can properly update the anchor peer definitions in our channel. Since we are overriding the default environment variables baked into the CLI container, this full command will be the following:

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp CORE_PEER_
↪ADDRESS=peer0.org2.example.com:9051 CORE_PEER_LOCALMSPID="Org2MSP" CORE_PEER_TLS_
↪ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt peer␣
↪channel join -b mychannel.block
```

Alternatively, you could choose to set these environment variables individually rather than passing in the entire string. Once they've been set, you simply need to issue the `peer channel join` command again and the CLI container will act on behalf of `peer0.org2.example.com`.

Update the anchor peers

The following commands are channel updates and they will propagate to the definition of the channel. In essence, we adding additional configuration information on top of the channel's genesis block. Note that we are not modifying the genesis block, but simply adding deltas into the chain that will define the anchor peers.

Update the channel definition to define the anchor peer for Org1 as `peer0.org1.example.com`:

```
peer channel update -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-
↪artifacts/Org1MSPanchors.tx --tls --cafile /opt/gopath/src/github.com/hyperledger/
↪fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/
↪msp/tlscacerts/tlsca.example.com-cert.pem
```

Now update the channel definition to define the anchor peer for Org2 as `peer0.org2.example.com`. Identically to the `peer channel join` command for the Org2 peer, we will need to preface this call with the appropriate environment variables.

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp CORE_PEER_
↪ADDRESS=peer0.org2.example.com:9051 CORE_PEER_LOCALMSPID="Org2MSP" CORE_PEER_TLS_
↪ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt peer␣
↪channel update -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-artifacts/
↪Org2MSPanchors.tx --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/
↪crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem
```

(continues on next page)

(continued from previous page)

Install and define a chaincode

Note: We will utilize a simple existing chaincode. To learn how to write your own chaincode, see the *Chaincode for Developers* tutorial.

Note: These instructions use the Fabric chaincode lifecycle introduced in the v2.0 release. If you would like to use the previous lifecycle to install and instantiate a chaincode, visit the v1.4 version of the Building your first network tutorial.

Applications interact with the blockchain ledger through `chaincode`. Therefore we need to install a chaincode on every peer that will execute and endorse our transactions. However, before we can interact with our chaincode, the members of the channel need to agree on a chaincode definition that establishes chaincode governance.

We need to package the chaincode before it can be installed on our peers. For each package you create, you need to provide a chaincode package label as a description of the chaincode. Use the following commands to package a sample Go, Node.js or Java chaincode.

Go

```
# before packaging Go chaincode, vendoring Go dependencies is required like the␣
↪following commands.
cd /opt/gopath/src/github.com/hyperledger/fabric-samples/chaincode/abstore/go
GO111MODULE=on go mod vendor
cd -

# this packages a Go chaincode.
# make note of the --lang flag to indicate "golang" chaincode
# for Go chaincode --path takes the relative path from $GOPATH/src
# The --label flag is used to create the package label
peer lifecycle chaincode package mycc.tar.gz --path github.com/hyperledger/fabric-
↪samples/chaincode/abstore/go/ --lang golang --label mycc_1
```

Node.js

```
# this packages a Node.js chaincode
# make note of the --lang flag to indicate "node" chaincode
# for node chaincode --path takes the absolute path to the Node.js chaincode
# The --label flag is used to create the package label
peer lifecycle chaincode package mycc.tar.gz --path /opt/gopath/src/github.com/
↪hyperledger/fabric-samples/chaincode/abstore/javascript/ --lang node --label mycc_1
```

Java

```
# this packages a java chaincode
# make note of the --lang flag to indicate "java" chaincode
# for java chaincode --path takes the absolute path to the Java chaincode
# The --label flag is used to create the package label
peer lifecycle chaincode package mycc.tar.gz --path /opt/gopath/src/github.com/
↪hyperledger/fabric-samples/chaincode/abstore/java/ --lang java --label mycc_1
```

Each of the above commands will create a chaincode package named `mycc.tar.gz`, which we can use to install the chaincode on our peers. Issue the following command to install the package on peer0 of Org1.

```
# this command installs a chaincode package on your peer
peer lifecycle chaincode install mycc.tar.gz
```

A successful install command will return a chaincode package identifier. You should see output similar to the following:

```
2019-03-13 13:48:53.691 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪001 Installed remotely: response:<status:200 payload:"\nEmycc_
↪1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173" >
2019-03-13 13:48:53.691 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪002 Chaincode code package identifier: mycc_
↪1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173
```

You can also find the chaincode package identifier by querying your peer for information about the packages you have installed.

```
# this returns the details of the chaincode packages installed on your peers
peer lifecycle chaincode queryinstalled
```

The command above will return the same package identifier as the install command. You should see output similar to the following:

```
Get installed chaincodes on peer:
Package ID: mycc_1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173,␣
↪Label: mycc_1
```

We are going to need the package ID for future commands, so let's go ahead and save it as an environment variable. Paste the package ID returned by the *peer lifecycle chaincode queryinstalled* command into the command below. The package ID may not be the same for all users, so you need to complete this step using the package ID returned from your console.

```
# Save the package ID as an environment variable.

CC_PACKAGE_ID=mycc_1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173
```

The endorsement policy of `mycc` will be set to require endorsements from a peer in both Org1 and Org2. Therefore, we also need to install the chaincode on a peer in Org2.

Modify the following four environment variables to issue the install command as Org2:

```
# Environment variables for PEER0 in Org2

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer0.org2.example.com:9051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

Now install the chaincode package onto peer0 of Org2. The following command will install the chaincode and return same identifier as the install command we issued as Org1.

```
# this installs a chaincode package on your peer
peer lifecycle chaincode install mycc.tar.gz
```

After you install the package, you need to approve a chaincode definition for your organization. The chaincode definition includes the important parameters of chaincode governance, including the chaincode name and version. The definition also includes the package identifier used to associate the chaincode package installed on your peers with a chaincode definition approved by your organization.

Because we set the environment variables to operate as Org2, we can use the following command to approve a definition of the `mycc` chaincode for Org2. The approval is distributed to peers within each organization, so the command does not need to target every peer within an organization.

```
# this approves a chaincode definition for your org
# make note of the --package-id flag that provides the package ID
# use the --init-required flag to request the ``Init`` function be invoked to␣
↪initialize the chaincode
peer lifecycle chaincode approveformyorg --channelID $CHANNEL_NAME --name mycc --
↪version 1.0 --init-required --package-id $CC_PACKAGE_ID --sequence 1 --tls true --
↪cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪example.com-cert.pem
```

We could have provided a `--signature-policy` or `--channel-config-policy` argument to the command above to set the chaincode endorsement policy. The endorsement policy specifies how many peers belonging to different channel members need to validate a transaction against a given chaincode. Because we did not set a policy, the definition of `mycc` will use the default endorsement policy, which requires that a transaction be endorsed by a majority of channel members present when the transaction is submitted. This implies that if new organizations are added to or removed from the channel, the endorsement policy is updated automatically to require more or fewer endorsements. In this tutorial, the default policy will require an endorsement from a peer belonging to Org1 **AND** Org2 (i.e. two endorsements). See the *Endorsement policies* documentation for more details on policy implementation.

All organizations need to agree on the definition before they can use the chaincode. Modify the following four environment variables to operate as Org1:

```
# Environment variables for PEER0

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
CORE_PEER_LOCALMSPID="Org1MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

You can now approve a definition for the `mycc` chaincode as Org1. Chaincode is approved at the organization level. You can issue the command once even if you have multiple peers.

```
# this defines a chaincode for your org
# make note of the --package-id flag that provides the package ID
# use the --init-required flag to request the Init function be invoked to initialize␣
↪the chaincode
peer lifecycle chaincode approveformyorg --channelID $CHANNEL_NAME --name mycc --
↪version 1.0 --init-required --package-id $CC_PACKAGE_ID --sequence 1 --tls true --
↪cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪example.com-cert.pem
```

Once a sufficient number of channel members have approved a chaincode definition, one member can commit the definition to the channel. By default a majority of channel members need to approve a definition before it can be committed. It is possible to check whether the chaincode definition is ready to be committed and view the current approvals by organization by issuing the following query:

```
# the flags used for this command are identical to those used for approveformyorg
# except for --package-id which is not required since it is not stored as part of
# the definition
peer lifecycle chaincode checkcommitreadiness --channelID $CHANNEL_NAME --name mycc --
↪version 1.0 --init-required --sequence 1 --tls true --cafile /opt/gopath/src/github.
↪com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem --output json
```

The command will produce as output a JSON map showing if the organizations in the channel have approved the chaincode definition provided in the checkcommitreadiness command. In this case, given that both organizations have approved, we obtain:

```
{
        "Approvals": {
                "Org1MSP": true,
                "Org2MSP": true
        }
}
```

Since both channel members have approved the definition, we can now commit it to the channel using the following command. You can issue this command as either Org1 or Org2. Note that the transaction targets peers in Org1 and Org2 to collect endorsements.

```
# this commits the chaincode definition to the channel
peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID $CHANNEL_NAME␣
↪--name mycc --version 1.0 --sequence 1 --init-required --tls true --cafile /opt/
↪gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.
↪com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem --
↪peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/src/github.
↪com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.
↪org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --
↪tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

Invoking the chaincode

After a chaincode definition has been committed to a channel, we are ready to invoke the chaincode and start interacting with the ledger. We requested the execution of the `Init` function in the chaincode definition using the `--init-required` flag. As a result, we need to pass the `--isInit` flag to its first invocation and supply the arguments to the `Init` function. Issue the following command to initialize the chaincode and put the initial data on the ledger.

```
# be sure to set the -C and -n flags appropriately
# use the --isInit flag if you are invoking an Init function
peer chaincode invoke -o orderer.example.com:7050 --isInit --tls true --cafile /opt/
↪gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.
↪com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C
↪$CHANNEL_NAME -n mycc --peerAddresses peer0.org1.example.com:7051 --
↪tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
↪peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles /opt/gopath/src/github.
↪com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.
↪org2.example.com/tls/ca.crt -c '{"Args":["Init","a","100","b","100"]}' --
↪waitForEvent
```

The first invoke will start the chaincode container. We may need to wait for the container to start. Node.js images will take longer.

Query

Let's query the chaincode to make sure that the container was properly started and the state DB was populated. The syntax for query is as follows:

```
# be sure to set the -C and -n flags appropriately

peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

Invoke

Now let's move `10` from `a` to `b`. This transaction will cut a new block and update the state DB. The syntax for invoke is as follows:

```
# be sure to set the -C and -n flags appropriately
peer chaincode invoke -o orderer.example.com:7050 --tls true --cafile /opt/gopath/src/
↪github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n␣
↪mycc --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/src/
↪github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/
↪peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --
↪tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"Args":["invoke","a","b","10"]}' --waitForEvent
```

Query

Let's confirm that our previous invocation executed properly. We initialized the key `a` with a value of `100` and just removed `10` with our previous invocation. Therefore, a query against `a` should return `90`. The syntax for query is as follows.

```
# be sure to set the -C and -n flags appropriately

peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

We should see the following:

```
Query Result: 90
```

Install the chaincode on an additional peer

If you want additional peers to interact with the ledger, then you will need to join them to the channel and install the same chaincode package on the peers. You only need to approve the chaincode definition once from your organization. A chaincode container will be launched for each peer as soon as they try to interact with that specific chaincode. Again, be cognizant of the fact that the Node.js images will be slower to build and start upon the first invoke.

We will install the chaincode on a third peer, peer1 in Org2. Modify the following four environment variables to issue the install command against peer1 in Org2:

```
# Environment variables for PEER1 in Org2

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer1.org2.example.com:10051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
```

Now install the `mycc` package on peer1 of Org2:

```
# this command installs a chaincode package on your peer
peer lifecycle chaincode install mycc.tar.gz
```

Query

Let's confirm that we can issue the query to Peer1 in Org2. We initialized the key `a` with a value of `100` and just removed `10` with our previous invocation. Therefore, a query against `a` should still return `90`.

Peer1 in Org2 must first join the channel before it can respond to queries. The channel can be joined by issuing the following command:

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp CORE_PEER_
↪ADDRESS=peer1.org2.example.com:10051 CORE_PEER_LOCALMSPID="Org2MSP" CORE_PEER_TLS_
↪ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt peer␣
↪channel join -b mychannel.block
```

After the join command returns, the query can be issued. The syntax for query is as follows.

```
# be sure to set the -C and -n flags appropriately

peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

We should see the following:

```
Query Result: 90
```

If you received an error, it may be because it takes a few seconds for the peer to join and catch up to the current blockchain height. You may re-query as needed. Feel free to perform additional invokes as well.

What's happening behind the scenes?

Note: These steps describe the scenario in which `script.sh` is run by './byfn.sh up'. Clean your network with `./byfn.sh down` and ensure this command is active. Then use the same docker-compose prompt to launch your network again

- A script - `script.sh` - is baked inside the CLI container. The script drives the `createChannel` command against the supplied channel name and uses the channel.tx file for channel configuration.
- The output of `createChannel` is a genesis block - `<your_channel_name>.block` - which gets stored on the peers' file systems and contains the channel configuration specified from channel.tx.

- The `joinChannel` command is exercised for all four peers, which takes as input the previously generated genesis block. This command instructs the peers to join `<your_channel_name>` and create a chain starting with `<your_channel_name>.block`.
- Now we have a channel consisting of four peers, and two organizations. This is our `TwoOrgsChannel` profile.
- `peer0.org1.example.com` and `peer1.org1.example.com` belong to Org1; `peer0.org2.example.com` and `peer1.org2.example.com` belong to Org2
- These relationships are defined through the `crypto-config.yaml` and the MSP path is specified in our docker compose.
- The anchor peers for Org1MSP (`peer0.org1.example.com`) and Org2MSP (`peer0.org2.example.com`) are then updated. We do this by passing the `Org1MSPanchors.tx` and `Org2MSPanchors.tx` artifacts to the ordering service along with the name of our channel.
- A chaincode - **abstore** - is packaged and installed on `peer0.org1.example.com` and `peer0.org2.example.com`
- The chaincode is then separately approved by Org1 and Org2, and then committed on the channel. Since an endorsement policy was not specified, the channel's default endorsement policy of a majority of organizations will get utilized, meaning that any transaction must be endorsed by a peer tied to Org1 and Org2.
- The chaincode Init is then called which starts the container for the target peer, and initializes the key value pairs associated with the chaincode. The initial values for this example are ["a","100" "b","200"]. This first invoke results in a container by the name of `dev-peer0.org2.example.com-mycc-1.0` starting.
- A query against the value of "a" is issued to `peer0.org2.example.com`. A container for Org2 peer0 by the name of `dev-peer0.org2.example.com-mycc-1.0` was started when the chaincode was initialized. The result of the query is returned. No write operations have occurred, so a query against "a" will still return a value of "100".
- An invoke is sent to `peer0.org1.example.com` and `peer0.org2.example.com` to move "10" from "a" to "b"
- A query is sent to `peer0.org2.example.com` for the value of "a". A value of 90 is returned, correctly reflecting the previous transaction during which the value for key "a" was modified by 10.
- The chaincode - **abstore** - is installed on `peer1.org2.example.com`
- A query is sent to `peer1.org2.example.com` for the value of "a". This starts a third chaincode container by the name of `dev-peer1.org2.example.com-mycc-1.0`. A value of 90 is returned, correctly reflecting the previous transaction during which the value for key "a" was modified by 10.

What does this demonstrate?

Chaincode **MUST** be installed on a peer in order for it to successfully perform read/write operations against the ledger. Furthermore, a chaincode container is not started for a peer until an `init` or traditional transaction - read/write - is performed against that chaincode (e.g. query for the value of "a"). The transaction causes the container to start. Also, all peers in a channel maintain an exact copy of the ledger which comprises the blockchain to store the immutable, sequenced record in blocks, as well as a state database to maintain a snapshot of the current state. This includes those peers that do not have chaincode installed on them (like `peer1.org1.example.com` in the above example) . Finally, the chaincode is accessible after it is installed (like `peer1.org2.example.com` in the above example) because its definition has already been committed on the channel.

How do I see these transactions?

Check the logs for the CLI Docker container.

```
docker logs -f cli
```

You should see the following output:

```
2017-05-16 17:08:01.366 UTC [msp] GetLocalMSP -> DEBU 004 Returning existing local MSP
2017-05-16 17:08:01.366 UTC [msp] GetDefaultSigningIdentity -> DEBU 005 Obtaining␣
↪default signing identity
2017-05-16 17:08:01.366 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext:␣
↪0AB1070A6708031A0C08F1E3ECC80510...6D7963631A0A0A0571756572790A0161
2017-05-16 17:08:01.367 UTC [msp/identity] Sign -> DEBU 007 Sign: digest:␣
↪E61DB37F4E8B0D32C9FE10E3936BA9B8CD278FAA1F3320B08712164248285C54
Query Result: 90
2017-05-16 17:08:15.158 UTC [main] main -> INFO 008 Exiting.....
===================== Query successful on peer1.org2 on channel 'mychannel'␣
↪=====================

===================== All GOOD, BYFN execution completed =====================



 _____   _   _   ____
| ____| | \ | | |  _ \
|  _|   |  \| | | | | |
| |___  | |\  | | |_| |
|_____| |_| \_| |____/
```

You can scroll through these logs to see the various transactions.

How can I see the chaincode logs?

You can inspect the individual chaincode containers to see the separate transactions executed against each container. Use the following command to find the list of running containers to find your chaincode containers:

```
$ docker ps -a
CONTAINER ID        IMAGE                                                                  ␣
↪                                                                                          ␣
↪                 COMMAND                  CREATED              STATUS              ␣
↪PORTS                                 NAMES
7aa7d9e199f5        dev-peer1.org2.example.com-mycc_1-
↪27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8-
↪2eba360c66609a3ba78327c2c86bc3abf041c78f5a35553191a1acf1efdd5a0d   "chaincode -peer.
↪add..."   About a minute ago   Up About a minute                                          ␣
↪    dev-peer1.org2.example.com-mycc_1-
↪27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8
82ce129c0fe6        dev-peer0.org2.example.com-mycc_1-
↪27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8-
↪1297906045aa77086daba21aba47e8eef359f9498b7cb2b010dff3e2a354565a   "chaincode -peer.
↪add..."   About a minute ago   Up About a minute                                          ␣
↪    dev-peer0.org2.example.com-mycc_1-
↪27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8
eaef1a8f7acf        dev-peer0.org1.example.com-mycc_1-
↪27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8-
↪00d8dbefd85a4aeb9428b7df95df9744be1325b2a60900ac7a81796e67e4280a   "chaincode -peer.
↪add..."   2 minutes ago        Up 2 minutes                                               ␣
↪    dev-peer0.org1.example.com-mycc_1-
↪27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8
da403175b785        hyperledger/fabric-tools:latest                                        ␣
↪
↪                 "/bin/bash"              4 minutes ago        Up 4 minutes              ␣
↪                                   cli
```

(continues on next page)

(continued from previous page)

```
c62a8d03818f        hyperledger/fabric-peer:latest                                          ␣
↪                                                                                           ␣
↪                "peer node start"        4 minutes ago        Up 4 minutes              ␣
↪7051/tcp, 0.0.0.0:9051->9051/tcp     peer0.org2.example.com
06593c4f3e53        hyperledger/fabric-peer:latest                                          ␣
↪                                                                                           ␣
↪                "peer node start"        4 minutes ago        Up 4 minutes        0.
↪0.0.0:7051->7051/tcp               peer0.org1.example.com
4ddc928ebffe        hyperledger/fabric-orderer:latest                                       ␣
↪                                                                                           ␣
↪                "orderer"                4 minutes ago        Up 4 minutes        0.
↪0.0.0:7050->7050/tcp               orderer.example.com
6d79e95ec059        hyperledger/fabric-peer:latest                                          ␣
↪                                                                                           ␣
↪                "peer node start"        4 minutes ago        Up 4 minutes              ␣
↪7051/tcp, 0.0.0.0:10051->10051/tcp   peer1.org2.example.com
6aad6b40fd30        hyperledger/fabric-peer:latest                                          ␣
↪                                                                                           ␣
↪                "peer node start"        4 minutes ago        Up 4 minutes              ␣
↪7051/tcp, 0.0.0.0:8051->8051/tcp     peer1.org1.example.com
```

The chaincode containers are the images starting with *dev-peer*. You can then use the container ID to find the logs from each chaincode container.

```
$ docker logs 7aa7d9e199f5
ABstore Init
Aval = 100, Bval = 100
ABstore Invoke
Aval = 90, Bval = 110

$ docker logs eaef1a8f7acf
ABstore Init
Aval = 100, Bval = 100
ABstore Invoke
Query Response:{"Name":"a","Amount":"100"}
ABstore Invoke
Aval = 90, Bval = 110
ABstore Invoke
Query Response:{"Name":"a","Amount":"90"}
```

You can also see the peer logs to view chaincode invoke messages and block commit messages:

```
$ docker logs peer0.org1.example.com
```

7.9.7 Understanding the Docker Compose topology

The BYFN sample offers us two flavors of Docker Compose files, both of which are extended from the `docker-compose-base.yaml` (located in the `base` folder). Our first flavor, `docker-compose-cli.yaml`, provides us with a CLI container, along with an orderer, four peers. We use this file for the entirety of the instructions on this page.

Note: the remainder of this section covers a docker-compose file designed for the SDK. Refer to the Node SDK repo for details on running these tests.

The second flavor, `docker-compose-e2e.yaml`, is constructed to run end-to-end tests using the Node.js SDK. Aside from functioning with the SDK, its primary differentiation is that there are containers for the fabric-ca servers. As a result, we are able to send REST calls to the organizational CAs for user registration and enrollment.

If you want to use the `docker-compose-e2e.yaml` without first running the byfn.sh script, then we will need to make four slight modifications. We need to point to the private keys for our Organization's CA's. You can locate these values in your crypto-config folder. For example, to locate the private key for Org1 we would follow this path - `crypto-config/peerOrganizations/org1.example.com/ca/`. The private key is a long hash value followed by `_sk`. The path for Org2 would be - `crypto-config/peerOrganizations/org2.example.com/ca/`.

In the `docker-compose-e2e.yaml` update the FABRIC_CA_SERVER_TLS_KEYFILE variable for ca0 and ca1. You also need to edit the path that is provided in the command to start the ca server. You are providing the same private key twice for each CA container.

7.9.8 Using CouchDB

The state database can be switched from the default (goleveldb) to CouchDB. The same chaincode functions are available with CouchDB, however, there is the added ability to perform rich and complex queries against the state database data content contingent upon the chaincode data being modeled as JSON.

To use CouchDB instead of the default database (goleveldb), follow the same procedures outlined earlier for generating the artifacts, except when starting the network pass `docker-compose-couch.yaml` as well:

```
docker-compose -f docker-compose-cli.yaml -f docker-compose-couch.yaml -f docker-
↪compose-etcdraft2.yaml up -d
```

abstore should now work using CouchDB underneath.

Note: If you choose to implement mapping of the fabric-couchdb container port to a host port, please make sure you are aware of the security implications. Mapping of the port in a development environment makes the CouchDB REST API available, and allows the visualization of the database via the CouchDB web interface (Fauxton). Production environments would likely refrain from implementing port mapping in order to restrict outside access to the CouchDB containers.

You can use **abstore** chaincode against the CouchDB state database using the steps outlined above, however in order to exercise the CouchDB query capabilities you will need to use a chaincode that has data modeled as JSON. The sample chaincode **marbles02** has been written to demostrate the queries you can issue from your chaincode if you are using a CouchDB database. You can locate the **marbles02** chaincode in the `fabric/examples/chaincode/go` directory.

We will follow the same process to create and join the channel as outlined in the *Create & Join Channel* section above. Once you have joined your peer(s) to the channel, use the following steps to interact with the **marbles02** chaincode:

- Package and install the chaincode on `peer0.org1.example.com`:

```
# before packaging Go chaincode, vendoring dependencies is required.
cd /opt/gopath/src/github.com/hyperledger/fabric-samples/chaincode/marbles02/go
GO111MODULE=on go mod vendor
cd -

# package and install the Go chaincode
peer lifecycle chaincode package marbles.tar.gz --path github.com/hyperledger/fabric-
↪samples/chaincode/marbles02/go/ --lang golang --label marbles_1
peer lifecycle chaincode install marbles.tar.gz
```

The install command will return a chaincode packageID that you will use to approve a chaincode definition.

```
2019-04-08 20:10:32.568 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪001 Installed remotely: response:<status:200 payload:"\nJmarbles_
↪1:cfb623954827aef3f35868764991cc7571b445a45cfd3325f7002f14156d61ae\022\tmarbles_1" >
2019-04-08 20:10:32.568 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO␣
↪002 Chaincode code package identifier: marbles_
↪1:cfb623954827aef3f35868764991cc7571b445a45cfd3325f7002f14156d61ae
```

- Save the packageID as an environment variable so you can pass it to future commands:

```
CC_PACKAGE_ID=marbles_
↪1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173
```

- Approve a chaincode definition as Org1:

```
# be sure to modify the $CHANNEL_NAME variable accordingly for the command

peer lifecycle chaincode approveformyorg --channelID $CHANNEL_NAME --name marbles --
↪version 1.0 --package-id $CC_PACKAGE_ID --sequence 1 --tls true --cafile /opt/
↪gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.
↪com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

- Install the chaincode on `peer0.org2.example.com`:

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer0.org2.example.com:9051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
peer lifecycle chaincode install marbles.tar.gz
```

- Approve a chaincode definition as Org2, and then commit the definition to the channel:

```
# be sure to modify the $CHANNEL_NAME variable accordingly for the command

peer lifecycle chaincode approveformyorg --channelID $CHANNEL_NAME --name marbles --
↪version 1.0 --package-id $CC_PACKAGE_ID --sequence 1 --tls true --cafile /opt/
↪gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.
↪com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID $CHANNEL_NAME␣
↪--name marbles --version 1.0 --sequence 1 --tls true --cafile /opt/gopath/src/
↪github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem --peerAddresses peer0.
↪org1.example.com:7051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/
↪fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/
↪tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles /opt/
↪gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.
↪com/peers/peer0.org2.example.com/tls/ca.crt
```

- We can now create some marbles. The first invoke of the chaincode will start the chaincode container. You may need to wait for the container to start.

```
# be sure to modify the $CHANNEL_NAME variable accordingly

peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/
↪github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n␣
↪marbles --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/
↪src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/
↪peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051␣
↪--tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"Args":["initMarble","marble1","blue","35","tom"]}'
```

(continues on next page)

(continued from previous page)

Once the container has started, you can issue additional commands to create some marbles and move them around:

```
# be sure to modify the $CHANNEL_NAME variable accordingly

peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/
↪github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n␣
↪marbles --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/
↪src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/
↪peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051␣
↪--tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"Args":["initMarble","marble2","red","50","tom"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/
↪github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n␣
↪marbles --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/
↪src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/
↪peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051␣
↪--tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"Args":["initMarble","marble3","blue","70","tom"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/
↪github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n␣
↪marbles --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/
↪src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/
↪peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051␣
↪--tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"Args":["transferMarble","marble2","jerry"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/
↪github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n␣
↪marbles --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/
↪src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/
↪peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051␣
↪--tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"Args":["transferMarblesBasedOnColor","blue","jerry"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/
↪github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
↪orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n␣
↪marbles --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/
↪src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/
↪peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051␣
↪--tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{
↪"Args":["delete","marble1"]}'
```

- If you chose to map the CouchDB ports in docker-compose, you can now view the state database through the CouchDB web interface (Fauxton) by opening a browser and navigating to the following URL:

  `http://localhost:5984/_utils`

You should see a database named `mychannel` (or your unique channel name) and the documents inside it.

Note: For the below commands, be sure to update the $CHANNEL_NAME variable appropriately.

You can run regular queries from the CLI (e.g. reading `marble2`):

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["readMarble","marble2"]}
↪'
```

The output should display the details of `marble2`:

```
Query Result: {"color":"red","docType":"marble","name":"marble2","owner":"jerry","size
↪":50}
```

You can retrieve the history of a specific marble - e.g. `marble1`:

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["getHistoryForMarble",
↪"marble1"]}'
```

The output should display the transactions on `marble1`:

```
Query Result: [{"TxId":
↪"1c3d3caf124c89f91a4c0f353723ac736c58155325f02890adebaa15e16e6464", "Value":{
↪"docType":"marble","name":"marble1","color":"blue","size":35,"owner":"tom"}},{"TxId
↪":"755d55c281889eaeebf405586f9e25d71d36eb3d35420af833a20a2f53a3eefd", "Value":{
↪"docType":"marble","name":"marble1","color":"blue","size":35,"owner":"jerry"}},{
↪"TxId":"819451032d813dde6247f85e56a89262555e04f14788ee33e28b232eef36d98f", "Value":}
↪]
```

You can also perform rich queries on the data content, such as querying marble fields by owner `jerry`:

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarblesByOwner",
↪"jerry"]}'
```

The output should display the two marbles owned by `jerry`:

```
Query Result: [{"Key":"marble2", "Record":{"color":"red","docType":"marble","name":
↪"marble2","owner":"jerry","size":50}},{"Key":"marble3", "Record":{"color":"blue",
↪"docType":"marble","name":"marble3","owner":"jerry","size":70}}]
```

7.9.9 Why CouchDB

CouchDB is a kind of NoSQL solution. It is a document-oriented database where document fields are stored as key-value maps. Fields can be either a simple key-value pair, list, or map. In addition to keyed/composite-key/key-range queries which are supported by LevelDB, CouchDB also supports full data rich queries capability, such as non-key queries against the whole blockchain data, since its data content is stored in JSON format and fully queryable. Therefore, CouchDB can meet chaincode, auditing, reporting requirements for many use cases that not supported by LevelDB.

CouchDB can also enhance the security for compliance and data protection in the blockchain. As it is able to implement field-level security through the filtering and masking of individual attributes within a transaction, and only authorizing the read-only permission if needed.

7.9.10 A Note on Data Persistence

If data persistence is desired on the peer container or the CouchDB container, one option is to mount a directory in the docker-host into a relevant directory in the container. For example, you may add the following two lines in the peer container specification in the `docker-compose-base.yaml` file:

```
volumes:
 - /var/hyperledger/peer0:/var/hyperledger/production
```

For the CouchDB container, you may add the following two lines in the CouchDB container specification:

```
volumes:
 - /var/hyperledger/couchdb0:/opt/couchdb/data
```

7.9.11 Troubleshooting

- Always start your network fresh. Use the following command to remove artifacts, crypto, containers and chaincode images:

  ```
  ./byfn.sh down
  ```

  Note: You **will** see errors if you do not remove old containers and images.

- If you see Docker errors, first check your docker version (*Prerequisites*), and then try restarting your Docker process. Problems with Docker are oftentimes not immediately recognizable. For example, you may see errors resulting from an inability to access crypto material mounted within a container.

  If they persist remove your images and start from scratch:

  ```
  docker rm -f $(docker ps -aq)
  docker rmi -f $(docker images -q)
  ```

- If you see errors on your create, approve, commit, invoke or query commands, make sure you have properly updated the channel name and chaincode name. There are placeholder values in the supplied sample commands.
- If you see the below error:

  ```
  Error: Error endorsing chaincode: rpc error: code = 2 desc = Error installing␣
  ↪chaincode code mycc:1.0(chaincode /var/hyperledger/production/chaincodes/mycc.1.
  ↪0 exits)
  ```

  You likely have chaincode images (e.g. `dev-peer1.org2.example.com-mycc-1.0` or `dev-peer0.org1.example.com-mycc-1.0`) from prior runs. Remove them and try again.

  ```
  docker rmi -f $(docker images | grep peer[0-9]-peer[0-9] | awk '{print $3}')
  ```

- If you see something similar to the following:

  ```
  Error connecting: rpc error: code = 14 desc = grpc: RPC failed fast due to␣
  ↪transport failure
  Error: rpc error: code = 14 desc = grpc: RPC failed fast due to transport failure
  ```

  Make sure you are running your network against the "1.0.0" images that have been retagged as "latest".

- If you see the below error:

```
[configtx/tool/localconfig] Load -> CRIT 002 Error reading configuration:␣
↪Unsupported Config Type ""
panic: Error reading configuration: Unsupported Config Type ""
```

Then you did not set the `FABRIC_CFG_PATH` environment variable properly. The configtxgen tool needs this variable in order to locate the configtx.yaml. Go back and execute an `export FABRIC_CFG_PATH=$PWD`, then recreate your channel artifacts.

- To cleanup the network, use the `down` option:

```
./byfn.sh down
```

- If you see an error stating that you still have "active endpoints", then prune your Docker networks. This will wipe your previous networks and start you with a fresh environment:

```
docker network prune
```

You will see the following message:

```
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N]
```

Select `y`.

- If you see an error similar to the following:

```
/bin/bash: ./scripts/script.sh: /bin/bash^M: bad interpreter: No such file or␣
↪directory
```

Ensure that the file in question (**script.sh** in this example) is encoded in the Unix format. This was most likely caused by not setting `core.autocrlf` to `false` in your Git configuration (see *Windows extras*). There are several ways of fixing this. If you have access to the vim editor for instance, open the file:

```
vim ./fabric-samples/first-network/scripts/script.sh
```

Then change its format by executing the following vim command:

```
:set ff=unix
```

Note: If you continue to see errors, share your logs on the **fabric-questions** channel on Hyperledger Rocket Chat or on StackOverflow.

7.10 Videos

Refer to the Hyperledger Fabric channel on YouTube

This collection contains developers demonstrating various v1 features and components such as: ledger, channels, gossip, SDK, chaincode, MSP, and more…

CHAPTER 8

Deploying a production network

This deployment guide is a high level overview of the proper sequence for setting up production Fabric network components, in addition to best practices and a few of the many considerations to keep in mind when deploying. Note that this topic will discuss "setting up the network" as a holistic process from the perspective of a single individual. More likely than not, real world production networks will not be set up by a single individual but as a collaborative effort directed by several individuals (a collection of banks each setting up their own components, for example) instead.

The process for deploying a Fabric network is complex and presumes an understanding of Public Key Infrastructure and managing distributed systems. If you are a smart contract or application developer, you should not need this level of expertise in deploying a production level Fabric network. However, you might need to be aware of how networks are deployed in order to develop effective smart contracts and applications.

If all you need is a development environment to test chaincode, smart contracts, and applications against, check out *Using the Fabric test network*. It includes two organizations, each owning one peer, and a single ordering service organization that owns a single ordering node. **This test network is not meant to provide a blueprint for deploying production components, and should not be used as such, as it makes assumptions and decisions that production deployments will not make.**

The guide will give you an overview of the steps of setting up production components and a production network:

- *Step one: Decide on your network configuration*
- *Step two: Set up a cluster for your resources*
- *Step three: Set up your CAs*
- *Step four: Use the CA to create identities and MSPs*
- ***Step five: Deploy nodes***
  - *Create a peer*
  - *Create an ordering node*

8.1 Step one: Decide on your network configuration

The structure of a blockchain network must be dictated by the use case. These fundamental business decisions will vary according to your use case, but let's consider a few scenarios.

In contrast to development environments or proofs of concept, security, resource management, and high availability become a priority when operating in production. How many nodes do you need to satisfy high availability, and in what data centers do you wish to deploy them in to satisfy both the needs of disaster recovery and data residency? How will you ensure that your private keys and roots of trust remain secure?

In addition to the above, here is a sampling of the decisions you will need to make before deploying components:

- **Certificate Authority configuration**. As part of the overall decisions you have to make about your peers (how many, how many on each channel, and so on) and about your ordering service (how many nodes, who will own them), you also have to decide on how the CAs for your organization will be deployed. Production networks should be using Transport Layer Security (TLS), which will require setting up a TLS CA and using it to generate TLS certficates. This TLS CA will need to be deployed before your enrollment CA. We'll discuss this more in *Step three: Set up your CAs*.
- **Use Organizational Units or not?** Some organizations might find it necessary to establish Organizational Units to create a separation between certain identities and MSPs created by a single CA.
- **Database type.** Some channels in a network might require all data to be modeled in a way *CouchDB as the State Database* can understand, while other networks, prioritizing speed, might decide that all peers will use LevelDB. Note that channels should not have peers that use both CouchDB and LevelDB on them, as the two database types model data slightly differently.
- **Channels and private data.** Some networks might decide that *Channels* are the best way to ensure privacy and isolation for certain transactions. Others might decide that a single channel, along with *Private data*, better serves their need for privacy.
- **Container orchestration.** Different users might also make different decisions about their container orchestration, creating separate containers for their peer process, logging for the peer, CouchDB, gRPC communications, and chaincode, while other users might decide to combine some of these processes.
- **Chaincode deployment method.** Users now have the option to deploy their chaincode using either the built in build and run support, a customized build and run using the *External Builders and Launchers*, or using an *Chaincode as an external service*.
- **Using firewalls.** In a production deployment, components belonging to one organization might need access to components from other organizations, necessitating the use of firewalls and advanced networking configuration. For example, applications using the Fabric SDK require access to all endorsing peers from all organizations and the ordering services for all channels. Similarly, peers need access to the ordering service on the channels that they are receiving new blocks from.

However and wherever your components are deployed, you will need a high degree of expertise in your management system of choice (such as Kubernetes) in order to efficiently operate your network. Similarly, the structure of the network must be designed to fit the business use case and any relevant laws and regulations government of the industry in which the network will be designed to function.

This deployment guide will not go through every iteration and potential network configuration, but does give common guidelines and rules to consider.

8.2 Step two: Set up a cluster for your resources

Generally speaking, Fabric is agnostic to the method used to deploy and manage it. It is possible, for example, to deploy and manage a peer from a laptop. For a number of reasons, this is likely to be unadvisable, but there is nothing

in Fabric that prohibits it.

As long as you have the ability to deploy containers, whether locally (or behind a firewall), or in a cloud, it should be possible to stand up components and connect them to each other. However, Kubernetes features a number of helpful tools that have made it a popular container management platform for deploying and managing Fabric networks. For more information about Kubernetes, check out the Kubernetes documentation. This topic will mostly limit its scope to the binaries and provide instructions that can be applied when using a Docker deployment or Kubernetes.

However and wherever you choose to deploy your components, you will need to make sure you have enough resources for the components to run effectively. The sizes you need will largely depend on your use case. If you plan to join a single peer to several high volume channels, it will need much more CPU and memory than a peer a user plans to join to a single channel. As a rough estimate, plan to dedicate approximately three times the resources to a peer as you plan to allocate to a single ordering node (as you will see below, it is recommended to deploy at least three and optimally five nodes in an ordering service). Similarly, you should need approximately a tenth of the resources for a CA as you will for a peer. You will also need to add storage to your cluster (some cloud providers may provide storage) as you cannot configure Persistent Volumes and Persistent Volume Claims without storage being set up with your cloud provider first.

By deploying a proof of concept network and testing it under load, you will have a better sense of the resources you will require.

8.2.1 Managing your infrastructure

The exact methods and tools you use to manage your backend will depend on the backend you choose. However, here are some considerations worth noting.

- Using secret objects to securely store important configuration files in your cluster. For information about Kubernetes secrets, check out Kubernetes secrets. You also have the option to use Hardened Security Modules (HSMs) or encrypted Persistent Volumes (PVs). Along similar lines, after deploying Fabric components, you will likely want to connect to a container on your own backend, for example using a private repo in a service like Docker Hub. In that case, you will need to code the login information in the form of a Kubernetes secret and include it in the YAML file when deploying components.
- Cluster considerations and node sizing. In step 2 above, we discussed a general outline for how to think about the sizings of nodes. Your use case, as well as a robust period of development, is the only way you will truly know how how large your peers, ordering nodes, and CAs will need to be.
- How you choose to mount your volumes. It is a best practice to mount the volumes relevant to your nodes external to the place where your nodes are deployed. This will allow you to reference these volumes later on (for example, restarting a node or a container that has crashed) without having to redeploy or regenerate your crypto material.
- How you will monitor your resources. It is critical that you establish a strategy and method for monitoring the resources used by your individual nodes and the resources deployed to your cluster generally. As you join your peers to more channels, you will need likely need to increase its CPU and memory allocation. Similarly, you will need to make sure you have enough storage space for your state database and blockchain.

8.3 Step three: Set up your CAs

The first component that must be deployed in a Fabric network is a CA. This is because the certificates associated with a node (not just for the node itself but also the certificates identifying who can administer the node) must be created before the node itself can be deployed. While it is not necessary to use the Fabric CA to create these certificates, the Fabric CA also creates MSP structures that are needed for components and organizations to be properly defined. If a user chooses to use a CA other than the Fabric CA, they will have to create the MSP folders themselves.

- One CA (or more, if you are using intermediate CAs — more on intermediate CAs below) is used to generate (through a process called "enrollment") the certificates of the admin of an organization, the MSP of that organization, and any nodes owned by that organization. This CA will also generate the certificates for any additional users. Because of its role in "enrolling" identities, this CA is sometimes called the "enrollment CA" or the "ecert CA".
- The other CA generates the certificates used to secure communications on Transport Layer Security (TLS). For this reason, this CA is often referred to as a "TLS CA". These TLS certificates are attached to actions as a way of preventing "man in the middle" attacks. Note that the TLS CA is only used for issuing certificates for nodes and can be shut down when that activity is completed. Users have the option to use one way (client only) TLS as well as two way (server and client) TLS, with the latter also known as "mutual TLS". Because specifying that your network will be using TLS (which is recommended) should be decided before deploying the "enrollment" CA (the YAML file specifying the configuration of this CA has a field for enabling TLS), you should deploy your TLS CA first and use its root certificate when bootstrapping your enrollment CA. This TLS certificate will also be used by the `fabric-ca client` when connecting to the enrollment CA to enroll identities for users and nodes.

While all of the non-TLS certificates associated with an organization can be created by a single "root" CA (that is, a CA that is its own root of trust), for added security organizations can decide to use "intermediate" CAs whose certificates are created by a root CA (or another intermediate CA that eventually leads back to a root CA). Because a compromise in the root CA leads to a collapse for its entire trust domain (the certs for the admins, nodes, and any CAs it has generated certificates for), intermediate CAs are a useful way to limit the exposure of the root CA. Whether you choose to use intermediate CAs will depend on the needs of your use case. They are not mandatory. Note that it is also possible to configure a Lightweight Directory Access Protocol (LDAP) to manage identities on a Fabric network for those enterprises that already have this implementation and do not want to add a layer of identity management to their existing infrastructure. The LDAP effectively pre registers all of the members of the directory and allows them to enroll based on the criteria given.

In a production network, it is recommended to deploy at least one CA per organization for enrollment purposes and another for TLS. For example, if you deploy three peers that are associated with one organization and an ordering node that is associated with an ordering organization, you will need at least four CAs. Two of the CAs will be for the peer organization (generating the enrollment and TLS certificates for the peer, admins, communications, and the folder structure of the MSP representing the organization) and the other two will be for the orderer organization. Note that users will generally only register and enroll with the enrollment CA, while nodes will register and enroll with both the enrollment CA (where the node will get its signing certificates that identify it when it attempts to sign its actions) and with the TLS CA (where it will get the TLS certificates it uses to authenticate its communications).

To install the Fabric CA client, which is used to register and enroll identities, follow the instructions in the Fabric CA User's Guide. Note that there are a number of configuration options when deploying a CA (the database type, for example, or whether to use LDAP).

For an example of how to setup a CA and enroll its admin, check out Setup Orderer Org CA. For an example of how to set up a TLS CA, check out Setup TLS CA. Note that bootstrapping the CA server means assigning a username and password that functions as "registering" the CA admin.

8.4 Step four: Use the CA to create identities and MSPs

After you have created your CAs, you can use them to create the certificates for the identities and components related to your organization (which is represented by an MSP). For each organization, you will need to, at a minimum:

- **Register and enroll an admin identity and create an MSP**. After the CA that will be associated with an organization has been created, it can be used to first register an identity and then enroll it. In the first step, a username and password for the identity is assigned by the admin of the CA. Attributes and affiliations can also be given to the identity (for example, a `role` of `admin`, which is necessary for organization admins). After the identity has been registered, it can be enrolled by using the username and password. The CA will generate

two certificates for this identity — a public certificate (also known as a signcert) known to the other members of the network, and the private key (stored in the `keystore` folder) used to sign actions taken by the identity. The CA will also generate an MSP file containing the public certificate of the CA issuing the certificate and the root of trust for the CA (this may or may not be the same CA). This MSP can be thought of as defining the organization associated with the identity of the admin. For an example of how this process looks, check out the this example of how an admin is enrolled. In cases where the admin of the org will also be an admin of a node (which will be typical), **you must create the org admin identity before creating the local MSP of a node, since the certificate of the node admin must be used when creating the local MSP**.

- **Register and enroll node identities**. Just as an org admin identity is registered and enrolled, the identity of a node must be registered and enrolled with both an enrollment CA and the TLS CA. For this reason, it can be useful for your enrollment CA and TLS to share a database (which allows the node identity to only be registered once and enrolled by each CA server separately), though this is an optional configuration option. Instead of giving a node a role of `admin` or `user` when registering it with the enrollment CA, give it a role of `peer` or `orderer`. As with the admin, attributes and affiliations for this identity can also be assigned. The MSP structure for a node is known as a "local MSP", since the permissions assigned to the identities are only relevant at the local (node) level. This MSP is created when the node identity is created, and is used when bootstrapping the node. You will use the TLS root certificate generated when enrolling with the TLS CA when joining your organization to the channel (this certificate must be added to the org MSP that was created when you enrolled your admin) and when using the peer binary as a CLI client to make calls to other peers (as in `peer chaincode invoke`) or ordering nodes (as in `peer channel fetch`) because there is no `orderer` CLI. It is not necessary to add the TLS root certificates to the local MSP of a node because these certificates are contained in the channel configuration.

For more conceptual information about identities and permissions in a Fabric-based blockchain network, see *Identity* and membership/membership.html.

For a look at how to use a CA to generate an admin identity and MSP, check out Enroll Org1's Admin.

To see how to use the enrollment CA to and the TLS CA to generate the certificates for a node, check out Setup Org1's Peers.

8.5 Step five: Deploy nodes

Once you have gathered all of the certificates and MSPs you need, you're almost ready to create a node. As discussed above, there are a number of valid ways to deploy nodes.

8.5.1 Create a peer

Before you can create a peer, you will need to customize the configuration file for the peer. In Fabric, this file is called `core.yaml`. You can find a sample `core.yaml` configuration file in the sampleconfig directory of Hyperledger Fabric.

As you can see in the file, there are quite a number of parameters you either have the option to set or will need to set for your node to work properly. In general, if you do not have the need to change a tuning value, leave it alone. You will, however, likely need to adjust the various addresses, specify the database type you want to use, as well as to specify where the MSP for the node is located.

You have two main options for tuning your configuration.

1. Edit the YAML file bundled with the binaries.
2. Use environment variable overrides when deploying.
3. Specify flags on CLI commands.

Option 1 has the advantage of persisting your changes whenever you bring down and bring back up the node. The downside is that you will have to port the options you customized to the new YAML when upgrading to a new binary version (you should use the latest YAML when upgrading to a new version).

Either way, here are some values in `core.yaml` you must review.

- `peer.localMspID`: this is the name of the local MSP of your peer organization. This MSP is where your peer organization admins will be listed as well as the peer organization's root CA and TLS CA certificates.
- `peer.mspConfigPath`: the place where the local MSP for the peer is located. Note that it is a best practice to mount this volume external to your container. This ensures that even if the container is stopped (for example, during a maintenance cycle) that the MSPs are not lost and have to be recreated.
- `peer.address`: represents the endpoint to other peers in the same organization, an important consideration when establishing gossip communication within an organization.
- `peer.tls`: When you set the `enabled` value to `true` (as should be done in a production network), you will have to specify the locations of the relevant TLS certificates. Note that all of the nodes in a network (both the peers and the ordering nodes) must either all have TLS enabled or not enabled. For production networks, it is highly recommended to enable TLS. As with your MSP, it is a best practice to mount this volume external to your container.
- `ledger`: users have a number of decisions to make about their ledger, including the state database type (LevelDB or CouchDB, for example), and its location (specified in `fileSystemPath`). Note that for CouchDB in particular it is a best practice to operate your state database external to the peer, as you will be better able to allocate specific resources to the database this way.
- `gossip`: there are a number of configuration options to think about when setting up *Gossip data dissemination protocol*, including the `externalEndpoint` (which makes peers discoverable to peers owned by other organizations) as well as the `bootstrap` address (which identifies a peer in the peer's own organization).
- `chaincode.externalBuilders`: this field is important to set when using cc_service.html.

When you're comfortable with how your peer has been configured, how your volumes are mounted, and your backend configuration, you can run the command to launch the peer (this command will depend on your backend configuration).

8.5.2 Create an ordering node

Unlike the creation of a peer, you will need to create a genesis block (or reference a block that has already been created, if adding an ordering node to an existing ordering service) and specify the path to it before launching the ordering node.

In Fabric, this configuration file for ordering nodes is called `orderer.yaml`. You can find a sample `orderer.yaml` configuration file in the sampleconfig directory of Hyperledger Fabric. Note that `orderer.yaml` is different than the "genesis block" of an ordering service. This block, which includes the initial configuration of the orderer system channel, must be created before an ordering node is created because it is used to bootstrap the node.

As with the peer, you will see that there are quite a number of parameters you either have the option to set or will need to set for your node to work properly. In general, if you do not have the need to change a tuning value, leave it alone.

You have two main options for tuning your configuration.

1. Edit the YAML file bundled with the binaries.
2. Use environment variable overrides when deploying.
3. Specify flags on CLI commands.

Option 1 has the advantage of persisting your changes whenever you bring down and bring back up the node. The downside is that you will have to port the options you customized to the new YAML when upgrading to a new binary version (you should use the latest YAML when upgrading to a new version).

Either way, here are some values in `orderer.yaml` you must review. You will notice that some of these fields are the same as those in `core.yaml` only with different names.

- `General.LocalMSPID`: this is the name of the local MSP, generated by your CA, of your orderer organization.
- `General.LocalMSPDir`: the place where the local MSP for the ordering node is located. Note that it is a best practice to mount this volume external to your container.
- `General.ListenAddress` and `General.ListenPort`: represents the endpoint to other ordering nodes in the same organization.
- `FileLedger`: although ordering nodes do not have a state database, they still all carry copies of the blockchain, as this allows them to verify permissions using the latest config block. Therefore the ledger fields should be customized with the correct file path.
- `Cluster`: these values are important for ordering service nodes that communicate with other ordering nodes, such as in a Raft based ordering service.
- `General.BootstrapFile`: this is the name of the configuration block used to bootstrap an ordering node. If this node is the first node generated in an ordering service, this file will have to be generated and is known as the "genesis block".
- `General.BootstrapMethod`: the method by which the bootstrap block is given. For now, this can only be `file`, in which the file in the `BootstrapFile` is specified. Starting in 2.0, you can specify `none` to simply start the orderer without bootstrapping.
- `Consensus`: determines the key/value pairs allowed by the consensus plugin (Raft ordering services are supported and recommended) for the Write Ahead Logs (`WALDir`) and Snapshots (`SnapDir`).

When you're comfortable with how your ordering node has been configured, how your volumes are mounted, and your backend configuration, you can run the command to launch the ordering node (this command will depend on your backend configuration).

8.6 Next steps

Blockchain networks are all about connection, so once you've deployed nodes, you'll obviously want to connect them to other nodes! If you have a peer organization and a peer, you'll want to join your organization to a consortium and join or *Channels*. If you have an ordering node, you will want to add peer organizations to your consortium. You'll also want to learn how to develop chaincode, which you can learn about in the topics *The scenario* and *Fabric chaincode lifecycle*.

Part of the process of connecting nodes and creating channels will involve modifying policies to fit the use cases of business networks. For more information about policies, check out *Policies*.

One of the common tasks in a Fabric will be the editing of existing channels. For a tutorial about that process, check out *Updating a channel configuration*. One popular channel update is to add an org to an existing channel. For a tutorial about that specific process, check out *Adding an Org to a Channel*. For information about upgrading nodes after they have been deployed, check out *Upgrading your components*.

CHAPTER 9

Operations Guides

9.1 Setting up an ordering node

In this topic, we'll describe the process for bootstrapping an ordering node. If you want more information about the different ordering service implementations and their relative strengths and weaknesses, check out our conceptual documentation about ordering.

Broadly, this topic will involve a few interrelated steps:

1. Creating the organization your ordering node belongs to (if you have not already done so)
2. Configuring your node (using `orderer.yaml`)
3. Creating the genesis block for the orderer system channel
4. Bootstrapping the orderer

Note: this topic assumes you have already pulled the Hyperledger Fabric orderer images from docker hub.

9.1.1 Create an organization definition

Like peers, all orderers must belong to an organization that must be created before the orderer itself is created. This organization has a definition encapsulated by a Membership Service Provider (MSP) that is created by a Certificate Authority (CA) dedicated to creating the certificates and MSP for the organization.

For information about creating a CA and using it to create users and an MSP, check out the Fabric CA user's guide.

9.1.2 Configure your node

The configuration of the orderer is handled through a `yaml` file called `orderer.yaml`. The `FABRIC_CFG_PATH` environment variable is used to point to an `orderer.yaml` file you've configured, which will extract a series of files and certificates on your file system.

To look at a sample `orderer.yaml`, check out the `fabric-samples` github repo, which **should be read and studied closely** before proceeding. Note in particular a few values:

- `LocalMSPID` — this is the name of the MSP, generated by your CA, of your orderer organization. This is where your orderer organization admins will be listed.
- `LocalMSPDir` — the place in your file system where the local MSP is located.
- `# TLS enabled`, `Enabled: false`. This is where you specify whether you want to enable TLS. If you set this value to `true`, you will have to specify the locations of the relevant TLS certificates. Note that this is mandatory for Raft nodes.
- `BootstrapFile` — this is the name of the genesis block you will generate for this ordering service.
- `BootstrapMethod` — the method by which the bootstrap block is given. For now, this can only be `file`, in which the file in the `BootstrapFile` is specified.

If you are deploying this node as part of a cluster (for example, as part of a cluster of Raft nodes), make note of the `Cluster` and `Consensus` sections.

If you plan to deploy a Kafka based ordering service, you will need to complete the `Kafka` section.

9.1.3 Generate the genesis block of the orderer

The first block of a newly created channel is known as a "genesis block". If this genesis block is being created as part of the creation of a **new network** (in other words, if the orderer being created will not be joined to an existing cluster of orderers), then this genesis block will be the first block of the "orderer system channel" (also known as the "ordering system channel"), a special channel managed by the orderer admins which includes a list of the organizations permitted to create channels. *The genesis block of the orderer system channel is special: it must be created and included in the configuration of the node before the node can be started.*

To learn how to create a genesis block using the `configtxgen` tool, check out Channel Configuration (configtx).

9.1.4 Bootstrap the ordering node

Once you have built the images, created the MSP, configured your `orderer.yaml`, and created the genesis block, you're ready to start your orderer using a command that will look similar to:

```
docker-compose -f docker-compose-cli.yaml up -d --no-deps orderer.example.com
```

With the address of your orderer replacing `orderer.example.com`.

9.2 Membership Service Providers (MSP)

The document serves to provide details on the setup and best practices for MSPs.

Membership Service Provider (MSP) is a Hyperledger Fabric component that offers an abstraction of membership operations.

In particular, an MSP abstracts away all cryptographic mechanisms and protocols behind issuing certificates, validating certificates, and user authentication. An MSP may define its own notion of identity, and the rules by which those identities are governed (identity validation) and authenticated (signature generation and verification).

A Hyperledger Fabric blockchain network can be governed by one or more MSPs. This provides modularity of membership operations, and interoperability across different membership standards and architectures.

In the rest of this document we elaborate on the setup of the MSP implementation supported by Hyperledger Fabric, and discuss best practices concerning its use.

9.2.1 MSP Configuration

To setup an instance of the MSP, its configuration needs to be specified locally at each peer and orderer (to enable peer and orderer signing), and on the channels to enable peer, orderer, client identity validation, and respective signature verification (authentication) by and for all channel members.

Firstly, for each MSP a name needs to be specified in order to reference that MSP in the network (e.g. `msp1`, `org2`, and `org3.divA`). This is the name under which membership rules of an MSP representing a consortium, organization or organization division is to be referenced in a channel. This is also referred to as the *MSP Identifier* or *MSP ID*. MSP Identifiers are required to be unique per MSP instance. For example, shall two MSP instances with the same identifier be detected at the system channel genesis, orderer setup will fail.

In the case of the default MSP implementation, a set of parameters need to be specified to allow for identity (certificate) validation and signature verification. These parameters are deduced by RFC5280, and include:

- A list of self-signed (X.509) CA certificates to constitute the *root of trust*
- A list of X.509 certificates to represent intermediate CAs this provider considers for certificate validation; these certificates ought to be certified by exactly one of the certificates in the root of trust; intermediate CAs are optional parameters
- A list of X.509 certificates representing the administrators of this MSP with a verifiable certificate path to exactly one of the CA certificates of the root of trust; owners of these certificates are authorized to request changes to this MSP configuration (e.g. root CAs, intermediate CAs)
- A list of Organizational Units that valid members of this MSP should include in their X.509 certificate; this is an optional configuration parameter, used when, e.g., multiple organizations leverage the same root of trust, and intermediate CAs, and have reserved an OU field for their members
- A list of certificate revocation lists (CRLs) each corresponding to exactly one of the listed (intermediate or root) MSP Certificate Authorities; this is an optional parameter
- A list of self-signed (X.509) certificates to constitute the *TLS root of trust* for TLS certificates.
- A list of X.509 certificates to represent intermediate TLS CAs this provider considers; these certificates ought to be certified by exactly one of the certificates in the TLS root of trust; intermediate CAs are optional parameters.

Valid identities for this MSP instance are required to satisfy the following conditions:

- They are in the form of X.509 certificates with a verifiable certificate path to exactly one of the root of trust certificates;
- They are not included in any CRL;
- And they *list* one or more of the Organizational Units of the MSP configuration in the `OU` field of their X.509 certificate structure.

For more information on the validity of identities in the current MSP implementation, we refer the reader to msp-identity-validity-rules.

In addition to verification related parameters, for the MSP to enable the node on which it is instantiated to sign or authenticate, one needs to specify:

- The signing key used for signing by the node (currently only ECDSA keys are supported), and
- The node's X.509 certificate, that is a valid identity under the verification parameters of this MSP.

It is important to note that MSP identities never expire; they can only be revoked by adding them to the appropriate CRLs. Additionally, there is currently no support for enforcing revocation of TLS certificates.

9.2.2 How to generate MSP certificates and their signing keys?

Openssl can be used to generate X.509 certificates and keys. Please note that Hyperledger Fabric does not support RSA key and certificates.

Alternatively, the `cryptogen` tool can be used as described in *Getting Started*.

Hyperledger Fabric CA can also be used to generate the keys and certificates needed to configure an MSP.

9.2.3 MSP setup on the peer & orderer side

To set up a local MSP (for either a peer or an orderer), the administrator should create a folder (e.g. `$MY_PATH/mspconfig`) that contains six subfolders and a file:

1. a folder `admincerts` to include PEM files each corresponding to an administrator certificate
2. a folder `cacerts` to include PEM files each corresponding to a root CA's certificate
3. (optional) a folder `intermediatecerts` to include PEM files each corresponding to an intermediate CA's certificate
4. (optional) a file `config.yaml` to configure the supported Organizational Units and identity classifications (see respective sections below).
5. (optional) a folder `crls` to include the considered CRLs
6. a folder `keystore` to include a PEM file with the node's signing key; we emphasise that currently RSA keys are not supported
7. a folder `signcerts` to include a PEM file with the node's X.509 certificate
8. (optional) a folder `tlscacerts` to include PEM files each corresponding to a TLS root CA's certificate
9. (optional) a folder `tlsintermediatecerts` to include PEM files each corresponding to an intermediate TLS CA's certificate

In the configuration file of the node (core.yaml file for the peer, and orderer.yaml for the orderer), one needs to specify the path to the mspconfig folder, and the MSP Identifier of the node's MSP. The path to the mspconfig folder is expected to be relative to FABRIC_CFG_PATH and is provided as the value of parameter `mspConfigPath` for the peer, and `LocalMSPDir` for the orderer. The identifier of the node's MSP is provided as a value of parameter `localMspId` for the peer and `LocalMSPID` for the orderer. These variables can be overridden via the environment using the CORE prefix for peer (e.g. CORE_PEER_LOCALMSPID) and the ORDERER prefix for the orderer (e.g. ORDERER_GENERAL_LOCALMSPID). Notice that for the orderer setup, one needs to generate, and provide to the orderer the genesis block of the system channel. The MSP configuration needs of this block are detailed in the next section.

Reconfiguration of a "local" MSP is only possible manually, and requires that the peer or orderer process is restarted. In subsequent releases we aim to offer online/dynamic reconfiguration (i.e. without requiring to stop the node by using a node managed system chaincode).

9.2.4 Organizational Units

In order to configure the list of Organizational Units that valid members of this MSP should include in their X.509 certificate, the `config.yaml` file needs to specify the organizational unit (OU, for short) identifiers. You can find an example below:

```
OrganizationalUnitIdentifiers:
  - Certificate: "cacerts/cacert1.pem"
    OrganizationalUnitIdentifier: "commercial"
  - Certificate: "cacerts/cacert2.pem"
    OrganizationalUnitIdentifier: "administrators"
```

The above example declares two organizational unit identifiers: **commercial** and **administrators**. An MSP identity is valid if it carries at least one of these organizational unit identifiers. The `Certificate` field refers to the CA or intermediate CA certificate path under which identities, having that specific OU, should be validated. The path is relative to the MSP root folder and cannot be empty.

9.2.5 Identity Classification

The default MSP implementation allows organizations to further classify identities into clients, admins, peers, and orderers based on the OUs of their x509 certificates.

- An identity should be classified as a **client** if it transacts on the network.
- An identity should be classified as an **admin** if it handles administrative tasks such as joining a peer to a channel or signing a channel configuration update transaction.
- An identity should be classified as a **peer** if it endorses or commits transactions.
- An identity should be classified as an **orderer** if belongs to an ordering node.

In order to define the clients, admins, peers, and orderers of a given MSP, the `config.yaml` file needs to be set appropriately. You can find an example NodeOU section of the `config.yaml` file below:

```
NodeOUs:
  Enable: true
  # For each identity classification that you would like to utilize, specify
  # an OU identifier.
  # You can optionally configure that the OU identifier must be issued by a specific␣
↪CA
  # or intermediate certificate from your organization. However, it is typical to NOT
  # configure a specific Certificate. By not configuring a specific Certificate, you␣
↪will be
  # able to add other CA or intermediate certs later, without having to reissue all␣
↪credentials.
  # For this reason, the sample below comments out the Certificate field.
  ClientOUIdentifier:
    # Certificate: "cacerts/cacert.pem"
    OrganizationalUnitIdentifier: "client"
  AdminOUIdentifier:
    # Certificate: "cacerts/cacert.pem"
    OrganizationalUnitIdentifier: "admin"
  PeerOUIdentifier:
    # Certificate: "cacerts/cacert.pem"
    OrganizationalUnitIdentifier: "peer"
  OrdererOUIdentifier:
    # Certificate: "cacerts/cacert.pem"
    OrganizationalUnitIdentifier: "orderer"
```

Identity classification is enabled when `NodeOUs.Enable` is set to `true`. Then the client (admin, peer, orderer) organizational unit identifier is defined by setting the properties of the `NodeOUs.ClientOUIdentifier` (`NodeOUs.AdminOUIdentifier`, `NodeOUs.PeerOUIdentifier`, `NodeOUs.OrdererOUIdentifier`) key:

1. `OrganizationalUnitIdentifier`: Is the OU value that the x509 certificate needs to contain to be considered a client (admin, peer, orderer respectively). If this field is empty, then the classification is not applied.
2. `Certificate`: (Optional) Set this to the path of the CA or intermediate CA certificate under which client (peer, admin or orderer) identities should be validated. The field is relative to the MSP root folder. Only a single Certificate can be specified. If you do not set this field, then the identities are validated under any CA defined in the organization's MSP configuration, which could be desirable in the future if you need to add other CA or intermediate certificates.

Notice that if the `NodeOUs.ClientOUIdentifier` section (`NodeOUs.AdminOUIdentifier`, `NodeOUs.PeerOUIdentifier`, `NodeOUs.OrdererOUIdentifier`) is missing, then the classification is not applied. If `NodeOUs.Enable` is set to `true` and no classification keys are defined, then identity classification is assumed to be disabled.

Identities can use organizational units to be classified as either a client, an admin, a peer, or an orderer. The four classifications are mutually exclusive. The 1.1 channel capability needs to be enabled before identities can be classified as clients or peers. The 1.4.3 channel capability needs to be enabled for identities to be classified as an admin or orderer.

Classification allows identities to be classified as admins (and conduct administrator actions) without the certificate being stored in the `admincerts` folder of the MSP. Instead, the `admincerts` folder can remain empty and administrators can be created by enrolling identities with the admin OU. Certificates in the `admincerts` folder will still grant the role of administrator to their bearer, provided that they possess the client or admin OU.

9.2.6 Channel MSP setup

At the genesis of the system, verification parameters of all the MSPs that appear in the network need to be specified, and included in the system channel's genesis block. Recall that MSP verification parameters consist of the MSP identifier, the root of trust certificates, intermediate CA and admin certificates, as well as OU specifications and CRLs. The system genesis block is provided to the orderers at their setup phase, and allows them to authenticate channel creation requests. Orderers would reject the system genesis block, if the latter includes two MSPs with the same identifier, and consequently the bootstrapping of the network would fail.

For application channels, the verification components of only the MSPs that govern a channel need to reside in the channel's genesis block. We emphasize that it is **the responsibility of the application** to ensure that correct MSP configuration information is included in the genesis blocks (or the most recent configuration block) of a channel prior to instructing one or more of their peers to join the channel.

When bootstrapping a channel with the help of the configtxgen tool, one can configure the channel MSPs by including the verification parameters of MSP in the mspconfig folder, and setting that path in the relevant section in `configtx.yaml`.

Reconfiguration of an MSP on the channel, including announcements of the certificate revocation lists associated to the CAs of that MSP is achieved through the creation of a `config_update` object by the owner of one of the administrator certificates of the MSP. The client application managed by the admin would then announce this update to the channels in which this MSP appears.

9.2.7 Best Practices

In this section we elaborate on best practices for MSP configuration in commonly met scenarios.

1) Mapping between organizations/corporations and MSPs

We recommend that there is a one-to-one mapping between organizations and MSPs. If a different type of mapping is chosen, the following needs to be to considered:

- **One organization employing various MSPs.** This corresponds to the case of an organization including a variety of divisions each represented by its MSP, either for management independence reasons, or for privacy

reasons. In this case a peer can only be owned by a single MSP, and will not recognize peers with identities from other MSPs as peers of the same organization. The implication of this is that peers may share through gossip organization-scoped data with a set of peers that are members of the same subdivision, and NOT with the full set of providers constituting the actual organization.

- **Multiple organizations using a single MSP.** This corresponds to a case of a consortium of organizations that are governed by similar membership architecture. One needs to know here that peers would propagate organization-scoped messages to the peers that have an identity under the same MSP regardless of whether they belong to the same actual organization. This is a limitation of the granularity of MSP definition, and/or of the peer's configuration.

2) One organization has different divisions (say organizational units), to which it wants to grant access to different channels.

Two ways to handle this:

- **Define one MSP to accommodate membership for all organization's members**. Configuration of that MSP would consist of a list of root CAs, intermediate CAs and admin certificates; and membership identities would include the organizational unit (`OU`) a member belongs to. Policies can then be defined to capture members of a specific `role` (should be one of: peer, admin, client, orderer, member), and these policies may constitute the read/write policies of a channel or endorsement policies of a chaincode. Specifying custom OUs in the profile section of `configtx.yaml` is currently not configured. A limitation of this approach is that gossip peers would consider peers with membership identities under their local MSP as members of the same organization, and would consequently gossip with them organization-scoped data (e.g. their status).
- **Defining one MSP to represent each division**. This would involve specifying for each division, a set of certificates for root CAs, intermediate CAs, and admin Certs, such that there is no overlapping certification path across MSPs. This would mean that, for example, a different intermediate CA per subdivision is employed. Here the disadvantage is the management of more than one MSPs instead of one, but this circumvents the issue present in the previous approach. One could also define one MSP for each division by leveraging an OU extension of the MSP configuration.

3) Separating clients from peers of the same organization.

In many cases it is required that the "type" of an identity is retrievable from the identity itself (e.g. it may be needed that endorsements are guaranteed to have derived by peers, and not clients or nodes acting solely as orderers).

There is limited support for such requirements.

One way to allow for this separation is to create a separate intermediate CA for each node type - one for clients and one for peers/orderers; and configure two different MSPs - one for clients and one for peers/orderers. Channels this organization should be accessing would need to include both MSPs, while endorsement policies will leverage only the MSP that refers to the peers. This would ultimately result in the organization being mapped to two MSP instances, and would have certain consequences on the way peers and clients interact.

Gossip would not be drastically impacted as all peers of the same organization would still belong to one MSP. Peers can restrict the execution of certain system chaincodes to local MSP based policies. For example, peers would only execute "joinChannel" request if the request is signed by the admin of their local MSP who can only be a client (end-user should be sitting at the origin of that request). We can go around this inconsistency if we accept that the only clients to be members of a peer/orderer MSP would be the administrators of that MSP.

Another point to be considered with this approach is that peers authorize event registration requests based on membership of request originator within their local MSP. Clearly, since the originator of the request is a client, the request originator is always deemed to belong to a different MSP than the requested peer and the peer would reject the request.

4) Admin and CA certificates.

It is important to set MSP admin certificates to be different than any of the certificates considered by the MSP for `root of trust`, or intermediate CAs. This is a common (security) practice to separate the duties of management of membership components from the issuing of new certificates, and/or validation of existing ones.

5) Blacklisting an intermediate CA.

As mentioned in previous sections, reconfiguration of an MSP is achieved by reconfiguration mechanisms (manual reconfiguration for the local MSP instances, and via properly constructed `config_update` messages for MSP instances of a channel). Clearly, there are two ways to ensure an intermediate CA considered in an MSP is no longer considered for that MSP's identity validation:

1. Reconfigure the MSP to no longer include the certificate of that intermediate CA in the list of trusted intermediate CA certs. For the locally configured MSP, this would mean that the certificate of this CA is removed from the `intermediatecerts` folder.
2. Reconfigure the MSP to include a CRL produced by the root of trust which denounces the mentioned intermediate CA's certificate.

In the current MSP implementation we only support method (1) as it is simpler and does not require blacklisting the no longer considered intermediate CA.

6) CAs and TLS CAs

MSP identities' root CAs and MSP TLS certificates' root CAs (and relative intermediate CAs) need to be declared in different folders. This is to avoid confusion between different classes of certificates. It is not forbidden to reuse the same CAs for both MSP identities and TLS certificates but best practices suggest to avoid this in production.

9.3 Using a Hardware Security Module (HSM)

You can use a Hardware Security Module (HSM) to generate and store the private keys used by your Fabric nodes. An HSM protects your private keys and handles cryptographic operations, which allows your peers and ordering nodes to sign and endorse transactions without exposing their private keys. Currently, Fabric only supports the PKCS11 standard to communicate with an HSM.

9.3.1 Configuring an HSM

To use an HSM with your Fabric node, you need to update the BCCSP (Crypto Service Provider) section of the node configuration file such as core.yaml or orderer.yaml. In BCCSP section, you need to select PKCS11 as the provider and provide the path to the PKCS11 library that you would like to use. You also need to provide the label and pin of the token that you created for your cryptographic operations. You can use one token to generate and store multiple keys.

The prebuilt Hyperledger Fabric Docker images are not enabled to use PKCS11. If you are deploying Fabric using docker, you need to build your own images and enable PKCS11 using the following command:

```
make docker GO_TAGS=pkcs11
```

You also need to ensure that the PKCS11 library is available to be used by the node by installing it or mounting it inside the container.

Example

The following example demonstrates how to configure a Fabric node to use an HSM.

First, you will need to install an implementation of the PKCS11 interface. This example uses the softhsm open source implementation. After downloading and configuring softhsm, you will need to set the SOFTHSM2_CONF environment variable to point to the softhsm2 configuration file.

You can then use softhsm to create the token that will handle the cryptographic operations of your Fabric node inside an HSM slot. In this example, we create a token labelled "fabric" and set the pin to "71811222". After you have

created the token, update the configuration file to use PKCS11 and your token as the crypto service provider. You can find an example BCCSP section below:

```
###############################################################################
# BCCSP (BlockChain Crypto Service Provider) section is used to select which
# crypto library implementation to use
###############################################################################
bccsp:
  default: PKCS11
  pkcs11:
    Library: /etc/hyperledger/fabric/libsofthsm2.so
    Pin: 71811222
    Label: fabric
    hash: SHA2
    security: 256
```

You can also use environment variables to override the relevant fields of the configuration file. If you are connecting to an HSM using the Fabric CA server, you need to set the following environment variables:

```
FABRIC_CA_SERVER_BCCSP_DEFAULT=PKCS11
FABRIC_CA_SERVER_BCCSP_PKCS11_LIBRARY=/etc/hyperledger/fabric/libsofthsm2.so
FABRIC_CA_SERVER_BCCSP_PKCS11_PIN=71811222
FABRIC_CA_SERVER_BCCSP_PKCS11_LABEL=fabric
```

If you are deploying your nodes using docker compose, after building your own images, you can update your docker compose files to mount the softhsm library and configuration file inside the container using volumes. As an example, you would add the following environment and volumes variables to your docker compose file:

```
  environment:
     - SOFTHSM2_CONF=/etc/hyperledger/fabric/config.file
  volumes:
     - /home/softhsm/config.file:/etc/hyperledger/fabric/config.file
     - /usr/local/Cellar/softhsm/2.1.0/lib/softhsm/libsofthsm2.so:/etc/hyperledger/
↪fabric/libsofthsm2.so
```

9.3.2 Setting up a network using HSM

If you are deploying Fabric nodes using an HSM, your private keys need to be generated inside the HSM rather than inside the `keystore` folder of the node's local MSP folder. The `keystore` folder of the MSP will remain empty. Instead, the Fabric node will use the subject key identifier of the signing certificate in the `signcerts` folder to retrieve the private key from inside the HSM. The process for creating the MSP folders will be different depending on if you are using a Fabric Certificate Authority (CA) your own CA.

Using a Fabric CA

You can set up a Fabric CA to use an HSM by making the same edits to the configuration file as you would make to a peer or ordering node. Because you can use Fabric CA to generate keys inside an HSM, the process of creating the local MSP folders is straightforward. Use the following steps:

1. Create an HSM token and point to it in the Fabric CA server configuration file. When the Fabric CA server starts, it will generate the CA signing certificate inside your HSM. If you are not concerned about exposing your CA signing certificate, you can skip this step.
2. Use the Fabric CA client to register the peer or ordering node identities with your CA.

3. Edit the Fabric CA client config file or environment variables to use your HSM as the crypto service provider. Then for each node, use the Fabric CA client to generate the component MSP folder by enrolling against the node identity. The enroll command will generate the private key inside your HSM.
4. Update the BCCSP section of the peer or orderer configuration file to use PKCS11 and your token as the crypto service provider. Point to the MSP that was generated using the Fabric CA client. Once it is deployed, the peer or orderer node will be able sign and endorse transactions with the private key protected by the HSM.

Using an HSM with your own CA

If you are using your own Certificate Authority to deploy Fabric components, you can use an HSM using the following steps:

1. Configure your CA to communicate with an HSM using PKCS11 and create a token. Then use your CA to generate the private key and signing certificate for each node, with the private key generated inside the HSM.
2. Use your CA to build the node MSP folder. Place the signing certificate that you generated in step 1 inside the `signcerts` folder. You can leave the `keystore` folder empty.
3. Update the peer or orderer configuration file to use PKCS11 and your token as the crypto service provider. Point to the MSP folder that you created with the signing certificate inside. Once it has deployed, the peer or ordering node will be able to sign and endorse transactions using the HSM.

9.4 Channel Configuration (configtx)

Shared configuration for a Hyperledger Fabric blockchain network is stored in a collection configuration transactions, one per channel. Each configuration transaction is usually referred to by the shorter name *configtx*.

Channel configuration has the following important properties:

1. **Versioned**: All elements of the configuration have an associated version which is advanced with every modification. Further, every committed configuration receives a sequence number.
2. **Permissioned**: Each element of the configuration has an associated policy which governs whether or not modification to that element is permitted. Anyone with a copy of the previous configtx (and no additional info) may verify the validity of a new config based on these policies.
3. **Hierarchical**: A root configuration group contains sub-groups, and each group of the hierarchy has associated values and policies. These policies can take advantage of the hierarchy to derive policies at one level from policies of lower levels.

9.4.1 Anatomy of a configuration

Configuration is stored as a transaction of type `HeaderType_CONFIG` in a block with no other transactions. These blocks are referred to as *Configuration Blocks*, the first of which is referred to as the *Genesis Block*.

The proto structures for configuration are stored in `fabric-protos/common/configtx.proto`. The Envelope of type `HeaderType_CONFIG` encodes a `ConfigEnvelope` message as the `Payload data` field. The proto for `ConfigEnvelope` is defined as follows:

```
message ConfigEnvelope {
    Config config = 1;
    Envelope last_update = 2;
}
```

The `last_update` field is defined below in the **Updates to configuration** section, but is only necessary when validating the configuration, not reading it. Instead, the currently committed configuration is stored in the `config` field, containing a `Config` message.

```
message Config {
    uint64 sequence = 1;
    ConfigGroup channel_group = 2;
}
```

The `sequence` number is incremented by one for each committed configuration. The `channel_group` field is the root group which contains the configuration. The `ConfigGroup` structure is recursively defined, and builds a tree of groups, each of which contains values and policies. It is defined as follows:

```
message ConfigGroup {
    uint64 version = 1;
    map<string,ConfigGroup> groups = 2;
    map<string,ConfigValue> values = 3;
    map<string,ConfigPolicy> policies = 4;
    string mod_policy = 5;
}
```

Because `ConfigGroup` is a recursive structure, it has hierarchical arrangement. The following example is expressed for clarity in Go syntax.

```
// Assume the following groups are defined
var root, child1, child2, grandChild1, grandChild2, grandChild3 *ConfigGroup

// Set the following values
root.Groups["child1"] = child1
root.Groups["child2"] = child2
child1.Groups["grandChild1"] = grandChild1
child2.Groups["grandChild2"] = grandChild2
child2.Groups["grandChild3"] = grandChild3

// The resulting config structure of groups looks like:
// root:
//     child1:
//         grandChild1
//     child2:
//         grandChild2
//         grandChild3
```

Each group defines a level in the config hierarchy, and each group has an associated set of values (indexed by string key) and policies (also indexed by string key).

Values are defined by:

```
message ConfigValue {
    uint64 version = 1;
    bytes value = 2;
    string mod_policy = 3;
}
```

Policies are defined by:

```
message ConfigPolicy {
    uint64 version = 1;
    Policy policy = 2;
```

(continues on next page)

(continued from previous page)

```
    string mod_policy = 3;
}
```

Note that Values, Policies, and Groups all have a `version` and a `mod_policy`. The `version` of an element is incremented each time that element is modified. The `mod_policy` is used to govern the required signatures to modify that element. For Groups, modification is adding or removing elements to the Values, Policies, or Groups maps (or changing the `mod_policy`). For Values and Policies, modification is changing the Value and Policy fields respectively (or changing the `mod_policy`). Each element's `mod_policy` is evaluated in the context of the current level of the config. Consider the following example mod policies defined at `Channel.Groups["Application"]` (Here, we use the Go map reference syntax, so `Channel.Groups["Application"].Policies["policy1"]` refers to the base `Channel` group's `Application` group's `Policies` map's `policy1` policy.)

- `policy1` maps to `Channel.Groups["Application"].Policies["policy1"]`
- `Org1/policy2` maps to `Channel.Groups["Application"].Groups["Org1"].Policies["policy2"]`
- `/Channel/policy3` maps to `Channel.Policies["policy3"]`

Note that if a `mod_policy` references a policy which does not exist, the item cannot be modified.

9.4.2 Configuration updates

Configuration updates are submitted as an `Envelope` message of type `HeaderType_CONFIG_UPDATE`. The `Payload` `data` of the transaction is a marshaled `ConfigUpdateEnvelope`. The `ConfigUpdateEnvelope` is defined as follows:

```
message ConfigUpdateEnvelope {
    bytes config_update = 1;
    repeated ConfigSignature signatures = 2;
}
```

The `signatures` field contains the set of signatures which authorizes the config update. Its message definition is:

```
message ConfigSignature {
    bytes signature_header = 1;
    bytes signature = 2;
}
```

The `signature_header` is as defined for standard transactions, while the signature is over the concatenation of the `signature_header` bytes and the `config_update` bytes from the `ConfigUpdateEnvelope` message.

The `ConfigUpdateEnvelope` `config_update` bytes are a marshaled `ConfigUpdate` message which is defined as follows:

```
message ConfigUpdate {
    string channel_id = 1;
    ConfigGroup read_set = 2;
    ConfigGroup write_set = 3;
}
```

The `channel_id` is the channel ID the update is bound for, this is necessary to scope the signatures which support this reconfiguration.

The `read_set` specifies a subset of the existing configuration, specified sparsely where only the `version` field is set and no other fields must be populated. The particular `ConfigValue` `value` or `ConfigPolicy` `policy` fields

should never be set in the read_set. The ConfigGroup may have a subset of its map fields populated, so as to reference an element deeper in the config tree. For instance, to include the Application group in the read_set, its parent (the Channel group) must also be included in the read set, but, the Channel group does not need to populate all of the keys, such as the Orderer group key, or any of the values or policies keys.

The write_set specifies the pieces of configuration which are modified. Because of the hierarchical nature of the configuration, a write to an element deep in the hierarchy must contain the higher level elements in its write_set as well. However, for any element in the write_set which is also specified in the read_set at the same version, the element should be specified sparsely, just as in the read_set.

For example, given the configuration:

```
Channel: (version 0)
    Orderer (version 0)
    Application (version 3)
       Org1 (version 2)
```

To submit a configuration update which modifies Org1, the read_set would be:

```
Channel: (version 0)
    Application: (version 3)
```

and the write_set would be

```
Channel: (version 0)
    Application: (version 3)
        Org1 (version 3)
```

When the CONFIG_UPDATE is received, the orderer computes the resulting CONFIG by doing the following:

1. Verifies the channel_id and read_set. All elements in the read_set must exist at the given versions.
2. Computes the update set by collecting all elements in the write_set which do not appear at the same version in the read_set.
3. Verifies that each element in the update set increments the version number of the element update by exactly 1.
4. Verifies that the signature set attached to the ConfigUpdateEnvelope satisfies the mod_policy for each element in the update set.
5. Computes a new complete version of the config by applying the update set to the current config.
6. Writes the new config into a ConfigEnvelope which includes the CONFIG_UPDATE as the last_update field and the new config encoded in the config field, along with the incremented sequence value.
7. Writes the new ConfigEnvelope into a Envelope of type CONFIG, and ultimately writes this as the sole transaction in a new configuration block.

When the peer (or any other receiver for Deliver) receives this configuration block, it should verify that the config was appropriately validated by applying the last_update message to the current config and verifying that the orderer-computed config field contains the correct new configuration.

9.4.3 Permitted configuration groups and values

Any valid configuration is a subset of the following configuration. Here we use the notation peer.<MSG> to define a ConfigValue whose value field is a marshaled proto message of name <MSG> defined in fabric-protos/peer/configuration.proto. The notations common.<MSG>, msp.<MSG>, and orderer.<MSG> correspond similarly, but with their messages defined in fabric-protos/common/configuration.proto,

`fabric-protos/msp/mspconfig.proto`, and `fabric-protos/orderer/configuration.proto` respectively.

Note, that the keys `{{org_name}}` and `{{consortium_name}}` represent arbitrary names, and indicate an element which may be repeated with different names.

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Application":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                        "AnchorPeers":peer.AnchorPeers,
                    },
                },
            },
        },
        "Orderer":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                    },
                },
            },

            Values:map<string, *ConfigValue> {
                "ConsensusType":orderer.ConsensusType,
                "BatchSize":orderer.BatchSize,
                "BatchTimeout":orderer.BatchTimeout,
                "KafkaBrokers":orderer.KafkaBrokers,
            },
        },
        "Consortiums":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{consortium_name}}:&ConfigGroup{
                    Groups:map<string, *ConfigGroup> {
                        {{org_name}}:&ConfigGroup{
                            Values:map<string, *ConfigValue>{
                                "MSP":msp.MSPConfig,
                            },
                        },
                    },
                    Values:map<string, *ConfigValue> {
                        "ChannelCreationPolicy":common.Policy,
                    }
                },
            },
        },
    },

    Values: map<string, *ConfigValue> {
        "HashingAlgorithm":common.HashingAlgorithm,
        "BlockHashingDataStructure":common.BlockDataHashingStructure,
        "Consortium":common.Consortium,
        "OrdererAddresses":common.OrdererAddresses,
    },
}
```

9.4.4 Orderer system channel configuration

The ordering system channel needs to define ordering parameters, and consortiums for creating channels. There must be exactly one ordering system channel for an ordering service, and it is the first channel to be created (or more accurately bootstrapped). It is recommended never to define an Application section inside of the ordering system channel genesis configuration, but may be done for testing. Note that any member with read access to the ordering system channel may see all channel creations, so this channel's access should be restricted.

The ordering parameters are defined as the following subset of config:

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Orderer":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                    },
                },
            },

            Values:map<string, *ConfigValue> {
                "ConsensusType":orderer.ConsensusType,
                "BatchSize":orderer.BatchSize,
                "BatchTimeout":orderer.BatchTimeout,
                "KafkaBrokers":orderer.KafkaBrokers,
            },
        },
    },
```

Each organization participating in ordering has a group element under the `Orderer` group. This group defines a single parameter `MSP` which contains the cryptographic identity information for that organization. The `Values` of the `Orderer` group determine how the ordering nodes function. They exist per channel, so `orderer.BatchTimeout` for instance may be specified differently on one channel than another.

At startup, the orderer is faced with a filesystem which contains information for many channels. The orderer identifies the system channel by identifying the channel with the consortiums group defined. The consortiums group has the following structure.

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Consortiums":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{consortium_name}}:&ConfigGroup{
                    Groups:map<string, *ConfigGroup> {
                        {{org_name}}:&ConfigGroup{
                            Values:map<string, *ConfigValue>{
                                "MSP":msp.MSPConfig,
                            },
                        },
                    },
                    Values:map<string, *ConfigValue> {
                        "ChannelCreationPolicy":common.Policy,
                    }
                },
            },
        },
```

(continues on next page)

(continued from previous page)

```
    },
},
```

Note that each consortium defines a set of members, just like the organizational members for the ordering orgs. Each consortium also defines a `ChannelCreationPolicy`. This is a policy which is applied to authorize channel creation requests. Typically, this value will be set to an `ImplicitMetaPolicy` requiring that the new members of the channel sign to authorize the channel creation. More details about channel creation follow later in this document.

9.4.5 Application channel configuration

Application configuration is for channels which are designed for application type transactions. It is defined as follows:

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Application":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                        "AnchorPeers":peer.AnchorPeers,
                    },
                },
            },
        },
    },
}
```

Just like with the `Orderer` section, each organization is encoded as a group. However, instead of only encoding the `MSP` identity information, each org additionally encodes a list of `AnchorPeers`. This list allows the peers of different organizations to contact each other for peer gossip networking.

The application channel encodes a copy of the orderer orgs and consensus options to allow for deterministic updating of these parameters, so the same `Orderer` section from the orderer system channel configuration is included. However from an application perspective this may be largely ignored.

9.4.6 Channel creation

When the orderer receives a `CONFIG_UPDATE` for a channel which does not exist, the orderer assumes that this must be a channel creation request and performs the following.

1. The orderer identifies the consortium which the channel creation request is to be performed for. It does this by looking at the `Consortium` value of the top level group.
2. The orderer verifies that the organizations included in the `Application` group are a subset of the organizations included in the corresponding consortium and that the `ApplicationGroup` is set to `version 1`.
3. The orderer verifies that if the consortium has members, that the new channel also has application members (creation consortiums and channels with no members is useful for testing only).
4. The orderer creates a template configuration by taking the `Orderer` group from the ordering system channel, and creating an `Application` group with the newly specified members and specifying its `mod_policy` to be the `ChannelCreationPolicy` as specified in the consortium config. Note that the policy is evaluated in the context of the new configuration, so a policy requiring `ALL` members, would require signatures from all the new channel members, not all the members of the consortium.

5. The orderer then applies the `CONFIG_UPDATE` as an update to this template configuration. Because the `CONFIG_UPDATE` applies modifications to the `Application` group (its `version` is `1`), the config code validates these updates against the `ChannelCreationPolicy`. If the channel creation contains any other modifications, such as to an individual org's anchor peers, the corresponding mod policy for the element will be invoked.
6. The new `CONFIG` transaction with the new channel config is wrapped and sent for ordering on the ordering system channel. After ordering, the channel is created.

9.5 Endorsement policies

Every chaincode has an endorsement policy which specifies the set of peers on a channel that must execute chaincode and endorse the execution results in order for the transaction to be considered valid. These endorsement policies define the organizations (through their peers) who must "endorse" (i.e., approve of) the execution of a proposal.

Note: Recall that **state**, represented by key-value pairs, is separate from blockchain data. For more on this, check out our *Ledger* documentation.

As part of the transaction validation step performed by the peers, each validating peer checks to make sure that the transaction contains the appropriate **number** of endorsements and that they are from the expected sources (both of these are specified in the endorsement policy). The endorsements are also checked to make sure they're valid (i.e., that they are valid signatures from valid certificates).

9.5.1 Multiple ways to require endorsement

By default, endorsement policies are specified in the chaincode definition, which is agreed to by channel members and then committed to a channel (that is, one endorsement policy covers all of the state associated with a chaincode).

For private data collections, you can also specify an endorsement policy at the private data collection level, which would override the chaincode level endorsement policy for any keys in the private data collection, thereby further restricting which organizations can write to a private data collection.

Finally, there are cases where it may be necessary for a particular public channel state or private data collection state (a particular key-value pair, in other words) to have a different endorsement policy. This **state-based endorsement** allows the chaincode-level or collection-level endorsement policies to be overridden by a different policy for the specified keys.

To illustrate the circumstances in which the various types of endorsement policies might be used, consider a channel on which cars are being exchanged. The "creation" — also known as "issuance" – of a car as an asset that can be traded (putting the key-value pair that represents it into the world state, in other words) would have to satisfy the chaincode-level endorsement policy. To see how to set a chaincode-level endorsement policy, check out the section below.

If the key representing the car requires a specific endorsement policy, it can be defined either when the car is created or afterwards. There are a number of reasons why it might be necessary or preferable to set a state-specific endorsement policy. The car might have historical importance or value that makes it necessary to have the endorsement of a licensed appraiser. Also, the owner of the car (if they're a member of the channel) might also want to ensure that their peer signs off on a transaction. In both cases, **an endorsement policy is required for a particular asset that is different from the default endorsement policies for the other assets associated with that chaincode.**

We'll show you how to define a state-based endorsement policy in a subsequent section. But first, let's see how we set a chaincode-level endorsement policy.

9.5.2 Setting chaincode-level endorsement policies

Chaincode-level endorsement policies are agreed to by channel members when they approve a chaincode definition for their organization. A sufficient number of channel members need to approve a chaincode definition to meet the `Channel/Application/LifecycleEndorsement` policy, which by default is set to a majority of channel members, before the definition can be committed to the channel. Once the definition has been committed, the chaincode is ready to use. Any invoke of the chaincode that writes data to the ledger will need to be validated by enough channel members to meet the endorsement policy.

You can specify an endorsement policy for a chainocode using the Fabric SDKs. For an example, visit the How to install and start your chaincode in the Node.js SDK documentation. You can also create an endorsement policy from your CLI when you approve and commit a chaincode definition with the Fabric peer binaries by using the `---signature-policy` flag.

Note: Don't worry about the policy syntax (`'Org1.member'`, et all) right now. We'll talk more about the syntax in the next section.

For example:

```
peer lifecycle chaincode approveformyorg --channelID mychannel ---signature-policy
↪"AND('Org1.member', 'Org2.member')" --name mycc --version 1.0 --package-id mycc_
↪1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173 --sequence 1 --
↪tls true --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪example.com-cert.pem --waitForEvent
```

The above command approves the chaincode definition of `mycc` with the policy `AND('Org1.member', 'Org2.member')` which would require that a member of both Org1 and Org2 sign the transaction. After a sufficient number of channel members approve a chaincode definition for `mycc`, the definition and endorsement policy can be committed to the channel using the command below:

```
peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID mychannel ---
↪signature-policy "AND('Org1.member', 'Org2.member')" --name mycc --version 1.0 --
↪sequence 1 --init-required --tls true --cafile /opt/gopath/src/github.com/
↪hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.
↪example.com/msp/tlscacerts/tlsca.example.com-cert.pem --waitForEvent --
↪peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/src/github.
↪com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.
↪org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --
↪tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

Notice that, if the identity classification is enabled (see *Membership Service Providers (MSP)*), one can use the `PEER` role to restrict endorsement to only peers.

For example:

```
peer lifecycle chaincode approveformyorg --channelID mychannel ---signature-policy
↪"AND('Org1.peer', 'Org2.peer')" --name mycc --version 1.0 --package-id mycc_
↪1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173 --sequence 1 --
↪tls true --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪example.com-cert.pem --waitForEvent
```

In addition to the specifying an endorsement policy from the CLI or SDK, a chaincode can also use policies in the channel configuration as endorsement policies. You can use the ‘‘–channel-config-policy‘‘flag to select a channel

policy with format used by the channel configuration and by ACLs.

For example:

```
peer lifecycle chaincode approveformyorg --channelID mychannel --channel-config-
↪policy Channel/Application/Admins --name mycc --version 1.0 --package-id mycc_
↪1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173 --sequence 1 --
↪tls true --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪example.com-cert.pem --waitForEvent
```

If you do not specify a policy, the chaincode definition will use the `Channel/Application/Endorsement` policy by default, which requires that a transaction be validated by a majority of channel members. This policy depends on the membership of the channel, so it will be updated automatically when organizations are added or removed from a channel. One advantage of using channel policies is that they can be written to be updated automatically with channel membership.

If you specify an endorsement policy using the `---signature-policy` flag or the SDK, you will need to update the policy when organizations join or leave the channel. A new organization added to the channel after the chaincode has been defined will be able to query a chaincode (provided the query has appropriate authorization as defined by channel policies and any application level checks enforced by the chaincode) but will not be able to execute or endorse the chaincode. Only organizations listed in the endorsement policy syntax will be able sign transactions.

Endorsement policy syntax

As you can see above, policies are expressed in terms of principals ("principals" are identities matched to a role). Principals are described as `'MSP.ROLE'`, where `MSP` represents the required MSP ID and `ROLE` represents one of the four accepted roles: `member`, `admin`, `client`, and `peer`.

Here are a few examples of valid principals:

- `'Org0.admin'`: any administrator of the `Org0` MSP
- `'Org1.member'`: any member of the `Org1` MSP
- `'Org1.client'`: any client of the `Org1` MSP
- `'Org1.peer'`: any peer of the `Org1` MSP

The syntax of the language is:

`EXPR(E[, E...])`

Where `EXPR` is either `AND`, `OR`, or `OutOf`, and `E` is either a principal (with the syntax described above) or another nested call to `EXPR`.

For example:

- `AND('Org1.member', 'Org2.member', 'Org3.member')` requests one signature from each of the three principals.
- `OR('Org1.member', 'Org2.member')` requests one signature from either one of the two principals.
- `OR('Org1.member', AND('Org2.member', 'Org3.member'))` requests either one signature from a member of the `Org1` MSP or one signature from a member of the `Org2` MSP and one signature from a member of the `Org3` MSP.
- `OutOf(1, 'Org1.member', 'Org2.member')`, which resolves to the same thing as `OR('Org1.member', 'Org2.member')`.

- Similarly, `OutOf(2, 'Org1.member', 'Org2.member')` is equivalent to `AND('Org1.member', 'Org2.member')`, and `OutOf(2, 'Org1.member', 'Org2.member', 'Org3.member')` is equivalent to `OR(AND('Org1.member', 'Org2.member'), AND('Org1.member', 'Org3.member'), AND('Org2.member', 'Org3.member'))`.

9.5.3 Setting collection-level endorsement policies

Similar to chaincode-level endorsement policies, when you approve and commit a chaincode definition, you can also specify the chaincode's private data collections and corresponding collection-level endorsement policies. If a collection-level endorsement policy is set, transactions that write to a private data collection key will require that the specified organization peers have endorsed the transaction.

You can use collection-level endorsement policies to restrict which organization peers can write to the private data collection key namespace, for example to ensure that non-authorized organizations cannot write to a collection, and to have confidence that any state in a private data collection has been endorsed by the required collection organization(s).

The collection-level endorsement policy may be less restrictive or more restrictive than the chaincode-level endorsement policy and the collection's private data distribution policy. For example a majority of organizations may be required to endorse a chaincode transaction, but a specific organization may be required to endorse a transaction that includes a key in a specific collection.

The syntax for collection-level endorsement policies exactly matches the syntax for chaincode-level endorsement policies — in the collection configuration you can specify an `endorsementPolicy` with either a `signaturePolicy` or `channelConfigPolicy`. For more details see *Private Data*.

9.5.4 Setting key-level endorsement policies

Setting regular chaincode-level or collection-level endorsement policies is tied to the lifecycle of the corresponding chaincode. They can only be set or modified when defining the chaincode on a channel.

In contrast, key-level endorsement policies can be set and modified in a more granular fashion from within a chaincode. The modification is part of the read-write set of a regular transaction.

The shim API provides the following functions to set and retrieve an endorsement policy for/from a regular key.

Note: `ep` below stands for the "endorsement policy", which can be expressed either by using the same syntax described above or by using the convenience function described below. Either method will generate a binary version of the endorsement policy that can be consumed by the basic shim API.

```
SetStateValidationParameter(key string, ep []byte) error
GetStateValidationParameter(key string) ([]byte, error)
```

For keys that are part of *Private data* in a collection the following functions apply:

```
SetPrivateDataValidationParameter(collection, key string, ep []byte) error
GetPrivateDataValidationParameter(collection, key string) ([]byte, error)
```

To help set endorsement policies and marshal them into validation parameter byte arrays, the Go shim provides an extension with convenience functions that allow the chaincode developer to deal with endorsement policies in terms of the MSP identifiers of organizations, see KeyEndorsementPolicy:

```
type KeyEndorsementPolicy interface {
    // Policy returns the endorsement policy as bytes
    Policy() ([]byte, error)
```

(continues on next page)

(continued from previous page)

```
    // AddOrgs adds the specified orgs to the list of orgs that are required
    // to endorse
    AddOrgs(roleType RoleType, organizations ...string) error

    // DelOrgs delete the specified channel orgs from the existing key-level␣
↪endorsement
    // policy for this KVS key. If any org is not present, an error will be returned.
    DelOrgs(organizations ...string) error

    // ListOrgs returns an array of channel orgs that are required to endorse changes
    ListOrgs() ([]string)
}
```

For example, to set an endorsement policy for a key where two specific orgs are required to endorse the key change, pass both org `MSPIDs` to `AddOrgs()`, and then call `Policy()` to construct the endorsement policy byte array that can be passed to `SetStateValidationParameter()`.

To add the shim extension to your chaincode as a dependency, see *Managing external dependencies for chaincode written in Go*.

9.5.5 Validation

At commit time, setting a value of a key is no different from setting the endorsement policy of a key — both update the state of the key and are validated based on the same rules.

Validation	no validation parameter set	validation parameter set
modify value	check chaincode or collection ep	check key-level ep
modify key-level ep	check chaincode or collection ep	check key-level ep

As we discussed above, if a key is modified and no key-level endorsement policy is present, the chaincode-level or collection-level endorsement policy applies by default. This is also true when a key-level endorsement policy is set for a key for the first time — the new key-level endorsement policy must first be endorsed according to the pre-existing chaincode-level or collection-level endorsement policy.

If a key is modified and a key-level endorsement policy is present, the key-level endorsement policy overrides the chaincode-level or collection-level endorsement policy. In practice, this means that the key-level endorsement policy can be either less restrictive or more restrictive than the chaincode-level or collection-level endorsement policies. Because the chaincode-level or collection-level endorsement policy must be satisfied in order to set a key-level endorsement policy for the first time, no trust assumptions have been violated.

If a key's endorsement policy is removed (set to nil), the chaincode-level or collection-level endorsement policy becomes the default again.

If a transaction modifies multiple keys with different associated key-level endorsement policies, all of these policies need to be satisfied in order for the transaction to be valid.

9.6 Pluggable transaction endorsement and validation

9.6.1 Motivation

When a transaction is validated at time of commit, the peer performs various checks before applying the state changes that come with the transaction itself:

- Validating the identities that signed the transaction.
- Verifying the signatures of the endorsers on the transaction.
- Ensuring the transaction satisfies the endorsement policies of the namespaces of the corresponding chaincodes.

There are use cases which demand custom transaction validation rules different from the default Fabric validation rules, such as:

- **UTXO (Unspent Transaction Output):** When the validation takes into account whether the transaction doesn't double spend its inputs.
- **Anonymous transactions:** When the endorsement doesn't contain the identity of the peer, but a signature and a public key are shared that can't be linked to the peer's identity.

9.6.2 Pluggable endorsement and validation logic

Fabric allows for the implementation and deployment of custom endorsement and validation logic into the peer to be associated with chaincode handling in a pluggable manner. This logic can be either compiled into the peer as built in selectable logic, or compiled and deployed alongside the peer as a Go plugin.

By default, A chaincode will use the built in endorsement and validation logic. However, users have the option of selecting custom endorsement and validation plugins as part of the chaincode definition. An administrator can extend the endorsement/validation logic available to the peer by customizing the peer's local configuration.

9.6.3 Configuration

Each peer has a local configuration (`core.yaml`) that declares a mapping between the endorsement/validation logic name and the implementation that is to be run.

The default logic are called `ESCC` (with the "E" standing for endorsement) and `VSCC` (validation), and they can be found in the peer local configuration in the `handlers` section:

```
handlers:
    endorsers:
      escc:
        name: DefaultEndorsement
    validators:
      vscc:
        name: DefaultValidation
```

When the endorsement or validation implementation is compiled into the peer, the `name` property represents the initialization function that is to be run in order to obtain the factory that creates instances of the endorsement/validation logic.

The function is an instance method of the `HandlerLibrary` construct under `core/handlers/library/library.go` and in order for custom endorsement or validation logic to be added, this construct needs to be extended with any additional methods.

Since this is cumbersome and poses a deployment challenge, one can also deploy custom endorsement and validation as a Go plugin by adding another property under the `name` called `library`.

For example, if we have custom endorsement and validation logic which is implemented as a plugin, we would have the following entries in the configuration in `core.yaml`:

```
handlers:
    endorsers:
      escc:
```

(continues on next page)

(continued from previous page)

```
        name: DefaultEndorsement
      custom:
        name: customEndorsement
        library: /etc/hyperledger/fabric/plugins/customEndorsement.so
    validators:
      vscc:
        name: DefaultValidation
      custom:
        name: customValidation
        library: /etc/hyperledger/fabric/plugins/customValidation.so
```

And we'd have to place the `.so` plugin files in the peer's local file system.

The name of the custom plugin needs to be referenced by the chaincode definition to be used by the chaincode. If you are using the peer CLI to approve the chaincode definition, use the `--escc` and `--vscc` flag to select the name of the custom endorsement or validation library. If you are using the Fabric SDK for Node.js, visit How to install and start your chaincode. For more information, see *Fabric chaincode lifecycle*.

Note: Hereafter, custom endorsement or validation logic implementation is going to be referred to as "plugins", even if they are compiled into the peer.

9.6.4 Endorsement plugin implementation

To implement an endorsement plugin, one must implement the `Plugin` interface found in `core/handlers/endorsement/api/endorsement.go`:

```
// Plugin endorses a proposal response
type Plugin interface {
    // Endorse signs the given payload(ProposalResponsePayload bytes), and optionally␣
↪mutates it.
    // Returns:
    // The Endorsement: A signature over the payload, and an identity that is used to␣
↪verify the signature
    // The payload that was given as input (could be modified within this function)
    // Or error on failure
    Endorse(payload []byte, sp *peer.SignedProposal) (*peer.Endorsement, []byte,␣
↪error)

    // Init injects dependencies into the instance of the Plugin
    Init(dependencies ...Dependency) error
}
```

An endorsement plugin instance of a given plugin type (identified either by the method name as an instance method of the `HandlerLibrary` or by the plugin `.so` file path) is created for each channel by having the peer invoke the `New` method in the `PluginFactory` interface which is also expected to be implemented by the plugin developer:

```
// PluginFactory creates a new instance of a Plugin
type PluginFactory interface {
    New() Plugin
}
```

The `Init` method is expected to receive as input all the dependencies declared under `core/handlers/endorsement/api/`, identified as embedding the `Dependency` interface.

After the creation of the `Plugin` instance, the `Init` method is invoked on it by the peer with the `dependencies` passed as parameters.

Currently Fabric comes with the following dependencies for endorsement plugins:

- `SigningIdentityFetcher`: Returns an instance of `SigningIdentity` based on a given signed proposal:

```
// SigningIdentity signs messages and serializes its public identity to bytes
type SigningIdentity interface {
    // Serialize returns a byte representation of this identity which is used to␣
↪verify
    // messages signed by this SigningIdentity
    Serialize() ([]byte, error)

    // Sign signs the given payload and returns a signature
    Sign([]byte) ([]byte, error)
}
```

- `StateFetcher`: Fetches a **State** object which interacts with the world state:

```
// State defines interaction with the world state
type State interface {
    // GetPrivateDataMultipleKeys gets the values for the multiple private data items␣
↪in a single call
    GetPrivateDataMultipleKeys(namespace, collection string, keys []string) ([][]byte,
↪ error)

    // GetStateMultipleKeys gets the values for multiple keys in a single call
    GetStateMultipleKeys(namespace string, keys []string) ([][]byte, error)

    // GetTransientByTXID gets the values private data associated with the given txID
    GetTransientByTXID(txID string) ([]*rwset.TxPvtReadWriteSet, error)

    // Done releases resources occupied by the State
    Done()
 }
```

9.6.5 Validation plugin implementation

To implement a validation plugin, one must implement the `Plugin` interface found in `core/handlers/validation/api/validation.go`:

```
// Plugin validates transactions
type Plugin interface {
    // Validate returns nil if the action at the given position inside the transaction
    // at the given position in the given block is valid, or an error if not.
    Validate(block *common.Block, namespace string, txPosition int, actionPosition␣
↪int, contextData ...ContextDatum) error

    // Init injects dependencies into the instance of the Plugin
    Init(dependencies ...Dependency) error
}
```

Each `ContextDatum` is additional runtime-derived metadata that is passed by the peer to the validation plugin. Currently, the only `ContextDatum` that is passed is one that represents the endorsement policy of the chaincode:

```
 // SerializedPolicy defines a serialized policy
type SerializedPolicy interface {
      validation.ContextDatum

      // Bytes returns the bytes of the SerializedPolicy
      Bytes() []byte
 }
```

A validation plugin instance of a given plugin type (identified either by the method name as an instance method of the `HandlerLibrary` or by the plugin `.so` file path) is created for each channel by having the peer invoke the `New` method in the `PluginFactory` interface which is also expected to be implemented by the plugin developer:

```
// PluginFactory creates a new instance of a Plugin
type PluginFactory interface {
    New() Plugin
}
```

The `Init` method is expected to receive as input all the dependencies declared under `core/handlers/validation/api/`, identified as embedding the `Dependency` interface.

After the creation of the `Plugin` instance, the **Init** method is invoked on it by the peer with the dependencies passed as parameters.

Currently Fabric comes with the following dependencies for validation plugins:

- `IdentityDeserializer`: Converts byte representation of identities into `Identity` objects that can be used to verify signatures signed by them, be validated themselves against their corresponding MSP, and see whether they satisfy a given **MSP Principal**. The full specification can be found in `core/handlers/validation/api/identities/identities.go`.
- `PolicyEvaluator`: Evaluates whether a given policy is satisfied:

```
// PolicyEvaluator evaluates policies
type PolicyEvaluator interface {
    validation.Dependency

    // Evaluate takes a set of SignedData and evaluates whether this set of␣
↪signatures satisfies
    // the policy with the given bytes
    Evaluate(policyBytes []byte, signatureSet []*common.SignedData) error
}
```

- `StateFetcher`: Fetches a `State` object which interacts with the world state:

```
// State defines interaction with the world state
type State interface {
    // GetStateMultipleKeys gets the values for multiple keys in a single call
    GetStateMultipleKeys(namespace string, keys []string) ([][]byte, error)

    // GetStateRangeScanIterator returns an iterator that contains all the key-values␣
↪between given key ranges.
    // startKey is included in the results and endKey is excluded. An empty startKey␣
↪refers to the first available key
    // and an empty endKey refers to the last available key. For scanning all the␣
↪keys, both the startKey and the endKey
    // can be supplied as empty strings. However, a full scan should be used␣
↪judiciously for performance reasons.
    // The returned ResultsIterator contains results of type *KV which is defined in␣
↪fabric-protos/ledger/queryresult.
```

(continues on next page)

(continued from previous page)

```
    GetStateRangeScanIterator(namespace string, startKey string, endKey string)␣
↪(ResultsIterator, error)

    // GetStateMetadata returns the metadata for given namespace and key
    GetStateMetadata(namespace, key string) (map[string][]byte, error)

    // GetPrivateDataMetadata gets the metadata of a private data item identified by␣
↪a tuple <namespace, collection, key>
    GetPrivateDataMetadata(namespace, collection, key string) (map[string][]byte,␣
↪error)

    // Done releases resources occupied by the State
    Done()
}
```

9.6.6 Important notes

- **Validation plugin consistency across peers:** In future releases, the Fabric channel infrastructure would guarantee that the same validation logic is used for a given chaincode by all peers in the channel at any given blockchain height in order to eliminate the chance of mis-configuration which would might lead to state divergence among peers that accidentally run different implementations. However, for now it is the sole responsibility of the system operators and administrators to ensure this doesn't happen.
- **Validation plugin error handling:** Whenever a validation plugin can't determine whether a given transaction is valid or not, because of some transient execution problem like inability to access the database, it should return an error of type **ExecutionFailureError** that is defined in `core/handlers/validation/api/validation.go`. Any other error that is returned, is treated as an endorsement policy error and marks the transaction as invalidated by the validation logic. However, if an `ExecutionFailureError` is returned, the chain processing halts instead of marking the transaction as invalid. This is to prevent state divergence between different peers.
- **Error handling for private metadata retrieval**: In case a plugin retrieves metadata for private data by making use of the `StateFetcher` interface, it is important that errors are handled as follows: `CollConfigNotDefinedError` and `InvalidCollNameError`, signalling that the specified collection does not exist, should be handled as deterministic errors and should not lead the plugin to return an `ExecutionFailureError`.
- **Importing Fabric code into the plugin**: Importing code that belongs to Fabric other than protobufs as part of the plugin is highly discouraged, and can lead to issues when the Fabric code changes between releases, or can cause inoperability issues when running mixed peer versions. Ideally, the plugin code should only use the dependencies given to it, and should import the bare minimum other than protobufs.

9.7 Access Control Lists (ACL)

9.7.1 What is an Access Control List?

Note: This topic deals with access control and policies on a channel administration level. To learn about access control within a chaincode, check out our chaincode for developers tutorial.

Fabric uses access control lists (ACLs) to manage access to resources by associating a Policy with a resource. Fabric contains a number of default ACLs. In this document, we'll talk about how they're formatted and how the defaults can be overridden.

But before we can do that, it's necessary to understand a little about resources and policies.

Resources

Users interact with Fabric by targeting a user chaincode, or an events stream source, or system chaincode that are called in the background. As such, these endpoints are considered "resources" on which access control should be exercised.

Application developers need to be aware of these resources and the default policies associated with them. The complete list of these resources are found in `configtx.yaml`. You can look at a sample `configtx.yaml` file here.

The resources named in `configtx.yaml` is an exhaustive list of all internal resources currently defined by Fabric. The loose convention adopted there is `<component>/<resource>`. So `cscc/GetConfigBlock` is the resource for the `GetConfigBlock` call in the `CSCC` component.

Policies

Policies are fundamental to the way Fabric works because they allow the identity (or set of identities) associated with a request to be checked against the policy associated with the resource needed to fulfill the request. Endorsement policies are used to determine whether a transaction has been appropriately endorsed. The policies defined in the channel configuration are referenced as modification policies as well as for access control, and are defined in the channel configuration itself.

Policies can be structured in one of two ways: as `Signature` policies or as an `ImplicitMeta` policy.

`Signature` policies

These policies identify specific users who must sign in order for a policy to be satisfied. For example:

```
Policies:
  MyPolicy:
    Type: Signature
    Rule: "Org1.Peer OR Org2.Peer"
```

This policy construct can be interpreted as: *the policy named `MyPolicy` can only be satisfied by the signature of an identity with role of "a peer from Org1" or "a peer from Org2".*

Signature policies support arbitrary combinations of `AND`, `OR`, and `NOutOf`, allowing the construction of extremely powerful rules like: "An admin of org A and two other admins, or 11 of 20 org admins".

`ImplicitMeta` policies

`ImplicitMeta` policies aggregate the result of policies deeper in the configuration hierarchy that are ultimately defined by `Signature` policies. They support default rules like "A majority of the organization admins". These policies use a different but still very simple syntax as compared to `Signature` policies: `<ALL|ANY|MAJORITY> <sub_policy>`.

For example: `ANY Readers` or `MAJORITY Admins`.

Note that in the default policy configuration `Admins` have an operational role. Policies that specify that only Admins — or some subset of Admins — have access to a resource will tend to be for sensitive or operational aspects of the network (such as instantiating chaincode on a channel). `Writers` will tend to be able to propose ledger updates, such as a transaction, but will not typically have administrative permissions. `Readers` have a passive role. They can access information but do not have the permission to propose ledger updates nor do can they perform administrative

tasks. These default policies can be added to, edited, or supplemented, for example by the new `peer` and `client` roles (if you have `NodeOU` support).

Here's an example of an `ImplicitMeta` policy structure:

```
Policies:
  AnotherPolicy:
    Type: ImplicitMeta
    Rule: "MAJORITY Admins"
```

Here, the policy `AnotherPolicy` can be satisfied by the `MAJORITY` of `Admins`, where `Admins` is eventually being specified by lower level `Signature` policy.

Where is access control specified?

Access control defaults exist inside `configtx.yaml`, the file that `configtxgen` uses to build channel configurations.

Access control can be updated in one of two ways, either by editing `configtx.yaml` itself, which will be used when creating new channel configurations, or by updating access control in the channel configuration of an existing channel.

9.7.2 How ACLs are formatted in `configtx.yaml`

ACLs are formatted as a key-value pair consisting of a resource function name followed by a string. To see what this looks like, reference this sample configtx.yaml file.

Two excerpts from this sample:

```
# ACL policy for invoking chaincodes on peer
peer/Propose: /Channel/Application/Writers
```

```
# ACL policy for sending block events
event/Block: /Channel/Application/Readers
```

These ACLs define that access to `peer/Propose` and `event/Block` resources is restricted to identities satisfying the policy defined at the canonical path `/Channel/Application/Writers` and `/Channel/Application/Readers`, respectively.

Updating ACL defaults in `configtx.yaml`

In cases where it will be necessary to override ACL defaults when bootstrapping a network, or to change the ACLs before a channel has been bootstrapped, the best practice will be to update `configtx.yaml`.

Let's say you want to modify the `peer/Propose` ACL default — which specifies the policy for invoking chaincodes on a peer – from `/Channel/Application/Writers` to a policy called `MyPolicy`.

This is done by adding a policy called `MyPolicy` (it could be called anything, but for this example we'll call it `MyPolicy`). The policy is defined in the `Application.Policies` section inside `configtx.yaml` and specifies a rule to be checked to grant or deny access to a user. For this example, we'll be creating a `Signature` policy identifying `SampleOrg.admin`.

```
Policies: &ApplicationDefaultPolicies
    Readers:
        Type: ImplicitMeta
        Rule: "ANY Readers"
    Writers:
        Type: ImplicitMeta
        Rule: "ANY Writers"
    Admins:
        Type: ImplicitMeta
        Rule: "MAJORITY Admins"
    MyPolicy:
        Type: Signature
        Rule: "OR('SampleOrg.admin')"
```

Then, edit the `Application: ACLs` section inside `configtx.yaml` to change `peer/Propose` from this:

`peer/Propose: /Channel/Application/Writers`

To this:

`peer/Propose: /Channel/Application/MyPolicy`

Once these fields have been changed in `configtx.yaml`, the `configtxgen` tool will use the policies and ACLs defined when creating a channel creation transaction. When appropriately signed and submitted by one of the admins of the consortium members, a new channel with the defined ACLs and policies is created.

Once `MyPolicy` has been bootstrapped into the channel configuration, it can also be referenced to override other ACL defaults. For example:

```
SampleSingleMSPChannel:
    Consortium: SampleConsortium
    Application:
        <<: *ApplicationDefaults
        ACLs:
            <<: *ACLsDefault
            event/Block: /Channel/Application/MyPolicy
```

This would restrict the ability to subscribe to block events to `SampleOrg.admin`.

If channels have already been created that want to use this ACL, they'll have to update their channel configurations one at a time using the following flow:

Updating ACL defaults in the channel config

If channels have already been created that want to use `MyPolicy` to restrict access to `peer/Propose` — or if they want to create ACLs they don't want other channels to know about — they'll have to update their channel configurations one at a time through config update transactions.

Note: Channel configuration transactions are an involved process we won't delve into here. If you want to read more about them check out our document on channel configuration updates and our "Adding an Org to a Channel" tutorial.

After pulling, translating, and stripping the configuration block of its metadata, you would edit the configuration by adding `MyPolicy` under `Application: policies`, where the `Admins`, `Writers`, and `Readers` policies already live.

```
"MyPolicy": {
  "mod_policy": "Admins",
  "policy": {
```

(continues on next page)

(continued from previous page)

```
    "type": 1,
    "value": {
      "identities": [
        {
          "principal": {
            "msp_identifier": "SampleOrg",
            "role": "ADMIN"
          },
          "principal_classification": "ROLE"
        }
      ],
      "rule": {
        "n_out_of": {
          "n": 1,
          "rules": [
            {
              "signed_by": 0
            }
          ]
        }
      },
      "version": 0
    }
  },
  "version": "0"
},
```

Note in particular the `msp_identifer` and `role` here.

Then, in the ACLs section of the config, change the `peer/Propose` ACL from this:

```
"peer/Propose": {
  "policy_ref": "/Channel/Application/Writers"
```

To this:

```
"peer/Propose": {
  "policy_ref": "/Channel/Application/MyPolicy"
```

Note: If you do not have ACLs defined in your channel configuration, you will have to add the entire ACL structure.

Once the configuration has been updated, it will need to be submitted by the usual channel update process.

Satisfying an ACL that requires access to multiple resources

If a member makes a request that calls multiple system chaincodes, all of the ACLs for those system chaincodes must be satisfied.

For example, `peer/Propose` refers to any proposal request on a channel. If the particular proposal requires access to two system chaincodes that requires an identity satisfying `Writers` and one system chaincode that requires an identity satisfying `MyPolicy`, then the member submitting the proposal must have an identity that evaluates to "true" for both `Writers` and `MyPolicy`.

In the default configuration, `Writers` is a signature policy whose `rule` is `SampleOrg.member`. In other words, "any member of my organization". `MyPolicy`, listed above, has a rule of `SampleOrg.admin`, or "any admin of my organization". To satisfy these ACLs, the member would have to be both an administrator and a member of `SampleOrg`. By default, all administrators are members (though not all administrators are members), but it is

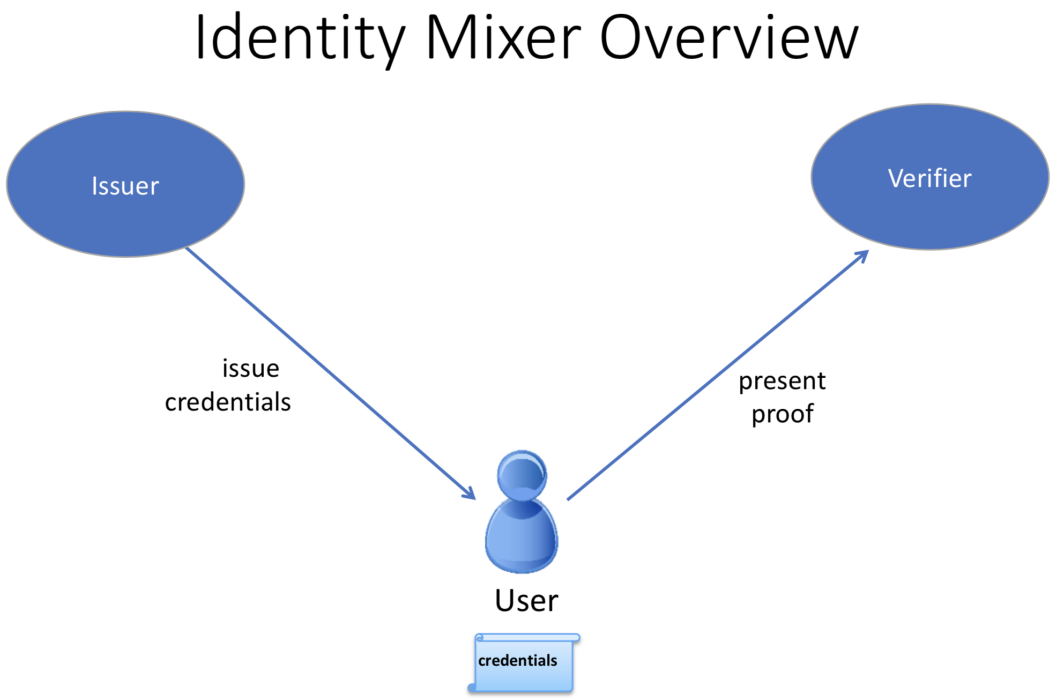
possible to overwrite these policies to whatever you want them to be. As a result, it's important to keep track of these policies to ensure that the ACLs for peer proposals are not impossible to satisfy (unless that is the intention).

9.8 MSP Implementation with Identity Mixer

9.8.1 What is Idemix?

Idemix is a cryptographic protocol suite, which provides strong authentication as well as privacy-preserving features such as **anonymity**, the ability to transact without revealing the identity of the transactor, and **unlinkability**, the ability of a single identity to send multiple transactions without revealing that the transactions were sent by the same identity.

There are three actors involved in an Idemix flow: **user**, **issuer**, and **verifier**.



- An issuer certifies a set of user's attributes are issued in the form of a digital certificate, hereafter called "credential".
- The user later generates a "zero-knowledge proof" of possession of the credential and also selectively discloses only the attributes the user chooses to reveal. The proof, because it is zero-knowledge, reveals no additional information to the verifier, issuer, or anyone else.

As an example, suppose "Alice" needs to prove to Bob (a store clerk) that she has a driver's license issued to her by the DMV.

In this scenario, Alice is the user, the DMV is the issuer, and Bob is the verifier. In order to prove to Bob that Alice has a driver's license, she could show it to him. However, Bob would then be able to see Alice's name, address, exact age, etc. — much more information than Bob needs to know.

Instead, Alice can use Idemix to generate a "zero-knowledge proof" for Bob, which only reveals that she has a valid driver's license and nothing else.

So from the proof:

- Bob does not learn any additional information about Alice other than the fact that she has a valid license (anonymity).
- If Alice visits the store multiple times and generates a proof each time for Bob, Bob would not be able to tell from the proof that it was the same person (unlinkability).

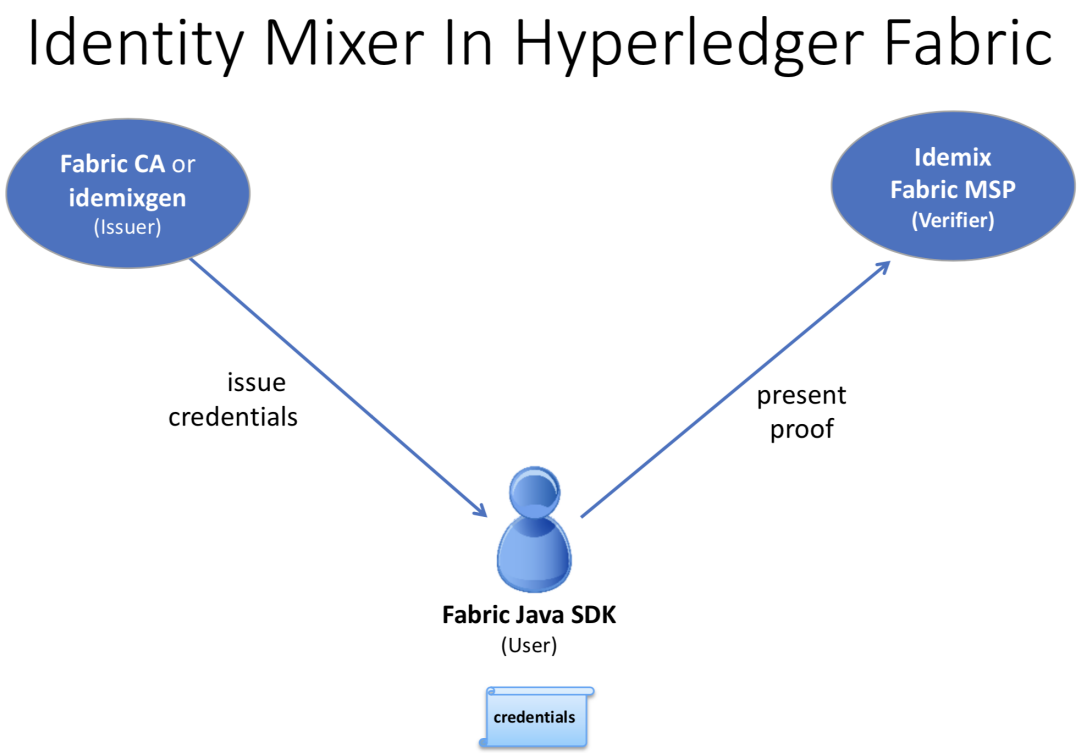
Idemix authentication technology provides the trust model and security guarantees that are similar to what is ensured by standard X.509 certificates but with underlying cryptographic algorithms that efficiently provide advanced privacy features including the ones described above. We'll compare Idemix and X.509 technologies in detail in the technical section below.

9.8.2 How to use Idemix

To understand how to use Idemix with Hyperledger Fabric, we need to see which Fabric components correspond to the user, issuer, and verifier in Idemix.

- The Fabric Java SDK is the API for the **user**. In the future, other Fabric SDKs will also support Idemix.
- Fabric provides two possible Idemix **issuers**:
  1. Fabric CA for production environments or development, and
  2. the *idemixgen* tool for development environments.
- The **verifier** is an Idemix MSP in Fabric.

In order to use Idemix in Hyperledger Fabric, the following three basic steps are required:



Compare the roles in this image to the ones above.

1. Consider the issuer.

Fabric CA (version 1.3 or later) has been enhanced to automatically function as an Idemix issuer. When `fabric-ca-server` is started (or initialized via the `fabric-ca-server init` command), the following two files are automatically created in the home directory of the `fabric-ca-server`: `IssuerPublicKey` and `IssuerRevocationPublicKey`. These files are required in step 2.

For a development environment and if you are not using Fabric CA, you may use `idemixgen` to create these files.

2. Consider the verifier.

   You need to create an Idemix MSP using the `IssuerPublicKey` and `IssuerRevocationPublicKey` from step 1.

   For example, consider the following excerpt from configtx.yaml in the Hyperledger Java SDK sample:

```
- &Org1Idemix
    # defaultorg defines the organization which is used in the sampleconfig
    # of the fabric.git development environment
    name: idemixMSP1

    # id to load the msp definition as
    id: idemixMSPID1

    msptype: idemix
    mspdir: crypto-config/peerOrganizations/org3.example.com
```

   The `msptype` is set to `idemix` and the contents of the `mspdir` directory (`crypto-config/peerOrganizations/org3.example.com/msp` in this example) contains the `IssuerPublicKey` and `IssuerRevocationPublicKey` files.

   Note that in this example, `Org1Idemix` represents the Idemix MSP for `Org1` (not shown), which would also have an X509 MSP.

3. Consider the user. Recall that the Java SDK is the API for the user.

   There is only a single additional API call required in order to use Idemix with the Java SDK: the `idemixEnroll` method of the `org.hyperledger.fabric_ca.sdk.HFCAClient` class. For example, assume `hfcaClient` is your HFCAClient object and `x509Enrollment` is your `org.hyperledger.fabric.sdk.Enrollment` associated with your X509 certificate.

   The following call will return an `org.hyperledger.fabric.sdk.Enrollment` object associated with your Idemix credential.

```
IdemixEnrollment idemixEnrollment = hfcaClient.idemixEnroll(x509enrollment,
↪"idemixMSPID1");
```

   Note also that `IdemixEnrollment` implements the `org.hyperledger.fabric.sdk.Enrollment` interface and can, therefore, be used in the same way that one uses the X509 enrollment object, except, of course, that this automatically provides the privacy enhancing features of Idemix.

9.8.3 Idemix and chaincode

From a verifier perspective, there is one more actor to consider: chaincode. What can chaincode learn about the transactor when an Idemix credential is used?

The cid (Client Identity) library (for Go only) has been extended to support the `GetAttributeValue` function when an Idemix credential is used. However, as mentioned in the "Current limitations" section below, there are only two attributes which are disclosed in the Idemix case: `ou` and `role`.

If Fabric CA is the credential issuer:

- the value of the *ou* attribute is the identity's **affiliation** (e.g. "org1.department1");
- the value of the `role` attribute will be either 'member' or 'admin'. A value of 'admin' means that the identity is an MSP administrator. By default, identities created by Fabric CA will return the 'member' role. In order to create an 'admin' identity, register the identity with the `role` attribute and a value of `2`.

For an example of setting an affiliation in the Java SDK see this sample.

For an example of using the CID library in go chaincode to retrieve attributes, see this go chaincode.

Idemix organizations cannot be used to endorse a chaincode or approve a chaincode definition. This needs to be taken into account when you set the LifecycleEndorsement and Endorsement policies on your channels. For more information, see the limitations section below.

9.8.4 Current limitations

The current version of Idemix does have a few limitations.

- **Idemix organizations and endorsement policies**

  Idemix organizations cannot be used to endorse a chaincode transaction or approve a chaincode definition. By default, the `Channel/Application/LifecycleEndorsement` and `Channel/Application/Endorsement` policies will require signatures from a majority of organizations active on the channel. This implies that a channel that contains a large number of Idemix organizations may not be able to reach the majority needed to fulfill the default policy. For example, if a channel has two MSP Organizations and two Idemix organizations, the channel policy will require that three out of four organizations approve a chaincode definition to commit that definition to the channel. Because Idemix organizations cannot approve a chaincode definition, the policy will only be able to validate two out of four signatures.

  If your channel contains a sufficient number of Idemix organizations to affect the endorsement policy, you can use a signature policy to explicitly specify the required MSP organizations.

- **Fixed set of attributes**

  It not yet possible to issue or use an Idemix credential with custom attributes. Custom attributes will be supported in a future release.

  The following four attributes are currently supported:

  1. Organizational Unit attribute ("ou"):
  - Usage: same as X.509
  - Type: String
  - Revealed: always
  2. Role attribute ("role"):
  - Usage: same as X.509
  - Type: integer
  - Revealed: always
  3. Enrollment ID attribute
  - Usage: uniquely identify a user — same in all enrollment credentials that belong to the same user (will be used for auditing in the future releases)
  - Type: BIG
  - Revealed: never in the signature, only when generating an authentication token for Fabric CA
  4. Revocation Handle attribute

  - Usage: uniquely identify a credential (will be used for revocation in future releases)
  - Type: integer
  - Revealed: never

- **Revocation is not yet supported**

  Although much of the revocation framework is in place as can be seen by the presence of a revocation handle attribute mentioned above, revocation of an Idemix credential is not yet supported.

- **Peers do not use Idemix for endorsement**

  Currently, Idemix MSP is used by the peers only for signature verification. Signing with Idemix is only done via Client SDK. More roles (including a 'peer' role) will be supported by Idemix MSP.

9.8.5 Technical summary

Comparing Idemix credentials to X.509 certificates

The certificate/credential concept and the issuance process are very similar in Idemix and X.509 certs: a set of attributes is digitally signed with a signature that cannot be forged and there is a secret key to which a credential is cryptographically bound.

The main difference between a standard X.509 certificate and an Identity Mixer credential is the signature scheme that is used to certify the attributes. The signatures underlying the Identity Mixer system allow for efficient proofs of the possession of a signature and the corresponding attributes without revealing the signature and (selected) attribute values themselves. We use zero-knowledge proofs to ensure that such "knowledge" or "information" is not revealed while ensuring that the signature over some attributes is valid and the user is in possession of the corresponding credential secret key.

Such proofs, like X.509 certificates, can be verified with the public key of the authority that originally signed the credential and cannot be successfully forged. Only the user who knows the credential secret key can generate the proofs about the credential and its attributes.

With regard to unlinkability, when an X.509 certificate is presented, all attributes have to be revealed to verify the certificate signature. This implies that all certificate usages for signing transactions are linkable.

To avoid such linkability, fresh X.509 certificates need to be used every time, which results in complex key management and communication and storage overhead. Furthermore, there are cases where it is important that not even the CA issuing the certificates is able to link all the transactions to the user.

Idemix helps to avoid linkability with respect to both the CA and verifiers, since even the CA is not able to link proofs to the original credential. Neither the issuer nor a verifier can tell whether two proofs were derived from the same credential (or from two different ones).

More details on the concepts and features of the Identity Mixer technology are described in the paper Concepts and Languages for Privacy-Preserving Attribute-Based Authentication.

Topology Information

Given the above limitations, it is recommended to have only one Idemix-based MSP per channel or, at the extreme, per network. Indeed, for example, having multiple Idemix-based MSPs per channel would allow a party, reading the ledger of that channel, to tell apart transactions signed by parties belonging to different Idemix-based MSPs. This is because, each transaction leak the MSP-ID of the signer. In other words, Idemix currently provides only anonymity of clients among the same organization (MSP).

In the future, Idemix could be extended to support anonymous hierarchies of Idemix-based Certification Authorities whose certified credentials can be verified by using a unique public-key, therefore achieving anonymity across organizations (MSPs). This would allow multiple Idemix-based MSPs to coexist in the same channel.

In principal, a channel can be configured to have a single Idemix-based MSP and multiple X.509-based MSPs. Of course, the interaction between these MSP can potential leak information. An assessment of the leaked information need to be done case by case.wq

Underlying cryptographic protocols

Idemix technology is built from a blind signature scheme that supports multiple messages and efficient zero-knowledge proofs of signature possession. All of the cryptographic building blocks for Idemix were published at the top conferences and journals and verified by the scientific community.

This particular Idemix implementation for Fabric uses a pairing-based signature scheme that was briefly proposed by Camenisch and Lysyanskaya and described in detail by Au et al.. The ability to prove knowledge of a signature in a zero-knowledge proof Camenisch et al. was used.

9.9 Identity Mixer MSP configuration generator (idemixgen)

This document describes the usage for the `idemixgen` utility, which can be used to create configuration files for the identity mixer based MSP. Two commands are available, one for creating a fresh CA key pair, and one for creating an MSP config using a previously generated CA key.

9.9.1 Directory Structure

The `idemixgen` tool will create directories with the following structure:

```
- /ca/
    IssuerSecretKey
    IssuerPublicKey
    RevocationKey
- /msp/
    IssuerPublicKey
    RevocationPublicKey
- /user/
    SignerConfig
```

The `ca` directory contains the issuer secret key (including the revocation key) and should only be present for a CA. The `msp` directory contains the information required to set up an MSP verifying idemix signatures. The `user` directory specifies a default signer.

9.9.2 CA Key Generation

CA (issuer) keys suitable for identity mixer can be created using command `idemixgen ca-keygen`. This will create directories `ca` and `msp` in the working directory.

9.9.3 Adding a Default Signer

After generating the `ca` and `msp` directories with `idemixgen ca-keygen`, a default signer specified in the `user` directory can be added to the config with `idemixgen signerconfig`.

```
$ idemixgen signerconfig -h
usage: idemixgen signerconfig [<flags>]

Generate a default signer for this Idemix MSP

Flags:
    -h, --help               Show context-sensitive help (also try --help-long and --
↪help-man).
    -u, --org-unit=ORG-UNIT  The Organizational Unit of the default signer
    -a, --admin              Make the default signer admin
    -e, --enrollment-id=ENROLLMENT-ID
                             The enrollment id of the default signer
    -r, --revocation-handle=REVOCATION-HANDLE
                             The handle used to revoke this signer
```

For example, we can create a default signer that is a member of organizational unit "OrgUnit1", with enrollment identity "johndoe", revocation handle "1234", and that is an admin, with the following command:

```
idemixgen signerconfig -u OrgUnit1 --admin -e "johndoe" -r 1234
```

9.10 The Operations Service

The peer and the orderer host an HTTP server that offers a RESTful "operations" API. This API is unrelated to the Fabric network services and is intended to be used by operators, not administrators or "users" of the network.

The API exposes the following capabilities:

- Log level management
- Health checks
- Prometheus target for operational metrics (when configured)

9.10.1 Configuring the Operations Service

The operations service requires two basic pieces of configuration:

- The **address** and **port** to listen on.
- The **TLS certificates** and **keys** to use for authentication and encryption. Note, **these certificates should be generated by a separate and dedicated CA**. Do not use a CA that has generated certificates for any organizations in any channels.

Peer

For each peer, the operations server can be configured in the `operations` section of `core.yaml`:

```
operations:
  # host and port for the operations server
  listenAddress: 127.0.0.1:9443

  # TLS configuration for the operations endpoint
  tls:
    # TLS enabled
```

(continues on next page)

(continued from previous page)

```
    enabled: true

    # path to PEM encoded server certificate for the operations server
    cert:
      file: tls/server.crt

    # path to PEM encoded server key for the operations server
    key:
      file: tls/server.key

    # most operations service endpoints require client authentication when TLS
    # is enabled. clientAuthRequired requires client certificate authentication
    # at the TLS layer to access all resources.
    clientAuthRequired: false

    # paths to PEM encoded ca certificates to trust for client authentication
    clientRootCAs:
      files: []
```

The `listenAddress` key defines the host and port that the operation server will listen on. If the server should listen on all addresses, the host portion can be omitted.

The `tls` section is used to indicate whether or not TLS is enabled for the operations service, the location of the service's certificate and private key, and the locations of certificate authority root certificates that should be trusted for client authentication. When `enabled` is true, most of the operations service endpoints require client authentication, therefore `clientRootCAs.files` must be set. When `clientAuthRequired` is `true`, the TLS layer will require clients to provide a certificate for authentication on every request. See Operations Security section below for more details.

Orderer

For each orderer, the operations server can be configured in the *Operations* section of `orderer.yaml`:

```
Operations:
  # host and port for the operations server
  ListenAddress: 127.0.0.1:8443

  # TLS configuration for the operations endpoint
  TLS:
    # TLS enabled
    Enabled: true

    # PrivateKey: PEM-encoded tls key for the operations endpoint
    PrivateKey: tls/server.key

    # Certificate governs the file location of the server TLS certificate.
    Certificate: tls/server.crt

    # Paths to PEM encoded ca certificates to trust for client authentication
    ClientRootCAs: []

    # Most operations service endpoints require client authentication when TLS
    # is enabled. ClientAuthRequired requires client certificate authentication
    # at the TLS layer to access all resources.
    ClientAuthRequired: false
```

The `ListenAddress` key defines the host and port that the operations server will listen on. If the server should listen on all addresses, the host portion can be omitted.

The `TLS` section is used to indicate whether or not TLS is enabled for the operations service, the location of the service's certificate and private key, and the locations of certificate authority root certificates that should be trusted for client authentication. When `Enabled` is true, most of the operations service endpoints require client authentication, therefore `RootCAs` must be set. When `ClientAuthRequired` is `true`, the TLS layer will require clients to provide a certificate for authentication on every request. See Operations Security section below for more details.

Operations Security

As the operations service is focused on operations and intentionally unrelated to the Fabric network, it does not use the Membership Services Provider for access control. Instead, the operations service relies entirely on mutual TLS with client certificate authentication.

When TLS is disabled, authorization is bypassed and any client that can connect to the operations endpoint will be able to use the API.

When TLS is enabled, a valid client certificate must be provided in order to access all resources unless explicitly noted otherwise below.

When clientAuthRequired is also enabled, the TLS layer will require a valid client certificate regardless of the resource being accessed.

Log Level Management

The operations service provides a `/logspec` resource that operators can use to manage the active logging spec for a peer or orderer. The resource is a conventional REST resource and supports `GET` and `PUT` requests.

When a `GET /logspec` request is received by the operations service, it will respond with a JSON payload that contains the current logging specification:

```
{"spec":"info"}
```

When a `PUT /logspec` request is received by the operations service, it will read the body as a JSON payload. The payload must consist of a single attribute named `spec`.

```
{"spec":"chaincode=debug:info"}
```

If the spec is activated successfully, the service will respond with a `204 "No Content"` response. If an error occurs, the service will respond with a `400 "Bad Request"` and an error payload:

```
{"error":"error message"}
```

9.10.2 Health Checks

The operations service provides a `/healthz` resource that operators can use to help determine the liveness and health of peers and orderers. The resource is a conventional REST resource that supports GET requests. The implementation is intended to be compatible with the liveness probe model used by Kubernetes but can be used in other contexts.

When a `GET /healthz` request is received, the operations service will call all registered health checkers for the process. When all of the health checkers return successfully, the operations service will respond with a `200 "OK"` and a JSON body:

```
{
  "status": "OK",
  "time": "2009-11-10T23:00:00Z"
}
```

If one or more of the health checkers returns an error, the operations service will respond with a `503 "Service Unavailable"` and a JSON body that includes information about which health checker failed:

```
{
  "status": "Service Unavailable",
  "time": "2009-11-10T23:00:00Z",
  "failed_checks": [
    {
      "component": "docker",
      "reason": "failed to connect to Docker daemon: invalid endpoint"
    }
  ]
}
```

In the current version, the only health check that is registered is for Docker. Future versions will be enhanced to add additional health checks.

When TLS is enabled, a valid client certificate is not required to use this service unless `clientAuthRequired` is set to `true`.

9.10.3 Metrics

Some components of the Fabric peer and orderer expose metrics that can help provide insight into the behavior of the system. Operators and administrators can use this information to better understand how the system is performing over time.

Configuring Metrics

Fabric provides two ways to expose metrics: a **pull** model based on Prometheus and a **push** model based on StatsD.

Prometheus

A typical Prometheus deployment scrapes metrics by requesting them from an HTTP endpoint exposed by instrumented targets. As Prometheus is responsible for requesting the metrics, it is considered a pull system.

When configured, a Fabric peer or orderer will present a `/metrics` resource on the operations service.

Peer

A peer can be configured to expose a `/metrics` endpoint for Prometheus to scrape by setting the metrics provider to `prometheus` in the `metrics` section of `core.yaml`.

```
metrics:
  provider: prometheus
```

Orderer

An orderer can be configured to expose a `/metrics` endpoint for Prometheus to scrape by setting the metrics provider to `prometheus` in the `Metrics` section of `orderer.yaml`.

```
Metrics:
  Provider: prometheus
```

StatsD

StatsD is a simple statistics aggregation daemon. Metrics are sent to a `statsd` daemon where they are collected, aggregated, and pushed to a backend for visualization and alerting. As this model requires instrumented processes to send metrics data to StatsD, this is considered a push system.

Peer

A peer can be configured to send metrics to StatsD by setting the metrics provider to `statsd` in the `metrics` section of `core.yaml`. The `statsd` subsection must also be configured with the address of the StatsD daemon, the network type to use (`tcp` or `udp`), and how often to send the metrics. An optional `prefix` may be specified to help differentiate the source of the metrics — for example, differentiating metrics coming from separate peers — that would be prepended to all generated metrics.

```
metrics:
  provider: statsd
  statsd:
    network: udp
    address: 127.0.0.1:8125
    writeInterval: 10s
    prefix: peer-0
```

Orderer

An orderer can be configured to send metrics to StatsD by setting the metrics provider to `statsd` in the `Metrics` section of `orderer.yaml`. The `Statsd` subsection must also be configured with the address of the StatsD daemon, the network type to use (`tcp` or `udp`), and how often to send the metrics. An optional `prefix` may be specified to help differentiate the source of the metrics.

```
Metrics:
    Provider: statsd
    Statsd:
      Network: udp
      Address: 127.0.0.1:8125
      WriteInterval: 30s
      Prefix: org-orderer
```

For a look at the different metrics that are generated, check out *Metrics Reference*.

9.11 Metrics Reference

9.11.1 Orderer Metrics

Prometheus

The following orderer metrics are exported for consumption by Prometheus.

Name	Type	Description
blockcutter_block_fill_duration	histogram	The time from first transaction enqueing to the block being cut in seco
broadcast_enqueue_duration	histogram	The time to enqueue a transaction in seconds.
broadcast_processed_count	counter	The number of transactions processed.
broadcast_validate_duration	histogram	The time to validate a transaction in seconds.
cluster_comm_egress_queue_capacity	gauge	Capacity of the egress queue.
cluster_comm_egress_queue_length	gauge	Length of the egress queue.
cluster_comm_egress_queue_workers	gauge	Count of egress queue workers.
cluster_comm_egress_stream_count	gauge	Count of streams to other nodes.
cluster_comm_egress_tls_connection_count	gauge	Count of TLS connections to other nodes.
cluster_comm_ingress_stream_count	gauge	Count of streams from other nodes.
cluster_comm_msg_dropped_count	counter	Count of messages dropped.
cluster_comm_msg_send_time	histogram	The time it takes to send a message in seconds.
consensus_etcdraft_active_nodes	gauge	Number of active nodes in this channel.
consensus_etcdraft_cluster_size	gauge	Number of nodes in this channel.
consensus_etcdraft_committed_block_number	gauge	The block number of the latest block committed.
consensus_etcdraft_config_proposals_received	counter	The total number of proposals received for config type transactions.
consensus_etcdraft_data_persist_duration	histogram	The time taken for etcd/raft data to be persisted in storage (in seconds)
consensus_etcdraft_is_leader	gauge	The leadership status of the current node: 1 if it is the leader else 0.
consensus_etcdraft_leader_changes	counter	The number of leader changes since process start.
consensus_etcdraft_normal_proposals_received	counter	The total number of proposals received for normal type transactions.
consensus_etcdraft_proposal_failures	counter	The number of proposal failures.
consensus_etcdraft_snapshot_block_number	gauge	The block number of the latest snapshot.
consensus_kafka_batch_size	gauge	The mean batch size in bytes sent to topics.
consensus_kafka_compression_ratio	gauge	The mean compression ratio (as percentage) for topics.
consensus_kafka_incoming_byte_rate	gauge	Bytes/second read off brokers.
consensus_kafka_last_offset_persisted	gauge	The offset specified in the block metadata of the most recently commi
consensus_kafka_outgoing_byte_rate	gauge	Bytes/second written to brokers.
consensus_kafka_record_send_rate	gauge	The number of records per second sent to topics.
consensus_kafka_records_per_request	gauge	The mean number of records sent per request to topics.

Table 1 – continued from previous page

Name	Type	Description
consensus_kafka_request_latency	gauge	The mean request latency in ms to brokers.
consensus_kafka_request_rate	gauge	Requests/second sent to brokers.
consensus_kafka_request_size	gauge	The mean request size in bytes to brokers.
consensus_kafka_response_rate	gauge	Requests/second sent to brokers.
consensus_kafka_response_size	gauge	The mean response size in bytes from brokers.
couchdb_processing_time	histogram	Time taken in seconds for the function to complete request to CouchDB
deliver_blocks_sent	counter	The number of blocks sent by the deliver service.
deliver_requests_completed	counter	The number of deliver requests that have been completed.
deliver_requests_received	counter	The number of deliver requests that have been received.
deliver_streams_closed	counter	The number of GRPC streams that have been closed for the deliver se
deliver_streams_opened	counter	The number of GRPC streams that have been opened for the deliver se
fabric_version	gauge	The active version of Fabric.
grpc_comm_conn_closed	counter	gRPC connections closed. Open minus closed is the active number of
grpc_comm_conn_opened	counter	gRPC connections opened. Open minus closed is the active number of
grpc_server_stream_messages_received	counter	The number of stream messages received.
grpc_server_stream_messages_sent	counter	The number of stream messages sent.
grpc_server_stream_request_duration	histogram	The time to complete a stream request.
grpc_server_stream_requests_completed	counter	The number of stream requests completed.
grpc_server_stream_requests_received	counter	The number of stream requests received.
grpc_server_unary_request_duration	histogram	The time to complete a unary request.
grpc_server_unary_requests_completed	counter	The number of unary requests completed.
grpc_server_unary_requests_received	counter	The number of unary requests received.
ledger_blockchain_height	gauge	Height of the chain in blocks.
ledger_blockstorage_commit_time	histogram	Time taken in seconds for committing the block to storage.

Table 1 – continued from previous page

Name	Type	Description
logging_entries_checked	counter	Number of log entries checked against the active logging level
logging_entries_written	counter	Number of log entries that are written

StatsD

The following orderer metrics are emitted for consumption by StatsD. The `%{variable_name}` nomenclature represents segments that vary based on context.

For example, `%{channel}` will be replaced with the name of the channel associated with the metric.

Bucket	Type	Description
blockcutter.block_fill_duration.%{channel}	histogram	The time from first transaction enquei
broadcast.enqueue_duration.%{channel}.%{type}.%{status}	histogram	The time to enqueue a transaction in s
broadcast.processed_count.%{channel}.%{type}.%{status}	counter	The number of transactions processed
broadcast.validate_duration.%{channel}.%{type}.%{status}	histogram	The time to validate a transaction in se
cluster.comm.egress_queue_capacity.%{host}.%{msg_type}.%{channel}	gauge	Capacity of the egress queue.
cluster.comm.egress_queue_length.%{host}.%{msg_type}.%{channel}	gauge	Length of the egress queue.
cluster.comm.egress_queue_workers.%{channel}	gauge	Count of egress queue workers.
cluster.comm.egress_stream_count.%{channel}	gauge	Count of streams to other nodes.
cluster.comm.egress_tls_connection_count	gauge	Count of TLS connections to other no
cluster.comm.ingress_stream_count	gauge	Count of streams from other nodes.
cluster.comm.msg_dropped_count.%{host}.%{channel}	counter	Count of messages dropped.
cluster.comm.msg_send_time.%{host}.%{channel}	histogram	The time it takes to send a message in
consensus.etcdraft.active_nodes.%{channel}	gauge	Number of active nodes in this channe
consensus.etcdraft.cluster_size.%{channel}	gauge	Number of nodes in this channel.
consensus.etcdraft.committed_block_number.%{channel}	gauge	The block number of the latest block c
consensus.etcdraft.config_proposals_received.%{channel}	counter	The total number of proposals receive
consensus.etcdraft.data_persist_duration.%{channel}	histogram	The time taken for etcd/raft data to be
consensus.etcdraft.is_leader.%{channel}	gauge	The leadership status of the current no
consensus.etcdraft.leader_changes.%{channel}	counter	The number of leader changes since p
consensus.etcdraft.normal_proposals_received.%{channel}	counter	The total number of proposals receive
consensus.etcdraft.proposal_failures.%{channel}	counter	The number of proposal failures.
consensus.etcdraft.snapshot_block_number.%{channel}	gauge	The block number of the latest snapsh
consensus.kafka.batch_size.%{topic}	gauge	The mean batch size in bytes sent to to
consensus.kafka.compression_ratio.%{topic}	gauge	The mean compression ratio (as perce
consensus.kafka.incoming_byte_rate.%{broker_id}	gauge	Bytes/second read off brokers.
consensus.kafka.last_offset_persisted.%{channel}	gauge	The offset specified in the block metad
consensus.kafka.outgoing_byte_rate.%{broker_id}	gauge	Bytes/second written to brokers.
consensus.kafka.record_send_rate.%{topic}	gauge	The number of records per second sen
consensus.kafka.records_per_request.%{topic}	gauge	The mean number of records sent per
consensus.kafka.request_latency.%{broker_id}	gauge	The mean request latency in ms to bro
consensus.kafka.request_rate.%{broker_id}	gauge	Requests/second sent to brokers.
consensus.kafka.request_size.%{broker_id}	gauge	The mean request size in bytes to brok
consensus.kafka.response_rate.%{broker_id}	gauge	Requests/second sent to brokers.
consensus.kafka.response_size.%{broker_id}	gauge	The mean response size in bytes from
couchdb.processing_time.%{database}.%{function_name}.%{result}	histogram	Time taken in seconds for the function
deliver.blocks_sent.%{channel}.%{filtered}.%{data_type}	counter	The number of blocks sent by the deli
deliver.requests_completed.%{channel}.%{filtered}.%{data_type}.%{success}	counter	The number of deliver requests that ha
deliver.requests_received.%{channel}.%{filtered}.%{data_type}	counter	The number of deliver requests that ha

Table 2 – continued from previous page

Bucket	Type	Description
deliver.streams_closed	counter	The number of GRPC streams that hav
deliver.streams_opened	counter	The number of GRPC streams that hav
fabric_version.%{version}	gauge	The active version of Fabric.
grpc.comm.conn_closed	counter	gRPC connections closed. Open minu
grpc.comm.conn_opened	counter	gRPC connections opened. Open minu
grpc.server.stream_messages_received.%{service}.%{method}	counter	The number of stream messages recei
grpc.server.stream_messages_sent.%{service}.%{method}	counter	The number of stream messages sent.
grpc.server.stream_request_duration.%{service}.%{method}.%{code}	histogram	The time to complete a stream request
grpc.server.stream_requests_completed.%{service}.%{method}.%{code}	counter	The number of stream requests compl
grpc.server.stream_requests_received.%{service}.%{method}	counter	The number of stream requests receive
grpc.server.unary_request_duration.%{service}.%{method}.%{code}	histogram	The time to complete a unary request.
grpc.server.unary_requests_completed.%{service}.%{method}.%{code}	counter	The number of unary requests comple
grpc.server.unary_requests_received.%{service}.%{method}	counter	The number of unary requests receive
ledger.blockchain_height.%{channel}	gauge	Height of the chain in blocks.
ledger.blockstorage_commit_time.%{channel}	histogram	Time taken in seconds for committing
logging.entries_checked.%{level}	counter	Number of log entries checked agains
logging.entries_written.%{level}	counter	Number of log entries that are written

9.11.2 Peer Metrics

Prometheus

The following peer metrics are exported for consumption by Prometheus.

Name	Type	Description
chaincode_execute_timeouts	counter	The number of chaincode executions (Init or Invoke) that have timed
chaincode_launch_duration	histogram	The time to launch a chaincode.
chaincode_launch_failures	counter	The number of chaincode launches that have failed.
chaincode_launch_timeouts	counter	The number of chaincode launches that have timed out.
chaincode_shim_request_duration	histogram	The time to complete chaincode shim requests.
chaincode_shim_requests_completed	counter	The number of chaincode shim requests completed.
chaincode_shim_requests_received	counter	The number of chaincode shim requests received.
couchdb_processing_time	histogram	Time taken in seconds for the function to complete request to CouchDB
deliver_blocks_sent	counter	The number of blocks sent by the deliver service.

Table 3 – continued from previous page

Name	Type	Description
deliver_requests_completed	counter	The number of deliver requests that have been completed.
deliver_requests_received	counter	The number of deliver requests that have been received.
deliver_streams_closed	counter	The number of GRPC streams that have been closed for the deliver se
deliver_streams_opened	counter	The number of GRPC streams that have been opened for the deliver s
dockercontroller_chaincode_container_build_duration	histogram	The time to build a chaincode image in seconds.
endorser_chaincode_instantiation_failures	counter	The number of chaincode instantiations or upgrade that have failed.
endorser_duplicate_transaction_failures	counter	The number of failed proposals due to duplicate transaction ID.
endorser_endorsement_failures	counter	The number of failed endorsements.
endorser_proposal_acl_failures	counter	The number of proposals that failed ACL checks.
endorser_proposal_duration	histogram	The time to complete a proposal.
endorser_proposal_simulation_failures	counter	The number of failed proposal simulations
endorser_proposal_validation_failures	counter	The number of proposals that have failed initial validation.
endorser_proposals_received	counter	The number of proposals received.
endorser_successful_proposals	counter	The number of successful proposals.
fabric_version	gauge	The active version of Fabric.
gossip_comm_messages_received	counter	Number of messages received
gossip_comm_messages_sent	counter	Number of messages sent
gossip_comm_overflow_count	counter	Number of outgoing queue buffer overflows
gossip_leader_election_leader	gauge	Peer is leader (1) or follower (0)
gossip_membership_total_peers_known	gauge	Total known peers
gossip_payload_buffer_size	gauge	Size of the payload buffer
gossip_privdata_commit_block_duration	histogram	Time it takes to commit private data and the corresponding block (in s
gossip_privdata_fetch_duration	histogram	Time it takes to fetch missing private data from peers (in seconds)
gossip_privdata_list_missing_duration	histogram	Time it takes to list the missing private data (in seconds)
gossip_privdata_pull_duration	histogram	Time it takes to pull a missing private data element (in seconds)
gossip_privdata_purge_duration	histogram	Time it takes to purge private data (in seconds)
gossip_privdata_reconciliation_duration	histogram	Time it takes for reconciliation to complete (in seconds)
gossip_privdata_retrieve_duration	histogram	Time it takes to retrieve missing private data elements from the ledger
gossip_privdata_send_duration	histogram	Time it takes to send a missing private data element (in seconds)

Table 3 – continued from previous page

Name	Type	Description
gossip_privdata_validation_duration	histogram	Time it takes to validate a block (in seconds)
gossip_state_commit_duration	histogram	Time it takes to commit a block in seconds
gossip_state_height	gauge	Current ledger height
grpc_comm_conn_closed	counter	gRPC connections closed. Open minus closed is the active number of
grpc_comm_conn_opened	counter	gRPC connections opened. Open minus closed is the active number o
grpc_server_stream_messages_received	counter	The number of stream messages received.
grpc_server_stream_messages_sent	counter	The number of stream messages sent.
grpc_server_stream_request_duration	histogram	The time to complete a stream request.
grpc_server_stream_requests_completed	counter	The number of stream requests completed.
grpc_server_stream_requests_received	counter	The number of stream requests received.
grpc_server_unary_request_duration	histogram	The time to complete a unary request.
grpc_server_unary_requests_completed	counter	The number of unary requests completed.
grpc_server_unary_requests_received	counter	The number of unary requests received.
ledger_block_processing_time	histogram	Time taken in seconds for ledger block processing.
ledger_blockchain_height	gauge	Height of the chain in blocks.
ledger_blockstorage_and_pvtdata_commit_time	histogram	Time taken in seconds for committing the block and private data to st
ledger_blockstorage_commit_time	histogram	Time taken in seconds for committing the block to storage.
ledger_statedb_commit_time	histogram	Time taken in seconds for committing block changes to state db.
ledger_transaction_count	counter	Number of transactions processed.
logging_entries_checked	counter	Number of log entries checked against the active logging level
logging_entries_written	counter	Number of log entries that are written

StatsD

The following peer metrics are emitted for consumption by StatsD. The `%{variable_name}` nomenclature represents segments that vary based on context.

For example, `%{channel}` will be replaced with the name of the channel associated with the metric.

Bucket	Type	Description
chaincode.execute_timeouts.%{chaincode}	counter	The number of chainco
chaincode.launch_duration.%{chaincode}.%{success}	histogram	The time to launch a c
chaincode.launch_failures.%{chaincode}	counter	The number of chainco
chaincode.launch_timeouts.%{chaincode}	counter	The number of chainco
chaincode.shim_request_duration.%{type}.%{channel}.%{chaincode}.%{success}	histogram	The time to complete c
chaincode.shim_requests_completed.%{type}.%{channel}.%{chaincode}.%{success}	counter	The number of chainco
chaincode.shim_requests_received.%{type}.%{channel}.%{chaincode}	counter	The number of chainco
couchdb.processing_time.%{database}.%{function_name}.%{result}	histogram	Time taken in seconds
deliver.blocks_sent.%{channel}.%{filtered}.%{data_type}	counter	The number of blocks
deliver.requests_completed.%{channel}.%{filtered}.%{data_type}.%{success}	counter	The number of deliver
deliver.requests_received.%{channel}.%{filtered}.%{data_type}	counter	The number of deliver
deliver.streams_closed	counter	The number of GRPC
deliver.streams_opened	counter	The number of GRPC
dockercontroller.chaincode_container_build_duration.%{chaincode}.%{success}	histogram	The time to build a cha
endorser.chaincode_instantiation_failures.%{channel}.%{chaincode}	counter	The number of chainco
endorser.duplicate_transaction_failures.%{channel}.%{chaincode}	counter	The number of failed p
endorser.endorsement_failures.%{channel}.%{chaincode}.%{chaincodeerror}	counter	The number of failed e
endorser.proposal_acl_failures.%{channel}.%{chaincode}	counter	The number of propos
endorser.proposal_duration.%{channel}.%{chaincode}.%{success}	histogram	The time to complete a
endorser.proposal_simulation_failures.%{channel}.%{chaincode}	counter	The number of failed p
endorser.proposal_validation_failures	counter	The number of propos
endorser.proposals_received	counter	The number of propos
endorser.successful_proposals	counter	The number of success
fabric_version.%{version}	gauge	The active version of F
gossip.comm.messages_received	counter	Number of messages r
gossip.comm.messages_sent	counter	Number of messages s
gossip.comm.overflow_count	counter	Number of outgoing q
gossip.leader_election.leader.%{channel}	gauge	Peer is leader (1) or fo
gossip.membership.total_peers_known.%{channel}	gauge	Total known peers
gossip.payload_buffer.size.%{channel}	gauge	Size of the payload bu
gossip.privdata.commit_block_duration.%{channel}	histogram	Time it takes to comm
gossip.privdata.fetch_duration.%{channel}	histogram	Time it takes to fetch r
gossip.privdata.list_missing_duration.%{channel}	histogram	Time it takes to list the
gossip.privdata.pull_duration.%{channel}	histogram	Time it takes to pull a
gossip.privdata.purge_duration.%{channel}	histogram	Time it takes to purge
gossip.privdata.reconciliation_duration.%{channel}	histogram	Time it takes for recon
gossip.privdata.retrieve_duration.%{channel}	histogram	Time it takes to retriev
gossip.privdata.send_duration.%{channel}	histogram	Time it takes to send a
gossip.privdata.validation_duration.%{channel}	histogram	Time it takes to validat
gossip.state.commit_duration.%{channel}	histogram	Time it takes to comm
gossip.state.height.%{channel}	gauge	Current ledger height
grpc.comm.conn_closed	counter	gRPC connections clos
grpc.comm.conn_opened	counter	gRPC connections ope
grpc.server.stream_messages_received.%{service}.%{method}	counter	The number of stream
grpc.server.stream_messages_sent.%{service}.%{method}	counter	The number of stream
grpc.server.stream_request_duration.%{service}.%{method}.%{code}	histogram	The time to complete a
grpc.server.stream_requests_completed.%{service}.%{method}.%{code}	counter	The number of stream
grpc.server.stream_requests_received.%{service}.%{method}	counter	The number of stream
grpc.server.unary_request_duration.%{service}.%{method}.%{code}	histogram	The time to complete a
grpc.server.unary_requests_completed.%{service}.%{method}.%{code}	counter	The number of unary r

Table 4 – continued from previous page

Bucket	Type	Description
grpc.server.unary_requests_received.%{service}.%{method}	counter	The number of unary r
ledger.block_processing_time.%{channel}	histogram	Time taken in seconds
ledger.blockchain_height.%{channel}	gauge	Height of the chain in
ledger.blockstorage_and_pvtdata_commit_time.%{channel}	histogram	Time taken in seconds
ledger.blockstorage_commit_time.%{channel}	histogram	Time taken in seconds
ledger.statedb_commit_time.%{channel}	histogram	Time taken in seconds
ledger.transaction_count.%{channel}.%{transaction_type}.%{chaincode}.%{validation_code}	counter	Number of transaction
logging.entries_checked.%{level}	counter	Number of log entries
logging.entries_written.%{level}	counter	Number of log entries

9.12 External Builders and Launchers

Prior to Hyperledger Fabric 2.0, the process used to build and launch chaincode was part of the peer implementation and could not be easily customized. All chaincode installed on the peer would be "built" using language specific logic hard coded in the peer. This build process would generate a Docker container image that would be launched to execute chaincode that connected as a client to the peer.

This approach limited chaincode implementations to a handful of languages, required Docker to be part of the deployment environment, and prevented running chaincode as a long running server process.

Starting with Fabric 2.0, External Builders and Launchers address these limitations by enabling operators to extend the peer with programs that can build, launch, and discover chaincode. To leverage this capability you will need to create your own buildpack and then modify the peer core.yaml to include a new `externalBuilder` configuration element which lets the peer know an external builder is available. The following sections describe the details of this process.

Note that if no configured external builder claims a chaincode package, the peer will attempt to process the package as if it were created with the standard Fabric packaging tools such as the peer CLI or node SDK.

9.12.1 External builder model

Hyperledger Fabric External Builders and Launchers are loosely based on Heroku Buildpacks. A buildpack implementation is simply a collection of programs or scripts that transform application artifacts into something that can run. The buildpack model has been adapted for chaincode packages and extended to support chaincode execution and discovery.

External builder and launcher API

An external builder and launcher consists of four programs or scripts:

- `bin/detect`: Determine whether or not this buildpack should be used to build the chaincode package and launch it.
- `bin/build`: Transform the chaincode package into executable chaincode.
- `bin/release` (optional): Provide metadata to the peer about the chaincode.
- `bin/run` (optional): Run the chaincode.

bin/detect

The `bin/detect` script is responsible for determining whether or not a buildpack should be used to build a chaincode package and launch it. The peer invokes `detect` with two arguments:

```
bin/detect CHAINCODE_SOURCE_DIR CHAINCODE_METADATA_DIR
```

When `detect` is invoked, `CHAINCODE_SOURCE_DIR` contains the chaincode source and `CHAINCODE_METADATA_DIR` contains the `metadata.json` file from the chaincode package installed to the peer. The `CHAINCODE_SOURCE_DIR` and `CHAINCODE_METADATA_DIR` should be treated as read only inputs. If the buildpack should be applied to the chaincode source package, `detect` must return an exit code of `0`; any other exit code will indicate that the buildpack should not be applied.

The following is an example of a simple `detect` script for go chaincode:

```
#!/bin/bash

CHAINCODE_METADATA_DIR="$2"

# use jq to extract the chaincode type from metadata.json and exit with
# success if the chaincode type is golang
if [ "$(jq -r .type "$CHAINCODE_METADATA_DIR/metadata.json" | tr '[:upper:]'
↪'[:lower:]')" = "golang" ]; then
    exit 0
fi

exit 1
```

bin/build

The `bin/build` script is responsible for building, compiling, or transforming the contents of a chaincode package into artifacts that can be used by `release` and `run`. The peer invokes `build` with three arguments:

```
bin/build CHAINCODE_SOURCE_DIR CHAINCODE_METADATA_DIR BUILD_OUTPUT_DIR
```

When `build` is invoked, `CHAINCODE_SOURCE_DIR` contains the chaincode source and `CHAINCODE_METADATA_DIR` contains the `metadata.json` file from the chaincode package installed to the peer. `BUILD_OUTPUT_DIR` is the directory where `build` must place artifacts needed by `release` and `run`. The build script should treat the input directories `CHAINCODE_SOURCE_DIR` and `CHAINCODE_METADATA_DIR` as read only, but the `BUILD_OUTPUT_DIR` is writeable.

When `build` completes with an exit code of `0`, the contents of `BUILD_OUTPUT_DIR` will be copied to the persistent storage maintained by the peer; any other exit code will be considered a failure.

The following is an example of a simple `build` script for go chaincode:

```
#!/bin/bash

CHAINCODE_SOURCE_DIR="$1"
CHAINCODE_METADATA_DIR="$2"
BUILD_OUTPUT_DIR="$3"

# extract package path from metadata.json
GO_PACKAGE_PATH="$(jq -r .path "$CHAINCODE_METADATA_DIR/metadata.json")"
if [ -f "$CHAINCODE_SOURCE_DIR/src/go.mod" ]; then
    cd "$CHAINCODE_SOURCE_DIR/src"
```

(continues on next page)

(continued from previous page)

```
    go build -v -mod=readonly -o "$BUILD_OUTPUT_DIR/chaincode" "$GO_PACKAGE_PATH"
else
    GO111MODULE=off go build -v  -o "$BUILD_OUTPUT_DIR/chaincode" "$GO_PACKAGE_PATH"
fi

# save statedb index metadata to provide at release
if [ -d "$CHAINCODE_SOURCE_DIR/META-INF" ]; then
    cp -a "$CHAINCODE_SOURCE_DIR/META-INF" "$BUILD_OUTPUT_DIR/"
fi
```

bin/release

The `bin/release` script is responsible for providing chaincode metadata to the peer. `bin/release` is optional. If it is not provided, this step is skipped. The peer invokes `release` with two arguments:

```
bin/release BUILD_OUTPUT_DIR RELEASE_OUTPUT_DIR
```

When `release` is invoked, `BUILD_OUTPUT_DIR` contains the artifacts populated by the `build` program and should be treated as read only input. `RELEASE_OUTPUT_DIR` is the directory where `release` must place artifacts to be consumed by the peer.

When `release` completes, the peer will consume two types of metadata from `RELEASE_OUTPUT_DIR`:

- state database index definitions for CouchDB
- external chaincode server connection information (`chaincode/server/connection.json`)

If CouchDB index definitions are required for the chaincode, `release` is responsible for placing the indexes into the `statedb/couchdb/indexes` directory under `RELEASE_OUTPUT_DIR`. The indexes must have a `.json` extension. See the CouchDB indexes documentation for details.

In cases where a chaincode server implementation is used, `release` is responsible for populating `chaincode/server/connection.json` with the address of the chaincode server and any TLS assets required to communicate with the chaincode. When server connection information is provided to the peer, `run` will not be called. See the Chaincode Server documentation for details.

The following is an example of a simple `release` script for go chaincode:

```
#!/bin/bash

BUILD_OUTPUT_DIR="$1"
RELEASE_OUTPUT_DIR="$2"

# copy indexes from META-INF/* to the output directory
if [ -d "$BUILD_OUTPUT_DIR/META-INF" ] ; then
   cp -a "$BUILD_OUTPUT_DIR/META-INF/"* "$RELEASE_OUTPUT_DIR/"
fi
```

bin/run

The `bin/run` script is responsible for running chaincode. The peer invokes `run` with two arguments:

```
bin/run BUILD_OUTPUT_DIR RUN_METADATA_DIR
```

When `run` is called, `BUILD_OUTPUT_DIR` contains the artifacts populated by the `build` program and `RUN_METADATA_DIR` is populated with a file called `chaincode.json` that contains the information necessary for chaincode to connect and register with the peer. Note that the `bin/run` script should treat these `BUILD_OUTPUT_DIR` and `RUN_METADATA_DIR` directories as read only input. The keys included in `chaincode.json` are:

- `chaincode_id`: The unique ID associated with the chaincode package.
- `peer_address`: The address in `host:port` format of the `ChaincodeSupport` gRPC server endpoint hosted by the peer.
- `client_cert`: The PEM encoded TLS client certificate generated by the peer that must be used when the chaincode establishes its connection to the peer.
- `client_key`: The PEM encoded client key generated by the peer that must be used when the chaincode establishes its connection to the peer.
- `root_cert`: The PEM encoded TLS root certificate for the `ChaincodeSupport` gRPC server endpoint hosted by the peer.
- `mspid`: The local mspid of the peer.

When `run` terminates, the peer considers the chaincode terminated. If another request arrives for the chaincode, the peer will attempt to start another instance of the chaincode by invoking `run` again. The contents of `chaincode.json` must not be cached across invocations.

The following is an example of a simple `run` script for go chaincode:

```
#!/bin/bash

BUILD_OUTPUT_DIR="$1"
RUN_METADATA_DIR="$2"

# setup the environment expected by the go chaincode shim
export CORE_CHAINCODE_ID_NAME="$(jq -r .chaincode_id "$RUN_METADATA_DIR/chaincode.json
↪")"
export CORE_PEER_TLS_ENABLED="true"
export CORE_TLS_CLIENT_CERT_FILE="$RUN_METADATA_DIR/client.crt"
export CORE_TLS_CLIENT_KEY_FILE="$RUN_METADATA_DIR/client.key"
export CORE_PEER_TLS_ROOTCERT_FILE="$RUN_METADATA_DIR/root.crt"
export CORE_PEER_LOCALMSPID="$(jq -r .mspid "$RUN_METADATA_DIR/chaincode.json")"

# populate the key and certificate material used by the go chaincode shim
jq -r .client_cert "$RUN_METADATA_DIR/chaincode.json" > "$CORE_TLS_CLIENT_CERT_FILE"
jq -r .client_key  "$RUN_METADATA_DIR/chaincode.json" > "$CORE_TLS_CLIENT_KEY_FILE"
jq -r .root_cert   "$RUN_METADATA_DIR/chaincode.json" > "$CORE_PEER_TLS_ROOTCERT_FILE"
if [ -z "$(jq -r .client_cert "$RUN_METADATA_DIR/chaincode.json")" ]; then
    export CORE_PEER_TLS_ENABLED="false"
fi

# exec the chaincode to replace the script with the chaincode process
exec "$BUILD_OUTPUT_DIR/chaincode" -peer.address="$(jq -r .peer_address "$ARTIFACTS/
↪chaincode.json")"
```

9.12.2 Configuring external builders and launchers

Configuring the peer to use external builders involves adding an externalBuilder element under the chaincode configuration block in the `core.yaml` that defines external builders. Each external builder definition must include a name (used for logging) and the path to parent of the `bin` directory containing the builder scripts.

An optional list of environment variable names to propagate from the peer when invoking the external builder scripts can also be provided.

The following example defines two external builders:

```
chaincode:
  externalBuilders:
  - name: my-golang-builder
    path: /builders/golang
    environmentWhitelist:
    - GOPROXY
    - GONOPROXY
    - GOSUMDB
    - GONOSUMDB
  - name: noop-builder
    path: /builders/binary
```

In this example, the implementation of "my-golang-builder" is contained within the `/builders/golang` directory and its build scripts are located in `/builders/golang/bin`. When the peer invokes any of the build scripts associated with "my-golang-builder", it will propagate only the values of the environment variables in the whitelist.

Note: The following environment variables are always propagated to external builders:

- LD_LIBRARY_PATH
- LIBPATH
- PATH
- TMPDIR

When an `externalBuilder` configuration is present, the peer will iterate over the list of builders in the order provided, invoking `bin/detect` until one completes successfully. If no builder completes `detect` successfully, the peer will fallback to using the legacy Docker build process implemented within the peer. This means that external builders are completely optional.

In the example above, the peer will attempt to use "my-golang-builder", followed by "noop-builder", and finally the peer internal build process.

9.12.3 Chaincode packages

As part of the new lifecycle introduced with Fabric 2.0, the chaincode package format changed from serialized protocol buffer messages to a gzip compressed POSIX tape archive. Chaincode packages created with `peer lifecycle chaincode package` use this new format.

Lifecycle chaincode package contents

A lifecycle chaincode package contains two files. The first file, `code.tar.gz` is a gzip compressed POSIX tape archive. This file includes the source artifacts for the chaincode. Packages created by the peer CLI will place the chaincode implementation source under the `src` directory and chaincode metadata (like CouchDB indexes) under the `META-INF` directory.

The second file, `metadata.json` is a JSON document with three keys:

- `type`: the chaincode type (e.g. GOLANG, JAVA, NODE)
- `path`: for go chaincode, the GOPATH or GOMOD relative path to the main chaincode package; undefined for other types

- `label`: the chaincode label that is used to generate the package-id by which the package is identified within the new chaincode lifecycle process.

Note that the `type` and `path` fields are only utilized by docker platform builds.

Chaincode packages and external builders

When a chaincode package is installed to a peer, the contents of `code.tar.gz` and `metadata.json` are not processed prior to calling external builders, except for the `label` field that is used by the new lifecycle process to compute the package id. This affords users a great deal of flexibility in how they package source and metadata that will be processed by external builders and launchers.

For example, a custom chaincode package could be constructed that contains a pre-compiled, implementation of chaincode in `code.tar.gz` with a `metadata.json` that allows a *binary buildpack* to detect the custom package, validate the hash of the binary, and run the program as chaincode.

Another example would be a chaincode package that only contains state database index definitions and the data necessary for an external launcher to connect to a running chaincode server. In this case, the `build` process would simply extract the metadata from the process and `release` would present it to the peer.

The only requirements are that `code.tar.gz` can only contain regular file and directory entries, and that the entries cannot contain paths that would result in files being written outside of the logical root of the chaincode package.

9.13 Chaincode as an external service

Fabric v2.0 supports chaincode deployment and execution outside of Fabric that enables users to manage a chaincode runtime independently of the peer. This facilitates deployment of chaincode on Fabric cloud deployments such as Kubernetes. Instead of building and launching the chaincode on every peer, chaincode can now run as a service whose lifecycle is managed outside of Fabric. This capability leverages the Fabric v2.0 external builder and launcher functionality which enables operators to extend a peer with programs to build, launch, and discover chaincode. Before reading this topic you should become familiar with the External Builder and Launcher content.

Prior to the availability of the external builders, the chaincode package content was required to be a set of source code files for a particular language which could be built and launched as a chaincode binary. The new external build and launcher functionality now allows users to optionally customize the build process. With respect to running the chaincode as an external service, the build process allows you to specify the endpoint information of the server where the chaincode is running. Hence the package simply consists of the externally running chaincode server endpoint information and TLS artifacts for secure connection. TLS is optional but highly recommended for all environments except a simple test environment.

The rest of this topic describes how to configure chaincode as an external service:

- *Packaging chaincode*
- *Configuring a peer to process external chaincode*
- *External builder and launcher sample scripts*
- *Writing chaincode to run as an external service*
- *Deploying the chaincode*
- *Running the chaincode as an external service*

9.13.1 Packaging chaincode

With the Fabric v2.0 chaincode lifecycle, chaincode is packaged and installed in a `.tar.gz` format. The following `myccpackage.tgz` archive demonstrates the required structure:

```
$ tar xvfz myccpackage.tgz
metadata.json
code.tar.gz
```

The chaincode package should be used to provide two pieces of information to the external builder and launcher process

- identify if the chaincode is an external service. The `bin/detect` section describes an approach using the `metadata.json` file
- provide chaincode endpoint information in a `connection.json` file placed in the release directory. The `bin/run` section describes the `connection.json` file

There is plenty of flexibility to gathering the above information. The sample scripts in the *External builder and launcher sample scripts* illustrate a simple approach to providing the information. As an example of flexibility, consider packaging couchdb index files (see Add the index to your chaincode folder). Sample scripts below describe an approach to packaging the files into code.tar.gz.

```
tar cfz code.tar.gz connection.json metadata
tar cfz $1-pkg.tgz metadata.json code.tar.gz
```

9.13.2 Configuring a peer to process external chaincode

In this section we go over the configuration needed

- to detect if the chaincode package identifies an external chaincode service
- to create the `connection.json` file in the release directory

Modify the peer core.yaml to include the externalBuilder

Assume the scripts are on the peer in the `bin` directory as follows

```
    <fully qualified path on the peer's env>
    └── bin
        ├── build
        ├── detect
        └── release
```

Modify the `chaincode` stanza of the peer `core.yaml` file to include the `externalBuilders` configuration element:

```
externalBuilders:
     - name: myexternal
       path: <fully qualified path on the peer's env>
```

External builder and launcher sample scripts

To help understand what each script needs to contain to work with the chaincode as an external service, this section contains samples of `bin/detect` `bin/build`, `bin/release`, and `bin/run` scripts.

Note: These samples use the `jq` command to parse json. You can run `jq --version` to check if you have it installed. Otherwise, install `jq` or suitably modify the scripts.

bin/detect

The `bin/detect script` is responsible for determining whether or not a buildpack should be used to build a chaincode package and launch it. For chaincode as an external service, the sample script looks for a `type` property set to `external` in the `metadata.json` file:

```
{"path":"","type":"external","label":"mycc"}
```

The peer invokes detect with two arguments:

```
bin/detect CHAINCODE_SOURCE_DIR CHAINCODE_METADATA_DIR
```

A sample `bin/detect` script could contain:

```
#!/bin/bash

set -euo pipefail

METADIR=$2
#check if the "type" field is set to "external"
if [ "$(jq -r .type "$METADIR/metadata.json")" == "external" ]; then
    exit 0
fi

exit 1
```

bin/build

For chaincode as an external service, the sample build script assumes the chaincode package's `code.tar.gz` file contains `connection.json` which it simply copies to the `BUILD_OUTPUT_DIR`. The peer invokes the build script with three arguments:

```
bin/build CHAINCODE_SOURCE_DIR CHAINCODE_METADATA_DIR BUILD_OUTPUT_DIR
```

A sample `bin/build` script could contain:

```
#!/bin/bash

set -euo pipefail

SOURCE=$1
OUTPUT=$3

#external chaincodes expect connection.json file in the chaincode package
if [ ! -f "$SOURCE/connection.json" ]; then
    >&2 echo "$SOURCE/connection.json not found"
    exit 1
fi

#simply copy the endpoint information to specified output location
cp $SOURCE/connection.json $OUTPUT/connection.json
```

(continues on next page)

(continued from previous page)

```
if [ -d "$SOURCE/metadata" ]; then
    cp -a $SOURCE/metadata $OUTPUT/metadata
fi

exit 0
```

bin/release

For chaincode as an external service, the `bin/release` script is responsible for providing the `connection.json` to the peer by placing it in the `RELEASE_OUTPUT_DIR`. The `connection.json` file has the following JSON structure

- **address** - chaincode server endpoint accessible from peer. Must be specified in ":" format.
- **dial_timeout** - interval to wait for connection to complete. Specified as a string qualified with time units (e.g, "10s", "500ms", "1m"). Default is "3s" if not specified.
- **tls_required** - true or false. If false, "client_auth_required", "client_key", "client_cert", and "root_cert" are not required. Default is "true".
- **client_auth_required** - if true, "client_key" and "client_cert" are required. Default is false. It is ignored if tls_required is false.
- **client_key** - PEM encoded string of the client private key.
- **client_cert** - PEM encoded string of the client certificate.
- **root_cert** - PEM encoded string of the server (peer) root certificate.

For example:

```
{
  "address": "your.chaincode.host.com:9999",
  "dial_timeout": "10s",
  "tls_required": "true",
  "client_auth_required": "true",
  "client_key": "-----BEGIN EC PRIVATE KEY----- ... -----END EC PRIVATE KEY-----",
  "client_cert": "-----BEGIN CERTIFICATE----- ... -----END CERTIFICATE-----",
  "root_cert": "-----BEGIN CERTIFICATE---- ... -----END CERTIFICATE-----"
}
```

As noted in the `bin/build` section, this sample assumes the chaincode package directly contains the `connection.json` file which the build script copies to the `BUILD_OUTPUT_DIR`. The peer invokes the release script with two arguments:

```
bin/release BUILD_OUTPUT_DIR RELEASE_OUTPUT_DIR
```

A sample `bin/release` script could contain:

```
#!/bin/bash

set -euo pipefail

BLD="$1"
RELEASE="$2"
```

(continues on next page)

(continued from previous page)

```
if [ -d "$BLD/metadata" ]; then
   cp -a "$BLD/metadata/"* "$RELEASE/"
fi

#external chaincodes expect artifacts to be placed under "$RELEASE"/chaincode/server
if [ -f $BLD/connection.json ]; then
   mkdir -p "$RELEASE"/chaincode/server
   cp $BLD/connection.json "$RELEASE"/chaincode/server

   #if tls_required is true, copy TLS files (using above example, the fully qualified␣
↪path for these fils would be "$RELEASE"/chaincode/server/tls)

   exit 0
fi

exit 1
```

9.13.3 Writing chaincode to run as an external service

Currently, the chaincode as an external service model is only supported by GO chaincode shim. In Fabric v2.0, the GO shim API adds a `ChaincodeServer` type that developers should use to create a chaincode server. The `Invoke` and `Query` APIs are unaffected. Developers should write to the `shim.ChaincodeServer` API, then build the chaincode and run it in the external environment of choice. Here is a simple sample chaincode program to illustrate the pattern:

```
package main

import (
        "fmt"

        "github.com/hyperledger/fabric-chaincode-go/shim"
        pb "github.com/hyperledger/fabric-protos-go/peer"
)

// SimpleChaincode example simple Chaincode implementation
type SimpleChaincode struct {
}

func (s *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
        // init code
}

func (s *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
        // invoke code
}

//NOTE - parameters such as ccid and endpoint information are hard coded here for␣
↪illustration. This can be passed in in a variety of standard ways
func main() {
       //The ccid is assigned to the chaincode on install (using the "peer lifecycle␣
↪chaincode install <package>" command) for instance
        ccid := "mycc:fcbf8724572d42e859a7dd9a7cd8e2efb84058292017df6e3d89178b64e6c831
↪"

        server := &shim.ChaincodeServer{
```

(continues on next page)

(continued from previous page)

```
                        CCID: ccid,
                        Address: "myhost:9999"
                        CC: new(SimpleChaincode),
                        TLSProps: shim.TLSProperties{
                                Disabled: true,
                        },
                }
        err := server.Start()
        if err != nil {
                fmt.Printf("Error starting Simple chaincode: %s", err)
        }
}
```

The key to running the chaincode as an external service is the use of `shim.ChaincodeServer`. This uses the new shim API `shim.ChaincodeServer` with the chaincode service properties described below:

- **CCID** (string)- CCID should match chaincode's package name on peer. This is the `CCID` associated with the installed chaincode as returned by the `peer lifecycle chaincode install <package>` CLI command. This can be obtained post-install using the "peer lifecycle chaincode queryinstalled" command.
- **Address** (string) - Address is the listen address of the chaincode server
- **CC** (Chaincode) - CC is the chaincode that handles Init and Invoke
- **TLSProps** (TLSProperties) - TLSProps is the TLS properties passed to chaincode server
- **KaOpts** (keepalive.ServerParameters) - KaOpts keepalive options, sensible defaults provided if nil

Then build the chaincode as suitable to your GO environment.

9.13.4 Deploying the chaincode

When the GO chaincode is ready for deployment, you can package the chaincode as explained in the *Packaging chaincode* section and deploy the chaincode as explained in the Fabric chaincode lifecycle concept topic.

9.13.5 Running the chaincode as an external service

Create the chaincode as specified in the *Writing chaincode to run as an external service* section. Run the built executable in your environment of choice, such as Kubernetes or directly as a process on the peer machine.

Using this chaincode as an external service model, installing the chaincode on each peer is no longer required. With the chaincode endpoint deployed to the peer instead and the chaincode running, you can continue the normal process of committing the chaincode definition to the channel and invoking the chaincode.

9.14 Error handling

9.14.1 General Overview

Hyperledger Fabric code should use the vendored package **github.com/pkg/errors** in place of the standard error type provided by Go. This package allows easy generation and display of stack traces with error messages.

9.14.2 Usage Instructions

github.com/pkg/errors should be used in place of all calls to `fmt.Errorf()` or `errors.New()`. Using this package will generate a call stack that will be appended to the error message.

Using this package is simple and will only require easy tweaks to your code.

First, you'll need to import **github.com/pkg/errors**.

Next, update all errors that are generated by your code to use one of the error creation functions (errors.New(), errors.Errorf(), errors.WithMessage(), errors.Wrap(), errors.Wrapf().

Note: See https://godoc.org/github.com/pkg/errors for complete documentation of the available error creation function. Also, refer to the General guidelines section below for more specific guidelines for using the package for Fabric code.

Finally, change the formatting directive for any logger or fmt.Printf() calls from `%s` to `%+v` to print the call stack along with the error message.

9.14.3 General guidelines for error handling in Hyperledger Fabric

- If you are servicing a user request, you should log the error and return it.
- If the error comes from an external source, such as a Go library or vendored package, wrap the error using errors.Wrap() to generate a call stack for the error.
- If the error comes from another Fabric function, add further context, if desired, to the error message using errors.WithMessage() while leaving the call stack unaffected.
- A panic should not be allowed to propagate to other packages.

9.14.4 Example program

The following example program provides a clear demonstration of using the package:

```
package main

import (
  "fmt"

  "github.com/pkg/errors"
)

func wrapWithStack() error {
  err := createError()
  // do this when error comes from external source (go lib or vendor)
  return errors.Wrap(err, "wrapping an error with stack")
}
func wrapWithoutStack() error {
  err := createError()
  // do this when error comes from internal Fabric since it already has stack trace
  return errors.WithMessage(err, "wrapping an error without stack")
}
func createError() error {
  return errors.New("original error")
```

(continues on next page)

(continued from previous page)

```
}

func main() {
  err := createError()
  fmt.Printf("print error without stack: %s\n\n", err)
  fmt.Printf("print error with stack: %+v\n\n", err)
  err = wrapWithoutStack()
  fmt.Printf("%+v\n\n", err)
  err = wrapWithStack()
  fmt.Printf("%+v\n\n", err)
}
```

9.15 Logging Control

9.15.1 Overview

Logging in the `peer` and `orderer` is provided by the `common/flogging` package. This package supports

- Logging control based on the severity of the message
- Logging control based on the software *logger* generating the message
- Different pretty-printing options based on the severity of the message

All logs are currently directed to `stderr`. Global and logger-level control of logging by severity is provided for both users and developers. There are currently no formalized rules for the types of information provided at each severity level. When submitting bug reports, developers may want to see full logs down to the DEBUG level.

In pretty-printed logs the logging level is indicated both by color and by a four-character code, e.g, "ERRO" for ERROR, "DEBU" for DEBUG, etc. In the logging context a *logger* is an arbitrary name (string) given by developers to groups of related messages. In the pretty-printed example below, the loggers `ledgermgmt`, `kvledger`, and `peer` are generating logs.

```
2018-11-01 15:32:38.268 UTC [ledgermgmt] initialize -> INFO 002 Initializing ledger␣
↪mgmt
2018-11-01 15:32:38.268 UTC [kvledger] NewProvider -> INFO 003 Initializing ledger␣
↪provider
2018-11-01 15:32:38.342 UTC [kvledger] NewProvider -> INFO 004 ledger provider␣
↪Initialized
2018-11-01 15:32:38.357 UTC [ledgermgmt] initialize -> INFO 005 ledger mgmt␣
↪initialized
2018-11-01 15:32:38.357 UTC [peer] func1 -> INFO 006 Auto-detected peer address: 172.
↪24.0.3:7051
2018-11-01 15:32:38.357 UTC [peer] func1 -> INFO 007 Returning peer0.org1.example.
↪com:7051
```

An arbitrary number of loggers can be created at runtime, therefore there is no "master list" of loggers, and logging control constructs can not check whether logging loggers actually do or will exist.

9.15.2 Logging specification

The logging levels of the `peer` and `orderer` commands are controlled by a logging specification, which is set via the `FABRIC_LOGGING_SPEC` environment variable.

The full logging level specification is of the form

```
[<logger>[,<logger>...]=]<level>[:[<logger>[,<logger>...]=]<level>...]
```

Logging severity levels are specified using case-insensitive strings chosen from

```
FATAL | PANIC | ERROR | WARNING | INFO | DEBUG
```

A logging level by itself is taken as the overall default. Otherwise, overrides for individual or groups of loggers can be specified using the

```
<logger>[,<logger>...]=<level>
```

syntax. Examples of specifications:

```
info                                        - Set default to INFO
warning:msp,gossip=warning:chaincode=info   - Default WARNING; Override for msp,␣
↪gossip, and chaincode
chaincode=info:msp,gossip=warning:warning   - Same as above
```

9.15.3 Logging format

The logging format of the `peer` and `orderer` commands is controlled via the `FABRIC_LOGGING_FORMAT` environment variable. This can be set to a format string, such as the default

```
"%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc} -> %{level:.4s}
↪%{id:03x}%{color:reset} %{message}"
```

to print the logs in a human-readable console format. It can be also set to `json` to output logs in JSON format.

9.15.4 Chaincode

Chaincode logging is the responsibility of the chaincode developer.

As independently executed programs, user-provided chaincodes may technically also produce output on stdout/stderr. While naturally useful for "devmode", these channels are normally disabled on a production network to mitigate abuse from broken or malicious code. However, it is possible to enable this output even for peer-managed containers (e.g. "netmode") on a per-peer basis via the CORE_VM_DOCKER_ATTACHSTDOUT=true configuration option.

Once enabled, each chaincode will receive its own logging channel keyed by its container-id. Any output written to either stdout or stderr will be integrated with the peer's log on a per-line basis. It is not recommended to enable this for production.

Stdout and stderr not forwarded to the peer container can be viewed from the chaincode container using standard commands for your container platform.

```
docker logs <chaincode_container_id>
kubectl logs -n <namespace> <pod_name>
oc logs -n <namespace> <pod_name>
```

9.16 Securing Communication With Transport Layer Security (TLS)

Fabric supports for secure communication between nodes using TLS. TLS communication can use both one-way (server only) and two-way (server and client) authentication.

9.16.1 Configuring TLS for peers nodes

A peer node is both a TLS server and a TLS client. It is the former when another peer node, application, or the CLI makes a connection to it and the latter when it makes a connection to another peer node or orderer.

To enable TLS on a peer node set the following peer configuration properties:

- `peer.tls.enabled` = `true`
- `peer.tls.cert.file` = fully qualified path of the file that contains the TLS server certificate
- `peer.tls.key.file` = fully qualified path of the file that contains the TLS server private key
- `peer.tls.rootcert.file` = fully qualified path of the file that contains the certificate chain of the certificate authority(CA) that issued TLS server certificate

By default, TLS client authentication is turned off when TLS is enabled on a peer node. This means that the peer node will not verify the certificate of a client (another peer node, application, or the CLI) during a TLS handshake. To enable TLS client authentication on a peer node, set the peer configuration property `peer.tls.clientAuthRequired` to `true` and set the `peer.tls.clientRootCAs.files` property to the CA chain file(s) that contain(s) the CA certificate chain(s) that issued TLS certificates for your organization's clients.

By default, a peer node will use the same certificate and private key pair when acting as a TLS server and client. To use a different certificate and private key pair for the client side, set the `peer.tls.clientCert.file` and `peer.tls.clientKey.file` configuration properties to the fully qualified path of the client certificate and key file, respectively.

TLS with client authentication can also be enabled by setting the following environment variables:

- `CORE_PEER_TLS_ENABLED` = `true`
- `CORE_PEER_TLS_CERT_FILE` = fully qualified path of the server certificate
- `CORE_PEER_TLS_KEY_FILE` = fully qualified path of the server private key
- `CORE_PEER_TLS_ROOTCERT_FILE` = fully qualified path of the CA chain file
- `CORE_PEER_TLS_CLIENTAUTHREQUIRED` = `true`
- `CORE_PEER_TLS_CLIENTROOTCAS_FILES` = fully qualified path of the CA chain file
- `CORE_PEER_TLS_CLIENTCERT_FILE` = fully qualified path of the client certificate
- `CORE_PEER_TLS_CLIENTKEY_FILE` = fully qualified path of the client key

When client authentication is enabled on a peer node, a client is required to send its certificate during a TLS handshake. If the client does not send its certificate, the handshake will fail and the peer will close the connection.

When a peer joins a channel, root CA certificate chains of the channel members are read from the config block of the channel and are added to the TLS client and server root CAs data structure. So, peer to peer communication, peer to orderer communication should work seamlessly.

9.16.2 Configuring TLS for orderer nodes

To enable TLS on an orderer node, set the following orderer configuration properties:

- `General.TLS.Enabled` = `true`
- `General.TLS.PrivateKey` = fully qualified path of the file that contains the server private key
- `General.TLS.Certificate` = fully qualified path of the file that contains the server certificate
- `General.TLS.RootCAs` = fully qualified path of the file that contains the certificate chain of the CA that issued TLS server certificate

By default, TLS client authentication is turned off on orderer, as is the case with peer. To enable TLS client authentication, set the following config properties:

- `General.TLS.ClientAuthRequired` = `true`
- `General.TLS.ClientRootCAs` = fully qualified path of the file that contains the certificate chain of the CA that issued the TLS server certificate

TLS with client authentication can also be enabled by setting the following environment variables:

- `ORDERER_GENERAL_TLS_ENABLED` = `true`
- `ORDERER_GENERAL_TLS_PRIVATEKEY` = fully qualified path of the file that contains the server private key
- `ORDERER_GENERAL_TLS_CERTIFICATE` = fully qualified path of the file that contains the server certificate
- `ORDERER_GENERAL_TLS_ROOTCAS` = fully qualified path of the file that contains the certificate chain of the CA that issued TLS server certificate
- `ORDERER_GENERAL_TLS_CLIENTAUTHREQUIRED` = `true`
- `ORDERER_GENERAL_TLS_CLIENTROOTCAS` = fully qualified path of the file that contains the certificate chain of the CA that issued TLS server certificate

9.16.3 Configuring TLS for the peer CLI

The following environment variables must be set when running peer CLI commands against a TLS enabled peer node:

- `CORE_PEER_TLS_ENABLED` = `true`
- `CORE_PEER_TLS_ROOTCERT_FILE` = fully qualified path of the file that contains cert chain of the CA that issued the TLS server cert

If TLS client authentication is also enabled on the remote server, the following variables must to be set in addition to those above:

- `CORE_PEER_TLS_CLIENTAUTHREQUIRED` = `true`
- `CORE_PEER_TLS_CLIENTCERT_FILE` = fully qualified path of the client certificate
- `CORE_PEER_TLS_CLIENTKEY_FILE` = fully qualified path of the client private key

When running a command that connects to orderer service, like *peer channel <create|update|fetch>* or *peer chaincode <invoke>*, following command line arguments must also be specified if TLS is enabled on the orderer:

- –tls
- –cafile <fully qualified path of the file that contains cert chain of the orderer CA>

If TLS client authentication is enabled on the orderer, the following arguments must be specified as well:

- –clientauth
- –keyfile <fully qualified path of the file that contains the client private key>
- –certfile <fully qualified path of the file that contains the client certificate>

9.16.4 Debugging TLS issues

Before debugging TLS issues, it is advisable to enable `GRPC debug` on both the TLS client and the server side to get additional information. To enable `GRPC debug`, set the environment variable `FABRIC_LOGGING_SPEC` to include `grpc=debug`. For example, to set the default logging level to `INFO` and the GRPC logging level to `DEBUG`, set the logging specification to `grpc=debug:info`.

If you see the error message `remote error: tls: bad certificate` on the client side, it usually means that the TLS server has enabled client authentication and the server either did not receive the correct client certificate or it received a client certificate that it does not trust. Make sure the client is sending its certificate and that it has been signed by one of the CA certificates trusted by the peer or orderer node.

If you see the error message `remote error: tls: bad certificate` in your chaincode logs, ensure that your chaincode has been built using the chaincode shim provided with Fabric v1.1 or newer.

9.17 Configuring and operating a Raft ordering service

Audience: *Raft ordering node admins*

9.17.1 Conceptual overview

For a high level overview of the concept of ordering and how the supported ordering service implementations (including Raft) work at a high level, check out our conceptual documentation on the Ordering Service.

To learn about the process of setting up an ordering node — including the creation of a local MSP and the creation of a genesis block — check out our documentation on Setting up an ordering node.

9.17.2 Configuration

While every Raft node must be added to the system channel, a node does not need to be added to every application channel. Additionally, you can remove and add a node from a channel dynamically without affecting the other nodes, a process described in the Reconfiguration section below.

Raft nodes identify each other using TLS pinning, so in order to impersonate a Raft node, an attacker needs to obtain the **private key** of its TLS certificate. As a result, it is not possible to run a Raft node without a valid TLS configuration.

A Raft cluster is configured in two planes:

- **Local configuration**: Governs node specific aspects, such as TLS communication, replication behavior, and file storage.
- **Channel configuration**: Defines the membership of the Raft cluster for the corresponding channel, as well as protocol specific parameters such as heartbeat frequency, leader timeouts, and more.

Recall, each channel has its own instance of a Raft protocol running. Thus, a Raft node must be referenced in the configuration of each channel it belongs to by adding its server and client TLS certificates (in `PEM` format) to the channel config. This ensures that when other nodes receive a message from it, they can securely confirm the identity of the node that sent the message.

The following section from `configtx.yaml` shows three Raft nodes (also called "consenters") in the channel:

```
        Consenters:
            - Host: raft0.example.com
              Port: 7050
              ClientTLSCert: path/to/ClientTLSCert0
              ServerTLSCert: path/to/ServerTLSCert0
            - Host: raft1.example.com
              Port: 7050
              ClientTLSCert: path/to/ClientTLSCert1
              ServerTLSCert: path/to/ServerTLSCert1
            - Host: raft2.example.com
              Port: 7050
```

(continues on next page)

(continued from previous page)

```
            ClientTLSCert: path/to/ClientTLSCert2
            ServerTLSCert: path/to/ServerTLSCert2
```

Note: an orderer will be listed as a consenter in the system channel as well as any application channels they're joined to.

When the channel config block is created, the `configtxgen` tool reads the paths to the TLS certificates, and replaces the paths with the corresponding bytes of the certificates.

Local configuration

The `orderer.yaml` has two configuration sections that are relevant for Raft orderers:

Cluster, which determines the TLS communication configuration. And **consensus**, which determines where Write Ahead Logs and Snapshots are stored.

Cluster parameters:

By default, the Raft service is running on the same gRPC server as the client facing server (which is used to send transactions or pull blocks), but it can be configured to have a separate gRPC server with a separate port.

This is useful for cases where you want TLS certificates issued by the organizational CAs, but used only by the cluster nodes to communicate among each other, and TLS certificates issued by a public TLS CA for the client facing API.

- `ClientCertificate`, `ClientPrivateKey`: The file path of the client TLS certificate and corresponding private key.
- `ListenPort`: The port the cluster listens on. If blank, the port is the same port as the orderer general port (`general.listenPort`)
- `ListenAddress`: The address the cluster service is listening on.
- `ServerCertificate`, `ServerPrivateKey`: The TLS server certificate key pair which is used when the cluster service is running on a separate gRPC server (different port).
- `SendBufferSize`: Regulates the number of messages in the egress buffer.

Note: `ListenPort`, `ListenAddress`, `ServerCertificate`, `ServerPrivateKey` must be either set together or unset together. If they are unset, they are inherited from the general TLS section, in example `general.tls.{privateKey, certificate}`.

There are also hidden configuration parameters for `general.cluster` which can be used to further fine tune the cluster communication or replication mechanisms:

- `DialTimeout`, `RPCTimeout`: Specify the timeouts of creating connections and establishing streams.
- `ReplicationBufferSize`: the maximum number of bytes that can be allocated for each in-memory buffer used for block replication from other cluster nodes. Each channel has its own memory buffer. Defaults to `20971520` which is `20MB`.
- `PullTimeout`: the maximum duration the ordering node will wait for a block to be received before it aborts. Defaults to five seconds.
- `ReplicationRetryTimeout`: The maximum duration the ordering node will wait between two consecutive attempts. Defaults to five seconds.
- `ReplicationBackgroundRefreshInterval`: the time between two consecutive attempts to replicate existing channels that this node was added to, or channels that this node failed to replicate in the past. Defaults to five minutes.

- `TLSHandshakeTimeShift`: If the TLS certificates of the ordering nodes expire and are not replaced in time (see TLS certificate rotation below), communication between them cannot be established, and it will be impossible to send new transactions to the ordering service. To recover from such a scenario, it is possible to make TLS handshakes between ordering nodes consider the time to be shifted backwards a given amount that is configured to `TLSHandshakeTimeShift`. In order to be as uninvasive as possible, this configuration option only effects ordering nodes that use a separate gRPC server for their intra-cluster communication. If your cluster is communicating via the same gRPC server that is used to service clients and peers, you need to first reconfigure your orderer by additionally setting `general.cluster.ListenPort`, `general.cluster.ListenAddress`, `ServerCertificate` and `ServerPrivateKey`, and then restarting the orderer in order for the new configuration to take effect.

Consensus parameters:

- `WALDir`: the location at which Write Ahead Logs for `etcd/raft` are stored. Each channel will have its own subdirectory named after the channel ID.
- `SnapDir`: specifies the location at which snapshots for `etcd/raft` are stored. Each channel will have its own subdirectory named after the channel ID.

There is also a hidden configuration parameter that can be set by adding it to the consensus section in the `orderer.yaml`:

- `EvictionSuspicion`: The cumulative period of time of channel eviction suspicion that triggers the node to pull blocks from other nodes and see if it has been evicted from the channel in order to confirm its suspicion. If the suspicion is confirmed (the inspected block doesn't contain the node's TLS certificate), the node halts its operation for that channel. A node suspects its channel eviction when it doesn't know about any elected leader nor can be elected as leader in the channel. Defaults to 10 minutes.

Channel configuration

Apart from the (already discussed) consenters, the Raft channel configuration has an `Options` section which relates to protocol specific knobs. It is currently not possible to change these values dynamically while a node is running. The node have to be reconfigured and restarted.

The only exceptions is `SnapshotIntervalSize`, which can be adjusted at runtime.

Note: It is recommended to avoid changing the following values, as a misconfiguration might lead to a state where a leader cannot be elected at all (i.e, if the `TickInterval` and `ElectionTick` are extremely low). Situations where a leader cannot be elected are impossible to resolve, as leaders are required to make changes. Because of such dangers, we suggest not tuning these parameters for most use cases.

- `TickInterval`: The time interval between two `Node.Tick` invocations.
- `ElectionTick`: The number of `Node.Tick` invocations that must pass between elections. That is, if a follower does not receive any message from the leader of current term before `ElectionTick` has elapsed, it will become candidate and start an election.
- `ElectionTick` must be greater than `HeartbeatTick`.
- `HeartbeatTick`: The number of `Node.Tick` invocations that must pass between heartbeats. That is, a leader sends heartbeat messages to maintain its leadership every `HeartbeatTick` ticks.
- `MaxInflightBlocks`: Limits the max number of in-flight append blocks during optimistic replication phase.
- `SnapshotIntervalSize`: Defines number of bytes per which a snapshot is taken.

9.17.3 Reconfiguration

The Raft orderer supports dynamic (meaning, while the channel is being serviced) addition and removal of nodes as long as only one node is added or removed at a time. Note that your cluster must be operational and able to achieve consensus before you attempt to reconfigure it. For instance, if you have three nodes, and two nodes fail, you will not be able to reconfigure your cluster to remove those nodes. Similarly, if you have one failed node in a channel with three nodes, you should not attempt to rotate a certificate, as this would induce a second fault. As a rule, you should never attempt any configuration changes to the Raft consenters, such as adding or removing a consenter, or rotating a consenter's certificate unless all consenters are online and healthy.

If you do decide to change these parameters, it is recommended to only attempt such a change during a maintenance cycle. Problems are most likely to occur when a configuration is attempted in clusters with only a few nodes while a node is down. For example, if you have three nodes in your consenter set and one of them is down, it means you have two out of three nodes alive. If you extend the cluster to four nodes while in this state, you will have only two out of four nodes alive, which is not a quorum. The fourth node won't be able to onboard because nodes can only onboard to functioning clusters (unless the total size of the cluster is one or two).

So by extending a cluster of three nodes to four nodes (while only two are alive) you are effectively stuck until the original offline node is resurrected.

Adding a new node to a Raft cluster is done by:

1. **Adding the TLS certificates** of the new node to the channel through a channel configuration update transaction. Note: the new node must be added to the system channel before being added to one or more application channels.
2. **Fetching the latest config block** of the system channel from an orderer node that's part of the system channel.
3. **Ensuring that the node that will be added is part of the system channel** by checking that the config block that was fetched includes the certificate of (soon to be) added node.
4. **Starting the new Raft node** with the path to the config block in the `General.BootstrapFile` configuration parameter.
5. **Waiting for the Raft node to replicate the blocks** from existing nodes for all channels its certificates have been added to. After this step has been completed, the node begins servicing the channel.
6. **Adding the endpoint** of the newly added Raft node to the channel configuration of all channels.

It is possible to add a node that is already running (and participates in some channels already) to a channel while the node itself is running. To do this, simply add the node's certificate to the channel config of the channel. The node will autonomously detect its addition to the new channel (the default value here is five minutes, but if you want the node to detect the new channel more quickly, reboot the node) and will pull the channel blocks from an orderer in the channel, and then start the Raft instance for that chain.

After it has successfully done so, the channel configuration can be updated to include the endpoint of the new Raft orderer.

Removing a node from a Raft cluster is done by:

1. Removing its endpoint from the channel config for all channels, including the system channel controlled by the orderer admins.
2. Removing its entry (identified by its certificates) from the channel configuration for all channels. Again, this includes the system channel.
3. Shut down the node.

Removing a node from a specific channel, but keeping it servicing other channels is done by:

1. Removing its endpoint from the channel config for the channel.
2. Removing its entry (identified by its certificates) from the channel configuration.

3. The second phase causes:
    - The remaining orderer nodes in the channel to cease communicating with the removed orderer node in the context of the removed channel. They might still be communicating on other channels.
    - The node that is removed from the channel would autonomously detect its removal either immediately or after `EvictionSuspicion` time has passed (10 minutes by default) and will shut down its Raft instance.

TLS certificate rotation for an orderer node

All TLS certificates have an expiration date that is determined by the issuer. These expiration dates can range from 10 years from the date of issuance to as little as a few months, so check with your issuer. Before the expiration date, you will need to rotate these certificates on the node itself and every channel the node is joined to, including the system channel.

For each channel the node participates in:

1. Update the channel configuration with the new certificates.
2. Replace its certificates in the file system of the node.
3. Restart the node.

Because a node can only have a single TLS certificate key pair, the node will be unable to service channels its new certificates have not been added to during the update process, degrading the capacity of fault tolerance. Because of this, **once the certificate rotation process has been started, it should be completed as quickly as possible.**

If for some reason the rotation of the TLS certificates has started but cannot complete in all channels, it is advised to rotate TLS certificates back to what they were and attempt the rotation later.

Certificate expiration related authentication

Whenever a client with an identity that has an expiration date (such as an identity based on an x509 certificate) sends a transaction to the orderer, the orderer checks whether its identity has expired, and if so, rejects the transaction submission.

However, it is possible to configure the orderer to ignore expiration of identities via enabling the `General.Authentication.NoExpirationChecks` configuration option in the `orderer.yaml`.

This should be done only under extreme circumstances, where the certificates of the administrators have expired, and due to this it is not possible to send configuration updates to replace the administrator certificates with renewed ones, because the config transactions signed by the existing administrators are now rejected because they have expired. After updating the channel it is recommended to change back to the default configuration which enforces expiration checks on identities.

9.17.4 Metrics

For a description of the Operations Service and how to set it up, check out our documentation on the Operations Service.

For a list at the metrics that are gathered by the Operations Service, check out our reference material on metrics.

While the metrics you prioritize will have a lot to do with your particular use case and configuration, there are two metrics in particular you might want to monitor:

- `consensus_etcdraft_is_leader`: identifies which node in the cluster is currently leader. If no nodes have this set, you have lost quorum.

- `consensus_etcdraft_data_persist_duration`: indicates how long write operations to the Raft cluster's persistent write ahead log take. For protocol safety, messages must be persisted durably, calling `fsync` where appropriate, before they can be shared with the consenter set. If this value begins to climb, this node may not be able to participate in consensus (which could lead to a service interruption for this node and possibly the network).

9.17.5 Troubleshooting

- The more stress you put on your nodes, the more you might have to change certain parameters. As with any system, computer or mechanical, stress can lead to a drag in performance. As we noted in the conceptual documentation, leader elections in Raft are triggered when follower nodes do not receive either a "heartbeat" messages or an "append" message that carries data from the leader for a certain amount of time. Because Raft nodes share the same communication layer across channels (this does not mean they share data — they do not!), if a Raft node is part of the consenter set in many channels, you might want to lengthen the amount of time it takes to trigger an election to avoid inadvertent leader elections.

9.18 Migrating from Kafka to Raft

Note: this document presumes a high degree of expertise with channel configuration update transactions. As the process for migration involves several channel configuration update transactions, do not attempt to migrate from Kafka to Raft without first familiarizing yourself with the Add an Organization to a Channel tutorial, which describes the channel update process in detail.

For users who want to transition channels from using Kafka-based ordering services to Raft-based ordering services, nodes at v1.4.2 or higher allow this to be accomplished through a series of configuration update transactions on each channel in the network.

This tutorial will describe this process at a high level, calling out specific details where necessary, rather than show each command in detail.

9.18.1 Assumptions and considerations

Before attempting migration, take the following into account:

1. This process is solely for migration from Kafka to Raft. Migrating between any other orderer consensus types is not currently supported.
2. Migration is one way. Once the ordering service is migrated to Raft, and starts committing transactions, it is not possible to go back to Kafka.
3. Because the ordering nodes must go down and be brought back up, downtime must be allowed during the migration.
4. Recovering from a botched migration is possible only if a backup is taken at the point in migration prescribed later in this document. If you do not take a backup, and migration fails, you will not be able to recover your previous state.
5. All channels must be migrated during the same maintenance window. It is not possible to migrate only some channels before resuming operations.
6. At the end of the migration process, every channel will have the same consenter set of Raft nodes. This is the same consenter set that will exist in the ordering system channel. This makes it possible to diagnose a successful migration.

7. Migration is done in place, utilizing the existing ledgers for the deployed ordering nodes. Addition or removal of orderers should be performed after the migration.

9.18.2 High level migration flow

Migration is carried out in five phases.

1. The system is placed into a maintenance mode where application transactions are rejected and only ordering service admins can make changes to the channel configuration.
2. The system is stopped, and a backup is taken in case an error occurs during migration.
3. The system is started, and each channel has its consensus type and metadata modified.
4. The system is restarted and is now operating on Raft consensus; each channel is checked to confirm that it has successfully achieved a quorum.
5. The system is moved out of maintenance mode and normal function resumes.

9.18.3 Preparing to migrate

There are several steps you should take before attempting to migrate.

- Design the Raft deployment, deciding which ordering service nodes are going to remain as Raft consenters. You should deploy at least three ordering nodes in your cluster, but note that deploying a consenter set of at least five nodes will maintain high availability should a node goes down, whereas a three node configuration will lose high availability once a single node goes down for any reason (for example, as during a maintenance cycle).
- Prepare the material for building the Raft `Metadata` configuration. **Note: all the channels should receive the same Raft `Metadata` configuration**. Refer to the Raft configuration guide for more information on these fields. Note: you may find it easiest to bootstrap a new ordering network with the Raft consensus protocol, then copy and modify the consensus metadata section from its config. In any case, you will need (for each ordering node):
  - `hostname`
  - `port`
  - `server certificate`
  - `client certificate`
- Compile a list of all channels (system and application) in the system. Make sure you have the correct credentials to sign the configuration updates. For example, the relevant ordering service admin identities.
- Ensure all ordering service nodes are running the same version of Fabric, and that this version is v1.4.2 or greater.
- Ensure all peers are running at least v1.4.2 of Fabric. Make sure all channels are configured with the channel capability that enables migration.
  - Orderer capability `V1_4_2` (or above).
  - Channel capability `V1_4_2` (or above).

Entry to maintenance mode

Prior to setting the ordering service into maintenance mode, it is recommended that the peers and clients of the network be stopped. Leaving peers or clients up and running is safe, however, because the ordering service will reject all of their requests, their logs will fill with benign but misleading failures.

Follow the process in the Add an Organization to a Channel tutorial to pull, translate, and scope the configuration of **each channel, starting with the system channel**. The only field you should change during this step is in the channel configuration at `/Channel/Orderer/ConsensusType`. In a JSON representation of the channel configuration, this would be `.channel_group.groups.Orderer.values.ConsensusType`.

The `ConsensusType` is represented by three values: `Type`, `Metadata`, and `State`, where:

- `Type` is either `kafka` or `etcdraft` (Raft). This value can only be changed while in maintenance mode.
- `Metadata` will be empty if the `Type` is kafka, but must carry valid Raft metadata if the `ConsensusType` is `etcdraft`. More on this below.
- `State` is either `STATE_NORMAL`, when the channel is processing transactions, or `STATE_MAINTENANCE`, during the migration process.

In the first step of the channel configuration update, only change the `State` from `STATE_NORMAL` to `STATE_MAINTENANCE`. Do not change the `Type` or the `Metadata` field yet. Note that the `Type` should currently be `kafka`.

While in maintenance mode, normal transactions, config updates unrelated to migration, and `Deliver` requests from the peers used to retrieve new blocks are rejected. This is done in order to prevent the need to both backup, and if necessary restore, peers during migration, as they only receive updates once migration has successfully completed. In other words, we want to keep the ordering service backup point, which is the next step, ahead of the peer's ledger, in order to be able to perform rollback if needed. However, ordering node admins can issue `Deliver` requests (which they need to be able to do in order to continue the migration process).

Verify that each ordering service node has entered maintenance mode on each of the channels. This can be done by fetching the last config block and making sure that the `Type`, `Metadata`, `State` on each channel is `kafka`, empty (recall that there is no metadata for Kafka), and `STATE_MAINTENANCE`, respectively.

If the channels have been updated successfully, the ordering service is now ready for backup.

Backup files and shut down servers

Shut down all ordering nodes, Kafka servers, and Zookeeper servers. It is important to **shutdown the ordering service nodes first**. Then, after allowing the Kafka service to flush its logs to disk (this typically takes about 30 seconds, but might take longer depending on your system), the Kafka servers should be shut down. Shutting down the Kafka brokers at the same time as the orderers can result in the filesystem state of the orderers being more recent than the Kafka brokers which could prevent your network from starting.

Create a backup of the file system of these servers. Then restart the Kafka service and then the ordering service nodes.

Switch to Raft in maintenance mode

The next step in the migration process is another channel configuration update for each channel. In this configuration update, switch the `Type` to `etcdraft` (for Raft) while keeping the `State` in `STATE_MAINTENANCE`, and fill in the `Metadata` configuration. It is highly recommended that the `Metadata` configuration be identical on all channels. If you want to establish different consenter sets with different nodes, you will be able to reconfigure the `Metadata` configuration after the system is restarted into `etcdraft` mode. Supplying an identical metadata object, and hence, an identical consenter set, means that when the nodes are restarted, if the system channel forms a quorum and can exit maintenance mode, other channels will likely be able do the same. Supplying different consenter sets to each channel can cause one channel to succeed in forming a cluster while another channel will fail.

Then, validate that each ordering service node has committed the `ConsensusType` change configuration update by pulling and inspecting the configuration of each channel.

Note: For each channel, the transaction that changes the `ConsensusType` must be the last configuration transaction before restarting the nodes (in the next step). If some other configuration transaction happens after this step, the nodes will most likely crash on restart, or result in undefined behavior.

Restart and validate leader

Note: exit of maintenance mode **must** be done **after** restart.

After the `ConsensusType` update has been completed on each channel, stop all ordering service nodes, stop all Kafka brokers and Zookeepers, and then restart only the ordering service nodes. They should restart as Raft nodes, form a cluster per channel, and elect a leader on each channel.

Note: Since Raft-based ordering service requires mutual TLS between orderer nodes, **additional configurations** are required before you start them again, see Section: Local Configuration for more details.

After restart process finished, make sure to **validate** that a leader has been elected on each channel by inspecting the node logs (you can see what to look for below). This will confirm that the process has been completed successfully.

When a leader is elected, the log will show, for each channel:

```
"Raft leader changed: 0 -> node-number channel=channel-name
node=node-number "
```

For example:

```
2019-05-26 10:07:44.075 UTC [orderer.consensus.etcdraft] serveRequest ->
INFO 047 Raft leader changed: 0 -> 1 channel=testchannel1 node=2
```

In this example `node 2` reports that a leader was elected (the leader is `node 1`) by the cluster of channel `testchannel1`.

Switch out of maintenance mode

Perform another channel configuration update on each channel (sending the config update to the same ordering node you have been sending configuration updates to until now), switching the `State` from `STATE_MAINTENANCE` to `STATE_NORMAL`. Start with the system channel, as usual. If it succeeds on the ordering system channel, migration is likely to succeed on all channels. To verify, fetch the last config block of the system channel from the ordering node, verifying that the `State` is now `STATE_NORMAL`. For completeness, verify this on each ordering node.

When this process is completed, the ordering service is now ready to accept all transactions on all channels. If you stopped your peers and application as recommended, you may now restart them.

9.18.4 Abort and rollback

If a problem emerges during the migration process **before exiting maintenance mode**, simply perform the rollback procedure below.

1. Shut down the ordering nodes and the Kafka service (servers and Zookeeper ensemble).
2. Rollback the file system of these servers to the backup taken at maintenance mode before changing the `ConsensusType`.
3. Restart said servers, the ordering nodes will bootstrap to Kafka in maintenance mode.
4. Send a configuration update exiting maintenance mode to continue using Kafka as your consensus mechanism, or resume the instructions after the point of backup and fix the error which prevented a Raft quorum from forming and retry migration with corrected Raft configuration `Metadata`.

There are a few states which might indicate migration has failed:

1. Some nodes crash or shutdown.
2. There is no record of a successful leader election per channel in the logs.
3. The attempt to flip to `STATE_NORMAL` mode on the system channel fails.

9.19 Bringing up a Kafka-based Ordering Service

9.19.1 Caveat emptor

This document assumes that the reader knows how to set up a Kafka cluster and a ZooKeeper ensemble, and keep them secure for general usage by preventing unauthorized access. The sole purpose of this guide is to identify the steps you need to take so as to have a set of Hyperledger Fabric ordering service nodes (OSNs) use your Kafka cluster and provide an ordering service to your blockchain network.

For information about the role orderers play in a network and in a transaction flow, checkout our *The Ordering Service* documentation.

For information on how to set up an ordering node, check out our *Setting up an ordering node* documentation.

For information about configuring Raft ordering services, check out *Configuring and operating a Raft ordering service*.

9.19.2 Big picture

Each channel maps to a separate single-partition topic in Kafka. When an OSN receives transactions via the `Broadcast` RPC, it checks to make sure that the broadcasting client has permissions to write on the channel, then relays (i.e. produces) those transactions to the appropriate partition in Kafka. This partition is also consumed by the OSN which groups the received transactions into blocks locally, persists them in its local ledger, and serves them to receiving clients via the `Deliver` RPC. For low-level details, refer to the document that describes how we came to this design. **Figure 8** is a schematic representation of the process described above.

9.19.3 Steps

Let `K` and `Z` be the number of nodes in the Kafka cluster and the ZooKeeper ensemble respectively:

1. At a minimum, `K` should be set to 4. (As we will explain in Step 4 below, this is the minimum number of nodes necessary in order to exhibit crash fault tolerance, i.e. with 4 brokers, you can have 1 broker go down, all channels will continue to be writeable and readable, and new channels can be created.)
2. `Z` will either be 3, 5, or 7. It has to be an odd number to avoid split-brain scenarios, and larger than 1 in order to avoid single point of failures. Anything beyond 7 ZooKeeper servers is considered overkill.

Then proceed as follows:

3. Orderers: **Encode the Kafka-related information in the network's genesis block.** If you are using `configtxgen`, edit `configtx.yaml`. Alternatively, pick a preset profile for the system channel's genesis block— so that:
   - `Orderer.OrdererType` is set to `kafka`.
   - `Orderer.Kafka.Brokers` contains the address of *at least two* of the Kafka brokers in your cluster in `IP:port` notation. The list does not need to be exhaustive. (These are your bootstrap brokers.)

4. Orderers: **Set the maximum block size.** Each block will have at most `Orderer.AbsoluteMaxBytes` bytes (not including headers), a value that you can set in `configtx.yaml`. Let the value you pick here be `A` and make note of it —— it will affect how you configure your Kafka brokers in Step 6.

5. Orderers: **Create the genesis block.** Use `configtxgen`. The settings you picked in Steps 3 and 4 above are system-wide settings, i.e. they apply across the network for all the OSNs. Make note of the genesis block's location.

6. Kafka cluster: **Configure your Kafka brokers appropriately.** Ensure that every Kafka broker has these keys configured:

   - `unclean.leader.election.enable = false` — Data consistency is key in a blockchain environment. We cannot have a channel leader chosen outside of the in-sync replica set, or we run the risk of overwriting the offsets that the previous leader produced, and —as a result— rewrite the blockchain that the orderers produce.

   - `min.insync.replicas = M` — Where you pick a value `M` such that `1 < M < N` (see `default.replication.factor` below). Data is considered committed when it is written to at least `M` replicas (which are then considered in-sync and belong to the in-sync replica set, or ISR). In any other case, the write operation returns an error. Then:

     - If up to `N−M` replicas —out of the `N` that the channel data is written to become unavailable, operations proceed normally.

     - If more replicas become unavailable, Kafka cannot maintain an ISR set of `M`, so it stops accepting writes. Reads work without issues. The channel becomes writeable again when `M` replicas get in-sync.

   - `default.replication.factor = N` — Where you pick a value `N` such that `N < K`. A replication factor of `N` means that each channel will have its data replicated to `N` brokers. These are the candidates for the ISR set of a channel. As we noted in the `min.insync.replicas section` above, not all of these brokers have to be available all the time. `N` should be set *strictly smaller* to `K` because channel creations cannot go forward if less than `N` brokers are up. So if you set `N = K`, a single broker going down means that no new channels can be created on the blockchain network — the crash fault tolerance of the ordering service is non-existent.

     Based on what we've described above, the minimum allowed values for `M` and `N` are 2 and 3 respectively. This configuration allows for the creation of new channels to go forward, and for all channels to continue to be writeable.

   - `message.max.bytes` and `replica.fetch.max.bytes` should be set to a value larger than `A`, the value you picked in `Orderer.AbsoluteMaxBytes` in Step 4 above. Add some buffer to account for headers —— 1 MiB is more than enough. The following condition applies:

     ```
     Orderer.AbsoluteMaxBytes < replica.fetch.max.bytes <= message.max.bytes
     ```

     (For completeness, we note that `message.max.bytes` should be strictly smaller to `socket.request.max.bytes` which is set by default to 100 MiB. If you wish to have blocks larger than 100 MiB you will need to edit the hard-coded value in `brokerConfig.Producer.MaxMessageBytes` in `fabric/orderer/kafka/config.go` and rebuild the binary from source. This is not advisable.)

   - `log.retention.ms = -1`. Until the ordering service adds support for pruning of the Kafka logs, you should disable time-based retention and prevent segments from expiring. (Size-based retention — see `log.retention.bytes` — is disabled by default in Kafka at the time of this writing, so there's no need to set it explicitly.)

7. Orderers: **Point each OSN to the genesis block.** Edit `General.BootstrapFile` in `orderer.yaml` so that it points to the genesis block created in Step 5 above. While at it, ensure all other keys in that YAML file are set appropriately.

8. Orderers: **Adjust polling intervals and timeouts.** (Optional step.)

- The `Kafka.Retry` section in the `orderer.yaml` file allows you to adjust the frequency of the metadata/producer/consumer requests, as well as the socket timeouts. (These are all settings you would expect to see in a Kafka producer or consumer.)
- Additionally, when a new channel is created, or when an existing channel is reloaded (in case of a just-restarted orderer), the orderer interacts with the Kafka cluster in the following ways:
  - It creates a Kafka producer (writer) for the Kafka partition that corresponds to the channel. . It uses that producer to post a no-op `CONNECT` message to that partition. . It creates a Kafka consumer (reader) for that partition.
  - If any of these steps fail, you can adjust the frequency with which they are repeated. Specifically they will be re-attempted every `Kafka.Retry.ShortInterval` for a total of `Kafka.Retry.ShortTotal`, and then every `Kafka.Retry.LongInterval` for a total of `Kafka.Retry.LongTotal` until they succeed. Note that the orderer will be unable to write to or read from a channel until all of the steps above have been completed successfully.

9. **Set up the OSNs and Kafka cluster so that they communicate over SSL.** (Optional step, but highly recommended.) Refer to the Confluent guide for the Kafka cluster side of the equation, and set the keys under `Kafka.TLS` in `orderer.yaml` on every OSN accordingly.
10. **Bring up the nodes in the following order: ZooKeeper ensemble, Kafka cluster, ordering service nodes.**

9.19.4 Additional considerations

1. **Preferred message size.** In Step 4 above (see *Steps* section) you can also set the preferred size of blocks by setting the `Orderer.Batchsize.PreferredMaxBytes` key. Kafka offers higher throughput when dealing with relatively small messages; aim for a value no bigger than 1 MiB.
2. **Using environment variables to override settings.** When using the sample Kafka and Zookeeper Docker images provided with Fabric (see `images/kafka` and `images/zookeeper` respectively), you can override a Kafka broker or a ZooKeeper server's settings by using environment variables. Replace the dots of the configuration key with underscores. For example, `KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE=false` will allow you to override the default value of `unclean.leader.election.enable`. The same applies to the OSNs for their *local* configuration, i.e. what can be set in `orderer.yaml`. For example `ORDERER_KAFKA_RETRY_SHORTINTERVAL=1s` allows you to override the default value for `Orderer.Kafka.Retry.ShortInterval`.

9.19.5 Kafka Protocol Version Compatibility

Fabric uses the sarama client library and vendors a version of it that supports Kafka 0.10 to 1.0, yet is still known to work with older versions.

Using the `Kafka.Version` key in `orderer.yaml`, you can configure which version of the Kafka protocol is used to communicate with the Kafka cluster's brokers. Kafka brokers are backward compatible with older protocol versions. Because of a Kafka broker's backward compatibility with older protocol versions, upgrading your Kafka brokers to a new version does not require an update of the `Kafka.Version` key value, but the Kafka cluster might suffer a performance penalty while using an older protocol version.

9.19.6 Debugging

Set environment variable `FABRIC_LOGGING_SPEC` to `DEBUG` and set `Kafka.Verbose` to `true` in `orderer.yaml` .

https://creativecommons.org/licenses/by/4.0/

CHAPTER 10

Upgrading to the latest release

If you're familiar with previous releases of Hyperledger Fabric, you're aware that upgrading the nodes and channels to the latest version of Fabric is, at a high level, a four step process.

1. Backup the ledger and MSPs.
2. Upgrade the orderer binaries in a rolling fashion to the latest Fabric version.
3. Upgrade the peer binaries in a rolling fashion to the latest Fabric version.
4. Update the orderer system channel and any application channels to the latest capability levels, where available. Note that some releases will have capabilities in all groups while other releases may have few or even no new capabilities at all.

For more information about capabilities, check out *Channel capabilities*.

For a look at how these upgrade processes are accomplished, please consult these tutorials:

1. *Considerations for getting to v2.x*. This topic discusses the important considerations for getting to the latest release from the previous release as well as from the most recent long term support (LTS) release.
2. *Upgrading your components*. Components should be upgraded to the latest version before updating any capabilities.
3. *Updating the capability level of a channel*. Completed after updating the versions of all nodes.
4. *Enabling the new chaincode lifecycle*. Necessary to add organization specific endorsement policies central to the new chaincode lifecycle for Fabric v2.x.

As the upgrading of nodes and increasing the capability levels of channels is by now considered a standard Fabric process, we will not show the specific commands for upgrading to the newest release. Similarly, there is no script in the `fabric-samples` repo that will upgrade a sample network from the previous release to this one, as there has been for previous releases.

Note: It is a best practice to upgrade your SDK to the latest version as a part of a general upgrade of your network. While the SDK will always be compatible with equivalent releases of Fabric and lower, it might be necessary to

upgrade to the latest SDK to leverage the latest Fabric features. Consult the documentation of the Fabric SDK you are using for information about how to upgrade.

10.1 Considerations for getting to v2.x

In this topic we'll cover recommendations for upgrading to the newest release from the previous release as well as from the most recent long term support (LTS) release.

10.1.1 Upgrading from 2.0 to 2.1

The 2.1 release of Fabric is a stabilization release, featuring bug fixes and other forms of code hardening. As such there are no particular considerations needed for getting from 2.0 to 2.1, and no new capability levels to update your channels to.

10.1.2 Upgrading to 2.1 from the 1.4.x long term support release

Before attempting to upgrade from v1.4.x to v2.x, make sure to consider the following:

Chaincode lifecycle

The new chaincode lifecycle that debuted in v2.0 allows multiple organizations to agree on how a chaincode will be operated before it can be used on a channel. For more information about the new chaincode lifecycle, check out Fabric chaincode lifecycle concept topic.

It is a best practice to upgrade all of the peers on a channel before enabling the `Channel` and `Application` capabilities that enable the new chaincode lifecycle (the `Channel` capability is not strictly required, but it makes sense to update it at this time). Note that any peers that are not at least at v2.0 will crash after enabling either capability, while any ordering nodes that are not at v2.0 will crash after the `Channel` capability has been enabled. This crashing behavior is intentional, as the peer or orderer cannot safely participate in the channel if it does not support the required capabilities.

After the `Application` capability has been updated to `V2_0` on a channel, you must use the v2.x lifecycle procedures to package, install, approve, and commit new chaincodes on the channel. As a result, make sure to be prepared for the new lifecycle before updating the capability.

The new lifecycle defaults to using the endorsement policy configured in the channel config (e.g., a `MAJORITY` of orgs). Therefore this endorsement policy should be added to the channel configuration when enabling capabilities on the channel.

For information about how to edit the relevant channel configurations to enable the new lifecycle by adding an endorsement policy for each organization, check out Enabling the new chaincode lifecycle.

Chaincode shim changes (Go chaincode only)

The recommended approach is to vendor the shim in your v1.4 Go chaincode before making upgrades to the peers and channels. If you do this, you do not need to make any additional changes to your chaincode.

If you did not vendor the shim in your v1.4 chaincode, the old v1.4 chaincode images will still technically work after upgrade, but you are in a risky state. If the chaincode image gets deleted from your environment for whatever reason, the next invoke on v2.x peer will try to rebuild the chaincode image and you'll get an error that the shim cannot be found.

At this point, you have two options:

1. If the entire channel is ready to upgrade chaincode, you can upgrade the chaincode on all peers and on the channel (using either the old or new lifecycle depending on the `Application` capability level you have enabled). The best practice at this point would be to vendor the new Go chaincode shim using modules.
2. If the entire channel is not yet ready to upgrade the chaincode, you can use peer environment variables to specify the v1.4 chaincode environment `ccenv` be used to rebuild the chaincode images. This v1.4 `ccenv` should still work with a v2.x peer.

Chaincode logger (Go chaincode only)

Support for user chaincodes to utilize the chaincode shim's logger via `NewLogger()` has been removed. Chaincodes that used the shim's `NewLogger()` must now shift to their own preferred logging mechanism.

For more information, check out Logging control.

Peer databases upgrade

For information about how to upgrade peers, check out our documentation on upgrading components. During the process for upgrading your peers, you will need to perform one additional step to upgrade the peer databases. The databases of all peers (which include not just the state database but the history database and other internal databases for the peer) must be rebuilt using the v2.x data format as part of the upgrade to v2.x. To trigger the rebuild, the databases must be dropped before the peer is started. The instructions below utilize the `peer node upgrade-dbs` command to drop the local databases managed by the peer and prepare them for upgrade, so that they can be rebuilt the first time the v2.x peer starts. If you are using CouchDB as the state database, since it is not a local database, the peer cannot automatically drop this database. You must drop the CouchDB state database yourself.

Follow the commands to upgrade a peer until the `docker run` command to launch the new peer container (you can skip the step where you set an `IMAGE_TAG`, since the `upgrade dbs` command is for the v2.x release of Fabric only, but you will need to set the `PEER_CONTAINER` and `LEDGERS_BACKUP` environment variables). Instead of the `docker run` command to launch the peer, run this command instead to drop and prepare the local databases managed by the peer (substitute `2.1` for `2.0` in these commands if you are upgrading to that binary version from the 1.4.x LTS):

```
docker run --rm -v /opt/backup/$PEER_CONTAINER/:/var/hyperledger/production/ \
            -v /opt/msp/:/etc/hyperledger/fabric/msp/ \
            --env-file ./env<name of node>.list \
            --name $PEER_CONTAINER \
            hyperledger/fabric-peer:2.0 peer node upgrade-dbs
```

If you are using CouchDB as the state database, also drop the CouchDB database. This can be done by removing the CouchDB /data volume directory.

Then issue this command to start the peer using the `2.0` tag:

```
docker run -d -v /opt/backup/$PEER_CONTAINER/:/var/hyperledger/production/ \
            -v /opt/msp/:/etc/hyperledger/fabric/msp/ \
            --env-file ./env<name of node>.list \
            --name $PEER_CONTAINER \
            hyperledger/fabric-peer:2.0 peer node start
```

The peer will rebuild the databases using the v2.x data format the first time it starts. Because rebuilding the databases can be a lengthy process (several hours, depending on the size of your databases), monitor the peer logs to check the status of the rebuild. Every 1000th block you will see a message like `[lockbasedtxmgr] CommitLostBlock -> INFO 041 Recommitting block [1000] to state database` indicating the rebuild is ongoing.

If the database is not dropped as part of the upgrade process, the peer start will return an error message stating that its databases are in the old format and must be dropped using the `peer node upgrade-dbs` command above (or dropped manually if using CouchDB state database). The node will then need to be restarted again.

Capabilities

The 2.0 release featured three new capabilities.

- **Application** `V2_0`: enables the new chaincode lifecycle as described in Fabric chaincode lifecycle concept topic.
- **Channel** `V2_0`: this capability has no changes, but is used for consistency with the application and orderer capability levels.
- **Orderer** `V2_0`: controls `UseChannelCreationPolicyAsAdmins`, changing the way that channel creation transactions are validated. When combined with the `-baseProfile` option of configtxgen, values which were previously inherited from the orderer system channel may now be overridden.

As with any update of the capability levels, make sure to upgrade your peer binaries before updating the `Application` and `Channel` capabilities, and make sure to upgrade your orderer binaries before updating the `Orderer` and `Channel` capabilities.

For information about how to set new capabilities, check out Updating the capability level of a channel.

Define ordering node endpoint per org (recommend)

Starting with version v1.4.2, it was recommended to define orderer endpoints in both the system channel and in all application channels at the organization level by adding a new `OrdererEndpoints` stanza within the channel configuration of an organization, replacing the the global `OrdererAddresses` section of channel configuration. If at least one organization has an ordering service endpoint defined at an organizational level, all orderers and peers will ignore the channel level endpoints when connecting to ordering nodes.

Utilizing organization level orderer endpoints is required when using service discovery with ordering nodes provided by multiple organizations. This allows clients to provide the correct organization TLS certificates.

If your channel configuration does not yet include `OrdererEndpoints` per org, you will need to perform a channel configuration update to add them to the config. First, create a JSON file that includes the new configuration stanza.

In this example, we will create a stanza for a single org called `OrdererOrg`. Note that if you have multiple ordering service organizations, they will all have to be updated to include endpoints. Let's call our JSON file `orglevelEndpoints.json`.

```
{
  "OrdererOrgEndpoint": {
      "Endpoints": {
          "mod_policy": "Admins",
          "value": {
              "addresses": [
                 "127.0.0.1:30000"
              ]
          }
      }
   }
}
```

Then, export the following environment variables:

- `CH_NAME`: the name of the channel being updated. Note that all system channels and application channels should contain organization endpoints for ordering nodes.
- `CORE_PEER_LOCALMSPID`: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the orderer organizations.
- `CORE_PEER_MSPCONFIGPATH`: the absolute path to the MSP representing your organization.
- `TLS_ROOT_CA`: the absolute path to the root CA certificate of the organization proposing the system channel update.
- `ORDERER_CONTAINER`: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.
- `ORGNAME`: The name of the organization you are currently updating. For example, `OrdererOrg`.

Once you have set the environment variables, navigate to Step 1: Pull and translate the config.

Once you have a `modified_config.json`, add the lifecycle organization policy (as listed in `orglevelEndpoints.json`) using this command:

```
jq -s ".[0] * {\"channel_group\":{\"groups\":{\"Orderer\": {\"groups\": {\"$ORGNAME\
↪": {\"values\": .[1].${ORGNAME}Endpoint}}}}}" config.json ./orglevelEndpoints.json␣
↪> modified_config.json
```

Then, follow the steps at Step 3: Re-encode and submit the config.

If every ordering service organization performs their own channel edit, they can edit the configuration without needing further signatures (by default, the only signature needed to edit parameters within an organization is an admin of that organization). If a different organization proposes the update, then the organization being edited will need to sign the channel update request.

10.2 Upgrading your components

Audience: network administrators, node administrators

For information about special considerations for the latest release of Fabric, check out Upgrading to the latest release of Fabric.

This topic will only cover the process for upgrading components. For information about how to edit a channel to change the capability level of your channels, check out Updating a channel capability.

Note: when we use the term "upgrade" in Hyperledger Fabric, we're referring to changing the version of a component (for example, going from one version of a binary to the next version). The term "update," on the other hand, refers not to versions but to configuration changes, such as updating a channel configuration or a deployment script. As there is no data migration, technically speaking, in Fabric, we will not use the term "migration" or "migrate" here.

10.2.1 Overview

At a high level, upgrading the binary level of your nodes is a two step process:

1. Backup the ledger and MSPs.
2. Upgrade binaries to the latest version.

If you own both ordering nodes and peers, it is a best practice to upgrade the ordering nodes first. If a peer falls behind or is temporarily unable to process certain transactions, it can always catch up. If enough ordering nodes go down, by comparison, a network can effectively cease to function.

This topic presumes that these steps will be performed using Docker CLI commands. If you are utilizing a different deployment method (Rancher, Kubernetes, OpenShift, etc) consult their documentation on how to use their CLI.

For native deployments, note that you will also need to update the YAML configuration file for the nodes (for example, the `orderer.yaml` file) with the one from the release artifacts.

To do this, backup the `orderer.yaml` or `core.yaml` file (for the peer) and replace it with the `orderer.yaml` or `core.yaml` file from the release artifacts. Then port any modified variables from the backed up `orderer.yaml` or `core.yaml` to the new one. Using a utility like `diff` may be helpful. Note that updating the YAML file from the release rather than updating your old YAML file **is the recommended way to update your node YAML files**, as it reduces the likelihood of making errors.

This tutorial assumes a Docker deployment where the YAML files will be baked into the images and environment variables will be used to overwrite the defaults in the configuration files.

10.2.2 Environment variables for the binaries

When you deploy a peer or an ordering node, you had to set a number of environment variables relevant to its configuration. A best practice is to create a file for these environment variables, give it a name relevant to the node being deployed, and save it somewhere on your local file system. That way you can be sure that when upgrading the peer or ordering node you are using the same variables you set when creating it.

Here's a list of some of the **peer** environment variables (with sample values — as you can see from the addresses, these environment variables are for a network deployed locally) that can be set that be listed in the file. Note that you may or may not need to set all of these environment variables:

```
CORE_PEER_TLS_ENABLED=true
CORE_PEER_GOSSIP_USELEADERELECTION=true
CORE_PEER_GOSSIP_ORGLEADER=false
CORE_PEER_PROFILE_ENABLED=true
CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
CORE_PEER_ID=peer0.org1.example.com
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
CORE_PEER_LISTENADDRESS=0.0.0.0:7051
CORE_PEER_CHAINCODEADDRESS=peer0.org1.example.com:7052
CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:7052
CORE_PEER_GOSSIP_BOOTSTRAP=peer0.org1.example.com:7051
CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051
CORE_PEER_LOCALMSPID=Org1MSP
```

Here are some **ordering node** variables (again, these are sample values) that might be listed in the environment variable file for a node. Again, you may or may not need to set all of these environment variables:

```
ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
ORDERER_GENERAL_GENESISMETHOD=file
ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
ORDERER_GENERAL_LOCALMSPID=OrdererMSP
ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
ORDERER_GENERAL_TLS_ENABLED=true
ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
ORDERER_GENERAL_CLUSTER_CLIENTCERTIFICATE=/var/hyperledger/orderer/tls/server.crt
ORDERER_GENERAL_CLUSTER_CLIENTPRIVATEKEY=/var/hyperledger/orderer/tls/server.key
ORDERER_GENERAL_CLUSTER_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
```

However you choose to set your environment variables, note that they will have to be set for each node you want to upgrade.

10.2.3 Ledger backup and restore

While we will demonstrate the process for backing up ledger data in this tutorial, it is not strictly required to backup the ledger data of a peer or an ordering node (assuming the node is part of a larger group of nodes in an ordering service). This is because, even in the worst case of catastrophic failure of a peer (such as a disk failure), the peer can be brought up with no ledger at all. You can then have the peer re-join the desired channels and as a result, the peer will automatically create a ledger for each of the channels and will start receiving the blocks via regular block transfer mechanism from either the ordering service or the other peers in the channel. As the peer processes blocks, it will also build up its state database.

However, backing up ledger data enables the restoration of a peer without the time and computational costs associated with bootstrapping from the genesis block and reprocessing all transactions, a process that can take hours (depending on the size of the ledger). In addition, ledger data backups may help to expedite the addition of a new peer, which can be achieved by backing up the ledger data from one peer and starting the new peer with the backed up ledger data.

This tutorial presumes that the file path to the ledger data has not been changed from the default value of `/var/hyperledger/production/` (for peers) or `/var/hyperledger/production/orderer` (for ordering nodes). If this location has been changed for your nodes, enter the path to the data on your ledgers in the commands below.

Note that there will be data for both the ledger and chaincodes at this file location. While it is a best practice to backup both, it is possible to skip the `stateLeveldb`, `historyLeveldb`, `chains/index` folders at `/var/hyperledger/production/ledgersData`. While skipping these folders reduces the storage needed for the backup, the peer recovery from the backed up data may take more time as these ledger artifacts will be re-constructed when the peer starts.

If using CouchDB as state database, there will be no `stateLeveldb` directory, as the state database data would be stored within CouchDB instead. But similarly, if peer starts up and finds CouchDB databases are missing or at lower block height (based on using an older CouchDB backup), the state database will be automatically re-constructed to catch up to current block height. Therefore, if you backup peer ledger data and CouchDB data separately, ensure that the CouchDB backup is always older than the peer backup.

10.2.4 Upgrade ordering nodes

Orderer containers should be upgraded in a rolling fashion (one at a time). At a high level, the ordering node upgrade process goes as follows:

1. Stop the ordering node.
2. Back up the ordering node's ledger and MSP.
3. Remove the ordering node container.
4. Launch a new ordering node container using the relevant image tag.

Repeat this process for each node in your ordering service until the entire ordering service has been upgraded.

Set command environment variables

Export the following environment variables before attempting to upgrade your ordering nodes.

- `ORDERER_CONTAINER`: the name of your ordering node container. Note that you will need to export this variable for each node when upgrading it.

- `LEDGERS_BACKUP`: the place in your local filesystem where you want to store the ledger being backed up. As you will see below, each node being backed up will have its own subfolder containing its ledger. You will need to create this folder.
- `IMAGE_TAG`: the Fabric version you are upgrading to. For example, `2.0`.

Note that you will have to set an **image tag** to ensure that the node you are starting using the correct images. The process you use to set the tag will depend on your deployment method.

Upgrade containers

Let's begin the upgrade process by **bringing down the orderer**:

```
docker stop $ORDERER_CONTAINER
```

Once the orderer is down, you'll want to **backup its ledger and MSP**:

```
docker cp $ORDERER_CONTAINER:/var/hyperledger/production/orderer/ ./$LEDGERS_BACKUP/
↪$ORDERER_CONTAINER
```

Then remove the ordering node container itself (since we will be giving our new container the same name as our old one):

```
docker rm -f $ORDERER_CONTAINER
```

Then you can launch the new ordering node container by issuing:

```
docker run -d -v /opt/backup/$ORDERER_CONTAINER/:/var/hyperledger/production/orderer/␣
↪\
            -v /opt/msp/:/etc/hyperledger/fabric/msp/ \
            --env-file ./env<name of node>.list \
            --name $ORDERER_CONTAINER \
            hyperledger/fabric-orderer:$IMAGE_TAG orderer
```

Once all of the ordering nodes have come up, you can move on to upgrading your peers.

10.2.5 Upgrade the peers

Peers should, like the ordering nodes, be upgraded in a rolling fashion (one at a time). As mentioned during the ordering node upgrade, ordering nodes and peers may be upgraded in parallel, but for the purposes of this tutorial we've separated the processes out. At a high level, we will perform the following steps:

1. Stop the peer.
2. Back up the peer's ledger and MSP.
3. Remove chaincode containers and images.
4. Remove the peer container.
5. Launch a new peer container using the relevant image tag.

Set command environment variables

Export the following environment variables before attempting to upgrade your peers.

- `PEER_CONTAINER`: the name of your peer container. Note that you will need to set this variable for each node.

- `LEDGERS_BACKUP`: the place in your local filesystem where you want to store the ledger being backed up. As you will see below, each node being backed up will have its own subfolder containing its ledger. You will need to create this folder.
- `IMAGE_TAG`: the Fabric version you are upgrading to. For example, `2.0`.

Note that you will have to set an **image tag** to ensure that the node you are starting is using the correct images. The process you use to set the tag will depend on your deployment method.

Repeat this process for each of your peers until every node has been upgraded.

Upgrade containers

Let's **bring down the first peer** with the following command:

```
docker stop $PEER_CONTAINER
```

We can then **backup the peer's ledger and MSP**:

```
docker cp $PEER_CONTAINER:/var/hyperledger/production ./$LEDGERS_BACKUP/$PEER_
↪CONTAINER
```

With the peer stopped and the ledger backed up, **remove the peer chaincode containers**:

```
CC_CONTAINERS=$(docker ps | grep dev-$PEER_CONTAINER | awk '{print $1}')
if [ -n "$CC_CONTAINERS" ] ; then docker rm -f $CC_CONTAINERS ; fi
```

And the peer chaincode images:

```
CC_IMAGES=$(docker images | grep dev-$PEER | awk '{print $1}')
if [ -n "$CC_IMAGES" ] ; then docker rmi -f $CC_IMAGES ; fi
```

Then remove the peer container itself (since we will be giving our new container the same name as our old one):

```
docker rm -f $PEER_CONTAINER
```

Then you can launch the new peer container by issuing:

```
docker run -d -v /opt/backup/$PEER_CONTAINER/:/var/hyperledger/production/ \
            -v /opt/msp/:/etc/hyperledger/fabric/msp/ \
            --env-file ./env<name of node>.list \
            --name $PEER_CONTAINER \
            hyperledger/fabric-peer:$IMAGE_TAG peer node start
```

You do not need to relaunch the chaincode container. When the peer gets a request for a chaincode, (invoke or query), it first checks if it has a copy of that chaincode running. If so, it uses it. Otherwise, as in this case, the peer launches the chaincode (rebuilding the image if required).

Verify peer upgrade completion

It's a best practice to ensure the upgrade has been completed properly with a chaincode invoke. Note that it should be possible to verify that a single peer has been successfully updated by querying one of the ledgers hosted on the peer. If you want to verify that multiple peers have been upgraded, and are updating your chaincode as part of the upgrade process, you should wait until peers from enough organizations to satisfy the endorsement policy have been upgraded.

Before you attempt this, you may want to upgrade peers from enough organizations to satisfy your endorsement policy. However, this is only mandatory if you are updating your chaincode as part of the upgrade process. If you are not

updating your chaincode as part of the upgrade process, it is possible to get endorsements from peers running at different Fabric versions.

10.2.6 Upgrade your CAs

To learn how to upgrade your Fabric CA server, click over to the CA documentation.

10.2.7 Upgrade Node SDK clients

Upgrade Fabric and Fabric CA before upgrading Node SDK clients. Fabric and Fabric CA are tested for backwards compatibility with older SDK clients. While newer SDK clients often work with older Fabric and Fabric CA releases, they may expose features that are not yet available in the older Fabric and Fabric CA releases, and are not tested for full compatibility.

Use NPM to upgrade any `Node.js` client by executing these commands in the root directory of your application:

```
npm install fabric-client@latest

npm install fabric-ca-client@latest
```

These commands install the new version of both the Fabric client and Fabric-CA client and write the new versions to `package.json`.

10.2.8 Upgrading CouchDB

If you are using CouchDB as state database, you should upgrade the peer's CouchDB at the same time the peer is being upgraded.

To upgrade CouchDB:

1. Stop CouchDB.
2. Backup CouchDB data directory.
3. Install the latest CouchDB binaries or update deployment scripts to use a new Docker image.
4. Restart CouchDB.

10.2.9 Upgrade Node chaincode shim

To move to the new version of the Node chaincode shim a developer would need to:

1. Change the level of `fabric-shim` in their chaincode `package.json` from their old level to the new one.
2. Repackage this new chaincode package and install it on all the endorsing peers in the channel.
3. Perform an upgrade to this new chaincode. To see how to do this, check out Peer chaincode commands.

10.2.10 Upgrade Chaincodes with vendored shim

For information about upgrading the Go chaincode shim specific to the v2.0 release, check out Chaincode shim changes.

A number of third party tools exist that will allow you to vendor a chaincode shim. If you used one of these tools, use the same one to update your vendored chaincode shim and re-package your chaincode.

If your chaincode vendors the shim, after updating the shim version, you must install it to all peers which already have the chaincode. Install it with the same name, but a newer version. Then you should execute a chaincode upgrade on each channel where this chaincode has been deployed to move to the new version.

10.3 Updating the capability level of a channel

Audience: network administrators, node administrators

If you're not familiar with capabilities, check out Capabilities before proceeding, paying particular attention to the fact that **peers and orderers that belong to the channel must be upgraded before enabling capabilities**.

For information about any new capability levels in the latest release of Fabric, check out Upgrading your components.

Note: when we use the term "upgrade" in Hyperledger Fabric, we're referring to changing the version of a component (for example, going from one version of a binary to the next version). The term "update," on the other hand, refers not to versions but to configuration changes, such as updating a channel configuration or a deployment script. As there is no data migration, technically speaking, in Fabric, we will not use the term "migration" or "migrate" here.

10.3.1 Prerequisites and considerations

If you haven't already done so, ensure you have all of the dependencies on your machine as described in Prerequisites. This will ensure that you have the latest versions of the tools required to make a channel configuration update.

Although Fabric binaries can and should be upgraded in a rolling fashion, it is important to **finish upgrading binaries before enabling capabilities**. Any binaries which are not upgraded to at least the level of the relevant capabilities will crash to indicate a misconfiguration which could otherwise result in a ledger fork.

Once a capability has been enabled, it becomes part of the permanent record for that channel. This means that even after disabling the capability, old binaries will not be able to participate in the channel because they cannot process beyond the block which enabled the capability to get to the block which disables it. As a result, once a capability has been enabled, disabling it is neither recommended nor supported.

For this reason, think of enabling channel capabilities as a point of no return. Please experiment with the new capabilities in a test setting and be confident before proceeding to enable them in production.

10.3.2 Overview

In this tutorial, we will show the process for updating capabilities in all of the parts of the configuration of both the ordering system channel and any application channels.

Whether you will need to update every part of the configuration for all of your channels will depend on the contents of the latest release as well as your own use case. For more information, check out Upgrading to the latest version of Fabric. Note that it may be necessary to update to the newest capability levels before using the features in the latest release, and it is considered a best practice to always be at the latest binary versions and capability levels.

Because updating the capability level of a channel involves the configuration update transaction process, we will be relying on our Updating a channel configuration topic for many of the commands.

As with any channel configuration update, updating capabilities is, at a high level, a three step process (for each channel):

1. Get the latest channel config
2. Create a modified channel config
3. Create a config update transaction

We will enable these capabilities in the following order:

1. *Orderer system channel*

- Orderer group
- Channel group

1. *Application channels*

- Orderer group
- Channel group
- Application group

While it is possible to edit multiple parts of the configuration of a channel at the same time, in this tutorial we will show how this process is done incrementally. In other words, we will not bundle a change to the `Orderer` group and the `Channel` group of the system channel into one configuration change. This is because not every release will have both a new `Orderer` group capability and a `Channel` group capability.

Note that in production networks, it will not be possible or desirable for one user to be able to update all of these channels (and parts of configurations) unilaterally. The orderer system channel, for example, is administered exclusively by ordering organization admins (though it is possible to add peer organizations as ordering service organizations). Similarly, updating either the `Orderer` or `Channel` groups of a channel configuration requires the signature of an ordering service organization in addition to peer organizations. Distributed systems require collaborative management.

Create a capabilities config file

Note that this tutorial presumes that a file called `capabilities.json` has been created and includes the capability updates you want to make to the various sections of the config. It also uses `jq` to apply the edits to the modified config file.

Note that you are not obligated to create a file like `capabilities.json` or to use a tool like `jq`. The modified config can also be edited manually (after it has been pulled, translated, and scoped). Check out this sample channel configuration for reference.

However, the process described here (using a JSON file and a tool like `jq`) does have the advantage of being scriptable, making it suitable for proposing configuration updates to a large number of channels. This is why it is **the recommended way to update channels**.

In this example, the `capabilities.json` file looks like this (note: if you are updating your channel as part of Upgrading to the latest version of Fabric you will need to set the capabilities to the levels appropriate to that release):

```
{
  "channel": {
      "mod_policy": "Admins",
          "value": {
              "capabilities": {
                  "V2_0": {}
              }
          },
      "version": "0"
  },
  "orderer": {
      "mod_policy": "Admins",
          "value": {
              "capabilities": {
                  "V2_0": {}
              }
```

(continues on next page)

(continued from previous page)

```
            },
        "version": "0"
    },
    "application": {
        "mod_policy": "Admins",
            "value": {
                "capabilities": {
                    "V2_0": {}
                }
            },
        "version": "0"
    }
}
```

Note that by default peer organizations are not admins of the orderer system channel and will therefore be unable to propose configuration updates to it. An orderer organization admin would have to create a file like this (without the `application` group capability, which does not exist in the system channel) to propose updating the system channel configuration. Note that because application channel copy the system channel configuration by default, unless a different channel profile is created which specifies capability levels, the `Channel` and `Orderer` group capabilities for the application channel will be the same as those in the network's system channel.

10.3.3 Orderer system channel capabilities

Because application channels copy the configuration of the orderer system channel by default, it is considered a best practice to update the capabilities of the system channel before any application channels. This mirrors the process of updating ordering nodes to the newest version before peers, as described in Upgrading your components.

Note that the orderer system channel is administered by ordering service organizations. By default this will be a single organization (the organization that created the initial nodes in the ordering service), but more organizations can be added here (for example, if multiple organizations have contributed nodes to the ordering service).

Make sure all of the ordering nodes in your ordering service have been upgraded to the required binary level before updating the `Orderer` and `Channel` capability. If an ordering node is not at the required level, it will be unable to process the config block with the capability and will crash. Similarly, note that if a new channel is created on this ordering service, all of the peers that will be joined to it must be at least to the node level corresponding to the `Channel` and `Application` capabilities, otherwise they will also crash when attempting to process the config block. For more information, check out Capabilities.

Set environment variables

You will need to export the following variables:

- `CH_NAME`: the name of the system channel being updated.
- `CORE_PEER_LOCALMSPID`: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the orderer organizations.
- `TLS_ROOT_CA`: the absolute path to the TLS cert of your ordering node(s).
- `CORE_PEER_MSPCONFIGPATH`: the absolute path to the MSP representing your organization.
- `ORDERER_CONTAINER`: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.

`Orderer` group

For the commands on how to pull, translate, and scope the channel config, navigate to Step 1: Pull and translate the config. Once you have a `modified_config.json`, add the capabilities to the `Orderer` group of the config (as listed in `capabilities.json`) using this command:

```
jq -s '.[0] * {"channel_group":{"groups":{"Orderer": {"values": {"Capabilities": .[1].
↪orderer}}}}}' config.json ./capabilities.json > modified_config.json
```

Then, follow the steps at Step 3: Re-encode and submit the config.

Note that because you are updating the system channel, the `mod_policy` for the system channel will only require the signature of ordering service organization admins.

`Channel` group

Once again, navigate to Step 1: Pull and translate the config. Once you have a `modified_config.json`, add the capabilities to the `Channel` group of the config (as listed in `capabilities.json`) using this command:

```
jq -s '.[0] * {"channel_group":{"values": {"Capabilities": .[1].channel}}}' config.
↪json ./capabilities.json > modified_config.json
```

Then, follow the steps at Step 3: Re-encode and submit the config.

Note that because you are updating the system channel, the `mod_policy` for the system channel will only require the signature of ordering service organization admins. In an application channel, as you'll see, you would normally need to satisfy both the `MAJORITY Admins` policy of both the `Application` group (consisting of the MSPs of peer organizations) and the `Orderer` group (consisting of ordering service organizations), assuming you have not changed the default values.

10.3.4 Enable capabilities on existing channels

Now that we have updating the capabilities on the orderer system channel, we need to updating the configuration of any existing application channels you want to update.

As you will see, the configuration of application channels is very similar to that of the system channel. This is what allows us to re-use `capabilities.json` and the same commands we used for updating the system channel (using different environment variables which we will discuss below).

Make sure all of the ordering nodes in your ordering service and peers on the channel have been upgraded to the required binary level before updating capabilities. If a peer or an ordering node is not at the required level, it will be unable to process the config block with the capability and will crash. For more information, check out Capabilities.

Set environment variables

You will need to export the following variables:

- `CH_NAME`: the name of the application channel being updated. You will have to reset this variable for every channel you update.
- `CORE_PEER_LOCALMSPID`: the MSP ID of the organization proposing the channel update. This will be the MSP of your peer organization.
- `TLS_ROOT_CA`: the absolute path to the TLS cert of your peer organization.

- `CORE_PEER_MSPCONFIGPATH`: the absolute path to the MSP representing your organization.
- `ORDERER_CONTAINER`: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.

`Orderer` group

Navigate to Step 1: Pull and translate the config. Once you have a `modified_config.json`, add the capabilities to the `Orderer` group of the config (as listed in `capabilities.json`) using this command:

```
jq -s '.[0] * {"channel_group":{"groups":{"Orderer": {"values": {"Capabilities": .[1].
↪orderer}}}}}' config.json ./capabilities.json > modified_config.json
```

Then, follow the steps at Step 3: Re-encode and submit the config.

Note the `mod_policy` for this capability defaults to the `MAJORITY` of the `Admins` of the `Orderer` group (in other words, a majority of the admins of the ordering service). Peer organizations can propose an update to this capability, but their signatures will not satisfy the relevant policy in this case.

`Channel` group

Navigate to Step 1: Pull and translate the config. Once you have a `modified_config.json`, add the capabilities to the `Channel` group of the config (as listed in `capabilities.json`) using this command:

```
jq -s '.[0] * {"channel_group":{"values": {"Capabilities": .[1].channel}}}' config.
↪json ./capabilities.json > modified_config.json
```

Then, follow the steps at Step 3: Re-encode and submit the config.

Note that the `mod_policy` for this capability defaults to requiring signatures from both the `MAJORITY` of `Admins` in the `Application` and `Orderer` groups. In other words, both a majority of the peer organization admins and ordering service organization admins must sign this request.

`Application` group

Navigate to Step 1: Pull and translate the config. Once you have a `modified_config.json`, add the capabilities to the `Application` group of the config (as listed in `capabilities.json`) using this command:

```
jq -s '.[0] * {"channel_group":{"groups":{"Application": {"values": {"Capabilities": .
↪[1].application}}}}}' config.json ./capabilities.json > modified_config.json
```

Then, follow the steps at Step 3: Re-encode and submit the config.

Note that the `mod_policy` for this capability defaults to requiring signatures from the `MAJORITY` of `Admins` in the `Application` group. In other words, a majority of peer organizations will need to approve. Ordering service admins have no say in this capability.

As a result, be very careful to not change this capability to a level that does not exist. Because ordering nodes neither understand nor validate `Application` capabilities, they will approve a configuration to any level and send the new config block to the peers to be committed to their ledgers. However, the peers will be unable to process the capability and will crash. And even it was possible to drive a corrected configuration change to a valid capability level, the previous config block with the faulty capability would still exist on the ledger and cause peers to crash when trying to process it.

This is one reason why a file like `capabilities.json` can be useful. It prevents a simple user error — for example, setting the `Application` capability to `V20` when the intent was to set it to `V2_0` — that can cause a channel to be unusable and unrecoverable.

10.3.5 Verify a transaction after capabilities have been enabled

It's a best practice to ensure that capabilities have been enabled successfully with a chaincode invoke on all channels. If any nodes that do not understand new capabilities have not been upgraded to a sufficient binary level, they will crash. You will have to upgrade their binary level before they can be successfully restarted.

10.4 Enabling the new chaincode lifecycle

Users upgrading from v1.4.x to v2.0 will have to edit their channel configurations to enable the new lifecycle features. This process involves a series of channel configuration updates the relevant users will have to perform.

Note that the `Channel` and `Application` capabilities of your application channels will have to be updated to `V2_0` for the new chaincode lifecycle to work. Check out Considerations for getting to 2.0 for more information.

Updating a channel configuration is, at a high level, a three step process (for each channel):

1. Get the latest channel config
2. Create a modified channel config
3. Create a config update transaction

We will be performing these channel configuration updates by leveraging a file called `enable_lifecycle.json`, which contains all of the updates we will be making in the channel configurations. Note that in a production setting it is likely that multiple users would be making these channel update requests. However, for the sake of simplicity, we are presenting all of the updates as how they would appear in a single file.

10.4.1 Create `enable_lifecycle.json`

Note that in addition to using `enable_lifecycle.json`, this tutorial also uses `jq` to apply the edits to the modified config file. The modified config can also be edited manually (after it has been pulled, translated, and scoped). Check out this sample channel configuration for reference.

However, the process described here (using a JSON file and a tool like `jq`) does have the advantage of being scriptable, making it suitable for proposing configuration updates to a large number of channels, and is the recommended process for editing a channel configuration.

Note that the `enable_lifecycle.json` uses sample values, for example `org1Policies` and the `Org1ExampleCom`, which will be specific to your deployment):

```
{
  "org1Policies": {
      "Endorsement": {
           "mod_policy": "Admins",
           "policy": {
               "type": 1,
               "value": {
               "identities": [
                  {
                     "principal": {
                         "msp_identifier": "Org1ExampleCom",
```

(continues on next page)

(continued from previous page)

```
                        "role": "PEER"
                    },
                    "principal_classification": "ROLE"
                }
                ],
                "rule": {
                  "n_out_of": {
                       "n": 1,
                       "rules": [
                        {
                            "signed_by": 0
                        }
                       ]
                   }
                },
                "version": 0
            }
        },
        "version": "0"
    }
  },
  "org2Policies": {
    "Endorsement": {
        "mod_policy": "Admins",
        "policy": {
            "type": 1,
            "value": {
            "identities": [
                {
                    "principal": {
                        "msp_identifier": "Org2ExampleCom",
                        "role": "PEER"
                    },
                    "principal_classification": "ROLE"
                }
                ],
                "rule": {
                  "n_out_of": {
                       "n": 1,
                       "rules": [
                        {
                            "signed_by": 0
                        }
                       ]
                   }
                },
                "version": 0
            }
        },
        "version": "0"
    }
  },
  "appPolicies": {
      "Endorsement": {
          "mod_policy": "Admins",
          "policy": {
              "type": 3,
```

(continues on next page)

(continued from previous page)

```
                "value": {
                    "rule": "MAJORITY",
                    "sub_policy": "Endorsement"
                }
            },
            "version": "0"
        },
        "LifecycleEndorsement": {
            "mod_policy": "Admins",
            "policy": {
                "type": 3,
                "value": {
                    "rule": "MAJORITY",
                    "sub_policy": "Endorsement"
                }
            },
            "version": "0"
        }
    },
    "acls": {
        "_lifecycle/CheckCommitReadiness": {
            "policy_ref": "/Channel/Application/Writers"
        },
        "_lifecycle/CommitChaincodeDefinition": {
            "policy_ref": "/Channel/Application/Writers"
        },
        "_lifecycle/QueryChaincodeDefinition": {
            "policy_ref": "/Channel/Application/Readers"
        },
        "_lifecycle/QueryChaincodeDefinitions": {
            "policy_ref": "/Channel/Application/Readers"
        }
    }
}
```

Note: the "role" field of these new policies should say `'PEER'` if NodeOUs are enabled for the org, and `'MEMBER'` if they are not.

10.4.2 Edit the channel configurations

System channel updates

Because configuration changes to the system channel to enable the new lifecycle only involve parameters inside the configuration of the peer organizations within the channel configuration, each peer organization being edited will have to sign the relevant channel configuration update.

However, by default, the system channel can only be edited by system channel admins (typically these are admins of the ordering service organizations and not peer organizations), which means that the configuration updates to the peer organizations in the consortium will have to be proposed by a system channel admin and sent to the relevant peer organization to be signed.

You will need to export the following variables:

- `CH_NAME`: the name of the system channel being updated.

- CORE_PEER_LOCALMSPID: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the ordering service organizations.
- CORE_PEER_MSPCONFIGPATH: the absolute path to the MSP representing your organization.
- TLS_ROOT_CA: the absolute path to the root CA certificate of the organization proposing the system channel update.
- ORDERER_CONTAINER: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.
- ORGNAME: the name of the organization you are currently updating.
- CONSORTIUM_NAME: the name of the consortium being updated.

Once you have set the environment variables, navigate to Step 1: Pull and translate the config.

Once you have a modified_config.json, add the lifecycle organization policy (as listed in enable_lifecycle.json) using this command:

```
jq -s ".[0] * {\"channel_group\":{\"groups\":{\"Consortiums\":{\"groups\": {\"
↪$CONSORTIUM_NAME\": {\"groups\": {\"$ORGNAME\": {\"policies\": .[1].${ORGNAME}
↪Policies}}}}}}}}" config.json ./enable_lifecycle.json > modified_config.json
```

Then, follow the steps at Step 3: Re-encode and submit the config.

As stated above, these changes will have to be proposed by a system channel admin and sent to the relevant peer organization for signature.

Application channel updates

Edit the peer organizations

We need to perform a similar set of edits to all of the organizations on all application channels.

Note that unlike the system channel, peer organizations are able to make configuration update requests to application channels. If you are making a configuration change to your own organization, you will be able to make these changes without needing the signature of other organizations. However, if you are attempting to make a change to a different organization, that organization will have to approve the change.

You will need to export the following variables:

- CH_NAME: the name of the application channel being updated.
- ORGNAME: The name of the organization you are currently updating.
- TLS_ROOT_CA: the absolute path to the TLS cert of your ordering node.
- CORE_PEER_MSPCONFIGPATH: the absolute path to the MSP representing your organization.
- CORE_PEER_LOCALMSPID: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the peer organizations.
- ORDERER_CONTAINER: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.

Once you have set the environment variables, navigate to Step 1: Pull and translate the config.

Once you have a modified_config.json, add the lifecycle organization policy (as listed in enable_lifecycle.json) using this command:

```
jq -s ".[0] * {\"channel_group\":{\"groups\":{\"Application\": {\"groups\": {\"
↪$ORGNAME\": {\"policies\": .[1].${ORGNAME}Policies}}}}}}" config.json ./enable_
↪lifecycle.json > modified_config.json
```

(continues on next page)

(continued from previous page)

Then, follow the steps at Step 3: Re-encode and submit the config.

Edit the application channels

After all of the application channels have been updated to include V2_0 capabilities, endorsement policies for the new chaincode lifecycle must be added to each channel.

You can set the same environment you set when updating the peer organizations. Note that in this case you will not be updating the configuration of an org in the configuration, so the `ORGNAME` variable will not be used.

Once you have set the environment variables, navigate to Step 1: Pull and translate the config.

Once you have a `modified_config.json`, add the channel endorsement policy (as listed in `enable_lifecycle.json`) using this command:

```
jq -s '.[0] * {"channel_group":{"groups":{"Application": {"policies": .[1].
↪appPolicies}}}}' config.json ./enable_lifecycle.json > modified_config.json
```

Then, follow the steps at Step 3: Re-encode and submit the config.

For this channel update to be approved, the policy for modifying the `Channel/Application` section of the configuration must be satisfied. By default, this is a `MAJORITY` of the peer organizations on the channel.

Edit channel ACLs (optional)

The following Access Control List (ACL) in `enable_lifecycle.json` are the default values for the new lifecycle, though you have the option to change them depending on your use case.

```
"acls": {
 "_lifecycle/CheckCommitReadiness": {
   "policy_ref": "/Channel/Application/Writers"
 },
 "_lifecycle/CommitChaincodeDefinition": {
   "policy_ref": "/Channel/Application/Writers"
 },
 "_lifecycle/QueryChaincodeDefinition": {
   "policy_ref": "/Channel/Application/Readers"
 },
 "_lifecycle/QueryChaincodeDefinitions": {
   "policy_ref": "/Channel/Application/Readers"
```

You can leave the same environment in place as when you previously edited application channels.

Once you have the environment variables set, navigate to Step 1: Pull and translate the config.

Once you have a `modified_config.json`, add the ACLs (as listed in `enable_lifecycle.json`) using this command:

```
jq -s '.[0] * {"channel_group":{"groups":{"Application": {"values": {"ACLs": {"value
↪": {"acls": .[1].acls}}}}}}' config.json ./scripts/policies.json > modified_config.
↪json
```

Then, follow the steps at Step 3: Re-encode and submit the config.

For this channel update to be approved, the policy for modifying the `Channel/Application` section of the configuration must be satisfied. By default, this is a `MAJORITY` of the peer organizations on the channel.

10.4.3 Enable new lifecycle in `core.yaml`

If you follow the recommended process for using a tool like `diff` to compare the new version of `core.yaml` packaged with the binaries with your old one, you will not need to whitelist `_lifecycle: enable` because the new `core.yaml` has added it under `chaincode/system`.

However, if you are updating your old node YAML file directly, you will have to add `_lifecycle: enable` to the system chaincodes whitelist.

For more information about upgrading nodes, check out Upgrading your components.

CHAPTER 11

Commands Reference

11.1 peer

11.1.1 Description

The `peer` command has five different subcommands, each of which allows administrators to perform a specific set of tasks related to a peer. For example, you can use the `peer channel` subcommand to join a peer to a channel, or the `peer chaincode` command to deploy a smart contract chaincode to a peer.

11.1.2 Syntax

The `peer` command has five different subcommands within it:

```
peer chaincode [option] [flags]
peer channel   [option] [flags]
peer node      [option] [flags]
peer version   [option] [flags]
```

Each subcommand has different options available, and these are described in their own dedicated topic. For brevity, we often refer to a command (`peer`), a subcommand (`channel`), or subcommand option (`fetch`) simply as a **command**.

If a subcommand is specified without an option, then it will return some high level help text as described in the `--help` flag below.

11.1.3 Flags

Each `peer` subcommand has a specific set of flags associated with it, many of which are designated *global* because they can be used in all subcommand options. These flags are described with the relevant `peer` subcommand.

The top level `peer` command has the following flag:

- `--help`

  Use `--help` to get brief help text for any `peer` command. The `--help` flag is very useful – it can be used to get command help, subcommand help, and even option help.

  For example

```
peer --help
peer channel --help
peer channel list --help
```

  See individual `peer` subcommands for more detail.

11.1.4 Usage

Here is an example using the available flag on the `peer` command.

- Using the `--help` flag on the `peer channel join` command.

```
peer channel join --help

Joins the peer to a channel.

Usage:
  peer channel join [flags]

Flags:
  -b, --blockpath string   Path to file containing genesis block
  -h, --help               help for join

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded␣
↪X509 public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating␣
↪with the orderer endpoint
      --connTimeout duration                Timeout for client to connect␣
↪(default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

  This shows brief help syntax for the `peer channel join` command.

11.2 peer chaincode

The `peer chaincode` command allows administrators to perform chaincode related operations on a peer, such as installing, instantiating, invoking, packaging, querying, and upgrading chaincode.

11.2.1 Syntax

The `peer chaincode` command has the following subcommands:

- install
- instantiate
- invoke
- list
- package
- query
- signpackage
- upgrade

The different subcommand options (install, instantiate…) relate to the different chaincode operations that are relevant to a peer. For example, use the `peer chaincode install` subcommand option to install a chaincode on a peer, or the `peer chaincode query` subcommand option to query a chaincode for the current value on a peer's ledger.

Each peer chaincode subcommand is described together with its options in its own section in this topic.

11.2.2 Flags

Each `peer chaincode` subcommand has both a set of flags specific to an individual subcommand, as well as a set of global flags that relate to all `peer chaincode` subcommands. Not all subcommands would use these flags. For instance, the `query` subcommand does not need the `--orderer` flag.

The individual flags are described with the relevant subcommand. The global flags are

- `--cafile <string>`

  Path to file containing PEM-encoded trusted certificate(s) for the ordering endpoint

- `--certfile <string>`

  Path to file containing PEM-encoded X509 public key to use for mutual TLS communication with the orderer endpoint

- `--keyfile <string>`

  Path to file containing PEM-encoded private key to use for mutual TLS communication with the orderer endpoint

- `-o` or `--orderer <string>`

  Ordering service endpoint specified as `<hostname or IP address>:<port>`

- `--ordererTLSHostnameOverride <string>`

  The hostname override to use when validating the TLS connection to the orderer

- `--tls`

  Use TLS when communicating with the orderer endpoint

- `--transient <string>`

  Transient map of arguments in JSON encoding

11.2.3 peer chaincode install

```
Install a chaincode on a peer. This installs a chaincode deployment spec package (if␣
↪provided) or packages the specified chaincode before subsequently installing it.

Usage:
  peer chaincode install [flags]

Flags:
      --connectionProfile string       Connection profile that provides the necessary␣
↪connection information for the network. Note: currently only supported for␣
↪providing peer connection information
  -c, --ctor string                    Constructor message for the chaincode in JSON␣
↪format (default "{}")
  -h, --help                           help for install
  -l, --lang string                    Language the chaincode is written in (default
↪"golang")
  -n, --name string                    Name of the chaincode
  -p, --path string                    Path to chaincode
      --peerAddresses stringArray      The addresses of the peers to connect to
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag
  -v, --version string                 Version of the chaincode specified in install/
↪instantiate/upgrade commands

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
      --transient string                    Transient map of arguments in JSON␣
↪encoding
```

11.2.4 peer chaincode instantiate

```
Deploy the specified chaincode to the network.

Usage:
  peer chaincode instantiate [flags]

Flags:
  -C, --channelID string               The channel on which this command should be␣
↪executed
      --collections-config string      The fully qualified path to the collection␣
↪JSON file including the file name
```

(continues on next page)

(continued from previous page)

```
      --connectionProfile string       Connection profile that provides the necessary␣
↪connection information for the network. Note: currently only supported for␣
↪providing peer connection information
  -c, --ctor string                    Constructor message for the chaincode in JSON␣
↪format (default "{}")
  -E, --escc string                    The name of the endorsement system chaincode␣
↪to be used for this chaincode
  -h, --help                           help for instantiate
  -l, --lang string                    Language the chaincode is written in (default
↪"golang")
  -n, --name string                    Name of the chaincode
      --peerAddresses stringArray      The addresses of the peers to connect to
  -P, --policy string                  The endorsement policy associated to this␣
↪chaincode
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag
  -v, --version string                 Version of the chaincode specified in install/
↪instantiate/upgrade commands
  -V, --vscc string                    The name of the verification system chaincode␣
↪to be used for this chaincode

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
      --transient string                    Transient map of arguments in JSON␣
↪encoding
```

11.2.5 peer chaincode invoke

```
Invoke the specified chaincode. It will try to commit the endorsed transaction to the␣
↪network.

Usage:
  peer chaincode invoke [flags]

Flags:
  -C, --channelID string               The channel on which this command should be␣
↪executed
      --connectionProfile string       Connection profile that provides the necessary␣
↪connection information for the network. Note: currently only supported for␣
↪providing peer connection information
  -c, --ctor string                    Constructor message for the chaincode in JSON␣
↪format (default "{}")
```

(continues on next page)

(continued from previous page)

```
  -h, --help                          help for invoke
  -I, --isInit                        Is this invocation for init (useful for␣
↪supporting legacy chaincodes in the new lifecycle)
  -n, --name string                   Name of the chaincode
      --peerAddresses stringArray     The addresses of the peers to connect to
      --tlsRootCertFiles stringArray  If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag
      --waitForEvent                  Whether to wait for the event from each peer's␣
↪deliver filtered service signifying that the 'invoke' transaction has been␣
↪committed successfully
      --waitForEventTimeout duration  Time to wait for the event from each peer's␣
↪deliver filtered service signifying that the 'invoke' transaction has been␣
↪committed successfully (default 30s)

Global Flags:
      --cafile string                     Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                   Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                        Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration              Timeout for client to connect (default 3s)
      --keyfile string                    Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                    Ordering service endpoint
      --ordererTLSHostnameOverride string  The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                               Use TLS when communicating with the␣
↪orderer endpoint
      --transient string                  Transient map of arguments in JSON␣
↪encoding
```

11.2.6 peer chaincode list

```
Get the instantiated chaincodes in the channel if specify channel, or get installed␣
↪chaincodes on the peer

Usage:
  peer chaincode list [flags]

Flags:
  -C, --channelID string              The channel on which this command should be␣
↪executed
      --connectionProfile string      Connection profile that provides the necessary␣
↪connection information for the network. Note: currently only supported for␣
↪providing peer connection information
  -h, --help                          help for list
      --installed                     Get the installed chaincodes on a peer
      --instantiated                  Get the instantiated chaincodes on a channel
      --peerAddresses stringArray     The addresses of the peers to connect to
      --tlsRootCertFiles stringArray  If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag
```

(continues on next page)

(continued from previous page)

```
Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
      --transient string                    Transient map of arguments in JSON␣
↪encoding
```

11.2.7 peer chaincode package

```
Package a chaincode and write the package to a file.

Usage:
  peer chaincode package [outputfile] [flags]

Flags:
  -s, --cc-package                  create CC deployment spec for owner endorsements␣
↪instead of raw CC deployment spec
  -c, --ctor string                 Constructor message for the chaincode in JSON␣
↪format (default "{}")
  -h, --help                        help for package
  -i, --instantiate-policy string   instantiation policy for the chaincode
  -l, --lang string                 Language the chaincode is written in (default
↪"golang")
  -n, --name string                 Name of the chaincode
  -p, --path string                 Path to chaincode
  -S, --sign                        if creating CC deployment spec package for owner␣
↪endorsements, also sign it with local MSP
  -v, --version string              Version of the chaincode specified in install/
↪instantiate/upgrade commands

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

(continues on next page)

(continued from previous page)

```
      --transient string                    Transient map of arguments in JSON␣
↪encoding
```

11.2.8 peer chaincode query

```
Get endorsed result of chaincode function call and print it. It won't generate␣
↪transaction.

Usage:
  peer chaincode query [flags]

Flags:
  -C, --channelID string                The channel on which this command should be␣
↪executed
      --connectionProfile string        Connection profile that provides the necessary␣
↪connection information for the network. Note: currently only supported for␣
↪providing peer connection information
  -c, --ctor string                     Constructor message for the chaincode in JSON␣
↪format (default "{}")
  -h, --help                            help for query
  -x, --hex                             If true, output the query value byte array in␣
↪hexadecimal. Incompatible with --raw
  -n, --name string                     Name of the chaincode
      --peerAddresses stringArray       The addresses of the peers to connect to
  -r, --raw                             If true, output the query value as raw bytes,␣
↪otherwise format as a printable string
      --tlsRootCertFiles stringArray    If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
      --transient string                    Transient map of arguments in JSON␣
↪encoding
```

11.2.9 peer chaincode signpackage

```
Sign the specified chaincode package
```

(continues on next page)

(continued from previous page)

```
Usage:
  peer chaincode signpackage [flags]

Flags:
  -h, --help   help for signpackage

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
      --transient string                    Transient map of arguments in JSON␣
↪encoding
```

11.2.10 peer chaincode upgrade

```
Upgrade an existing chaincode with the specified one. The new chaincode will␣
↪immediately replace the existing chaincode upon the transaction committed.

Usage:
  peer chaincode upgrade [flags]

Flags:
  -C, --channelID string               The channel on which this command should be␣
↪executed
      --collections-config string      The fully qualified path to the collection␣
↪JSON file including the file name
      --connectionProfile string       Connection profile that provides the necessary␣
↪connection information for the network. Note: currently only supported for␣
↪providing peer connection information
  -c, --ctor string                    Constructor message for the chaincode in JSON␣
↪format (default "{}")
  -E, --escc string                    The name of the endorsement system chaincode␣
↪to be used for this chaincode
  -h, --help                           help for upgrade
  -l, --lang string                    Language the chaincode is written in (default
↪"golang")
  -n, --name string                    Name of the chaincode
  -p, --path string                    Path to chaincode
      --peerAddresses stringArray      The addresses of the peers to connect to
  -P, --policy string                  The endorsement policy associated to this␣
↪chaincode
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag
```

(continues on next page)

(continued from previous page)

```
  -v, --version string                 Version of the chaincode specified in install/
↪instantiate/upgrade commands
  -V, --vscc string                    The name of the verification system chaincode␣
↪to be used for this chaincode

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
      --transient string                    Transient map of arguments in JSON␣
↪encoding
```

11.2.11 Example Usage

peer chaincode instantiate examples

Here are some examples of the `peer chaincode instantiate` command, which instantiates the chaincode named `mycc` at version `1.0` on channel `mychannel`:

- Using the `--tls` and `--cafile` global flags to instantiate the chaincode in a network with TLS enabled:

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem
peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile $ORDERER_CA␣
↪-C mychannel -n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "AND (
↪'Org1MSP.peer','Org2MSP.peer')"

2018-02-22 16:33:53.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001␣
↪Using default escc
2018-02-22 16:33:53.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002␣
↪Using default vscc
2018-02-22 16:34:08.698 UTC [main] main -> INFO 003 Exiting.....
```

- Using only the command-specific options to instantiate the chaincode in a network with TLS disabled:

```
peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n mycc -v 1.
↪0 -c '{"Args":["init","a","100","b","200"]}' -P "AND ('Org1MSP.peer','Org2MSP.
↪peer')"


2018-02-22 16:34:09.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001␣
↪Using default escc
2018-02-22 16:34:09.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002␣
↪Using default vscc
```

(continues on next page)

(continued from previous page)

```
2018-02-22 16:34:24.698 UTC [main] main -> INFO 003 Exiting.....
```

peer chaincode invoke example

Here is an example of the `peer chaincode invoke` command:

- Invoke the chaincode named `mycc` at version `1.0` on channel `mychannel` on `peer0.org1.example.com:7051` and `peer0.org2.example.com:9051` (the peers defined by `--peerAddresses`), requesting to move 10 units from variable `a` to variable `b`:

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n mycc --
↪peerAddresses peer0.org1.example.com:7051 --peerAddresses peer0.org2.example.
↪com:9051 -c '{"Args":["invoke","a","b","10"]}'

2018-02-22 16:34:27.069 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001␣
↪Using default escc
2018-02-22 16:34:27.069 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002␣
↪Using default vscc
.
.
.
2018-02-22 16:34:27.106 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 00a␣
↪ESCC invoke result: version:1 response:<status:200 message:"OK" > payload:"\n␣
↪\237mM\376? [\214\002 \332\204\035\275q\227\2132A\n\204&\2106\037W|\346
↪#\3413\274\022Y\nE\022\024\n\004lscc\022\014\n\n\n\004mycc\022\002\010\003\022-
↪\n\004mycc\022
↪%\n\007\n\001a\022\002\010\003\n\007\n\001b\022\002\010\003\032\007\n\001a\032\00290\032\010\n
↪"\013\022\004mycc\032\0031.0" endorsement:<endorser:"\n\007Org1MSP\022\262\006--
↪---BEGIN CERTIFICATE-----\nMIICLjCCAdWgAwIBAgIRAJYomxY2cqHA/fbRnH5a/
↪bwwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcTDVNhbiBG\ncmF
↪/7JFDHATJXtLgJhkK5KosDdHuKLYbCqvge\n46u3AC16MZyJRvKBiw6jTTBLMA4GA1UdDwEB/
↪wQEAwIHgDAMBgNVHRMBAf8EAjAA\nMCsGA1UdIwQkMCKAIN7dJR9dimkFtkus0R5pAOlRz5SA3FB5t8Eaxl9A7lkgMAoG\
↪Xj3C8lA==\n-----END CERTIFICATE-----\n" signature:"0D\002 \022_
↪\342\350\344\231G&
↪\237\n\244\375\302J\220l\302\345\210\335D\250y\253P\0214:\221e\332@\002␣
↪\000\254\361\224\247\210\214L\277\370\222\213\217\301\r\341v\227\265\277\336\256^
↪\217\336\005y*\321\023\025\367" >
2018-02-22 16:34:27.107 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00b␣
↪Chaincode invoke successful. result: status:200
2018-02-22 16:34:27.107 UTC [main] main -> INFO 00c Exiting.....
```

  Here you can see that the invoke was submitted successfully based on the log message:

```
2018-02-22 16:34:27.107 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00b␣
↪Chaincode invoke successful. result: status:200
```

  A successful response indicates that the transaction was submitted for ordering successfully. The transaction will then be added to a block and, finally, validated or invalidated by each peer on the channel.

peer chaincode list example

Here are some examples of the `peer chaincode list` command:

- Using the `--installed` flag to list the chaincodes installed on a peer.

```
peer chaincode list --installed

Get installed chaincodes on peer:
Name: mycc, Version: 1.0, Path: github.com/hyperledger/fabric-samples/chaincode/
↪abstore/go, Id: 8cc2730fdafd0b28ef734eac12b29df5fc98ad98bdb1b7e0ef96265c3d893d61
2018-02-22 17:07:13.476 UTC [main] main -> INFO 001 Exiting.....
```

You can see that the peer has installed a chaincode called `mycc` which is at version `1.0`.

- Using the `--instantiated` in combination with the `-C` (channel ID) flag to list the chaincodes instantiated on a channel.

```
peer chaincode list --instantiated -C mychannel

Get instantiated chaincodes on channel mychannel:
Name: mycc, Version: 1.0, Path: github.com/hyperledger/fabric-samples/chaincode/
↪abstore/go, Escc: escc, Vscc: vscc
2018-02-22 17:07:42.969 UTC [main] main -> INFO 001 Exiting.....
```

You can see that chaincode `mycc` at version `1.0` is instantiated on channel `mychannel`.

peer chaincode package example

Here is an example of the `peer chaincode package` command, which packages the chaincode named `mycc` at version `1.1`, creates the chaincode deployment spec, signs the package using the local MSP, and outputs it as `ccpack.out`:

```
 peer chaincode package ccpack.out -n mycc -p github.com/hyperledger/fabric-samples/
↪chaincode/abstore/go -v 1.1 -s -S
 .
 .
 .
 2018-02-22 17:27:01.404 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003␣
↪Using default escc
 2018-02-22 17:27:01.405 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004␣
↪Using default vscc
 .
 .
 .
 2018-02-22 17:27:01.879 UTC [chaincodeCmd] chaincodePackage -> DEBU 011 Packaged␣
↪chaincode into deployment spec of size <3426>, with args = [ccpack.out]
 2018-02-22 17:27:01.879 UTC [main] main -> INFO 012 Exiting.....
```

peer chaincode query example

Here is an example of the `peer chaincode query` command, which queries the peer ledger for the chaincode named `mycc` at version `1.0` for the value of variable `a`:

- You can see from the output that variable `a` had a value of 90 at the time of the query.

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'

2018-02-22 16:34:30.816 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001␣
↪Using default escc
```

(continues on next page)

(continued from previous page)

```
2018-02-22 16:34:30.816 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002␣
↪Using default vscc
Query Result: 90
```

peer chaincode signpackage example

Here is an example of the `peer chaincode signpackage` command, which accepts an existing signed package and creates a new one with signature of the local MSP appended to it.

```
peer chaincode signpackage ccwith1sig.pak ccwith2sig.pak
Wrote signed package to ccwith2sig.pak successfully
2018-02-24 19:32:47.189 EST [main] main -> INFO 002 Exiting.....
```

peer chaincode upgrade example

Here is an example of the `peer chaincode upgrade` command, which upgrades the chaincode named `mycc` at version `1.1` on channel `mychannel` to version `1.2`, which contains a new variable `c`:

- Using the `--tls` and `--cafile` global flags to upgrade the chaincode in a network with TLS enabled:

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem
peer chaincode upgrade -o orderer.example.com:7050 --tls --cafile $ORDERER_CA -C␣
↪mychannel -n mycc -v 1.2 -c '{"Args":["init","a","100","b","200","c","300"]}' -
↪P "AND ('Org1MSP.peer','Org2MSP.peer')"
.
.
.
2018-02-22 18:26:31.433 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003␣
↪Using default escc
2018-02-22 18:26:31.434 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004␣
↪Using default vscc
2018-02-22 18:26:31.435 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java␣
↪chaincode enabled
2018-02-22 18:26:31.435 UTC [chaincodeCmd] upgrade -> DEBU 006 Get upgrade␣
↪proposal for chaincode <name:"mycc" version:"1.1" >
.
.
.
2018-02-22 18:26:46.687 UTC [chaincodeCmd] upgrade -> DEBU 009 endorse upgrade␣
↪proposal, get response <status:200 message:"OK" payload:"\n\004mycc\022\0031.
↪1\032\004escc\"\004vscc*,
↪\022\014\022\n\010\001\022\002\010\000\022\002\010\001\032\r\022\013\n\007Org1MSP\020\003\032\
↪\261g(^
↪v\021\220\240\332\251\014\204V\210P\310o\231\271\036\301\022\032\205fC[|=\215\372\223\022␣
↪\311b\025?
↪\323N\343\325\032\005\365\236\001XKj\004E\351\007\247\265fu\305j\367\331\275\253\307R\032␣
↪\014H#\014\272!#\345\306s\323\371\350\364\006.
↪\000\356\230\353\270\263\215\217\303\256\220i^\277\305\214: \375\200zY\275\203}
↪\375\244\205\035\340\226]l!uE\334\273\214\214\020\303\3474\360\014\234-
↪\006\315B\031\022\010\022\006\010\001\022\002\010\000\032\r\022\013\n\007Org1MSP\020\001
↪" >
.
```

(continues on next page)

(continued from previous page)

```
.
.
2018-02-22 18:26:46.693 UTC [chaincodeCmd] upgrade -> DEBU 00c Get Signed envelope
2018-02-22 18:26:46.693 UTC [chaincodeCmd] chaincodeUpgrade -> DEBU 00d Send␣
↪signed envelope to orderer
2018-02-22 18:26:46.908 UTC [main] main -> INFO 00e Exiting.....
```

- Using only the command-specific options to upgrade the chaincode in a network with TLS disabled:

```
peer chaincode upgrade -o orderer.example.com:7050 -C mychannel -n mycc -v 1.2 -c
↪'{"Args":["init","a","100","b","200","c","300"]}' -P "AND ('Org1MSP.peer',
↪'Org2MSP.peer')"
.
.
.
2018-02-22 18:28:31.433 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003␣
↪Using default escc
2018-02-22 18:28:31.434 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004␣
↪Using default vscc
2018-02-22 18:28:31.435 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java␣
↪chaincode enabled
2018-02-22 18:28:31.435 UTC [chaincodeCmd] upgrade -> DEBU 006 Get upgrade␣
↪proposal for chaincode <name:"mycc" version:"1.1" >
.
.
.
2018-02-22 18:28:46.687 UTC [chaincodeCmd] upgrade -> DEBU 009 endorse upgrade␣
↪proposal, get response <status:200 message:"OK" payload:"\n\004mycc\022\0031.
↪1\032\004escc\"\004vscc*,
↪\022\014\022\n\010\001\022\002\010\000\022\002\010\001\032\r\022\013\n\007Org1MSP\020\003\032\
↪\261g(^
↪v\021\220\240\332\251\014\204V\210P\310o\231\271\036\301\022\032\205fC[|=\215\372\223\022␣
↪\311b\025?
↪\323N\343\325\032\005\365\236\001XKj\004E\351\007\247\265fu\305j\367\331\275\253\307R\032␣
↪\014H#\014\272!#\345\306s\323\371\350\364\006.
↪\000\356\230\353\270\263\215\217\303\256\220i^\277\305\214: \375\200zY\275\203}
↪\375\244\205\035\340\226]l!uE\334\273\214\214\020\303\3474\360\014\234-
↪\006\315B\031\022\010\022\006\010\001\022\002\010\000\032\r\022\013\n\007Org1MSP\020\001
↪" >
.
.
.
2018-02-22 18:28:46.693 UTC [chaincodeCmd] upgrade -> DEBU 00c Get Signed envelope
2018-02-22 18:28:46.693 UTC [chaincodeCmd] chaincodeUpgrade -> DEBU 00d Send␣
↪signed envelope to orderer
2018-02-22 18:28:46.908 UTC [main] main -> INFO 00e Exiting.....
```

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.3 peer lifecycle chaincode

The `peer lifecycle chaincode` subcommand allows administrators to use the Fabric chaincode lifecycle to package a chaincode, install it on your peers, approve a chaincode definition for your organization, and then commit the definition to a channel. The chaincode is ready to be used after the definition has been successfully committed to the channel. For more information, visit Fabric chaincode lifecycle.

Note: These instructions use the Fabric chaincode lifecycle introduced in the v2.0 release. If you would like to use the old lifecycle to install and instantiate a chaincode, visit the peer chaincode command reference.

11.3.1 Syntax

The `peer lifecycle chaincode` command has the following subcommands:

- package
- install
- queryinstalled
- getinstalledpackage
- approveformyorg
- checkcommitreadiness
- commit
- querycommitted

Each peer lifecycle chaincode subcommand is described together with its options in its own section in this topic.

11.3.2 peer lifecycle

```
Perform _lifecycle operations

Usage:
  peer lifecycle [command]

Available Commands:
  chaincode   Perform chaincode operations:␣
↪package|install|queryinstalled|getinstalledpackage|approveformyorg|checkcommitreadiness|commit|quer

Flags:
  -h, --help   help for lifecycle

Use "peer lifecycle [command] --help" for more information about a command.
```

11.3.3 peer lifecycle chaincode

```
Perform chaincode operations:␣
↪package|install|queryinstalled|getinstalledpackage|approveformyorg|checkcommitreadiness|commit|quer

Usage:
  peer lifecycle chaincode [command]

Available Commands:
  approveformyorg      Approve the chaincode definition for my org.
  checkcommitreadiness Check whether a chaincode definition is ready to be committed␣
↪on a channel.
  commit               Commit the chaincode definition on the channel.
  getinstalledpackage  Get an installed chaincode package from a peer.
  install              Install a chaincode.
```

(continues on next page)

(continued from previous page)

```
  package              Package a chaincode
  querycommitted       Query the committed chaincode definitions by channel on a peer.
  queryinstalled       Query the installed chaincodes on a peer.

Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
  -h, --help                                help for chaincode
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint

Use "peer lifecycle chaincode [command] --help" for more information about a command.
```

11.3.4 peer lifecycle chaincode package

```
Package a chaincode and write the package to a file.

Usage:
  peer lifecycle chaincode package [outputfile] [flags]

Flags:
      --connectionProfile string       The fully qualified path to the connection␣
↪profile that provides the necessary connection information for the network. Note:␣
↪currently only supported for providing peer connection information
  -h, --help                           help for package
      --label string                   The package label contains a human-readable␣
↪description of the package
  -l, --lang string                    Language the chaincode is written in (default
↪"golang")
  -p, --path string                    Path to the chaincode
      --peerAddresses stringArray      The addresses of the peers to connect to
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
```

(continues on next page)

(continued from previous page)

```
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.3.5 peer lifecycle chaincode install

```
Install a chaincode on a peer.

Usage:
  peer lifecycle chaincode install [flags]

Flags:
      --connectionProfile string       The fully qualified path to the connection␣
↪profile that provides the necessary connection information for the network. Note:␣
↪currently only supported for providing peer connection information
  -h, --help                           help for install
      --peerAddresses stringArray      The addresses of the peers to connect to
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag

Global Flags:
      --cafile string                      Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                    Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                         Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration               Timeout for client to connect (default 3s)
      --keyfile string                     Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                     Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                Use TLS when communicating with the␣
↪orderer endpoint
```

11.3.6 peer lifecycle chaincode queryinstalled

```
Query the installed chaincodes on a peer.

Usage:
  peer lifecycle chaincode queryinstalled [flags]

Flags:
      --connectionProfile string       The fully qualified path to the connection␣
↪profile that provides the necessary connection information for the network. Note:␣
↪currently only supported for providing peer connection information
  -h, --help                           help for queryinstalled
  -O, --output string                  The output format for query results. Default␣
↪is human-readable plain-text. json is currently the only supported format.
```

(continues on next page)

(continued from previous page)

```
      --peerAddresses stringArray      The addresses of the peers to connect to
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.3.7 peer lifecycle chaincode getinstalledpackage

```
Get an installed chaincode package from a peer.

Usage:
  peer lifecycle chaincode getinstalledpackage [outputfile] [flags]

Flags:
      --connectionProfile string       The fully qualified path to the connection␣
↪profile that provides the necessary connection information for the network. Note:␣
↪currently only supported for providing peer connection information
  -h, --help                           help for getinstalledpackage
      --output-directory string        The output directory to use when writing a␣
↪chaincode install package to disk. Default is the current working directory.
      --package-id string              The identifier of the chaincode install package
      --peerAddresses stringArray      The addresses of the peers to connect to
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

(continues on next page)

(continued from previous page)

11.3.8 peer lifecycle chaincode approveformyorg

```
Approve the chaincode definition for my organization.

Usage:
  peer lifecycle chaincode approveformyorg [flags]

Flags:
      --channel-config-policy string   The endorsement policy associated to this␣
↪chaincode specified as a channel config policy reference
  -C, --channelID string               The channel on which this command should be␣
↪executed
      --collections-config string      The fully qualified path to the collection␣
↪JSON file including the file name
      --connectionProfile string       The fully qualified path to the connection␣
↪profile that provides the necessary connection information for the network. Note:␣
↪currently only supported for providing peer connection information
  -E, --endorsement-plugin string      The name of the endorsement plugin to be used␣
↪for this chaincode
  -h, --help                           help for approveformyorg
      --init-required                  Whether the chaincode requires invoking 'init'
  -n, --name string                    Name of the chaincode
      --package-id string              The identifier of the chaincode install package
      --peerAddresses stringArray      The addresses of the peers to connect to
      --sequence int                   The sequence number of the chaincode␣
↪definition for the channel (default 1)
      --signature-policy string        The endorsement policy associated to this␣
↪chaincode specified as a signature policy
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag
  -V, --validation-plugin string       The name of the validation plugin to be used␣
↪for this chaincode
  -v, --version string                 Version of the chaincode
      --waitForEvent                   Whether to wait for the event from each peer's␣
↪deliver filtered service signifying that the transaction has been committed␣
↪successfully (default true)
      --waitForEventTimeout duration   Time to wait for the event from each peer's␣
↪deliver filtered service signifying that the 'invoke' transaction has been␣
↪committed successfully (default 30s)

Global Flags:
      --cafile string                      Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                    Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                         Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration               Timeout for client to connect (default 3s)
      --keyfile string                     Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                     Ordering service endpoint
      --ordererTLSHostnameOverride string  The hostname override to use when␣
↪validating the TLS connection to the orderer.
```

(continues on next page)

(continued from previous page)

```
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.3.9 peer lifecycle chaincode checkcommitreadiness

```
Check whether a chaincode definition is ready to be committed on a channel.

Usage:
  peer lifecycle chaincode checkcommitreadiness [flags]

Flags:
      --channel-config-policy string   The endorsement policy associated to this␣
↪chaincode specified as a channel config policy reference
  -C, --channelID string               The channel on which this command should be␣
↪executed
      --collections-config string      The fully qualified path to the collection␣
↪JSON file including the file name
      --connectionProfile string       The fully qualified path to the connection␣
↪profile that provides the necessary connection information for the network. Note:␣
↪currently only supported for providing peer connection information
  -E, --endorsement-plugin string      The name of the endorsement plugin to be used␣
↪for this chaincode
  -h, --help                           help for checkcommitreadiness
      --init-required                  Whether the chaincode requires invoking 'init'
  -n, --name string                    Name of the chaincode
  -O, --output string                  The output format for query results. Default␣
↪is human-readable plain-text. json is currently the only supported format.
      --peerAddresses stringArray      The addresses of the peers to connect to
      --sequence int                   The sequence number of the chaincode␣
↪definition for the channel (default 1)
      --signature-policy string        The endorsement policy associated to this␣
↪chaincode specified as a signature policy
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag
  -V, --validation-plugin string       The name of the validation plugin to be used␣
↪for this chaincode
  -v, --version string                 Version of the chaincode

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.3.10 peer lifecycle chaincode commit

```
Commit the chaincode definition on the channel.

Usage:
  peer lifecycle chaincode commit [flags]

Flags:
      --channel-config-policy string   The endorsement policy associated to this␣
↪chaincode specified as a channel config policy reference
  -C, --channelID string               The channel on which this command should be␣
↪executed
      --collections-config string      The fully qualified path to the collection␣
↪JSON file including the file name
      --connectionProfile string       The fully qualified path to the connection␣
↪profile that provides the necessary connection information for the network. Note:␣
↪currently only supported for providing peer connection information
  -E, --endorsement-plugin string      The name of the endorsement plugin to be used␣
↪for this chaincode
  -h, --help                           help for commit
      --init-required                  Whether the chaincode requires invoking 'init'
  -n, --name string                    Name of the chaincode
      --peerAddresses stringArray      The addresses of the peers to connect to
      --sequence int                   The sequence number of the chaincode␣
↪definition for the channel (default 1)
      --signature-policy string        The endorsement policy associated to this␣
↪chaincode specified as a signature policy
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag
  -V, --validation-plugin string       The name of the validation plugin to be used␣
↪for this chaincode
  -v, --version string                 Version of the chaincode
      --waitForEvent                   Whether to wait for the event from each peer's␣
↪deliver filtered service signifying that the transaction has been committed␣
↪successfully (default true)
      --waitForEventTimeout duration   Time to wait for the event from each peer's␣
↪deliver filtered service signifying that the 'invoke' transaction has been␣
↪committed successfully (default 30s)

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.3.11 peer lifecycle chaincode querycommitted

```
Query the committed chaincode definitions by channel on a peer. Optional: provide a␣
↪chaincode name to query a specific definition.

Usage:
  peer lifecycle chaincode querycommitted [flags]

Flags:
  -C, --channelID string               The channel on which this command should be␣
↪executed
      --connectionProfile string       The fully qualified path to the connection␣
↪profile that provides the necessary connection information for the network. Note:␣
↪currently only supported for providing peer connection information
  -h, --help                           help for querycommitted
  -n, --name string                    Name of the chaincode
  -O, --output string                  The output format for query results. Default␣
↪is human-readable plain-text. json is currently the only supported format.
      --peerAddresses stringArray      The addresses of the peers to connect to
      --tlsRootCertFiles stringArray   If TLS is enabled, the paths to the TLS root␣
↪cert files of the peers to connect to. The order and number of certs specified␣
↪should match the --peerAddresses flag

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.3.12 Example Usage

peer lifecycle chaincode package example

A chaincode needs to be packaged before it can be installed on your peers. This example uses the `peer lifecycle chaincode package` command to package a Go chaincode.

- Use the `--label` flag to provide a chaincode package label of `myccv1` that your organization will use to identify the package.

```
peer lifecycle chaincode package mycc.tar.gz --path github.com/hyperledger/fabric-
↪samples/chaincode/abstore/go/ --lang golang --label myccv1
```

peer lifecycle chaincode install example

After the chaincode is packaged, you can use the `peer chaincode install` command to install the chaincode on your peers.

- Install the `mycc.tar.gz` package on `peer0.org1.example.com:7051` (the peer defined by `--peerAddresses`).

```
peer lifecycle chaincode install mycc.tar.gz --peerAddresses peer0.org1.example.
↪com:7051
```

If successful, the command will return the package identifier. The package ID is the package label combined with a hash of the chaincode package taken by the peer.

```
2019-03-13 13:48:53.691 UTC [cli.lifecycle.chaincode] submitInstallProposal ->␣
↪INFO 001 Installed remotely: response:<status:200 payload:
↪"\nEmycc:ebd89878c2bbccf62f68c36072626359376aa83c36435a058d453e8dbfd894cc" >
2019-03-13 13:48:53.691 UTC [cli.lifecycle.chaincode] submitInstallProposal ->␣
↪INFO 002 Chaincode code package identifier:␣
↪mycc:a7ca45a7cc85f1d89c905b775920361ed089a364e12a9b6d55ba75c965ddd6a9
```

peer lifecycle chaincode queryinstalled example

You need to use the chaincode package identifier to approve a chaincode definition for your organization. You can find the package ID for the chaincodes you have installed by using the `peer lifecycle chaincode queryinstalled` command:

```
peer lifecycle chaincode queryinstalled --peerAddresses peer0.org1.example.com:7051
```

A successful command will return the package ID associated with the package label.

```
Get installed chaincodes on peer:
Package ID: myccv1:a7ca45a7cc85f1d89c905b775920361ed089a364e12a9b6d55ba75c965ddd6a9,␣
↪Label: myccv1
```

- You can also use the `--output` flag to have the CLI format the output as JSON.

```
peer lifecycle chaincode queryinstalled --peerAddresses peer0.org1.example.
↪com:7051 --output json
```

If successful, the command will return the chaincodes you have installed as JSON.

```
{
  "installed_chaincodes": [
    {
      "package_id": "mycc_
↪1:aab9981fa5649cfe25369fce7bb5086a69672a631e4f95c4af1b5198fe9f845b",
      "label": "mycc_1",
      "references": {
        "mychannel": {
          "chaincodes": [
            {
              "name": "mycc",
              "version": "1"
            }
          ]
```

(continues on next page)

(continued from previous page)

```
            }
          }
        }
    ]
}
```

peer lifecycle chaincode getinstalledpackage example

You can retrieve an installed chaincode package from a peer using the `peer lifecycle chaincode getinstalledpackage` command. Use the package identifier returned by `queryinstalled`.

- Use the `--package-id` flag to pass in the chaincode package identifier. Use the `--output-directory` flag to specify where to write the chaincode package. If the output directory is not specified, the chaincode package will be written in the current directory.

```
peer lifecycle chaincode getinstalledpackage --package-id␣
↪myccv1:a7ca45a7cc85f1d89c905b775920361ed089a364e12a9b6d55ba75c965ddd6a9 --output-
↪directory /tmp --peerAddresses peer0.org1.example.com:7051
```

peer lifecycle chaincode approveformyorg example

Once the chaincode package has been installed on your peers, you can approve a chaincode definition for your organization. The chaincode definition includes the important parameters of chaincode governance, including the chaincode name, version and the endorsement policy.

Here is an example of the `peer lifecycle chaincode approveformyorg` command, which approves the definition of a chaincode named `mycc` at version `1.0` on channel `mychannel`.

- Use the `--package-id` flag to pass in the chaincode package identifier. Use the `--signature-policy` flag to define an endorsement policy for the chaincode. Use the `init-required` flag to request the execution of the `Init` function to initialize the chaincode.

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode approveformyorg  -o orderer.example.com:7050 --tls --
↪cafile $ORDERER_CA --channelID mychannel --name mycc --version 1.0 --init-
↪required --package-id␣
↪myccv1:a7ca45a7cc85f1d89c905b775920361ed089a364e12a9b6d55ba75c965ddd6a9 --
↪sequence 1 --signature-policy "AND ('Org1MSP.peer','Org2MSP.peer')"

2019-03-18 16:04:09.046 UTC [cli.lifecycle.chaincode] InitCmdFactory -> INFO 001␣
↪Retrieved channel (mychannel) orderer endpoint: orderer.example.com:7050
2019-03-18 16:04:11.253 UTC [chaincodeCmd] ClientWait -> INFO 002 txid␣
↪[efba188ca77889cc1c328fc98e0bb12d3ad0abcda3f84da3714471c7c1e6c13c] committed␣
↪with status (VALID) at peer0.org1.example.com:7051
```

- You can also use the `--channel-config-policy` flag use a policy inside the channel configuration as the chaincode endorsement policy. The default endorsement policy is `Channel/Application/Endorsement`

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode approveformyorg -o orderer.example.com:7050 --tls --
↪cafile $ORDERER_CA --channelID mychannel --name mycc --version 1.0 --init-
↪required --package-id␣
↪myccv1:a7ca45a7cc85f1d89c905b775920361ed089a364e12a9b6d55ba75c965ddd6a9 --
↪sequence 1 --channel-config-policy Channel/Application/Admins

2019-03-18 16:04:09.046 UTC [cli.lifecycle.chaincode] InitCmdFactory -> INFO 001␣
↪Retrieved channel (mychannel) orderer endpoint: orderer.example.com:7050
2019-03-18 16:04:11.253 UTC [chaincodeCmd] ClientWait -> INFO 002 txid␣
↪[efba188ca77889cc1c328fc98e0bb12d3ad0abcda3f84da3714471c7c1e6c13c] committed␣
↪with status (VALID) at peer0.org1.example.com:7051
```

peer lifecycle chaincode checkcommitreadiness example

You can check whether a chaincode definition is ready to be committed using the `peer lifecycle chaincode checkcommitreadiness` command, which will return successfully if a subsequent commit of the definition is expected to succeed. It also outputs which organizations have approved the chaincode definition. If an organization has approved the chaincode definition specified in the command, the command will return a value of true. You can use this command to learn whether enough channel members have approved a chaincode definition to meet the `Application/Channel/Endorsement` policy (a majority by default) before the definition can be committed to a channel.

- Here is an example of the `peer lifecycle chaincode checkcommitreadiness` command, which checks a chaincode named `mycc` at version `1.0` on channel `mychannel`.

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode checkcommitreadiness -o orderer.example.com:7050 --
↪channelID mychannel --tls --cafile $ORDERER_CA --name mycc --version 1.0 --init-
↪required --sequence 1
```

  If successful, the command will return the organizations that have approved the chaincode definition.

```
Chaincode definition for chaincode 'mycc', version '1.0', sequence '1' on channel
'mychannel' approval status by org:
Org1MSP: true
Org2MSP: true
```

- You can also use the `--output` flag to have the CLI format the output as JSON.

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode checkcommitreadiness -o orderer.example.com:7050 --
↪channelID mychannel --tls --cafile $ORDERER_CA --name mycc --version 1.0 --init-
↪required --sequence 1 --output json
```

  If successful, the command will return a JSON map that shows if an organization has approved the chaincode definition.

```
{
   "Approvals": {
      "Org1MSP": true,
      "Org2MSP": true
   }
}
```

peer lifecycle chaincode commit example

Once a sufficient number of organizations approve a chaincode definition for their organizations (a majority by default), one organization can commit the definition the channel using the `peer lifecycle chaincode commit` command:

- This command needs to target the peers of other organizations on the channel to collect their organization endorsement for the definition.

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID mychannel␣
↪--name mycc --version 1.0 --sequence 1 --init-required --tls --cafile $ORDERER_
↪CA --peerAddresses peer0.org1.example.com:7051 --peerAddresses peer0.org2.
↪example.com:9051

2019-03-18 16:14:27.258 UTC [chaincodeCmd] ClientWait -> INFO 001 txid␣
↪[b6f657a14689b27d69a50f39590b3949906b5a426f9d7f0dcee557f775e17882] committed␣
↪with status (VALID) at peer0.org2.example.com:9051
2019-03-18 16:14:27.321 UTC [chaincodeCmd] ClientWait -> INFO 002 txid␣
↪[b6f657a14689b27d69a50f39590b3949906b5a426f9d7f0dcee557f775e17882] committed␣
↪with status (VALID) at peer0.org1.example.com:7051
```

peer lifecycle chaincode querycommitted example

You can query the chaincode definitions that have been committed to a channel by using the `peer lifecycle chaincode querycommitted` command. You can use this command to query the current definition sequence number before upgrading a chaincode.

- You need to supply the chaincode name and channel name in order to query a specific chaincode definition and the organizations that have approved it.

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode querycommitted -o orderer.example.com:7050 --channelID␣
↪mychannel --name mycc --tls --cafile $ORDERER_CA --peerAddresses peer0.org1.
↪example.com:7051

Committed chaincode definition for chaincode 'mycc' on channel 'mychannel':
Version: 1, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc
Approvals: [Org1MSP: true, Org2MSP: true]
```

- You can also specify just the channel name in order to query all chaincode definitions on that channel.

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
↪tlsca.example.com-cert.pem

peer lifecycle chaincode querycommitted -o orderer.example.com:7050 --channelID␣
↪mychannel --tls --cafile $ORDERER_CA --peerAddresses peer0.org1.example.com:7051

Committed chaincode definitions on channel 'mychannel':
Name: mycc, Version: 1, Sequence: 1, Endorsement Plugin: escc, Validation Plugin:␣
↪vscc
Name: yourcc, Version: 2, Sequence: 3, Endorsement Plugin: escc, Validation␣
↪Plugin: vscc
```

- You can also use the `--output` flag to have the CLI format the output as JSON.
  - For querying a specific chaincode definition

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem

peer lifecycle chaincode querycommitted -o orderer.example.com:7050 --
↪channelID mychannel --name mycc --tls --cafile $ORDERER_CA --peerAddresses␣
↪peer0.org1.example.com:7051 --output json
```

If successful, the command will return a JSON that has committed chaincode definition for chaincode 'mycc' on channel 'mychannel'.

```
{
  "sequence": 1,
  "version": "1",
  "endorsement_plugin": "escc",
  "validation_plugin": "vscc",
  "validation_parameter": "EiAvQ2hhbm5lbC9BcHBsaWNhdGlvbi9FbmRvcnNlbWVudA==",
  "collections": {},
  "init_required": true,
  "approvals": {
    "Org1MSP": true,
    "Org2MSP": true
  }
}
```

The `validation_parameter` is base64 encoded. An example of the command to decode it is as follows.

```
echo EiAvQ2hhbm5lbC9BcHBsaWNhdGlvbi9FbmRvcnNlbWVudA== | base64 -d

 /Channel/Application/Endorsement
```

  - For querying all chaincode definitions on that channel

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/
↪tlscacerts/tlsca.example.com-cert.pem

peer lifecycle chaincode querycommitted -o orderer.example.com:7050 --
↪channelID mychannel --tls --cafile $ORDERER_CA --peerAddresses peer0.org1.
↪example.com:7051 --output json
```

If successful, the command will return a JSON that has committed chaincode definitions on channel 'mychannel'.

```
{
  "chaincode_definitions": [
    {
      "name": "mycc",
      "sequence": 1,
      "version": "1",
      "endorsement_plugin": "escc",
      "validation_plugin": "vscc",
      "validation_parameter":
↪"EiAvQ2hhbm5lbC9BcHBsaWNhdGlvbi9FbmRvcnNlbWVudA==",
      "collections": {},
      "init_required": true
    },
    {
      "name": "yourcc",
      "sequence": 3,
      "version": "2",
      "endorsement_plugin": "escc",
      "validation_plugin": "vscc",
      "validation_parameter":
↪"EiAvQ2hhbm5lbC9BcHBsaWNhdGlvbi9FbmRvcnNlbWVudA==",
      "collections": {}
    }
  ]
}
```

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.4 peer channel

The `peer channel` command allows administrators to perform channel related operations on a peer, such as joining a channel or listing the channels to which a peer is joined.

11.4.1 Syntax

The `peer channel` command has the following subcommands:

- create
- fetch
- getinfo
- join
- list
- signconfigtx
- update

11.4.2 peer channel

```
Operate a channel: create|fetch|join|list|update|signconfigtx|getinfo.

Usage:
  peer channel [command]

Available Commands:
  create       Create a channel
  fetch        Fetch a block
  getinfo      get blockchain information of a specified channel.
  join         Joins the peer to a channel.
  list         List of channels peer has joined.
  signconfigtx Signs a configtx update.
  update       Send a configtx update.

Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
  -h, --help                                help for channel
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint

Use "peer channel [command] --help" for more information about a command.
```

11.4.3 peer channel create

```
Create a channel and write the genesis block to a file.

Usage:
  peer channel create [flags]

Flags:
  -c, --channelID string     In case of a newChain command, the channel ID to create.␣
↪It must be all lower case, less than 250 characters long and match the regular␣
↪expression: [a-z][a-z0-9.-]*
  -f, --file string          Configuration transaction file generated by a tool such␣
↪as configtxgen for submitting to orderer
  -h, --help                 help for create
      --outputBlock string   The path to write the genesis block for the channel.␣
↪(default ./<channelID>.block)
  -t, --timeout duration     Channel creation timeout (default 10s)

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
```

(continues on next page)

(continued from previous page)

```
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.4.4 peer channel fetch

```
Fetch a specified block, writing it to a file.

Usage:
  peer channel fetch <newest|oldest|config|(number)> [outputfile] [flags]

Flags:
      --bestEffort         Whether fetch requests should ignore errors and return␣
↪blocks on a best effort basis
  -c, --channelID string   In case of a newChain command, the channel ID to create.␣
↪It must be all lower case, less than 250 characters long and match the regular␣
↪expression: [a-z][a-z0-9.-]*
  -h, --help               help for fetch

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.4.5 peer channel getinfo

```
get blockchain information of a specified channel. Requires '-c'.

Usage:
  peer channel getinfo [flags]

Flags:
```

(continues on next page)

(continued from previous page)

```
  -c, --channelID string   In case of a newChain command, the channel ID to create.␣
↪It must be all lower case, less than 250 characters long and match the regular␣
↪expression: [a-z][a-z0-9.-]*
  -h, --help               help for getinfo

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.4.6 peer channel join

```
Joins the peer to a channel.

Usage:
  peer channel join [flags]

Flags:
  -b, --blockpath string   Path to file containing genesis block
  -h, --help               help for join

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.4.7 peer channel list

```
List of channels peer has joined.
```

(continues on next page)

(continued from previous page)

```
Usage:
  peer channel list [flags]

Flags:
  -h, --help   help for list

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.4.8 peer channel signconfigtx

```
Signs the supplied configtx update file in place on the filesystem. Requires '-f'.

Usage:
  peer channel signconfigtx [flags]

Flags:
  -f, --file string   Configuration transaction file generated by a tool such as␣
↪configtxgen for submitting to orderer
  -h, --help          help for signconfigtx

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.4.9 peer channel update

```
Signs and sends the supplied configtx update file to the channel. Requires '-f', '-o',
↪ '-c'.

Usage:
  peer channel update [flags]

Flags:
  -c, --channelID string   In case of a newChain command, the channel ID to create.␣
↪It must be all lower case, less than 250 characters long and match the regular␣
↪expression: [a-z][a-z0-9.-]*
  -f, --file string        Configuration transaction file generated by a tool such as␣
↪configtxgen for submitting to orderer
  -h, --help               help for update

Global Flags:
      --cafile string                       Path to file containing PEM-encoded␣
↪trusted certificate(s) for the ordering endpoint
      --certfile string                     Path to file containing PEM-encoded X509␣
↪public key to use for mutual TLS communication with the orderer endpoint
      --clientauth                          Use mutual TLS when communicating with␣
↪the orderer endpoint
      --connTimeout duration                Timeout for client to connect (default 3s)
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
  -o, --orderer string                      Ordering service endpoint
      --ordererTLSHostnameOverride string   The hostname override to use when␣
↪validating the TLS connection to the orderer.
      --tls                                 Use TLS when communicating with the␣
↪orderer endpoint
```

11.4.10 Example Usage

peer channel create examples

Here's an example that uses the `--orderer` global flag on the `peer channel create` command.

- Create a sample channel `mychannel` defined by the configuration transaction contained in file `./createchannel.tx`. Use the orderer at `orderer.example.com:7050`.

```
peer channel create -c mychannel -f ./createchannel.tx --orderer orderer.example.
↪com:7050

2018-02-25 08:23:57.548 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and␣
↪orderer connections initialized
2018-02-25 08:23:57.626 UTC [channelCmd] InitCmdFactory -> INFO 019 Endorser and␣
↪orderer connections initialized
2018-02-25 08:23:57.834 UTC [channelCmd] readBlock -> INFO 020 Received block: 0
2018-02-25 08:23:57.835 UTC [main] main -> INFO 021 Exiting.....
```

  Block 0 is returned indicating that the channel has been successfully created.

Here's an example of the `peer channel create` command option.

- Create a new channel `mychannel` for the network, using the orderer at ip address `orderer.example.com:7050`. The configuration update transaction required to create this channel is defined the file `./`

`createchannel.tx`. Wait 30 seconds for the channel to be created.

```
peer channel create -c mychannel --orderer orderer.example.com:7050 -f ./
↪createchannel.tx -t 30s

2018-02-23 06:31:58.568 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser␣
↪and orderer connections initialized
2018-02-23 06:31:58.669 UTC [channelCmd] InitCmdFactory -> INFO 019 Endorser␣
↪and orderer connections initialized
2018-02-23 06:31:58.877 UTC [channelCmd] readBlock -> INFO 020 Received block: 0
2018-02-23 06:31:58.878 UTC [main] main -> INFO 021 Exiting.....

ls -l

-rw-r--r-- 1 root root 11982 Feb 25 12:24 mychannel.block
```

You can see that channel `mychannel` has been successfully created, as indicated in the output where block 0 (zero) is added to the blockchain for this channel and returned to the peer, where it is stored in the local directory as `mychannel.block`.

Block zero is often called the *genesis block* as it provides the starting configuration for the channel. All subsequent updates to the channel will be captured as configuration blocks on the channel's blockchain, each of which supersedes the previous configuration.

peer channel fetch example

Here's some examples of the `peer channel fetch` command.

- Using the `newest` option to retrieve the most recent channel block, and store it in the file `mychannel.block`.

```
peer channel fetch newest mychannel.block -c mychannel --orderer orderer.example.
↪com:7050

2018-02-25 13:10:16.137 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and␣
↪orderer connections initialized
2018-02-25 13:10:16.144 UTC [channelCmd] readBlock -> INFO 00a Received block: 32
2018-02-25 13:10:16.145 UTC [main] main -> INFO 00b Exiting.....

ls -l

-rw-r--r-- 1 root root 11982 Feb 25 13:10 mychannel.block
```

You can see that the retrieved block is number 32, and that the information has been written to the file `mychannel.block`.

- Using the `(block number)` option to retrieve a specific block – in this case, block number 16 – and store it in the default block file.

```
peer channel fetch 16  -c mychannel --orderer orderer.example.com:7050

2018-02-25 13:46:50.296 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and␣
↪orderer connections initialized
2018-02-25 13:46:50.302 UTC [channelCmd] readBlock -> INFO 00a Received block: 16
2018-02-25 13:46:50.302 UTC [main] main -> INFO 00b Exiting.....

ls -l
```

(continues on next page)

(continued from previous page)

```
-rw-r--r-- 1 root root 11982 Feb 25 13:10 mychannel.block
-rw-r--r-- 1 root root  4783 Feb 25 13:46 mychannel_16.block
```

You can see that the retrieved block is number 16, and that the information has been written to the default file `mychannel_16.block`.

For configuration blocks, the block file can be decoded using the `configtxlator` command. See this command for an example of decoded output. User transaction blocks can also be decoded, but a user program must be written to do this.

peer channel getinfo example

Here's an example of the `peer channel getinfo` command.

- Get information about the local peer for channel `mychannel`.

```
peer channel getinfo -c mychannel

2018-02-25 15:15:44.135 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and␣
↪orderer connections initialized
Blockchain info: {"height":5,"currentBlockHash":"JgK9lcaPUNmFb5Mp1qe1SVMsx3o/
↪22Ct4+n5tejcXCw=","previousBlockHash":
↪"f8lZXoAn3gF86zrFq7L1DzW2aKuabH9Ow6SIE5Y04a4="}
2018-02-25 15:15:44.139 UTC [main] main -> INFO 006 Exiting.....
```

You can see that the latest block for channel `mychannel` is block 5. You can also see the cryptographic hashes for the most recent blocks in the channel's blockchain.

peer channel join example

Here's an example of the `peer channel join` command.

- Join a peer to the channel defined in the genesis block identified by the file `./mychannel.genesis.block`. In this example, the channel block was previously retrieved by the `peer channel fetch` command.

```
peer channel join -b ./mychannel.genesis.block

2018-02-25 12:25:26.511 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and␣
↪orderer connections initialized
2018-02-25 12:25:26.571 UTC [channelCmd] executeJoin -> INFO 006 Successfully␣
↪submitted proposal to join channel
2018-02-25 12:25:26.571 UTC [main] main -> INFO 007 Exiting.....
```

You can see that the peer has successfully made a request to join the channel.

peer channel list example

Here's an example of the `peer channel list` command.

- List the channels to which a peer is joined.

```
peer channel list

2018-02-25 14:21:20.361 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and␣
↪orderer connections initialized
```

(continues on next page)

(continued from previous page)

```
Channels peers has joined:
mychannel
2018-02-25 14:21:20.372 UTC [main] main -> INFO 006 Exiting.....
```

You can see that the peer is joined to channel `mychannel`.

peer channel signconfigtx example

Here's an example of the `peer channel signconfigtx` command.

- Sign the `channel update` transaction defined in the file `./updatechannel.tx`. The example lists the configuration transaction file before and after the command.

```
ls -l

-rw-r--r--  1 anthonyodowd  staff   284 25 Feb 18:16 updatechannel.tx

peer channel signconfigtx -f updatechannel.tx

2018-02-25 18:16:44.456 GMT [channelCmd] InitCmdFactory -> INFO 001 Endorser and
↪orderer connections initialized
2018-02-25 18:16:44.459 GMT [main] main -> INFO 002 Exiting.....

ls -l

-rw-r--r--  1 anthonyodowd  staff  2180 25 Feb 18:16 updatechannel.tx
```

You can see that the peer has successfully signed the configuration transaction by the increase in the size of the file `updatechannel.tx` from 284 bytes to 2180 bytes.

peer channel update example

Here's an example of the `peer channel update` command.

- Update the channel `mychannel` using the configuration transaction defined in the file `./updatechannel.tx`. Use the orderer at ip address `orderer.example.com:7050` to send the configuration transaction to all peers in the channel to update their copy of the channel configuration.

```
peer channel update -c mychannel -f ./updatechannel.tx -o orderer.example.com:7050

2018-02-23 06:32:11.569 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and
↪orderer connections initialized
2018-02-23 06:32:11.626 UTC [main] main -> INFO 010 Exiting.....
```

At this point, the channel `mychannel` has been successfully updated.

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.5 peer version

The `peer version` command displays the version information of the peer. It displays version, Commit SHA, Go version, OS/architecture, and chaincode information. For example:

```
 peer:
   Version: 2.1.0
   Commit SHA: b78d79b
   Go version: go1.14.1
   OS/Arch: linux/amd64
   Chaincode:
    Base Docker Label: org.hyperledger.fabric
    Docker Namespace: hyperledger
```

11.5.1 Syntax

The `peer version` command takes no arguments.

11.5.2 peer version

```
Print current version of the fabric peer server.

Usage:
  peer version [flags]

Flags:
  -h, --help   help for version
```

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.6 peer node

The `peer node` command allows an administrator to start a peer node, reset all channels in a peer to the genesis block, or rollback a channel to a given block number.

11.6.1 Syntax

The `peer node` command has the following subcommands:

- start
- reset
- rollback

11.6.2 peer node start

```
Starts a node that interacts with the network.

Usage:
  peer node start [flags]

Flags:
  -h, --help                help for start
      --peer-chaincodedev   start peer in chaincode development mode
```

11.6.3 peer node reset

```
Resets all channels to the genesis block. When the command is executed, the peer must␣
↪be offline. When the peer starts after the reset, it will receive blocks starting␣
↪with block number one from an orderer or another peer to rebuild the block store␣
↪and state database.

Usage:
  peer node reset [flags]

Flags:
  -h, --help   help for reset
```

11.6.4 peer node rollback

```
Rolls back a channel to a specified block number. When the command is executed, the␣
↪peer must be offline. When the peer starts after the rollback, it will receive␣
↪blocks, which got removed during the rollback, from an orderer or another peer to␣
↪rebuild the block store and state database.

Usage:
  peer node rollback [flags]

Flags:
  -b, --blockNumber uint   Block number to which the channel needs to be rolled back␣
↪to.
  -c, --channelID string   Channel to rollback.
  -h, --help               help for rollback
```

11.6.5 Example Usage

peer node start example

The following command:

```
peer node start --peer-chaincodedev
```

starts a peer node in chaincode development mode. Normally chaincode containers are started and maintained by peer. However in chaincode development mode, chaincode is built and started by the user. This mode is useful during chaincode development phase for iterative development. See more information on development mode in the chaincode tutorial.

peer node reset example

```
peer node reset
```

resets all channels in the peer to the genesis block, i.e., the first block in the channel. The command also records the pre-reset height of each channel in the file system. Note that the peer process should be stopped while executing this command. If the peer process is running, this command detects that and returns an error instead of performing the reset. When the peer is started after performing the reset, the peer will fetch the blocks for each channel which were removed by the reset command (either from other peers or orderers) and commit the blocks up to the pre-reset height. Until all channels reach the pre-reset height, the peer will not endorse any transactions.

peer node rollback example

The following command:

```
peer node rollback -c ch1 -b 150
```

rolls back the channel ch1 to block number 150. The command also records the pre-rolled back height of channel ch1 in the file system. Note that the peer should be stopped while executing this command. If the peer process is running, this command detects that and returns an error instead of performing the rollback. When the peer is started after performing the rollback, the peer will fetch the blocks for channel ch1 which were removed by the rollback command (either from other peers or orderers) and commit the blocks up to the pre-rolled back height. Until the channel ch1 reaches the pre-rolled back height, the peer will not endorse any transaction for any channel.

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.7 configtxgen

The `configtxgen` command allows users to create and inspect channel config related artifacts. The content of the generated artifacts is dictated by the contents of `configtx.yaml`.

11.7.1 Syntax

The `configtxgen` tool has no sub-commands, but supports flags which can be set to accomplish a number of tasks.

11.7.2 configtxgen

```
Usage of configtxgen:
  -asOrg string
        Performs the config generation as a particular organization (by name), only␣
↪including values in the write set that org (likely) has privilege to set
  -channelCreateTxBaseProfile string
        Specifies a profile to consider as the orderer system channel current state␣
↪to allow modification of non-application parameters during channel create tx␣
↪generation. Only valid in conjunction with 'outputCreateChannelTx'.
  -channelID string
        The channel ID to use in the configtx
  -configPath string
        The path containing the configuration to use (if set)
  -inspectBlock string
        Prints the configuration contained in the block at the specified path
  -inspectChannelCreateTx string
        Prints the configuration contained in the transaction at the specified path
  -outputAnchorPeersUpdate string
        [DEPRECATED] Creates a config update to update an anchor peer (works only␣
↪with the default channel creation, and only for the first update)
  -outputBlock string
        The path to write the genesis block to (if set)
  -outputCreateChannelTx string
        The path to write a channel creation configtx to (if set)
  -printOrg string
        Prints the definition of an organization as JSON. (useful for adding an org␣
↪to a channel manually)
```

(continues on next page)

(continued from previous page)

```
  -profile string
        The profile from configtx.yaml to use for generation.
  -version
        Show version information
```

11.7.3 Usage

Output a genesis block

Write a genesis block to `genesis_block.pb` for channel `orderer-system-channel` for profile `SampleSingleMSPRaftV1_1`.

```
configtxgen -outputBlock genesis_block.pb -profile SampleSingleMSPRaftV1_1 -channelID␣
↪orderer-system-channel
```

Output a channel creation tx

Write a channel creation transaction to `create_chan_tx.pb` for profile `SampleSingleMSPChannelV1_1`.

```
configtxgen -outputCreateChannelTx create_chan_tx.pb -profile␣
↪SampleSingleMSPChannelV1_1 -channelID application-channel-1
```

Inspect a genesis block

Print the contents of a genesis block named `genesis_block.pb` to the screen as JSON.

```
configtxgen -inspectBlock genesis_block.pb
```

Inspect a channel creation tx

Print the contents of a channel creation tx named `create_chan_tx.pb` to the screen as JSON.

```
configtxgen -inspectChannelCreateTx create_chan_tx.pb
```

Print an organization definition

Construct an organization definition based on the parameters such as MSPDir from `configtx.yaml` and print it as JSON to the screen. (This output is useful for channel reconfiguration workflows, such as adding a member).

```
configtxgen -printOrg Org1
```

Output anchor peer tx

Output a configuration update transaction to `anchor_peer_tx.pb` which sets the anchor peers for organization Org1 as defined in profile SampleSingleMSPChannelV1_1 based on `configtx.yaml`.

```
configtxgen -outputAnchorPeersUpdate anchor_peer_tx.pb -profile␣
↪SampleSingleMSPChannelV1_1 -asOrg Org1
```

11.7.4 Configuration

The `configtxgen` tool's output is largely controlled by the content of `configtx.yaml`. This file is searched for at `FABRIC_CFG_PATH` and must be present for `configtxgen` to operate.

Refer to the sample `configtx.yaml` shipped with Fabric for all possible configuration options. You may find this file in the `config` directory of the release artifacts tar, or you may find it under the `sampleconfig` folder if you are building from source.

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.8 configtxlator

The `configtxlator` command allows users to translate between protobuf and JSON versions of fabric data structures and create config updates. The command may either start a REST server to expose its functions over HTTP or may be utilized directly as a command line tool.

11.8.1 Syntax

The `configtxlator` tool has five sub-commands, as follows:

- start
- proto_encode
- proto_decode
- compute_update
- version

11.8.2 configtxlator start

```
usage: configtxlator start [<flags>]

Start the configtxlator REST server

Flags:
  --help                Show context-sensitive help (also try --help-long and
                        --help-man).
  --hostname="0.0.0.0"  The hostname or IP on which the REST server will listen
  --port=7059           The port on which the REST server will listen
  --CORS=CORS ...       Allowable CORS domains, e.g. '*' or 'www.example.com'
                        (may be repeated).
```

11.8.3 configtxlator proto_encode

```
usage: configtxlator proto_encode --type=TYPE [<flags>]

Converts a JSON document to protobuf.

Flags:
  --help                Show context-sensitive help (also try --help-long and
                        --help-man).
  --type=TYPE           The type of protobuf structure to encode to. For
                        example, 'common.Config'.
  --input=/dev/stdin    A file containing the JSON document.
  --output=/dev/stdout  A file to write the output to.
```

11.8.4 configtxlator proto_decode

```
usage: configtxlator proto_decode --type=TYPE [<flags>]

Converts a proto message to JSON.

Flags:
  --help                Show context-sensitive help (also try --help-long and
                        --help-man).
  --type=TYPE           The type of protobuf structure to decode from. For
                        example, 'common.Config'.
  --input=/dev/stdin    A file containing the proto message.
  --output=/dev/stdout  A file to write the JSON document to.
```

11.8.5 configtxlator compute_update

```
usage: configtxlator compute_update --channel_id=CHANNEL_ID [<flags>]

Takes two marshaled common.Config messages and computes the config update which
transitions between the two.

Flags:
  --help                   Show context-sensitive help (also try --help-long and
                           --help-man).
  --original=ORIGINAL      The original config message.
  --updated=UPDATED        The updated config message.
  --channel_id=CHANNEL_ID  The name of the channel for this update.
  --output=/dev/stdout     A file to write the JSON document to.
```

11.8.6 configtxlator version

```
usage: configtxlator version

Show version information

Flags:
  --help  Show context-sensitive help (also try --help-long and --help-man).
```

11.8.7 Examples

Decoding

Decode a block named `fabric_block.pb` to JSON and print to stdout.

```
configtxlator proto_decode --input fabric_block.pb --type common.Block
```

Alternatively, after starting the REST server, the following curl command performs the same operation through the REST API.

```
curl -X POST --data-binary @fabric_block.pb "${CONFIGTXLATOR_URL}/protolator/decode/
↪common.Block"
```

Encoding

Convert a JSON document for a policy from stdin to a file named `policy.pb`.

```
configtxlator proto_encode --type common.Policy --output policy.pb
```

Alternatively, after starting the REST server, the following curl command performs the same operation through the REST API.

```
curl -X POST --data-binary /dev/stdin "${CONFIGTXLATOR_URL}/protolator/encode/common.
↪Policy" > policy.pb
```

Pipelines

Compute a config update from `original_config.pb` and `modified_config.pb` and decode it to JSON to stdout.

```
configtxlator compute_update --channel_id testchan --original original_config.pb --
↪updated modified_config.pb | configtxlator proto_decode --type common.ConfigUpdate
```

Alternatively, after starting the REST server, the following curl commands perform the same operations through the REST API.

```
curl -X POST -F channel=testchan -F "original=@original_config.pb" -F
↪"updated=@modified_config.pb" "${CONFIGTXLATOR_URL}/configtxlator/compute/update-
↪from-configs" | curl -X POST --data-binary /dev/stdin "${CONFIGTXLATOR_URL}/
↪protolator/encode/common.ConfigUpdate"
```

11.8.8 Additional Notes

The tool name is a portmanteau of *configtx* and *translator* and is intended to convey that the tool simply converts between different equivalent data representations. It does not generate configuration. It does not submit or retrieve configuration. It does not modify configuration itself, it simply provides some bijective operations between different views of the configtx format.

There is no configuration file `configtxlator` nor any authentication or authorization facilities included for the REST server. Because `configtxlator` does not have any access to data, key material, or other information which might be considered sensitive, there is no risk to the owner of the server in exposing it to other clients. However,

because the data sent by a user to the REST server might be confidential, the user should either trust the administrator of the server, run a local instance, or operate via the CLI.

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.9 cryptogen

`cryptogen` is an utility for generating Hyperledger Fabric key material. It is provided as a means of preconfiguring a network for testing purposes. It would normally not be used in the operation of a production network.

11.9.1 Syntax

The `cryptogen` command has five subcommands, as follows:

- help
- generate
- showtemplate
- extend
- version

11.9.2 cryptogen help

```
usage: cryptogen [<flags>] <command> [<args> ...]

Utility for generating Hyperledger Fabric key material

Flags:
  --help  Show context-sensitive help (also try --help-long and --help-man).

Commands:
  help [<command>...]
    Show help.

  generate [<flags>]
    Generate key material

  showtemplate
    Show the default configuration template

  version
    Show version information

  extend [<flags>]
    Extend existing network
```

11.9.3 cryptogen generate

```
usage: cryptogen generate [<flags>]

Generate key material

Flags:
  --help                    Show context-sensitive help (also try --help-long
                            and --help-man).
  --output="crypto-config"  The output directory in which to place artifacts
  --config=CONFIG           The configuration template to use
```

11.9.4 cryptogen showtemplate

```
usage: cryptogen showtemplate

Show the default configuration template

Flags:
  --help  Show context-sensitive help (also try --help-long and --help-man).
```

11.9.5 cryptogen extend

```
usage: cryptogen extend [<flags>]

Extend existing network

Flags:
  --help                   Show context-sensitive help (also try --help-long and
                           --help-man).
  --input="crypto-config"  The input directory in which existing network place
  --config=CONFIG          The configuration template to use
```

11.9.6 cryptogen version

```
usage: cryptogen version

Show version information

Flags:
  --help  Show context-sensitive help (also try --help-long and --help-man).
```

11.9.7 Usage

Here's an example using the different available flags on the `cryptogen extend` command.

```
    cryptogen extend --input="crypto-config" --config=config.yaml

    org3.example.com
```

Where config.yaml adds a new peer organization called `org3.example.com`

This work is licensed under a Creative Commons Attribution 4.0 International License.

11.10 Service Discovery CLI

The discovery service has its own Command Line Interface (CLI) which uses a YAML configuration file to persist properties such as certificate and private key paths, as well as MSP ID.

The `discover` command has the following subcommands:

- saveConfig
- peers
- config
- endorsers

And the usage of the command is shown below:

```
usage: discover [<flags>] <command> [<args> ...]

Command line client for fabric discovery service

Flags:
  --help                    Show context-sensitive help (also try --help-long and --
↪help-man).
  --configFile=CONFIGFILE   Specifies the config file to load the configuration from
  --peerTLSCA=PEERTLSCA     Sets the TLS CA certificate file path that verifies the␣
↪TLS peer's certificate
  --tlsCert=TLSCERT         (Optional) Sets the client TLS certificate file path that␣
↪is used when the peer enforces client authentication
  --tlsKey=TLSKEY           (Optional) Sets the client TLS key file path that is used␣
↪when the peer enforces client authentication
  --userKey=USERKEY         Sets the user's key file path that is used to sign␣
↪messages sent to the peer
  --userCert=USERCERT       Sets the user's certificate file path that is used to␣
↪authenticate the messages sent to the peer
  --MSP=MSP                 Sets the MSP ID of the user, which represents the CA(s)␣
↪that issued its user certificate

Commands:
  help [<command>...]
    Show help.

  peers [<flags>]
    Discover peers

  config [<flags>]
    Discover channel config

  endorsers [<flags>]
    Discover chaincode endorsers

  saveConfig
    Save the config passed by flags into the file specified by --configFile
```

11.10.1 Configuring external endpoints

Currently, to see peers in service discovery they need to have `EXTERNAL_ENDPOINT` to be configured for them. Otherwise, Fabric assumes the peer should not be disclosed.

To define these endpoints, you need to specify them in the `core.yaml` of the peer, replacing the sample endpoint below with the ones of your peer.

```
CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer1.org1.example.com:8051
```

11.10.2 Persisting configuration

To persist the configuration, a config file name should be supplied via the flag `--configFile`, along with the command `saveConfig`:

```
discover --configFile conf.yaml --peerTLSCA tls/ca.crt --userKey msp/keystore/
↪ea4f6a38ac7057b6fa9502c2f5f39f182e320f71f667749100fe7dd94c23ce43_sk --userCert msp/
↪signcerts/User1\@org1.example.com-cert.pem  --MSP Org1MSP saveConfig
```

By executing the above command, configuration file would be created:

```
$ cat conf.yaml
version: 0
tlsconfig:
  certpath: ""
  keypath: ""
  peercacertpath: /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/users/User1@org1.example.com/tls/ca.crt
  timeout: 0s
signerconfig:
  mspid: Org1MSP
  identitypath: /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/users/User1@org1.example.com/msp/signcerts/
↪User1@org1.example.com-cert.pem
  keypath: /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪peerOrganizations/org1.example.com/users/User1@org1.example.com/msp/keystore/
↪ea4f6a38ac7057b6fa9502c2f5f39f182e320f71f667749100fe7dd94c23ce43_sk
```

When the peer runs with TLS enabled, the discovery service on the peer requires the client to connect to it with mutual TLS, which means it needs to supply a TLS certificate. The peer is configured by default to request (but not to verify) client TLS certificates, so supplying a TLS certificate isn't needed (unless the peer's `tls.clientAuthRequired` is set to `true`).

When the discovery CLI's config file has a certificate path for `peercacertpath`, but the `certpath` and `keypath` aren't configured as in the above - the discovery CLI generates a self-signed TLS certificate and uses this to connect to the peer.

When the `peercacertpath` isn't configured, the discovery CLI connects without TLS , and this is highly not recommended, as the information is sent over plaintext, un-encrypted.

11.10.3 Querying the discovery service

The discoveryCLI acts as a discovery client, and it needs to be executed against a peer. This is done via specifying the `--server` flag. In addition, the queries are channel-scoped, so the `--channel` flag must be used.

The only query that doesn't require a channel is the local membership peer query, which by default can only be used by administrators of the peer being queried.

The discover CLI supports all server-side queries:

- Peer membership query

- Configuration query
- Endorsers query

Let's go over them and see how they should be invoked and parsed:

11.10.4 Peer membership query:

```
$ discover --configFile conf.yaml peers --channel mychannel  --server peer0.org1.
↪example.com:7051
[
    {
        "MSPID": "Org2MSP",
        "LedgerHeight": 5,
        "Endpoint": "peer0.org2.example.com:9051",
        "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKTCCAc+gAwIBAgIRANK4WBck5gKuzTxVQIwhYMUwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkN
↪ecJNvdAV2zmSx5Sf2qospVAH1MYCHyudDEvkiRuBPgmCdOdwJsE0g+h\nz0nZdKq6/
↪X+jTTBLMA4GA1UdDwEB/
↪wQEAwIHgDAMBgNVHRMBAf8EAjAAMCsGA1Ud\nIwQkMCKAIFZMuZfUtY6n2iyxaVr3rl+x5lU0CdG9x7KAeYydQGTMMAoGCCqGSN
↪LJ7j3I9NEPQ/B1BpnJP+UNPnGO2peVrM/
↪mJ1nVgIgS1ZA\nA1tsxuDyllaQuHx2P+P9NDFdjXx5T08lZhxuWYM=\n-----END CERTIFICATE-----\n
↪",
        "Chaincodes": [
            "mycc"
        ]
    },
    {
        "MSPID": "Org2MSP",
        "LedgerHeight": 5,
        "Endpoint": "peer1.org2.example.com:10051",
        "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKDCCAc+gAwIBAgIRALnNJzplCrYy4Y8CjZtqL7AwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkN
↪YMn1hS6sM+bFDgkJKalG7s9Hg3URF0aGpy51R\nU+4F9Muo+XajTTBLMA4GA1UdDwEB/
↪wQEAwIHgDAMBgNVHRMBAf8EAjAAMCsGA1Ud\nIwQkMCKAIFZMuZfUtY6n2iyxaVr3rl+x5lU0CdG9x7KAeYydQGTMMAoGCCqGSN
↪ExunQ==\n-----END CERTIFICATE-----\n",
        "Chaincodes": [
            "mycc"
        ]
    },
    {
        "MSPID": "Org1MSP",
        "LedgerHeight": 5,
        "Endpoint": "peer0.org1.example.com:7051",
        "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKDCCAc6gAwIBAgIQP18LeXtEXGoN8pTqzXTHZTAKBggqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMBEGA1UECBMKQ2F
↪1Rg/ynSk\nNNItaMlaCDZOaQvxJEl6o3fqx1PVFlfXE4NarY3OO1N3YZI41hWWoXksSwJu/
↪35S\nM7wMEzw+3KNNMEswDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/
↪wQCMAAwKwYDVR0j\nBCQwIoAgcecTOxTes6rfgyxHH6KIW7hsRAw2bhP9ikCHkvtv/
↪RcwCgYIKoZIzj0E\nAwIDSAAwRQIhAKiJEv79XBmr8gGY6kHrGL0L3sq95E7IsCYzYdAQHj+DAiBPcBTg\nRuA0/
↪/Kq+3aHJ2T0KpKHqD3FfhZZolKDkcrkwQ==\n-----END CERTIFICATE-----\n",
        "Chaincodes": [
            "mycc"
        ]
    },
    {
        "MSPID": "Org1MSP",
```

(continues on next page)

(continued from previous page)

```
        "LedgerHeight": 5,
        "Endpoint": "peer1.org1.example.com:8051",
        "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICJzCCAc6gAwIBAgIQO7zMEHlMfRhnP6Xt65jwtDAKBggqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMBEGA1UECBMKQ2F
↪Q2g\nRHw5rk3SYw+OMFw9jNbsJJyC5ttJRvc12Dn7lQ8ZR9hW1vLQ3NtqO/
↪couccDJcHg\nt47iHBNadaNNMEswDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/
↪wQCMAAwKwYDVR0j\nBCQwIoAgcecTOxTes6rfgyxHH6KIW7hsRAw2bhP9ikCHkvtv/
↪RcwCgYIKoZIzj0E\nAwIDRwAwRAIgGHGtRVxcFVeMQr9yRlebs23OXEECNo6hNqd/
↪4ChLwwoCIBFKFd6t\nlL5BVzVMGQyXWcZGrjFgl4+fDrwjmMe+jAfa\n-----END CERTIFICATE-----\n
↪",
        "Chaincodes": null
    }
]
```

As seen, this command outputs a JSON containing membership information about all the peers in the channel that the peer queried possesses.

The `Identity` that is returned is the enrollment certificate of the peer, and it can be parsed with a combination of `jq` and `openssl`:

```
$ discover --configFile conf.yaml peers --channel mychannel  --server peer0.org1.
↪example.com:7051  | jq .[0].Identity | sed "s/\\\n/\n/g" | sed "s/\"//g"  | openssl␣
↪x509 -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            55:e9:3f:97:94:d5:74:db:e2:d6:99:3c:01:24:be:bf
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, ST=California, L=San Francisco, O=org2.example.com, CN=ca.org2.
↪example.com
        Validity
            Not Before: Jun  9 11:58:28 2018 GMT
            Not After : Jun  6 11:58:28 2028 GMT
        Subject: C=US, ST=California, L=San Francisco, OU=peer, CN=peer0.org2.example.
↪com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:f5:69:7a:11:65:d9:85:96:65:b7:b7:1b:08:77:
                    43:de:cb:ad:3a:79:ec:cc:2a:bc:d7:93:68:ae:92:
                    1c:4b:d8:32:47:d6:3d:72:32:f1:f1:fb:26:e4:69:
                    c2:eb:c9:45:69:99:78:d7:68:a9:77:09:88:c6:53:
                    01:2a:c1:f8:c0
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Authority Key Identifier:
                ␣
↪keyid:8E:58:82:C9:0A:11:10:A9:0B:93:03:EE:A0:54:42:F4:A3:EF:11:4C:82:B6:F9:CE:10:A2:1E:24:AB:13:82

    Signature Algorithm: ecdsa-with-SHA256
```

(continues on next page)

(continued from previous page)

```
        30:44:02:20:29:3f:55:2b:9f:7b:99:b2:cb:06:ca:15:3f:93:
        a1:3d:65:5c:7b:79:a1:7a:d1:94:50:f0:cd:db:ea:61:81:7a:
        02:20:3b:40:5b:60:51:3c:f8:0f:9b:fc:ae:fc:21:fd:c8:36:
        a3:18:39:58:20:72:3d:1a:43:74:30:f3:56:01:aa:26
```

11.10.5 Configuration query:

The configuration query returns a mapping from MSP IDs to orderer endpoints, as well as the `FabricMSPConfig` which can be used to verify all peer and orderer nodes by the SDK:

```
$ discover --configFile conf.yaml config --channel mychannel  --server peer0.org1.
↪example.com:7051
{
    "msps": {
        "OrdererOrg": {
            "name": "OrdererMSP",
            "root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNMekNDQWRhZ0F3SUJBZ0lSQU1pWkxUb3RmMHR6VTRzNUdIdkQ0UjR3Q2
↪"
            ],
            "admins": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDVENDQWJDZ0F3SUJBZ0lRR2wzTjhaSzRDekRRQmZqYVpwMVF5VEFLQn
↪"
            ],
            "crypto_config": {
                "signature_hash_family": "SHA2",
                "identity_identifier_hash_function": "SHA256"
            },
            "tls_root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNORENDQWR1Z0F3SUJBZ0lRZDdodzFIaHNZTXI2a25ETWJrZThTakFLQn
↪"
            ]
        },
        "Org1MSP": {
            "name": "Org1MSP",
            "root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ0lSQU1nN2VETnhwS0t0ZGl0TDRVNDRZMUl3Q2
↪"
            ],
            "admins": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNLakNDQWRDZ0F3SUJBZ0lRRTRFK0tqSHgwdTlzRSsxZUgrL1dOakFLQn
↪"
            ],
            "crypto_config": {
                "signature_hash_family": "SHA2",
                "identity_identifier_hash_function": "SHA256"
            },
            "tls_root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNTVENDQWUrZ0F3SUJBZ0lRZlRWTE9iTENVUjdxVEY3Z283UXgvakFLQn
↪"
```

(continues on next page)

(continued from previous page)

```
            ],
            "fabric_node_ous": {
                "enable": true,
                "client_ou_identifier": {
                    "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ0lSQU1nN2VETnhwS0t0ZGl0TDRVNDRZMUl3Q2
↪",
                    "organizational_unit_identifier": "client"
                },
                "peer_ou_identifier": {
                    "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ0lSQU1nN2VETnhwS0t0ZGl0TDRVNDRZMUl3Q2
↪",
                    "organizational_unit_identifier": "peer"
                }
            }
        },
        "Org2MSP": {
            "name": "Org2MSP",
            "root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ0lSQUx2SWV2KzE4Vm9LZFR2V1RLNCtaZ2d3Q2
↪"
            ],
            "admins": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNLVENDQWRDZ0F3SUJBZ0lRU1lpeE1vdmpoM1N2c25WMmFUOXl1REFLQn
↪"
            ],
            "crypto_config": {
                "signature_hash_family": "SHA2",
                "identity_identifier_hash_function": "SHA256"
            },
            "tls_root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNTakNDQWZDZ0F3SUJBZ0lSQUtoUFFxUGZSYnVpSktqL0JRanQ3RXN3Q2
↪"
            ],
            "fabric_node_ous": {
                "enable": true,
                "client_ou_identifier": {
                    "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ0lSQUx2SWV2KzE4Vm9LZFR2V1RLNCtaZ2d3Q2
↪",
                    "organizational_unit_identifier": "client"
                },
                "peer_ou_identifier": {
                    "certificate":
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ0lSQUx2SWV2KzE4Vm9LZFR2V1RLNCtaZ2d3Q2
↪",
                    "organizational_unit_identifier": "peer"
                }
            }
        },
        "Org3MSP": {
            "name": "Org3MSP",
            "root_certs": [
```

(continues on next page)

(continued from previous page)

```
                "CgJPVQoEUm9sZQoMRW5yb2xsbWVudElEChBSZXZvY2F0aW9uSGFuZGxlEkQKIKoEXcq/
↪psdYnMKCiT79N+dS1hM8k+SuzU1blOgTuN++EiBe2m3E+FjWLuQGMNRGRrEVTMqTvC4A/
↪5jvCLv2ja1sZxpECiDBbI0kwetxAwFzHwb1hi8TlkGW3OofvuVzfFt9VlewcRIgyvsxG5/
↪THdWyKJTdNx8Gle2hoCbVF0Y1/
↪DQESBjGOGciRAog25fMyWps+FLOjzj1vIsGUyO457ri3YMvmUcycIH2FvQSICTtzaFvSPUiDtNtAVz+uetuB9kfmjUdUSQxjyXU
↪uaJMuVph7Dy/
↪icgnAtVYHShET41O0Eh3Q5BIgy5q9VMQrch9VW5yajhY8dH1uA593gKd5kBqGdLfiXzAiRAogAnUYq/
↪kwKzFfmIm/
↪W4nZxi1kjG2C8NRjsYYBkeAOQ6wSIGyX5GGmwgvxgXXehNWBfijyNIJALGRVhO8YtBqr+vnrKogBCiDHR1XQsDbpcBoZFJ09V9
↪wwC+tq3oBIgSWT/
↪peiO2BI0DecypKfgMpVR8DWXl8ZHSrPISsL3Mc8aINem9+BOezLwFKCbtVH1KAHIRLyyiNP+TkIKW6x9RkThIiAbIJCYU6O02E
↪1lHxV0vtWdIsKCTLx2EZmDJECiCPXeyUyFzPS3iFv8CQUOLCPZxf6buZS5JlM6EE/
↪gCRaxIgmF9GKPLLmEoA77+AU3J8Iwnu9pBxnaHtUlyf/F9p30c6RAogG7ENKWlOZ4aF0HprqXAjl++Iao7/
↪iE8xeVcKRlmfq1ASIGtmmavDAVS2bw3zClQd4ZBD2DrqCBO9NPOcLNB0IWeIQiCjxTdbmcuBNINZYWe+5fWyI1oY9LavKzDVkd
↪PRAmBaeTQLXdbMxIthxM2gw+Zkc5+IJEWX"
            ],
            "intermediate_certs": [
                "CtgCCkQKIP0UVivtH8NlnRNrZuuu6jpaj2ZbEB4/
↪secGS57MfbINEiDSJweLUMIQSW12jugBQG81lIQflJWvi7vi925u+PU/
↪+xJECiDgOGdNbAiGSoHmTjKhT22fqUqYLIVh+JBHetm4kF4skhIg9XTWRkUqtsfYKENzPgm7ZUSmCHNF8xH7Vnhuc1EpAUgaINw
↪cIiCnlRj+mfNVAJGKthLgQBB/
↪JKM14NbUeutyJtTgrmDDiCogme25qGvxJfgQNnzldMMicVyiI6YMfnoThAUyqsTzyXkqIAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪El+BA2fOV0wyJxXKIjpgx+ehOEBTKqxPPJeSogAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAESIFYUenRvjbmEh+37Y
↪gwzULTJbCAoVg9XfCiROs4cU5oSv4Q80iYWtonAnvsSIE6mYFdzisBU21rhxjfYE7kk3Xjih9A1idJp7TSjfmorGiBwIEbnxUKj
↪yIgBVTjvNOIwpBC7qZJKX6yn4tMvoCCGpiz4BKBEUqtBJsaZzBlAjBwZ4WXYOttkhsNA2r94gBfLUdx/
↪4VhW4hwUImcztlau1T14UlNzJolCNkdiLc9CqsCMQD6OBkgDWGq9UlhkK9dJBzU+RElcZdSfVV1hDbbqt+lFRWOzzEkZ+BXCR1k
↪"
            ],
            "admins": [

↪"LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUhZd0VBWUhLb1pJemowQ0FRWUZLNEVFQUNJRFlnQUVUYk13SEZteEpEMWR3Sj
↪"
            ]
        }
    },
    "orderers": {
        "OrdererOrg": {
            "endpoint": [
                {
                    "host": "orderer.example.com",
                    "port": 7050
                }
            ]
        }
    }
}
```

It's important to note that the certificates here are base64 encoded, and thus should decoded in a manner similar to the following:

```
$ discover --configFile conf.yaml config --channel mychannel  --server peer0.org1.
↪example.com:7051 | jq .msps.OrdererOrg.root_certs[0] | sed "s/\"//g" | base64 --
↪decode | openssl x509 -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
```

(continues on next page)

(continued from previous page)

```
        Serial Number:
            c8:99:2d:3a:2d:7f:4b:73:53:8b:39:18:7b:c3:e1:1e
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, ST=California, L=San Francisco, O=example.com, CN=ca.example.com
        Validity
            Not Before: Jun  9 11:58:28 2018 GMT
            Not After : Jun  6 11:58:28 2028 GMT
        Subject: C=US, ST=California, L=San Francisco, O=example.com, CN=ca.example.
↪com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:28:ac:9e:51:8d:a4:80:15:0a:ff:ae:c9:61:d6:
                    08:67:b0:15:c3:c7:99:46:61:63:0a:10:a6:42:6a:
                    b0:af:14:0c:c0:e2:5b:b4:a1:c3:f0:07:7e:5b:7c:
                    c4:b2:95:13:95:81:4b:6a:b9:e3:87:a4:f3:2c:7c:
                    ae:00:91:9e:32
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment, Certificate Sign, CRL Sign
            X509v3 Extended Key Usage:
                Any Extended Key Usage
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                60:9D:F2:30:26:CE:8F:65:81:41:AD:96:15:0E:24:8D:A0:9D:C5:79:C1:17:BF:FE:E5:1B:FB:75:50:10:A6:4C
    Signature Algorithm: ecdsa-with-SHA256
         30:44:02:20:3d:e1:a7:6c:99:3f:87:2a:36:44:51:98:37:11:
         d8:a0:47:7a:33:ff:30:c1:09:a6:05:ec:b0:53:53:39:c1:0e:
         02:20:6b:f4:1d:48:e0:72:e4:c2:ef:b0:84:79:d4:2e:c2:c5:
         1b:6f:e4:2f:56:35:51:18:7d:93:51:86:05:84:ce:1f
```

11.10.6 Endorsers query:

To query for the endorsers of a chaincode call, additional flags need to be supplied:

- The `--chaincode` flag is mandatory and it provides the chaincode name(s). To query for a chaincode-to-chaincode invocation, one needs to repeat the `--chaincode` flag with all the chaincodes.
- The `--collection` is used to specify private data collections that are expected to used by the chaincode(s). To map from thechaincodes passed via `--chaincode` to the collections, the following syntax should be used: `collection=CC:Collection1,Collection2,...`.
- The `--noPrivateReads` is used to indicate that the transaction is not expected to read private data for a certain chaincode. This is useful for private data "blind writes", among other things.

For example, to query for a chaincode invocation that results in both cc1 and cc2 to be invoked, as well as writes to private data collection col1 by cc2, one needs to specify: `--chaincode=cc1 --chaincode=cc2 --collection=cc2:col1`

If chaincode cc2 is not expected to read from collection `col1` then `--noPrivateReads=cc2` should be used.

Below is the output of an endorsers query for chaincode **mycc** when the endorsement policy is `AND('Org1.peer', 'Org2.peer')`:

```
$ discover --configFile conf.yaml endorsers --channel mychannel  --server peer0.org1.
↪example.com:7051 --chaincode mycc
[
    {
        "Chaincode": "mycc",
        "EndorsersByGroups": {
            "G0": [
                {
                    "MSPID": "Org1MSP",
                    "LedgerHeight": 5,
                    "Endpoint": "peer0.org1.example.com:7051",
                    "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKDCCAc+gAwIBAgIRANTiKfUVHVGnrYVzEy1ZSKIwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkN
↪k\n/CtORCDPQ02jTTBLMA4GA1UdDwEB/
↪wQEAwIHgDAMBgNVHRMBAf8EAjAAMCsGA1Ud\nIwQkMCKAIOBdQLF+cMWa6e1p2CpOEx7SHUinzVvd55hLm7w6v72oMAoGCCqGSM
↪zwD08t7hJxNe8MwgP8/48fAiBiC0cr\nu99oLsRNCFB7R3egyKg1YYao0KWTrr1T+rK9Bg==\n-----END␣
↪CERTIFICATE-----\n"
                }
            ],
            "G1": [
                {
                    "MSPID": "Org2MSP",
                    "LedgerHeight": 5,
                    "Endpoint": "peer1.org2.example.com:10051",
                    "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKDCCAc+gAwIBAgIRAIs6fFxk4Y5cJxSwTjyJ9A8wCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkN
↪cq\n0cGrMKR93vKjTTBLMA4GA1UdDwEB/
↪wQEAwIHgDAMBgNVHRMBAf8EAjAAMCsGA1Ud\nIwQkMCKAII5YgskKERCpC5MD7qBUQvSj7xFMgrb5zhCiHiSrE4KgMAoGCCqGSM
↪OidQ2SBR7OZyMAzgXc5nAabWZpdkuQ==\n-----END CERTIFICATE-----\n"
                },
                {
                    "MSPID": "Org2MSP",
                    "LedgerHeight": 5,
                    "Endpoint": "peer0.org2.example.com:9051",
                    "Identity": "-----BEGIN CERTIFICATE-----\nMIICJzCCAc6gAwIBAgIQVek/
↪l5TVdNvi1pk8ASS+vzAKBggqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMNU
↪BAQDAgeAMAwGA1UdEwEB/
↪wQCMAAwKwYDVR0j\nBCQwIoAgjliCyQoREKkLkwPuoFRC9KPvEUyCtvnOEKIeJKsTgqAwCgYIKoZIzj0E\nAwIDRwAwRAIgKT9V
↪yu/CH9yDajGDlYIHI9GkN0MPNWAaom\n-----END CERTIFICATE-----\n"
                }
            ]
        },
        "Layouts": [
            {
                "quantities_by_group": {
                    "G0": 1,
                    "G1": 1
                }
            }
        ]
    }
]
```

11.10.7 Not using a configuration file

It is possible to execute the discovery CLI without having a configuration file, and just passing all needed configuration as commandline flags. The following is an example of a local peer membership query which loads administrator credentials:

```
$ discover --peerTLSCA tls/ca.crt --userKey msp/keystore/
↪cf31339d09e8311ac9ca5ed4e27a104a7f82f1e5904b3296a170ba4725ffde0d_sk --userCert msp/
↪signcerts/Admin\@org1.example.com-cert.pem --MSP Org1MSP --tlsCert tls/client.crt --
↪tlsKey tls/client.key peers --server peer0.org1.example.com:7051
[
    {
        "MSPID": "Org1MSP",
        "Endpoint": "peer1.org1.example.com:8051",
        "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICJzCCAc6gAwIBAgIQO7zMEHlMfRhnP6Xt65jwtDAKBggqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMBEGA1UECBMKQ2F
↪Q2g\nRHw5rk3SYw+OMFw9jNbsJJyC5ttJRvc12Dn7lQ8ZR9hW1vLQ3NtqO/
↪couccDJcHg\nt47iHBNadaNNMEswDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/
↪wQCMAAwKwYDVR0j\nBCQwIoAgcecTOxTes6rfgyxHH6KIW7hsRAw2bhP9ikCHkvtv/
↪RcwCgYIKoZIzj0E\nAwIDRwAwRAIgGHGtRVxcFVeMQr9yRlebs23OXEECNo6hNqd/
↪4ChLwwoCIBFKFd6t\nlL5BVzVMGQyXWcZGrjFgl4+fDrwjmMe+jAfa\n-----END CERTIFICATE-----\n
↪",
    },
    {
        "MSPID": "Org1MSP",
        "Endpoint": "peer0.org1.example.com:7051",
        "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKDCCAc6gAwIBAgIQP18LeXtEXGoN8pTqzXTHZTAKBggqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMBEGA1UECBMKQ2F
↪1Rg/ynSk\nNNItaMlaCDZOaQvxJEl6o3fqx1PVFlfXE4NarY3OO1N3YZI41hWWoXksSwJu/
↪35S\nM7wMEzw+3KNNMEswDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/
↪wQCMAAwKwYDVR0j\nBCQwIoAgcecTOxTes6rfgyxHH6KIW7hsRAw2bhP9ikCHkvtv/
↪RcwCgYIKoZIzj0E\nAwIDSAAwRQIhAKiJEv79XBmr8gGY6kHrGL0L3sq95E7IsCYzYdAQHj+DAiBPcBTg\nRuA0/
↪/Kq+3aHJ2T0KpKHqD3FfhZZolKDkcrkwQ==\n-----END CERTIFICATE-----\n",
    },
    {
        "MSPID": "Org2MSP",
        "Endpoint": "peer0.org2.example.com:9051",
        "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKTCCAc+gAwIBAgIRANK4WBck5gKuzTxVQIwhYMUwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkN
↪ecJNvdAV2zmSx5Sf2qospVAH1MYCHyudDEvkiRuBPgmCdOdwJsE0g+h\nz0nZdKq6/
↪X+jTTBLMA4GA1UdDwEB/
↪wQEAwIHgDAMBgNVHRMBAf8EAjAAMCsGA1Ud\nIwQkMCKAIFZMuZfUtY6n2iyxaVr3rl+x5lU0CdG9x7KAeYydQGTMMAoGCCqGSN
↪LJ7j3I9NEPQ/B1BpnJP+UNPnGO2peVrM/
↪mJ1nVgIgS1ZA\nA1tsxuDyllaQuHx2P+P9NDFdjXx5T08lZhxuWYM=\n-----END CERTIFICATE-----\n
↪",
    },
    {
        "MSPID": "Org2MSP",
        "Endpoint": "peer1.org2.example.com:10051",
        "Identity": "-----BEGIN CERTIFICATE-----
↪\nMIICKDCCAc+gAwIBAgIRALnNJzplCrYy4Y8CjZtqL7AwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkN
↪YMn1hS6sM+bFDgkJKalG7s9Hg3URF0aGpy51R\nU+4F9Muo+XajTTBLMA4GA1UdDwEB/
↪wQEAwIHgDAMBgNVHRMBAf8EAjAAMCsGA1Ud\nIwQkMCKAIFZMuZfUtY6n2iyxaVr3rl+x5lU0CdG9x7KAeYydQGTMMAoGCCqGSN
↪ExunQ==\n-----END CERTIFICATE-----\n",
    }
]
```

11.11 Fabric-CA Commands

The Hyperledger Fabric CA is a Certificate Authority (CA) for Hyperledger Fabric. The commands available for the fabric-ca client and fabric-ca server are described in the links below.

11.11.1 Fabric-CA Client

The fabric-ca-client command allows you to manage identities (including attribute management) and certificates (including renewal and revocation).

More information on `fabric-ca-client` commands can be found here.

11.11.2 Fabric-CA Server

The fabric-ca-server command allows you to initialize and start a server process which may host one or more certificate authorities.

More information on `fabric-ca-server` commands can be found here.

CHAPTER 12

Architecture Reference

12.1 Hyperledger Fabric SDKs

Hyperledger Fabric intends to offer a number of SDKs for a wide variety of programming languages. The first two delivered are the Node.js and Java SDKs. We hope to provide Python, REST and Go SDKs in a subsequent release.

- Hyperledger Fabric Node SDK documentation.
- Hyperledger Fabric Java SDK documentation.

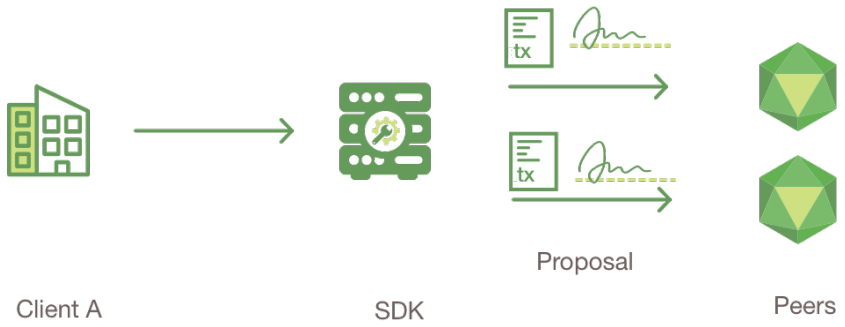
12.2 Transaction Flow

This document outlines the transactional mechanics that take place during a standard asset exchange. The scenario includes two clients, A and B, who are buying and selling radishes. They each have a peer on the network through which they send their transactions and interact with the ledger.

Assumptions

This flow assumes that a channel is set up and running. The application user has registered and enrolled with the organization's Certificate Authority (CA) and received back necessary cryptographic material, which is used to authenticate to the network.

The chaincode (containing a set of key value pairs representing the initial state of the radish market) is installed on the peers and deployed to the channel. The chaincode contains logic defining a set of transaction instructions and the agreed upon price for a radish. An endorsement policy has also been set for this chaincode, stating that both `peerA` and `peerB` must endorse any transaction.

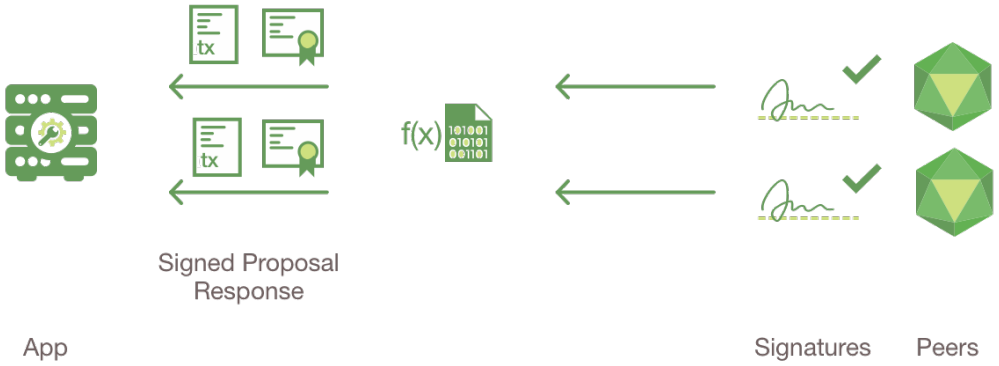


1. **Client A initiates a transaction**

What's happening? Client A is sending a request to purchase radishes. This request targets `peerA` and `peerB`, who are respectively representative of Client A and Client B. The endorsement policy states that both peers must endorse any transaction, therefore the request goes to `peerA` and `peerB`.

Next, the transaction proposal is constructed. An application leveraging a supported SDK (Node, Java, Python) utilizes one of the available API's to generate a transaction proposal. The proposal is a request to invoke a chaincode function with certain input parameters, with the intent of reading and/or updating the ledger.

The SDK serves as a shim to package the transaction proposal into the properly architected format (protocol buffer over gRPC) and takes the user's cryptographic credentials to produce a unique signature for this transaction proposal.



2. **Endorsing peers verify signature & execute the transaction**

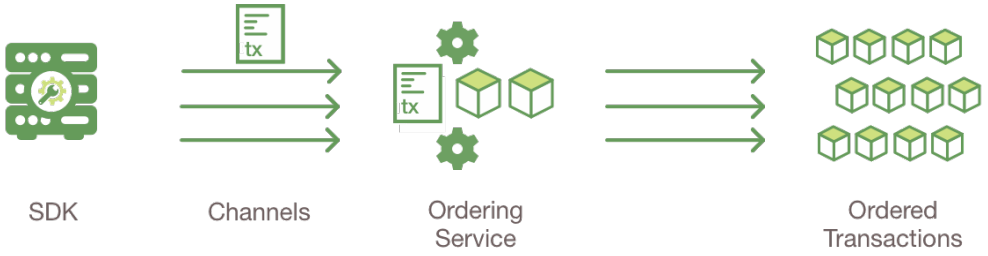
The endorsing peers verify (1) that the transaction proposal is well formed, (2) it has not been submitted already in the past (replay-attack protection), (3) the signature is valid (using the MSP), and (4) that the submitter (Client A, in the example) is properly authorized to perform the proposed operation on that channel (namely, each endorsing peer ensures that the submitter satisfies the channel's *Writers* policy). The endorsing peers take the transaction proposal inputs as arguments to the invoked chaincode's function. The chaincode is then executed against the current state database to produce transaction results including a response value, read set, and write set (i.e. key/value pairs representing an asset to create or update). No updates are made to the ledger at this point. The set of these values, along with the endorsing peer's signature is passed back as a "proposal response" to the SDK which parses the payload for the application to consume.

Note: The MSP is a peer component that allows peers to verify transaction requests arriving from clients and to

sign transaction results (endorsements). The writing policy is defined at channel creation time and determines which users are entitled to submit a transaction to that channel. For more information about membership, check out our *Membership Service Provider (MSP)* documentation.

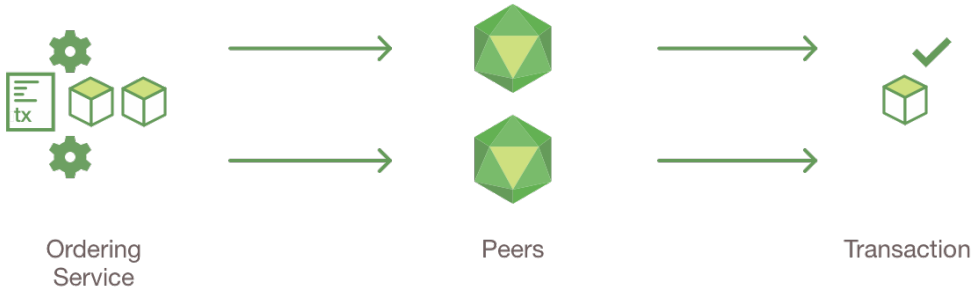
3. **Proposal responses are inspected**

The application verifies the endorsing peer signatures and compares the proposal responses to determine if the proposal responses are the same. If the chaincode is only querying the ledger, the application would only inspect the query response and would typically not submit the transaction to the ordering service. If the client application intends to submit the transaction to the ordering service to update the ledger, the application determines if the specified endorsement policy has been fulfilled before submitting (i.e. did peerA and peerB both endorse). The architecture is such that even if an application chooses not to inspect responses or otherwise forwards an unendorsed transaction, the endorsement policy will still be enforced by peers and upheld at the commit validation phase.



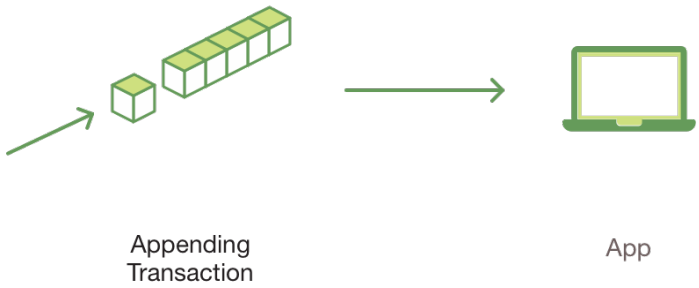
4. **Client assembles endorsements into a transaction**

The application "broadcasts" the transaction proposal and response within a "transaction message" to the ordering service. The transaction will contain the read/write sets, the endorsing peers signatures and the Channel ID. The ordering service does not need to inspect the entire content of a transaction in order to perform its operation, it simply receives transactions from all channels in the network, orders them chronologically by channel, and creates blocks of transactions per channel.



5. **Transaction is validated and committed**

The blocks of transactions are "delivered" to all peers on the channel. The transactions within the block are validated to ensure endorsement policy is fulfilled and to ensure that there have been no changes to ledger state for read set variables since the read set was generated by the transaction execution. Transactions in the block are tagged as being valid or invalid.



6. **Ledger updated**

Each peer appends the block to the channel's chain, and for each valid transaction the write sets are committed to current state database. An event is emitted by each peer to notify the client application that the transaction (invocation) has been immutably appended to the chain, as well as notification of whether the transaction was validated or invalidated.

Note: Applications should listen for the transaction event after submitting a transaction, for example by using the `submitTransaction` API, which automatically listen for transaction events. Without listening for transaction events, you will not know whether your transaction has actually been ordered, validated, and committed to the ledger.

See the sequence diagram to better understand the transaction flow.

https://creativecommons.org/licenses/by/4.0/

12.3 Service Discovery

12.3.1 Why do we need service discovery?

In order to execute chaincode on peers, submit transactions to orderers, and to be updated about the status of transactions, applications connect to an API exposed by an SDK.

However, the SDK needs a lot of information in order to allow applications to connect to the relevant network nodes. In addition to the CA and TLS certificates of the orderers and peers on the channel – as well as their IP addresses and port numbers – it must know the relevant endorsement policies as well as which peers have the chaincode installed (so the application knows which peers to send chaincode proposals to).

Prior to v1.2, this information was statically encoded. However, this implementation is not dynamically reactive to network changes (such as the addition of peers who have installed the relevant chaincode, or peers that are temporarily offline). Static configurations also do not allow applications to react to changes of the endorsement policy itself (as might happen when a new organization joins a channel).

In addition, the client application has no way of knowing which peers have updated ledgers and which do not. As a result, the application might submit proposals to peers whose ledger data is not in sync with the rest of the network, resulting in transaction being invalidated upon commit and wasting resources as a consequence.

The **discovery service** improves this process by having the peers compute the needed information dynamically and present it to the SDK in a consumable manner.

12.3.2 How service discovery works in Fabric

The application is bootstrapped knowing about a group of peers which are trusted by the application developer/administrator to provide authentic responses to discovery queries. A good candidate peer to be used by the client application is one that is in the same organization. Note that in order for peers to be known to the discovery service, they must have an `EXTERNAL_ENDPOINT` defined. To see how to do this, check out our *Service Discovery CLI* documentation.

The application issues a configuration query to the discovery service and obtains all the static information it would have otherwise needed to communicate with the rest of the nodes of the network. This information can be refreshed at any point by sending a subsequent query to the discovery service of a peer.

The service runs on peers – not on the application – and uses the network metadata information maintained by the gossip communication layer to find out which peers are online. It also fetches information, such as any relevant endorsement policies, from the peer's state database.

With service discovery, applications no longer need to specify which peers they need endorsements from. The SDK can simply send a query to the discovery service asking which peers are needed given a channel and a chaincode ID. The discovery service will then compute a descriptor comprised of two objects:

1. **Layouts**: a list of groups of peers and a corresponding amount of peers from each group which should be selected.
2. **Group to peer mapping**: from the groups in the layouts to the peers of the channel. In practice, each group would most likely be peers that represent individual organizations, but because the service API is generic and ignorant of organizations this is just a "group".

The following is an example of a descriptor from the evaluation of a policy of `AND(Org1, Org2)` where there are two peers in each of the organizations.

```
Layouts: [
     QuantitiesByGroup: {
       "Org1": 1,
       "Org2": 1,
     }
],
EndorsersByGroups: {
  "Org1": [peer0.org1, peer1.org1],
  "Org2": [peer0.org2, peer1.org2]
}
```

In other words, the endorsement policy requires a signature from one peer in Org1 and one peer in Org2. And it provides the names of available peers in those orgs who can endorse (`peer0` and `peer1` in both Org1 and in Org2).

The SDK then selects a random layout from the list. In the example above, the endorsement policy is Org1 `AND` Org2. If instead it was an `OR` policy, the SDK would randomly select either Org1 or Org2, since a signature from a peer from either Org would satisfy the policy.

After the SDK has selected a layout, it selects from the peers in the layout based on a criteria specified on the client side (the SDK can do this because it has access to metadata like ledger height). For example, it can prefer peers with higher ledger heights over others – or to exclude peers that the application has discovered to be offline – according to the number of peers from each group in the layout. If no single peer is preferable based on the criteria, the SDK will randomly select from the peers that best meet the criteria.

Capabilities of the discovery service

The discovery service can respond to the following queries:

- **Configuration query**: Returns the `MSPConfig` of all organizations in the channel along with the orderer endpoints of the channel.
- **Peer membership query**: Returns the peers that have joined the channel.
- **Endorsement query**: Returns an endorsement descriptor for given chaincode(s) in a channel.
- **Local peer membership query**: Returns the local membership information of the peer that responds to the query. By default the client needs to be an administrator for the peer to respond to this query.

Special requirements

When the peer is running with TLS enabled the client must provide a TLS certificate when connecting to the peer. If the peer isn't configured to verify client certificates (clientAuthRequired is false), this TLS certificate can be self-signed.

12.4 Defining capability requirements

As discussed in *Channel capabilities*, capability requirements are defined per channel in the channel configuration (found in the channel's most recent configuration block). The channel configuration contains three locations, each of which defines a capability of a different type.

Capability Type	Canonical Path	JSON Path
Channel	/Channel/Capabilities	.channel_group.values.Capabilities
Orderer	/Channel/Orderer/Capabilities	.channel_group.groups.Orderer.values.Capabilities
Application	/Channel/Application/Capabilities	.channel_group.groups.Application.values. Capabilities

12.4.1 Setting Capabilities

Capabilities are set as part of the channel configuration (either as part of the initial configuration – which we'll talk about in a moment – or as part of a reconfiguration).

Note: For more information about how to update a channel configuration, check out *Updating a channel configuration*.

Because new channels copy the configuration of the ordering system channel by default, new channels will automatically be configured to work with the orderer and channel capabilities of the ordering system channel and the application capabilities specified by the channel creation transaction.

Capabilities in an Initial Configuration

In the `configtx.yaml` file distributed in the `config` directory of the release artifacts, there is a `Capabilities` section which enumerates the possible capabilities for each capability type (Channel, Orderer, and Application).

Note that there is a `Capabilities` section defined at the root level (for the channel capabilities), and at the Orderer level (for orderer capabilities).

When defining the orderer system channel there is no Application section, as those capabilities are defined during the creation of an application channel.

12.5 Channels

A Hyperledger Fabric `channel` is a private "subnet" of communication between two or more specific network members, for the purpose of conducting private and confidential transactions. A channel is defined by members (organizations), anchor peers per member, the shared ledger, chaincode application(s) and the ordering service node(s). Each transaction on the network is executed on a channel, where each party must be authenticated and authorized to transact on that channel. Each peer that joins a channel, has its own identity given by a membership services provider (MSP), which authenticates each peer to its channel peers and services.

To create a new channel, the client SDK calls configuration system chaincode and references properties such as `anchor peers`, and members (organizations). This request creates a `genesis block` for the channel ledger, which stores configuration information about the channel policies, members and anchor peers. When adding a new member to an existing channel, either this genesis block, or if applicable, a more recent reconfiguration block, is shared with the new member.

Note: See the *Channel Configuration (configtx)* section for more details on the properties and proto structures of config transactions.

The election of a `leading peer` for each member on a channel determines which peer communicates with the ordering service on behalf of the member. If no leader is identified, an algorithm can be used to identify the leader. The consensus service orders transactions and delivers them, in a block, to each leading peer, which then distributes the block to its member peers, and across the channel, using the `gossip` protocol.

Although any one anchor peer can belong to multiple channels, and therefore maintain multiple ledgers, no ledger data can pass from one channel to another. This separation of ledgers, by channel, is defined and implemented by configuration chaincode, the identity membership service and the gossip data dissemination protocol. The dissemination of data, which includes information on transactions, ledger state and channel membership, is restricted to peers with verifiable membership on the channel. This isolation of peers and ledger data, by channel, allows network members that require private and confidential transactions to coexist with business competitors and other restricted members, on the same blockchain network.

12.6 CouchDB as the State Database

12.6.1 State Database options

The current options for the peer state database are LevelDB and CouchDB. LevelDB is the default key-value state database embedded in the peer process. CouchDB is an alternative external state database. Like the LevelDB key-value store, CouchDB can store any binary data that is modeled in chaincode (CouchDB attachments are used internally for non-JSON data). As a document object store, CouchDB allows you to store data in JSON format, issue rich queries against your data, and use indexes to support your queries.

Both LevelDB and CouchDB support core chaincode operations such as getting and setting a key (asset), and querying based on keys. Keys can be queried by range, and composite keys can be modeled to enable equivalence queries against multiple parameters. For example a composite key of `owner,asset_id` can be used to query all assets owned by a certain entity. These key-based queries can be used for read-only queries against the ledger, as well as in transactions that update the ledger.

Modeling your data in JSON allows you to issue rich queries against the values of your data, instead of only being able to query the keys. This makes it easier for your applications and chaincode to read the data stored on the blockchain ledger. Using CouchDB can help you meet auditing and reporting requirements for many use cases that are not supported by LevelDB. If you use CouchDB and model your data in JSON, you can also deploy indexes with your chaincode. Using indexes makes queries more flexible and efficient and enables you to query large datasets from chaincode.

CouchDB runs as a separate database process alongside the peer, therefore there are additional considerations in terms of setup, management, and operations. You may consider starting with the default embedded LevelDB, and move to CouchDB if you require the additional complex rich queries. It is a good practice to model asset data as JSON, so that you have the option to perform complex rich queries if needed in the future.

Note: The key for a CouchDB JSON document can only contain valid UTF-8 strings and cannot begin with an underscore ("_"). Whether you are using CouchDB or LevelDB, you should avoid using U+0000 (nil byte) in keys.

JSON documents in CouchDB cannot use the following values as top level field names. These values are reserved for internal use.

- `Any field beginning with an underscore, "_"`
- `~version`

12.6.2 Using CouchDB from Chaincode

Chaincode queries

Most of the chaincode shim APIs can be utilized with either LevelDB or CouchDB state database, e.g. `GetState`, `PutState`, `GetStateByRange`, `GetStateByPartialCompositeKey`. Additionally when you utilize CouchDB as the state database and model assets as JSON in chaincode, you can perform rich queries against the JSON in the state database by using the `GetQueryResult` API and passing a CouchDB query string. The query string follows the CouchDB JSON query syntax.

The marbles02 fabric sample demonstrates use of CouchDB queries from chaincode. It includes a `queryMarblesByOwner()` function that demonstrates parameterized queries by passing an owner id into chaincode. It then queries the state data for JSON documents matching the docType of "marble" and the owner id using the JSON query syntax:

```
{"selector":{"docType":"marble","owner":<OWNER_ID>}}
```

The responses to rich queries are useful for understanding the data on the ledger. However, there is no guarantee that the result set for a rich query will be stable between the chaincode execution and commit time. As a result, you should not use a rich query and update the channel ledger in a single transaction. For example, if you perform a rich query for all assets owned by Alice and transfer them to Bob, a new asset may be assigned to Alice by another transaction between chaincode execution time and commit time.

CouchDB pagination

Fabric supports paging of query results for rich queries and range based queries. APIs supporting pagination allow the use of page size and bookmarks to be used for both range and rich queries. To support efficient pagination, the Fabric pagination APIs must be used. Specifically, the CouchDB `limit` keyword will not be honored in CouchDB queries since Fabric itself manages the pagination of query results and implicitly sets the pageSize limit that is passed to CouchDB.

If a pageSize is specified using the paginated query APIs (`GetStateByRangeWithPagination()`, `GetStateByPartialCompositeKeyWithPagination()`, and `GetQueryResultWithPagination()`), a set of results (bound by the pageSize) will be returned to the chaincode along with a bookmark. The bookmark can be returned from chaincode to invoking clients, which can use the bookmark in a follow on query to receive the next "page" of results.

The pagination APIs are for use in read-only transactions only, the query results are intended to support client paging requirements. For transactions that need to read and write, use the non-paginated chaincode query APIs. Within chaincode you can iterate through result sets to your desired depth.

Regardless of whether the pagination APIs are utilized, all chaincode queries are bound by `totalQueryLimit` (default 100000) from `core.yaml`. This is the maximum number of results that chaincode will iterate through and return to the client, in order to avoid accidental or malicious long-running queries.

Note: Regardless of whether chaincode uses paginated queries or not, the peer will query CouchDB in batches based on `internalQueryLimit` (default 1000) from `core.yaml`. This behavior ensures reasonably sized result sets are passed between the peer and CouchDB when executing chaincode, and is transparent to chaincode and the calling client.

An example using pagination is included in the *Using CouchDB* tutorial.

CouchDB indexes

Indexes in CouchDB are required in order to make JSON queries efficient and are required for any JSON query with a sort. Indexes enable you to query data from chaincode when you have a large amount of data on your ledger. Indexes can be packaged alongside chaincode in a `/META-INF/statedb/couchdb/indexes` directory. Each index must be defined in its own text file with extension `*.json` with the index definition formatted in JSON following the CouchDB index JSON syntax. For example, to support the above marble query, a sample index on the `docType` and `owner` fields is provided:

```
{"index":{"fields":["docType","owner"]},"ddoc":"indexOwnerDoc", "name":"indexOwner",
↪"type":"json"}
```

The sample index can be found here.

Any index in the chaincode's `META-INF/statedb/couchdb/indexes` directory will be packaged up with the chaincode for deployment. The index will be deployed to a peers channel and chaincode specific database when the

chaincode package is installed on the peer and the chaincode definition is committed to the channel. If you install the chaincode first and then commit the the chaincode definition to the channel, the index will be deployed at commit time. If the chaincode has already been defined on the channel and the chaincode package subsequently installed on a peer joined to the channel, the index will be deployed at chaincode **installation** time.

Upon deployment, the index will automatically be utilized by chaincode queries. CouchDB can automatically determine which index to use based on the fields being used in a query. Alternatively, in the selector query the index can be specified using the `use_index` keyword.

The same index may exist in subsequent versions of the chaincode that gets installed. To change the index, use the same index name but alter the index definition. Upon installation/instantiation, the index definition will get re-deployed to the peer's state database.

If you have a large volume of data already, and later install the chaincode, the index creation upon installation may take some time. Similarly, if you have a large volume of data already and commit the definition of a subsequent chaincode version, the index creation may take some time. Avoid calling chaincode functions that query the state database at these times as the chaincode query may time out while the index is getting initialized. During transaction processing, the indexes will automatically get refreshed as blocks are committed to the ledger. If the peer crashes during chaincode installation, the couchdb indexes may not get created. If this occurs, you need to reinstall the chaincode to create the indexes.

12.6.3 CouchDB Configuration

CouchDB is enabled as the state database by changing the `stateDatabase` configuration option from goleveldb to CouchDB. Additionally, the `couchDBAddress` needs to configured to point to the CouchDB to be used by the peer. The username and password properties should be populated with an admin username and password if CouchDB is configured with a username and password. Additional options are provided in the `couchDBConfig` section and are documented in place. Changes to the *core.yaml* will be effective immediately after restarting the peer.

You can also pass in docker environment variables to override core.yaml values, for example `CORE_LEDGER_STATE_STATEDATABASE` and `CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS`.

Below is the `stateDatabase` section from *core.yaml*:

```
state:
  # stateDatabase - options are "goleveldb", "CouchDB"
  # goleveldb - default state database stored in goleveldb.
  # CouchDB - store state database in CouchDB
  stateDatabase: goleveldb
  # Limit on the number of records to return per query
  totalQueryLimit: 10000
  couchDBConfig:
     # It is recommended to run CouchDB on the same server as the peer, and
     # not map the CouchDB container port to a server port in docker-compose.
     # Otherwise proper security must be provided on the connection between
     # CouchDB client (on the peer) and server.
     couchDBAddress: couchdb:5984
     # This username must have read and write authority on CouchDB
     username:
     # The password is recommended to pass as an environment variable
     # during start up (e.g. LEDGER_COUCHDBCONFIG_PASSWORD).
     # If it is stored here, the file must be access control protected
     # to prevent unintended users from discovering the password.
     password:
     # Number of retries for CouchDB errors
     maxRetries: 3
     # Number of retries for CouchDB errors during peer startup
```

(continues on next page)

(continued from previous page)

```
    maxRetriesOnStartup: 10
    # CouchDB request timeout (unit: duration, e.g. 20s)
    requestTimeout: 35s
    # Limit on the number of records per each CouchDB query
    # Note that chaincode queries are only bound by totalQueryLimit.
    # Internally the chaincode may execute multiple CouchDB queries,
    # each of size internalQueryLimit.
    internalQueryLimit: 1000
    # Limit on the number of records per CouchDB bulk update batch
    maxBatchUpdateSize: 1000
    # Warm indexes after every N blocks.
    # This option warms any indexes that have been
    # deployed to CouchDB after every N blocks.
    # A value of 1 will warm indexes after every block commit,
    # to ensure fast selector queries.
    # Increasing the value may improve write efficiency of peer and CouchDB,
    # but may degrade query response time.
    warmIndexesAfterNBlocks: 1
```

CouchDB hosted in docker containers supplied with Hyperledger Fabric have the capability of setting the CouchDB username and password with environment variables passed in with the `COUCHDB_USER` and `COUCHDB_PASSWORD` environment variables using Docker Compose scripting.

For CouchDB installations outside of the docker images supplied with Fabric, the local.ini file of that installation must be edited to set the admin username and password.

Docker compose scripts only set the username and password at the creation of the container. The *local.ini* file must be edited if the username or password is to be changed after creation of the container.

If you choose to map the fabric-couchdb container port to a host port, make sure you are aware of the security implications. Mapping the CouchDB container port in a development environment exposes the CouchDB REST API and allows you to visualize the database via the CouchDB web interface (Fauxton). In a production environment you should refrain from mapping the host port to restrict access to the CouchDB container. Only the peer will be able to access the CouchDB container.

Note: CouchDB peer options are read on each peer startup.

12.6.4 Good practices for queries

Avoid using chaincode for queries that will result in a scan of the entire CouchDB database. Full length database scans will result in long response times and will degrade the performance of your network. You can take some of the following steps to avoid long queries:

- When using JSON queries:
  - Be sure to create indexes in the chaincode package.
  - Avoid query operators such as `$or`, `$in` and `$regex`, which lead to full database scans.
- For range queries, composite key queries, and JSON queries:
  - Utilize paging support instead of one large result set.
- If you want to build a dashboard or collect aggregate data as part of your application, you can query an off-chain database that replicates the data from your blockchain network. This will allow you to query and analyze the blockchain data in a data store optimized for your needs, without degrading the performance of your network or

disrupting transactions. To achieve this, applications may use block or chaincode events to write transaction data to an off-chain database or analytics engine. For each block received, the block listener application would iterate through the block transactions and build a data store using the key/value writes from each valid transaction's `rwset`. The *Peer channel-based event services* provide replayable events to ensure the integrity of downstream data stores.

12.7 Peer channel-based event services

12.7.1 General overview

In previous versions of Fabric, the peer event service was known as the event hub. This service sent events any time a new block was added to the peer's ledger, regardless of the channel to which that block pertained, and it was only accessible to members of the organization running the eventing peer (i.e., the one being connected to for events).

Starting with v1.1, there are new services which provide events. These services use an entirely different design to provide events on a per-channel basis. This means that registration for events occurs at the level of the channel instead of the peer, allowing for fine-grained control over access to the peer's data. Requests to receive events are accepted from identities outside of the peer's organization (as defined by the channel configuration). This also provides greater reliability and a way to receive events that may have been missed (whether due to a connectivity issue or because the peer is joining a network that has already been running).

12.7.2 Available services

- `Deliver`

This service sends entire blocks that have been committed to the ledger. If any events were set by a chaincode, these can be found within the `ChaincodeActionPayload` of the block.

- `DeliverWithPrivateData`

This service sends the same data as the `Deliver` service, and additionally includes any private data from collections that the client's organization is authorized to access.

- `DeliverFiltered`

This service sends "filtered" blocks, minimal sets of information about blocks that have been committed to the ledger. It is intended to be used in a network where owners of the peers wish for external clients to primarily receive information about their transactions and the status of those transactions. If any events were set by a chaincode, these can be found within the `FilteredChaincodeAction` of the filtered block.

Note: The payload of chaincode events will not be included in filtered blocks.

12.7.3 How to register for events

Registration for events is done by sending an envelope containing a deliver seek info message to the peer that contains the desired start and stop positions, the seek behavior (block until ready or fail if not ready). There are helper variables `SeekOldest` and `SeekNewest` that can be used to indicate the oldest (i.e. first) block or the newest (i.e. last) block on the ledger. To have the services send events indefinitely, the `SeekInfo` message should include a stop position of `MAXINT64`.

Note: If mutual TLS is enabled on the peer, the TLS certificate hash must be set in the envelope's channel header.

By default, the event services use the Channel Readers policy to determine whether to authorize requesting clients for events.

12.7.4 Overview of deliver response messages

The event services send back `DeliverResponse` messages.

Each message contains one of the following:

- status – HTTP status code. Each of the services will return the appropriate failure code if any failure occurs; otherwise, it will return `200 - SUCCESS` once the service has completed sending all information requested by the `SeekInfo` message.
- block – returned only by the `Deliver` service.
- block and private data – returned only by the `DeliverWithPrivateData` service.
- filtered block – returned only by the `DeliverFiltered` service.

A filtered block contains:

- channel ID.
- number (i.e. the block number).
- array of filtered transactions.
- transaction ID.
  - type (e.g. `ENDORSER_TRANSACTION`, `CONFIG`).
  - transaction validation code.
- **filtered transaction actions.**
  - **array of filtered chaincode actions.**
    - chaincode event for the transaction (with the payload nilled out).

12.7.5 SDK event documentation

For further details on using the event services, refer to the SDK documentation.

12.8 Private Data

Note: This topic assumes an understanding of the conceptual material in the documentation on private data.

12.8.1 Private data collection definition

A collection definition contains one or more collections, each having a policy definition listing the organizations in the collection, as well as properties used to control dissemination of private data at endorsement time and, optionally, whether the data will be purged.

Beginning with the Fabric chaincode lifecycle introduced with Fabric v2.0, the collection definition is part of the chaincode definition. The collection is approved by channel members, and then deployed when the chaincode definition is committed to the channel. The collection file needs to be the same for all channel members. If you are using the peer CLI to approve and commit the chaincode definition, use the `--collections-config` flag to specify the path to the collection definition file. If you are using the Fabric SDK for Node.js, visit How to install and start your chaincode. To use the previous lifecycle process to deploy a private data collection, use the `--collections-config` flag when instantiating your chaincode.

Collection definitions are composed of the following properties:

- `name`: Name of the collection.
- `policy`: The private data collection distribution policy defines which organizations' peers are allowed to persist the collection data expressed using the `Signature` policy syntax, with each member being included in an `OR` signature policy list. To support read/write transactions, the private data distribution policy must define a broader set of organizations than the chaincode endorsement policy, as peers must have the private data in order to endorse proposed transactions. For example, in a channel with ten organizations, five of the organizations might be included in a private data collection distribution policy, but the endorsement policy might call for any three of the organizations to endorse.
- `requiredPeerCount`: Minimum number of peers (across authorized organizations) that each endorsing peer must successfully disseminate private data to before the peer signs the endorsement and returns the proposal response back to the client. Requiring dissemination as a condition of endorsement will ensure that private data is available in the network even if the endorsing peer(s) become unavailable. When `requiredPeerCount` is `0`, it means that no distribution is **required**, but there may be some distribution if `maxPeerCount` is greater than zero. A `requiredPeerCount` of `0` would typically not be recommended, as it could lead to loss of private data in the network if the endorsing peer(s) becomes unavailable. Typically you would want to require at least some distribution of the private data at endorsement time to ensure redundancy of the private data on multiple peers in the network.
- `maxPeerCount`: For data redundancy purposes, the maximum number of other peers (across authorized organizations) that each endorsing peer will attempt to distribute the private data to. If an endorsing peer becomes unavailable between endorsement time and commit time, other peers that are collection members but who did not yet receive the private data at endorsement time, will be able to pull the private data from peers the private data was disseminated to. If this value is set to `0`, the private data is not disseminated at endorsement time, forcing private data pulls against endorsing peers on all authorized peers at commit time.
- `blockToLive`: Represents how long the data should live on the private database in terms of blocks. The data will live for this specified number of blocks on the private database and after that it will get purged, making this data obsolete from the network so that it cannot be queried from chaincode, and cannot be made available to requesting peers. To keep private data indefinitely, that is, to never purge private data, set the `blockToLive` property to `0`.
- `memberOnlyRead`: a value of `true` indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed read access to private data. If a client from a non-member org attempts to execute a chaincode function that performs a read of a private data key, the chaincode invocation is terminated with an error. Utilize a value of `false` if you would like to encode more granular access control within individual chaincode functions.
- `memberOnlyWrite`: a value of `true` indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed to write private data from chaincode. If a client from a non-member org attempts to execute a chaincode function that performs a write on a private data key, the

chaincode invocation is terminated with an error. Utilize a value of `false` if you would like to encode more granular access control within individual chaincode functions, for example you may want certain clients from non-member organization to be able to create private data in a certain collection.

- `endorsementPolicy`: An optional endorsement policy to utilize for the collection that overrides the chaincode level endorsement policy. A collection level endorsement policy may be specified in the form of a `signaturePolicy` or may be a `channelConfigPolicy` reference to an existing policy from the channel configuration. The `endorsementPolicy` may be the same as the collection distribution `policy`, or may require fewer or additional organization peers.

Here is a sample collection definition JSON file, containing an array of two collection definitions:

```
[
 {
    "name": "collectionMarbles",
    "policy": "OR('Org1MSP.member', 'Org2MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive":1000000,
    "memberOnlyRead": true,
    "memberOnlyWrite": true
 },
 {
    "name": "collectionMarblePrivateDetails",
    "policy": "OR('Org1MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive":3,
    "memberOnlyRead": true,
    "memberOnlyWrite":true,
    "endorsementPolicy": {
      "signaturePolicy": "OR('Org1MSP.member')"
    }
 }
]
```

This example uses the organizations from the Fabric test network, `Org1` and `Org2`. The policy in the `collectionMarbles` definition authorizes both organizations to the private data. This is a typical configuration when the chaincode data needs to remain private from the ordering service nodes. However, the policy in the `collectionMarblePrivateDetails` definition restricts access to a subset of organizations in the channel (in this case `Org1`). Additionally, writing to this collection requires endorsement from an `Org1` peer, even though the chaincode level endorsement policy may require endorsement from `Org1` or `Org2`. And since "memberOnlyWrite" is true, only clients from `Org1` may invoke chaincode that writes to the private data collection. In this way you can control which organizations are entrusted to write to certain private data collections.

12.8.2 Private data dissemination

Since private data is not included in the transactions that get submitted to the ordering service, and therefore not included in the blocks that get distributed to all peers in a channel, the endorsing peer plays an important role in disseminating private data to other peers of authorized organizations. This ensures the availability of private data in the channel's collection, even if endorsing peers become unavailable after their endorsement. To assist with this dissemination, the `maxPeerCount` and `requiredPeerCount` properties in the collection definition control the degree of dissemination at endorsement time.

If the endorsing peer cannot successfully disseminate the private data to at least the `requiredPeerCount`, it will return an error back to the client. The endorsing peer will attempt to disseminate the private data to peers of different organizations, in an effort to ensure that each authorized organization has a copy of the private data. Since transactions

are not committed at chaincode execution time, the endorsing peer and recipient peers store a copy of the private data in a local `transient store` alongside their blockchain until the transaction is committed.

When authorized peers do not have a copy of the private data in their transient data store at commit time (either because they were not an endorsing peer or because they did not receive the private data via dissemination at endorsement time), they will attempt to pull the private data from another authorized peer, *for a configurable amount of time* based on the peer property `peer.gossip.pvtData.pullRetryThreshold` in the peer configuration `core.yaml` file.

Note: The peers being asked for private data will only return the private data if the requesting peer is a member of the collection as defined by the private data dissemination policy.

Considerations when using `pullRetryThreshold`:

- If the requesting peer is able to retrieve the private data within the `pullRetryThreshold`, it will commit the transaction to its ledger (including the private data hash), and store the private data in its state database, logically separated from other channel state data.
- If the requesting peer is not able to retrieve the private data within the `pullRetryThreshold`, it will commit the transaction to it's blockchain (including the private data hash), without the private data.
- If the peer was entitled to the private data but it is missing, then that peer will not be able to endorse future transactions that reference the missing private data - a chaincode query for a key that is missing will be detected (based on the presence of the key's hash in the state database), and the chaincode will receive an error.

Therefore, it is important to set the `requiredPeerCount` and `maxPeerCount` properties large enough to ensure the availability of private data in your channel. For example, if each of the endorsing peers become unavailable before the transaction commits, the `requiredPeerCount` and `maxPeerCount` properties will have ensured the private data is available on other peers.

Note: For collections to work, it is important to have cross organizational gossip configured correctly. Refer to our documentation on *Gossip data dissemination protocol*, paying particular attention to the "anchor peers" and "external endpoint" configuration.

12.8.3 Referencing collections from chaincode

A set of shim APIs are available for setting and retrieving private data.

The same chaincode data operations can be applied to channel state data and private data, but in the case of private data, a collection name is specified along with the data in the chaincode APIs, for example `PutPrivateData(collection,key,value)` and `GetPrivateData(collection,key)`.

A single chaincode can reference multiple collections.

12.8.4 Referencing implicit collections from chaincode

Starting in v2.0, an implicit private data collection can be used for each organization in a channel, so that you don't have to define collections if you'd like to utilize per-organization collections. Each org-specific implicit collection has a distribution policy and endorsement policy of the matching organization. You can therefore utilize implicit collections for use cases where you'd like to ensure that a specific organization has written to a collection key namespace. The v2.0 chaincode lifecycle uses implicit collections to track which organizations have approved a chaincode definition. Similarly, you can use implicit collections in application chaincode to track which organizations have approved or voted for some change in state.

To write and read an implicit private data collection key, in the `PutPrivateData` and `GetPrivateData` chaincode APIs, specify the collection parameter as `"_implicit_org_<MSPID>"`, for example `"_implicit_org_Org1MSP"`.

Note: Application defined collection names are not allowed to start with an underscore, therefore there is no chance for an implicit collection name to collide with an application defined collection name.

How to pass private data in a chaincode proposal

Since the chaincode proposal gets stored on the blockchain, it is also important not to include private data in the main part of the chaincode proposal. A special field in the chaincode proposal called the `transient` field can be used to pass private data from the client (or data that chaincode will use to generate private data), to chaincode invocation on the peer. The chaincode can retrieve the `transient` field by calling the GetTransient() API. This `transient` field gets excluded from the channel transaction.

Protecting private data content

If the private data is relatively simple and predictable (e.g. transaction dollar amount), channel members who are not authorized to the private data collection could try to guess the content of the private data via brute force hashing of the domain space, in hopes of finding a match with the private data hash on the chain. Private data that is predictable should therefore include a random "salt" that is concatenated with the private data key and included in the private data value, so that a matching hash cannot realistically be found via brute force. The random "salt" can be generated at the client side (e.g. by sampling a secure psuedo-random source) and then passed along with the private data in the transient field at the time of chaincode invocation.

Access control for private data

Until version 1.3, access control to private data based on collection membership was enforced for peers only. Access control based on the organization of the chaincode proposal submitter was required to be encoded in chaincode logic. Collection configuration options `memberOnlyRead` (since version v1.4) and `memberOnlyWrite` (since version v2.0) can automatically enforce that the chaincode proposal submitter must be from a collection member in order to read or write private data keys. For more information about collection configuration definitions and how to set them, refer back to the *Private data collection definition* section of this topic.

Note: If you would like more granular access control, you can set `memberOnlyRead` and `memberOnlyWrite` to false. You can then apply your own access control logic in chaincode, for example by calling the GetCreator() chaincode API or using the client identity chaincode library .

Querying Private Data

Private data collection can be queried just like normal channel data, using shim APIs:

- `GetPrivateDataByRange(collection, startKey, endKey string)`
- `GetPrivateDataByPartialCompositeKey(collection, objectType string, keys []string)`

And for the CouchDB state database, JSON content queries can be passed using the shim API:

- `GetPrivateDataQueryResult(collection, query string)`

Limitations:

- Clients that call chaincode that executes range or rich JSON queries should be aware that they may receive a subset of the result set, if the peer they query has missing private data, based on the explanation in Private Data Dissemination section above. Clients can query multiple peers and compare the results to determine if a peer may be missing some of the result set.
- Chaincode that executes range or rich JSON queries and updates data in a single transaction is not supported, as the query results cannot be validated on the peers that don't have access to the private data, or on peers that are missing the private data that they have access to. If a chaincode invocation both queries and updates private data, the proposal request will return an error. If your application can tolerate result set changes between chaincode execution and validation/commit time, then you could call one chaincode function to perform the query, and then call a second chaincode function to make the updates. Note that calls to GetPrivateData() to retrieve individual keys can be made in the same transaction as PutPrivateData() calls, since all peers can validate key reads based on the hashed key version.

Using Indexes with collections

The topic *CouchDB as the State Database* describes indexes that can be applied to the channel's state database to enable JSON content queries, by packaging indexes in a `META-INF/statedb/couchdb/indexes` directory at chaincode installation time. Similarly, indexes can also be applied to private data collections, by packaging indexes in a `META-INF/statedb/couchdb/collections/<collection_name>/indexes` directory. An example index is available here.

12.8.5 Considerations when using private data

Private data purging

Private data can be periodically purged from peers. For more details, see the `blockToLive` collection definition property above.

Additionally, recall that prior to commit, peers store private data in a local transient data store. This data automatically gets purged when the transaction commits. But if a transaction was never submitted to the channel and therefore never committed, the private data would remain in each peer's transient store. This data is purged from the transient store after a configurable number blocks by using the peer's `peer.gossip.pvtData.transientstoreMaxBlockRetention` property in the peer `core.yaml` file.

Updating a collection definition

To update a collection definition or add a new collection, you can update the chaincode definition and pass the new collection configuration in the chaincode approve and commit transactions, for example using the `--collections-config` flag if using the CLI. If a collection configuration is specified when updating the chaincode definition, a definition for each of the existing collections must be included.

When updating a chaincode definition, you can add new private data collections, and update existing private data collections, for example to add new members to an existing collection or change one of the collection definition properties. Note that you cannot update the collection name or the blockToLive property, since a consistent blockToLive is required regardless of a peer's block height.

Collection updates becomes effective when a peer commits the block with the updated chaincode definition. Note that collections cannot be deleted, as there may be prior private data hashes on the channel's blockchain that cannot be removed.

Private data reconciliation

Starting in v1.4, peers of organizations that are added to an existing collection will automatically fetch private data that was committed to the collection before they joined the collection.

This private data "reconciliation" also applies to peers that were entitled to receive private data but did not yet receive it — because of a network failure, for example — by keeping track of private data that was "missing" at the time of block commit.

Private data reconciliation occurs periodically based on the `peer.gossip.pvtData.reconciliationEnabled` and `peer.gossip.pvtData.reconcileSleepInterval` properties in core.yaml. The peer will periodically attempt to fetch the private data from other collection member peers that are expected to have it.

Note that this private data reconciliation feature only works on peers running v1.4 or later of Fabric.

12.9 Read-Write set semantics

This document discusses the details of the current implementation about the semantics of read-write sets.

12.9.1 Transaction simulation and read-write set

During simulation of a transaction at an `endorser`, a read-write set is prepared for the transaction. The `read set` contains a list of unique keys and their committed version numbers (but not values) that the transaction reads during simulation. The `write set` contains a list of unique keys (though there can be overlap with the keys present in the read set) and their new values that the transaction writes. A delete marker is set (in the place of new value) for the key if the update performed by the transaction is to delete the key.

Further, if the transaction writes a value multiple times for a key, only the last written value is retained. Also, if a transaction reads a value for a key, the value in the committed state is returned even if the transaction has updated the value for the key before issuing the read. In another words, Read-your-writes semantics are not supported.

As noted earlier, the versions of the keys are recorded only in the read set; the write set just contains the list of unique keys and their latest values set by the transaction.

There could be various schemes for implementing versions. The minimal requirement for a versioning scheme is to produce non-repeating identifiers for a given key. For instance, using monotonically increasing numbers for versions can be one such scheme. In the current implementation, we use a blockchain height based versioning scheme in which the height of the committing transaction is used as the latest version for all the keys modified by the transaction. In this scheme, the height of a transaction is represented by a tuple (txNumber is the height of the transaction within the block). This scheme has many advantages over the incremental number scheme - primarily, it enables other components such as statedb, transaction simulation and validation to make efficient design choices.

Following is an illustration of an example read-write set prepared by simulation of a hypothetical transaction. For the sake of simplicity, in the illustrations, we use the incremental numbers for representing the versions.

```
<TxReadWriteSet>
  <NsReadWriteSet name="chaincode1">
    <read-set>
      <read key="K1", version="1">
      <read key="K2", version="1">
    </read-set>
    <write-set>
      <write key="K1", value="V1">
      <write key="K3", value="V2">
```

(continues on next page)

(continued from previous page)

```
      <write key="K4", isDelete="true">
    </write-set>
  </NsReadWriteSet>
<TxReadWriteSet>
```

Additionally, if the transaction performs a range query during simulation, the range query as well as its results will be added to the read-write set as `query-info`.

12.9.2 Transaction validation and updating world state using read-write set

A `committer` uses the read set portion of the read-write set for checking the validity of a transaction and the write set portion of the read-write set for updating the versions and the values of the affected keys.

In the validation phase, a transaction is considered `valid` if the version of each key present in the read set of the transaction matches the version for the same key in the world state - assuming all the preceding `valid` transactions (including the preceding transactions in the same block) are committed (*committed-state*). An additional validation is performed if the read-write set also contains one or more query-info.

This additional validation should ensure that no key has been inserted/deleted/updated in the super range (i.e., union of the ranges) of the results captured in the query-info(s). In other words, if we re-execute any of the range queries (that the transaction performed during simulation) during validation on the committed-state, it should yield the same results that were observed by the transaction at the time of simulation. This check ensures that if a transaction observes phantom items during commit, the transaction should be marked as invalid. Note that the this phantom protection is limited to range queries (i.e., `GetStateByRange` function in the chaincode) and not yet implemented for other queries (i.e., `GetQueryResult` function in the chaincode). Other queries are at risk of phantoms, and should therefore only be used in read-only transactions that are not submitted to ordering, unless the application can guarantee the stability of the result set between simulation and validation/commit time.

If a transaction passes the validity check, the committer uses the write set for updating the world state. In the update phase, for each key present in the write set, the value in the world state for the same key is set to the value as specified in the write set. Further, the version of the key in the world state is changed to reflect the latest version.

12.9.3 Example simulation and validation

This section helps with understanding the semantics through an example scenario. For the purpose of this example, the presence of a key, `k`, in the world state is represented by a tuple `(k,ver,val)` where `ver` is the latest version of the key `k` having `val` as its value.

Now, consider a set of five transactions `T1, T2, T3, T4, and T5`, all simulated on the same snapshot of the world state. The following snippet shows the snapshot of the world state against which the transactions are simulated and the sequence of read and write activities performed by each of these transactions.

```
World state: (k1,1,v1), (k2,1,v2), (k3,1,v3), (k4,1,v4), (k5,1,v5)
T1 -> Write(k1, v1'), Write(k2, v2')
T2 -> Read(k1), Write(k3, v3')
T3 -> Write(k2, v2'')
T4 -> Write(k2, v2'''), read(k2)
T5 -> Write(k6, v6'), read(k5)
```

Now, assume that these transactions are ordered in the sequence of T1,..,T5 (could be contained in a single block or different blocks)

1. `T1` passes validation because it does not perform any read. Further, the tuple of keys `k1` and `k2` in the world state are updated to `(k1,2,v1'), (k2,2,v2')`

2. `T2` fails validation because it reads a key, `k1`, which was modified by a preceding transaction - `T1`
3. `T3` passes the validation because it does not perform a read. Further the tuple of the key, `k2`, in the world state is updated to `(k2,3,v2'')`
4. `T4` fails the validation because it reads a key, `k2`, which was modified by a preceding transaction `T1`
5. `T5` passes validation because it reads a key, `k5,` which was not modified by any of the preceding transactions

Note: Transactions with multiple read-write sets are not yet supported.

12.10 Gossip data dissemination protocol

Hyperledger Fabric optimizes blockchain network performance, security, and scalability by dividing workload across transaction execution (endorsing and committing) peers and transaction ordering nodes. This decoupling of network operations requires a secure, reliable and scalable data dissemination protocol to ensure data integrity and consistency. To meet these requirements, Fabric implements a **gossip data dissemination protocol**.

12.10.1 Gossip protocol

Peers leverage gossip to broadcast ledger and channel data in a scalable fashion. Gossip messaging is continuous, and each peer on a channel is constantly receiving current and consistent ledger data from multiple peers. Each gossiped message is signed, thereby allowing Byzantine participants sending faked messages to be easily identified and the distribution of the message(s) to unwanted targets to be prevented. Peers affected by delays, network partitions, or other causes resulting in missed blocks will eventually be synced up to the current ledger state by contacting peers in possession of these missing blocks.

The gossip-based data dissemination protocol performs three primary functions on a Fabric network:

1. Manages peer discovery and channel membership, by continually identifying available member peers, and eventually detecting peers that have gone offline.
2. Disseminates ledger data across all peers on a channel. Any peer with data that is out of sync with the rest of the channel identifies the missing blocks and syncs itself by copying the correct data.
3. Bring newly connected peers up to speed by allowing peer-to-peer state transfer update of ledger data.

Gossip-based broadcasting operates by peers receiving messages from other peers on the channel, and then forwarding these messages to a number of randomly selected peers on the channel, where this number is a configurable constant. Peers can also exercise a pull mechanism rather than waiting for delivery of a message. This cycle repeats, with the result of channel membership, ledger and state information continually being kept current and in sync. For dissemination of new blocks, the **leader** peer on the channel pulls the data from the ordering service and initiates gossip dissemination to peers in its own organization.

12.10.2 Leader election

The leader election mechanism is used to **elect** one peer per organization which will maintain connection with the ordering service and initiate distribution of newly arrived blocks across the peers of its own organization. Leveraging leader election provides the system with the ability to efficiently utilize the bandwidth of the ordering service. There are two possible modes of operation for a leader election module:

1. **Static** — a system administrator manually configures a peer in an organization to be the leader.
2. **Dynamic** — peers execute a leader election procedure to select one peer in an organization to become leader.

Static leader election

Static leader election allows you to manually define one or more peers within an organization as leader peers. Please note, however, that having too many peers connect to the ordering service may result in inefficient use of bandwidth. To enable static leader election mode, configure the following parameters within the section of `core.yaml`:

```
peer:
    # Gossip related configuration
    gossip:
        useLeaderElection: false
        orgLeader: true
```

Alternatively these parameters could be configured and overridden with environmental variables:

```
export CORE_PEER_GOSSIP_USELEADERELECTION=false
export CORE_PEER_GOSSIP_ORGLEADER=true
```

Note: The following configuration will keep peer in **stand-by** mode, i.e. peer will not try to become a leader:

```
export CORE_PEER_GOSSIP_USELEADERELECTION=false
export CORE_PEER_GOSSIP_ORGLEADER=false
```

2. Setting `CORE_PEER_GOSSIP_USELEADERELECTION` and `CORE_PEER_GOSSIP_ORGLEADER` with `true` value is ambiguous and will lead to an error.
3. In static configuration organization admin is responsible to provide high availability of the leader node in case for failure or crashes.

Dynamic leader election

Dynamic leader election enables organization peers to **elect** one peer which will connect to the ordering service and pull out new blocks. This leader is elected for an organization's peers independently.

A dynamically elected leader sends **heartbeat** messages to the rest of the peers as an evidence of liveness. If one or more peers don't receive **heartbeats** updates during a set period of time, they will elect a new leader.

In the worst case scenario of a network partition, there will be more than one active leader for organization to guarantee resiliency and availability to allow an organization's peers to continue making progress. After the network partition has been healed, one of the leaders will relinquish its leadership. In a steady state with no network partitions, there will be **only** one active leader connecting to the ordering service.

Following configuration controls frequency of the leader **heartbeat** messages:

```
peer:
    # Gossip related configuration
    gossip:
        election:
            leaderAliveThreshold: 10s
```

In order to enable dynamic leader election, the following parameters need to be configured within `core.yaml`:

```
peer:
    # Gossip related configuration
    gossip:
        useLeaderElection: true
        orgLeader: false
```

Alternatively these parameters could be configured and overridden with environment variables:

```
export CORE_PEER_GOSSIP_USELEADERELECTION=true
export CORE_PEER_GOSSIP_ORGLEADER=false
```

12.10.3 Anchor peers

Anchor peers are used by gossip to make sure peers in different organizations know about each other.

When a configuration block that contains an update to the anchor peers is committed, peers reach out to the anchor peers and learn from them about all of the peers known to the anchor peer(s). Once at least one peer from each organization has contacted an anchor peer, the anchor peer learns about every peer in the channel. Since gossip communication is constant, and because peers always ask to be told about the existence of any peer they don't know about, a common view of membership can be established for a channel.

For example, let's assume we have three organizations—*A*, *B*, *C*— in the channel and a single anchor peer—*peer0.orgC*— defined for organization *C*. When *peer1.orgA* (from organization *A*) contacts *peer0.orgC*, it will tell it about *peer0.orgA*. And when at a later time *peer1.orgB* contacts *peer0.orgC*, the latter would tell the former about *peer0.orgA*. From that point forward, organizations *A* and *B* would start exchanging membership information directly without any assistance from *peer0.orgC*.

As communication across organizations depends on gossip in order to work, there must be at least one anchor peer defined in the channel configuration. It is strongly recommended that every organization provides its own set of anchor peers for high availability and redundancy. Note that the anchor peer does not need to be the same peer as the leader peer.

External and internal endpoints

In order for gossip to work effectively, peers need to be able to obtain the endpoint information of peers in their own organization as well as from peers in other organizations.

When a peer is bootstrapped it will use `peer.gossip.bootstrap` in its `core.yaml` to advertise itself and exchange membership information, building a view of all available peers within its own organization.

The `peer.gossip.bootstrap` property in the `core.yaml` of the peer is used to bootstrap gossip **within an organization**. If you are using gossip, you will typically configure all the peers in your organization to point to an initial set of bootstrap peers (you can specify a space-separated list of peers). The internal endpoint is usually auto-computed by the peer itself or just passed explicitly via `core.peer.address` in `core.yaml`. If you need to overwrite this value, you can export `CORE_PEER_GOSSIP_ENDPOINT` as an environment variable.

Bootstrap information is similarly required to establish communication **across organizations**. The initial cross-organization bootstrap information is provided via the "anchor peers" setting described above. If you want to make other peers in your organization known to other organizations, you need to set the `peer.gossip.externalendpoint` in the `core.yaml` of your peer. If this is not set, the endpoint information of the peer will not be broadcast to peers in other organizations.

To set these properties, issue:

```
export CORE_PEER_GOSSIP_BOOTSTRAP=<a list of peer endpoints within the peer's org>
export CORE_PEER_GOSSIP_EXTERNALENDPOINT=<the peer endpoint, as known outside the org>
```

12.10.4 Gossip messaging

Online peers indicate their availability by continually broadcasting "alive" messages, with each containing the **public key infrastructure (PKI)** ID and the signature of the sender over the message. Peers maintain channel membership by

collecting these alive messages; if no peer receives an alive message from a specific peer, this "dead" peer is eventually purged from channel membership. Because "alive" messages are cryptographically signed, malicious peers can never impersonate other peers, as they lack a signing key authorized by a root certificate authority (CA).

In addition to the automatic forwarding of received messages, a state reconciliation process synchronizes **world state** across peers on each channel. Each peer continually pulls blocks from other peers on the channel, in order to repair its own state if discrepancies are identified. Because fixed connectivity is not required to maintain gossip-based data dissemination, the process reliably provides data consistency and integrity to the shared ledger, including tolerance for node crashes.

Because channels are segregated, peers on one channel cannot message or share information on any other channel. Though any peer can belong to multiple channels, partitioned messaging prevents blocks from being disseminated to peers that are not in the channel by applying message routing policies based on a peers' channel subscriptions.

Note: 1. Security of point-to-point messages are handled by the peer TLS layer, and do not require signatures. Peers are authenticated by their certificates, which are assigned by a CA. Although TLS certs are also used, it is the peer certificates that are authenticated in the gossip layer. Ledger blocks are signed by the ordering service, and then delivered to the leader peers on a channel.

2. Authentication is governed by the membership service provider for the peer. When the peer connects to the channel for the first time, the TLS session binds with the membership identity. This essentially authenticates each peer to the connecting peer, with respect to membership in the network and channel.

CHAPTER 13

Frequently Asked Questions

13.1 Endorsement

Endorsement architecture:

Question How many peers in the network need to endorse a transaction?

Answer The number of peers required to endorse a transaction is driven by the endorsement policy that is specified in the chaincode definition.

Question Does an application client need to connect to all peers?

Answer Clients only need to connect to as many peers as are required by the endorsement policy for the chaincode.

13.2 Security & Access Control

Question How do I ensure data privacy?

Answer There are various aspects to data privacy. First, you can segregate your network into channels, where each channel represents a subset of participants that are authorized to see the data for the chaincodes that are deployed to that channel.

Second, you can use private-data to keep ledger data private from other organizations on the channel. A private data collection allows a defined subset of organizations on a channel the ability to endorse, commit, or query private data without having to create a separate channel. Other participants on the channel receive only a hash of the data. For more information refer to the *Using Private Data in Fabric* tutorial. Note that the key concepts topic also explains when to use private data instead of a channel.

Third, as an alternative to Fabric hashing the data using private data, the client application can hash or encrypt the data before calling chaincode. If you hash the data then you will need to provide a means to share the source data. If you encrypt the data then you will need to provide a means to share the decryption keys.

Fourth, you can restrict data access to certain roles in your organization, by building access control into the chaincode logic.

Fifth, ledger data at rest can be encrypted via file system encryption on the peer, and data in-transit is encrypted via TLS.

Question Do the orderers see the transaction data?

Answer No, the orderers only order transactions, they do not open the transactions. If you do not want the data to go through the orderers at all, then utilize the private data feature of Fabric. Alternatively, you can hash or encrypt the data in the client application before calling chaincode. If you encrypt the data then you will need to provide a means to share the decryption keys.

13.3 Application-side Programming Model

Question How do application clients know the outcome of a transaction?

Answer The transaction simulation results are returned to the client by the endorser in the proposal response. If there are multiple endorsers, the client can check that the responses are all the same, and submit the results and endorsements for ordering and commitment. Ultimately the committing peers will validate or invalidate the transaction, and the client becomes aware of the outcome via an event, that the SDK makes available to the application client.

Question How do I query the ledger data?

Answer Within chaincode you can query based on keys. Keys can be queried by range, and composite keys can be modeled to enable equivalence queries against multiple parameters. For example a composite key of (owner,asset_id) can be used to query all assets owned by a certain entity. These key-based queries can be used for read-only queries against the ledger, as well as in transactions that update the ledger.

If you model asset data as JSON in chaincode and use CouchDB as the state database, you can also perform complex rich queries against the chaincode data values, using the CouchDB JSON query language within chaincode. The application client can perform read-only queries, but these responses are not typically submitted as part of transactions to the ordering service.

Question How do I query the historical data to understand data provenance?

Answer The chaincode API `GetHistoryForKey()` will return history of values for a key.

Question How to guarantee the query result is correct, especially when the peer being queried may be recovering and catching up on block processing?

Answer The client can query multiple peers, compare their block heights, compare their query results, and favor the peers at the higher block heights.

13.4 Chaincode (Smart Contracts and Digital Assets)

Question Does Hyperledger Fabric support smart contract logic?

Answer Yes. We call this feature *Chaincode*. It is our interpretation of the smart contract method/algorithm, with additional features.

A chaincode is programmatic code deployed on the network, where it is executed and validated by chain validators together during the consensus process. Developers can use chaincodes to develop business contracts, asset definitions, and collectively-managed decentralized applications.

Question How do I create a business contract?

Answer There are generally two ways to develop business contracts: the first way is to code individual contracts into standalone instances of chaincode; the second way, and probably the more efficient way, is to use chaincode to create decentralized applications that manage the life cycle of one or multiple types of business contracts, and let end users instantiate instances of contracts within these applications.

Question How do I create assets?

Answer Users can use chaincode (for business rules) and membership service (for digital tokens) to design assets, as well as the logic that manages them.

There are two popular approaches to defining assets in most blockchain solutions: the stateless UTXO model, where account balances are encoded into past transaction records; and the account model, where account balances are kept in state storage space on the ledger.

Each approach carries its own benefits and drawbacks. This blockchain technology does not advocate either one over the other. Instead, one of our first requirements was to ensure that both approaches can be easily implemented.

Question Which languages are supported for writing chaincode?

Answer Chaincode can be written in any programming language and executed in containers. Currently, Go, Node.js and Java chaincode are supported.

Question Does the Hyperledger Fabric have native currency?

Answer No. However, if you really need a native currency for your chain network, you can develop your own native currency with chaincode. One common attribute of native currency is that some amount will get transacted (the chaincode defining that currency will get called) every time a transaction is processed on its chain.

13.5 Differences in Most Recent Releases

Question Where can I find what are the highlighted differences between releases?

Answer The differences between any subsequent releases are provided together with the *Releases*.

Question Where to get help for the technical questions not answered above?

Answer Please use StackOverflow.

13.6 Ordering Service

Question **I have an ordering service up and running and want to switch consensus algorithms. How do I do that?**

Answer This is explicitly not supported.

Question **What is the orderer system channel?**

Answer The orderer system channel (sometimes called ordering system channel) is the channel the orderer is initially bootstrapped with. It is used to orchestrate channel creation. The orderer system channel defines consortia and the initial configuration for new channels. At channel creation time, the organization definition in the consortium, the `/Channel` group's values and policies, as well as the `/Channel/Orderer` group's values and policies, are all combined to form the new initial channel definition.

Question **If I update my application channel, should I update my orderer system channel?**

Answer Once an application channel is created, it is managed independently of any other channel (including the orderer system channel). Depending on the modification, the change may or may not be desirable to port to other channels. In general, MSP changes should be synchronized across all channels, while policy changes are more likely to be specific to a particular channel.

Question Can I have an organization act both in an ordering and application role?

Answer Although this is possible, it is a highly discouraged configuration. By default the `/Channel/Orderer/BlockValidation` policy allows any valid certificate of the ordering organizations to sign blocks. If an organization is acting both in an ordering and application role, then this policy should be updated to restrict block signers to the subset of certificates authorized for ordering.

Question I want to write a consensus implementation for Fabric. Where do I begin?

Answer A consensus plugin needs to implement the `Consenter` and `Chain` interfaces defined in the consensus package. There is a plugin built against raft . You can study it to learn more for your own implementation. The ordering service code can be found under the orderer package.

Question I want to change my ordering service configurations, e.g. batch timeout, after I start the network, what should I do?

Answer This falls under reconfiguring the network. Please consult the topic on *configtxlator*.

13.6.1 BFT

Question When is a BFT version of the ordering service going to be available?

Answer No date has been set. We are working towards a release during the 2.x cycle, i.e. it will come with a minor version upgrade in Fabric.

Note: Users who are migrating from Gerrit to GitHub: You can follow simple Git workflows to move your development from Gerrit to GitHub. After forking the Fabric repo, simply push the branches you want to save from your current Gerrit-based local repo to your remote forked repository. Once you've pushed the changes you want to save, simply delete your local Gerrit-based repository and clone your fork.

For a basic Git workflow recommendation please see our doc at github/github.

CHAPTER 14

Contributions Welcome!

14.1 Documentation style guide for contributors

Audience: documentation writers and editors

While this style guide will also refer to best practices using ReStructured Text (also known as RST), in general we advise writing documentation in Markdown, as it's a more universally accepted documentation standard. Both formats are usable, however, and if you decide to write a topic in RST (or are editing an RST topic), be sure to refer to this style guide.

When in doubt, use the docs themselves for guidance on how to format things.

- For RST formatting.
- For Markdown formatting.

If you just want to look at how things are formatted, you can navigate to the Fabric repo to look at the raw file by clicking on `Edit on Github` link in the upper right hand corner of the page. Then click the `Raw` tab. This will show you the formatting of the doc. **Do not attempt to edit the file on Github.** If you want to make a change, clone the repo and follow the instructions in Contributing for creating pull requests.

Line lengths.

If you look at the raw versions of the documentation, you will see that many topics conform to a line length of roughly 70 characters. This restriction is no longer necessary, so you are free to make lines as long as you want.

Tutorials should have a list of steps at the top.

A list of steps (with links to the corresponding sections) at the beginning of a tutorial helps users find particular steps they're interested in. For an example, check out Use private data in Fabric.

"Fabric", "Hyperledger Fabric" or "HLF"?

The first usage should be "Hyperledger Fabric" and afterwards only "Fabric". Don't use "HLF" or "Hyperledger" by itself.

Chaincode vs. Chaincodes?

One chaincode is a "chaincode". If you're talking about several chaincodes, use "chaincodes".

When to bold?

Not too often. The best use of them is either as a summary or as a way of drawing attention to concepts you want to talk about. "A blockchain network contains a ledger, at least one chaincode, and peers", especially if you're going to be talking about those things in that paragraph. Avoid using them simply to emphasize a single word, as in something like "Blockchain networks **must** use propery security protocols".

When to surround something in back tics `nnn`?

This is useful to draw attention to words that either don't make sense in plain English or when referencing parts of the code (especially if you've put code snippets in your doc). So for example, when talking about the fabric-samples directory, surround `fabric-samples` with back tics. Same with a code function like `hf.Revoker`. It might also make sense to put back tics around words that do make sense in plain English that are part of the code if you're referencing them in a code context. For example, when referencing an `attribute` as part of an Access Control List.

Is it ever appropriate to use a dash?

Dashes can be incredibly useful but they're not necessarily as technically appropriate as using separate declarative sentences. Let's consider this example sentence:

```
This leaves us with a trimmed down JSON object --- config.json, located in the fabric-
↪samples folder inside first-network --- which will serve as the baseline for our␣
↪config update.
```

There are a number of ways to present this same information, but in this case the dashes break up the information while keeping it as part of the same thought. If you use a dash, make sure to use the "em" dash, which is three times longer than a hyphen. These dashes should have a space before and after them.

When to use hyphens?

Hyphens are mostly commonly used as part of a "compound adjective". For example, "jet-powered car". Note that the compound adjective must immediately precede the noun being modified. In other words, "jet powered" does not by itself need a hyphen. When in doubt, use Google, as compound adjectives are tricky and are a popular discussion on grammar discussion boards.

How many spaces after a period?

One.

How should numbers be rendered?

Number zero through nine are spelled out. One, two, three, four, etc. Numbers after 10 are rendered as numbers.

Exceptions to this would be usages from code. In that case, use whatever's in the code. And also examples like Org1. Don't write it as OrgOne.

Try to avoid using the same words too often.

If you can avoid using a word twice in one sentence, please do so. Not using it more than twice in a single paragraph is better. Of course sometimes it might not be possible to avoid this — a doc about stateDB is likely to be replete with uses of the word "database" or "ledger". But excessive usage of any particular word has a tendency to have a numbing effect on the reader.

How should files be named?

For example, `identity_use_case_tutorial.md`. While not all files use this standard, new files should adhere to it.

Capitalization rules for doc titles.

The standard rules for capitalization in sentences should be followed. In other words, unless a word is the first word in the title or a proper noun, do not capitalize its first letter. For example, "Understanding Identities in Fabric" should

be "Understanding identities in Fabric". While not every doc follows this standard yet, it is the standard we're moving to and should be followed for new topics.

Headings inside of topics should follow the same standard.

Use the Oxford comma?

Yes, it's better.

For example: "I'd like to thank my parents, Ayn Rand and God". Compared to: "I'd like to thank my parents, Ayn Rand, and God."

JSON vs .json?

Use "JSON". The same applies for any file format (for example, YAML).

Referring to the reader.

It's perfectly fine to use the word "you" or "we".

Ampersands (&).

Not a substitute for the word "and". Avoid them unless you have a reason to use it (such as in a code snippet that includes it).

Acronyms.

The first usage of an acronym should be spelled out, unless it's an acronym that's in such wide usage this is unneeded. For example, "Software Development Kit (SDK)" on first usage. Then use "SDK" afterward.

Captions.

These should be in italics, and it's the only real valid use for italics in our docs.

Commands.

In general, put each command in its own snippet. It reads better, especially when commands are long. An exception to this rule is when suggesting the export of a number of environment variables.

Code snippets.

In Markdown, if you want to post sample code, use three back tics to set off the snippet. For example:

```
Code goes here.

Even more code goes here.

And still more.
```

In RST, you will need to set off the code snippet using formatting similar to this:

```
.. code:: bash

   Code goes here.
```

You can substitute `bash` for a language like Java or Go, where appropriate.

Enumerated lists in markdown.

Note that in Markdown, enumerated lists will not work if you separate the numbers with a space. Markdown sees this as the start of a new list, not a continuation of the old one (every number will be `1.`). If you need an enumerated list, you will have to use RST. Bulleted lists are a good substitute in Markdown, and are the recommended alternative.

Linking.

When linking to another doc, use relative links, not direct links. When naming a link, do not just call it "link". Use a more creative and descriptive name. For accessibility reasons, the link name should also make it clear that it is a link. For example, check out this link to Google.

curl vs cURL.

The tool is called "cURL". The commands themselves are "curl" commands.

Fabric CA.

Do not call it "fabric-CA", "fabricCA", or FabricCA. It is the Fabric CA.

Raft and RAFT.

"Raft" is not an acronym. Do not call it a "RAFT ordering service".

All docs have to end with a license statement.

In RST, it's this:

```
.. Licensed under Creative Commons Attribution 4.0 International License
   https://creativecommons.org/licenses/by/4.0/
```

In markdown:

```
<!--- Licensed under Creative Commons Attribution 4.0 International License
https://creativecommons.org/licenses/by/4.0/ -->
```

How many spaces for indentation?

Depends on the use case, I suppose. Frequently it's necessary, especially in RST, to indent two spaces, especially in a code block. In a NOTE box in RST, you have to indent to the space after the colon after note, like this:

```
.. note:: Some words and stuff etc etc etc (line continues until the 70 character␣
↪limit line)
          the line directly below has to start at the same space as the one above.
```

When to use which type of heading.

Check out this topic on headings.

In RST, use this:

```
Chapter 1 Title
===============

Section 1.1 Title
-----------------

Subsection 1.1.1 Title
~~~~~~~~~~~~~~~~~~~~~~

Section 1.2 Title
-----------------
```

Note that the length of what's under the title has to be the same as the length of the title itself. This isn't a problem in Atom, which gives each character the same width by default (this is called "monospacing", if you're ever on Jeopardy! and need that information.

In markdown, it's somewhat simpler. You go:

```
# The Name of the Doc (this will get pulled for the TOC).

## First subsection

## Second subsection
```

Both file formats don't like when these things are done out of order. For example, you might want a `####` to be the first thing after your `#` Title. Markdown won't allow it. Similarly, RST will default to whatever order you give to the title formats (as they appear in the first sections of your doc).

We welcome contributions to Hyperledger in many forms, and there's always plenty to do!

First things first, please review the Hyperledger Code of Conduct before participating. It is important that we keep things civil.

14.2 Ways to contribute

There are many ways you can contribute to Hyperledger Fabric, both as a user and as a developer.

As a user:

- *Making Feature/Enhancement Proposals*
- *Reporting bugs*
- Help test an upcoming Epic on the release roadmap. Contact the Epic assignee via the Jira work item or on RocketChat.

As a developer:

- If you only have a little time, consider picking up a "help-wanted" task, see *Fixing issues and working stories*.
- If you can commit to full-time development, either propose a new feature (see *Making Feature/Enhancement Proposals*) and bring a team to implement it, or join one of the teams working on an existing Epic. If you see an Epic that interests you on the release roadmap, contact the Epic assignee via the Jira work item or on RocketChat.

14.3 Getting a Linux Foundation account

In order to participate in the development of the Hyperledger Fabric project, you will need a Linux Foundation account. Once you have a LF ID you will be able to access all the Hyperledger community tools, including Jira issue management, RocketChat, and the Wiki (for editing, only).

Follow the steps below to create a Linux Foundation account if you don't already have one.

1. Go to the Linux Foundation ID website.
2. Select the option `I need to create a Linux Foundation ID`, and fill out the form that appears.
3. Wait a few minutes, then look for an email message with the subject line: "Validate your Linux Foundation ID email".
4. Open the received URL to validate your email address.
5. Verify that your browser displays the message `You have successfully validated your e-mail address`.
6. Access Jira issue management, or RocketChat.

14.4 Project Governance

Hyperledger Fabric is managed under an open governance model as described in our charter. Projects and sub-projects are lead by a set of maintainers. New sub-projects can designate an initial set of maintainers that will be approved by the top-level project's existing maintainers when the project is first approved.

14.4.1 Maintainers

The Fabric project is lead by the project's top level maintainers. The maintainers are responsible for reviewing and merging all patches submitted for review, and they guide the overall technical direction of the project within the guidelines established by the Hyperledger Technical Steering Committee (TSC).

14.4.2 Becoming a maintainer

The project's maintainers will, from time-to-time, consider adding or removing a maintainer. An existing maintainer can submit a change set to the maintainers file. A nominated Contributor may become a Maintainer by a majority approval of the proposal by the existing Maintainers. Once approved, the change set is then merged and the individual is added to (or alternatively, removed from) the maintainers group. Maintainers may be removed by explicit resignation, for prolonged inactivity (3 or more months), or for some infraction of the code of conduct or by consistently demonstrating poor judgement. A maintainer removed for inactivity should be restored following a sustained resumption of contributions and reviews (a month or more) demonstrating a renewed commitment to the project.

14.4.3 Release cadence

The Fabric maintainers have settled on a quarterly (approximately) release cadence (see releases). At any given time, there will be a stable LTS (long term support) release branch, as well as the master branch for upcoming new features. Follow the discussion on the #fabric-release channel in RocketChat.

14.4.4 Making Feature/Enhancement Proposals

First, take time to review JIRA to be sure that there isn't already an open (or recently closed) proposal for the same function. If there isn't, to make a proposal we recommend that you open a JIRA Epic or Story, whichever seems to best fit the circumstance and link or inline a "one pager" of the proposal that states what the feature would do and, if possible, how it might be implemented. It would help also to make a case for why the feature should be added, such as identifying specific use case(s) for which the feature is needed and a case for what the benefit would be should the feature be implemented. Once the JIRA issue is created, and the "one pager" either attached, inlined in the description field, or a link to a publicly accessible document is added to the description, send an introductory email to the fabric@lists.hyperledger.org mailing list linking the JIRA issue, and soliciting feedback.

Discussion of the proposed feature should be conducted in the JIRA issue itself, so that we have a consistent pattern within our community as to where to find design discussion.

Getting the support of three or more of the Hyperledger Fabric maintainers for the new feature will greatly enhance the probability that the feature's related PRs will be included in a subsequent release.

14.4.5 Maintainers meeting

The maintainers hold regular maintainers meetings. The purpose of the maintainers meeting is to plan for and review the progress of releases, and to discuss the technical and operational direction of the project and sub-projects.

Please see the wiki for maintainer meeting details.

New feature/enhancement proposals as described above should be presented to a maintainers meeting for consideration, feedback and acceptance.

14.4.6 Release roadmap

The Fabric release roadmap of epics is maintained in JIRA.

14.4.7 Communications

We use RocketChat for communication and Google Hangouts™ for screen sharing between developers. Our development planning and prioritization is done in JIRA, and we take longer running discussions/decisions to the mailing list.

14.5 Contribution guide

14.5.1 Install prerequisites

Before we begin, if you haven't already done so, you may wish to check that you have all the *prerequisites* installed on the platform(s) on which you'll be developing blockchain applications and/or operating Hyperledger Fabric.

14.5.2 Getting help

If you are looking for something to work on, or need some expert assistance in debugging a problem or working out a fix to an issue, our community is always eager to help. We hang out on Chat, IRC (#hyperledger on freenode.net) and the mailing lists. Most of us don't bite :grin: and will be glad to help. The only silly question is the one you don't ask. Questions are in fact a great way to help improve the project as they highlight where our documentation could be clearer.

14.5.3 Reporting bugs

If you are a user and you have found a bug, please submit an issue using JIRA. Before you create a new JIRA issue, please try to search the existing items to be sure no one else has previously reported it. If it has been previously reported, then you might add a comment that you also are interested in seeing the defect fixed.

Note: If the defect is security-related, please follow the Hyperledger security bug reporting process.

If it has not been previously reported, you may either submit a PR with a well documented commit message describing the defect and the fix, or you may create a new JIRA. Please try to provide sufficient information for someone else to reproduce the issue. One of the project's maintainers should respond to your issue within 24 hours. If not, please bump the issue with a comment and request that it be reviewed. You can also post to the relevant Hyperledger Fabric channel in Hyperledger Chat. For example, a doc bug should be broadcast to `#fabric-documentation`, a database bug to `#fabric-ledger`, and so on...

14.5.4 Submitting your fix

If you just submitted a JIRA for a bug you've discovered, and would like to provide a fix, we would welcome that gladly! Please assign the JIRA issue to yourself, then submit a pull request (PR). Please refer to github/github for a detailed workflow.

14.5.5 Fixing issues and working stories

Review the issues list and find something that interests you. You could also check the "help-wanted" list. It is wise to start with something relatively straight forward and achievable, and that no one is already assigned. If no one is assigned, then assign the issue to yourself. Please be considerate and rescind the assignment if you cannot finish in a reasonable time, or add a comment saying that you are still actively working the issue if you need a little more time.

14.5.6 Reviewing submitted Pull Requests (PRs)

Another way to contribute and learn about Hyperledger Fabric is to help the maintainers with the review of the PRs that are open. Indeed maintainers have the difficult role of having to review all the PRs that are being submitted and evaluate whether they should be merged or not. You can review the code and/or documentation changes, test the changes, and tell the submitters and maintainers what you think. Once your review and/or test is complete just reply to the PR with your findings, by adding comments and/or voting. A comment saying something like "I tried it on system X and it works" or possibly "I got an error on system X: xxx " will help the maintainers in their evaluation. As a result, maintainers will be able to process PRs faster and everybody will gain from it.

Just browse through the open PRs on GitHub to get started.

14.5.7 PR Aging

As the Fabric project has grown, so too has the backlog of open PRs. One problem that nearly all projects face is effectively managing that backlog and Fabric is no exception. In an effort to keep the backlog of Fabric and related project PRs manageable, we are introducing an aging policy which will be enforced by bots. This is consistent with how other large projects manage their PR backlog.

14.5.8 PR Aging Policy

The Fabric project maintainers will automatically monitor all PR activity for delinquency. If a PR has not been updated in 2 weeks, a reminder comment will be added requesting that the PR either be updated to address any outstanding comments or abandoned if it is to be withdrawn. If a delinquent PR goes another 2 weeks without an update, it will be automatically abandoned. If a PR has aged more than 2 months since it was originally submitted, even if it has activity, it will be flagged for maintainer review.

If a submitted PR has passed all validation but has not been reviewed in 72 hours (3 days), it will be flagged to the #fabric-pr-review channel daily until it receives a review comment(s).

This policy applies to all official Fabric projects (fabric, fabric-ca, fabric-samples, fabric-test, fabric-sdk-node, fabric-sdk-java, fabric-gateway-java, fabric-chaincode-node, fabric-chaincode-java, fabric-chaincode-evm, fabric-baseimage, and fabric-amcl).

14.5.9 Setting up development environment

Next, try *building the project* in your local development environment to ensure that everything is set up correctly.

14.5.10 What makes a good pull request?

- One change at a time. Not five, not three, not ten. One and only one. Why? Because it limits the blast area of the change. If we have a regression, it is much easier to identify the culprit commit than if we have some composite change that impacts more of the code.
- Include a link to the JIRA story for the change. Why? Because a) we want to track our velocity to better judge what we think we can deliver and when and b) because we can justify the change more effectively. In many cases, there should be some discussion around a proposed change and we want to link back to that from the change itself.
- Include unit and integration tests (or changes to existing tests) with every change. This does not mean just happy path testing, either. It also means negative testing of any defensive code that it correctly catches input errors. When you write code, you are responsible to test it and provide the tests that demonstrate that your change does what it claims. Why? Because without this we have no clue whether our current code base actually works.
- Unit tests should have NO external dependencies. You should be able to run unit tests in place with `go test` or equivalent for the language. Any test that requires some external dependency (e.g. needs to be scripted to run another component) needs appropriate mocking. Anything else is not unit testing, it is integration testing by definition. Why? Because many open source developers do Test Driven Development. They place a watch on the directory that invokes the tests automagically as the code is changed. This is far more efficient than having to run a whole build between code changes. See this definition of unit testing for a good set of criteria to keep in mind for writing effective unit tests.
- Minimize the lines of code per PR. Why? Maintainers have day jobs, too. If you send a 1,000 or 2,000 LOC change, how long do you think it takes to review all of that code? Keep your changes to < 200-300 LOC, if possible. If you have a larger change, decompose it into multiple independent changes. If you are adding a bunch of new functions to fulfill the requirements of a new capability, add them separately with their tests, and then write the code that uses them to deliver the capability. Of course, there are always exceptions. If you add a small change and then add 300 LOC of tests, you will be forgiven;-) If you need to make a change that has broad impact or a bunch of generated code (protobufs, etc.). Again, there can be exceptions.

Note: Large pull requests, e.g. those with more than 300 LOC are more than likely not going to receive an approval, and you'll be asked to refactor the change to conform with this guidance.

- Write a meaningful commit message. Include a meaningful 55 (or less) character title, followed by a blank line, followed by a more comprehensive description of the change. Each change MUST include the JIRA identifier corresponding to the change (e.g. [FAB-1234]). This can be in the title but should also be in the body of the commit message.

Note: Example commit message:

```
[FAB-1234] fix foobar() panic

Fix [FAB-1234] added a check to ensure that when foobar(foo string)
is called, that there is a non-empty string argument.
```

Finally, be responsive. Don't let a pull request fester with review comments such that it gets to a point that it requires a rebase. It only further delays getting it merged and adds more work for you - to remediate the merge conflicts.

14.6 Legal stuff

Note: Each source file must include a license header for the Apache Software License 2.0. See the template of the license header.

We have tried to make it as easy as possible to make contributions. This applies to how we handle the legal aspects of contribution. We use the same approach—the Developer's Certificate of Origin 1.1 (DCO)—that the Linux® Kernel community uses to manage code contributions.

We simply ask that when submitting a patch for review, the developer must include a sign-off statement in the commit message.

Here is an example Signed-off-by line, which indicates that the submitter accepts the DCO:

```
Signed-off-by: John Doe <john.doe@example.com>
```

You can include this automatically when you commit a change to your local git repository using `git commit -s`.

14.7 Related Topics

14.7.1 Using Jira to understand current work items

This document has been created to give further insight into the work in progress towards the Hyperledger Fabric v1 architecture based on the community roadmap. The requirements for the roadmap are being tracked in Jira.

It was determined to organize in sprints to better track and show a prioritized order of items to be implemented based on feedback received. We've done this via boards. To see these boards and the priorities click on **Boards -> Manage Boards**:

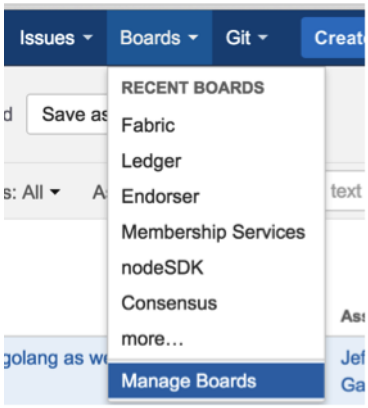


Fig. 1: Jira boards

Now on the left side of the screen click on **All boards**:

On this page you will see all the public (and restricted) boards that have been created. If you want to see the items with current sprint focus, click on the boards where the column labeled **Visibility** is **All Users** and the column **Board type** is labeled **Scrum**. For example the **Board Name** Consensus:

When you click on Consensus under **Board name** you will be directed to a page that contains the following columns:

The meanings to these columns are as follows:

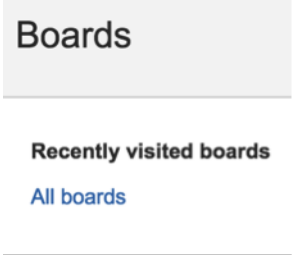


Fig. 2: Jira boards

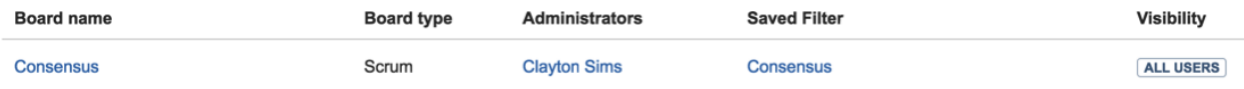


Fig. 3: Jira boards

- Backlog – list of items slated for the current sprint (sprints are defined in 2 week iterations), but are not currently in progress
- In progress – items currently being worked by someone in the community.
- In Review – items waiting to be reviewed and merged in GitHub
- Done – items merged and complete in the sprint.

If you want to see all items in the backlog for a given feature set, click on the stacked rows on the left navigation of the screen:

This shows you items slated for the current sprint at the top, and all items in the backlog at the bottom. Items are listed in priority order.

If there is an item you are interested in working on, want more information or have questions, or if there is an item that you feel needs to be in higher priority, please add comments directly to the Jira item. All feedback and help is very much appreciated.

14.7.2 Setting up the development environment

Prerequisites

- Git client, Go, and Docker as described at *Prerequisites*
- (macOS) Xcode must be installed
- (macOS) you may need to install gnutar, as macOS comes with bsdtar as the default, but the build uses some gnutar flags. You can use Homebrew to install it as follows:

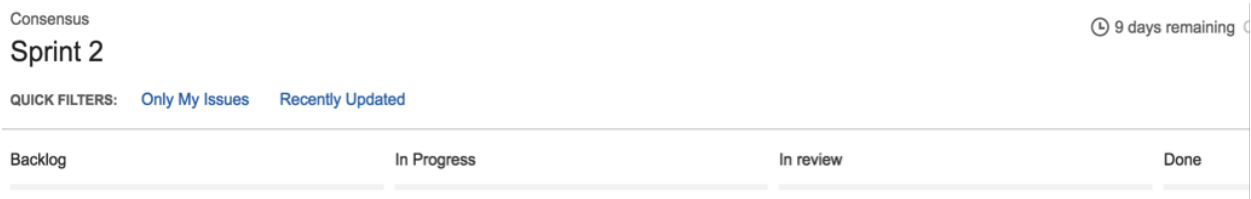


Fig. 4: Jira boards

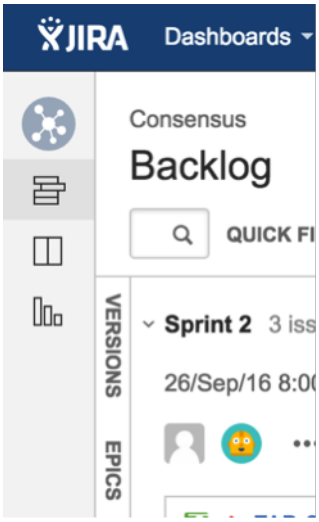


Fig. 5: Jira boards

```
brew install gnu-tar
```

- (macOS) If you install gnutar, you should prepend the "gnubin" directory to the $PATH environment variable with something like:

```
export PATH=/usr/local/opt/gnu-tar/libexec/gnubin:$PATH
```

- (macOS) Libtool. You can use Homebrew to install it as follows:

```
brew install libtool
```

- (only if using Vagrant) - Vagrant - 1.9 or later
- (only if using Vagrant) - VirtualBox - 5.0 or later
- BIOS Enabled Virtualization - Varies based on hardware
- Note: The BIOS Enabled Virtualization may be within the CPU or Security settings of the BIOS

Steps

Set your GOPATH

Make sure you have properly setup your Host's GOPATH environment variable. This allows for both building within the Host and the VM.

In case you installed Go into a different location from the standard one your Go distribution assumes, make sure that you also set GOROOT environment variable.

Note to Windows users

If you are running Windows, before running any `git clone` commands, run the following command.

```
git config --get core.autocrlf
```

If `core.autocrlf` is set to `true`, you must set it to `false` by running

```
git config --global core.autocrlf false
```

If you continue with `core.autocrlf` set to `true`, the `vagrant up` command will fail with the error:

```
./setup.sh:  /bin/bash^M: bad interpreter:  No such file or directory
```

Cloning the Hyperledger Fabric source

First navigate to https://github.com/hyperledger/fabric and fork the fabric repository using the fork button in the top-right corner

Since Hyperledger Fabric is written in `Go`, you'll need to clone the forked repository to your $GOPATH/src directory. If your $GOPATH has multiple path components, then you will want to use the first one. There's a little bit of setup needed:

```
cd $GOPATH/src
mkdir -p github.com/<your_github_userid>
cd github.com/<your_github_userid>
git clone https://github.com/<your_github_userid>/fabric
```

14.7.3 Building Hyperledger Fabric

The following instructions assume that you have already set up your *development environment*.

To build Hyperledger Fabric:

```
cd $GOPATH/src/github.com/hyperledger/fabric
make dist-clean all
```

Building the documentation

If you are contributing to the documentation, you can build the Fabric documentation on your local machine. This allows you to check the formatting of your changes using your web browser before you open a pull request.

You need to download the following prerequisites before you can build the documentation:

- Python 3.7
- Pipenv

After you make your updates to the documentation source files, you can generate a build that includes your changes by running the following commands:

```
cd fabric/docs
pipenv install
pipenv shell
make html
```

This will generate all the html files in the `docs/build/html` folder. You can open any file to start browsing the updated documentation using your browser. If you want to make additional edits to the documentation, you can rerun `make html` to incorporate the changes.

Running the unit tests

Before running the unit tests, a PKCS #11 cryptographic token implementation must be installed and configured. The PKCS #11 API is used by the bccsp component of Fabric to interact with devices, such as hardware security modules (HSMs), that store cryptographic information and perform cryptographic computations. For test environments, SoftHSM can be used to satisfy this requirement.

SoftHSM can be installed with the following commands:

```
sudo apt install libsofthsm2 # Ubuntu
sudo yum install softhsm     # CentOS
brew install softhsm         # macOS
```

Once SoftHSM is installed, additional configuration may be required. For example, the default configuration file stores token data in a system directory that unprivileged users are unable to write to.

Configuration typically involves copying `/etc/softhsm2.conf` to `$HOME/.config/softhsm2/softhsm2.conf` and changing `directories.tokendir` to an appropriate location. Please see the man page for `softhsm2.conf` for details.

After SoftHSM has been configured, the following command can be used to initialize the required token:

```
softhsm2-util --init-token --slot 0 --label "ForFabric" --so-pin 1234 --pin 98765432
```

If the test cannot find libsofthsm2.so in your environment, specify its path, the PIN and the label of the token through environment variables. For example, on macOS:

```
export PKCS11_LIB="/usr/local/Cellar/softhsm/2.5.0/lib/softhsm/libsofthsm2.so"
export PKCS11_PIN=98765432
export PKCS11_LABEL="ForFabric"
```

Use the following sequence to run all unit tests:

```
cd $GOPATH/src/github.com/hyperledger/fabric
make unit-test
```

To run a subset of tests, set the TEST_PKGS environment variable. Specify a list of packages (separated by space), for example:

```
export TEST_PKGS="github.com/hyperledger/fabric/core/ledger/..."
make unit-test
```

To run a specific test use the `-run RE` flag where RE is a regular expression that matches the test case name. To run tests with verbose output use the `-v` flag. For example, to run the `TestGetFoo` test case, change to the directory containing the `foo_test.go` and call/execute

```
go test -v -run=TestGetFoo
```

Running Node.js Client SDK Unit Tests

You must also run the Node.js unit tests to ensure that the Node.js client SDK is not broken by your changes. To run the Node.js unit tests, follow the instructions here.

14.7.4 Configuration

Configuration utilizes the viper and cobra libraries.

There is a **core.yaml** file that contains the configuration for the peer process. Many of the configuration settings can be overridden on the command line by setting ENV variables that match the configuration setting, but by prefixing with *'CORE_'*. For example, setting *peer.networkId* can be accomplished with:

```
CORE_PEER_NETWORKID=custom-network-id peer
```

14.7.5 Coding guidelines

Coding in Go

We code in Go™ and try to follow the best practices and style outlined in Effective Go and the supplemental rules from the Go Code Review Comments wiki.

We also recommend new contributors review the following before submitting pull requests:

- Practical Go
- Go Proverbs

The following tools are executed against all pull requests. Any errors flagged by these tools must be addressed before the code will be merged:

- gofmt -s
- goimports
- go vet

Testing

Unit tests are expected to accompany all production code changes. These tests should be fast, provide very good coverage for new and modified code, and support parallel execution.

Two matching libraries are commonly used in our tests. When modifying code, please use the matching library that has already been chosen for the package.

- gomega
- testify/assert

Any fixtures or data required by tests should generated or placed under version control. When fixtures are generated, they must be placed in a temporary directory created by `ioutil.TempDir` and cleaned up when the test terminates. When fixtures are placed under version control, they should be created inside a `testdata` folder; documentation that describes how to regenerate the fixtures should be provided in the tests or a `README.txt`. Sharing fixtures across packages is strongly discouraged.

When fakes or mocks are needed, they must be generated. Bespoke, hand-coded mocks are a maintenance burden and tend to include simulations that inevitably diverge from reality. Within Fabric, we use `go generate` directives to manage the generation with the following tools:

- counterfeiter
- mockery

API Documentation

The API documentation for Hyperledger Fabric's Go APIs is available in GoDoc.

14.7.6 Adding or updating Go packages

Hyperledger Fabric vendors dependencies. This means that all required packages reside in the `$GOPATH/src/github.com/hyperledger/fabric/vendor` folder. Go will use packages in this folder instead of the GOPATH when the `go install` or `go build` commands are executed. To manage the packages in the `vendor` folder, we use dep.

CHAPTER 15

Glossary

Terminology is important, so that all Hyperledger Fabric users and developers agree on what we mean by each specific term. What is a smart contract for example. The documentation will reference the glossary as needed, but feel free to read the entire thing in one sitting if you like; it's pretty enlightening!

15.1 Anchor Peer

Used by gossip to make sure peers in different organizations know about each other.

When a configuration block that contains an update to the anchor peers is committed, peers reach out to the anchor peers and learn from them about all of the peers known to the anchor peer(s). Once at least one peer from each organization has contacted an anchor peer, the anchor peer learns about every peer in the channel. Since gossip communication is constant, and because peers always ask to be told about the existence of any peer they don't know about, a common view of membership can be established for a channel.

For example, let's assume we have three organizations — `A`, `B`, `C` — in the channel and a single anchor peer — `peer0.orgC` — defined for organization `C`. When `peer1.orgA` (from organization `A`) contacts `peer0.orgC`, it will tell `peer0.orgC` about `peer0.orgA`. And when at a later time `peer1.orgB` contacts `peer0.orgC`, the latter would tell the former about `peer0.orgB`. From that point forward, organizations `A` and `B` would start exchanging membership information directly without any assistance from `peer0.orgC`.

As communication across organizations depends on gossip in order to work, there must be at least one anchor peer defined in the channel configuration. It is strongly recommended that every organization provides its own set of anchor peers for high availability and redundancy.

15.2 ACL

An ACL, or Access Control List, associates access to specific peer resources (such as system chaincode APIs or event services) to a *Policy* (which specifies how many and what types of organizations or roles are required). The ACL is part of a channel's configuration. It is therefore persisted in the channel's configuration blocks, and can be updated using the standard configuration update mechanism.

An ACL is formatted as a list of key-value pairs, where the key identifies the resource whose access we wish to control, and the value identifies the channel policy (group) that is allowed to access it. For example `lscc/GetDeploymentSpec: /Channel/Application/Readers` defines that the access to the life cycle chaincode `GetDeploymentSpec` API (the resource) is accessible by identities which satisfy the `/Channel/Application/Readers` policy.

A set of default ACLs is provided in the `configtx.yaml` file which is used by configtxgen to build channel configurations. The defaults can be set in the top level "Application" section of `configtx.yaml` or overridden on a per profile basis in the "Profiles" section.

15.3 Block

A block contains an ordered set of transactions. It is cryptographically linked to the preceding block, and in turn it is linked to be subsequent blocks. The first block in such a chain of blocks is called the **genesis block**. Blocks are created by the ordering service, and then validated and committed by peers.

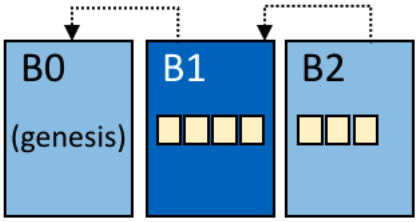


Fig. 1: Block B1 is linked to block B0. Block B2 is linked to block B1.

15.4 Chain

The ledger's chain is a transaction log structured as hash-linked blocks of transactions. Peers receive blocks of transactions from the ordering service, mark the block's transactions as valid or invalid based on endorsement policies and concurrency violations, and append the block to the hash chain on the peer's file system.

15.5 Chaincode

See *Smart-Contract*.

15.6 Channel

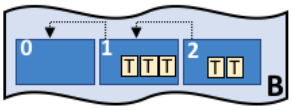


Fig. 2: Blockchain B contains blocks 0, 1, 2.

A channel is a private blockchain overlay which allows for data isolation and confidentiality. A channel-specific ledger is shared across the peers in the channel, and transacting parties must be authenticated to a channel in order to interact with it. Channels are defined by a *Configuration-Block*.

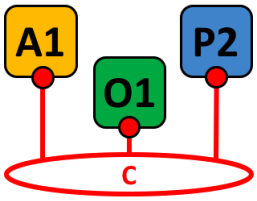


Fig. 3: Channel C connects application A1, peer P2 and ordering service O1.

15.7 Commit

Each *Peer* on a channel validates ordered blocks of transactions and then commits (writes/appends) the blocks to its replica of the channel *Ledger*. Peers also mark each transaction in each block as valid or invalid.

15.8 Concurrency Control Version Check

Concurrency Control Version Check is a method of keeping ledger state in sync across peers on a channel. Peers execute transactions in parallel, and before committing to the ledger, peers check whether the state read at the time the transaction was executed has been modified. If the data read for the transaction has changed between execution time and commit time, then a Concurrency Control Version Check violation has occurred, and the transaction is marked as invalid on the ledger and values are not updated in the state database.

15.9 Configuration Block

Contains the configuration data defining members and policies for a system chain (ordering service) or channel. Any configuration modifications to a channel or overall network (e.g. a member leaving or joining) will result in a new configuration block being appended to the appropriate chain. This block will contain the contents of the genesis block, plus the delta.

15.10 Consensus

A broader term overarching the entire transactional flow, which serves to generate an agreement on the order and to confirm the correctness of the set of transactions constituting a block.

15.11 Consenter set

In a Raft ordering service, these are the ordering nodes actively participating in the consensus mechanism on a channel. If other ordering nodes exist on the system channel, but are not a part of a channel, they are not part of that channel's consenter set.

15.12 Consortium

A consortium is a collection of non-orderer organizations on the blockchain network. These are the organizations that form and join channels and that own peers. While a blockchain network can have multiple consortia, most blockchain

networks have a single consortium. At channel creation time, all organizations added to the channel must be part of a consortium. However, an organization that is not defined in a consortium may be added to an existing channel.

15.13 Chaincode definition

A chaincode definition is used by organizations to agree on the parameters of a chaincode before it can be used on a channel. Each channel member that wants to use the chaincode to endorse transactions or query the ledger needs to approve a chaincode definition for their organization. Once enough channel members have approved a chaincode definition to meet the Lifecycle Endorsement policy (which is set to a majority of organizations in the channel by default), the chaincode definition can be committed to the channel. After the definition is committed, the first invoke of the chaincode (or, if requested, the execution of the Init function) will start the chaincode on the channel.

15.14 Dynamic Membership

Hyperledger Fabric supports the addition/removal of members, peers, and ordering service nodes, without compromising the operationality of the overall network. Dynamic membership is critical when business relationships adjust and entities need to be added/removed for various reasons.

15.15 Endorsement

Refers to the process where specific peer nodes execute a chaincode transaction and return a proposal response to the client application. The proposal response includes the chaincode execution response message, results (read set and write set), and events, as well as a signature to serve as proof of the peer's chaincode execution. Chaincode applications have corresponding endorsement policies, in which the endorsing peers are specified.

15.16 Endorsement policy

Defines the peer nodes on a channel that must execute transactions attached to a specific chaincode application, and the required combination of responses (endorsements). A policy could require that a transaction be endorsed by a minimum number of endorsing peers, a minimum percentage of endorsing peers, or by all endorsing peers that are assigned to a specific chaincode application. Policies can be curated based on the application and the desired level of resilience against misbehavior (deliberate or not) by the endorsing peers. A transaction that is submitted must satisfy the endorsement policy before being marked as valid by committing peers.

15.17 Follower

In a leader based consensus protocol, such as Raft, these are the nodes which replicate log entries produced by the leader. In Raft, the followers also receive "heartbeat" messages from the leader. In the event that the leader stops sending those message for a configurable amount of time, the followers will initiate a leader election and one of them will be elected leader.

15.18 Genesis Block

The configuration block that initializes the ordering service, or serves as the first block on a chain.

15.19 Gossip Protocol

The gossip data dissemination protocol performs three functions: 1) manages peer discovery and channel membership; 2) disseminates ledger data across all peers on the channel; 3) syncs ledger state across all peers on the channel. Refer to the *Gossip* topic for more details.

15.20 Hyperledger Fabric CA

Hyperledger Fabric CA is the default Certificate Authority component, which issues PKI-based certificates to network member organizations and their users. The CA issues one root certificate (rootCert) to each member and one enrollment certificate (ECert) to each authorized user.

15.21 Init

A method to initialize a chaincode application. All chaincodes need to have an an Init function. By default, this function is never executed. However you can use the chaincode definition to request the execution of the Init function in order to initialize the chaincode.

15.22 Install

The process of placing a chaincode on a peer's file system.

15.23 Instantiate

The process of starting and initializing a chaincode application on a specific channel. After instantiation, peers that have the chaincode installed can accept chaincode invocations.

NOTE: *This method i.e. Instantiate was used in the 1.4.x and older versions of the chaincode lifecycle. For the current procedure used to start a chaincode on a channel with the new Fabric chaincode lifecycle introduced as part of Fabric v2.0, see Chaincode-definition_.*

15.24 Invoke

Used to call chaincode functions. A client application invokes chaincode by sending a transaction proposal to a peer. The peer will execute the chaincode and return an endorsed proposal response to the client application. The client application will gather enough proposal responses to satisfy an endorsement policy, and will then submit the transaction results for ordering, validation, and commit. The client application may choose not to submit the transaction results. For example if the invoke only queried the ledger, the client application typically would not submit the read-only transaction, unless there is desire to log the read on the ledger for audit purpose. The invoke includes a channel identifier, the chaincode function to invoke, and an array of arguments.

15.25 Leader

In a leader based consensus protocol, like Raft, the leader is responsible for ingesting new log entries, replicating them to follower ordering nodes, and managing when an entry is considered committed. This is not a special **type** of orderer. It is only a role that an orderer may have at certain times, and then not others, as circumstances determine.

15.26 Leading Peer

Each *Organization* can own multiple peers on each channel that they subscribe to. One or more of these peers should serve as the leading peer for the channel, in order to communicate with the network ordering service on behalf of the organization. The ordering service delivers blocks to the leading peer(s) on a channel, who then distribute them to other peers within the same organization.

15.27 Ledger

A ledger consists of two distinct, though related, parts – a "blockchain" and the "state database", also known as "world state". Unlike other ledgers, blockchains are **immutable** – that is, once a block has been added to the chain, it cannot be changed. In contrast, the "world state" is a database containing the current value of the set of key-value pairs that have been added, modified or deleted by the set of validated and committed transactions in the blockchain.

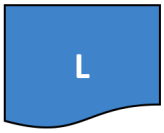


Fig. 4: A Ledger, 'L'

It's helpful to think of there being one **logical** ledger for each channel in the network. In reality, each peer in a channel maintains its own copy of the ledger – which is kept consistent with every other peer's copy through a process called **consensus**. The term **Distributed Ledger Technology** (**DLT**) is often associated with this kind of ledger – one that is logically singular, but has many identical copies distributed across a set of network nodes (peers and the ordering service).

15.28 Log entry

The primary unit of work in a Raft ordering service, log entries are distributed from the leader orderer to the followers. The full sequence of such entries known as the "log". The log is considered to be consistent if all members agree on the entries and their order.

15.29 Member

See *Organization*.

15.30 Membership Service Provider

The Membership Service Provider (MSP) refers to an abstract component of the system that provides credentials to clients, and peers for them to participate in a Hyperledger Fabric network. Clients use these credentials to authenticate their transactions, and peers use these credentials to authenticate transaction processing results (endorsements). While strongly connected to the transaction processing components of the

Fig. 5: An MSP, 'ORG.MSP'

systems, this interface aims to have membership services components defined, in such a way that alternate implementations of this can be smoothly plugged in without modifying the core of transaction processing components of the system.

15.31 Membership Services

Membership Services authenticates, authorizes, and manages identities on a permissioned blockchain network. The membership services code that runs in peers and orderers both authenticates and authorizes blockchain operations. It is a PKI-based implementation of the Membership Services Provider (MSP) abstraction.

15.32 Ordering Service

Also known as **orderer**. A defined collective of nodes that orders transactions into a block and then distributes blocks to connected peers for validation and commit. The ordering service exists independent of the peer processes and orders transactions on a first-come-first-serve basis for all channels on the network. It is designed to support pluggable implementations beyond the out-of-the-box Kafka and Raft varieties. It is a common binding for the overall network; it contains the cryptographic identity material tied to each *Member*.

15.33 Organization

Also known as "members", organizations are invited to join the blockchain network by a blockchain network provider. An organization is joined to a network by adding its Membership Service Provider (*MSP*) to the network. The MSP defines how other members of the network may verify that signatures (such as those over transactions) were generated by a valid identity, issued by that organization. The particular access rights of identities within an MSP are governed by policies which are also agreed upon when the organization is joined to the network. An organization can be as large as a multi-national corporation or as small as an individual. The transaction endpoint of an organization is a *Peer*. A collection of organizations form a *Consortium*. While all of the organizations on a network are members, not every organization will be part of a consortium.

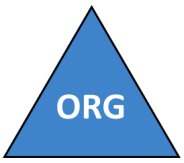


Fig. 6: An organization, 'ORG'

15.34 Peer

A network entity that maintains a ledger and runs chaincode containers in order to perform read/write operations to the ledger. Peers are owned and maintained by members.

Fig. 7: A peer, 'P'

15.35 Policy

Policies are expressions composed of properties of digital identities, for example: `Org1.Peer OR Org2.Peer`. They are used to restrict access to resources on a blockchain network. For instance, they dictate who can read from or write to a channel, or who can use a specific chaincode API via an *ACL*. Policies may be defined in `configtx.yaml` prior to bootstrapping an ordering service or creating a channel, or they can be specified

when instantiating chaincode on a channel. A default set of policies ship in the sample `configtx.yaml` which will be appropriate for most networks.

15.36 Private Data

Confidential data that is stored in a private database on each authorized peer, logically separate from the channel ledger data. Access to this data is restricted to one or more organizations on a channel via a private data collection definition. Unauthorized organizations will have a hash of the private data on the channel ledger as evidence of the transaction data. Also, for further privacy, hashes of the private data go through the *Ordering-Service*, not the private data itself, so this keeps private data confidential from Orderer.

15.37 Private Data Collection (Collection)

Used to manage confidential data that two or more organizations on a channel want to keep private from other organizations on that channel. The collection definition describes a subset of organizations on a channel entitled to store a set of private data, which by extension implies that only these organizations can transact with the private data.

15.38 Proposal

A request for endorsement that is aimed at specific peers on a channel. Each proposal is either an Init or an Invoke (read/write) request.

15.39 Query

A query is a chaincode invocation which reads the ledger current state but does not write to the ledger. The chaincode function may query certain keys on the ledger, or may query for a set of keys on the ledger. Since queries do not change ledger state, the client application will typically not submit these read-only transactions for ordering, validation, and commit. Although not typical, the client application can choose to submit the read-only transaction for ordering, validation, and commit, for example if the client wants auditable proof on the ledger chain that it had knowledge of specific ledger state at a certain point in time.

15.40 Quorum

This describes the minimum number of members of the cluster that need to affirm a proposal so that transactions can be ordered. For every consenter set, this is a **majority** of nodes. In a cluster with five nodes, three must be available for there to be a quorum. If a quorum of nodes is unavailable for any reason, the cluster becomes unavailable for both read and write operations and no new logs can be committed.

15.41 Raft

New for v1.4.1, Raft is a crash fault tolerant (CFT) ordering service implementation based on the etcd library of the *Raft protocol* <https://raft.github.io/raft.pdf>'_. Raft follows a "leader and follower" model, where a leader node is elected (per channel) and its decisions are replicated by the followers. Raft ordering services should be easier to set

up and manage than Kafka-based ordering services, and their design allows organizations to contribute nodes to a distributed ordering service.

15.42 Software Development Kit (SDK)

The Hyperledger Fabric client SDK provides a structured environment of libraries for developers to write and test chaincode applications. The SDK is fully configurable and extensible through a standard interface. Components, including cryptographic algorithms for signatures, logging frameworks and state stores, are easily swapped in and out of the SDK. The SDK provides APIs for transaction processing, membership services, node traversal and event handling.

Currently, the two officially supported SDKs are for Node.js and Java, while two more – Python and Go – are not yet official but can still be downloaded and tested.

15.43 Smart Contract

A smart contract is code – invoked by a client application external to the blockchain network – that manages access and modifications to a set of key-value pairs in the *World State* via *Transaction*. In Hyperledger Fabric, smart contracts are packaged as chaincode. Chaincode is installed on peers and then defined and used on one or more channels.

15.44 State Database

World state data is stored in a state database for efficient reads and queries from chaincode. Supported databases include levelDB and couchDB.

15.45 System Chain

Contains a configuration block defining the network at a system level. The system chain lives within the ordering service, and similar to a channel, has an initial configuration containing information such as: MSP information, policies, and configuration details. Any change to the overall network (e.g. a new org joining or a new ordering node being added) will result in a new configuration block being added to the system chain.

The system chain can be thought of as the common binding for a channel or group of channels. For instance, a collection of financial institutions may form a consortium (represented through the system chain), and then proceed to create channels relative to their aligned and varying business agendas.

15.46 Transaction

Transactions are created when a chaincode is invoked from a client application to read or write data from the ledger. Fabric application clients submit transaction proposals to endorsing peers for execution and endorsement, gather the signed (endorsed) responses from those endorsing peers, and then package the results and endorsements into a transaction that is submitted to the ordering service. The ordering service orders and places transactions in a block that is broadcast to the peers which validate and commit the transactions to the ledger and update world state.

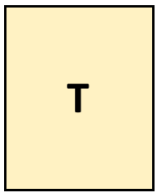


Fig. 8: A transaction, 'T'

15.47 World State

Also known as the "current state", the world state is a component of the HyperLedger Fabric *Ledger*. The world state represents the latest values for all keys included in the chain transaction log. Chaincode executes transaction proposals against world state data because the world state provides direct access to the latest value of these keys rather than having to calculate them by traversing the entire transaction log. The world state will change every time the value of a key changes (for example, when the ownership of a car – the "key" – is transferred from one owner to another – the "value") or when a new key is added (a car is created). As a result, the world state is critical to a transaction flow, since the current state of a key-value pair must be known before it can be changed. Peers commit the latest values to the ledger world state for each valid transaction included in a processed block.

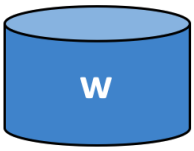


Fig. 9: The World State, 'W'

CHAPTER 16

Releases

Hyperledger Fabric releases are documented on the Fabric github page.

CHAPTER 17

Still Have Questions?

We try to maintain a comprehensive set of documentation for various audiences. However, we realize that often there are questions that remain unanswered. For any technical questions relating to Hyperledger Fabric not answered here, please use StackOverflow. Another approach to getting your questions answered to send an email to the mailing list (fabric@lists.hyperledger.org), or ask your questions on RocketChat (an alternative to Slack) on the #fabric or #fabric-questions channel.

Note: Please, when asking about problems you are facing tell us about the environment in which you are experiencing those problems including the OS, which version of Docker you are using, etc.

CHAPTER 18

Status

Hyperledger Fabric is in the *Active* state. For more information on the history of this project see our wiki page. Information on what *Active* entails can be found in the Hyperledger Project Lifecycle document.

Note: If you have questions not addressed by this documentation, or run into issues with any of the tutorials, please visit the *Still Have Questions?* page for some tips on where to find additional help.