hyperledger-fabricdocs Documentation

Release master

hyperledger

Jul 09, 2018

Table of Contents

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.

Hyperledger Fabric delivers a uniquely elastic and extensible architecture, distinguishing it from alternative blockchain solutions. Planning for the future of enterprise blockchain requires building on top of a fully-vetted, open source architecture; Hyperledger Fabric is your starting point.

It's recommended for first-time users to begin by going through the *Getting Started* section in order to gain familiarity with the Hyperledger Fabric components and the basic transaction flow. Once comfortable, continue exploring the library for demos, technical specifications, APIs, etc.

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.

Before diving in, watch how Hyperledger Fabric is Building a Blockchain for Business:

CHAPTER 1

Getting Started

Prerequisites

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.

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: 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
```

Go Programming Language

Hyperledger Fabric uses the Go programming language 1.9.x for many of its components.

Note: Building with Go version 1.8.x is not supported

- Go - version 1.9.x

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.

Node.js Runtime and NPM

If you will be developing applications for Hyperledger Fabric leveraging the Hyperledger Fabric SDK for Node.js, you will need to have version 8.9.x of Node.js installed.

Note: Node.js version 9.x is not supported at this time.

- Node.js - version 8.9.x or greater

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
```

Windows extras

If you are developing on Windows 7, you will want to work within the Docker Quickstart Terminal which uses Git Bash and provides a better alternative to the built-in Windows shell.

However 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 may have difficulties with operations involving the `make` and `docker` commands.

On Windows 10 you should use the native Docker distribution and you may use the Windows PowerShell. However, for the *Download Platform-specific 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 install and use a newer version 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.

Hyperledger Fabric Samples

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 Hyperledger Fabric samples applications repository and open that in a terminal window. Then, execute the following commands:

```
git clone -b master https://github.com/hyperledger/fabric-samples.git
cd fabric-samples
git checkout {TAG}
```

Note: To ensure the samples are compatible with the version of Fabric binaries you download below, checkout the samples `{TAG}` that matches your Fabric version, for example, v1.1.0. To see a list of all fabric-samples tags, use command "git tag".

Download Platform-specific Binaries

Next, we will install the Hyperledger Fabric platform-specific binaries. This process was designed to complement the Hyperledger Fabric Samples above, but can be used independently. If you are not installing the samples above, then simply create and enter a directory into which to extract the contents of the platform-specific binaries.

Please execute the following command from within the directory into which you will extract the platform-specific binaries:

```
curl -sSL https://goo.gl/6wtTN5 | bash -s 1.1.0
```

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://github.com/hyperledger/fabric/blob/master/scripts/bootstrap.sh

Note: You can use the command above for any published version of Hyperledger Fabric. Simply replace '1.1.0' with the version identifier of the version you wish to install.

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 four platform-specific binaries:

- `cryptogen` ,
- `configtxgen` ,

- `configtxlator`,
- `peer`
- `orderer` and
- `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 "x86_64-1.x.x" and one tagged as "latest".

Note: On different architectures, the x86_64 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.

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.

Install Binaries and Docker Images

While we work on developing real installers for the Hyperledger Fabric binaries, we provide a script that will *Download Platform-specific Binaries* to your system. The script also will download the Docker images to your local registry.

Hyperledger Fabric Samples

We offer a set of sample applications that you may wish to install these *Hyperledger Fabric Samples* before starting with the tutorials as the tutorials leverage the sample code.

API Documentation

The API documentation for Hyperledger Fabric's Golang APIs can be found on the godoc site for Fabric. If you plan on doing any development using these APIs, you may want to bookmark those links now.

Hyperledger Fabric SDKs

Hyperledger Fabric offers a number of SDKs to support various programming languages. There are two officially released SDKs for Node.js and Java:

- Hyperledger Fabric Node SDK and Node SDK documentation.
- Hyperledger Fabric Java SDK.

In addition, there are three more SDKs that have not yet been officially released (for Python, Go and REST), but they are still available for downloading and testing:

- Hyperledger Fabric Python SDK.
- Hyperledger Fabric Go SDK.
- Hyperledger Fabric REST SDK.

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 2

Key Concepts

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.

Hyperledger Fabric delivers a uniquely elastic and extensible architecture, distinguishing it from alternative blockchain solutions. Planning for the future of enterprise blockchain requires building on top of a fully vetted, open-source architecture; Hyperledger Fabric is your starting point.

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.

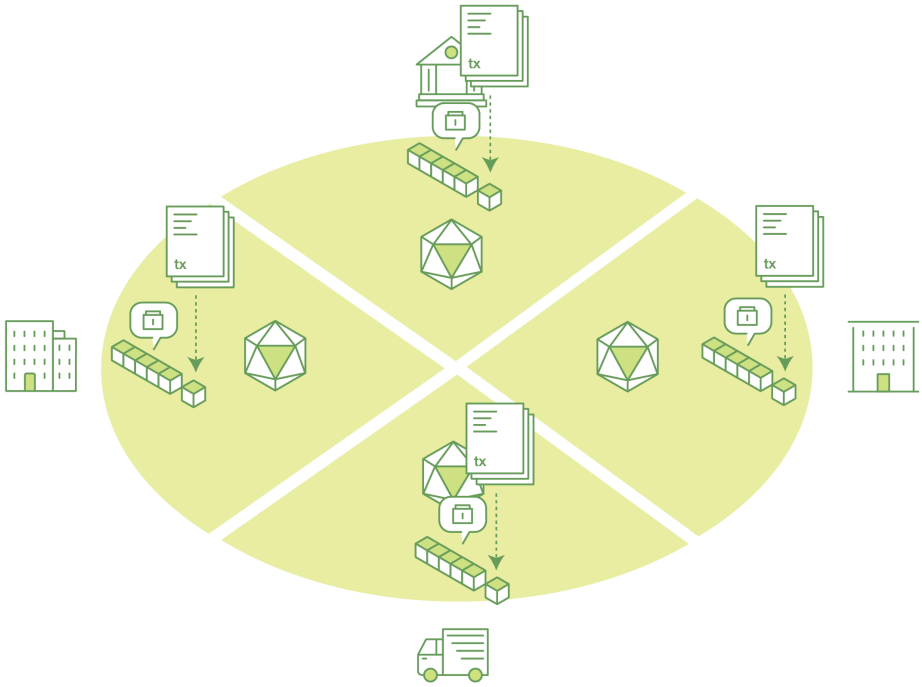
Once comfortable – or if you're already familiar with blockchain and Hyperledger Fabric – go to *Getting Started* and from there explore the demos, technical specifications, APIs, etc.

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.

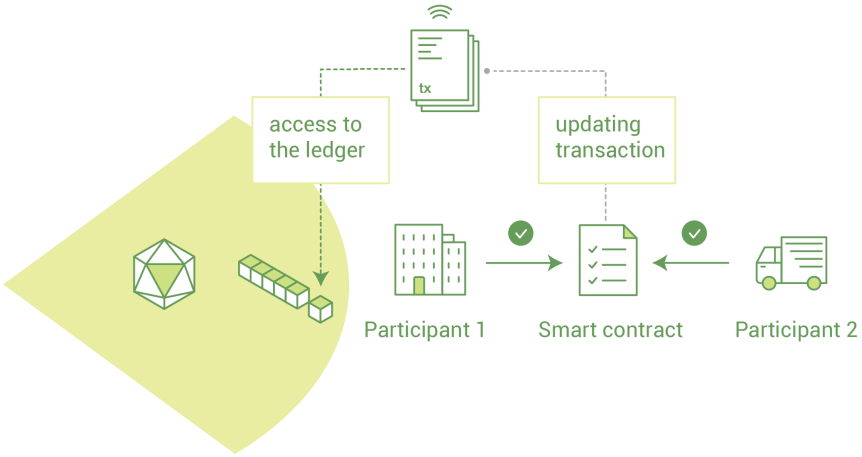
A blockchain ledger is often described as **decentralized** because it is replicated across many network participants, each of whom **collaborate** in its maintenance. We'll see that decentralization and collaboration are powerful attributes that mirror the way businesses exchange goods and services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

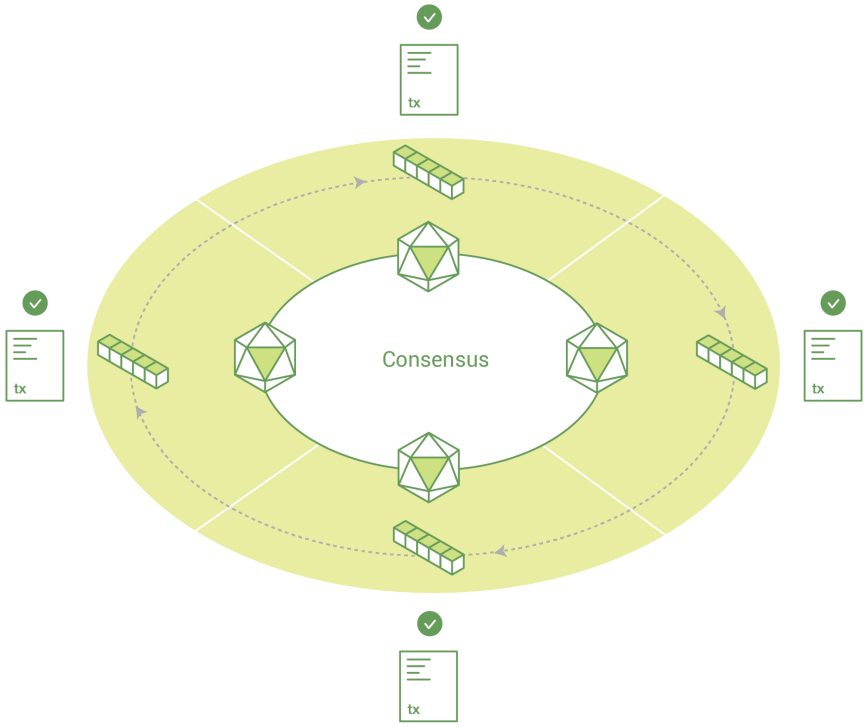


Smart contracts are not only a key mechanism for encapsulating information and keeping it simple across the network, they can also be written to allow participants to execute certain aspects of transactions automatically.

A smart contract can, for example, be written to stipulate the cost of shipping an item that changes depending on when it 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**.



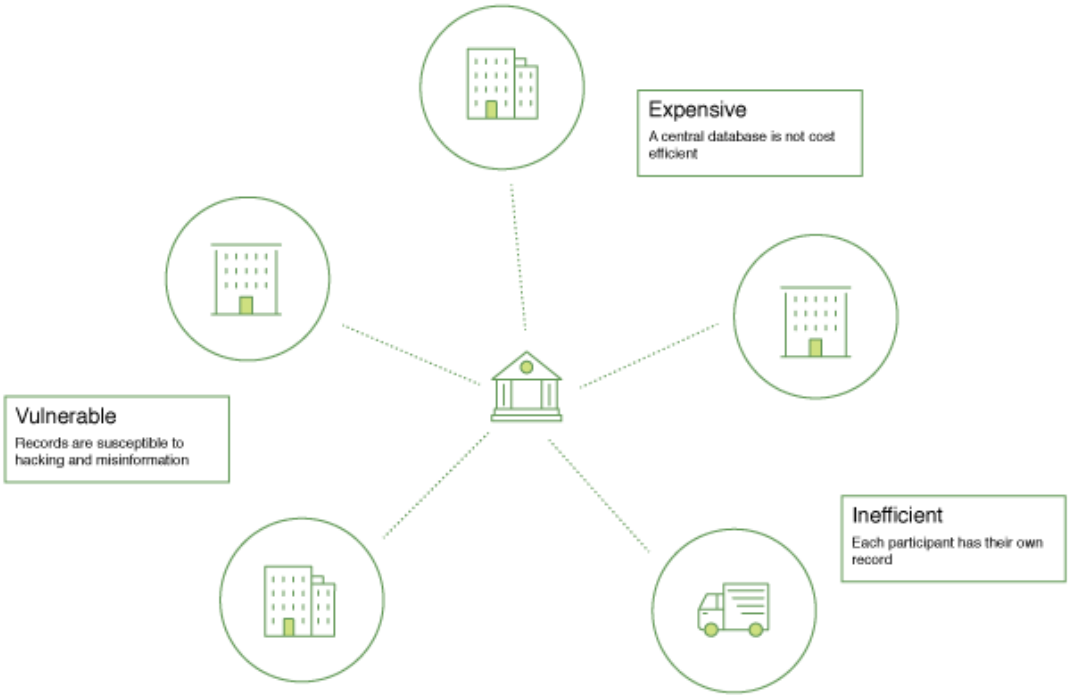
We'll learn a lot more about ledgers, smart contracts and consensus later. For now, it's enough to think of a blockchain as a shared, replicated transaction system which is updated via smart contracts and kept consistently synchronized through a collaborative process called consensus.

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.

What is Hyperledger Fabric?

The Linux Foundation founded Hyperledger 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. The currently supported chaincode language is Go with support for Java and other languages coming in future releases.

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.

We'll learn more about the Hyperledger Fabric consensus mechanisms, which currently include SOLO, Kafka, and will soon extend to SBFT (Simplified Byzantine Fault Tolerance), in another document.

Where can I learn more?

Getting Started

We provide a number of tutorials where you'll be introduced to most of the key components within a blockchain network, learn more about how they interact with each other, and then you'll actually get the code and run some simple transactions against a running blockchain network. We also provide tutorials for those of you thinking of operating a blockchain network using Hyperledger Fabric.

Hyperledger Fabric Model

A deeper look at the components and concepts brought up in this introduction as well as a few others and describes how they work together in a sample transaction flow.

Hyperledger Fabric Functionalities

Hyperledger Fabric is an implementation of distributed ledger technology (DLT) that delivers enterprise-ready network security, scalability, confidentiality and performance, in a modular blockchain architecture. Hyperledger Fabric delivers the following blockchain network functionalities:

Identity management

To enable permissioned networks, Hyperledger Fabric provides a membership identity service that manages user IDs and authenticates all participants on the network. Access control lists can be used to provide additional layers of permission through authorization of specific network operations. For example, a specific user ID could be permitted to invoke a chaincode application, but blocked from deploying new chaincode.

Privacy and confidentiality

Hyperledger Fabric enables competing business interests, and any groups that require private, confidential transactions, to coexist on the same permissioned network. Private **channels** are restricted messaging paths that can be used to provide transaction privacy and confidentiality for specific subsets of network members. All data, including transaction, member and channel information, on a channel are invisible and inaccessible to any network members not explicitly granted access to that channel.

Efficient processing

Hyperledger Fabric assigns network roles by node type. To provide concurrency and parallelism to the network, transaction execution is separated from transaction ordering and commitment. Executing transactions prior to ordering them enables each peer node to process multiple transactions simultaneously. This concurrent execution increases processing efficiency on each peer and accelerates delivery of transactions to the ordering service.

In addition to enabling parallel processing, the division of labor unburdens ordering nodes from the demands of transaction execution and ledger maintenance, while peer nodes are freed from ordering (consensus) workloads. This bifurcation of roles also limits the processing required for authorization and authentication; all peer nodes do not have to trust all ordering nodes, and vice versa, so processes on one can run independently of verification by the other.

Chaincode functionality

Chaincode applications encode logic that is invoked by specific types of transactions on the channel. Chaincode that defines parameters for a change of asset ownership, for example, ensures that all transactions that transfer ownership are subject to the same rules and requirements. **System chaincode** is distinguished as chaincode that defines operating parameters for the entire channel. Lifecycle and configuration system chaincode defines the rules for the channel; endorsement and validation system chaincode defines the requirements for endorsing and validating transactions.

Modular design

Hyperledger Fabric implements a modular architecture to provide functional choice to network designers. Specific algorithms for identity, ordering (consensus) and encryption, for example, can be plugged in to any Hyperledger Fabric network. The result is a universal blockchain architecture that any industry or public domain can adopt, with the assurance that its networks will be interoperable across market, regulatory and geographic boundaries.

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 through Channels* - Channels enable multi-lateral transactions with the high degrees of privacy and confidentiality 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.

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.

You can easily define and use assets in your Hyperledger Fabric applications using the Hyperledger Composer tool.

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.

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.

- 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
- Channel's 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."

Privacy through Channels

Hyperledger Fabric employs an immutable ledger on a per-channel basis, as well as chaincodes 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 only include 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).

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 only be decrypted by a user in possession of the corresponding key that was used to generate the cipher text. For further details on chaincode encryption, see the *Chaincode for Developers* topic.

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.

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 ultimately achieved 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 hierarchal 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, but 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.

Identity

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 digitial 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 how this is achieved 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).

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.

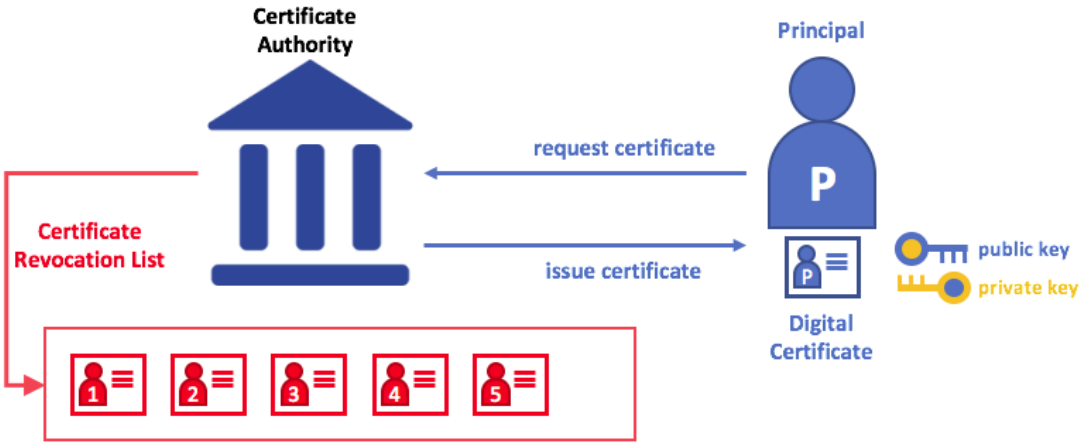
Having a valid credit card is not enough — it must also be accepted by the store! PKIs and MSPs work together in the same way — a PKI provides a list of identities, and an MSP says which of these are members of a given organization that participates in the network.

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.

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 with 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:

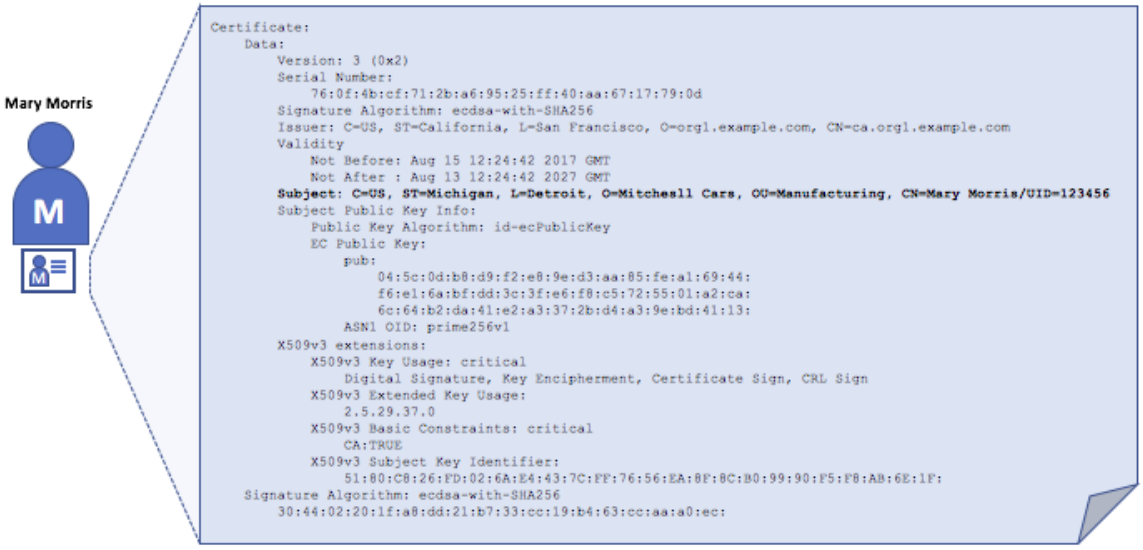
- **Digital Certificates**
- **Public and Private Keys**
- **Certificate Authorities**
- **Certificate Revocation Lists**

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

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.

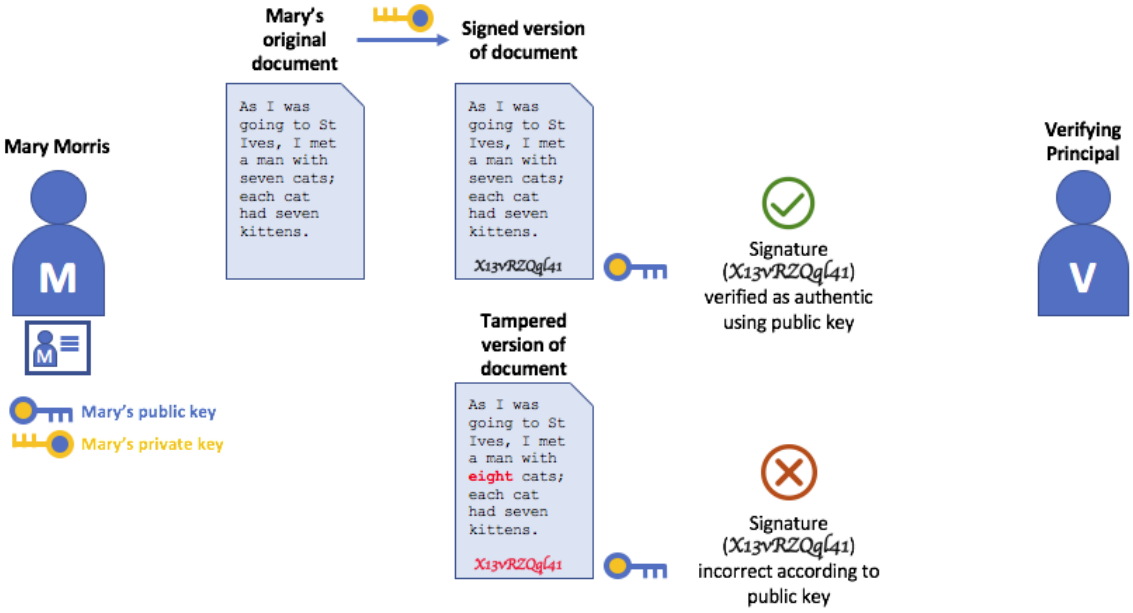
Authentication, Public keys, and Private Keys

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.

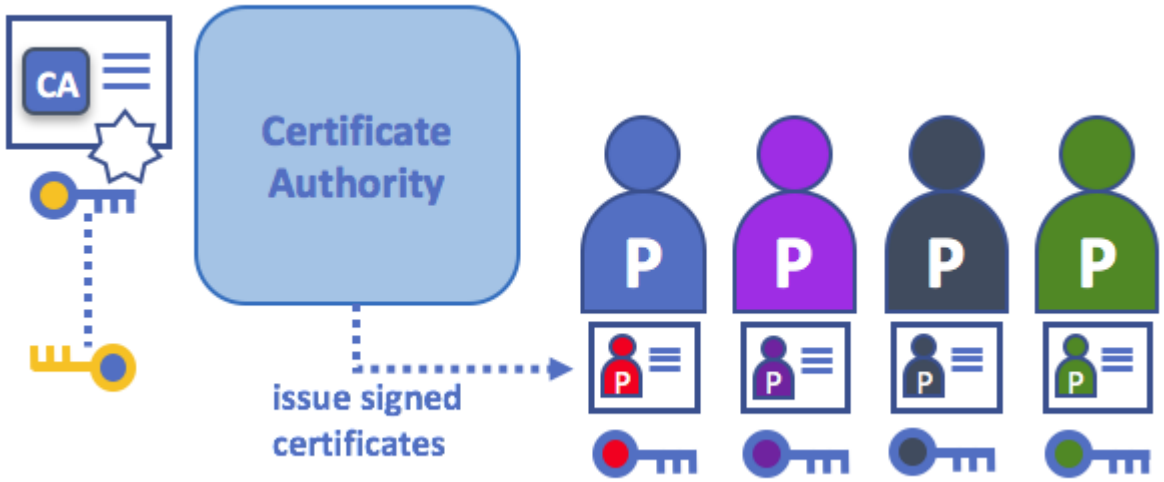


In the example above, Mary uses her private key to sign the message. The signature can be verified by anyone who sees the signed message using her public key.

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.

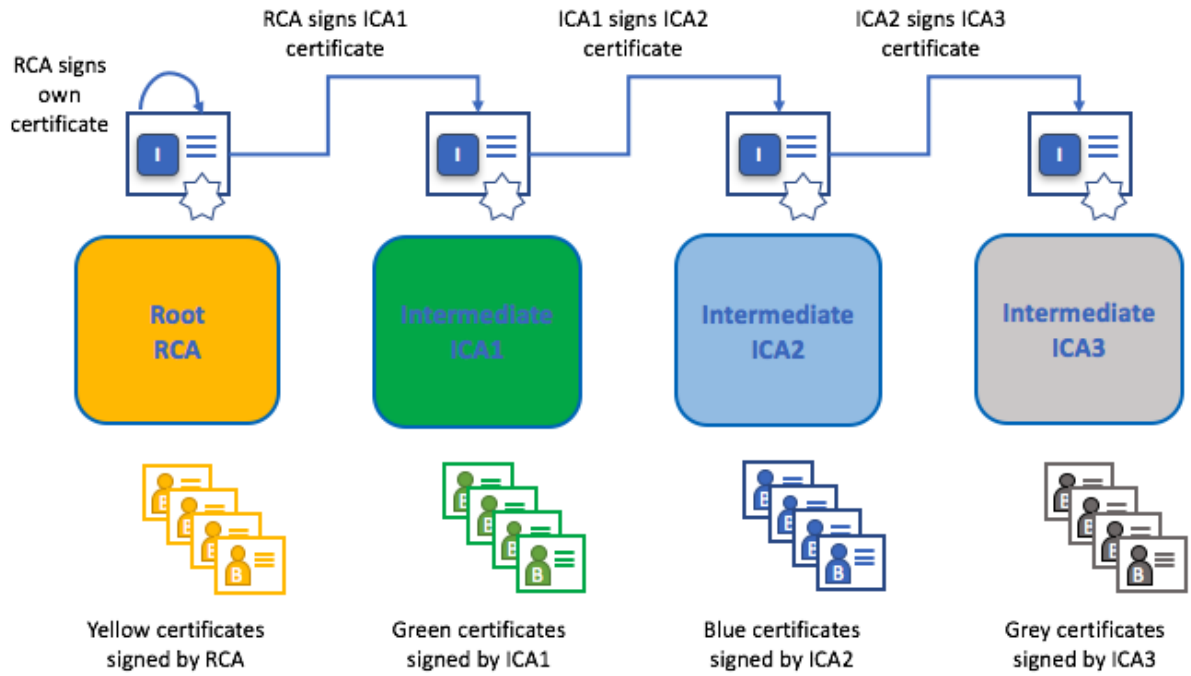
Certificates can be widely disseminated, as they do not include either the actors' nor the CA's private keys. As such they can be used as anchor of trusts for authenticating messages coming from different actors.

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

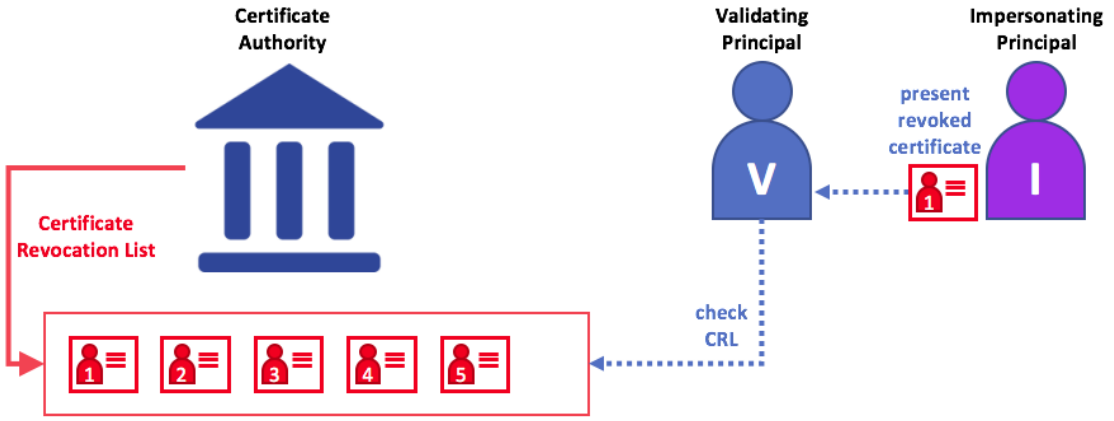
It's because CAs are so important that Fabric provides a built-in CA component to allow you to create CAs in the blockchain networks you form. This component — known as **Fabric CA** is a private root CA provider capable of managing digital identities of Fabric participants that have the form of X.509 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/commerical 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.

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.

Membership

If you've read through the documentation on identity you've seen how a PKI can provide verifiable identities through a chain of trust. Now let's see how these identities can be used to represent the trusted members of a blockchain network.

This is where a **Membership Service** Provider (MSP) comes into play — **it identifies which Root CAs and Intermediate CAs are trusted to define the members of a trust domain, e.g., an organization**, either by listing the identities of their members, or by identifying which CAs are authorized to issue valid identities for their members, or — as will usually be the case — through a combination of both.

The power of an MSP goes beyond simply listing who is a network participant or member of a channel. An MSP can identify specific **roles** an actor might play either within the scope of the organization the MSP represents (e.g., admins, or as members of a sub-organization group), and sets the basis for defining **access privileges** in the context of a network and channel (e.g., channel admins, readers, writers).

The configuration of an MSP is advertised to all the channels where members of the corresponding organization participate (in the form of a **channel MSP**). In addition to the channel MSP, peers, orderers, and clients also maintain a **local MSP** to authenticate member messages outside the context of a channel and to define the permissions over a particular component (who has the ability to install chaincode on a peer, for example).

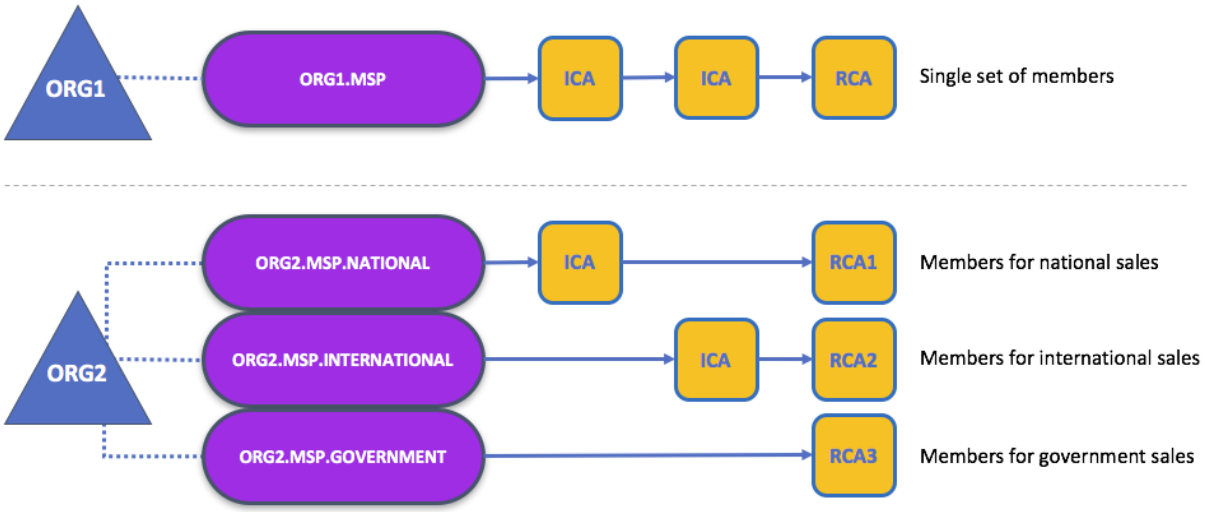
In addition, an MSP can allow for the identification of a list of identities that have been revoked — as discussed in the Identity documentation — but we will talk about how that process also extends to an MSP.

We'll talk more about local and channel MSPs in a moment. For now let's see what MSPs do in general.

Mapping MSPs to Organizations

An **organization** is a 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. Note that this is different from the organization concept defined in an X.509 certificate, which we'll talk about later.

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.



Two different MSP configurations for an organization. The first configuration shows the typical relationship between an MSP and an organization — a single MSP defines the list of members of an organization. In the second configuration, different MSPs are used to represent different organizational groups with national, international, and governmental affiliation.

Organizational Units and MSPs

An organization is often divided up into multiple **organizational units** (OUs), each of which has a certain set of responsibilities. 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.

We'll see later how OUs can be helpful to control the parts of an organization who are considered to be the members of a blockchain network. For example, only identities from the `ORG1-MANUFACTURING` OU might be able to access a channel, whereas `ORG1-DISTRIBUTION` cannot.

Finally, though this is a slight misuse of OUs, they can sometimes be used by different organizations in a consortium to distinguish each other. In such cases, the different organizations use the same Root CAs and Intermediate CAs for their chain of trust, but assign the `OU` field to identify members of each organization. We'll also see how to configure MSPs to achieve this later.

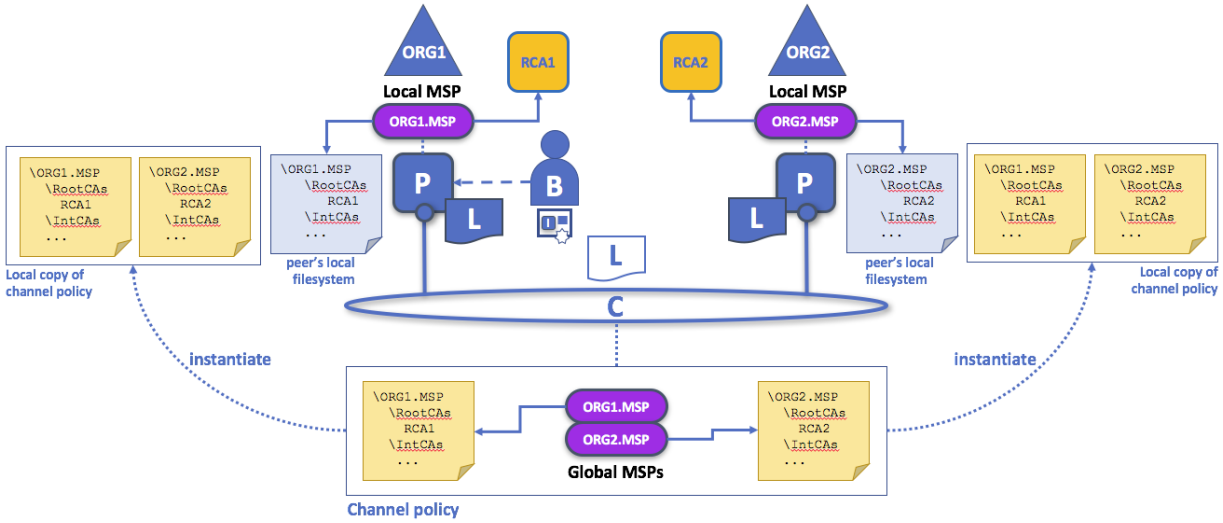
Local and Channel MSPs

MSPs appear in two places in a blockchain network: channel configuration (**channel MSPs**), and locally on an actor's premise (**local MSP**). **Local MSPs are defined for clients (users) and for nodes (peers and orderers)**. Node local MSPs define the permissions for that node (who the peer admins are, for example). The local MSPs of the users allow the user side 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 (an org admin, for example, in configuration transactions).

Every node and user 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).

In contrast, **channel MSPs define administrative and participatory rights at the channel level**. Every organization participating in a channel must have an MSP defined for it. Peers and orderers on a channel will all 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.

The key difference here between local and channel MSPs is not how they function — both turn identities into roles — but their **scope**.



Local and channel MSPs. The trust domain (e.g., the organization) of each peer is defined by the peer's local MSP, e.g., ORG1 or ORG2. Representation of an organization on a channel is achieved by adding the organization's MSP to the channel configuration. For example, the channel of this figure is managed by both ORG1 and ORG2. Similar principles apply for the network, orderers, and users, but these are not shown here for simplicity.

You may find it helpful to see how local and channel MSPs are used by seeing what happens when a blockchain administrator installs and instantiates a smart contract, as shown in the *diagram above*.

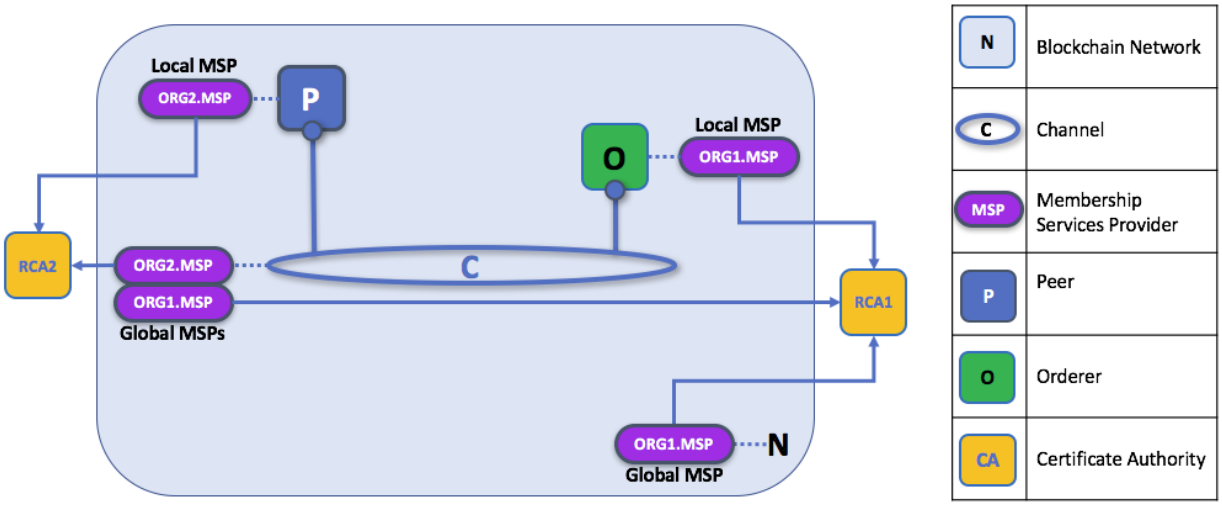
An administrator `B` connects to the peer with an identity issued by `RCA1` and stored in their local MSP. When `B` tries to install a smart contract on the peer, the peer checks its local MSP, `ORG1-MSP`, to verify that the identity of `B` is indeed a member of `ORG1`. A successful verification will allow the install command to complete successfully. Subsequently, `B` wishes to instantiate the smart contract on the channel. Because this is a channel operation, all organizations on the channel must agree to it. Therefore, the peer must check the MSPs of the channel before it can successfully commit this command. (Other things must happen too, but concentrate on the above for now.)

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 or user. 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.

MSP Levels

The split between channel and local MSPs reflects the needs of organizations to administer their local resources, such as a peer or orderer nodes, and their channel resources, such as ledgers, smart contracts, and consortia, which operate at the channel or network level. It's helpful to think of these MSPs as being at different **levels**, with **MSPs at a higher level relating to network administration concerns** while **MSPs at a lower level handle identity for the administration of private resources**. MSPs are mandatory at every level of administration — they must be defined for the network, channel, peer, orderer, and users.



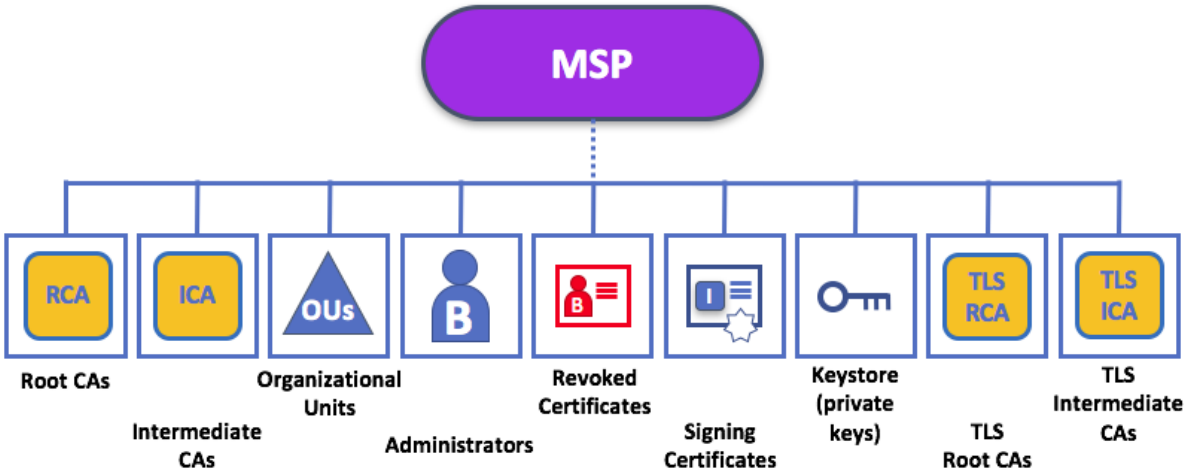
MSP Levels. The MSPs for the peer and orderer are local, whereas the MSPs for a channel (including the network configuration channel) are shared across all participants of that channel. In this figure, the network configuration channel is administered by ORG1, but another application channel can be managed 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. 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.

- **Network MSP:** The configuration of a network defines who are the members in the network — by defining the MSPs of the participant organizations — as well as which of these members are authorized to perform administrative tasks (e.g., creating a channel).
- **Channel MSP:** It is important for a channel to maintain the MSPs of its members separately. A channel provides private communications between a particular set of organizations which in turn have administrative control over it. Channel policies interpreted in the context of that channel's MSPs define who has ability to participate in certain action on the channel, e.g., adding organizations, or instantiating chaincodes. Note that there is no necessary relationship between the permission to administrate a channel and the ability to administrate the network configuration channel (or any other channel). Administrative rights exist within the scope of what is being administrated (unless the rules have been written otherwise — see the discussion of the `ROLE` attribute below).
- **Peer MSP:** This local MSP is defined on the file system of each peer and there is a single MSP instance for each peer. Conceptually, it performs exactly the same function as channel MSPs with the restriction that it only applies to the peer where it is defined. An example of an action whose authorization is evaluated using the peer's local MSP is the installation of a chaincode on the peer.

- **Orderer MSP:** Like a peer MSP, 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.

MSP Structure

So far, you've seen that the most important element of an MSP are the specification of the root or intermediate CAs that are used to establish an actor's or node's membership in the respective organization. There are, however, more elements that are used in conjunction with these two to assist with membership functions.



The figure above shows how a local MSP is stored on a local filesystem. Even though channel MSPs are not physically structured in exactly this way, it's still a helpful way to think about them.

As you can see, there are nine elements to an MSP. It's easiest to think of these elements in a directory structure, where the MSP name is the root folder name with each subfolder representing different elements of an MSP configuration.

Let's describe these folders in a little more detail and see why they are important.

- **Root CAs:** 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 X.509 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.

- **Intermediate CAs:** 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 an 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.

- **Organizational Units (OUs):** These are listed 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 OU in it.

Specifying OUs is optional. If no OUs are listed, all 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.

- **Administrators:** This folder contains a list of identities that define the actors who have the role of administrators for this organization. For the standard MSP type, there should be one or more X.509 certificates in this list.

  It's worth noting that just because a 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.

  Even though an X.509 certificate has a `ROLE` attribute (specifying, for example, that a actor is an `admin`), this refers to a actor's role within its organization rather than on the blockchain network. This is similar to the purpose of the `OU` attribute, which — if it has been defined — refers to a actor's place in the organization.

  The `ROLE` attribute **can** be used to confer administrative rights at the channel level if the policy for that channel has been written to allow any administrator from an organization (or certain organizations) permission to perform certain channel functions (such as instantiating chaincode). In this way, an organizational role can confer a network role.

- **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 X.509 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 an MSP, local or channel, can quickly revoke a actor or node from an organization by advertising the updated CRL of the CA the revoked certificate as issued by. This "list of lists" is optional. It will only become populated as certificates are revoked.

- **Node Identity:** This folder contains the identity of the node, i.e., cryptographic material that — in combination to the content of `KeyStore` — would allow the node to authenticate itself in the messages that is sends to other participants of its channels and network. For X.509 based identities, this folder contains an **X.509 certificate**. This is the certificate a peer places in a transaction proposal response, for example, to indicate that the peer has endorsed it — which can subsequently be checked against the resulting transaction's endorsement policy at validation time.

  This folder is mandatory for local MSPs, and there must be exactly one X.509 certificate for the node. It is not used for channel MSPs.

- **`KeyStore` for Private Key:** This folder is defined for the local MSP of a peer or orderer node (or in an client's local MSP), and 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 private 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 MSPs** does not include this folder, as channel MSPs solely aim to offer identity validation functionalities and not signing abilities.

- **TLS Root CA:** This folder contains a list of self-signed X.509 certificates of the Root CAs trusted by this organization **for TLS communications**. 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 X.509 certificate in this folder.

- **TLS Intermediate CA:** This folder contains a list intermediate CA certificates CAs trusted by the organization represented by this MSP **for TLS communications**. 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.

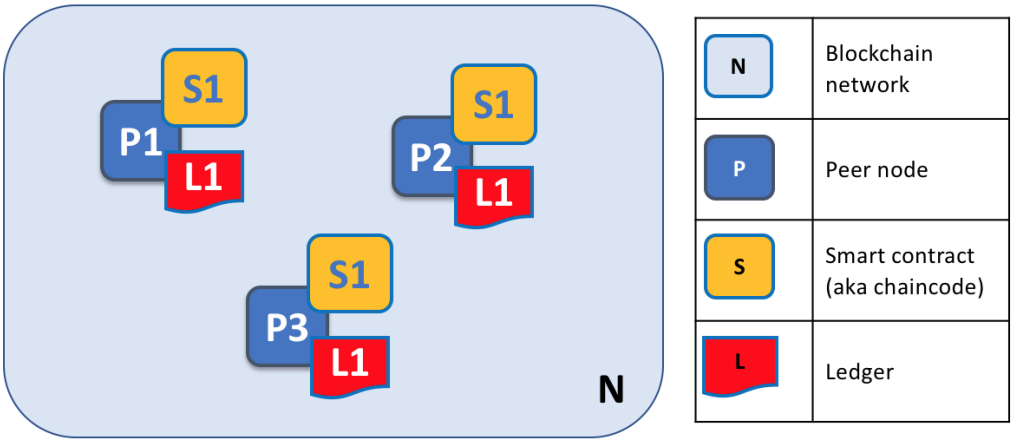
  For more information about TLS, click here.

If you've read this doc as well as our doc on Identity), you should have a pretty good grasp of how identities and membership 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.

Peers

A blockchain network is primarily comprised of a set of peer nodes. 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. 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 Hyperledger 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 topics. This topic focusses on peers, and their relationship to those other elements in a Hyperledger Fabric blockchain network.



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

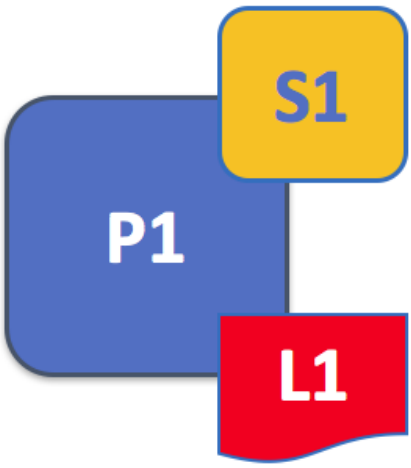
Peers can be created, started, stopped, reconfigured, and even deleted. They expose a set of APIs that enable administrators and applications to interact with the services that they provide. We'll learn more about these services in this topic.

A word on terminology

Hyperledger 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 this term. It's the same thing!

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 topic.

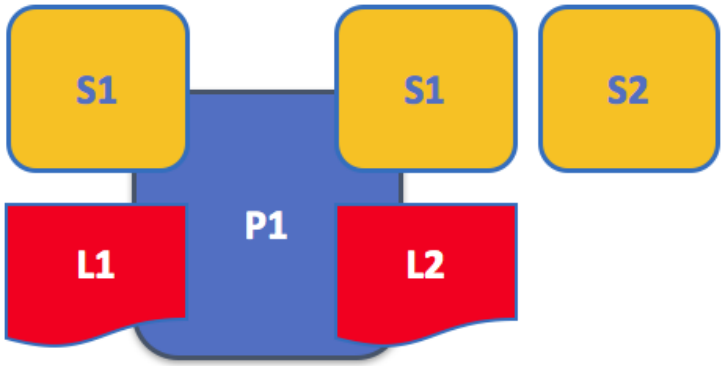


A peer hosts instances of ledgers and instances of chaincodes. In this example, P1 hosts an instance of ledger L1 and an instance of chaincode S1. There can be many ledgers and chaincodes hosted on an individual peer.

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 Hyperledger Fabric blockchain network. When a peer is first created, it has neither ledgers nor chaincodes. We'll see later how ledgers get created, and 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 peer configuration is to have a single ledger, but it's absolutely appropriate for a peer to host two or more ledgers when required.

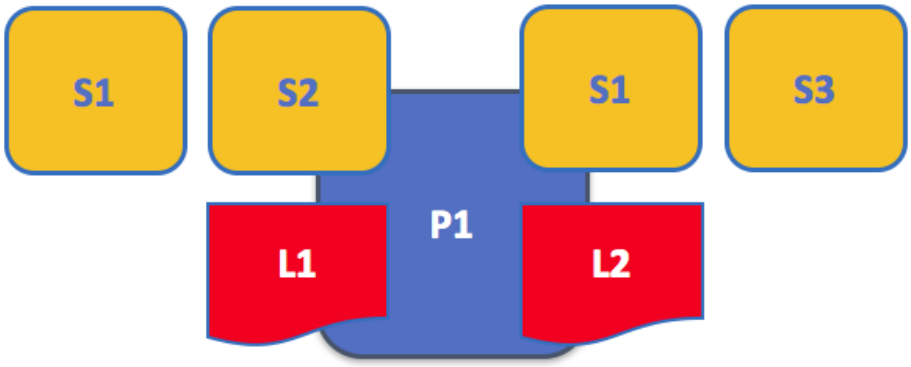


A peer hosting multiple ledgers. Peers host one or more ledgers, and each ledger has zero or more chaincodes that apply to them. In this example, we can see that the peer P1 hosts ledgers L1 and L2. Ledger L1 is accessed using chaincode S1. Ledger L2 on the other hand can be accessed using chaincodes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

Although it is perfectly possible for a peer to host a ledger instance without hosting any chaincodes which access it, it's very 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 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. L1 is accessed by chaincodes S1 and S2, whereas L2 is accessed by S3 and S1. We can see that S1 can access both L1 and L2.

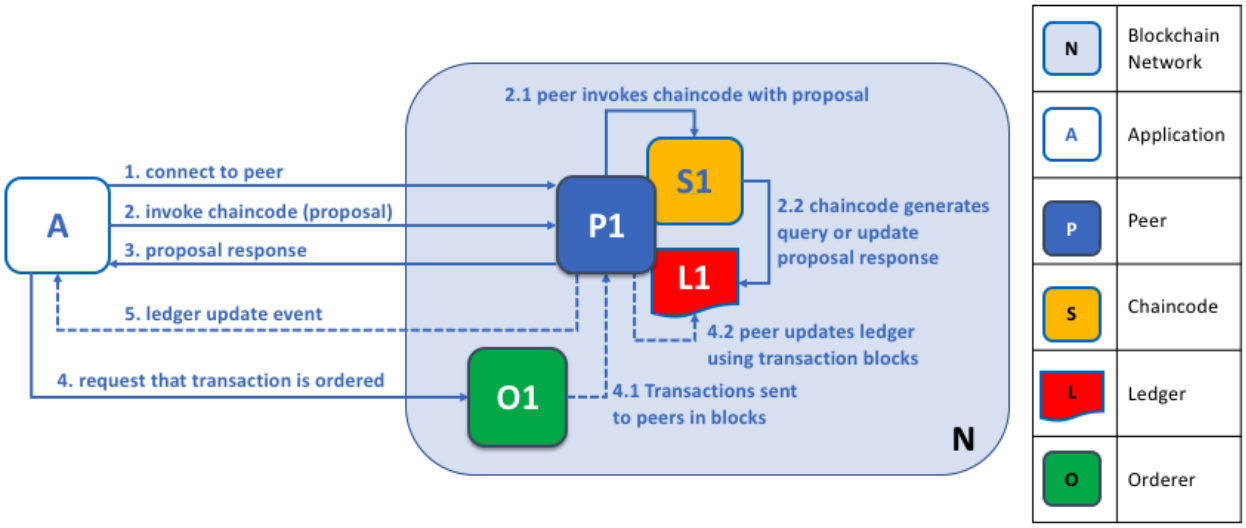
We'll see a little later why the concept of **channels** in Hyperldeger Fabric is important when hosting multiple ledgers or multiple chaincodes on a peer.

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 Hyperledger 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 Hyperledger 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 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 the 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 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 the all the responses, which it sends it 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 applying 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 because all the information required to satisfy the query is in the peer's local copy of the ledger. Peers do not consult with other peers in order to return a query to 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

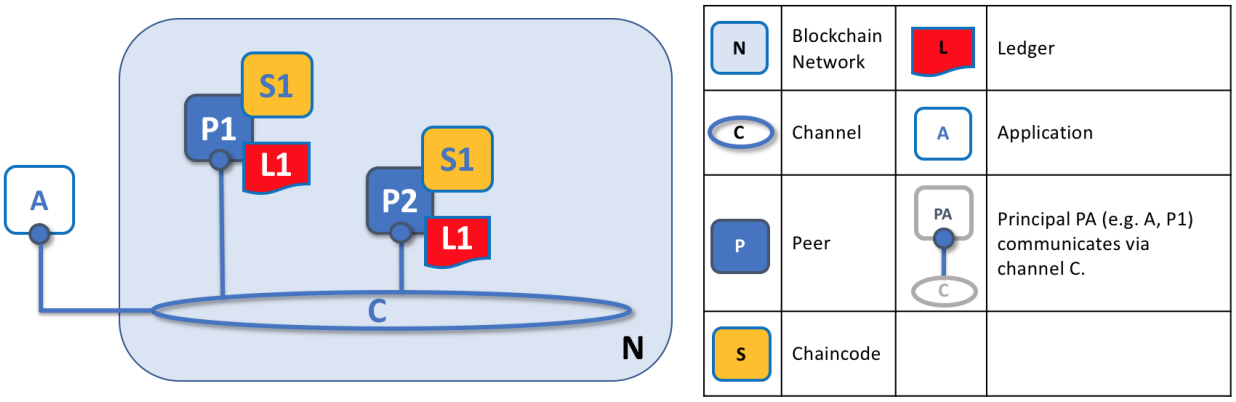
to other peers' prior agreement. The first extra step – four – requires that applications send an appropriate set of matching proposed updates to the entire network of peers as a transaction for commitment to their respective ledgers. This is achieved by the application using an **orderer** to package transactions into blocks, and distribute 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 topic, you'll learn more about the detailed nature of of this ordering process – and for a really detailed look at this process see the Transaction Flow topic.

Peers and Channels

Although this topic is about peers rather than channels, it's worth spending a little time understanding how peers interact with each other, and 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 come together to collectively share and manage identical copies of the ledger for 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 crossover between them. Nevertheless each group is its own entity, with "rules" of a kind.



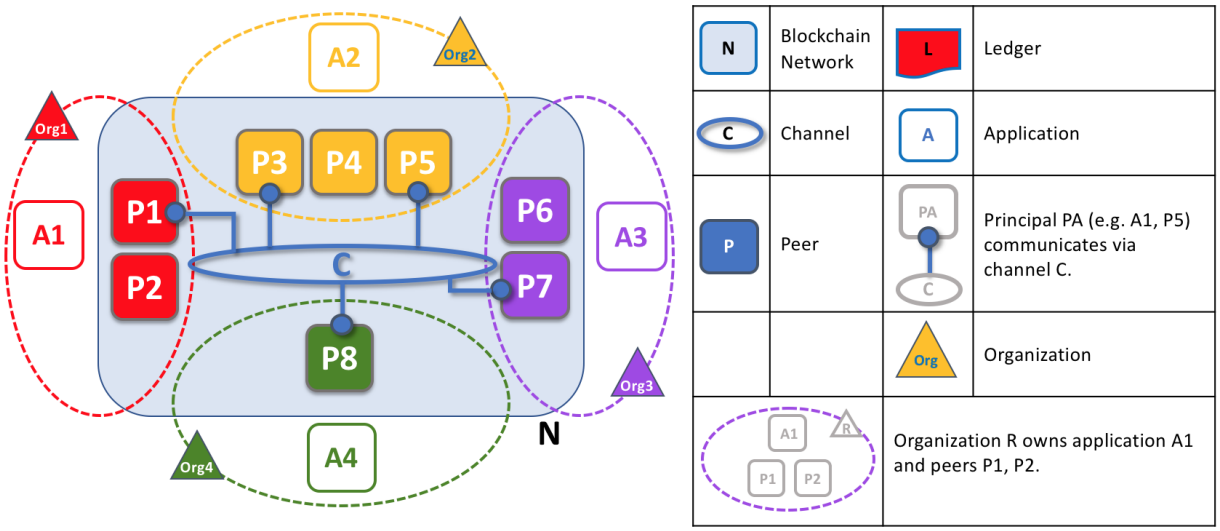
Channels allow a specific set of peers and applications to communicate with each other within a blockchain network. In this example, application A can communicate directly with peers P1 and P2 using channel C. You can think of the channel as a pathway for communications between particular applications 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.*

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 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.

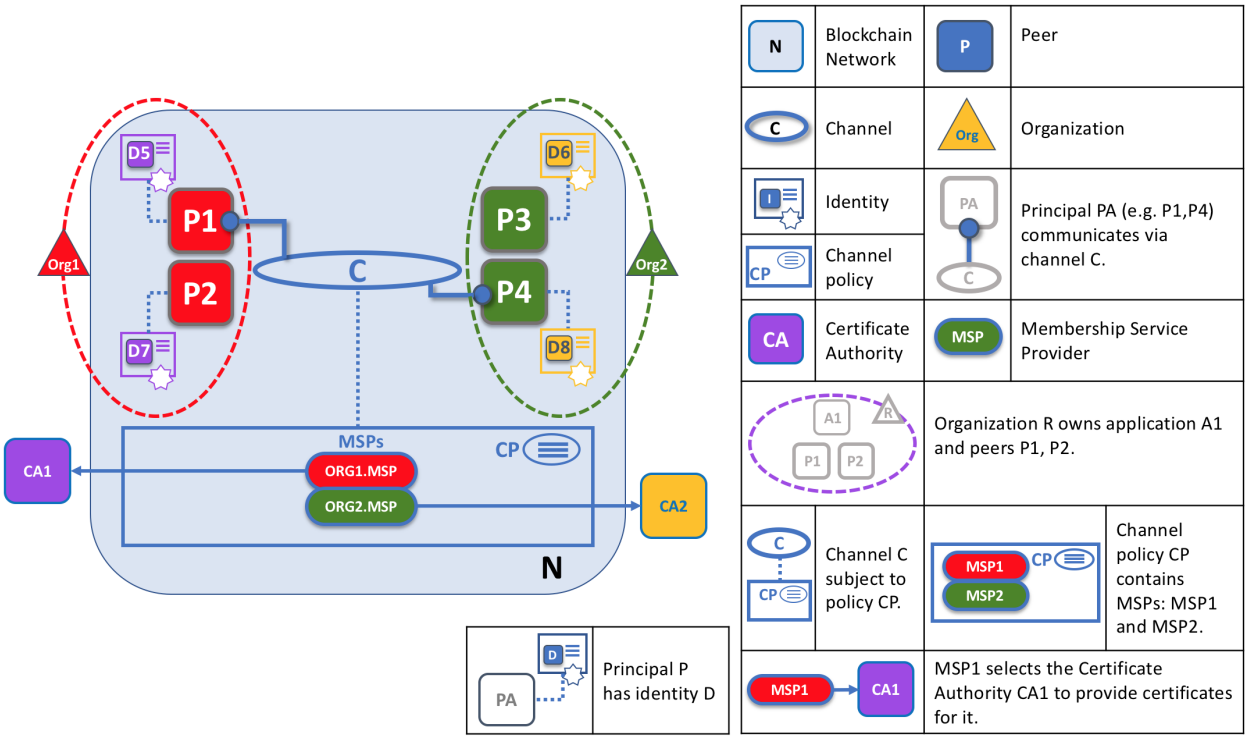
Applications either connect to peers in their organization, or peers in another organization, depending on the nature of the ledger interaction that's required. For ledger-query interactions, applications typically connect to their own organization's peers. For ledger-update interactions, we'll see later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

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 only be owned by a single organization, and is therefore associated with a single MSP. We'll learn more about peer access control later in this topic, and there's an entire topic 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.

And 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, 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 identity associated with it that identifies it as 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.

Peers and Orderers

We've seen that peers form a blockchain network, hosting ledgers and chaincodes 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 is mediated by special nodes called *orderers*, and it's these nodes to which we now turn our attention.

An update transaction is quite different to a query transaction because a single peer cannot, on its own, update the ledger – it 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* – and takes much longer to complete than a 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.

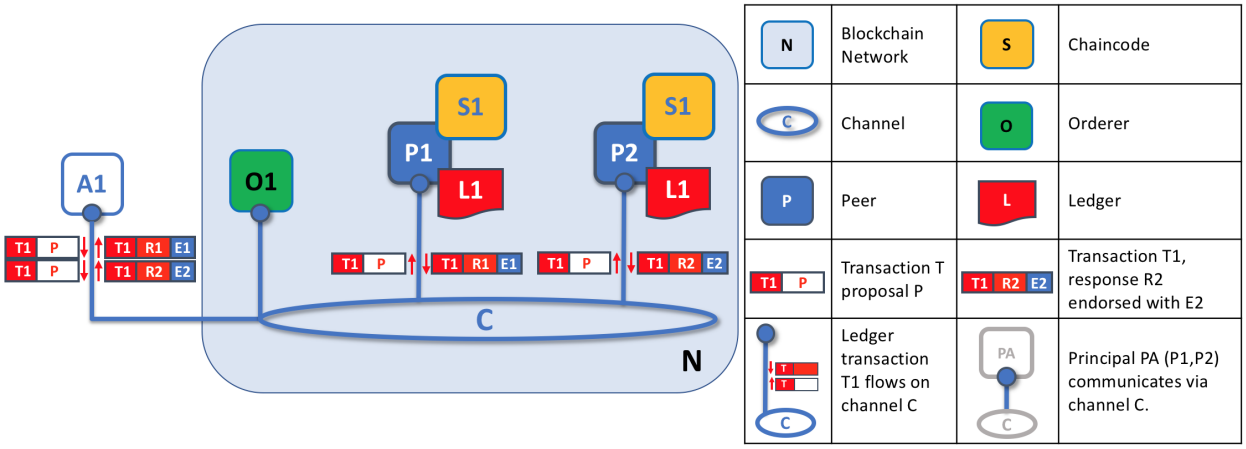
Specifically, applications that want to update the ledger are involved in a 3-phase process, which ensures that all the peers in a blockchain network keep their ledgers consistent with each other. In the first phase, applications work with a subset of *endorsing peers*, each of which provide an endorsement 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 final phase, these blocks are distributed back to every peer where each transaction is validated before being applied 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 peer 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 the peer signs it and returns 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.

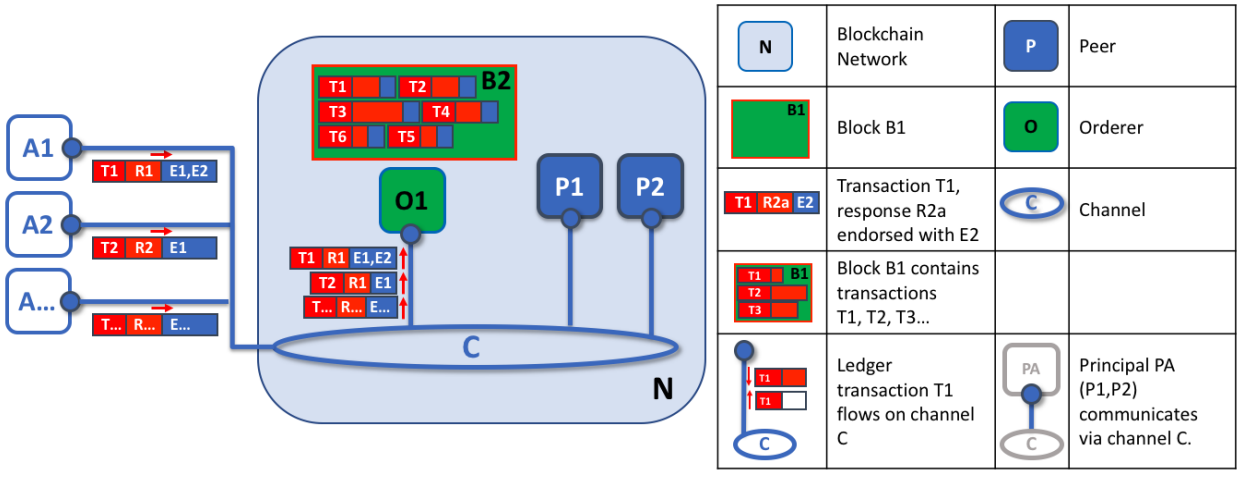
A peer endorses a proposal response by adding its digital signature, and signing the entire payload using its private key. This endorsement can be subsequently used to prove that this organization's peer generated a particular response. In our example, if peer P1 is owned by organization Org1, endorsement 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 a different time 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 topic, 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: Packaging

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. It orders each transaction relative to other transactions, and packages batches of transactions into blocks ready for distribution back to all peers connected to the orderer, including the original endorsing peers.



The first role of an orderer 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 node! (This example shows a very simplified orderer configuration.)

An orderer receives proposed ledger updates concurrently from many different applications in the network on a particular channel. Its job is to arrange these proposed updates into a well-defined sequence, and package them into *blocks* for subsequent distribution. These blocks will become the *blocks* of the blockchain! Once an orderer has generated a block of the desired size, or after a maximum elapsed time, it will be sent to all peers connected to it on a particular channel. We'll see how this block is processed in phase 3.

It's worth noting that the sequencing of transactions in a block is not necessarily the same as the order of arrival of transactions at the orderer! Transactions can be packaged in any order into a block, and it's this sequence that becomes the order of execution. What's important is that there **is** a strict order, rather than **what** that order is.

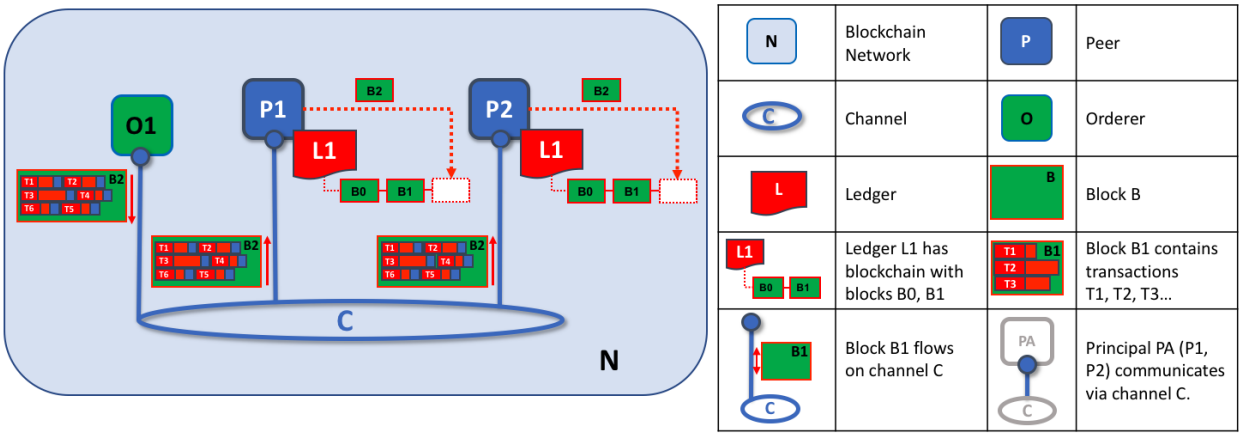
This strict ordering of transactions within blocks makes Hyperledger Fabric a little different to some other blockchains where the same transaction can be packaged into multiple different blocks. In Hyperledger Fabric, this cannot happen – the blocks generated by a collection of orderers are said to be **final** because once a transaction has been written to a block, its position in the ledger is immutably assured. Hyperledger Fabric's finality means that a disastrous occurrence known as a **ledger fork** cannot occur. Once transactions are captured in a block, history cannot be be rewritten for that transaction at a future point in time.

We can see also see that whereas peers host the ledger and chaincodes, orderers most definitely do not. Every transaction that arrives at an orderer is mechanically packaged in a block – the orderer makes no judgement as to the value of a transaction, it simply packages it. That's an important behavior of Hyperledger Fabric – all transactions are marshalled into in a strict order – transactions are never dropped or de-prioritized.

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, packaging them into blocks, ready for distribution.

Phase 3: Validation

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 applied 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 applied to the ledger. Failed transactions are retained for audit, but are not applied 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.

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 each peer's copy of the ledger is kept consistent across the network 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 only done in 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 to every peer in the channel, whether or not they endorsed the transaction. This specialization of endorsing peers is designed to help scalability.

Finally, every time a block is committed to 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 Hyperledger 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.

Ledger

The ledger is the sequenced, tamper-resistant record of all state transitions. 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 state. There is one ledger per channel. Each peer maintains a copy of the ledger for each channel of which they are a member.

Chain

The chain is a transaction log, structured as hash-linked blocks, where each block contains a sequence of N transactions. The block header includes a hash of the block's transactions, as well as a hash of the prior block's header. In this way, all transactions on the ledger are sequenced and cryptographically linked together. In other words, it is not possible to tamper with the ledger data, without breaking the hash links. The hash of the latest block represents every transaction that has come before, making it possible to ensure that all peers are in a consistent and trusted state.

The chain is stored on the peer file system (either local or attached storage), efficiently supporting the append-only nature of the blockchain workload.

State Database

The ledger's current state data represents the latest values for all keys ever included in the chain transaction log. Since current state represents all latest key values known to the channel, it is sometimes referred to as World State.

Chaincode invocations execute transactions against the current state data. To make these chaincode interactions extremely efficient, the latest values of all keys are stored in a state database. The state database is simply an indexed view into the chain's transaction log, it can therefore be regenerated from the chain at any time. The state database will automatically get recovered (or generated if needed) upon peer startup, before transactions are accepted.

State database options include LevelDB and CouchDB. LevelDB is the default state database embedded in the peer process and stores chaincode data as key-value pairs. CouchDB is an optional alternative external state database that provides addition query support when your chaincode data is modeled as JSON, permitting rich queries of the JSON content. See *CouchDB as the State Database* for more information on CouchDB.

Transaction Flow

At a high level, the transaction flow consists of a transaction proposal sent by an application client to specific endorsing peers. The endorsing peers verify the client signature, and execute a chaincode function to simulate the transaction. The output is the chaincode results, a set of key-value versions that were read in the chaincode (read set), and the set of keys/values that were written in chaincode (write set). The proposal response gets sent back to the client along with an endorsement signature.

The client assembles the endorsements into a transaction payload and broadcasts it to an ordering service. The ordering service delivers ordered transactions as blocks to all peers on a channel.

Before committal, peers will validate the transactions. First, they will check the endorsement policy to ensure that the correct allotment of the specified peers have signed the results, and they will authenticate the signatures against the transaction payload.

Secondly, peers will perform a versioning check against the transaction read set, to ensure data integrity and protect against threats such as double-spending. Hyperledger Fabric has concurrency control whereby transactions execute in parallel (by endorsers) to increase throughput, and upon commit (by all peers) each transaction is verified to ensure

that no other transaction has modified data it has read. In other words, it ensures that the data that was read during chaincode execution has not changed since execution (endorsement) time, and therefore the execution results are still valid and can be committed to the ledger state database. If the data that was read has been changed by another transaction, then the transaction in the block is marked as invalid and is not applied to the ledger state database. The client application is alerted, and can handle the error or retry as appropriate.

See the *Transaction Flow*, *Read-Write set semantics*, and *CouchDB as the State Database* topics for a deeper dive on transaction structure, concurrency control, and the state DB.

Use Cases

The Hyperledger Requirements WG is documenting a number of blockchain use cases and maintaining an inventory here.

CHAPTER 3

Tutorials

We offer tutorials to get you started with Hyperledger Fabric. The first is oriented to the Hyperledger Fabric **application developer**, *Writing Your First Application*. It takes you through the process of writing your first blockchain application for Hyperledger Fabric using the Hyperledger Fabric Node SDK.

The second tutorial is oriented towards the Hyperledger Fabric network operators, *Building Your First Network*. This one walks you through the process of establishing a blockchain network using Hyperledger Fabric and provides a basic sample application to test it out.

There are also tutorials for updating your channel, *Adding an Org to a Channel*, and upgrading your network to a later version of Hyperledger Fabric, *Upgrading Your Network Components*.

Finally, we offer two chaincode tutorials. One oriented to developers, *Chaincode for Developers*, and the other oriented to operators, *Chaincode for Operators*.

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.

Building Your First Network

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 see configuration and panic errors.

The build your first network (BYFN) scenario provisions a sample Hyperledger Fabric network consisting of two organizations, each maintaining two peer nodes, and a "solo" ordering service.

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 download and install the *Hyperledger Fabric Samples*. 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.

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 comprised of 4 peers representing two different organizations, and an orderer node. It will also launch a container to run a scripted execution that will join peers to a channel, deploy and instantiate chaincode and drive execution of transactions against the deployed chaincode.

Here's the help text for the `byfn.sh` script:

```
./byfn.sh --help
Usage:
byfn.sh up|down|restart|generate [-c <channel name>] [-t <timeout>] [-d <delay>] [-f
↪<docker-compose-file>] [-s <dbtype>]
byfn.sh -h|--help (print this message)
  -m <mode> - one of 'up', 'down', 'restart' or 'generate'
    - '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
  -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) or node
  -a - don't ask for confirmation before proceeding

  Typically, one would first generate the required certificates and
  genesis block, then bring up the network. e.g.:

      byfn.sh -m generate -c mychannel
      byfn.sh -m up -c mychannel -s couchdb
```

If you choose not to supply a channel name, then the script will use a default name of `mychannel` . The CLI timeout parameter (specified with the -t flag) is an optional value; if you choose not to set it, then the CLI will give up on query requests made after the default setting of 10 seconds.

Generate Network Artifacts

Ready to give it a go? Okay then! Execute the following command:

```
./byfn.sh -m 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 with channel 'mychannel' and CLI timeout of '10
↪'
Continue? [Y/n] y
proceeding ...
/Users/xxx/dev/fabric-samples/bin/cryptogen

############################################################
##### Generate certificates using cryptogen tool #########
############################################################
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 -m up
```

The above command will compile Golang chaincode images and spin up the corresponding containers. Go is the default chaincode language, however there is also support for Node.js 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 -m up -l node
```

Note: View the Hyperledger Fabric Shim documentation for more info on the node.js chaincode shim APIs.

Once again, you will be prompted as to whether you wish to continue or abort. Respond with a `y` or hit the return key:

```
Starting with channel 'mychannel' and CLI timeout of '10'
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 on peer1.org2 on channel 'mychannel' is successful␣
↪=====================

===================== 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 -m 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 `CORE_LOGGING_LEVEL` 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
    - CORE_LOGGING_LEVEL=DEBUG
    #- CORE_LOGGING_LEVEL=INFO
```

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.

Before running the tool, let's take a quick look at a snippet from the `crypto-config.yaml` . Pay specific attention to the "Name", "Domain" and "Specs" parameters under the `OrdererOrgs` header:

```
OrdererOrgs:
#---------------------------------------------------------
# Orderer
# ---------------------------------------------------------
- Name: Orderer
  Domain: example.com
  CA:
      Country: US
      Province: California
      Locality: San Francisco
  #   OrganizationalUnit: Hyperledger Fabric
  #   StreetAddress: address for org # default nil
  #   PostalCode: postalCode for org # default nil
  # ---------------------------------------------------------
  # "Specs" - See PeerOrgs below for complete description
# ---------------------------------------------------------
  Specs:
    - Hostname: orderer
# ---------------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
 # ---------------------------------------------------------
PeerOrgs:
# ---------------------------------------------------------
# Org1
# ---------------------------------------------------------
- Name: Org1
  Domain: org1.example.com
  EnableNodeOUs: true
```

The naming convention for a network entity is as follows - "{{.Hostname}}.{{.Domain}}". So using our ordering node as a reference point, we are left with an ordering node named - `orderer.example.com` that is tied to an MSP ID of `Orderer` . This file contains extensive documentation on the definitions and syntax. You can also refer to the *Membership Service Providers (MSP)* documentation for a deeper dive on MSP.

After we run the `cryptogen` tool, the generated certificates and keys will be saved to a folder titled `crypto-config` .

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 top of this file. You will notice that we have two unique headers. One for the orderer genesis block - `TwoOrgsOrdererGenesis` - and one for our channel - `TwoOrgsChannel`.

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.

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 TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/
↪genesis.block
```

You should see an output similar to the following in your terminal:

```
2017-10-26 19:21:56.301 EDT [common/tools/configtxgen] main -> INFO 001 Loading␣
↪configuration
2017-10-26 19:21:56.309 EDT [common/tools/configtxgen] doOutputBlock -> INFO 002␣
↪Generating genesis block
2017-10-26 19:21:56.309 EDT [common/tools/configtxgen] doOutputBlock -> INFO 003␣
↪Writing 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 this project.

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
```

You should see an ouput 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
```

Start 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 your network:

```
docker-compose -f docker-compose-cli.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.

The CLI container will stick around idle for 1000 seconds. If it's gone when you need it you can restart it with a simple command:

```
docker start cli
```

Environment variables

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, then you will need to provide these values accordingly. Inspect the `docker-compose-base.yaml` for the specific paths:

```
# 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
```

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:

```
root@0d78bb69300d:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

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:

```
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-ID.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-ID.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 *Environment variables* 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:7051 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:7051 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
```

Install & Instantiate Chaincode

Note: We will utilize a simple existing chaincode. To learn how to write your own chaincode, see the *Chaincode for Developers* tutorial.

Applications interact with the blockchain ledger through `chaincode`. As such we need to install the chaincode on every peer that will execute and endorse our transactions, and then instantiate the chaincode on the channel.

First, install the sample Go or Node.js chaincode onto one of the four peer nodes. These commands place the specified source code flavor onto our peer's filesystem.

Note: You can only install one version of the source code per chaincode name and version. The source code exists on the peer's file system in the context of chaincode name and version; it is language agnostic. Similarly the instantiated chaincode container will be reflective of whichever language has been installed on the peer.

Golang

```
# this installs the Go chaincode
peer chaincode install -n mycc -v 1.0 -p github.com/chaincode/chaincode_example02/go/
```

Node.js

```
# this installs the Node.js chaincode
# make note of the -l flag; we use this to specify the language
peer chaincode install -n mycc -v 1.0 -l node -p /opt/gopath/src/github.com/chaincode/
↪chaincode_example02/node/
```

Next, instantiate the chaincode on the channel. This will initialize the chaincode on the channel, set the endorsement policy for the chaincode, and launch a chaincode container for the targeted peer. Take note of the `-P` argument. This is our policy where we specify the required level of endorsement for a transaction against this chaincode to be validated.

In the command below you'll notice that we specify our policy as `-P "OR ('Org0MSP.peer','Org1MSP.peer')"` . This means that we need "endorsement" from a peer belonging to Org1 **OR** Org2 (i.e. only one endorsement). If we changed the syntax to `AND` then we would need two endorsements.

Golang

```
# be sure to replace the $CHANNEL_NAME environment variable if you have not exported␣
↪it
# if you did not install your chaincode with a name of mycc, then modify that␣
↪argument as well

peer chaincode instantiate -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␣
↪mycc -v 1.0 -c '{"Args":["init","a", "100", "b","200"]}' -P "OR ('Org1MSP.peer',
↪'Org2MSP.peer')"
```

Node.js

Note: The instantiation of the Node.js chaincode will take roughly a minute. The command is not hanging; rather it is installing the fabric-shim layer as the image is being compiled.

```
# be sure to replace the $CHANNEL_NAME environment variable if you have not exported␣
↪it
# if you did not install your chaincode with a name of mycc, then modify that␣
↪argument as well
# notice that we must pass the -l flag after the chaincode name to identify the␣
↪language

peer chaincode instantiate -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␣
↪mycc -l node -v 1.0 -c '{"Args":["init","a", "100", "b","200"]}' -P "OR ('Org1MSP.
↪peer','Org2MSP.peer')"
```

See the endorsement policies documentation for more details on policy implementation.

If you want additional peers to interact with ledger, then you will need to join them to the channel, and install the same name, version and language of the chaincode source onto the appropriate peer's filesystem. 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 compile.

Once the chaincode has been instantiated on the channel, we can forgo the `l` flag. We need only pass in the channel identifier and name of the chaincode.

Query

Let's query for the value of `a` to make sure the chaincode was properly instantiated 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 --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 -c '{"Args":["invoke","a","b","10"]}'
```

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 reveal `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
```

Feel free to start over and manipulate the key value pairs and subsequent invocations.

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 - **chaincode_example02** - is installed on `peer0.org1.example.com` and `peer0.org2.example.com`
- The chaincode is then "instantiated" on `peer0.org2.example.com`. Instantiation adds the chaincode to the channel, 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 "instantiation" results in a container by the name of `dev-peer0.org2.example.com-mycc-1.0` starting.
- The instantiation also passes in an argument for the endorsement policy. The policy is defined as `-P "OR ('Org1MSP.peer','Org2MSP.peer')"`, meaning that any transaction must be endorsed by a peer tied to Org1 or Org2.
- A query against the value of "a" is issued to `peer0.org1.example.com`. The chaincode was previously installed on `peer0.org1.example.com`, so this will start a container for Org1 peer0 by the name of `dev-peer0.org1.example.com-mycc-1.0`. The result of the query is also 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` to move "10" from "a" to "b"
- The chaincode is then 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 it has already been instantiated.

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 on peer1.org2 on channel 'mychannel' is successful␣
↪=====================

===================== All GOOD, BYFN execution completed =====================


 _____   _   _   ____
| ____| | \ | | |  _ \
|  _|   |  \| | | | | |
| |___  | |\  | | |_| |
|_____| |_| \_| |____/
```

You can scroll through these logs to see the various transactions.

How can I see the chaincode logs?

Inspect the individual chaincode containers to see the separate transactions executed against each container. Here is the combined output from each container:

```
$ docker logs dev-peer0.org2.example.com-mycc-1.0
04:30:45.947 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Init
Aval = 100, Bval = 200

$ docker logs dev-peer0.org1.example.com-mycc-1.0
04:31:10.569 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Invoke
Query Response:{"Name":"a","Amount":"100"}
ex02 Invoke
Aval = 90, Bval = 210

$ docker logs dev-peer1.org2.example.com-mycc-1.0
04:31:30.420 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Invoke
Query Response:{"Name":"a","Amount":"90"}
```

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.

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 up -d
```

chaincode_example02 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 **chaincode_example02** 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, (e.g. **marbles02**). 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:

- Install and instantiate the chaincode on `peer0.org1.example.com` :

```
# be sure to modify the $CHANNEL_NAME variable accordingly for the instantiate command

peer chaincode install -n marbles -v 1.0 -p github.com/chaincode/marbles02/go
peer chaincode instantiate -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 -v 1.0 -c '{"Args":["init"]}' -P "OR ('Org0MSP.peer','Org1MSP.peer')"
```

- 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 -c '{"Args":["initMarble","marble1","blue","35","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 -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 -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 -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 -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 -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}}]
```

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.

In addition, CouchDB falls into the AP-type (Availability and Partition Tolerance) of the CAP theorem. It uses a master-master replication model with `Eventual Consistency` . More information can be found on the Eventual Consistency page of the CouchDB documentation. However, under each fabric peer, there is no database replicas, writes to database are guaranteed consistent and durable (not `Eventual Consistency`).

CouchDB is the first external pluggable state database for Fabric, and there could and should be other external database options. For example, IBM enables the relational database for its blockchain. And the CP-type (Consistency and Partition Tolerance) databases may also in need, so as to enable data consistency without application level guarantee.

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
```

Troubleshooting

- Always start your network fresh. Use the following command to remove artifacts, crypto, containers and chaincode images:

```
./byfn.sh -m 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, instantiate, 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 -m 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` .

Note: If you continue to see errors, share your logs on the **fabric-questions** channel on Hyperledger Rocket Chat or on StackOverflow.

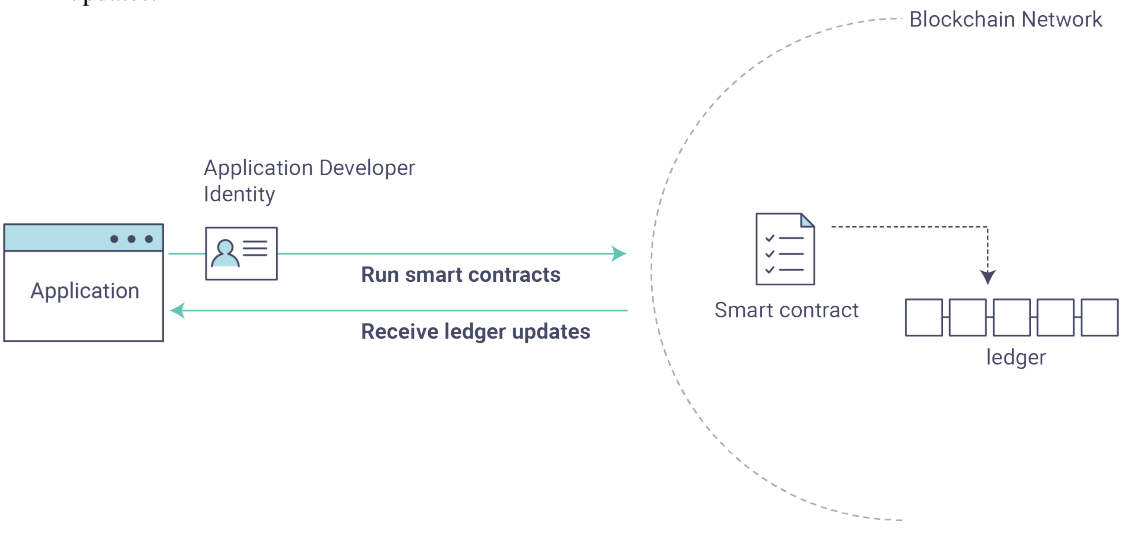
Writing Your First Application

Note: If you're not yet familiar with the fundamental architecture of a Fabric network, you may want to visit the *Introduction* and *Building Your First Network* documentation prior to continuing.

In this section we'll be looking at a handful of sample programs to see how Fabric apps work. These apps (and the smart contract they use) – collectively known as `fabcar` – provide a broad demonstration of Fabric functionality. Notably, we will show the process for interacting with a Certificate Authority and generating enrollment certificates, after which we will leverage these identities to query and update a ledger.

We'll go through three principle steps:

1. Setting up a development environment. Our application needs a network to interact with, so we'll download one stripped down to just the components we need for registration/enrollment, queries and updates:



2. Learning the parameters of the sample smart contract our app will use. Our smart contract contains various functions that allow us to interact with the ledger in different ways. We'll go in and inspect that smart contract to learn about the functions our applications will be using.

3. Developing the applications to be able to query and update assets on the ledger. We'll get into the app code itself (our apps have been written in Javascript) and manually manipulate the variables to run different kinds of queries and updates.

After completing this tutorial you should have a basic understanding of how an application is programmed in conjunction with a smart contract to interact with the ledger (i.e. the peer) on a Fabric network.

Setting up your Dev Environment

If you've already run through *Building Your First Network*, you should have your dev environment setup and will have downloaded *fabric-samples* as well as the accompanying artifacts. To run this tutorial, what you need to do now is tear down any existing networks you have, which you can do by issuing the following:

```
./byfn.sh -m down
```

If you don't have a development environment and the accompanying artifacts for the network and applications, visit the *Prerequisites* page and ensure you have the necessary dependencies installed on your machine.

Next, visit the *Hyperledger Fabric Samples* page and follow the provided instructions. Return to this tutorial once you have cloned the `fabric-samples` repository, and downloaded the latest stable Fabric images and available utilities.

At this point everything should be installed. Navigate to the `fabcar` subdirectory within your `fabric-samples` repository and take a look at what's inside:

```
cd fabric-samples/fabcar && ls
```

You should see the following:

```
enrollAdmin.js     invoke.js       package.json    query.js        registerUser.js␣
↪startFabric.sh
```

Before starting we also need to do a little housekeeping. Run the following command to kill any stale or active containers:

```
docker rm -f $(docker ps -aq)
```

Clear any cached networks:

```
# Press 'y' when prompted by the command

docker network prune
```

And lastly if you've already run through this tutorial, you'll also want to delete the underlying chaincode image for the `fabcar` smart contract. If you're a user going through this content for the first time, then you won't have this chaincode image on your system:

```
docker rmi dev-peer0.org1.example.com-fabcar-1.0-
↪5c906e402ed29f20260ae42283216aa75549c571e2e380f3615826365d8269ba
```

Install the clients & launch the network

Note: The following instructions require you to be in the `fabcar` subdirectory within your local clone of the `fabric-samples` repo. Remain at the root of this subdirectory for the remainder of this tutorial.

Run the following command to install the Fabric dependencies for the applications. We are concerned with `fabric-ca-client` which will allow our app(s) to communicate with the CA server and retrieve identity material, and with `fabric-client` which allows us to load the identity material and talk to the peers and ordering service.

```
npm install
```

Launch your network using the `startFabric.sh` shell script. This command will spin up our various Fabric entities and launch a smart contract container for chaincode written in Golang:

```
./startFabric.sh
```

You also have the option of running this tutorial against chaincode written in Node.js. If you'd like to pursue this route, issue the following command instead:

```
./startFabric.sh node
```

Note: Be aware that the Node.js chaincode scenario will take roughly 90 seconds to complete; perhaps longer. The script is not hanging, rather the increased time is a result of the fabric-shim being installed as the chaincode image is being built.

Alright, now that you've got a sample network and some code, let's take a look at how the different pieces fit together.

How Applications Interact with the Network

For a more in-depth look at the components in our `fabcar` network (and how they're deployed) as well as how applications interact with those components on more of a granular level, see understand_fabcar_network.

Developers more interested in seeing what applications **do** – as well as looking at the code itself to see how an application is constructed – should continue. For now, the most important thing to know is that applications use a software development kit (SDK) to access the **APIs** that permit queries and updates to the ledger.

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.

To stream your CA logs, split your terminal or open a new shell and issue the following:

```
docker logs -f ca.example.com
```

Now hop back to your terminal with the `fabcar` content...

When we launched our network, an admin user – `admin` – was registered with our Certificate Authority. Now we need to send an enroll call to the CA server and retrieve the enrollment certificate (eCert) for this user. We won't delve into enrollment details here, but suffice it to say that the SDK and by extension our applications need this cert in order to form a user object for the admin. We will then use this admin object to subsequently register and enroll a new user. Send the admin enroll call to the CA server:

```
node enrollAdmin.js
```

This program will invoke a certificate signing request (CSR) and ultimately output an eCert and key material into a newly created folder – `hfc-key-store` – at the root of this project. Our apps will then look to this location when they need to create or load the identity objects for our various users.

Register and Enroll `user1`

With our newly generated admin eCert, we will now communicate with the CA server once more to register and enroll a new user. This user – `user1` – will be the identity we use when querying and updating the ledger. It's important to note here that it is the `admin` identity that is issuing the registration and enrollment calls for our new user (i.e. this user is acting in the role of a registrar). Send the register and enroll calls for `user1` :

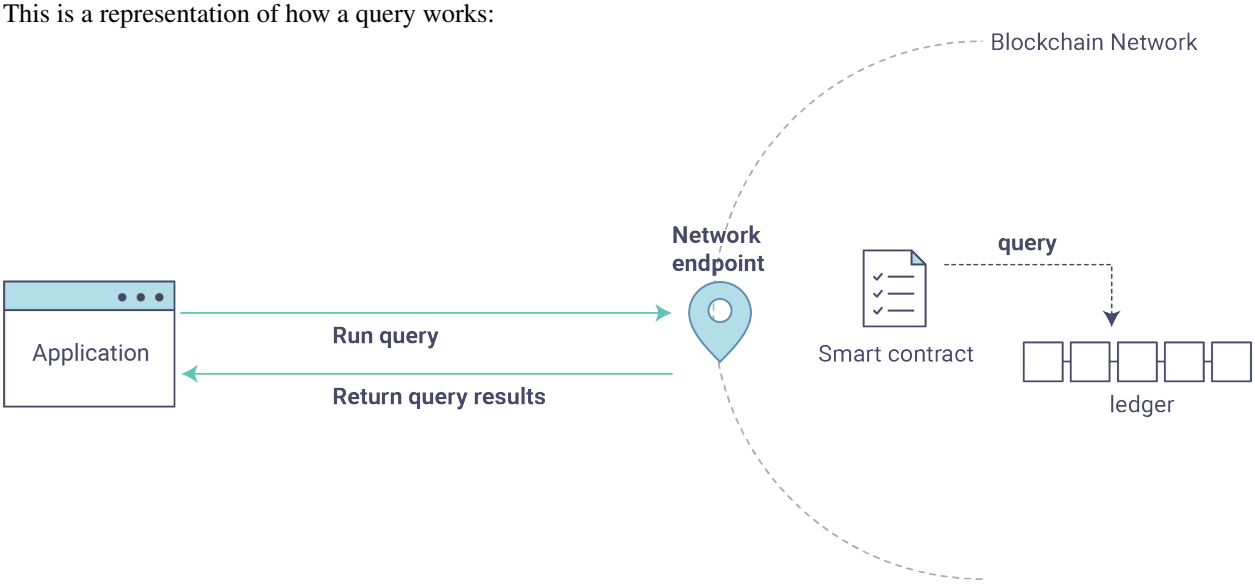
```
node registerUser.js
```

Similar to the admin enrollment, this program invokes a CSR and outputs the keys and eCert into the `hfc-key-store` subdirectory. So now we have identity material for two separate users – `admin` & `user1` . Time to interact with the ledger...

Querying the Ledger

Queries are how you read data from the ledger. This data is stored as a series of key-value pairs, and you can query for the value of a single key, multiple keys, or – if the ledger is written in a rich data storage format like JSON – perform complex searches against it (looking for all assets that contain certain keywords, for example).

This is a representation of how a query works:



First, let's run our `query.js` program to return a listing of all the cars on the ledger. We will use our second identity – `user1` – as the signing entity for this application. The following line in our program specifies `user1` as the signer:

```
fabric_client.getUserContext('user1', true);
```

Recall that the `user1` enrollment material has already been placed into our `hfc-key-store` subdirectory, so we simply need to tell our application to grab that identity. With the user object defined, we can now proceed with reading from the ledger. A function that will query all the cars, `queryAllCars` , is pre-loaded in the app, so we can simply run the program as is:

```
node query.js
```

It should return something like this:

```
Successfully loaded user1 from persistence
Query has completed, checking results
Response is  [{"Key":"CAR0", "Record":{"colour":"blue","make":"Toyota","model":"Prius
↪","owner":"Tomoko"}},
```

```
{"Key":"CAR1",   "Record":{"colour":"red","make":"Ford","model":"Mustang","owner":
↪"Brad"}},
{"Key":"CAR2", "Record":{"colour":"green","make":"Hyundai","model":"Tucson","owner":
↪"Jin Soo"}},
{"Key":"CAR3", "Record":{"colour":"yellow","make":"Volkswagen","model":"Passat","owner
↪":"Max"}},
{"Key":"CAR4", "Record":{"colour":"black","make":"Tesla","model":"S","owner":"Adriana
↪"}},
{"Key":"CAR5", "Record":{"colour":"purple","make":"Peugeot","model":"205","owner":
↪"Michel"}},
{"Key":"CAR6", "Record":{"colour":"white","make":"Chery","model":"S22L","owner":"Aarav
↪"}},
{"Key":"CAR7", "Record":{"colour":"violet","make":"Fiat","model":"Punto","owner":"Pari
↪"}},
{"Key":"CAR8", "Record":{"colour":"indigo","make":"Tata","model":"Nano","owner":
↪"Valeria"}},
{"Key":"CAR9", "Record":{"colour":"brown","make":"Holden","model":"Barina","owner":
↪"Shotaro"}}]
```

These are the 10 cars. A black Tesla Model S owned by Adriana, a red Ford Mustang owned by Brad, a violet Fiat Punto owned by Pari, and so on. The ledger is key-value based and, in our implementation, the key is `CAR0` through `CAR9` . This will become particularly important in a moment.

Let's take a closer look at this program. Use an editor (e.g. atom or visual studio) and open `query.js` .

The initial section of the application defines certain variables such as channel name, cert store location and network endpoints. In our sample app, these variables have been baked-in, but in a real app these variables would have to be specified by the app dev.

```
var channel = fabric_client.newChannel('mychannel');
var peer = fabric_client.newPeer('grpc://localhost:7051');
channel.addPeer(peer);

var member_user = null;
var store_path = path.join(__dirname, 'hfc-key-store');
console.log('Store path:'+store_path);
var tx_id = null;
```

This is the chunk where we construct our query:

```
// queryCar chaincode function - requires 1 argument, ex: args: ['CAR4'],
// queryAllCars chaincode function - requires no arguments , ex: args: [''],
const request = {
  //targets : --- letting this default to the peers assigned to the channel
  chaincodeId: 'fabcar',
  fcn: 'queryAllCars',
  args: ['']
};
```

When the application ran, it invoked the `fabcar` chaincode on the peer, ran the `queryAllCars` function within it, and passed no arguments to it.

To take a look at the available functions within our smart contract, navigate to the `chaincode/fabcar/go` sub-directory at the root of `fabric-samples` and open `fabcar.go` in your editor.

Note: These same functions are defined within the Node.js version of the `fabcar` chaincode.

You'll see that we have the following functions available to call: `initLedger` , `queryCar` , `queryAllCars` , `createCar` , and `changeCarOwner` .

Let's take a closer look at the `queryAllCars` function to see how it interacts with the ledger.

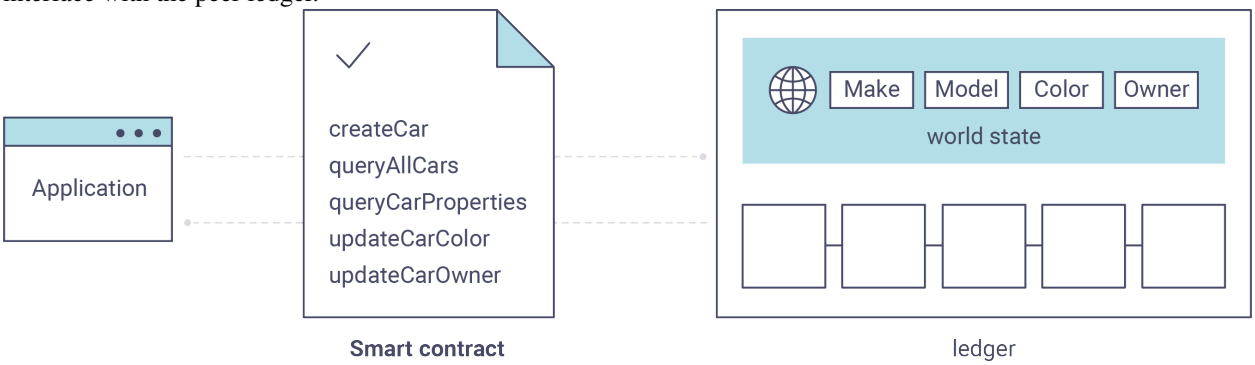
```
func (s *SmartContract) queryAllCars(APIstub shim.ChaincodeStubInterface) sc.Response
↪{

      startKey := "CAR0"
      endKey := "CAR999"

      resultsIterator, err := APIstub.GetStateByRange(startKey, endKey)
```

This defines the range of `queryAllCars` . Every car between `CAR0` and `CAR999` – 1,000 cars in all, assuming every key has been tagged properly – will be returned by the query.

Below is a representation of how an app would call different functions in chaincode. Each function must be coded against an available API in the chaincode shim interface, which in turn allows the smart contract container to properly interface with the peer ledger.



We can see our `queryAllCars` function, as well as one called `createCar` , that will allow us to update the ledger and ultimately append a new block to the chain in a moment.

But first, go back to the `query.js` program and edit the constructor request to query `CAR4` . We do this by changing the function in `query.js` from `queryAllCars` to `queryCar` and passing `CAR4` as the specific key.

The `query.js` program should now look like this:

```
const request = {
  //targets : --- letting this default to the peers assigned to the channel
  chaincodeId: 'fabcar',
  fcn: 'queryCar',
  args: ['CAR4']
};
```

Save the program and navigate back to your `fabcar` directory. Now run the program again:

```
node query.js
```

You should see the following:

```
{"colour":"black","make":"Tesla","model":"S","owner":"Adriana"}
```

If you go back and look at the result from when we queried every car before, you can see that `CAR4` was Adriana's black Tesla model S, which is the result that was returned here.

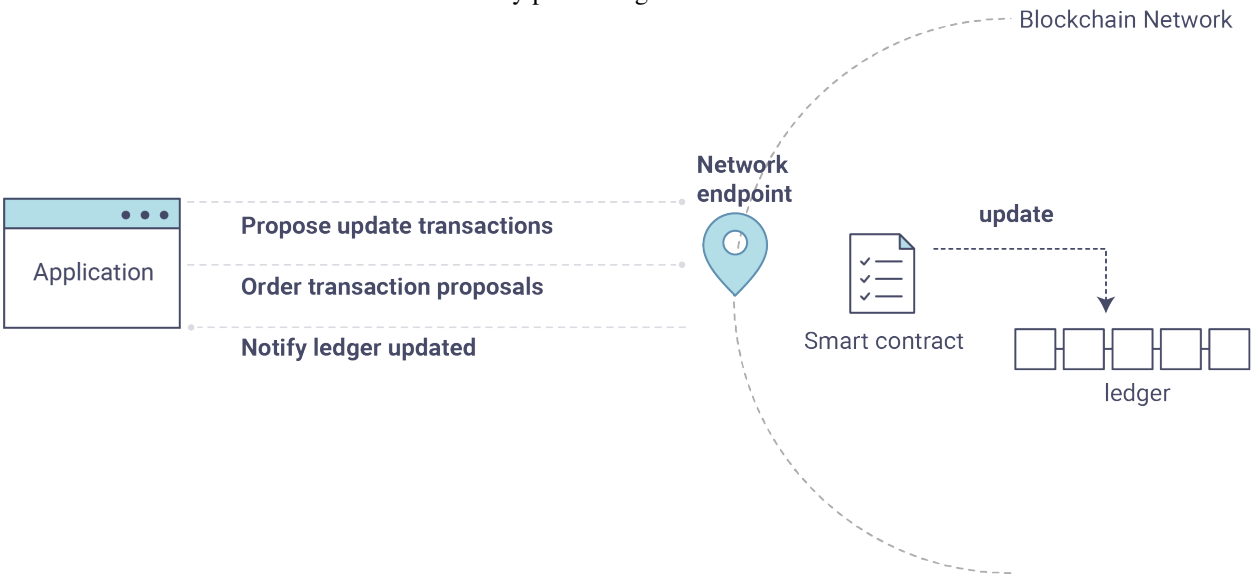
Using the `queryCar` function, we can query against any key (e.g. `CAR0`) and get whatever make, model, color, and owner correspond to that car.

Great. At this point you should be comfortable with the basic query functions in the smart contract and the handful of parameters in the query program. Time to update the ledger...

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 car.

Below we can see how this process works. An update is proposed, endorsed, then returned to the application, which in turn sends it to be ordered and written to every peer's ledger:



Our first update to the ledger will be to create a new car. We have a separate Javascript program – `invoke.js` – that we will use to make updates. Just as with queries, use an editor to open the program and navigate to the code block where we construct our invocation:

```
// createCar chaincode function - requires 5 args, ex: args: ['CAR12', 'Honda',
↪'Accord', 'Black', 'Tom'],
// changeCarOwner chaincode function - requires 2 args , ex: args: ['CAR10', 'Barry'],
// must send the proposal to endorsing peers
var request = {
  //targets: let default to the peer assigned to the client
  chaincodeId: 'fabcar',
  fcn: '',
  args: [''],
  chainId: 'mychannel',
  txId: tx_id
};
```

You'll see that we can call one of two functions – `createCar` or `changeCarOwner` . First, let's create a red Chevy Volt and give it to an owner named Nick. We're up to `CAR9` on our ledger, so we'll use `CAR10` as the identifying key here. Edit this code block to look like this:

```
var request = {
  //targets: let default to the peer assigned to the client
  chaincodeId: 'fabcar',
  fcn: 'createCar',
  args: ['CAR10', 'Chevy', 'Volt', 'Red', 'Nick'],
  chainId: 'mychannel',
```

```
  txId: tx_id
};
```

Save it and run the program:

```
node invoke.js
```

There will be some output in the terminal about `ProposalResponse` and promises. However, all we're concerned with is this message:

```
The transaction has been committed on peer localhost:7053
```

To see that this transaction has been written, go back to `query.js` and change the argument from `CAR4` to `CAR10` .

In other words, change this:

```
const request = {
  //targets : --- letting this default to the peers assigned to the channel
  chaincodeId: 'fabcar',
  fcn: 'queryCar',
  args: ['CAR4']
};
```

To this:

```
const request = {
  //targets : --- letting this default to the peers assigned to the channel
  chaincodeId: 'fabcar',
  fcn: 'queryCar',
  args: ['CAR10']
};
```

Save once again, then query:

```
node query.js
```

Which should return this:

```
Response is  {"colour":"Red","make":"Chevy","model":"Volt","owner":"Nick"}
```

Congratulations. You've created a car!

So now that we've done that, let's say that Nick is feeling generous and he wants to give his Chevy Volt to someone named Dave.

To do this go back to `invoke.js` and change the function from `createCar` to `changeCarOwner` and input the arguments like this:

```
var request = {
  //targets: let default to the peer assigned to the client
  chaincodeId: 'fabcar',
  fcn: 'changeCarOwner',
  args: ['CAR10', 'Dave'],
  chainId: 'mychannel',
  txId: tx_id
};
```

The first argument – `CAR10` – reflects 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 `CAR10` key:

```
node query.js
```

It should return this result:

```
Response is  {"colour":"Red","make":"Chevy","model":"Volt","owner":"Dave"}
```

The ownership of `CAR10` has been changed from Nick to Dave.

Note: In a real world application the chaincode 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.

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 the network. 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.

In subsequent documents we'll learn how to actually **write** a smart contract and how some of these more low level application functions can be leveraged (especially relating to identity and membership services).

Additional Resources

The Hyperledger Fabric Node SDK repo is an excellent resource for deeper documentation and sample code. You can also consult the Fabric community and component experts on Hyperledger Rocket Chat.

Adding an Org to a Channel

Note: Ensure that you have downloaded the appropriate images and binaries as outlined in *Hyperledger Fabric Samples* 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). In particular, your version of the `fabric-samples` folder must include the `eyfn.sh` ("Extending Your First Network") script and its related scripts.

This tutorial serves as an extension to the *Building Your First Network* (BYFN) tutorial, and will demonstrate the addition of a new organization – `Org3` – to the application channel (`mychannel`) autogenerated by BYFN. It assumes a strong understanding of BYFN, including the usage and functionality of the aforementioned utilities.

While we will focus solely on the integration of a new organization here, the same approach can be adopted when performing 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).

Note: Make sure the automated `byfn.sh` script runs without error on your machine before continuing. If you have exported your binaries and the related tools (`cryptogen` , `configtxgen` , etc) into your PATH variable, you'll be able to modify the commands accordingly without passing the fully qualified path.

Setup the Environment

We will be operating from the root of the `first-network` subdirectory within your local clone of `fabric-samples` . Change into that directory now. You will also want to open a few extra terminals for ease of use.

First, use the `byfn.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:

```
./byfn.sh -m down
```

Now generate the default BYFN artifacts:

```
./byfn.sh -m generate
```

And launch the network making use of the scripted execution within the CLI container:

```
./byfn.sh -m up
```

Now that you have a clean version of BYFN running on your machine, you have two different paths you can pursue. First, we offer a fully commented script that will carry out a config transaction update to bring Org3 into the network.

Also, we will show a "manual" version of the same process, showing each step and explaining what it accomplishes (since we show you how to bring down your network before this manual process, you could also run the script and then look at each step).

Bring Org3 into the Channel with the Script

You should be in `first-network` . To use the script, simply issue the following:

```
./eyfn.sh up
```

The output here is well worth reading. You'll see the Org3 crypto material being added, the config update being created and signed, and then chaincode being installed to allow Org3 to execute ledger queries.

If everything goes well, you'll get this message:

```
========= All GOOD, EYFN test execution completed ===========
```

`eyfn.sh` can be used with the same Node.js chaincode and database options as `byfn.sh` by issuing the following (instead of `./byfn.sh -m -up`):

```
./byfn.sh up -c testchannel -s couchdb -l node
```

And then:

```
./eyfn.sh up -c testchannel -s couchdb -l node
```

For those who want to take a closer look at this process, the rest of the doc will show you each command for making a channel update and what it does.

Bring Org3 into the Channel Manually

Note: The manual steps outlined below assume that the `CORE_LOGGING_LEVEL` in the `cli` and *Org3cli*' containers is set to `DEBUG`.

For the `cli` container, 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
    #- CORE_LOGGING_LEVEL=INFO
    - CORE_LOGGING_LEVEL=DEBUG
```

For the `Org3cli` container, you can set this by modifying the `docker-compose-org3.yaml` file in the `first-network` directory. e.g.

```
Org3cli:
  container_name: Org3cli
  image: hyperledger/fabric-tools:$IMAGE_TAG
  tty: true
  stdin_open: true
  environment:
    - GOPATH=/opt/gopath
    - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
    #- CORE_LOGGING_LEVEL=INFO
    - CORE_LOGGING_LEVEL=DEBUG
```

If you've used the `eyfn.sh` script, you'll need to bring your network down. This can be done by issuing:

```
./eyfn.sh down
```

This will bring down the network, delete all the containers and undo what we've done to add Org3.

When the network is down, bring it back up again.

```
./byfn.sh -m generate
```

Then:

```
./byfn.sh -m up
```

This will bring your network back to the same state it was in before you executed the `eyfn.sh` script.

Now we're ready to add Org3 manually. As a first step, we'll need to generate Org3's crypto material.

Generate the Org3 Crypto Material

In another terminal, change into the `org3-artifacts` subdirectory from `first-network`.

```
cd org3-artifacts
```

There are two `yaml` files of interest here: `org3-crypto.yaml` and `configtx.yaml`. First, generate the crypto material for Org3:

```
../../bin/cryptogen generate --config=./org3-crypto.yaml
```

This command reads in our new crypto `yaml` file – `org3-crypto.yaml` – and leverages `cryptogen` to generate the keys and certificates for an Org3 CA as well as two peers bound to this new Org. As with the BYFN implementation, this crypto material is put into a newly generated `crypto-config` folder within the present working directory (in our case, `org3-artifacts`).

Now use the `configtxgen` utility to print out the Org3-specific configuration material in JSON. 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 > ../channel-
↪artifacts/org3.json
```

The above command creates a JSON file – `org3.json` – and outputs it into the `channel-artifacts` subdirectory at the root of `first-network`. This file contains the policy definitions for Org3, as well as three important certificates presented in base 64 format: the admin user certificate (which will be needed to act as the admin of Org3 later on), a CA root cert, and a TLS root cert. In an upcoming step we will append this JSON file to the channel configuration.

Our final piece of housekeeping is to port the Orderer Org's MSP material into the Org3 `crypto-config` directory. In particular, we are concerned with the Orderer's TLS root cert, which will allow for secure communication between Org3 entities and the network's ordering node.

```
cd ../ && cp -r crypto-config/ordererOrganizations org3-artifacts/crypto-config/
```

Now we're ready to update the channel configuration...

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, 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.

First, exec into the CLI container. Recall that this container has been mounted with the BYFN `crypto-config` library, giving us access to the MSP material for the two original peer organizations and the Orderer Org. The bootstrapped identity is the Org1 admin user, meaning that any steps where we want to act as Org2 will require the export of MSP-specific environment variables.

```
docker exec -it cli bash
```

Now install the `jq` tool into the container. This tool allows script interactions with JSON files returned by the `configtxlator` tool:

```
apt update && apt install -y jq
```

Export the ORDERER_CA and CHANNEL_NAME variables:

```
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  && 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 CLI container, you will also need to re-export the two environment variables – ORDERER_CA and CHANNEL_NAME . The jq installation will persist. You need not install it a second time.

Fetch the Configuration

Now we have a CLI 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).

```
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, there was a decent amount of output in the terminal. The last line in the logs is of interest:

```
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 BYFN script 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

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 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.

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*.

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 ./channel-artifacts/org3.json > modified_config.json
```

Now, within the CLI container we have two JSON files of interest – `config.json` and `modified_config.json` . The initial file contains only Org1 and Org2 material, whereas "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":"mychannel", "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
```

Sign and Submit the Config Update

Almost done!

We now have a protobuf binary – `org3_update_in_envelope.pb` – within our CLI 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 the Org1 Admin. Remember that the CLI container is bootstrapped with the Org1 MSP material, so we simply need to issue the `peer channel signconfigtx` command:

```
peer channel signconfigtx -f org3_update_in_envelope.pb
```

The final step is to switch the CLI 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/
↪crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt

export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp

export CORE_PEER_ADDRESS=peer0.org2.example.com:7051
```

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. From another shell, issue a `docker logs -f orderer.example.com` command 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 digest indication similar to the following if your update has been submitted successfully:

```
2018-02-24 18:56:33.499 UTC [msp/identity] Sign -> DEBU 00f Sign: digest:␣
↪3207B24E40DE2FAB87A2E42BC004FEAA1E6FDCA42977CB78C64F05A88E556ABA
```

You will also see the submission of our configuration transaction:

```
2018-02-24 18:56:33.499 UTC [channelCmd] update -> INFO 010 Successfully submitted␣
↪channel update
```

The successful channel update call returns a new block – block 5 – to all of the peers on the channel. If you remember, blocks 0-2 are the initial channel configurations while blocks 3 and 4 are the instantiation and invocation of the `mycc` chaincode. As such, block 5 serves as the most recent channel configuration with Org3 now defined on the channel.

Inspect the logs for `peer0.org1.example.com`:

```
docker logs -f peer0.org1.example.com
```

Follow the demonstrated process to fetch and decode the new config block if you wish to inspect its contents.

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 in *peer-base.yaml*.

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.

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` .

First, let's launch the containers for the Org3 peers and an Org3-specific CLI.

Open a new terminal and from `first-network` kick off the Org3 docker compose:

```
docker-compose -f docker-compose-org3.yaml up -d
```

This new compose file has been configured to bridge across our initial network, so the two peers and the CLI container will be able to resolve with the existing peers and ordering node. With the three new containers now running, exec into the Org3-specific CLI container:

```
docker exec -it Org3cli bash
```

Just as we did with the initial CLI container, export the two key environment variables: `ORDERER_CA` and `CHANNEL_NAME` :

```
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 && export CHANNEL_NAME=mychannel
```

Check to make sure the variables have been properly set:

```
echo $ORDERER_CA && echo $CHANNEL_NAME
```

Now let's send a call to the ordering service asking for the genesis block of `mychannel` . The ordering service is able to verify the Org3 signature attached to this call as a result of our successful channel update. If Org3 has not been successfully appended to the channel config, the ordering service should 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 (i.e. the genesis block). If we simply passed the `peer channel fetch config` command, then we would have received block 5 – the

updated config with Org3 defined. However, we can't begin our ledger with a downstream block – we must start with block 0.

Issue the `peer channel join` command and pass in the genesis block – `mychannel.block` :

```
peer channel join -b mychannel.block
```

If you want to join the second peer for Org3, export the `TLS` and `ADDRESS` variables and reissue the `peer channel join` command:

```
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪crypto/peerOrganizations/org3.example.com/peers/peer1.org3.example.com/tls/ca.crt &&
↪ export CORE_PEER_ADDRESS=peer1.org3.example.com:7051

peer channel join -b mychannel.block
```

Upgrade and Invoke Chaincode

The final piece of the puzzle is to increment the chaincode version and update the endorsement policy to include Org3. Since we know that an upgrade is coming, we can forgo the futile exercise of installing version 1 of the chaincode. We are solely concerned with the new version where Org3 will be part of the endorsement policy, therefore we'll jump directly to version 2 of the chaincode.

From the Org3 CLI:

```
peer chaincode install -n mycc -v 2.0 -p github.com/chaincode/chaincode_example02/go/
```

Modify the environment variables accordingly and reissue the command if you want to install the chaincode on the second peer of Org3. Note that a second installation is not mandated, as you only need to install chaincode on peers that are going to serve as endorsers or otherwise interface with the ledger (i.e. query only). Peers will still run the validation logic and serve as committers without a running chaincode container.

Now jump back to the **original** CLI container and install the new version on the Org1 and Org2 peers. We submitted the channel update call with the Org2 admin identity, so the container is still acting on behalf of `peer0.org2` :

```
peer chaincode install -n mycc -v 2.0 -p github.com/chaincode/chaincode_example02/go/
```

Flip to the `peer0.org1` identity:

```
export CORE_PEER_LOCALMSPID="Org1MSP"

export 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

export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp

export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
```

And install again:

```
peer chaincode install -n mycc -v 2.0 -p github.com/chaincode/chaincode_example02/go/
```

Now we're ready to upgrade the chaincode. There have been no modifications to the underlying source code, we are simply adding Org3 to the endorsement policy for a chaincode – `mycc` – on `mychannel` .

Note: Any identity satisfying the chaincode's instantiation policy can issue the upgrade call. By default, these identities are the channel Admins.

Send the call:

```
peer chaincode upgrade -o orderer.example.com:7050 --tls $CORE_PEER_TLS_ENABLED --
↪cafile $ORDERER_CA -C $CHANNEL_NAME -n mycc -v 2.0 -c '{"Args":["init","a","90","b",
↪"210"]}' -P "OR ('Org1MSP.peer','Org2MSP.peer','Org3MSP.peer')"
```

You can see in the above command that we are specifying our new version by means of the `v` flag. You can also see that the endorsement policy has been modified to `-P "OR ('Org1MSP.peer','Org2MSP.peer','Org3MSP.peer')"` , reflecting the addition of Org3 to the policy. The final area of interest is our constructor request (specified with the `c` flag).

As with an instantiate call, a chaincode upgrade requires usage of the `init` method. **If** your chaincode requires arguments be passed to the `init` method, then you will need to do so here.

The upgrade call adds a new block – block 6 – to the channel's ledger and allows for the Org3 peers to execute transactions during the endorsement phase. Hop back to the Org3 CLI container and issue a query for the value of `a` . This will take a bit of time because a chaincode image needs to be built for the targeted peer, and the container needs to start:

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

We should see a response of `Query Result: 90` .

Now issue an invocation to move `10` from `a` to `b` :

```
peer chaincode invoke -o orderer.example.com:7050 --tls $CORE_PEER_TLS_ENABLED --
↪cafile $ORDERER_CA -C $CHANNEL_NAME -n mycc -c '{"Args":["invoke","a","b","10"]}'
```

Query one final time:

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

We should see a response of `Query Result: 80` , accurately reflecting the update of this chaincode's world state.

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 ever-growing `peer channel` commands, provide us with the functionality to accomplish this task.

Upgrading Your Network Components

Note: When we use the term "upgrade" in this documentation, we're primarily referring to changing the version of a component (for example, going from a v1.0.x binary to a v1.1 binary). 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.

Overview

Because the *Building Your First Network* (BYFN) tutorial defaults to the "latest" binaries, if you have run it since the release of v1.1, your machine will have v1.1 binaries and tools installed on it and you will not be able to upgrade them.

As a result, this tutorial will provide a network based on Hyperledger Fabric v1.0.6 binaries as well as the v1.1 binaries you will be upgrading to. In addition, we will show how to update channel configurations to recognize *Capability Requirements*.

However, because BYFN does not support the following components, our script for upgrading BYFN will not cover them:

- **Fabric-CA**
- **Kafka**
- **SDK**

The process for upgrading these components will be covered in a section following the tutorial.

At a high level, our upgrade tutorial will perform the following steps:

1. Back up the ledger and MSPs.
2. Upgrade the orderer binaries to Fabric v1.1.
3. Upgrade the peer binaries to Fabric v1.1.
4. Enable v1.1 channel capability requirements.

Note: In production environments, the orderers and peers can be upgraded on a rolling basis simultaneously (in other words, you don't need to upgrade your orderers before upgrading your peers). Where extra care must be taken is in enabling capabilities. All of the orderers and peers must be upgraded before that step (if only some orderers have been upgraded when capabilities have been enabled, a catastrophic state fork can be created).

This tutorial will demonstrate how to perform each of these steps individually with CLI commands.

Prerequisites

If you haven't already done so, ensure you have all of the dependencies on your machine as described in *Prerequisites*.

Launch a v1.0.6 Network

To begin, we will provision a basic network running Fabric v1.0.6 images. This network will consist of two organizations, each maintaining two peer nodes, and a "solo" ordering service.

We will be operating from the `first-network` subdirectory within your local clone of `fabric-samples`. Change into that directory now. You will also want to open a few extra terminals for ease of use.

Clean up

We want to operate from a known state, so we will use the `byfn.sh` script to initially tidy up. This command will kill any active or stale docker containers and remove any previously generated artifacts. Run the following command:

```
./byfn.sh -m down
```

Generate the Crypto and Bring Up the Network

With a clean environment, launch our v1.0.6 BYFN network using these four commands:

```
git fetch origin

git checkout v1.0.6

./byfn.sh -m generate

./byfn.sh -m up -t 3000 -i 1.0.6
```

Note: If you have locally built v1.0.6 images, then they will be used by the example. If you get errors, consider cleaning up v1.0.6 images and running the example again. This will download 1.0.6 images from docker hub.

If BYFN has launched properly, you will see:

```
==================== All GOOD, BYFN execution completed ====================
```

We are now ready to upgrade our network to Hyperledger Fabric v1.1.

Get the newest samples

Note: The instructions below pertain to whatever is the most recently published version of v1.1.x, starting with 1.1.0-rc1. Please substitute '1.1.x' with the version identifier of the published release that you are testing. e.g. replace 'v1.1.x' with 'v1.1.0'.

Before completing the rest of the tutorial, it's important to get the v1.1.x version of the samples, you can do this by:

```
git fetch origin

git checkout v1.1.x
```

Want to upgrade now?

We have a script that will upgrade all of the components in BYFN as well as enabling capabilities. Afterwards, we will walk you through the steps in the script and describe what each piece of code is doing in the upgrade process.

To run the script, issue these commands:

```
# Note, replace '1.1.x' with a specific version, for example '1.1.0'.
# Don't pass the image flag '-i 1.1.x' if you prefer to default to 'latest' images.

./byfn.sh upgrade -i 1.1.x
```

If the upgrade is successful, you should see the following:

```
==================== All GOOD, End-2-End UPGRADE Scenario execution completed␣
↪====================
```

if you want to upgrade the network manually, simply run `./byfn.sh -m down` again and perform the steps up to – but not including – `./byfn.sh upgrade -i 1.1.x`. Then proceed to the next section.

Note: Many of the commands you'll run in this section will not result in any output. In general, assume no output is good output.

Upgrade the Orderer Containers

Note: Pay **CLOSE** attention to your orderer upgrades. If they are not done correctly – specifically, if only some orderers are upgraded and not others – a state fork could be created (meaning, ledgers would no longer be consistent). This MUST be avoided.

Orderer containers should be upgraded in a rolling fashion (one at a time). At a high level, the orderer upgrade process goes as follows:

1. Stop the orderer.
2. Back up the orderer's ledger and MSP.
3. Restart the orderer with the latest images.
4. Verify upgrade completion.

As a consequence of leveraging BYFN, we have a solo orderer setup, therefore, we will only perform this process once. In a Kafka setup, however, this process will have to be performed for each orderer.

Note: This tutorial uses a docker deployment. For native deployments, replace the file `orderer` with the one from the release artifacts. Backup the `orderer.yaml` and replace it with the `orderer.yaml` file from the release artifacts. Then port any modified variables from the backed up `orderer.yaml` to the new one. Utilizing a utility like `diff` may be helpful. To decrease confusion, the variable `General.TLS.ClientAuthEnabled` has been renamed to `General.TLS.ClientAuthRequired` (just as it is specified in the peer configuration.). If the old name for this variable is still present in the `orderer.yaml` file, the new `orderer` binary will fail to start.

Let's begin the upgrade process by **bringing down the orderer**:

```
docker stop orderer.example.com

export LEDGERS_BACKUP=./ledgers-backup

# Note, replace '1.1.x' with a specific version, for example '1.1.0'.
# Set IMAGE_TAG to 'latest' if you prefer to default to the images tagged 'latest' on␣
↪your system.

export IMAGE_TAG=`uname -m`-1.1.x
```

We have created a variable for a directory to put file backups into, and exported the `IMAGE_TAG` we'd like to move to.

Once the orderer is down, you'll want to **backup its ledger and MSP**:

```
mkdir -p $LEDGERS_BACKUP

docker cp orderer.example.com:/var/hyperledger/production/orderer/ ./$LEDGERS_BACKUP/
↪orderer.example.com
```

In a production network this process would be repeated for each of the Kafka-based orderers in a rolling fashion.

Now **download and restart the orderer** with our new fabric image:

```
docker-compose -f docker-compose-cli.yaml up -d --no-deps orderer.example.com
```

Because our sample uses a "solo" ordering service, there are no other orderers in the network that the restarted orderer must sync up to. However, in a production network leveraging Kafka, it will be a best practice to issue `peer channel fetch <blocknumber>` after restarting the orderer to verify that it has caught up to the other orderers.

Upgrade the Peer Containers

Next, let's look at how to upgrade peer containers to Fabric v1.1. Peer containers should, like the orderers, be upgraded in a rolling fashion (one at a time). As mentioned during the orderer upgrade, orderers 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. Restart the peer with with latest image.
5. Verify upgrade completion.

We have four peers running in our network. We will perform this process once for each peer, totaling four upgrades.

Note: Again, this tutorial utilizes a docker deployment. For **native** deployments, replace the file `peer` with the one from the release artifacts. Backup your `core.yaml` and replace it with the one from the release artifacts. Port any modified variables from the backed up `core.yaml` to the new one. Utilizing a utility like `diff` may be helpful.

Let's **bring down the first peer** with the following command:

```
export PEER=peer0.org1.example.com

docker stop $PEER
```

We can then **backup the peer's ledger and MSP**:

```
mkdir -p $LEDGERS_BACKUP

docker cp $PEER:/var/hyperledger/production ./$LEDGERS_BACKUP/$PEER
```

With the peer stopped and the ledger backed up, **remove the peer chaincode containers**:

```
CC_CONTAINERS=$(docker ps | grep dev-$PEER | 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
```

Now we'll re-launch the peer using the v1.1 image tag:

```
docker-compose -f docker-compose-cli.yaml up -d --no-deps $PEER
```

Note: Although, BYFN supports using CouchDB, we opted for a simpler implementation in this tutorial. If you are using CouchDB, however, follow the instructions in the **Upgrading CouchDB** section below at this time and then issue this command instead of the one above:

```
docker-compose -f docker-compose-cli.yaml -f docker-compose-couch.yaml up -d --no-
↪deps $PEER
```

We'll talk more generally about how to update CouchDB after the tutorial.

Verify Upgrade Completion

We've completed the upgrade for our first peer, but before we move on let's check to ensure the upgrade has been completed properly with a chaincode invoke. Let's move `10` from `a` to `b` using these commands:

```
docker-compose -f docker-compose-cli.yaml up -d --no-deps cli

docker exec -it cli bash

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 mychannel -n mycc␣
↪-c '{"Args":["invoke","a","b","10"]}'
```

Our query earlier revealed a to have a value of `90` and we have just removed `10` with our invoke. Therefore, a query against `a` should reveal `80` . Let's see:

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
```

We should see the following:

```
Query Result: 80
```

After verifying the peer was upgraded correctly, make sure to issue an `exit` to leave the container before continuing to upgrade your peers. You can do this by repeating the process above with a different peer name exported.

```
export PEER=peer1.org1.example.com
export PEER=peer0.org2.example.com
export PEER=peer1.org2.example.com
```

Note: All peers must be upgraded BEFORE enabling capabilities.

Enable Capabilities for the Channels

Because v1.0.x Fabric binaries do not understand the concept of channel capabilities, extra care must be taken when initially enabling capabilities for a channel.

Although Fabric binaries can and should be upgraded in a rolling fashion, **it is critical that the ordering admins not attempt to enable v1.1 capabilities until all orderer binaries are at v1.1.x+**. If any orderer is executing v1.0.x code, and capabilities are enabled for a channel, the blockchain will fork as v1.0.x orderers invalidate the change and

v1.1.x+ orderers accept it. This is an exception for the v1.0 to v1.1 upgrade. For future upgrades, such as v1.1 to v1.2, the ordering network will handle the upgrade more gracefully and prevent the state fork.

In order to minimize the chance of a fork, attempts to enable the application or channel v1.1 capabilities before enabling the orderer v1.1 capability will be rejected. Since the orderer v1.1 capability can only be enabled by the ordering admins, making it a prerequisite for the other capabilities prevents application admins from accidentally enabling capabilities before the orderer is ready to support them.

Note: Once a capability has been enabled, disabling it is not recommended or supported.

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.

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.

Note that enabling capability requirements on a channel which a v1.0.0 peer is joined to will result in a crash of the peer. This crashing behavior is deliberate because it indicates a misconfiguration which might result in a state fork.

The error message displayed by failing v1.0.x peers will say:

```
Cannot commit block to the ledger due to Error validating config which passed
initial validity checks: ConfigEnvelope LastUpdate did not produce the supplied
config result
```

We will enable capabilities in the following order:

1. Orderer System Channel

1. Orderer Group

2. Channel Group

2. Individual Channels

1. Orderer Group

2. Channel Group

3. Application Group

Note: In order to minimize the chance of a fork a best practice is to enable the orderer system capability first and then enable individual channel capabilities.

For each group, we will enable the capabilities in the following order:

1. Get the latest channel config

2. Create a modified channel config

3. Create a config update transaction

Note: This process will be accomplished through a series of config update transactions, one for each channel group. In a real world production network, these channel config updates would be handled by the admins for each channel. Because BYFN all exists on a single machine, it is possible for us to update each of these channels.

For more information on updating channel configs, click on *Adding an Org to a Channel* or the doc on *Updating a Channel Configuration*.

Get back into the `cli` container by reissuing `docker exec -it cli bash`.

Now let's check the set environment variables with:

```
env|grep PEER
```

You'll also need to install `jq`:

```
apt-get update

apt-get install -y jq
```

Orderer System Channel Capabilities

Let's set our environment variables for the orderer system channel. Issue each of these commands:

```
CORE_PEER_LOCALMSPID="OrdererMSP"

CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪example.com-cert.pem

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/users/Admin@example.com/msp

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
```

And let's set our channel name to `testchainid`:

```
CH_NAME=testchainid
```

Orderer Group

The first step in updating a channel configuration is getting the latest config block:

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CH_NAME  --
↪tls --cafile $ORDERER_CA
```

Note: We require configtxlator v1.0.0 or higher for this next step.

To make our config easy to edit, let's convert the config block to JSON using configtxlator:

```
configtxlator proto_decode --input config_block.pb --type common.Block --output␣
↪config_block.json
```

This command uses `jq` to remove the headers, metadata, and signatures from the config:

```
jq .data.data[0].payload.data.config config_block.json > config.json
```

Next, add capabilities to the orderer group. The following command will create a copy of the config file and add our new capabilities to it:

```
jq -s '.[0] * {"channel_group":{"groups":{"Orderer": {"values": {"Capabilities": .[1]}
↪}}}}' config.json ./scripts/capabilities.json > modified_config.json
```

Note what we're changing here: `Capabilities` are being added as a `value` of the `orderer` group under `channel_group`. The specific channel we're working in is not noted in this command, but recall that it's the orderer system channel `testchainid`. It should be updated first because it is **this** channel's configuration that will be copied by default during the creation of any new channel.

Now we can create the config update:

```
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
```

Package the config update into a transaction:

```
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:

Note: The command below both signs and submits the transaction to the ordering service.

```
peer channel update -f config_update_in_envelope.pb -c $CH_NAME -o orderer.example.
↪com:7050 --tls true --cafile $ORDERER_CA
```

Our config update transaction represents the difference between the original config and the modified one, but the orderer will translate this into a full channel config.

Channel Group

Now let's move on to enabling capabilities for the channel group at the orderer system level.

The first step, as before, is to get the latest channel configuration.

Note: This set of commands is exactly the same as the steps from the orderer group.

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CH_NAME --
↪tls --cafile $ORDERER_CA
```

```
configtxlator proto_decode --input config_block.pb --type common.Block --output␣
↪config_block.json

jq .data.data[0].payload.data.config config_block.json > config.json
```

Next, create a modified channel config:

```
jq -s '.[0] * {"channel_group":{"values": {"Capabilities": .[1]}}}' config.json ./
↪scripts/capabilities.json > modified_config.json
```

Note what we're changing here: `Capabilities` are being added as a `value` of the top level `channel_group` (in the `testchainid` channel, as before).

Create the config update transaction:

Note: This set of commands is exactly the same as the third step from the orderer group.

```
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
```

Package the config update into a transaction:

```
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.example.
↪com:7050 --tls true --cafile $ORDERER_CA
```

Enabling Capabilities on Existing Channels

Set the channel name to `mychannel` :

```
CH_NAME=mychannel
```

Orderer Group

Get the channel config:

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CH_NAME --
↪tls --cafile $ORDERER_CA

configtxlator proto_decode --input config_block.pb --type common.Block --output␣
↪config_block.json

jq .data.data[0].payload.data.config config_block.json > config.json
```

Let's add capabilities to the orderer group. The following command will create a copy of the config file and add our new capabilities to it:

```
jq -s '.[0] * {"channel_group":{"groups":{"Orderer": {"values": {"Capabilities": .[1]}
↪}}}}' config.json ./scripts/capabilities.json > modified_config.json
```

Note what we're changing here: `Capabilities` are being added as a `value` of the `orderer` group under `channel_group`. This is exactly what we changed before, only now we're working with the config to the channel `mychannel` instead of `testchainid`.

Create the config update:

```
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
```

Package the config update into a transaction:

```
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.example.
↪com:7050 --tls true --cafile $ORDERER_CA
```

Channel Group

Note: While this may seem repetitive, remember that we're performing the same process on different groups. In a production network, as we've said, this process would likely be split up among the various channel admins.

Fetch, decode, and scope the config:

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CH_NAME --
↪tls --cafile $ORDERER_CA
```

```
configtxlator proto_decode --input config_block.pb --type common.Block --output␣
↪config_block.json

jq .data.data[0].payload.data.config config_block.json > config.json
```

Create a modified config:

```
jq -s '.[0] * {"channel_group":{"values": {"Capabilities": .[1]}}}' config.json ./
↪scripts/capabilities.json > modified_config.json
```

Note what we're changing here: `Capabilities` are being added as a `value` of the top level `channel_group` (in `mychannel` , as before).

Create the config update:

```
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
```

Package the config update into a transaction:

```
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
```

Because we're updating the config of the `channel` group, the relevant orgs – Org1, Org2, and the OrdererOrg – need to sign it. This task would usually be performed by the individual org admins, but in BYFN, as we've said, this task falls to us.

First, switch into Org1 and sign the update:

```
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

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

peer channel signconfigtx -f config_update_in_envelope.pb
```

And do the same as Org2:

```
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
```

```
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.org1.example.com:7051

peer channel signconfigtx -f config_update_in_envelope.pb
```

And as the OrdererOrg:

```
CORE_PEER_LOCALMSPID="OrdererMSP"

CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
↪example.com-cert.pem

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
↪ordererOrganizations/example.com/users/Admin@example.com/msp

peer channel update -f config_update_in_envelope.pb -c $CH_NAME -o orderer.example.
↪com:7050 --tls true --cafile $ORDERER_CA
```

Application Group

For the application group, we will need to reset the environment variables as one organization:

```
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

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
```

Now, get the latest channel config (this process should be very familiar by now):

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CH_NAME --
↪tls --cafile $ORDERER_CA

configtxlator proto_decode --input config_block.pb --type common.Block --output␣
↪config_block.json

jq .data.data[0].payload.data.config config_block.json > config.json
```

Create a modified channel config:

```
jq -s '.[0] * {"channel_group":{"groups":{"Application": {"values": {"Capabilities": .
↪[1]}}}}}' config.json ./scripts/capabilities.json > modified_config.json
```

Note what we're changing here: `Capabilities` are being added as a `value` of the `Application` group under `channel_group` (in `mychannel`).

Create a config update transaction:

```
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
```

Package the config update into a transaction:

```
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
```

Org1 signs the transaction:

```
peer channel signconfigtx -f config_update_in_envelope.pb
```

Set the environment variables as Org2:

```
export CORE_PEER_LOCALMSPID="Org2MSP"

export 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

export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
↪crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp

export CORE_PEER_ADDRESS=peer0.org2.example.com:7051
```

Org2 submits the config update transaction with its signature:

```
peer channel update -f config_update_in_envelope.pb -c $CH_NAME -o orderer.example.
↪com:7050 --tls true --cafile $ORDERER_CA
```

Congratulations! You have now enabled capabilities on all of your channels.

Verify that Capabilities are Enabled

But let's test just to make sure by moving `10` from `a` to `b`, as before:

```
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 mychannel -n mycc␣
↪-c '{"Args":["invoke","a","b","10"]}'
```

And then querying the value of `a`, which should reveal a value of `70`. Let's see:

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
```

We should see the following:

```
Query Result: 70
```

In which case we have successfully added capabilities to all of our channels.

Note: Although all peer binaries in the network should have been upgraded prior to this point, enabling capability requirements on a channel which a v1.0.0 peer is joined to will result in a crash of the peer. This crashing behavior is deliberate because it indicates a misconfiguration which might result in a state fork.

Upgrading Components BYFN Does Not Support

Although this is the end of our update tutorial, there are other components that exist in production networks that are not supported by the BYFN sample. In this section, we'll talk through the process of updating them.

Fabric CA Container

To learn how to upgrade your Fabric CA server, click over to the CA documentation.

Upgrade Node SDK Clients

Note: Upgrade Fabric CA before upgrading Node SDK Clients.

Use NPM to upgrade any `Node.js` client by executing these commands in the root directory of your application:

```
npm install fabric-client@1.1

npm install fabric-ca-client@1.1
```

These commands install the new version of both the Fabric client and Fabric-CA client and write the new versions `package.json`.

Upgrading the Kafka Cluster

It is not required, but it is recommended that the Kafka cluster be upgraded and kept up to date along with the rest of Fabric. Newer versions of Kafka support older protocol versions, so you may upgrade Kafka before or after the rest of Fabric.

If your Kafka cluster is older than Kafka v0.11.0, this upgrade is especially recommended as it hardens replication in order to better handle crash faults.

Refer to the official Apache Kafka documentation on upgrading Kafka from previous versions to upgrade the Kafka cluster brokers.

Please note that the Kafka cluster might experience a negative performance impact if the orderer is configured to use a Kafka protocol version that is older than the Kafka broker version. The Kafka protocol version is set using either the `Kafka.Version` key in the `orderer.yaml` file or via the `ORDERER_KAFKA_VERSION` environment variable in a Docker deployment. Fabric v1.0 provided sample Kafka docker images containing Kafka version 0.9.0.1. Fabric v1.1 provides sample Kafka docker images containing Kafka version v1.0.0.

Note: You must configure the Kafka protocol version used by the orderer to match your Kafka cluster version, even if it was not set before. For example, if you are using the sample Kafka images provided with Fabric v1.0.x, either set the `ORDERER_KAFKA_VERSION` environment variable, or the `Kafka.Version` key in the `orderer.yaml` to `0.9.0.1`. If you are unsure about your Kafka cluster version, you can configure the orderer's Kafka protocol version to `0.9.0.1` for maximum compatibility and update the setting afterwards when you have determined your Kafka cluster version.

Upgrading Zookeeper

An Apache Kafka cluster requires an Apache Zookeeper cluster. The Zookeeper API has been stable for a long time and, as such, almost any version of Zookeeper is tolerated by Kafka. Refer to the Apache Kafka upgrade documentation in case there is a specific requirement to upgrade to a specific version of Zookeeper. If you would like to upgrade your Zookeeper cluster, some information on upgrading Zookeeper cluster can be found in the Zookeeper FAQ.

Upgrading CouchDB

If you are using CouchDB as state database, 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. Delete CouchDB data directory.
4. Install CouchDB v2.1.1 binaries or update deployment scripts to use a new Docker image (CouchDB v2.1.1 pre-configured Docker image is provided alongside Fabric v1.1).
5. Restart CouchDB.

The reason to delete the CouchDB data directory is that upon startup the v1.1 peer will rebuild the CouchDB state databases from the blockchain transactions. Starting in v1.1, there will be an internal CouchDB database for each `channel_chaincode` combination (for each chaincode instantiated on each channel that the peer has joined).

Upgrade Chaincodes With Vendored Shim

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 vendoring 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.

If you did not vendor your chaincode, you can skip this step entirely.

Chaincode Tutorials

What is Chaincode?

Chaincode is a program, written in Go, node.js, and eventually in other programming languages such as 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". State created by a chaincode is 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.

Two Personas

We offer two different perspectives on chaincode. One, from the perspective of an application developer developing a blockchain application/solution entitled *Chaincode for Developers*, and the other, *Chaincode for Operators* oriented to the blockchain network operator who is responsible for managing a blockchain network, and who would leverage the Hyperledger Fabric API to install, instantiate, and upgrade chaincode, but would likely not be involved in the development of a chaincode application.

Chaincode for Developers

What is Chaincode?

Chaincode is a program, written in Go, node.js, that implements a prescribed interface. Eventually, other programming languages such as Java, will be supported. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode 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.

Chaincode API

Every chaincode program must implement the `Chaincode interface`:

- Go
- node.js

whose methods are called in response to received transactions. In particular the `Init` method is called when a chaincode receives an `instantiate` or `upgrade` transaction so that the chaincode may perform any necessary initialization, including initialization of application state. The `Invoke` method is called in response to receiving an `invoke` transaction to process transaction proposals.

The other interface in the chaincode "shim" APIs is the `ChaincodeStubInterface`:

- Go
- node.js

which is used to access and modify the ledger, and to make invocations between chaincodes.

In this tutorial, we will demonstrate the use of these APIs by implementing a simple chaincode application that manages simple "assets".

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"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/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()

  // 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())
    }

    // 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"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/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")
    }

    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)
    }
}
```

Building Chaincode

Now let's compile your chaincode.

```
go get -u --tags nopkcs11 github.com/hyperledger/fabric/core/chaincode/shim
go build --tags nopkcs11
```

Assuming there are no errors, now we can proceed to the next step, testing your chaincode.

Testing Using dev mode

Normally chaincodes are started and maintained by peer. However in "dev mode", chaincode is built and started by the user. This mode is useful during chaincode development phase for rapid code/build/run/debug cycle turnaround.

We start "dev mode" by leveraging pre-generated orderer and channel artifacts for a sample dev network. As such, the user can immediately jump into the process of compiling chaincode and driving calls.

Install Hyperledger Fabric Samples

If you haven't already done so, please install the *Hyperledger Fabric Samples*.

Navigate to the `chaincode-docker-devmode` directory of the `fabric-samples` clone:

```
cd chaincode-docker-devmode
```

Download Docker images

We need four Docker images in order for "dev mode" to run against the supplied docker compose script. If you installed the `fabric-samples` repo clone and followed the instructions to *Download Platform-specific Binaries*, then you should have the necessary Docker images installed locally.

Note: If you choose to manually pull the images then you must retag them as `latest` .

Issue a `docker images` command to reveal your local Docker Registry. You should see something similar to following:

```
docker images
REPOSITORY                    TAG                                 IMAGE ID
↪  CREATED              SIZE
hyperledger/fabric-tools      latest                              b7bfddf508bc
↪About an hour ago   1.46GB
hyperledger/fabric-tools      x86_64-1.1.0                        b7bfddf508bc
↪About an hour ago   1.46GB
hyperledger/fabric-orderer    latest                              ce0c810df36a
↪About an hour ago   180MB
hyperledger/fabric-orderer    x86_64-1.1.0                        ce0c810df36a
↪About an hour ago   180MB
hyperledger/fabric-peer       latest                              b023f9be0771
↪About an hour ago   187MB
hyperledger/fabric-peer       x86_64-1.1.0                        b023f9be0771
↪About an hour ago   187MB
hyperledger/fabric-javaenv    latest                              82098abb1a17
↪About an hour ago   1.52GB
```

```
hyperledger/fabric-javaenv      x86_64-1.1.0                    82098abb1a17
 ↪About an hour ago   1.52GB
hyperledger/fabric-ccenv        latest                          c8b4909d8d46
 ↪About an hour ago   1.39GB
hyperledger/fabric-ccenv        x86_64-1.1.0                    c8b4909d8d46
 ↪About an hour ago   1.39GB
```

Note: If you retrieved the images through the *Download Platform-specific Binaries*, then you will see additional images listed. However, we are only concerned with these four.

Now open three terminals and navigate to your `chaincode-docker-devmode` directory in each.

Terminal 1 - Start the network

```
docker-compose -f docker-compose-simple.yaml up
```

The above starts the network with the `SingleSampleMSPSolo` orderer profile and launches the peer in "dev mode". It also launches two additional containers - one for the chaincode environment and a CLI to interact with the chaincode. The commands for create and join channel are embedded in the CLI container, so we can jump immediately to the chaincode calls.

Terminal 2 - Build & start the chaincode

```
docker exec -it chaincode bash
```

You should see the following:

```
root@d2629980e76b:/opt/gopath/src/chaincode#
```

Now, compile your chaincode:

```
cd sacc
go build
```

Now run the chaincode:

```
CORE_PEER_ADDRESS=peer:7052 CORE_CHAINCODE_ID_NAME=mycc:0 ./sacc
```

The chaincode is started with peer and chaincode logs indicating successful registration with the peer. Note that at this stage the chaincode is not associated with any channel. This is done in subsequent steps using the `instantiate` command.

Terminal 3 - Use the chaincode

Even though you are in `--peer-chaincodedev` mode, you still have to install the chaincode so the life-cycle system chaincode can go through its checks normally. This requirement may be removed in future when in `--peer-chaincodedev` mode.

We'll leverage the CLI container to drive these calls.

```
docker exec -it cli bash
```

```
peer chaincode install -p chaincodedev/chaincode/sacc -n mycc -v 0
peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a","10"]}' -C myc
```

Now issue an invoke to change the value of "a" to "20".

```
peer chaincode invoke -n mycc -c '{"Args":["set", "a", "20"]}' -C myc
```

Finally, query `a` . We should see a value of `20` .

```
peer chaincode query -n mycc -c '{"Args":["query","a"]}' -C myc
```

Testing new chaincode

By default, we mount only `sacc` . However, you can easily test different chaincodes by adding them to the `chaincode` subdirectory and relaunching your network. At this point they will be accessible in your `chaincode` container.

Chaincode encryption

In certain scenarios, it may be useful to encrypt values associated with a key in their entirety or simply in part. For example, if a person's social security number or address was being written to the ledger, then you likely would not want this data to appear in plaintext. Chaincode encryption is achieved by leveraging the entities extension which is a BCCSP wrapper with commodity factories and functions to perform cryptographic operations such as encryption and elliptic curve digital signatures. For example, to encrypt, the invoker of a chaincode passes in a cryptographic key via the transient field. The same key may then be used for subsequent query operations, allowing for proper decryption of the encrypted state values.

For more information and samples, see the Encc Example within the `fabric/examples` directory. Pay specific attention to the `utils.go` helper program. This utility loads the chaincode shim APIs and Entities extension and builds a new class of functions (e.g. `encryptAndPutState` & `getStateAndDecrypt`) that the sample encryption chaincode then leverages. As such, the chaincode can now marry the basic shim APIs of `Get` and `Put` with the added functionality of `Encrypt` and `Decrypt` .

Managing external dependencies for chaincode written in Go

If your chaincode requires packages not provided by the Go standard library, you will need to include those packages with your chaincode. 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. `peer chaincode package` and `peer chaincode install` operations will then include code associated with the dependencies into the chaincode package.

Chaincode for Operators

What is Chaincode?

Chaincode is a program, written in Go, and eventually in other programming languages such as 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". State created by a chaincode is 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 the following sections, we will explore chaincode through the eyes of a blockchain network operator, Noah. For Noah's interests, we will focus on chaincode lifecycle operations; the process of packaging, installing, instantiating and upgrading the chaincode as a function of the chaincode's operational lifecycle within a blockchain network.

Chaincode lifecycle

The Hyperledger Fabric API enables interaction with the various nodes in a blockchain network - the peers, orderers and MSPs - and it also allows one to package, install, instantiate and upgrade chaincode on the endorsing peer nodes. The Hyperledger Fabric language-specific SDKs abstract the specifics of the Hyperledger Fabric API to facilitate application development, though it can be used to manage a chaincode's lifecycle. Additionally, the Hyperledger Fabric API can be accessed directly via the CLI, which we will use in this document.

We provide four commands to manage a chaincode's lifecycle: `package` , `install` , `instantiate` , and `upgrade` . In a future release, we are considering adding `stop` and `start` transactions to disable and re-enable a chaincode without having to actually uninstall it. After a chaincode has been successfully installed and instantiated, the chaincode is active (running) and can process transactions via the `invoke` transaction. A chaincode may be upgraded any time after it has been installed.

Packaging

The chaincode package consists of 3 parts:

- the chaincode, as defined by `ChaincodeDeploymentSpec` or CDS. The CDS defines the chaincode package in terms of the code and other properties such as name and version,
- an optional instantiation policy which can be syntactically described by the same policy used for endorsement and described in *Endorsement policies*, and
- a set of signatures by the entities that "own" the chaincode.

The signatures serve the following purposes:

- to establish an ownership of the chaincode,
- to allow verification of the contents of the package, and
- to allow detection of package tampering.

The creator of the instantiation transaction of the chaincode on a channel is validated against the instantiation policy of the chaincode.

Creating the package

There are two approaches to packaging chaincode. One for when you want to have multiple owners of a chaincode, and hence need to have the chaincode package signed by multiple identities. This workflow requires that we initially create a signed chaincode package (a `SignedCDS`) which is subsequently passed serially to each of the other owners for signing.

The simpler workflow is for when you are deploying a SignedCDS that has only the signature of the identity of the node that is issuing the `install` transaction.

We will address the more complex case first. However, you may skip ahead to the *Installing chaincode* section below if you do not need to worry about multiple owners just yet.

To create a signed chaincode package, use the following command:

```
peer chaincode package -n mycc -p github.com/hyperledger/fabric/examples/chaincode/go/
↪chaincode_example02 -v 0 -s -S -i "AND('OrgA.admin')" ccpack.out
```

The `-s` option creates a package that can be signed by multiple owners as opposed to simply creating a raw CDS. When `-s` is specified, the `-S` option must also be specified if other owners are going to need to sign. Otherwise, the process will create a SignedCDS that includes only the instantiation policy in addition to the CDS.

The `-S` option directs the process to sign the package using the MSP identified by the value of the `localMspid` property in `core.yaml` .

The `-S` option is optional. However if a package is created without a signature, it cannot be signed by any other owner using the `signpackage` command.

The optional `-i` option allows one to specify an instantiation policy for the chaincode. The instantiation policy has the same format as an endorsement policy and specifies which identities can instantiate the chaincode. In the example above, only the admin of OrgA is allowed to instantiate the chaincode. If no policy is provided, the default policy is used, which only allows the admin identity of the peer's MSP to instantiate chaincode.

Package signing

A chaincode package that was signed at creation can be handed over to other owners for inspection and signing. The workflow supports out-of-band signing of chaincode package.

The ChaincodeDeploymentSpec may be optionally be signed by the collective owners to create a SignedChaincodeDeploymentSpec (or SignedCDS). The SignedCDS contains 3 elements:

1. The CDS contains the source code, the name, and version of the chaincode.
2. An instantiation policy of the chaincode, expressed as endorsement policies.
3. The list of chaincode owners, defined by means of Endorsement.

Note: Note that this endorsement policy is determined out-of-band to provide proper MSP principals when the chaincode is instantiated on some channels. If the instantiation policy is not specified, the default policy is any MSP administrator of the channel.

Each owner endorses the ChaincodeDeploymentSpec by combining it with that owner's identity (e.g. certificate) and signing the combined result.

A chaincode owner can sign a previously created signed package using the following command:

```
peer chaincode signpackage ccpack.out signedccpack.out
```

Where `ccpack.out` and `signedccpack.out` are the input and output packages, respectively. `signedccpack.out` contains an additional signature over the package signed using the Local MSP.

Installing chaincode

The `install` transaction packages a chaincode's source code into a prescribed format called a `ChaincodeDeploymentSpec` (or CDS) and installs it on a peer node that will run that chaincode.

Note: You must install the chaincode on **each** endorsing peer node of a channel that will run your chaincode.

When the `install` API is given simply a `ChaincodeDeploymentSpec` , it will default the instantiation policy and include an empty owner list.

Note: Chaincode should only be installed on endorsing peer nodes of the owning members of the chaincode to protect the confidentiality of the chaincode logic from other members on the network. Those members without the chaincode, can't be the endorsers of the chaincode's transactions; that is, they can't execute the chaincode. However, they can still validate and commit the transactions to the ledger.

To install a chaincode, send a SignedProposal to the `lifecycle system chaincode` (LSCC) described in the *System Chaincode* section. For example, to install the **sacc** sample chaincode described in section *Simple Asset Chaincode* using the CLI, the command would look like the following:

```
peer chaincode install -n asset_mgmt -v 1.0 -p sacc
```

The CLI internally creates the SignedChaincodeDeploymentSpec for **sacc** and sends it to the local peer, which calls the `Install` method on the LSCC. The argument to the `-p` option specifies the path to the chaincode, which must be located within the source tree of the user's `GOPATH` , e.g. `$GOPATH/src/sacc` . See the *CLI* section for a complete description of the command options.

Note that in order to install on a peer, the signature of the SignedProposal must be from 1 of the peer's local MSP administrators.

Instantiate

The `instantiate` transaction invokes the `lifecycle System Chaincode` (LSCC) to create and initialize a chaincode on a channel. This is a chaincode-channel binding process: a chaincode may be bound to any number of channels and operate on each channel individually and independently. In other words, regardless of how many other channels on which a chaincode might be installed and instantiated, state is kept isolated to the channel to which a transaction is submitted.

The creator of an `instantiate` transaction must satisfy the instantiation policy of the chaincode included in SignedCDS and must also be a writer on the channel, which is configured as part of the channel creation. This is important for the security of the channel to prevent rogue entities from deploying chaincodes or tricking members to execute chaincodes on an unbound channel.

For example, recall that the default instantiation policy is any channel MSP administrator, so the creator of a chaincode instantiate transaction must be a member of the channel administrators. When the transaction proposal arrives at the endorser, it verifies the creator's signature against the instantiation policy. This is done again during the transaction validation before committing it to the ledger.

The instantiate transaction also sets up the endorsement policy for that chaincode on the channel. The endorsement policy describes the attestation requirements for the transaction result to be accepted by members of the channel.

For example, using the CLI to instantiate the **sacc** chaincode and initialize the state with `john` and `0` , the command would look like the following:

```
peer chaincode instantiate -n sacc -v 1.0 -c '{"Args":["john","0"]}' -P "OR ('Org1.
↪member','Org2.member')"
```

Note: Note the endorsement policy (CLI uses polish notation), which requires an endorsement from either member of Org1 or Org2 for all transactions to **sacc**. That is, either Org1 or Org2 must sign the result of executing the *Invoke* on **sacc** for the transactions to be valid.

After being successfully instantiated, the chaincode enters the active state on the channel and is ready to process any transaction proposals of type ENDORSER_TRANSACTION. The transactions are processed concurrently as they arrive at the endorsing peer.

Upgrade

A chaincode may be upgraded any time by changing its version, which is part of the SignedCDS. Other parts, such as owners and instantiation policy are optional. However, the chaincode name must be the same; otherwise it would be considered as a totally different chaincode.

Prior to upgrade, the new version of the chaincode must be installed on the required endorsers. Upgrade is a transaction similar to the instantiate transaction, which binds the new version of the chaincode to the channel. Other channels bound to the old version of the chaincode still run with the old version. In other words, the `upgrade` transaction only affects one channel at a time, the channel to which the transaction is submitted.

Note: Note that since multiple versions of a chaincode may be active simultaneously, the upgrade process doesn't automatically remove the old versions, so user must manage this for the time being.

There's one subtle difference with the `instantiate` transaction: the `upgrade` transaction is checked against the current chaincode instantiation policy, not the new policy (if specified). This is to ensure that only existing members specified in the current instantiation policy may upgrade the chaincode.

Note: Note that during upgrade, the chaincode `Init` function is called to perform any data related updates or re-initialize it, so care must be taken to avoid resetting states when upgrading chaincode.

Stop and Start

Note that `stop` and `start` lifecycle transactions have not yet been implemented. However, you may stop a chaincode manually by removing the chaincode container and the SignedCDS package from each of the endorsers. This is done by deleting the chaincode's container on each of the hosts or virtual machines on which the endorsing peer nodes are running, and then deleting the SignedCDS from each of the endorsing peer nodes:

Note: TODO - in order to delete the CDS from the peer node, you would need to enter the peer node's container, first. We really need to provide a utility script that can do this.

```
docker rm -f <container id>
rm /var/hyperledger/production/chaincodes/<ccname>:<ccversion>
```

Stop would be useful in the workflow for doing upgrade in controlled manner, where a chaincode can be stopped on a channel on all peers before issuing an upgrade.

CLI

Note: We are assessing the need to distribute platform-specific binaries for the Hyperledger Fabric `peer` binary. For the time being, you can simply invoke the commands from within a running docker container.

To view the currently available CLI commands, execute the following command from within a running `fabric-peer` Docker container:

```
docker run -it hyperledger/fabric-peer bash
# peer chaincode --help
```

Which shows output similar to the example below:

```
Usage:
  peer chaincode [command]

Available Commands:
  install     Package the specified chaincode into a deployment spec and save it on␣
↪the peer's path.
  instantiate Deploy the specified chaincode to the network.
  invoke      Invoke the specified chaincode.
  list        Get the instantiated chaincodes on a channel or installed chaincodes on␣
↪a peer.
  package     Package the specified chaincode into a deployment spec.
  query       Query using the specified chaincode.
  signpackage Sign the specified chaincode package
  upgrade     Upgrade chaincode.

Flags:
      --cafile string      Path to file containing PEM-encoded trusted certificate(s)␣
↪for the ordering endpoint
  -h, --help               help for chaincode
  -o, --orderer string     Ordering service endpoint
      --tls                Use TLS when communicating with the orderer endpoint
      --transient string   Transient map of arguments in JSON encoding

Global Flags:
      --logging-level string       Default logging level and overrides, see core.yaml␣
↪for full syntax
      --test.coverprofile string   Done (default "coverage.cov")
  -v, --version

Use "peer chaincode [command] --help" for more information about a command.
```

To facilitate its use in scripted applications, the `peer` command always produces a non-zero return code in the event of command failure.

Example of chaincode commands:

```
peer chaincode install -n mycc -v 0 -p path/to/my/chaincode/v0
peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a", "b", "c"]}' -C mychannel
peer chaincode install -n mycc -v 1 -p path/to/my/chaincode/v1
peer chaincode upgrade -n mycc -v 1 -c '{"Args":["d", "e", "f"]}' -C mychannel
```

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","e"]}'
peer chaincode invoke -o orderer.example.com:7050  --tls --cafile $ORDERER_CA -C␣
↪mychannel -n mycc -c '{"Args":["invoke","a","b","10"]}'
```

System chaincode

System chaincode has the same programming model except that it runs within the peer process rather than in an isolated container like normal chaincode. Therefore, system chaincode is built into the peer executable and doesn't follow the same lifecycle described above. In particular, **install**, **instantiate** and **upgrade** do not apply to system chaincodes.

The purpose of system chaincode is to shortcut gRPC communication cost between peer and chaincode, and tradeoff the flexibility in management. For example, a system chaincode can only be upgraded with the peer binary. It must also register with a fixed set of parameters compiled in and doesn't have endorsement policies or endorsement policy functionality.

System chaincode is used in Hyperledger Fabric to implement a number of system behaviors so that they can be replaced or modified as appropriate by a system integrator.

The current list of system chaincodes:

1. LSCC Lifecycle system chaincode handles lifecycle requests described above.
2. CSCC Configuration system chaincode handles channel configuration on the peer side.
3. QSCC Query system chaincode provides ledger query APIs such as getting blocks and transactions.
4. ESCC Endorsement system chaincode handles endorsement by signing the transaction proposal response.
5. VSCC Validation system chaincode handles the transaction validation, including checking endorsement policy and multiversioning concurrency control.

Care must be taken when modifying or replacing these system chaincodes, especially LSCC, ESCC and VSCC since they are in the main transaction execution path. It is worth noting that as VSCC validates a block before committing it to the ledger, it is important that all peers in the channel compute the same validation to avoid ledger divergence (non-determinism). So special care is needed if VSCC is modified or replaced.

System Chaincode Plugins

System chaincodes are specialized chaincodes that run as part of the peer process as opposed to user chaincodes that run in separate docker containers. As such they have more access to resources in the peer and can be used for implementing features that are difficult or impossible to be implemented through user chaincodes. Examples of System Chaincodes are ESCC (Endorser System Chaincode) for endorsing proposals, QSCC (Query System Chaincode) for ledger and other fabric related queries and VSCC (Validation System Chaincode) for validating a transaction at commit time respectively.

Unlike a user chaincode, a system chaincode is not installed and instantiated using proposals from SDKs or CLI. It is registered and deployed by the peer at start-up.

System chaincodes can be linked to a peer in two ways: statically and dynamically using Go plugins. This tutorial will outline how to develop and load system chaincodes as plugins.

Developing Plugins

A system chaincode is a program written in Go and loaded using the Go plugin package.

A plugin includes a main package with exported symbols and is built with the command `go build -buildmode=plugin`.

Every system chaincode must implement the Chaincode Interface and export a constructor method that matches the signature `func New() shim.Chaincode` in the main package. An example can be found in the repository at `examples/plugin/scc`.

Existing chaincodes such as the QSCC can also serve as templates for certain features - such as access control - that are typically implemented through system chaincodes. The existing system chaincodes also serve as a reference for best-practices on things like logging and testing.

Note: On imported packages: the Go standard library requires that a plugin must include the same version of imported packages as the host application (fabric, in this case)

Configuring Plugins

Plugins are configured in the `chaincode.systemPlugin` section in *core.yaml*:

```
chaincode:
  systemPlugins:
    - enabled: true
      name: mysyscc
      path: /opt/lib/syscc.so
      invokableExternal: true
      invokableCC2CC: true
```

A system chaincode must also be whitelisted in the `chaincode.system` section in *core.yaml*:

```
chaincode:
  system:
    mysyscc: enable
```

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 4

Operations Guides

Upgrading from v1.0.x

At a high level, upgrading a Fabric network to v1.1 can be performed by following these steps:

- Upgrade binaries for orderers, peers, and fabric-ca. These upgrades may be done in parallel.
- Upgrade client SDKs.
- Enable v1.1 channel capability requirements.
- (Optional) Upgrade the Kafka cluster.

To help understand this process, we've created the *Upgrading Your Network Components* tutorial that will take you through most of the major upgrade steps, including upgrading peers, orderers, as well as enabling capability requirements.

Because our tutorial leverages the *Building Your First Network* (BYFN) sample, it has certain limitations (it does not use Fabric CA, for example). Therefore we have included a section at the end of the tutorial that will show how to upgrade your CA, Kafka clusters, CouchDB, Zookeeper, vendored chaincode shims, and Node SDK clients.

If you want to learn more about capability requirements, click here.

Updating a Channel Configuration

What is a Channel Configuration?

Channel configurations contain all of the information relevant to the administration of a channel. Most importantly, the channel configuration specifies which organizations are members of channel, but it also includes other channel-wide configuration information such as channel access policies and block batch sizes.

This configuration is stored on the ledger in a **block**, and is therefore known as a configuration (config) block. Configuration blocks contain a single configuration. The first of these blocks is known as the "genesis block" and contains the initial configuration required to bootstrap a channel. Each time the configuration of a channel changes it is done through a new configuration block, with the latest configuration block representing the current channel configuration. Orderers and peers keep the current channel configuration in memory to facilitate all channel operations such as cutting a new block and validating block transactions.

Because configurations are stored in blocks, updating a config happens through a process called a "configuration transaction" (even though the process is a little different from a normal transaction). Updating a config is a process of pulling the config, translating into a format that humans can read, modifying it and then submitting it for approval.

For a more in-depth look at the process for pulling a config and translating it into JSON, check out Adding an Org to a Channel. In this doc, we'll be focusing on the different ways you can edit a config and the process for getting it signed.

Editing a Config

Channels are highly configurable, but not infinitely so. Different configuration elements have different modification policies (which specify the group of identities required to sign the config update).

To see the scope of what's possible to change it's important to look at a config in JSON format. The Adding an Org to a Channel tutorial generates one, so if you've gone through that doc you can simply refer to it. For those who have not, we'll provide one here (for ease of readability, it might be helpful to put this config into a viewer that supports JSON folding, like atom or Visual Studio).

Click here to see the config

```
{
"channel_group": {
  "groups": {
    "Application": {
      "groups": {
        "Org1MSP": {
          "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
                }
              },
              "version": "0"
            },
            "Readers": {
              "mod_policy": "Admins",
              "policy": {
                "type": 1,
                "value": {
```

```
                "identities": [
                  {
                    "principal": {
                      "msp_identifier": "Org1MSP",
                      "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": "Org1MSP",
                      "role": "MEMBER"
                    },
                    "principal_classification": "ROLE"
                  }
                ],
                "rule": {
                  "n_out_of": {
                    "n": 1,
                    "rules": [
                      {
                        "signed_by": 0
                      }
                    ]
                  }
                },
                "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": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNHRENDQWIrZ0F3SUJBZ0lRSWlyVmg3NVcwWmh0UjEzdmltdmliakFLQ
↪"
                  ],
                  "crypto_config": {
                    "identity_identifier_hash_function": "SHA256",
                    "signature_hash_family": "SHA2"
                  },
                  "name": "Org1MSP",
                  "root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNRekNDQWVxZ0F3SUJBZ0lSQU03ZVdTaVM4V3VVM2haMU9tR255eXd3Q
↪"
                  ],
                  "tls_root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNTVENDQWZDZ0F3SUJBZ0lSQUtsNEFQWmV6dWt0Nk8wYjRyYjY5Y0F3Q
↪"
                  ]
                },
                "type": 0
              },
              "version": "0"
            }
          },
          "version": "1"
        },
        "Org2MSP": {
          "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"
          },
          "Readers": {
            "mod_policy": "Admins",
            "policy": {
              "type": 1,
              "value": {
                "identities": [
                  {
                    "principal": {
                      "msp_identifier": "Org2MSP",
                      "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": "Org2MSP",
                      "role": "MEMBER"
                    },
                    "principal_classification": "ROLE"
                  }
                ],
```

```
                "rule": {
                  "n_out_of": {
                    "n": 1,
                    "rules": [
                      {
                        "signed_by": 0
                      }
                    ]
                  }
                },
                "version": 0
              }
            },
            "version": "0"
          }
        },
        "values": {
          "AnchorPeers": {
            "mod_policy": "Admins",
            "value": {
              "anchor_peers": [
                {
                  "host": "peer0.org2.example.com",
                  "port": 7051
                }
              ]
            },
            "version": "0"
          },
          "MSP": {
            "mod_policy": "Admins",
            "value": {
              "config": {
                "admins": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNHVENDQWNDZ0F3SUJBZ0lSQU5Pb1lIbk9seU94dTJxZFBteStyV293Q
↪"
                ],
                "crypto_config": {
                  "identity_identifier_hash_function": "SHA256",
                  "signature_hash_family": "SHA2"
                },
                "name": "Org2MSP",
                "root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNSRENDQWVxZ0F3SUJBZ0lSQU1pVXk5SGRSbXB5MDdsSjhRMlZNWXN3Q
↪"
                ],
                "tls_root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNTakNDQWZDZ0F3SUJBZ0lSQU9JNmRWUWMraHBZdkdMSlFQM1YwQU13Q
↪"
                ]
              },
              "type": 0
            },
            "version": "0"
          }
```

```
        },
        "version": "1"
      },
      "Org3MSP": {
        "groups": {},
        "mod_policy": "Admins",
        "policies": {
          "Admins": {
            "mod_policy": "Admins",
            "policy": {
              "type": 1,
              "value": {
                "identities": [
                  {
                    "principal": {
                      "msp_identifier": "Org3MSP",
                      "role": "ADMIN"
                    },
                    "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": "Org3MSP",
                      "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": "Org3MSP",
                        "role": "MEMBER"
                      },
                      "principal_classification": "ROLE"
                    }
                  ],
                  "rule": {
                    "n_out_of": {
                      "n": 1,
                      "rules": [
                        {
                          "signed_by": 0
                        }
                      ]
                    }
                  },
                  "version": 0
                }
              },
              "version": "0"
            }
          },
          "values": {
            "MSP": {
              "mod_policy": "Admins",
              "value": {
                "config": {
                  "admins": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNHRENDQWIrZ0F3SUJBZ0lRQUlSNWN4U0hpVm1kSm9uY3FJVUxXekFLQ
↪"
                  ],
                  "crypto_config": {
                    "identity_identifier_hash_function": "SHA256",
                    "signature_hash_family": "SHA2"
                  },
                  "name": "Org3MSP",
                  "root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNRakNDQWVtZ0F3SUJBZ0lRUkN1U2Y0RVJNaDdHQW1ydTFIQ2FZREFLQ
↪"
                  ],
                  "tls_root_certs": [
```

```
↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNTVENDQWZDZ0F3SUJBZ0lSQU9xc2JQQzFOVHJzclEvUUNpalh6K0F3Q
↪"
                  ]
                },
                "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"
        },
        "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"
        }
      },
      "version": "1"
    },
    "Orderer": {
      "groups": {
        "OrdererOrg": {
          "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,
                    "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": {
                      "n": 1,
                      "rules": [
                        {
                          "signed_by": 0
                        }
                      ]
                    }
                  },
                  "version": 0
                }
              },
              "version": "0"
            }
          },
          "values": {
            "MSP": {
              "mod_policy": "Admins",
              "value": {
                "config": {
                  "admins": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNDakNDQWJDZ0F3SUJBZ0lRSFNTTnIyMWRLTTB6THZ0dEdoQnpMVEFLQ
↪"
                  ],
                  "crypto_config": {
                    "identity_identifier_hash_function": "SHA256",
                    "signature_hash_family": "SHA2"
                  },
                  "name": "OrdererMSP",
                  "root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNMakNDQWRXZ0F3SUJBZ0lRY2cxUVZkVmU2Skd6YVU1cmxjcW4vakFLQ
↪"
                  ],
                  "tls_root_certs": [

↪"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNORENDQWR1Z0F3SUJBZ0lRYWJ5SUl6cldtUFNzSjJacisvRVpXVEFLQ
↪"
                  ]
                },
                "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": {
            "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"
        },
        "ChannelRestrictions": {
          "mod_policy": "Admins",
          "version": "0"
        },
        "ConsensusType": {
          "mod_policy": "Admins",
          "value": {
            "type": "solo"
          },
          "version": "0"
        }
      },
      "version": "0"
    }
  },
  "mod_policy": "",
  "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"
    },
    "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",
"type": 0
}
```


A config might look intimidating in this form, but once you study it you'll see that it has a logical structure.

Beyond the definitions of the policies – defining who can do certain things at the channel level, and who has the permission to change who can change the config – channels also have other kinds of features that can be modified using a config update. Adding an Org to a Channel takes you through one of the most important – adding an org to a channel. Some other things that are possible to change with a config update include:

- **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.


```
{
  "absolute_max_bytes": 102760448,
  "max_message_count": 10,
  "preferred_max_bytes": 524288
}
```

- **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.

```
{ "timeout": "2s" }
```

- **Channel Restrictions.** 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.

```
{
 "max_count":1000
}
```

- **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 vzlue is only set in the orderer system channel.

```
{
"type": 3,
"value": {
  "rule": "ANY",
  "sub_policy": "Admins"
  }
}
```

- **Kafka brokers.** 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. *Note that it is not possible to change your consensus type after it has been established (during the bootstrapping of the genesis block).*

```
{
  "brokers": [
    "kafka0:9092",
    "kafka1:9092",
    "kafka2:9092",
    "kafka3:9092"
  ]
}
```

- **Anchor Peers Definition.** Defines the location of the anchor peers for each Org.

```
{
  "host": "peer0.org2.example.com",
    "port": 7051
}
```

- **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.

```
{ "width": 4294967295 }
```

- **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.

```
{ "name": "SHA256" }
```

- **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.

```
{
  "type": 3,
  "value": {
    "rule": "ANY",
    "sub_policy": "Writers"
  }
}
```

- **Orderer Address.** 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.

```
{
  "addresses": [
    "orderer.example.com:7050"
  ]
}
```

Just as we add an Org by adding their artifacts and MSP information, you can remove them by reversing the process.

Note that once the consensus type has been defined and the network has been bootstrapped, it is not possible to change it through a configuration update.

There is another important channel configuration (especially for v1.1) known as **Capability Requirements**. It has its own doc that can be found here.

Let's say you want to edit the block batch size for the channel (because this is a single numeric field, it's one of the easiest changes to make). First to make referencing the JSON path easy, we define it as an environment variable.

To establish this, take a look at your config, find what you're looking for, and back track the path.

If you find batch size, for example, you'll see that it's a `value` of the `Orderer`. `Orderer` can be found under `groups`, which is under `channel_group`. The batch size value has a parameter under `value` of `max_message_count`.

Which would make the path this:

```
export MAXBATCHSIZEPATH=".channel_group.groups.Orderer.values.BatchSize.value.max_
↪message_count"
```

Next, display the value of that property:

```
jq "$MAXBATCHSIZEPATH" config.json
```

Which should return a value of `10` (in our sample network at least).

Now, let's set the new batch size and display the new value:

```
jq "$MAXBATCHSIZEPATH = 20" config.json > modified_config.json
jq "$MAXBATCHSIZEPATH" modified_config.json
```

Once you've modified the JSON, it's ready to be converted and submitted. The scripts and steps in Adding an Org to a Channel will take you through the process for converting the JSON, so let's look at the process of submitting it.

Get the Necessary Signatures

Once you've successfully generated the protobuf file, it's time to get it signed. To do this, you need to know the relevant policy for whatever it is you're trying to change.

By default, editing the configuration of:

- **A particular org** (for example, changing anchor peers) requires only the admin signature of that org.
- **The application** (like who the member orgs are) requires a majority of the application organizations' admins to sign.
- **The orderer** requires a majority of the ordering organizations' admins (of which there are by default only 1).
- **The top level `channel` group** requires both the agreement of a majority of application organization admins and orderer organization admins.

If you have made changes to the default policies in the channel, you'll need to compute your signature requirements accordingly.

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.

Membership Service Providers (MSP)

The document serves to provide details on the setup and best practices for MSPs.

Membership Service Provider (MSP) is a component that aims to offer an abstraction of a membership operation architecture.

In particular, MSP abstracts away all cryptographic mechanisms and protocols behind issuing and validating certificates, and user authentication. An MSP may define their 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.

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 default implementation of MSP, 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) 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 with a verifiable certificate path to exactly one of the certificates of the root of trust to represent the administrators of this MSP; 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 organisations 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 certificate.
- 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.

How to generate MSP certificates and their signing keys?

To generate X.509 certificates to feed its MSP configuration, the application can use Openssl. We emphasise that in Hyperledger Fabric there is no support for certificates including RSA keys.

Alternatively one can use `cryptogen` tool, whose operation is explained in *Getting Started*.

Hyperledger Fabric CA can also be used to generate the keys and certificates needed to configure an MSP.

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).

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 identifiers. Here is an example:

```
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.

Identity Classification

The default MSP implementation allows to further classify identities into clients and peers, based on the OUs of their x509 certificates. An identity should be classified as a **client** if it submits transactions, queries peers, etc. An identity should be classified as a **peer** if it endorses or commits transactions. In order to define clients and peers of a given MSP, the `config.yaml` file needs to be set appropriately. Here is an example:

```
NodeOUs:
  Enable: true
  ClientOUIdentifier:
    Certificate: "cacerts/cacert.pem"
    OrganizationalUnitIdentifier: "client"
  PeerOUIdentifier:
    Certificate: "cacerts/cacert.pem"
    OrganizationalUnitIdentifier: "peer"
```

As shown above, the `NodeOUs.Enable` is set to `true` , this enables the identify classification. Then, client (peer) identifiers are defined by setting the following properties for the `NodeOUs.ClientOUIdentifier` (`NodeOUs.PeerOUIdentifier`) key:

1. `OrganizationalUnitIdentifier` : Set this to the value that matches the OU that the x509 certificate of a client (peer) should contain.
2. `Certificate` : Set this to the CA or intermediate CA under which client (peer) identities should be validated. The field is relative to the MSP root folder. It can be empty, meaning that the identity's x509 certificate can be validated under any CA defined in the MSP configuration.

When the classification is enabled, MSP administrators need to be clients of that MSP, meaning that their x509 certificates need to carry the OU that identifies the clients. Notice also that, an identity can be either a client or a peer. The two classifications are mutually exclusive. If an identity is neither a client nor a peer, the validation will fail.

Finally, notice that for upgraded environments the 1.1 channel capability needs to be enabled before identify classification can be used.

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 emphasise 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.

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 organisations 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 `OU`, and these policies may constitute the read/write policies of a channel or endorsement policies of a chaincode. 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 organisation-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 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.

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.

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 golang notation.

```
// 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;
    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 golang 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.

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)
    Appplication (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.

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,
    },
}
```

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,
                    }
                },
            },
        },
    },
},
```

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.

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.

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.

Endorsement policies

Endorsement policies are used to instruct a peer on how to decide whether a transaction is properly endorsed. When a peer receives a transaction, it invokes the VSCC (Validation System Chaincode) associated with the transaction's Chaincode as part of the transaction validation flow to determine the validity of the transaction. Recall that a transaction contains one or more endorsement from as many endorsing peers. VSCC is tasked to make the following determinations:

- all endorsements are valid (i.e. they are valid signatures from valid certificates over the expected message)
- there is an appropriate number of endorsements
- endorsements come from the expected source(s)

Endorsement policies are a way of specifying the second and third points.

Endorsement policy syntax in the CLI

In the CLI, a simple language is used to express policies in terms of boolean expressions over principals.

A principal is described in terms of the MSP that is tasked to validate the identity of the signer and of the role that the signer has within that MSP. Four roles are supported: **member**, **admin**, **client**, and **peer**. Principals are described as `MSP .ROLE` , where `MSP` is the MSP ID that is required, and `ROLE` is one of the four strings `member` , `admin` , `client` and `peer` . Examples of valid principals are `'Org0.admin'` (any administrator of the `Org0` MSP) or `'Org1.member'` (any member of the `Org1` MSP), `'Org1.client'` (any client of the `Org1` MSP), and `'Org1.peer'` (any peer of the `Org1` MSP).

The syntax of the language is:

```
EXPR(E[,E...])
```

where `EXPR` is either `AND` or `OR` , representing the two boolean expressions 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 1 signature from each of the three principals
- `OR('Org1.member','Org2.member')` requests 1 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 1 signature from a member of the `Org2` MSP and 1 signature from a member of the `Org3` MSP.

Specifying endorsement policies for a chaincode

Using this language, a chaincode deployer can request that the endorsements for a chaincode be validated against the specified policy.

Note: 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')".

The policy can be specified at instantiate time using the `-P` switch, followed by the policy.

For example:

```
peer chaincode instantiate -C <channelid> -n mycc -P "AND('Org1.member', 'Org2.member
↪')"
```

This command deploys chaincode `mycc` with the policy `AND('Org1.member','Org2.member')` which would require that a member of both Org1 and Org2 sign the transaction.

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 chaincode instantiate -C <channelid> -n mycc -P "AND('Org1.peer', 'Org2.peer')"
```

Note: A new organization added to the channel after instantiation can 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 commit a transaction endorsed by it. The endorsement policy needs to be modified to allow transactions to be committed with endorsements from the new organization (see *Upgrade and Invoke Chaincode*).

Error handling

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.

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.

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.

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")
}
```

```
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)
}
```

Logging Control

Overview

Logging in the `peer` application and in the `shim` interface to chaincodes is programmed using facilities provided by the `github.com/op/go-logging` package. This package supports

- Logging control based on the severity of the message
- Logging control based on the software *module* generating the message
- Different pretty-printing options based on the severity of the message

All logs are currently directed to `stderr` , and the pretty-printing is currently fixed. However global and module-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, however when submitting bug reports the 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 4-character code, e.g, "ERRO" for ERROR, "DEBU" for DEBUG, etc. In the logging context a *module* is an arbitrary name (string) given by developers to groups of related messages. In the pretty-printed example below, the logging modules "peer", "rest" and "main" are generating logs.

```
16:47:09.634 [peer] GetLocalAddress -> INFO 033 Auto detected peer address: 9.3.158.
↪178:7051
16:47:09.635 [rest] StartOpenchainRESTServer -> INFO 035 Initializing the REST␣
↪service...
16:47:09.635 [main] serve -> INFO 036 Starting peer with id=name:"vp1" , network␣
↪id=dev, address=9.3.158.178:7051, discovery.rootnode=, validator=true
```

An arbitrary number of logging modules can be created at runtime, therefore there is no "master list" of modules, and logging control constructs can not check whether logging modules actually do or will exist. Also note that the logging module system does not understand hierarchy or wildcarding: You may see module names like "foo/bar" in the code, but the logging system only sees a flat string. It doesn't understand that "foo/bar" is related to "foo" in any way, or that "foo/*" might indicate all "submodules" of foo.

peer

The logging level of the `peer` command can be controlled from the command line for each invocation using the `--logging-level` flag, for example

```
peer node start --logging-level=debug
```

The default logging level for each individual `peer` subcommand can also be set in the core.yaml file. For example the key `logging.node` sets the default level for the `node` subcommmand. Comments in the file also explain how the logging level can be overridden in various ways by using environment varaibles.

Logging severity levels are specified using case-insensitive strings chosen from

```
CRITICAL | ERROR | WARNING | NOTICE | INFO | DEBUG
```

The full logging level specification for the `peer` is of the form

```
[<module>[,<module>...]=]<level>[:[<module>[,<module>...]=]<level>...]
```

A logging level by itself is taken as the overall default. Otherwise, overrides for individual or groups of modules can be specified using the

```
<module>[,<module>...]=<level>
```

syntax. Examples of specifications (valid for all of `--logging-level` , environment variable and core.yaml settings):

```
info                                        - Set default to INFO
warning:main,db=debug:chaincode=info        - Default WARNING; Override for␣
↪main,db,chaincode
chaincode=info:main=debug:db=debug:warning - Same as above
```

Go chaincodes

The standard mechanism to log within a chaincode application is to integrate with the logging transport exposed to each chaincode instance via the peer. The chaincode `shim` package provides APIs that allow a chaincode to create and manage logging objects whose logs will be formatted and interleaved consistently with the `shim` logs.

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.

API

`NewLogger(name string) *ChaincodeLogger` - Create a logging object for use by a chaincode

`(c *ChaincodeLogger) SetLevel(level LoggingLevel)` - Set the logging level of the logger

`(c *ChaincodeLogger) IsEnabledFor(level LoggingLevel) bool` - Return true if logs will be generated at the given level

`LogLevel(levelString string) (LoggingLevel,error)` - Convert a string to a `LoggingLevel`

A `LoggingLevel` is a member of the enumeration

```
LogDebug, LogInfo, LogNotice, LogWarning, LogError, LogCritical
```

which can be used directly, or generated by passing a case-insensitive version of the strings

```
DEBUG, INFO, NOTICE, WARNING, ERROR, CRITICAL
```

to the `LogLevel` API.

Formatted logging at various severity levels is provided by the functions

```
(c *ChaincodeLogger) Debug(args ...interface{})
(c *ChaincodeLogger) Info(args ...interface{})
(c *ChaincodeLogger) Notice(args ...interface{})
(c *ChaincodeLogger) Warning(args ...interface{})
(c *ChaincodeLogger) Error(args ...interface{})
(c *ChaincodeLogger) Critical(args ...interface{})

(c *ChaincodeLogger) Debugf(format string, args ...interface{})
(c *ChaincodeLogger) Infof(format string, args ...interface{})
(c *ChaincodeLogger) Noticef(format string, args ...interface{})
(c *ChaincodeLogger) Warningf(format string, args ...interface{})
(c *ChaincodeLogger) Errorf(format string, args ...interface{})
(c *ChaincodeLogger) Criticalf(format string, args ...interface{})
```

The `f` forms of the logging APIs provide for precise control over the formatting of the logs. The non-`f` forms of the APIs currently insert a space between the printed representations of the arguments, and arbitrarily choose the formats to use.

In the current implementation, the logs produced by the `shim` and a `ChaincodeLogger` are timestamped, marked with the logger *name* and severity level, and written to `stderr`. Note that logging level control is currently based on the *name* provided when the `ChaincodeLogger` is created. To avoid ambiguities, all `ChaincodeLogger` should be given unique names other than "shim". The logger *name* will appear in all log messages created by the logger. The `shim` logs as "shim".

Go language chaincodes can also control the logging level of the chaincode `shim` interface through the `SetLoggingLevel` API.

`SetLoggingLevel(LoggingLevel level)` - Control the logging level of the shim

The default logging level for the shim is `LogDebug`.

Below is a simple example of how a chaincode might create a private logging object logging at the `LogInfo` level, and also control the amount of logging provided by the `shim` based on an environment variable.

```
var logger = shim.NewLogger("myChaincode")

func main() {

    logger.SetLevel(shim.LogInfo)

    logLevel, _ := shim.LogLevel(os.Getenv("SHIM_LOGGING_LEVEL"))
    shim.SetLoggingLevel(logLevel)
    ...
}
```

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.

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.

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

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|instantiate>*, 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>

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 `CORE_LOGGING_GRPC` to `DEBUG`.

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. If your chaincode does not contain a vendored copy of the shim, deleting the chaincode container and restarting its peer will rebuild the chaincode container using the current shim version. If your chaincode vendored a previous version of the shim, review the documentation on how to upgrade-vendored-shim.

Bringing up a Kafka-based Ordering Service

Caveat emptor

This document assumes that the reader generally knows how to set up a Kafka cluster and a ZooKeeper ensemble. The 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.

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.

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 an 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` —or pick a preset profile for the system channel's genesis block— so that:
   (a) `Orderer.OrdererType` is set to `kafka`.
   (b) `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:

   (a) `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.

   (b) `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:

      i. If up to `N-M` replicas —out of the `N` that the channel data is written to— become unavailable, operations proceed normally.

      ii. 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.

   (c) `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.

   (d) `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.)

   (e) `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.GenesisFile` 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.)

   (a) 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.)

   (b) 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:

      i. It creates a Kafka producer (writer) for the Kafka partition that corresponds to the channel.

      ii. It uses that producer to post a no-op `CONNECT` message to that partition.

      iii. 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.**

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 — e.g. `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`.

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.

Debugging

Set `General.LogLevel` to `DEBUG` and `Kafka.Verbose` in `orderer.yaml` to `true`.

Example

Sample Docker Compose configuration files inline with the recommended settings above can be found under the `fabric/bddtests` directory. Look for `dc-orderer-kafka-base.yml` and `dc-orderer-kafka.yml`.

CHAPTER 5

Commands Reference

peer command

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.

Syntax

The `peer` command has five different subcommands within it:

```
peer chaincode [option] [flags]
peer channel   [option] [flags]
peer logging   [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.

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 flags:

- `--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.

- `--logging-level <string>`

  This flag sets the logging level for a peer when it is started.

  There are six possible values for `<string>` : `debug`, `info`, `notice`, `warning`, `error`, and `critical`.

  If `logging-level` is not explicitly specified, then it is taken from the `CORE_LOGGING_LEVEL` environment variable if it is set. If `CORE_LOGGING_LEVEL` is not set then the file `sampleconfig/core.yaml` is used to determined the logging level for the peer.

  You can find the current logging level for a specific component on the peer by running `peer logging getlevel <component-name>`.

- `--version`

  Use this flag to show detailed information about how the peer was built. This flag cannot be applied to `peer` subcommands or their options.

Usage

Here's some examples using the different available flags 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

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
      --keyfile string                      Path to file containing PEM-encoded␣
↪private key to use for mutual TLS communication with the orderer endpoint
      --logging-level string                Default logging level and overrides,␣
↪see core.yaml for full syntax
  -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
  -v, --version                             Display current version of fabric␣
↪peer server
```

This shows brief help syntax for the `peer channel join` command.

- Using the `--version` flag on the `peer` command.

```
peer --version

peer:
 Version: 1.1.0-alpha
 Go version: go1.9.2
 OS/Arch: linux/amd64
 Experimental features: false
 Chaincode:
  Base Image Version: 0.4.5
  Base Docker Namespace: hyperledger
  Base Docker Label: org.hyperledger.fabric
  Docker Namespace: hyperledger
```

This shows that this peer was built using an alpha of Hyperledger Fabric version 1.1.0, compiled with GOLANG 1.9.2. It can be used on Linux operating systems with AMD64 compatible instruction sets.

peer chaincode

Description

The `peer chaincode` subcommand allows administrators to perform chaincode related operations on a peer, such as installing, instantiating, invoking, packaging, querying, and upgrading chaincode.

Syntax

The `peer chaincode` subcommand has the following syntax:

```
peer chaincode install      [flags]
peer chaincode instantiate  [flags]
peer chaincode invoke       [flags]
peer chaincode list         [flags]
peer chaincode package      [flags]
peer chaincode query        [flags]
peer chaincode signpackage  [flags]
peer chaincode upgrade      [flags]
```

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.

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 specifed 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

- `--logging-level <string>`

  Default logging level and overrides, see core.yaml for full syntax

peer chaincode install

Install Description

The `peer chaincode install` command allows administrators to install chaincode onto the filesystem of a peer.

Install Syntax

The `peer chaincode install` command has the following syntax:

```
peer chaincode install [flags]
```

Note: An install can also be performed using a chaincode packaged via the `peer chaincode package` command (see the `peer chaincode package` section below for further details on packaging a chaincode for installation). The syntax using a chaincode package is as follows:

```
peer chaincode install [chaincode-package-file]
```

where `[chaincode-package-file]` is the output file from the `peer chaincode package` command.

Install Flags

The `peer chaincode install` command has the following command-specific flags:

- `-c,--ctor <string>`

  Constructor message for the chaincode in JSON format (default "{}")

- `-l,--lang <string>`

  Language the chaincode is written in (default "golang")

- `-n,--name <string>`

  Name of the chaincode that is being installed. It may consist of alphanumerics, dashes, and underscores

- `-p,--path <string>`

  Path to the chaincode that is being installed. For Golang (-l golang) chaincodes, this is the path relative to the GOPATH. For Node.js (-l node) chaincodes, this is either the absolute path or the relative path from where the install command is being performed

- `-v,--version <string>`

  Version of the chaincode that is being installed. It may consist of alphanumerics, dashes, underscores, periods, and plus signs

Install Usage

Here are some examples of the `peer chaincode install` command:

- To install chaincode named `mycc` at version `1.0` :

```
peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/
↪chaincode/go/chaincode_example02

.
.
.
2018-02-22 16:33:52.998 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003␣
↪Using default escc
2018-02-22 16:33:52.998 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004␣
↪Using default vscc
.
.
.
2018-02-22 16:33:53.194 UTC [chaincodeCmd] install -> DEBU 010 Installed remotely␣
↪response:<status:200 payload:"OK" >
2018-02-22 16:33:53.194 UTC [main] main -> INFO 011 Exiting.....
```

  Here you can see that the install completed successfully based on the log message:

```
2018-02-22 16:33:53.194 UTC [chaincodeCmd] install -> DEBU 010 Installed remotely␣
↪response:<status:200 payload:"OK" >
```

- To install chaincode package `ccpack.out` generated with the `package` subcommand

```
peer chaincode install ccpack.out

.
.
.
2018-02-22 18:18:05.584 UTC [chaincodeCmd] install -> DEBU 005 Installed remotely␣
↪response:<status:200 payload:"OK" >
2018-02-22 18:18:05.584 UTC [main] main -> INFO 006 Exiting.....
```

  Here you can see that the install completed successfully based on the log message:

```
2018-02-22 18:18:05.584 UTC [chaincodeCmd] install -> DEBU 005 Installed remotely␣
↪response:<status:200 payload:"OK" >
```

peer chaincode instantiate

Instantiate Description

The `peer chaincode instantiate` command allows administrators to instantiate chaincode on a channel of which the peer is a member.

Instantiate Syntax

The `peer chaincode instantiate` command has the following syntax:

```
peer chaincode instantiate [flags]
```

Instantiate Flags

The `peer chaincode instantiate` command has the following command-specific flags:

- `-C,--channelID <string>`

  Name of the channel where the chaincode should be instantiated

- `-c,--ctor <string>`

  Constructor message for the chaincode in JSON format (default "{}")

- `-E,--escc <string>`

  Name of the endorsement system chaincode to be used for this chaincode (default "escc")

- `-n,--name <string>`

  Name of the chaincode that is being instantiated

- `-P,--policy <string>`

  Endorsement policy associated to this chaincode. By default fabric will generate an endorsement policy equivalent to "any member from the organizations currently in the channel"

- `-v,--version <string>`

  Version of the chaincode that is being instantiated

- `-V,--vscc <string>`

  Name of the verification system chaincode to be used for this chaincode (default "vscc")

The global `peer` command flags also apply:

- `--cafile <string>`
- `--certfile <string>`
- `--keyfile <string>`
- `-o,--orderer <string>`
- `--ordererTLSHostnameOverride <string>`

- `--tls`
- `--transient <string>`

```
If `--orderer` flag is not specified, the command will attempt to retrieve
the orderer information for the channel from the peer before issuing the
instantiate command.
```

Instantiate Usage

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 "OR (
↪'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 "OR    ('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
2018-02-22 16:34:24.698 UTC [main] main -> INFO 003 Exiting.....
```

peer chaincode invoke

Invoke Description

The `peer chaincode invoke` command allows administrators to call chaincode functions on a peer using the supplied arguments. The CLI invokes chaincode by sending a transaction proposal to a peer. The peer will execute the chaincode and send the endorsed proposal response (or error) to the CLI. On receipt of a endorsed proposal response, the CLI will construct a transaction with it and send it to the orderer.

Invoke Syntax

The `peer chaincode invoke` command has the following syntax:

```
peer chaincode invoke [flags]
```

Invoke Flags

The `peer chaincode invoke` command has the following command-specific flags:

- `-C,--channelID <string>`

  Name of the chaincode that is being invoked

- `-c,--ctor <string>`

  Constructor message for the chaincode in JSON format (default "{}")

- `-n,--name <string>`

  Name of the chaincode that is being invoked

The global `peer` command flags also apply:

- `--cafile <string>`
- `--certfile <string>`
- `--keyfile <string>`
- `-o,--orderer <string>`
- `--ordererTLSHostnameOverride <string>`
- `--tls`
- `--transient <string>`

```
If `--orderer` flag is not specified, the command will attempt to retrieve
the orderer information for the channel from the peer before issuing the
invoke command.
```

Invoke Usage

Here is an example of the `peer chaincode invoke` command, which invokes the chaincode named `mycc` at version `1.0` on channel `mychannel`, requesting to move 10 units from variable `a` to variable `b`:

- ```
  peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n mycc -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\nMCsGA1UdIwQkMCKAIN7dJR9dIMKftkus0R5pAO1R25SA3FB5t8Eaxl9A7lkgMAoG
  ↪Xj3C8lA==\n-----END CERTIFICATE-----\n" signature:"0D\002 \022_
  ↪\342\350\344\231G&\237\n\244\375\302J\2201\302\345\210\335D\250y\253P\0214:
  ↪\221e\332@\002␣
  ```
  ```

```
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

List Description

The `peer chaincode list` command allows administrators to list the chaincodes installed on a peer or the chaincodes instantiated on a channel of which the peer is a member.

List Syntax

The `peer chaincode list` command has the following syntax:

```
peer chaincode list [--installed|--instantiated -C <channel-name>]
```

List Flags

The `peer chaincode instantiate` command has the following command-specific flags:

- `-C,--channelID <string>`

  Name of the channel to list instantiated chaincodes for

- `--installed`

  Use this flag to list the installed chaincodes on a peer

- `--instantiated`

  Use this flag to list the instantiated chaincodes on a channel that the peer is a member of

List Usage

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/examples/chaincode/
↪go/chaincode_example02, 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/examples/chaincode/
↪go/chaincode_example02, 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

Package Description

The `peer chaincode package` command allows administrators to package the materials necessary to perform a chaincode install. This ensures the same chaincode package can be consistently installed on multiple peers.

Package Syntax

The `peer chaincode package` command has the following syntax:

```
peer chaincode package [output-file] [flags]
```

Package Flags

The `peer chaincode package` command has the following command-specific flags:

- `-c,--ctor <string>`

  Constructor message for the chaincode in JSON format (default "{}")

- `-i,--instantiate-policy <string>`

  Instantiation policy for the chaincode. Currently only policies that require utmost 1 signature (e.g., "OR ('Org1MSP.peer','Org2MSP.peer')") are supported.

- `-l,--lang <string>`

  Language the chaincode is written in (default "golang")

- `-n,--name <string>`

  Name of the chaincode that is being installed. It may consist of alphanumerics, dashes, and underscores

- `-p,--path <string>`

Path to the chaincode that is being packaged. For Golang (-l golang) chaincodes, this is the path relative to the GOPATH. For Node.js (-l node) chaincodes, this is either the absolute path or the relative path from where the package command is being performed

- `-s,--cc-package`

  Create a package for storing chaincode ownership information in addition the raw chaincode deployment spec (however, see note below.)

- `-S,--sign`

  Used with the `-s` flag, specify this flag to add owner endorsements to the package using the local MSP (however, see note below.)

- `-v,--version <string>`

  Version of the chaincode that is being installed. It may consist of alphanumerics, dashes, underscores, periods, and plus signs

```
The metadata from `-s` and `-S` commands are not currently used. These commands
are meant for future extensions and will likely undergo implementation changes.
It is recommended that they are not used.
```

Package Usage

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/
  ↪examples/chaincode/go/chaincode_example02 -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

Query Description

The `peer chaincode query` command allows the chaincode to be queried by calling the `Invoke` method on the chaincode. The difference between the `query` and the `invoke` subcommands is that, on successful response, `invoke` proceeds to submit a transaction to the orderer whereas `query` just outputs the response, successful or otherwise, to stdout.
  ```

Query Syntax

The `peer chaincode query` command has the following syntax:

```
peer chaincode query [flags]
```

Query Flags

The `peer chaincode query` command has the following command-specific flags:

- `-C,--channelID <string>`

  Name of the channel where the chaincode should be queried

- `-c,--ctor <string>`

  Constructor message for the chaincode in JSON format (default "{}")

- `-n,--name <string>`

  Name of the chaincode that is being queried

- `-r --raw`

  Output the query value as raw bytes (default)

- `-x --hex`

  Output the query value byte array in hexadecimal. Incompatible with –raw

The global `peer` command flag also applies:

- `--transient <string>`

Query Usage

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` :

- ```
  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
  2018-02-22 16:34:30.816 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002␣
   ↪Using default vscc
  Query Result: 90
  ```

  You can see from the output that variable `a` had a value of 90 at the time of the query.

peer chaincode signpackage

signpackage Description

The `peer chaincode signpackage` command is used to add a signature to a given chaincode package created with the `peer chaincode package` command using `-s` and `-S` options.
  ```

signpackge Syntax

The `peer chaincode signpackage` command has the following syntax:

```
peer chaincode signpackage <inputpackage> <outputpackage>
```

signpackage Usage

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

Upgrade Description

The `peer chaincode upgrade` command allows administrators to upgrade the chaincode instantiated on a channel to a newer version.

Upgrade Syntax

The `peer chaincode upgrade` command has the following syntax:

```
peer chaincode upgrade [flags]
```

Upgrade Flags

The `peer chaincode upgrade` command has the following command-specific flags:

- `-C,--channelID <string>`

  Name of the channel where the chaincode should be upgraded

- `-c,--ctor <string>`

  Constructor message for the chaincode in JSON format (default "{}")

- `-E,--escc <string>`

  Name of the endorsement system chaincode to be used for this chaincode (default "escc")

- `-n,--name <string>`

  Name of the chaincode that is being upgraded

- `-P,--policy <string>`

  Endorsement policy associated to this chaincode. By default fabric will generate an endorsement policy equivalent to "any member from the organizations currently in the channel"

- `-v,--version <string>`

  Version of the upgraded chaincode

- `-V,--vscc <string>`

  Name of the verification system chaincode to be used for this chaincode (default "vscc")

The global `peer` command flags also apply:

- `--cafile <string>`
- `-o,--orderer <string>`
- `--tls`

```
If `--orderer` flag is not specified, the command will attempt to retrieve
the orderer information for the channel from the peer before issuing the
upgrade command.
```

Upgrade Usage

Here is an example of the `peer chaincode upgrade` command, which upgrades the chaincode named `mycc` at version `1.0` on channel `mychannel` to version `1.1`, 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 "OR    ('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\0
 ↪\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 18:26:46.693 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:

```
.
.
.
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\0
↪\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.....
```

peer channel

Description

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.

Syntax

The `peer channel` command has the following syntax:

```
peer channel create       [flags]
peer channel fetch        [flags]
peer channel getinfo      [flags]
peer channel join         [flags]
peer channel list         [flags]
peer channel signconfigtx [flags]
peer channel update       [flags]
```

For brevity, we often refer to a command (`peer`), a subcommand (`channel`), or subcommand option (`fetch`) simply as a **command**.

The different command options (`create` , `fetch` ...) relate to the different channel operations that are relevant to a peer. For example, use the `peer channel join` command to join a peer to a channel, or the `peer channel list` command to show the channels to which a peer is joined.

Each peer channel subcommand is described together with its options in its own section in this topic.

Flags

Each `peer channel` command option has a set of flags specific to it, and these are described with the relevant subcommand option.

All `peer channel` command options also have a set of global flags that can be applied to `peer channel` command options.

The global flags are as follows:

- `--cafile <string>`

  where `<string>` is a fully qualified path to a file containing a PEM-encoded certificate chain of the Certificate Authority of the orderer with whom the peer is communicating. Use in conjunction with the `--tls` flag.

- `--certfile <string>`

  where `<string>` is a fully qualified path to a file containing a PEM-encoded X.509 certificate used for mutual authentication with the orderer. Use in conjunction with the `--clientauth` flag.

- `--clientauth`

  Use this flag to enable mutual TLS communication with the orderer. Use in conjunction with the `--certfile` and `--keyfile` flags.

- `--keyfile <string>`

  where `<string>` is a fully qualified path to a file containing a PEM-encoded X.509 private key used for mutual authentication with the orderer. Use in conjunction with the `--clientauth` flag.

- `-o,--orderer <string>`

  where `<string>` is the fully qualified address and port of the orderer with whom the peer is communicating. If the port is not specified, it will default to port 7050.

- `--ordererTLSHostnameOverride <string>`

  where `<string>` is the hostname override to use when using TLS to communicate with the orderer specified by the `--orderer` flag. It is necessary to use this flag when the TLS handshake phase of communications between the peer and the orderer uses a different hostname than the subsequent message exchange phase. Use in conjunction with the `--tls` flag.

- `--tls`

Use this flag to enable TLS communications with an orderer. The certificates identified by `--cafile` will be used by TLS to authenticate the orderer.

Usage

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.txn`. Use the orderer at `orderer.example.com:7050`.

```
peer channel create -c mychannel -f ./createchannel.txn --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 -> DEBU 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.

peer channel create

Create Description

The `peer channel create` command allows administrators to create a new channel. This command connects to an orderer to perform this function – it is not performed on the peer, even though the `peer` command is used.

To create a channel, the administrator uses the command to submit a configuration update transaction to the orderer. This transaction describes the configuration changes required to create a new channel. Moreover, this transaction must be signed by the required organizations as defined by the current orderer configuration. Configuration transactions can be generated by the `configtxgen` command and signed by the `peer channel signconfigtx` command.

Create Syntax

The `peer channel create` command has the following syntax:

```
peer channel create [flags]
```

Create Flags

The `peer channel create` command has the following command specific flags:

- `-c,--channelID <string>`

  required, where `<string>` is the name of the channel which is to be created.

- `-f,--file <string>`

  required, where `<string>` identifies a file which contains the configuration transaction required to create this channel. It can be generated by `configtxgen` command.

- `-t,--timeout <integer>`

  optional, where `<integer>` specifies channel creation timeout in seconds. If not specified, the default is 5 seconds. Note that if the command times out, then the channel may or may not have been created.

The global `peer` command flags also apply as follows:

- `-o,--orderer` **required**
- `--cafile` **optional**
- `--certfile` **optional**
- `--clientuth` **optional**
- `--keyfile` **optional**
- `--ordererTLSHostnameOverride` **optional**
- `--tls` **optional**

Create Usage

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.txn`. Wait 30 seconds for the channel to be created.

```
peer channel create -c mychannel --orderer orderer.example.com:7050 -f ./
↪createchannel.txn -t 30

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 -> DEBU 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

Fetch Description

The `peer channel fetch` command allows a client to fetch a block from the orderer. The block may contain a configuration transaction or user transactions.

The client must have read access to the channel. This command connects to an orderer to perform this function – it is not performed on the peer, even though the `peer` client command is used.

Fetch Syntax

The `peer channel fetch` command has the following syntax:

```
peer channel fetch [newest|oldest|config|(block number)] [<outputFile>] [flags]
```

where

- `newest`

  returns the most recent block available at the orderer for the channel. This may be a user transaction block or a configuration block.

  This option will also return the block number of the most recent block.

- `oldest`

  returns the oldest block available at the orderer for the channel. This may be a user transaction block or a configuration block.

  This option will also return the block number of the oldest available block.

- `config`

  returns the most recent configuration block available at the orderer for the channel.

  This option will also return the block number of the most recent configuration block.

- `(block number)`

  returns the requested block for the channel. This may be a user transaction block or a configuration block.

  Specifying 0 will result in the genesis block for this channel being returned (if it is still available to the network orderer).

- `<outputFile>`

  specifies the name of the file where the fetched block is written. If `<outputFile>` is not specified, then the block is written to the local directory in a file named as follows:

  - `<channelID>_newest.block`
  - `<channelID>_oldest.block`
  - `<channelID>_config.block`
  - `<channelID>_(block number).block`

Fetch Flags

The `peer channel fetch` command has the following command specific flags:

- `-c,--channelID <string>`

  required, where `<string>` is the name of the channel for which the blocks are to be fetched from the orderer.

The global `peer` command flags also apply:

- `-o,--orderer` **required**
- `--cafile` **optional**
- `--certfile` **optional**
- `--clientuth` **optional**

- `--keyfile` **optional**
- `--ordererTLSHostnameOverride` **optional**
- `--tls` **optional**

Fetch Usage

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 -> DEBU 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 -> DEBU 00a Received block: 16
2018-02-25 13:46:50.302 UTC [main] main -> INFO 00b Exiting.....

ls -l

-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

GetInfo Description

The `peer channel getinfo` command allows administrators to retrieve information about the peer's local blockchain for a particular channel. This includes the current blockchain height, and the hashes of the current block and previous block. Remember that a peer can be joined to more than one channel.

This information can be useful when administrators need to understand the current state of a peer's blockchain, especially in comparison to other peers in the same channel.

GetInfo Syntax

The `peer channel getinfo` command has the following syntax:

```
peer channel getinfo [flags]
```

GetInfo Flags

The `peer channel getinfo` command has no specific flags.

None of the global `peer` command flags apply, since this command does not interact with an orderer.

GetInfo Usage

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 crytographic hashes for the most recent blocks in the channel's blockchain.

peer channel join

Join Description

The `peer channel join` command allows administrators to join a peer to an existing channel. The administrator achieves this by using the command to provide a channel genesis block to the peer. The peer will then automatically retrieve the channel's blocks from other peers in the network, or the orderer, depending on the configuration, and the availability of other peers.

The administrator can create a local genesis block for use by this command by retrieving block 0 from an existing channel using the `peer channel fetch` command option. The `peer channel create` command will also return a local genesis block when a new channel is created.

Join Syntax

The `peer channel join` command has the following syntax:

```
peer channel join [flags]
```

Join Flags

The `peer channel join` command has the following command specific flags:

- `-b,--blockpath <string>`

required, where `<string>` identifies a file containing the channel genesis block. This block can be retrieved using the `peer channel fetch` command, requesting block 0 from the channel, or using the `peer channel create` command.

None of the global `peer` command flags apply, since this command does not interact with an orderer.

Join Usage

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

List Description

The `peer channel list` command allows administrators list the channels to which a peer is joined.

List Syntax

The `peer channel list` command has the following syntax:

```
peer channel list [flags]
```

List Flags

The `peer channel list` command has no specific flags.

None of the global `peer` command flags apply, since this command does not interact with an orderer.

List Usage

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
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

SignConfigTx Description

The `peer channel signconfigtx` command helps administrators sign a configuration transaction with the peer's identity credentials prior to submission to an orderer. Typical configuration transactions include creating a channel or updating a channel configuration.

The administrator supplies an input file to the `signconfigtx` command which describes the configuration transaction. The command then adds the peer's public identity to the file, and signs the entire payload with the peer's private key. The command uses the peer's public and private credentials stored in its local MSP. A new file is not generated; the input file is updated in place.

`signconfigtx` only signs the configuration transaction; it does not create it, nor submit it to the orderer. Typically, the configuration transaction has been already created using the `configtxgen` command, and is subsequently submitted to the orderer by an appropriate command such as `peer channel update` .

SignConfigTx Syntax

The `peer channel signconfigtx` command has the following syntax:

```
peer channel signconfigtx [flags]
```

SignConfigTx Flags

The `peer channel signconfigtx` command has the following command specific flags:

- `-f,--file <string>`

required, where `<string>` identifies a file containing the channel configuration transaction to be signed on behalf of the peer.

None of the global `peer` command flags apply, since this command does not interact with an orderer.

SignConfigTx Usage

Here's an example of the `peer channel signconfigtx` command.

- Sign the `channel update` transaction defined in the file `./updatechannel.txn` . 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

Update Description

The `peer channel update` command allows administrators to update an existing channel.

To update a channel, the administrator uses the command to submit a configuration transaction to the orderer which describes the required channel configuration changes. This transaction must be signed by the required organizations as defined in the current channel configuration. Configuration transactions can be generated by the `configtxgen` command and signed by the `peer channel signconfigtx` command.

The update transaction is sent by the command to the orderer, which validates the change is authorized, and then distributes a configuration block to every peer on the channel. In this way, every peer on the channel maintains a consistent copy of the channel configuration.

Update Syntax

The `peer channel update` command has the following syntax:

```
peer channel update [flags]
```

Update flags

The `peer channel update` command has the following command specific flags:

- `-c,--channelID <string>`

  required, where `<string>` is the name of the channel which is to be updated.

- `-f,--file <string>`

  required, where `<string>` identifies a transaction configuration file. This file contains the configuration change required to this channel, and it can be generated by `configtxgen` command.

The global `peer` command flags also apply as follows:

- `-o,--orderer` **required**
- `--cafile` **optional**

- `--certfile` **optional**
- `--clientuth` **optional**
- `--keyfile` **optional**
- `--ordererTLSHostnameOverride` **optional**
- `--tls` **optional**

Update Usage

Here's an example of the `peer channel update` command.

- Update the channel `mychannel` using the configuration transaction defined in the file `./updatechannel.txn`. 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.txn -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.

peer version

Description

The `peer version` command displays the version information of the peer. It displays version, Go version, OS/architecture, if experimental features are turned on, and chaincode information. For example:

```
peer:
  Version: 1.1.0-beta-snapshot-a6c3447e
  Go version: go1.9.2
  OS/Arch: linux/amd64
  Experimental features: true
  Chaincode:
   Base Image Version: 0.4.5
   Base Docker Namespace: hyperledger
   Base Docker Label: org.hyperledger.fabric
   Docker Namespace: hyperledger
```

Syntax

The `peer version` command has the following syntax:

```
peer version
```

peer logging

Description

The `peer logging` subcommand allows administrators to dynamically view and configure the log levels of a peer.

Syntax

The `peer logging` subcommand has the following syntax:

```
peer logging getlevel
peer logging setlevel
peer logging revertlevels
```

The different subcommand options (getlevel, setlevel, and revertlevels) relate to the different logging operations that are relevant to a peer.

Each peer logging subcommand is described together with its options in its own section in this topic.

peer logging getlevel

Get Level Description

The `peer logging getlevel` command allows administrators to get the current level for a logging module.

Get Level Syntax

The `peer logging getlevel` command has the following syntax:

```
peer logging getlevel <module-name>
```

Get Level Flags

The `peer logging getlevel` command does not have any command-specific flags.

Get Level Usage

Here is an example of the `peer logging getlevel` command:

- To get the log level for module `peer` :

```
peer logging getlevel peer

2018-02-22 19:10:08.633 UTC [cli/logging] getLevel -> INFO 001 Current log level␣
↪for peer module 'peer': DEBUG
2018-02-22 19:10:08.633 UTC [main] main -> INFO 002 Exiting.....
```

peer logging setlevel

Set Level Description

The `peer logging setlevel` command allows administrators to set the current level for all logging modules that match the module name regular expression provided.

Set Level Syntax

The `peer logging setlevel` command has the following syntax:

```
peer logging setlevel <module-name-regular-expression> <log-level>
```

Set Level Flags

The `peer logging setlevel` command does not have any command-specific flags.

Set Level Usage

Here are some examples of the `peer logging setlevel` command:

- To set the log level for modules matching the regular expression `peer` to log level `WARNING` :

```
peer logging setlevel peer warning
2018-02-22 19:14:51.217 UTC [cli/logging] setLevel -> INFO 001 Log level set for␣
↪peer modules matching regular expression 'peer': WARNING
2018-02-22 19:14:51.217 UTC [main] main -> INFO 002 Exiting.....
```

- To set the log level for modules that match the regular expression `^gossip` (i.e. all of the `gossip` logging submodules of the form `gossip/<submodule>`) to log level `ERROR` :

```
peer logging setlevel ^gossip error

2018-02-22 19:16:46.272 UTC [cli/logging] setLevel -> INFO 001 Log level set for␣
↪peer modules matching regular expression '^gossip': ERROR
2018-02-22 19:16:46.272 UTC [main] main -> INFO 002 Exiting.....
```

peer logging revertlevels

Revert Levels Description

The `peer logging revertlevels` command allows administrators to revert the log levels of all modules to their level at the time the peer completed its startup process.

Revert Levels Syntax

The `peer logging revertlevels` command has the following syntax:

```
peer logging revertlevels
```

Revert Levels Flags

The `peer logging revertlevels` command does not have any command-specific flags.

Revert Levels Usage

Here is an example of the `peer logging revertlevels` command:

- ```
  peer logging revertlevels

  2018-02-22 19:18:38.428 UTC [cli/logging] revertLevels -> INFO 001 Log levels␣
  ↪reverted to the levels at the end of peer startup.
  2018-02-22 19:18:38.428 UTC [main] main -> INFO 002 Exiting.....
  ```

peer node

Description

The `peer node` subcommand allows an administrator to start a peer node or check the status of a peer node.

Syntax

The `peer node` subcommand has the following syntax:

```
peer node start [flags]
peer node status
```

peer node start

Start Description

The `peer node start` command allows administrators to start the peer node process.

The peer node process can be configured using configuration file *core.yaml*, which must be located in the directory specified by the environment variable **FABRIC_CFG_PATH**. For docker deployments, *core.yaml* is pre-configured in the peer container **FABRIC_CFG_PATH** directory. For native binary deployments, *core.yaml* is included with the release artifact distribution. The configuration properties located in *core.yaml* can be overridden using environment variables. For example, `peer.mspConfigPath` configuration property can be specified by defining **CORE_PEER_MSPCONFIGPATH** environment variable, where **CORE_** is the prefix for the environment variables.

Start Syntax

The `peer node start` command has the following syntax:

```
peer node start [flags]
```
```

Start Flags

The `peer node start` command has the following command specific flag:

- `--peer-chaincodedev`

  starts peer node in chaincode development mode. Normally chaincode containers are started and maintained by peer. However in devlopment 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.

The global `peer` command flags also apply as described in the `peer command` topic:

- –logging-level

peer node status

Status Description

The `peer node status` command allows administrators to see the status of the peer node process. It will show the status of the peer node process running at the `peer.address` specified in the peer configuration, or overridden by **CORE_PEER_ADDRESS** environment variable.

Status Syntax

The `peer node status` command has the following syntax:

```
peer node status
```

Status Flags

The `peer node status` command has no command specific flags.

configtxgen

Description

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` .

Syntax

The `configtxgen` tool has no sub-commands, but supports flags which can be set to accomplish a number of tasks.

```
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
  -channelID string
        The channel ID to use in the configtx (default "testchainid")
  -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
        Creates an 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)
  -profile string
        The profile from configtx.yaml to use for generation. (default
↪"SampleInsecureSolo")
  -version
        Show version information
```

Usage

Output a genesis block

Write a genesis block to `genesis_block.pb` for channel `orderer-system-channel` for profile `SampleSingleMSPSoloV1_1`.

```
configtxgen -outputBlock genesis_block.pb -profile SampleSingleMSPSoloV1_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
```

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.

This configuration file may be edited, or, individual properties may be overridden by setting environment variables, such as `CONFIGTX_ORDERER_ORDERERTYPE=kafka` .

For many `configtxgen` operations, a profile name must be supplied. Profiles are a way to express multiple similar configurations in a single file. For instance, one profile might define a channel with 3 orgs, and another might define one with 4 orgs. To accomplish this without the length of the file becoming burdensome, `configtx.yaml` depends on the standard YAML feature of anchors and references. Base parts of the configuration are tagged with an anchor like `&OrdererDefaults` and then merged into a profile with a reference like `<<: *OrdererDefaults` . Note, when `configtxgen` is operating under a profile, environment variable overrides do not need to include the profile prefix and may be referenced relative to the root element of the profile. For instance, do not specify `CONFIGTX_PROFILE_SAMPLEINSECURESOLO_ORDERER_ORDERERTYPE` , instead simply omit the profile specifics and use the `CONFIGTX` prefix followed by the elements relative to the profile name such as `CONFIGTX_ORDERER_ORDERERTYPE` .

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.

configtxlator

Description

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.

Syntax

The `configtxlator` tool has four sub-commands.

configtxlator start

Starts the REST server.

```
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
```

configtxlator proto_encode

Converts JSON documents into protobuf messages.

```
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.
```

configtxlator proto_decode

Converts protobuf messages into JSON documents.

```
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.
```

configtxlator compute_update

Computes a config update based on an original, and modified config.

```
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.
```

configtxlator version

Shows the version.

```
usage: configtxlator version

Show version information

Flags:
  --help  Show context-sensitive help (also try --help-long and --help-man).
```

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"
```

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.

Cryptogen Commands

Cryptogen is an utility for generating Hyperledger Fabric key material. It is mainly meant to be used for testing environment.

Syntax

The `cryptogen` command has different subcommands within it:

```
cryptogen [subcommand]
```

as follows

```
cryptogen generate
cryptogen showtemplate
cryptogen version
cryptogen extend
cryptogen help
cryptogen
```

These subcommands separate the different functions provided by they utility.

Within each subcommand there are many different options available and because of this, each is described in its own section in this topic.

If a command option is not specified then `cryptogen` will return some high level help text as described in in the `--help` flag below.

`cryptogen` flags

The `cryptogen` command also has a set of associated flags:

```
cryptogen [flags]
```

as follows

```
cryptogen --help
cryptogen generate --help
```

These flags provide more information about cryptogen, and are designated *global* because they can be used at any command level. For example the `--help` flag can provide help on the `cryptogen` command, the `cryptogen generate` command, as well as their respective options.

Flag details

- `--help`

  Use *help* to get brief help text for the `cryptogen` command. The `help` flag can often be used at different levels to get individual command help, or even a help on a command option. See individual commands for more detail.

Usage

Here's some examples using the different available flags on the *peer* command.

- `--help` flag

```
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
```

The `cryptogen generate` Command

The `cryptogen generate` command allows the generation of the key material.

Syntax

The `cryptogen generate` command has the following syntax:

```
cryptogen generate [<flags>]
```

`cryptogen generate` flags

The `cryptogen generate` command has different flags available to it, and because of this, each flag is described in the relevant command topic.

```
cryptogen generate [flags]
```

as follows

```
cryptogen generate --output="crypto-config"
cryptogen generate --config=CONFIG
```

The global `cryptogen` command flags also apply as described in the *cryptogen command* flags:

- `--help`

Flag details

- `--output="crypto-config"`

  the output directory in which to place artifacts.

- `--config=CONFIG`

  the configuration template to use.

Usage

Here's some examples using the different available flags on the `cryptogen generate` command.

```
./cryptogen generate --output="crypto-config"

org1.example.com
org2.example.com
```

The `cryptogen showtemplate` command

The `cryptogen showtemplate` command shows the default configuration template.

Syntax

The `cryptogen showtemplate` command has the following syntax:

```
cryptogen showtemplate
```

Usage

Output from the `cryptogen showtemplate` command is following:

```
cryptogen showtemplate

# ---------------------------------------------------------------------------
# "OrdererOrgs" - Definition of organizations managing orderer nodes
# ---------------------------------------------------------------------------
OrdererOrgs:
  # ---------------------------------------------------------------------------
  # Orderer
  # ---------------------------------------------------------------------------
  - Name: Orderer
    Domain: example.com

    # ---------------------------------------------------------------------------
    # "Specs" - See PeerOrgs below for complete description
    # ---------------------------------------------------------------------------
    Specs:
      - Hostname: orderer

# ---------------------------------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
# ---------------------------------------------------------------------------
PeerOrgs:
  # ---------------------------------------------------------------------------
  # Org1
  # ---------------------------------------------------------------------------
  - Name: Org1
    Domain: org1.example.com
    EnableNodeOUs: false

    # ---------------------------------------------------------------------------
    # "CA"
    # ---------------------------------------------------------------------------
    # Uncomment this section to enable the explicit definition of the CA for this
    # organization.  This entry is a Spec.  See "Specs" section below for details.
    # ---------------------------------------------------------------------------
    # CA:
    #    Hostname: ca # implicitly ca.org1.example.com
    #    Country: US
    #    Province: California
    #    Locality: San Francisco
    #    OrganizationalUnit: Hyperledger Fabric
    #    StreetAddress: address for org # default nil
    #    PostalCode: postalCode for org # default nil

    # ---------------------------------------------------------------------------
    # "Specs"
    # ---------------------------------------------------------------------------
```

```
    # Uncomment this section to enable the explicit definition of hosts in your
    # configuration.  Most users will want to use Template, below
    #
    # Specs is an array of Spec entries.  Each Spec entry consists of two fields:
    #   - Hostname:   (Required) The desired hostname, sans the domain.
    #   - CommonName: (Optional) Specifies the template or explicit override for
    #                 the CN.  By default, this is the template:
    #
    #                              "{{.Hostname}}.{{.Domain}}"
    #
    #                 which obtains its values from the Spec.Hostname and
    #                 Org.Domain, respectively.
    #   - SANS:       (Optional) Specifies one or more Subject Alternative Names
    #                 to be set in the resulting x509. Accepts template
    #                 variables {{.Hostname}}, {{.Domain}}, {{.CommonName}}. IP
    #                 addresses provided here will be properly recognized. Other
    #                 values will be taken as DNS names.
    #                 NOTE: Two implicit entries are created for you:
    #                     - {{ .CommonName }}
    #                     - {{ .Hostname }}
    # ---------------------------------------------------------------------------
    # Specs:
    #   - Hostname: foo # implicitly "foo.org1.example.com"
    #     CommonName: foo27.org5.example.com # overrides Hostname-based FQDN set above
    #     SANS:
    #       - "bar.{{.Domain}}"
    #       - "altfoo.{{.Domain}}"
    #       - "{{.Hostname}}.org6.net"
    #       - 172.16.10.31
    #   - Hostname: bar
    #   - Hostname: baz

    # ---------------------------------------------------------------------------
    # "Template"
    # ---------------------------------------------------------------------------
    # Allows for the definition of 1 or more hosts that are created sequentially
    # from a template. By default, this looks like "peer%d" from 0 to Count-1.
    # You may override the number of nodes (Count), the starting index (Start)
    # or the template used to construct the name (Hostname).
    #
    # Note: Template and Specs are not mutually exclusive.  You may define both
    # sections and the aggregate nodes will be created for you.  Take care with
    # name collisions
    # ---------------------------------------------------------------------------
    Template:
      Count: 1
      # Start: 5
      # Hostname: {{.Prefix}}{{.Index}} # default
      # SANS:
      #   - "{{.Hostname}}.alt.{{.Domain}}"

    # ---------------------------------------------------------------------------
    # "Users"
    # ---------------------------------------------------------------------------
    # Count: The number of user accounts _in addition_ to Admin
    # ---------------------------------------------------------------------------
    Users:
      Count: 1
```

```
  # ---------------------------------------------------------------------------
  # Org2: See "Org1" for full specification
  # ---------------------------------------------------------------------------
  - Name: Org2
    Domain: org2.example.com
    EnableNodeOUs: false
    Template:
      Count: 1
    Users:
      Count: 1
```

The `cryptogen extend` Command

The `cryptogen extend` command allows to extend an existing network, meaning generating all the additional key material needed by the new added entities.

Syntax

The `cryptogen extend` command has the following syntax:

```
cryptogen extend [<flags>]
```

`cryptogen extend` flags

The `cryptogen extend` command has different flags available to it, and because of this, each flag is described in the relevant command topic.

```
cryptogen extend [flags]
```

as follows

```
cryptogen extend --input="crypto-config"
cryptogen extend --config=CONFIG
```

The global `cryptogen` command flags also apply as described in the *cryptogen command* flags:

- `--help`

Flag details

- `--input="crypto-config"`

  the output directory in which to place artifacts.

- `--config=CONFIG`

  the configuration template to use.

Usage

Here's some examples using the different available flags on the `cryptogen extend` command.

```
cryptogen extend --input="crypto-config" --config=config.yaml

org3.example.com
```

Where config.yaml add a new peer organization called `org3.example.com`

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.

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.

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 6

Architecture Reference

Architecture Explained

The Hyperledger Fabric architecture delivers the following advantages:

- **Chaincode trust flexibility.** The architecture separates *trust assumptions* for chaincodes (blockchain applications) from trust assumptions for ordering. In other words, the ordering service may be provided by one set of nodes (orderers) and tolerate some of them to fail or misbehave, and the endorsers may be different for each chaincode.
- **Scalability.** As the endorser nodes responsible for particular chaincode are orthogonal to the orderers, the system may *scale* better than if these functions were done by the same nodes. In particular, this results when different chaincodes specify disjoint endorsers, which introduces a partitioning of chaincodes between endorsers and allows parallel chaincode execution (endorsement). Besides, chaincode execution, which can potentially be costly, is removed from the critical path of the ordering service.
- **Confidentiality.** The architecture facilitates deployment of chaincodes that have *confidentiality* requirements with respect to the content and state updates of its transactions.
- **Consensus modularity.** The architecture is *modular* and allows pluggable consensus (i.e., ordering service) implementations.

Part I: Elements of the architecture relevant to Hyperledger Fabric v1

1. System architecture
2. Basic workflow of transaction endorsement
3. Endorsement policies

   Part II: Post-v1 elements of the architecture

4. Ledger checkpointing (pruning)

1. System architecture

The blockchain is a distributed system consisting of many nodes that communicate with each other. The blockchain runs programs called chaincode, holds state and ledger data, and executes transactions. The chaincode is the central element as transactions are operations invoked on the chaincode. Transactions have to be "endorsed" and only endorsed transactions may be committed and have an effect on the state. There may exist one or more special chaincodes for management functions and parameters, collectively called *system chaincodes*.

1.1. Transactions

Transactions may be of two types:

- *Deploy transactions* create new chaincode and take a program as parameter. When a deploy transaction executes successfully, the chaincode has been installed "on" the blockchain.
- *Invoke transactions* perform an operation in the context of previously deployed chaincode. An invoke transaction refers to a chaincode and to one of its provided functions. When successful, the chaincode executes the specified function - which may involve modifying the corresponding state, and returning an output.

As described later, deploy transactions are special cases of invoke transactions, where a deploy transaction that creates new chaincode, corresponds to an invoke transaction on a system chaincode.

Remark: *This document currently assumes that a transaction either creates new chaincode or invokes an operation provided by *one* already deployed chaincode. This document does not yet describe: a) optimizations for query (read-only) transactions (included in v1), b) support for cross-chaincode transactions (post-v1 feature).*

1.2. Blockchain datastructures

1.2.1. State

The latest state of the blockchain (or, simply, *state*) is modeled as a versioned key-value store (KVS), where keys are names and values are arbitrary blobs. These entries are manipulated by the chaincodes (applications) running on the blockchain through `put` and `get` KVS-operations. The state is stored persistently and updates to the state are logged. Notice that versioned KVS is adopted as state model, an implementation may use actual KVSs, but also RDBMSs or any other solution.

More formally, state `s` is modeled as an element of a mapping `K -> (V X N)`, where:

- `K` is a set of keys
- `V` is a set of values
- `N` is an infinite ordered set of version numbers. Injective function `next: N -> N` takes an element of `N` and returns the next version number.

Both `V` and `N` contain a special element (empty type), which is in case of `N` the lowest element. Initially all keys are mapped to (,). For `s(k)=(v,ver)` we denote `v` by `s(k).value`, and `ver` by `s(k).version`.

KVS operations are modeled as follows:

- `put(k,v)` for `k K` and `v V`, takes the blockchain state `s` and changes it to `s'` such that `s'(k)=(v,next(s(k).version))` with `s'(k')=s(k')` for all `k'!=k`.
- `get(k)` returns `s(k)`.

State is maintained by peers, but not by orderers and clients.

State partitioning. Keys in the KVS can be recognized from their name to belong to a particular chaincode, in the sense that only transaction of a certain chaincode may modify the keys belonging to this chaincode. In principle, any chaincode can read the keys belonging to other chaincodes. *Support for cross-chaincode transactions, that modify the state belonging to two or more chaincodes is a post-v1 feature.*

1.2.2 Ledger

Ledger provides a verifiable history of all successful state changes (we talk about *valid* transactions) and unsuccessful attempts to change state (we talk about *invalid* transactions), occurring during the operation of the system.

Ledger is constructed by the ordering service (see Sec 1.3.3) as a totally ordered hashchain of *blocks* of (valid or invalid) transactions. The hashchain imposes the total order of blocks in a ledger and each block contains an array of totally ordered transactions. This imposes total order across all transactions.

Ledger is kept at all peers and, optionally, at a subset of orderers. In the context of an orderer we refer to the Ledger as to `OrdererLedger` , whereas in the context of a peer we refer to the ledger as to `PeerLedger` . `PeerLedger` differs from the `OrdererLedger` in that peers locally maintain a bitmask that tells apart valid transactions from invalid ones (see Section XX for more details).

Peers may prune `PeerLedger` as described in Section XX (post-v1 feature). Orderers maintain `OrdererLedger` for fault-tolerance and availability (of the `PeerLedger`) and may decide to prune it at anytime, provided that properties of the ordering service (see Sec. 1.3.3) are maintained.

The ledger allows peers to replay the history of all transactions and to reconstruct the state. Therefore, state as described in Sec 1.2.1 is an optional datastructure.

1.3. Nodes

Nodes are the communication entities of the blockchain. A "node" is only a logical function in the sense that multiple nodes of different types can run on the same physical server. What counts is how nodes are grouped in "trust domains" and associated to logical entities that control them.

There are three types of nodes:

1. **Client** or **submitting-client**: a client that submits an actual transaction-invocation to the endorsers, and broadcasts transaction-proposals to the ordering service.
2. **Peer**: a node that commits transactions and maintains the state and a copy of the ledger (see Sec, 1.2). Besides, peers can have a special **endorser** role.
3. **Ordering-service-node** or **orderer**: a node running the communication service that implements a delivery guarantee, such as atomic or total order broadcast.

The types of nodes are explained next in more detail.

1.3.1. Client

The client represents the entity that acts on behalf of an end-user. It must connect to a peer for communicating with the blockchain. The client may connect to any peer of its choice. Clients create and thereby invoke transactions.

As detailed in Section 2, clients communicate with both peers and the ordering service.

1.3.2. Peer

A peer receives ordered state updates in the form of *blocks* from the ordering service and maintain the state and the ledger.

Peers can additionally take up a special role of an **endorsing peer**, or an **endorser**. The special function of an *endorsing peer* occurs with respect to a particular chaincode and consists in *endorsing* a transaction before it is committed. Every chaincode may specify an *endorsement policy* that may refer to a set of endorsing peers. The policy defines the necessary and sufficient conditions for a valid transaction endorsement (typically a set of endorsers' signatures), as described later in Sections 2 and 3. In the special case of deploy transactions that install new chaincode the (deployment) endorsement policy is specified as an endorsement policy of the system chaincode.

1.3.3. Ordering service nodes (Orderers)

The *orderers* form the *ordering service*, i.e., a communication fabric that provides delivery guarantees. The ordering service can be implemented in different ways: ranging from a centralized service (used e.g., in development and testing) to distributed protocols that target different network and node fault models.

Ordering service provides a shared *communication channel* to clients and peers, offering a broadcast service for messages containing transactions. Clients connect to the channel and may broadcast messages on the channel which are then delivered to all peers. The channel supports *atomic* delivery of all messages, that is, message communication with total-order delivery and (implementation specific) reliability. In other words, the channel outputs the same messages to all connected peers and outputs them to all peers in the same logical order. This atomic communication guarantee is also called *total-order broadcast*, *atomic broadcast*, or *consensus* in the context of distributed systems. The communicated messages are the candidate transactions for inclusion in the blockchain state.

Partitioning (ordering service channels). Ordering service may support multiple *channels* similar to the *topics* of a publish/subscribe (pub/sub) messaging system. Clients can connect to a given channel and can then send messages and obtain the messages that arrive. Channels can be thought of as partitions - clients connecting to one channel are unaware of the existence of other channels, but clients may connect to multiple channels. Even though some ordering service implementations included with Hyperledger Fabric support multiple channels, for simplicity of presentation, in the rest of this document, we assume ordering service consists of a single channel/topic.

Ordering service API. Peers connect to the channel provided by the ordering service, via the interface provided by the ordering service. The ordering service API consists of two basic operations (more generally *asynchronous events*):

TODO add the part of the API for fetching particular blocks under client/peer specified sequence numbers.

- `broadcast(blob)` : a client calls this to broadcast an arbitrary message `blob` for dissemination over the channel. This is also called `request(blob)` in the BFT context, when sending a request to a service.
- `deliver(seqno,prevhash,blob)` : the ordering service calls this on the peer to deliver the message `blob` with the specified non-negative integer sequence number (`seqno`) and hash of the most recently delivered blob (`prevhash`). In other words, it is an output event from the ordering service. `deliver()` is also sometimes called `notify()` in pub-sub systems or `commit()` in BFT systems.

Ledger and block formation. The ledger (see also Sec. 1.2.2) contains all data output by the ordering service. In a nutshell, it is a sequence of `deliver(seqno,prevhash,blob)` events, which form a hash chain according to the computation of `prevhash` described before.

Most of the time, for efficiency reasons, instead of outputting individual transactions (blobs), the ordering service will group (batch) the blobs and output *blocks* within a single `deliver` event. In this case, the ordering service must impose and convey a deterministic ordering of the blobs within each block. The number of blobs in a block may be chosen dynamically by an ordering service implementation.

In the following, for ease of presentation, we define ordering service properties (rest of this subsection) and explain the workflow of transaction endorsement (Section 2) assuming one blob per `deliver` event. These are easily extended to blocks, assuming that a `deliver` event for a block corresponds to a sequence of individual `deliver` events for each blob within a block, according to the above mentioned deterministic ordering of blobs within a blocs.

Ordering service properties

The guarantees of the ordering service (or atomic-broadcast channel) stipulate what happens to a broadcasted message and what relations exist among delivered messages. These guarantees are as follows:

1. **Safety (consistency guarantees)**: As long as peers are connected for sufficiently long periods of time to the channel (they can disconnect or crash, but will restart and reconnect), they will see an *identical* series of delivered `(seqno,prevhash,blob)` messages. This means the outputs (`deliver()` events) occur in the *same order* on all peers and according to sequence number and carry *identical content* (`blob` and `prevhash`) for the same sequence number. Note this is only a *logical order*, and a `deliver(seqno,prevhash,blob)` on one peer is not required to occur in any real-time relation to `deliver(seqno,prevhash,blob)` that outputs the same message at another peer. Put differently, given a particular `seqno` , *no* two correct peers

deliver *different* `prevhash` or `blob` values. Moreover, no value `blob` is delivered unless some client (peer) actually called `broadcast(blob)` and, preferably, every broadcasted blob is only delivered *once*.

Furthermore, the `deliver()` event contains the cryptographic hash of the data in the previous `deliver()` event (`prevhash`). When the ordering service implements atomic broadcast guarantees, `prevhash` is the cryptographic hash of the parameters from the `deliver()` event with sequence number `seqno-1` . This establishes a hash chain across `deliver()` events, which is used to help verify the integrity of the ordering service output, as discussed in Sections 4 and 5 later. In the special case of the first `deliver()` event, `prevhash` has a default value.

2. **Liveness (delivery guarantee)**: Liveness guarantees of the ordering service are specified by a ordering service implementation. The exact guarantees may depend on the network and node fault model.

   In principle, if the submitting client does not fail, the ordering service should guarantee that every correct peer that connects to the ordering service eventually delivers every submitted transaction.

To summarize, the ordering service ensures the following properties:

- *Agreement.* For any two events at correct peers `deliver(seqno,prevhash0,blob0)` and `deliver(seqno,prevhash1,blob1)` with the same `seqno` , `prevhash0==prevhash1` and `blob0==blob1` ;
- *Hashchain integrity.* For any two events at correct peers `deliver(seqno-1,prevhash0,blob0)` and `deliver(seqno,prevhash,blob)` ,`prevhash = HASH(seqno-1||prevhash0||blob0)` .
- *No skipping*. If an ordering service outputs `deliver(seqno,prevhash,blob)` at a correct peer *p*, such that `seqno>0` , then *p* already delivered an event `deliver(seqno-1,prevhash0,blob0)` .
- *No creation*. Any event `deliver(seqno,prevhash,blob)` at a correct peer must be preceded by a `broadcast(blob)` event at some (possibly distinct) peer;
- *No duplication (optional, yet desirable).* For any two events `broadcast(blob)` and `broadcast(blob')` , when two events `deliver(seqno0,prevhash0,blob)` and `deliver(seqno1,prevhash1,blob')` occur at correct peers and `blob == blob'` , then `seqno0==seqno1` and `prevhash0==prevhash1` .
- *Liveness*. If a correct client invokes an event `broadcast(blob)` then every correct peer "eventually" issues an event `deliver(*,*,blob)` , where `*` denotes an arbitrary value.

2. Basic workflow of transaction endorsement

In the following we outline the high-level request flow for a transaction.

Remark: *Notice that the following protocol *does not* assume that all transactions are deterministic, i.e., it allows for non-deterministic transactions.**

2.1. The client creates a transaction and sends it to endorsing peers of its choice

To invoke a transaction, the client sends a `PROPOSE` message to a set of endorsing peers of its choice (possibly not at the same time - see Sections 2.1.2. and 2.3.). The set of endorsing peers for a given `chaincodeID` is made available to client via peer, which in turn knows the set of endorsing peers from endorsement policy (see Section 3). For example, the transaction could be sent to *all* endorsers of a given `chaincodeID` . That said, some endorsers could be offline, others may object and choose not to endorse the transaction. The submitting client tries to satisfy the policy expression with the endorsers available.

In the following, we first detail `PROPOSE` message format and then discuss possible patterns of interaction between submitting client and endorsers.

2.1.1. PROPOSE message format

The format of a PROPOSE message is <PROPOSE,tx,[anchor]> , where tx is a mandatory and anchor optional argument explained in the following.

- tx=<clientID,chaincodeID,txPayload,timestamp,clientSig> , where
  - clientID is an ID of the submitting client,
  - chaincodeID refers to the chaincode to which the transaction pertains,
  - txPayload is the payload containing the submitted transaction itself,
  - timestamp is a monotonically increasing (for every new transaction) integer maintained by the client,
  - clientSig is signature of a client on other fields of tx .

  The details of txPayload will differ between invoke transactions and deploy transactions (i.e., invoke transactions referring to a deploy-specific system chaincode). For an **invoke transaction**, txPayload would consist of two fields
  - txPayload = <operation,metadata> , where
    - operation denotes the chaincode operation (function) and arguments,
    - metadata denotes attributes related to the invocation.

  For a **deploy transaction**, txPayload would consist of three fields
  - txPayload = <source,metadata,policies> , where
    - source denotes the source code of the chaincode,
    - metadata denotes attributes related to the chaincode and application,
    - policies contains policies related to the chaincode that are accessible to all peers, such as the endorsement policy. Note that endorsement policies are not supplied with txPayload in a deploy transaction, but txPayload of a deploy contains endorsement policy ID and its parameters (see Section 3).
- anchor contains *read version dependencies*, or more specifically, key-version pairs (i.e., anchor is a subset of KxN), that binds or "anchors" the PROPOSE request to specified versions of keys in a KVS (see Section 1.2.). If the client specifies the anchor argument, an endorser endorses a transaction only upon *read* version numbers of corresponding keys in its local KVS match anchor (see Section 2.2. for more details).

Cryptographic hash of tx is used by all nodes as a unique transaction identifier tid (i.e., tid=HASH(tx)). The client stores tid in memory and waits for responses from endorsing peers.

2.1.2. Message patterns

The client decides on the sequence of interaction with endorsers. For example, a client would typically send <PROPOSE,tx> (i.e., without the anchor argument) to a single endorser, which would then produce the version dependencies (anchor) which the client can later on use as an argument of its PROPOSE message to other endorsers. As another example, the client could directly send <PROPOSE,tx> (without anchor) to all endorsers of its choice. Different patterns of communication are possible and client is free to decide on those (see also Section 2.3.).

2.2. The endorsing peer simulates a transaction and produces an endorsement signature

On reception of a <PROPOSE,tx,[anchor]> message from a client, the endorsing peer epID first verifies the client's signature clientSig and then simulates a transaction. If the client specifies anchor then endorsing peer

simulates the transactions only upon read version numbers (i.e., `readset` as defined below) of corresponding keys in its local KVS match those version numbers specified by `anchor`.

Simulating a transaction involves endorsing peer tentatively *executing* a transaction (`txPayload`), by invoking the chaincode to which the transaction refers (`chaincodeID`) and the copy of the state that the endorsing peer locally holds.

As a result of the execution, the endorsing peer computes *read version dependencies* (`readset`) and *state updates* (`writeset`), also called *MVCC+postimage info* in DB language.

Recall that the state consists of key-value pairs. All key-value entries are versioned; that is, every entry contains ordered version information, which is incremented each time the value stored under a key is updated. The peer that interprets the transaction records all key-value pairs accessed by the chaincode, either for reading or for writing, but the peer does not yet update its state. More specifically:

- Given state `s` before an endorsing peer executes a transaction, for every key `k` read by the transaction, pair `(k,s(k).version)` is added to `readset`.
- Additionally, for every key `k` modified by the transaction to the new value `v'`, pair `(k,v')` is added to `writeset`. Alternatively, `v'` could be the delta of the new value to previous value (`s(k).value`).

If a client specifies `anchor` in the `PROPOSE` message then client specified `anchor` must equal `readset` produced by endorsing peer when simulating the transaction.

Then, the peer forwards internally `tran-proposal` (and possibly `tx`) to the part of its (peer's) logic that endorses a transaction, referred to as **endorsing logic**. By default, endorsing logic at a peer accepts the `tran-proposal` and simply signs the `tran-proposal`. However, endorsing logic may interpret arbitrary functionality, to, e.g., interact with legacy systems with `tran-proposal` and `tx` as inputs to reach the decision whether to endorse a transaction or not.

If endorsing logic decides to endorse a transaction, it sends `<TRANSACTION-ENDORSED,tid,tran-proposal,epSig>` message to the submitting client(`tx.clientID`), where:

- `tran-proposal := (epID,tid,chaincodeID,txContentBlob,readset,writeset)`,

  where `txContentBlob` is chaincode/transaction specific information. The intention is to have `txContentBlob` used as some representation of `tx` (e.g., `txContentBlob=tx.txPayload`).

- `epSig` is the endorsing peer's signature on `tran-proposal`

Else, in case the endorsing logic refuses to endorse the transaction, an endorser *may* send a message `(TRANSACTION-INVALID,tid,REJECTED)` to the submitting client.

Notice that an endorser does not change its state in this step, the updates produced by transaction simulation in the context of endorsement do not affect the state!

2.3. The submitting client collects an endorsement for a transaction and broadcasts it through ordering service

The submitting client waits until it receives "enough" messages and signatures on `(TRANSACTION-ENDORSED,tid,*,*)` statements to conclude that the transaction proposal is endorsed. As discussed in Section 2.1.2., this may involve one or more round-trips of interaction with endorsers.

The exact number of "enough" depend on the chaincode endorsement policy (see also Section 3). If the endorsement policy is satisfied, the transaction has been *endorsed*; note that it is not yet committed. The collection of signed `TRANSACTION-ENDORSED` messages from endorsing peers which establish that a transaction is endorsed is called an *endorsement* and denoted by `endorsement`.

If the submitting client does not manage to collect an endorsement for a transaction proposal, it abandons this transaction with an option to retry later.

For transaction with a valid endorsement, we now start using the ordering service. The submitting client invokes ordering service using the `broadcast(blob)`, where `blob=endorsement`. If the client does not have capability of invoking ordering service directly, it may proxy its broadcast through some peer of its choice. Such a peer must be trusted by the client not to remove any message from the `endorsement` or otherwise the transaction may be deemed invalid. Notice that, however, a proxy peer may not fabricate a valid `endorsement`.

2.4. The ordering service delivers a transactions to the peers

When an event `deliver(seqno,prevhash,blob)` occurs and a peer has applied all state updates for blobs with sequence number lower than `seqno`, a peer does the following:

- It checks that the `blob.endorsement` is valid according to the policy of the chaincode (`blob.tran-proposal.chaincodeID`) to which it refers.
- In a typical case, it also verifies that the dependencies (`blob.endorsement.tran-proposal.readset`) have not been violated meanwhile. In more complex use cases, `tran-proposal` fields in endorsement may differ and in this case endorsement policy (Section 3) specifies how the state evolves.

Verification of dependencies can be implemented in different ways, according to a consistency property or "isolation guarantee" that is chosen for the state updates. **Serializability** is a default isolation guarantee, unless chaincode endorsement policy specifies a different one. Serializability can be provided by requiring the version associated with *every* key in the `readset` to be equal to that key's version in the state, and rejecting transactions that do not satisfy this requirement.

- If all these checks pass, the transaction is deemed *valid* or *committed*. In this case, the peer marks the transaction with 1 in the bitmask of the `PeerLedger`, applies `blob.endorsement.tran-proposal.writeset` to blockchain state (if `tran-proposals` are the same, otherwise endorsement policy logic defines the function that takes `blob.endorsement`).
- If the endorsement policy verification of `blob.endorsement` fails, the transaction is invalid and the peer marks the transaction with 0 in the bitmask of the `PeerLedger`. It is important to note that invalid transactions do not change the state.

Note that this is sufficient to have all (correct) peers have the same state after processing a deliver event (block) with a given sequence number. Namely, by the guarantees of the ordering service, all correct peers will receive an identical sequence of `deliver(seqno,prevhash,blob)` events. As the evaluation of the endorsement policy and evaluation of version dependencies in `readset` are deterministic, all correct peers will also come to the same conclusion whether a transaction contained in a blob is valid. Hence, all peers commit and apply the same sequence of transactions and update their state in the same way.

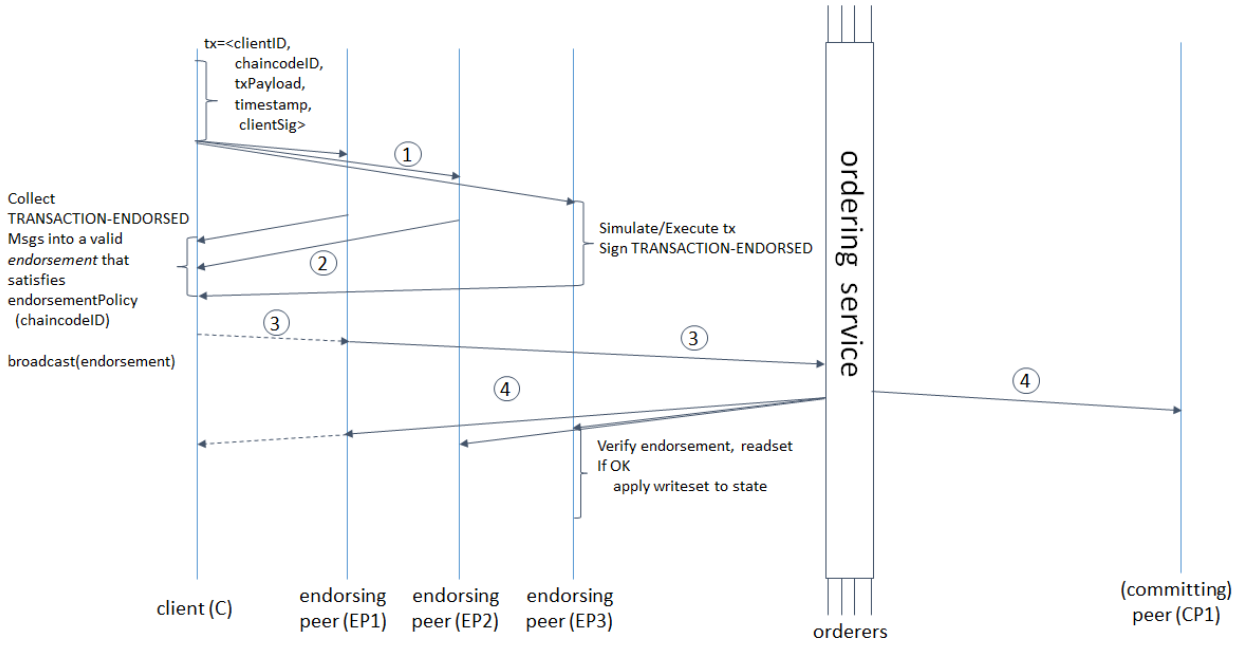


Figure 1. Illustration of one possible transaction flow (common-case path).

3. Endorsement policies

3.1. Endorsement policy specification

An **endorsement policy**, is a condition on what *endorses* a transaction. Blockchain peers have a pre-specified set of endorsement policies, which are referenced by a `deploy` transaction that installs specific chaincode. Endorsement policies can be parametrized, and these parameters can be specified by a `deploy` transaction.

To guarantee blockchain and security properties, the set of endorsement policies **should be a set of proven policies** with limited set of functions in order to ensure bounded execution time (termination), determinism, performance and security guarantees.

Dynamic addition of endorsement policies (e.g., by `deploy` transaction on chaincode deploy time) is very sensitive in terms of bounded policy evaluation time (termination), determinism, performance and security guarantees. Therefore, dynamic addition of endorsement policies is not allowed, but can be supported in future.

3.2. Transaction evaluation against endorsement policy

A transaction is declared valid only if it has been endorsed according to the policy. An invoke transaction for a chaincode will first have to obtain an *endorsement* that satisfies the chaincode's policy or it will not be committed. This takes place through the interaction between the submitting client and endorsing peers as explained in Section 2.

Formally the endorsement policy is a predicate on the endorsement, and potentially further state that evaluates to TRUE or FALSE. For deploy transactions the endorsement is obtained according to a system-wide policy (for example, from the system chaincode).

An endorsement policy predicate refers to certain variables. Potentially it may refer to:

1. keys or identities relating to the chaincode (found in the metadata of the chaincode), for example, a set of endorsers;
2. further metadata of the chaincode;

3. elements of the `endorsement` and `endorsement.tran-proposal` ;
4. and potentially more.

The above list is ordered by increasing expressiveness and complexity, that is, it will be relatively simple to support policies that only refer to keys and identities of nodes.

The evaluation of an endorsement policy predicate must be deterministic. An endorsement shall be evaluated locally by every peer such that a peer does *not* need to interact with other peers, yet all correct peers evaluate the endorsement policy in the same way.

3.3. Example endorsement policies

The predicate may contain logical expressions and evaluates to TRUE or FALSE. Typically the condition will use digital signatures on the transaction invocation issued by endorsing peers for the chaincode.

Suppose the chaincode specifies the endorser set `E = {Alice,Bob,Charlie,Dave,Eve,Frank,George}` . Some example policies:

- A valid signature from on the same `tran-proposal` from all members of E.
- A valid signature from any single member of E.
- Valid signatures on the same `tran-proposal` from endorsing peers according to the condition `(Alice OR Bob) AND (any two of: Charlie,Dave,Eve,Frank,George)` .
- Valid signatures on the same `tran-proposal` by any 5 out of the 7 endorsers. (More generally, for chaincode with `n > 3f` endorsers, valid signatures by any `2f+1` out of the `n` endorsers, or by any group of *more* than `(n+f)/2` endorsers.)
- Suppose there is an assignment of "stake" or "weights" to the endorsers, like `{Alice=49,Bob=15,Charlie=15,Dave=10,Eve=7,Frank=3,George=1}` , where the total stake is 100: The policy requires valid signatures from a set that has a majority of the stake (i.e., a group with combined stake strictly more than 50), such as `{Alice,X}` with any `X` different from George, or `{everyone together except Alice}` . And so on.
- The assignment of stake in the previous example condition could be static (fixed in the metadata of the chaincode) or dynamic (e.g., dependent on the state of the chaincode and be modified during the execution).
- Valid signatures from (Alice OR Bob) on `tran-proposal1` and valid signatures from `(any two of: Charlie,Dave,Eve,Frank,George)` on `tran-proposal2` , where `tran-proposal1` and `tran-proposal2` differ only in their endorsing peers and state updates.

How useful these policies are will depend on the application, on the desired resilience of the solution against failures or misbehavior of endorsers, and on various other properties.

4 (post-v1). Validated ledger and `PeerLedger` checkpointing (pruning)

4.1. Validated ledger (VLedger)

To maintain the abstraction of a ledger that contains only valid and committed transactions (that appears in Bitcoin, for example), peers may, in addition to state and Ledger, maintain the *Validated Ledger (or VLedger)*. This is a hash chain derived from the ledger by filtering out invalid transactions.

The construction of the VLedger blocks (called here *vBlocks*) proceeds as follows. As the `PeerLedger` blocks may contain invalid transactions (i.e., transactions with invalid endorsement or with invalid version dependencies), such transactions are filtered out by peers before a transaction from a block becomes added to a vBlock. Every peer does this by itself (e.g., by using the bitmask associated with `PeerLedger`). A vBlock is defined as a block without the

invalid transactions, that have been filtered out. Such vBlocks are inherently dynamic in size and may be empty. An illustration of vBlock construction is given in the figure below.

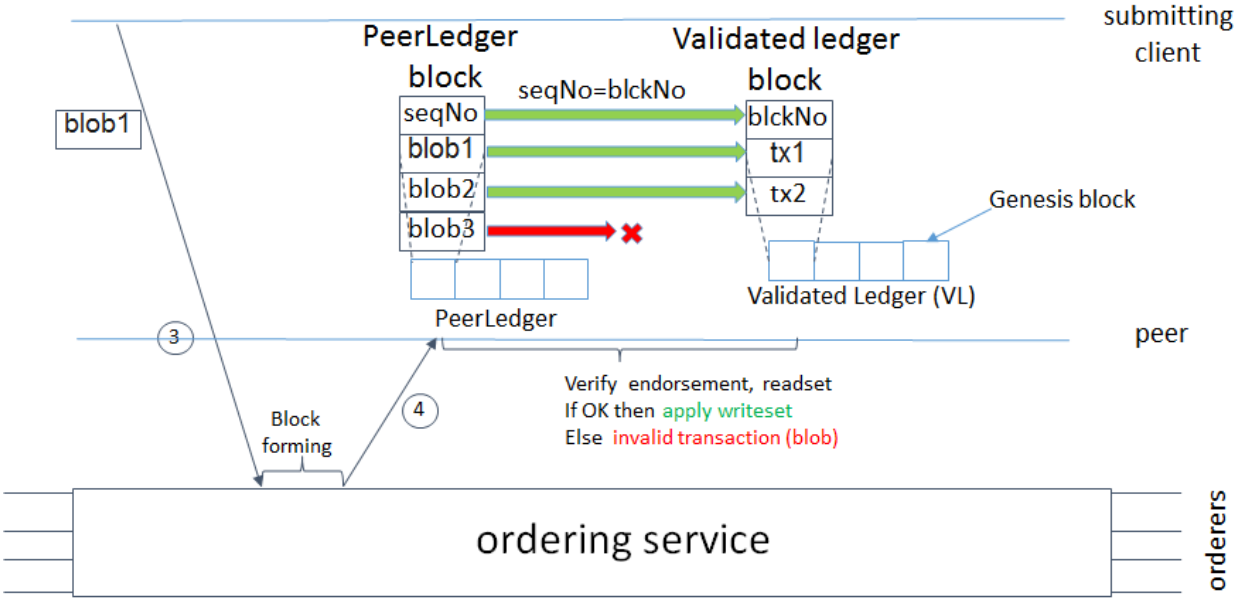


Figure 2. Illustration of validated ledger block (vBlock) formation from ledger (PeerLedger) blocks.

vBlocks are chained together to a hash chain by every peer. More specifically, every block of a validated ledger contains:

- The hash of the previous vBlock.
- vBlock number.
- An ordered list of all valid transactions committed by the peers since the last vBlock was computed (i.e., list of valid transactions in a corresponding block).
- The hash of the corresponding block (in `PeerLedger`) from which the current vBlock is derived.

All this information is concatenated and hashed by a peer, producing the hash of the vBlock in the validated ledger.

4.2. `PeerLedger` Checkpointing

The ledger contains invalid transactions, which may not necessarily be recorded forever. However, peers cannot simply discard `PeerLedger` blocks and thereby prune `PeerLedger` once they establish the corresponding vBlocks. Namely, in this case, if a new peer joins the network, other peers could not transfer the discarded blocks (pertaining to `PeerLedger`) to the joining peer, nor convince the joining peer of the validity of their vBlocks.

To facilitate pruning of the `PeerLedger` , this document describes a *checkpointing* mechanism. This mechanism establishes the validity of the vBlocks across the peer network and allows checkpointed vBlocks to replace the discarded `PeerLedger` blocks. This, in turn, reduces storage space, as there is no need to store invalid transactions. It also reduces the work to reconstruct the state for new peers that join the network (as they do not need to establish validity of individual transactions when reconstructing the state by replaying `PeerLedger` , but may simply replay the state updates contained in the validated ledger).

4.2.1. Checkpointing protocol

Checkpointing is performed periodically by the peers every *CHK* blocks, where *CHK* is a configurable parameter. To initiate a checkpoint, the peers broadcast (e.g., gossip) to other peers message

<CHECKPOINT,blocknohash,blockno,stateHash,peerSig> , where blockno is the current block-number and blocknohash is its respective hash, stateHash is the hash of the latest state (produced by e.g., a Merkle hash) upon validation of block blockno and peerSig is peer's signature on (CHECKPOINT,blocknohash,blockno,stateHash) , referring to the validated ledger.

A peer collects CHECKPOINT messages until it obtains enough correctly signed messages with matching blockno , blocknohash and stateHash to establish a *valid checkpoint* (see Section 4.2.2.).

Upon establishing a valid checkpoint for block number blockno with blocknohash , a peer:

- if blockno>latestValidCheckpoint.blockno , then a peer assigns latestValidCheckpoint=(blocknohash,blockno) ,
- stores the set of respective peer signatures that constitute a valid checkpoint into the set latestValidCheckpointProof ,
- stores the state corresponding to stateHash to latestValidCheckpointedState ,
- (optionally) prunes its PeerLedger up to block number blockno (inclusive).

4.2.2. Valid checkpoints

Clearly, the checkpointing protocol raises the following questions: *When can a peer prune its ''PeerLedger''? How many ''CHECKPOINT'' messages are "sufficiently many"?*. This is defined by a *checkpoint validity policy*, with (at least) two possible approaches, which may also be combined:

- *Local (peer-specific) checkpoint validity policy (LCVP).* A local policy at a given peer *p* may specify a set of peers which peer *p* trusts and whose CHECKPOINT messages are sufficient to establish a valid checkpoint. For example, LCVP at peer *Alice* may define that *Alice* needs to receive CHECKPOINT message from Bob, or from *both Charlie* and *Dave*.
- *Global checkpoint validity policy (GCVP).* A checkpoint validity policy may be specified globally. This is similar to a local peer policy, except that it is stipulated at the system (blockchain) granularity, rather than peer granularity. For instance, GCVP may specify that:
  - each peer may trust a checkpoint if confirmed by *11* different peers.
  - in a specific deployment in which every orderer is collocated with a peer in the same machine (i.e., trust domain) and where up to *f* orderers may be (Byzantine) faulty, each peer may trust a checkpoint if confirmed by *f+1* different peers collocated with orderers.

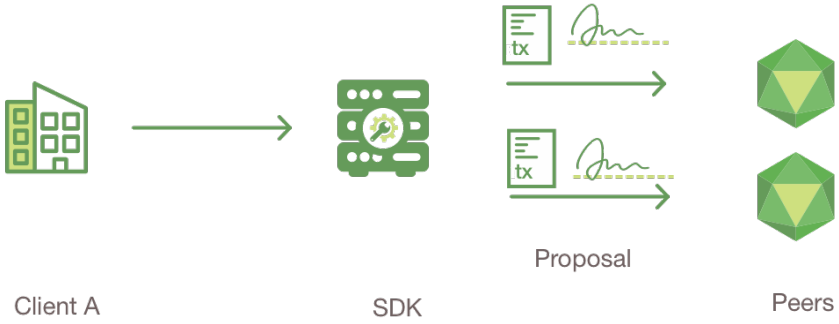
Transaction Flow

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.

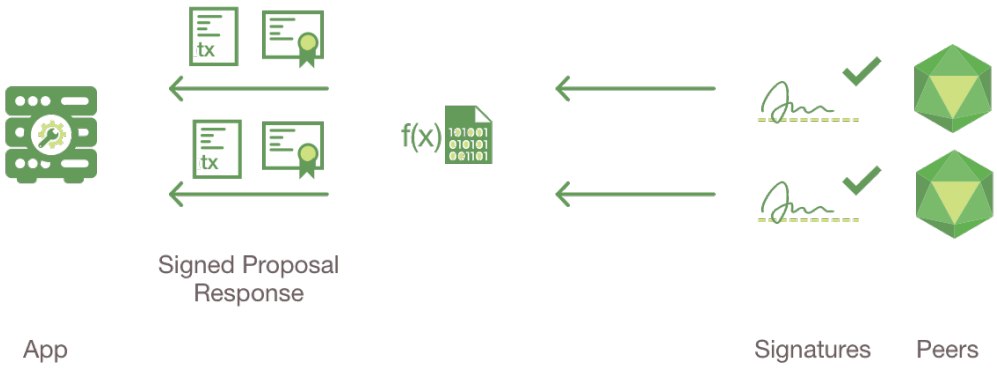
The chaincode (containing a set of key value pairs representing the initial state of the radish market) is installed on the peers and instantiated on the channel. The chaincode contains logic defining a set of transaction instructions and the agreed upon price for a radish. An endorsement policy has 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. The 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 which generates a transaction proposal. The proposal is a request to invoke a chaincode function so that data can be read and/or written to the ledger (i.e. write new key value pairs for the assets). 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.



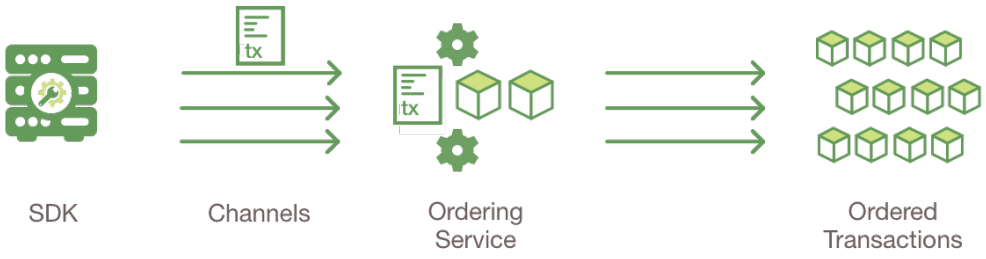
2. **Endorsing peers verify signature & execute the transaction**

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 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. 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.

{The MSP is a peer component that allows them 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 user is entitled to submit a transaction to that channel.}

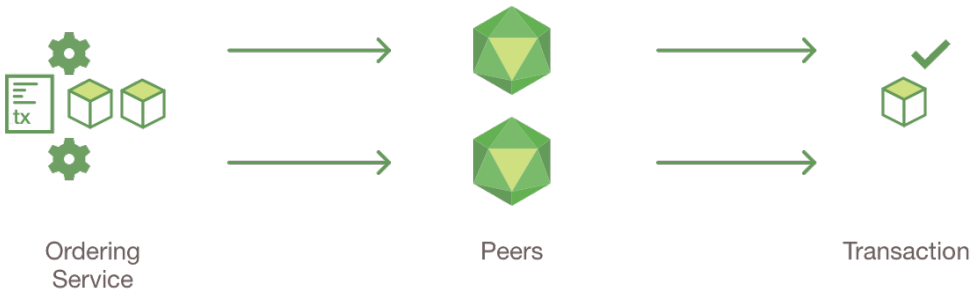
3. **Proposal responses are inspected**

The application verifies the endorsing peer signatures and compares the proposal responses to determine if the proposal responses are the same. If the chaincode only queried the ledger, the application would inspect the query response and would typically not submit the transaction to Ordering Service. If the client application intends to submit the transaction to 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.



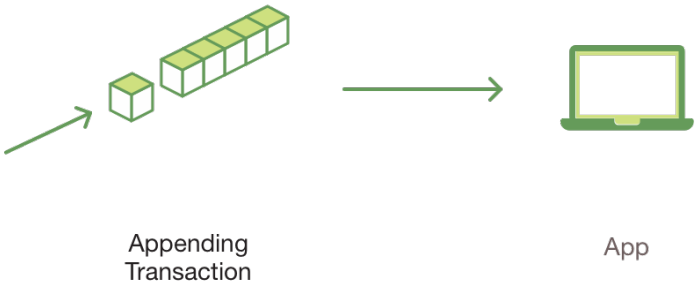
4. **Client assembles endorsements into a transaction**

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, 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: See the swimlane diagram to better understand the server side flow and the protobuffers.

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.

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 peer**s, 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 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.

Capability Requirements

Because Fabric is a distributed system that will usually involve multiple organizations (sometimes in different countries or even continents), it is possible (and typical) that many different versions of Fabric code will exist in the network. Nevertheless, it's vital that networks process transactions in the same way so that everyone has the same view of the current network state.

This means that every network – and every channel within that network – must define a set of what we call "capabilities" to be able to participate in processing transactions. For example, Fabric v1.1 introduces new MSP role types of "Peer" and "Client". However, if a v1.0 peer does not understand these new role types, it will not be able to appropriately evaluate an endorsement policy that references them. This means that before the new role types may be used,

the network must agree to enable the v1.1 `channel` capability requirement, ensuring that all peers come to the same decision.

Only binaries which support the required capabilities will be able to participate in the channel, and newer binary versions will not enable new validation logic until the corresponding capability is enabled. In this way, capability requirements ensure that even with disparate builds and versions, it is not possible for the network to suffer a state fork.

Defining Capability Requirements

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

- **Channel:** these capabilities apply to both peer and orderers and are located in the root `Channel` group.
- **Orderer:** apply to orderers only and are located in the `Orderer` group.
- **Application:** apply to peers only and are located in the `Application` group.

The capabilities are broken into these groups in order to align with the existing administrative structure. Updating orderer capabilities is something the ordering orgs would manage independent of the application orgs. Similarly, updating application capabilities is something only the application admins would manage. By splitting the capabilities between "Orderer" and "Application", a hypothetical network could run a v1.6 ordering service while supporting a v1.3 peer application network.

However, some capabilities cross both the 'Application' and 'Orderer' groups. As we saw earlier, adding a new MSP role type is something both the orderer and application admins agree to and need to recognize. The orderer must understand the meaning of MSP roles in order to allow the transactions to pass through ordering, while the peers must understand the roles in order to validate the transaction. These kinds of capabilities – which span both the application and orderer components – are defined in the top level "Channel" group.

Note: It is possible that the channel capabilities are defined to be at version v1.3 while the orderer and application capabilities are defined to be at version 1.1 and v1.4, respectively. Enabling a capability at the "Channel" group level does not imply that this same capability is available at the more specific "Orderer" and "Application" group levels.

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: We have a two documents that talk through different aspects of channel reconfigurations. First, we have a tutorial that will take you through the process of *Adding an Org to a Channel*. And we also have a document that talks through *Updating a Channel Configuration* which gives an overview of the different kinds of updates that are possible as well as a fuller look at the signature process.

Because new channels copy the configuration of the Orderer System Channel by default, new channels will automatically be configured to work with the orderer and channel capabilities of the Orderer System Channel and the

application capabilities specified by the channel creation transaction. Channels that already exist, however, must be reconfigured.

The schema for the Capabilities value is defined in the protobuf as:

```
message Capabilities {
      map<string, Capability> capabilities = 1;
}

message Capability { }
```

As an example, rendered in JSON:

```
{
    "capabilities": {
        "V1_1": {}
    }
}
```

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).

The simplest way to enable capabilities is to pick a v1.1 sample profile and customize it for your network. For example:

```
SampleSingleMSPSoloV1_1:
    Capabilities:
        <<: *GlobalCapabilities
    Orderer:
        <<: *OrdererDefaults
        Organizations:
            - *SampleOrg
        Capabilities:
            <<: *OrdererCapabilities
    Consortiums:
        SampleConsortium:
            Organizations:
                - *SampleOrg
```

Note that there is a `Capabilities` section defined at the root level (for the channel capabilities), and at the Orderer level (for orderer capabilities). The sample above uses a YAML reference to include the capabilities as defined at the bottom of the YAML.

When defining the orderer system channel there is no Application section, as those capabilities are defined during the creation of an application channel. To define a new channel's application capabilities at channel creation time, the application admins should model their channel creation transaction after the `SampleSingleMSPChannelV1_1` profile.

```
SampleSingleMSPChannelV1_1:
     Consortium: SampleConsortium
     Application:
         Organizations:
             - *SampleOrg
         Capabilities:
             <<: *ApplicationCapabilities
```

Here, the Application section has a new element `Capabilities` which references the `ApplicationCapabilities` section defined at the end of the YAML.

Note: The capabilities for the Channel and Orderer sections are inherited from the definition in the ordering system channel and are automatically included by the orderer during the process of channel creation.

CouchDB as the State Database

State Database options

State database options include LevelDB and CouchDB. LevelDB is the default key-value state database embedded in the peer process. CouchDB is an optional alternative external state database. Like the LevelDB key-value store, CouchDB can store any binary data that is modeled in chaincode (CouchDB attachment functionality is used internally for non-JSON binary data). But as a JSON document store, CouchDB additionally enables rich query against the chaincode data, when chaincode values (e.g. assets) are modeled as JSON data.

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.

If you model assets as JSON and use CouchDB, you can also perform complex rich queries against the chaincode data values, using the CouchDB JSON query language within chaincode. These types of queries are excellent for understanding what is on the ledger. Proposal responses for these types of queries are typically useful to the client application, but are not typically submitted as transactions to the ordering service. In fact, there is no guarantee the result set is stable between chaincode execution and commit time for rich queries, and therefore rich queries are not appropriate for use in update transactions, unless your application can guarantee the result set is stable between chaincode execution time and commit time, or can handle potential changes in subsequent transactions. 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, and you would miss this "phantom" item.

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 chaincode asset data as JSON, so that you have the option to perform complex rich queries if needed in the future.

Note: A JSON document cannot use the following field names at the top level. These are reserved for internal use.

- `_deleted`
- `_id`
- `_rev`
- `~version`

Using CouchDB from Chaincode

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>}}
```

Indexes in CouchDB are required in order to make JSON queries efficient and are required for any JSON query with a sort. 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. When the chaincode is both installed on a peer and instantiated on one of the peer's channels, the index will automatically be deployed to the peer's channel and chaincode specific state database (if it has been configured to use CouchDB). If you install the chaincode first and then instantiate the chaincode on the channel, the index will be deployed at chaincode **instantiation** time. If the chaincode is already instantiated on a channel and you later install the chaincode on a peer, 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 instantiate a subsequent version of the chaincode, 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.

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
  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
     maxRetriesOnStartup: 10
     # CouchDB request timeout (unit: duration, e.g. 20s)
     requestTimeout: 35s
     # Limit on the number of records to return per query
     queryLimit: 10000
```

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.

Note: CouchDB peer options are read on each peer startup.

Peer channel-based event services

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 two 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).

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.

- `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.

How to register for events

Registration for events from either service 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, both services use the Channel Readers policy to determine whether to authorize requesting clients for events.

Overview of deliver response messages

The event services send back `DeliverResponse` messages.

Each message contains one of the following:

- status – HTTP status code. Both 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.
- 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).

SDK event documentation

For further details on using the event services, refer to the SDK documentation.

Read-Write set semantics

This documents discusses the details of the current implementation about the semantics of read-write sets.

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 versions 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 for making 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"
      <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`.

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.

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.

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, Hyperledger Fabric implements a **gossip data dissemination protocol**.

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 causations 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 Hyperledger 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.

Leader election

The leader election mechanism is used to **elect** one peer per each organization which will maintain connection with ordering service and initiate distribution of newly arrived blocks across peers of its own organization. Leveraging leader election provides system with ability to efficiently utilize bandwidth of the ordering service. There are two possible operation modes for leader election module:

1. **Static** - system administrator manually configures one peer in the organization to be the leader, e.g. one to maintain open connection with the ordering service.
2. **Dynamic** - peers execute a leader election procedure to select one peer in an organization to become leader, pull blocks from the ordering service, and disseminate blocks to the other peers in the organization..

Static leader election

Using static leader election allows to manually define a set of leader peers within the organization, it's possible to define a single node to be a leader or all available peers, it should be taken into account that - making too many peers

to connect to the ordering service might lead to inefficient bandwidth utilization. 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 overrided with environmental variables:

```
export CORE_PEER_GOSSIP_USELEADERELECTION=false
export CORE_PEER_GOSSIP_ORGLEADER=true
```

Note:

1. 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_USELEADERELECTION` 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. Leader is elected for set of peers for each organization independently.

Elected leader is responsible to send the **heartbeat** messages to the rest of the peers as an evidence of liveness. If one or more peers won't get **heartbeats** updates during period of time, they will initiate a new round of leader election procedure, eventually selecting a new leader. In case of a network partition in the worst case there will be more than one active leader for organization thus to guarantee resiliency and availability allowing the organization's peers to continue making progress. After the network partition is healed one of the leaders will relinquish its leadership, therefore in steady state and in no presence of network partitions for each organization 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 overrided with environmental variables:

```
export CORE_PEER_GOSSIP_USELEADERELECTION=true
export CORE_PEER_GOSSIP_ORGLEADER=false
```

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 peers' channel subscriptions.

Notes:

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 7

Hyperledger Fabric FAQ

Endorsement

Endorsement architecture:

17. How many peers in the network need to endorse a transaction?

A. The number of peers required to endorse a transaction is driven by the endorsement policy that is specified at chaincode deployment time.

17. Does an application client need to connect to all peers?

A. Clients only need to connect to as many peers as are required by the endorsement policy for the chaincode.

Security & Access Control

Data Privacy and Access Control:

17. How do I ensure data privacy?

A. 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, within a channel you can restrict the input data to chaincode to the set of endorsers only, by using visibility settings. The visibility setting will determine whether input and output chaincode data is included in the submitted transaction, versus just output data. Third, you 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.

17. Do the orderers see the transaction data?

A. 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, and you are only concerned about the input data, then you can use visibility settings. The visibility setting will determine whether input and output chaincode data is included in the submitted transaction, versus just output data. Therefore, the input data can be private to the endorsers only. If you do not want the orderers to see chaincode output, then you can hash or encrypt the data before calling chaincode. If you hash the data then you will need to provide a meansto share the source data. If you encrypt the data then you will need to provide a means to share the decryption keys.

Application-side Programming Model

Transaction execution result:

17. How do application clients know the outcome of a transaction?

A. 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.

Ledger queries:

17. How do I query the ledger data?

A. 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.

17. How do I query the historical data to understand data provenance?

A. The chaincode API `GetHistoryForKey()` will return history of values for a key.

Q. How to guarantee the query result is correct, especially when the peer being queried may be recovering and catching up on block processing?

A. The client can query multiple peers, compare their block heights, compare their query results, and favor the peers at the higher block heights.

Chaincode (Smart Contracts and Digital Assets)

17. Does Hyperledger Fabric support smart contract logic?

A. 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.

17. How do I create a business contract?

A. 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.

17. How do I create assets?

A. 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.

17. Which languages are supported for writing chaincode?

A. Chaincode can be written in any programming language and executed in containers. The first fully supported chaincode language is Golang.

Support for additional languages and the development of a templating language have been discussed, and more details will be released in the near future.

It is also possible to build Hyperledger Fabric applications using Hyperledger Composer.

17. Does the Hyperledger Fabric have native currency?

A. 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.

Differences in Most Recent Releases

17. Where can I find what are the highlighted differences between releases?

A. The differences between any subsequent releases are provided together with the *Release Notes*.

17. Where to get help for the technical questions not answered above?

1. Please use StackOverflow.

CHAPTER 8

Ordering Service FAQ

General

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 are two plugins built against these interfaces already: solo and kafka. You can study them to take cues 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*.

Solo

Question How can I deploy Solo in production?

Answer Solo is not intended for production. It is not, and will never be, fault tolerant.

Kafka

Question How do I remove a node from the ordering service?

Answer This is a two step-process:

1. Add the node's certificate to the relevant orderer's MSP CRL to prevent peers/clients from connecting to it.
2. Prevent the node from connecting to the Kafka cluster by leveraging standard Kafka access control measures such as TLS CRLs, or firewalling.

Question I have never deployed a Kafka/ZK cluster before, and I want to use the Kafka-based ordering service. How do I proceed?

Answer The Hyperledger Fabric documentation assumes the reader generally has the operational expertise to setup, configure, and manage a Kafka cluster (see *Caveat emptor*). If you insist on proceeding without such expertise, you should complete, *at a minimum*, the first 6 steps of the Kafka Quickstart guide before experimenting with the Kafka-based ordering service.

Question Where can I find a Docker composition for a network that uses the Kafka-based ordering service?

Answer Consult the end-to-end CLI example.

Question Why is there a ZooKeeper dependency in the Kafka-based ordering service?

Answer Kafka uses it internally for coordination between its brokers.

Question I'm trying to follow the BYFN example and get a "service unavailable" error, what should I do?

Answer Check the ordering service's logs. A "Rejecting deliver request because of consenter error" log message is usually indicative of a connection problem with the Kafka cluster. Ensure that the Kafka cluster is set up properly, and is reachable by the ordering service's nodes.

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 1.x cycle, i.e. it will come with a minor version upgrade in Fabric. Track FAB-33 for updates.

CHAPTER 9

Contributions Welcome!

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.

Maintainers

Active Maintainers

Name	Gerrit	GitHub	RocketChat	email
Artem Barger	c0rwin	c0rwin	c0rwin	bartem@il.ibm.com
Binh Nguyen	binhn	binhn	binhn	binh1010010110@gmail.com
Chris Ferris	ChristopherFerris	christo4ferris	cbf	chris.ferris@gmail.com
Dave Enyeart	denyeart	denyeart	dave.enyeart	enyeart@us.ibm.com
Gari Singh	mastersingh24	mastersingh24	garisingh	gari.r.singh@gmail.com
Greg Haskins	greg.haskins	ghaskins	ghaskins	gregory.haskins@gmail.com
Jason Yellick	jyellick	jyellick	jyellick	jyellick@us.ibm.com
Jim Zhang	jimthematrix	jimthematrix	jimthematrix	jim_the_matrix@hotmail.com
Jonathan Levi	JonathanLevi	JonathanLevi	JonathanLevi	jonathan@hacera.com
Keith Smith	smithbk	smithbk	smithbk	bksmith@us.ibm.com
Kostas Christidis	kchristidis	kchristidis	kostas	kostas@gmail.com
Manish Sethi	manish-sethi	manish-sethi	manish-sethi	manish.sethi@gmail.com
Srinivasan Muralidharan	muralisr	muralisrini	muralisr	srinivasan.muralidharan99@gmail.com
Yacov Manevich	yacovm	yacovm	yacovm	yacovm@il.ibm.com
Yaoguo Jiang	jiangyaoguo	jiangyaoguo	jiangyaoguo	jiangyaoguo@gmail.com

Release Managers

Name	Gerrit	GitHub	RocketChat	email
Chris Ferris	ChristopherFerris	christo4ferris	cbf	chris.ferris@gmail.com
Dave Enyeart	denyeart	denyeart	dave.enyeart	enyeart@us.ibm.com
Gari Singh	mastersingh24	mastersingh24	garisingh	gari.r.singh@gmail.com

Retired Maintainers

Gabor Hosszu	hgabre	gabre	hgabor	gabor@digitalasset.com
Sheehan Anderson	sheehan	srderson	sheehan	sranderson@gmail.com
Tamas Blummer	TamasBlummer	tamasblummer	tamas	tamas@digitalasset.com

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**:

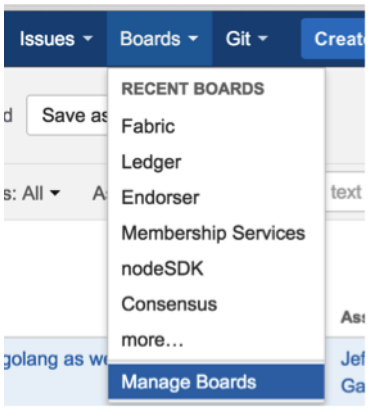


Fig. 9.1: Jira boards

Now on the left side of the screen click on **All boards**:

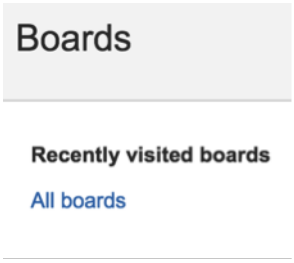


Fig. 9.2: Jira boards

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:

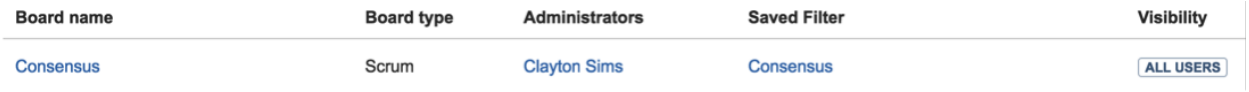


Fig. 9.3: Jira boards

When you click on Consensus under **Board name** you will be directed to a page that contains the following columns:

Fig. 9.4: Jira boards

The meanings to these columns are as follows:

- 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 Gerrit
- 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:

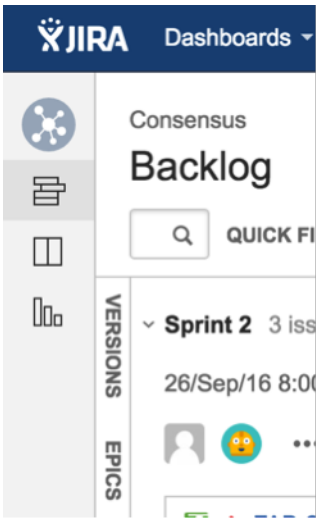


Fig. 9.5: Jira boards

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.

Setting up the development environment

Overview

Prior to the v1.0.0 release, the development environment utilized Vagrant running an Ubuntu image, which in turn launched Docker containers as a means of ensuring a consistent experience for developers who might be working with varying platforms, such as macOS, Windows, Linux, or whatever. Advances in Docker have enabled native support on the most popular development platforms: macOS and Windows. Hence, we have reworked our build to take full advantage of these advances. While we still maintain a Vagrant based approach that can be used for older versions of macOS and Windows that Docker does not support, we strongly encourage that the non-Vagrant development setup be used.

Note that while the Vagrant-based development setup could not be used in a cloud context, the Docker-based build does support cloud platforms such as AWS, Azure, Google and IBM to name a few. Please follow the instructions for Ubuntu builds, below.

Prerequisites

- Git client
- Go - 1.9 or later (for v1.0.X releases, use Go 1.7.X)
- (macOS) Xcode must be installed
- Docker - 17.06.2-ce or later
- Docker Compose - 1.14.0 or later
- Pip
- (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:

```
brew install gnu-tar --with-default-names
```

- (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

`pip` and `behave`

```
pip install --upgrade pip

#PIP packages required for some behave tests
pip install -r devenv/bddtests-requirements.txt
```

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

Since Hyperledger Fabric is written in `Go` , you'll need to clone the source 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/hyperledger
cd github.com/hyperledger
```

Recall that we are using `Gerrit` for source control, which has its own internal git repositories. Hence, we will need to clone from *Gerrit*. For brevity, the command is as follows:

```
git clone ssh://LFID@gerrit.hyperledger.org:29418/fabric && scp -p -P 29418␣
↪LFID@gerrit.hyperledger.org:hooks/commit-msg fabric/.git/hooks/
```

Note: Of course, you would want to replace `LFID` with your own *Linux Foundation ID*.

Bootstrapping the VM using Vagrant

If you are planning on using the Vagrant developer environment, the following steps apply. **Again, we recommend against its use except for developers that are limited to older versions of macOS and Windows that are not supported by Docker for Mac or Windows.**

```
cd $GOPATH/src/github.com/hyperledger/fabric/devenv
vagrant up
```

Go get coffee... this will take a few minutes. Once complete, you should be able to `ssh` into the Vagrant VM just created.

```
vagrant ssh
```

Once inside the VM, you can find the source under `$GOPATH/src/github.com/hyperledger/fabric`. It is also mounted as `/hyperledger`.

Building Hyperledger Fabric

Once you have all the dependencies installed, and have cloned the repository, you can proceed to *build and test* Hyperledger Fabric.

Notes

NOTE: Any time you change any of the files in your local fabric directory (under `$GOPATH/src/github.com/hyperledger/fabric`), the update will be instantly available within the VM fabric directory.

NOTE: If you intend to run the development environment behind an HTTP Proxy, you need to configure the guest so that the provisioning process may complete. You can achieve this via the *vagrant-proxyconf* plugin. Install with `vagrant plugin install vagrant-proxyconf` and then set the VAGRANT_HTTP_PROXY and VAGRANT_HTTPS_PROXY environment variables *before* you execute `vagrant up`. More details are available here: https://github.com/tmatilai/vagrant-proxyconf/

NOTE: The first time you run this command it may take quite a while to complete (it could take 30 minutes or more depending on your environment) and at times it may look like it's not doing anything. As long you don't get any error messages just leave it alone, it's all good, it's just cranking.

NOTE to Windows 10 Users: There is a known problem with vagrant on Windows 10 (see mitchellh/vagrant#6754). If the `vagrant up` command fails it may be because you do not have the Microsoft Visual C++ Redistributable package installed. You can download the missing package at the following address: http://www.microsoft.com/en-us/download/details.aspx?id=8328

NOTE: The inclusion of the miekg/pkcs11 package introduces an external dependency on the libtdl.h header file during a build of fabric. Please ensure your libtool and libtdhl-dev packages are installed. Otherwise, you may get a ltdl.h header missing error. You can download the missing package by command: `sudo apt-get install -y build-essential git make curl unzip g++ libtool`.

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
```

Running the unit tests

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.

Running Behave BDD Tests

Note: currently, the behave tests must be run from within in the Vagrant environment. See the *development environment* setup instructions if you have not already set up your Vagrant environment.

Behave tests will setup networks of peers with different security and consensus configurations and verify that transactions run properly. To run these tests

```
cd $GOPATH/src/github.com/hyperledger/fabric
make behave
```

Some of the Behave tests run inside Docker containers. If a test fails and you want to have the logs from the Docker containers, run the tests with this option:

```
cd $GOPATH/src/github.com/hyperledger/fabric/bddtests
behave -D logs=Y
```

Building outside of Vagrant

It is possible to build the project and run peers outside of Vagrant. Generally speaking, one has to 'translate' the vagrant setup file to the platform of your choice.

Building on Z

To make building on Z easier and faster, this script is provided (which is similar to the setup file provided for vagrant). This script has been tested only on RHEL 7.2 and has some assumptions one might want to re-visit (firewall settings, development as root user, etc.). It is however sufficient for development in a personally-assigned VM instance.

To get started, from a freshly installed OS:

```
sudo su
yum install git
mkdir -p $HOME/git/src/github.com/hyperledger
cd $HOME/git/src/github.com/hyperledger
```

```
git clone http://gerrit.hyperledger.org/r/fabric
source fabric/devenv/setupRHELonZ.sh
```

From this point, you can proceed as described above for the Vagrant development environment.

```
cd $GOPATH/src/github.com/hyperledger/fabric
make peer unit-test behave
```

Building on Power Platform

Development and build on Power (ppc64le) systems is done outside of vagrant as outlined *here*. For ease of setting up the dev environment on Ubuntu, invoke this script as root. This script has been validated on Ubuntu 16.04 and assumes certain things (like, development system has OS repositories in place, firewall setting etc) and in general can be improvised further.

To get started on Power server installed with Ubuntu, first ensure you have properly setup your Host's GOPATH environment variable. Then, execute the following commands to build the fabric code:

```
mkdir -p $GOPATH/src/github.com/hyperledger
cd $GOPATH/src/github.com/hyperledger
git clone http://gerrit.hyperledger.org/r/fabric
sudo ./fabric/devenv/setupUbuntuOnPPC64le.sh
cd $GOPATH/src/github.com/hyperledger/fabric
make dist-clean all
```

Building on Centos 7

You will have to build CouchDB from source because there is no package available from the distribution. If you are planning a multi-orderer arrangement, you will also need to install Apache Kafka from source. Apache Kafka includes both Zookeeper and Kafka executables and supporting artifacts.

```
export GOPATH={directory of your choice}
mkdir -p $GOPATH/src/github.com/hyperledger
FABRIC=$GOPATH/src/github.hyperledger/fabric
git clone https://github.com/hyperledger/fabric $FABRIC
cd $FABRIC
git checkout master # <-- only if you want the master branch
export PATH=$GOPATH/bin:$PATH
make native
```

If you are not trying to build for docker, you only need the natives.

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, logging level manipulation through the environment is shown below:

```
CORE_PEER_LOGGING_LEVEL=CRITICAL peer
```

Requesting a Linux Foundation Account

Contributions to the Hyperledger Fabric code base require a Linux Foundation account — follow the steps below to create one if you don't already have one.

Creating a Linux Foundation ID

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 Gerrit by selecting `Sign In`, and use your new Linux Foundation account ID to sign in.

Configuring Gerrit to Use SSH

Gerrit uses SSH to interact with your Git client. If you already have an SSH key pair, you can skip the part of this section that explains how to generate one.

What follows explains how to generate an SSH key pair in a Linux environment — follow the equivalent steps on your OS.

First, create an SSH key pair with the command:

```
ssh-keygen -t rsa -C "John Doe john.doe@example.com"
```

Note: This will ask you for a password to protect the private key as it generates a unique key. Please keep this password private, and DO NOT enter a blank password.

The generated SSH key pair can be found in the files `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub`.

Next, add the private key in the `id_rsa` file to your key ring, e.g.:

```
ssh-add ~/.ssh/id_rsa
```

Finally, add the public key of the generated key pair to the Gerrit server, with the following steps:

1. Go to Gerrit.
2. Click on your account name in the upper right corner.
3. From the pop-up menu, select `Settings`.
4. On the left side menu, click on `SSH Public Keys`.
5. Paste the contents of your public key `~/.ssh/id_rsa.pub` and click `Add key`.

Note: The `id_rsa.pub` file can be opened with any text editor. Ensure that all the contents of the file are selected, copied and pasted into the `Add SSH key` window in Gerrit.

Note: The SSH key generation instructions operate on the assumption that you are using the default naming. It is possible to generate multiple SSH keys and to name the resulting files differently. See the ssh-keygen documentation

for details on how to do that. Once you have generated non-default keys, you need to configure SSH to use the correct key for Gerrit. In that case, you need to create a `~/.ssh/config` file modeled after the one below.

```
host gerrit.hyperledger.org
 HostName gerrit.hyperledger.org
 IdentityFile ~/.ssh/id_rsa_hyperledger_gerrit
 User <LFID>
```

where <LFID> is your Linux Foundation ID and the value of IdentityFile is the name of the public key file you generated.

Warning: Potential Security Risk! Do not copy your private key `~/.ssh/id_rsa` . Use only the public `~/.ssh/id_rsa.pub` .

Checking Out the Source Code

Once you've set up SSH as explained in the previous section, you can clone the source code repository with the command:

```
git clone ssh://<LFID>@gerrit.hyperledger.org:29418/fabric fabric
```

You have now successfully checked out a copy of the source code to your local machine.

Working with Gerrit

Follow these instructions to collaborate on Hyperledger Fabric through the Gerrit review system.

Please be sure that you are subscribed to the mailing list and of course, you can reach out on chat if you need help.

Gerrit assigns the following roles to users:

- **Submitters**: May submit changes for consideration, review other code changes, and make recommendations for acceptance or rejection by voting +1 or -1, respectively.
- **Maintainers**: May approve or reject changes based upon feedback from reviewers voting +2 or -2, respectively.
- **Builders**: (e.g. Jenkins) May use the build automation infrastructure to verify the change.

Maintainers should be familiar with the *review process*. However, anyone is welcome to (and encouraged!) review changes, and hence may find that document of value.

Git-review

There's a **very** useful tool for working with Gerrit called git-review. This command-line tool can automate most of the ensuing sections for you. Of course, reading the information below is also highly recommended so that you understand what's going on behind the scenes.

Getting deeper into Gerrit

A comprehensive walk-through of Gerrit is beyond the scope of this document. There are plenty of resources available on the Internet. A good summary can be found here. We have also provided a set of *Best Practices* that you may find helpful.

Working with a local clone of the repository

To work on something, whether a new feature or a bugfix:

1. Open the Gerrit Projects page
2. Select the project you wish to work on.
3. Open a terminal window and clone the project locally using the `Clone with git hook` URL. Be sure that `ssh` is also selected, as this will make authentication much simpler:

```
git clone ssh://LFID@gerrit.hyperledger.org:29418/fabric && scp -p -P 29418␣
↪LFID@gerrit.hyperledger.org:hooks/commit-msg fabric/.git/hooks/
```

Note: If you are cloning the fabric project repository, you will want to clone it to the `$GOPATH/src/github.com/hyperledger` directory so that it will build, and so that you can use it with the Vagrant *development environment*.

4. Create a descriptively-named branch off of your cloned repository

```
cd fabric
git checkout -b issue-nnnn
```

5. Commit your code. For an in-depth discussion of creating an effective commit, please read *this document on submitting changes*.

```
git commit -s -a
```

Then input precise and readable commit msg and submit.

6. Any code changes that affect documentation should be accompanied by corresponding changes (or additions) to the documentation and tests. This will ensure that if the merged PR is reversed, all traces of the change will be reversed as well.

Submitting a Change

Currently, Gerrit is the only method to submit a change for review.

Note: Please review the *guidelines* for making and submitting a change.

Using git review

Note: if you prefer, you can use the *git-review* tool instead of the following. e.g.

Add the following section to `.git/config`, and replace `<USERNAME>` with your gerrit id.

```
[remote "gerrit"]
    url = ssh://<USERNAME>@gerrit.hyperledger.org:29418/fabric.git
    fetch = +refs/heads/*:refs/remotes/gerrit/*
```

Then submit your change with `git review`.

```
$ cd <your code dir>
$ git review
```

When you update your patch, you can commit with `git commit --amend` , and then repeat the `git review` command.

Not using git review

See the *directions for building the source code*.

When a change is ready for submission, Gerrit requires that the change be pushed to a special branch. The name of this special branch contains a reference to the final branch where the code should reside, once accepted.

For the Hyperledger Fabric repository, the special branch is called `refs/for/master` .

To push the current local development branch to the gerrit server, open a terminal window at the root of your cloned repository:

```
cd <your clone dir>
git push origin HEAD:refs/for/master
```

If the command executes correctly, the output should look similar to this:

```
Counting objects: 3, done.
Writing objects: 100% (3/3), 306 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote: Processing changes: new: 1, refs: 1, done
remote:
remote: New Changes:
remote:   https://gerrit.hyperledger.org/r/6 Test commit
remote:
To ssh://LFID@gerrit.hyperledger.org:29418/fabric
* [new branch]      HEAD -> refs/for/master
```

The gerrit server generates a link where the change can be tracked.

Reviewing Using Gerrit

- **Add**: This button allows the change submitter to manually add names of people who should review a change; start typing a name and the system will auto-complete based on the list of people registered and with access to the system. They will be notified by email that you are requesting their input.
- **Abandon**: This button is available to the submitter only; it allows a committer to abandon a change and remove it from the merge queue.
- **Change-ID**: This ID is generated by Gerrit (or system). It becomes useful when the review process determines that your commit(s) have to be amended. You may submit a new version; and if the same Change-ID header (and value) are present, Gerrit will remember it and present it as another version of the same change.
- **Status**: Currently, the example change is in review status, as indicated by "Needs Verified" in the upper-left corner. The list of Reviewers will all emit their opinion, voting +1 if they agree to the merge, -1 if they disagree. Gerrit users with a Maintainer role can agree to the merge or refuse it by voting +2 or -2 respectively.

Notifications are sent to the email address in your commit message's Signed-off-by line. Visit your Gerrit dashboard, to check the progress of your requests.

The history tab in Gerrit will show you the in-line comments and the author of the review.

Viewing Pending Changes

Find all pending changes by clicking on the `All --> Changes` link in the upper-left corner, or open this link.

If you collaborate in multiple projects, you may wish to limit searching to the specific branch through the search bar in the upper-right side.

Add the filter *project:fabric* to limit the visible changes to only those from Hyperledger Fabric.

List all current changes you submitted, or list just those changes in need of your input by clicking on `My --> Changes` or open this link

Submitting a Change to Gerrit

Carefully review the following before submitting a change. These guidelines apply to developers that are new to open source, as well as to experienced open source developers.

Change Requirements

This section contains guidelines for submitting code changes for review. For more information on how to submit a change using Gerrit, please see *Gerrit*.

Changes are submitted as Git commits. Each commit must contain:

- a short and descriptive subject line that is 72 characters or fewer, followed by a blank line.
- a change description with your logic or reasoning for the changes, followed by a blank line
- a Signed-off-by line, followed by a colon (Signed-off-by:)
- a Change-Id identifier line, followed by a colon (Change-Id:). Gerrit won't accept patches without this identifier.

A commit with the above details is considered well-formed.

All changes and topics sent to Gerrit must be well-formed. Informationally, `commit messages` must include:

- **what** the change does,
- **why** you chose that approach, and
- **how** you know it works – for example, which tests you ran.

Commits must *build cleanly* when applied on top of each other, thus avoiding breaking bisectability. Each commit must address a single identifiable issue and must be logically self-contained.

For example: One commit fixes whitespace issues, another renames a function and a third one changes the code's functionality. An example commit file is illustrated below in detail:

```
[FAB-XXXX] A short description of your change with no period at the end

You can add more details here in several paragraphs, but please keep each line
width less than 80 characters. A bug fix should include the issue number.

Change-Id: IF7b6ac513b2eca5f2bab9728ebd8b7e504d3cebe1
Signed-off-by: Your Name <commit-sender@email.address>
```

Include the issue ID in the one line description of your commit message for readability. Gerrit will link issue IDs automatically to the corresponding entry in Jira.

Each commit must also contain the following line at the bottom of the commit message:

```
Signed-off-by: Your Name <your@email.address>
```

The name in the Signed-off-by line and your email must match the change authorship information. Make sure your :file:`.git/config` is set up correctly. Always submit the full set of changes via Gerrit.

When a change is included in the set to enable other changes, but it will not be part of the final set, please let the reviewers know this.

Check that your change request is validated by the CI process

To ensure stability of the code and limit possible regressions, we use a Continuous Integration (CI) process based on Jenkins which triggers a build on several platforms and runs tests against every change request being submitted. It is your responsibility to check that your CR passes these tests. No CR will ever be merged if it fails the tests and you shouldn't expect anybody to pay attention to your CRs until they pass the CI tests.

To check on the status of the CI process, simply look at your CR on Gerrit, following the URL that was given to you as the result of the push in the previous step. The History section at the bottom of the page will display a set of actions taken by "Hyperledger Jobbuilder" corresponding to the CI process being executed.

Upon completion, "Hyperledger Jobbuilder" will add to the CR a *+1 vote* if successful and a *-1 vote* otherwise.

In case of failure, explore the logs linked from the CR History. If you spot a problem with your CR amend your commit and push it to update it. The CI process will kick off again.

If you see nothing wrong with your CR it might be that the CI process simply failed for some reason unrelated to your change. In that case you may want to restart the CI process by posting a reply to your CR with the simple content "reverify". Check the CI management page for additional information and options on this.

Reviewing a Change

1. Click on a link for incoming or outgoing review.
2. The details of the change and its current status are loaded:

- **Status:** Displays the current status of the change. In the example below, the status reads: Needs Verified.
- **Reply:** Click on this button after reviewing to add a final review message and a score, -1, 0 or +1.
- **Patch Sets:** If multiple revisions of a patch exist, this button enables navigation among revisions to see the changes. By default, the most recent revision is presented.
- **Download:** This button brings up another window with multiple options to download or checkout the current changeset. The button on the right copies the line to your clipboard. You can easily paste it into your git interface to work with the patch as you prefer.

Underneath the commit information, the files that have been changed by this patch are displayed.

3. Click on a filename to review it. Select the code base to differentiate against. The default is `Base` and it will generally be what is needed.
4. The review page presents the changes made to the file. At the top of the review, the presentation shows some general navigation options. Navigate through the patch set using the arrows on the top right corner. It is possible to go to the previous or next file in the set or to return to the main change screen. Click on the yellow sticky pad to add comments to the whole file.

The focus of the page is on the comparison window. The changes made are presented in green on the right versus the base version on the left. Double click to highlight the text within the actual change to provide feedback on a specific section of the code. Press *c* once the code is highlighted to add comments to that section.

5. After adding the comment, it is saved as a *Draft*.
6. Once you have reviewed all files and provided feedback, click the *green up arrow* at the top right to return to the main change page. Click the `Reply` button, write some final comments, and submit your score for the patch set. Click `Post` to submit the review of each reviewed file, as well as your final comment and score. Gerrit sends an email to the change-submitter and all listed reviewers. Finally, it logs the review for future reference. All individual comments are saved as *Draft* until the `Post` button is clicked.

Gerrit Recommended Practices

This document presents some best practices to help you use Gerrit more effectively. The intent is to show how content can be submitted easily. Use the recommended practices to reduce your troubleshooting time and improve participation in the community.

Browsing the Git Tree

Visit Gerrit then select `Projects --> List --> SELECT-PROJECT --> Branches`. Select the branch that interests you, click on `gitweb` located on the right-hand side. Now, `gitweb` loads your selection on the Git web interface and redirects appropriately.

Watching a Project

Visit Gerrit, then select `Settings`, located on the top right corner. Select `Watched Projects` and then add any projects that interest you.

Commit Messages

Gerrit follows the Git commit message format. Ensure the headers are at the bottom and don't contain blank lines between one another. The following example shows the format and content expected in a commit message:

Brief (no more than 50 chars) one line description.

Elaborate summary of the changes made referencing why (motivation), what was changed and how it was tested. Note also any changes to documentation made to remain consistent with the code changes, wrapping text at 72 chars/line.

Jira: FAB-100
Change-Id: LONGHEXHASH
Signed-off-by: Your Name your.email@example.org
AnotherExampleHeader: An Example of another Value

The Gerrit server provides a precommit hook to autogenerate the Change-Id which is one time use.

Recommended reading: How to Write a Git Commit Message

Avoid Pushing Untested Work to a Gerrit Server

To avoid pushing untested work to Gerrit.

Check your work at least three times before pushing your change to Gerrit. Be mindful of what information you are publishing.

Keeping Track of Changes

- Set Gerrit to send you emails:
- Gerrit will add you to the email distribution list for a change if a developer adds you as a reviewer, or if you comment on a specific Patch Set.
- Opening a change in Gerrit's review interface is a quick way to follow that change.
- Watch projects in the Gerrit projects section at `Gerrit` , select at least *New Changes, New Patch Sets, All Comments* and *Submitted Changes*.

Always track the projects you are working on; also see the feedback/comments mailing list to learn and help others ramp up.

Topic branches

Topic branches are temporary branches that you push to commit a set of logically-grouped dependent commits:

To push changes from `REMOTE/master` tree to Gerrit for being reviewed as a topic in **TopicName** use the following command as an example:

$ git push REMOTE HEAD:refs/for/master/TopicName

The topic will show up in the review `UI` and in the `Open Changes List` . Topic branches will disappear from the master tree when its content is merged.

Creating a Cover Letter for a Topic

You may decide whether or not you'd like the cover letter to appear in the history.

1. To make a cover letter that appears in the history, use this command:

```
git commit --allow-empty
```

Edit the commit message, this message then becomes the cover letter. The command used doesn't change any files in the source tree.

2. To make a cover letter that doesn't appear in the history follow these steps:

- Put the empty commit at the end of your commits list so it can be ignored without having to rebase.
- Now add your commits

```
git commit ...
git commit ...
git commit ...
```

- Finally, push the commits to a topic branch. The following command is an example:

```
git push REMOTE HEAD:refs/for/master/TopicName
```

If you already have commits but you want to set a cover letter, create an empty commit for the cover letter and move the commit so it becomes the last commit on the list. Use the following command as an example:

```
git rebase -i HEAD~#Commits
```

Be careful to uncomment the commit before moving it. `#Commits` is the sum of the commits plus your new cover letter.

Finding Available Topics

```
$ ssh -p 29418 gerrit.hyperledger.org gerrit query \ status:open project:fabric␣
↪branch:master \
| grep topic: | sort -u
```

- gerrit.hyperledger.org Is the current URL where the project is hosted.
- *status* Indicates the topic's current status: open , merged, abandoned, draft, merge conflict.
- *project* Refers to the current name of the project, in this case fabric.
- *branch* The topic is searched at this branch.
- *topic* The name of an specific topic, leave it blank to include them all.
- *sort* Sorts the found topics, in this case by update (-u).

Downloading or Checking Out a Change

In the review UI, on the top right corner, the **Download** link provides a list of commands and hyperlinks to checkout or download diffs or files.

We recommend the use of the *git review* plugin. The steps to install git review are beyond the scope of this document. Refer to the git review documentation for the installation process.

To check out a specific change using Git, the following command usually works:

```
git review -d CHANGEID
```

If you don't have Git-review installed, the following commands will do the same thing:

```
git fetch REMOTE refs/changes/NN/CHANGEIDNN/VERSION \ && git checkout FETCH_HEAD
```

For example, for the 4th version of change 2464, NN is the first two digits (24):

```
git fetch REMOTE refs/changes/24/2464/4 \ && git checkout FETCH_HEAD
```

Using Draft Branches

You can use draft branches to add specific reviewers before you publishing your change. The Draft Branches are pushed to `refs/drafts/master/TopicName`

The next command ensures a local branch is created:

```
git checkout -b BRANCHNAME
```

The next command pushes your change to the drafts branch under **TopicName**:

```
git push REMOTE HEAD:refs/drafts/master/TopicName
```

Using Sandbox Branches

You can create your own branches to develop features. The branches are pushed to the `refs/sandbox/USERNAME/BRANCHNAME` location.

These commands ensure the branch is created in Gerrit's server.

```
git checkout -b sandbox/USERNAME/BRANCHNAME
git push --set-upstream REMOTE HEAD:refs/heads/sandbox/USERNAME/BRANCHNAME
```

Usually, the process to create content is:

- develop the code,
- break the information into small commits,
- submit changes,
- apply feedback,
- rebase.

The next command pushes forcibly without review:

```
git push REMOTE sandbox/USERNAME/BRANCHNAME
```

You can also push forcibly with review:

```
git push REMOTE HEAD:ref/for/sandbox/USERNAME/BRANCHNAME
```

Updating the Version of a Change

During the review process, you might be asked to update your change. It is possible to submit multiple versions of the same change. Each version of the change is called a patch set.

Always maintain the **Change-Id** that was assigned. For example, there is a list of commits, **c0...c7**, which were submitted as a topic branch:

```
git log REMOTE/master..master

c0
...
c7

git push REMOTE HEAD:refs/for/master/SOMETOPIC
```

After you get reviewers' feedback, there are changes in **c3** and **c4** that must be fixed. If the fix requires rebasing, rebasing changes the commit Ids, see the rebasing section for more information. However, you must keep the same Change-Id and push the changes again:

```
git push REMOTE HEAD:refs/for/master/SOMETOPIC
```

This new push creates a patches revision, your local history is then cleared. However you can still access the history of your changes in Gerrit on the `review UI` section, for each change.

It is also permitted to add more commits when pushing new versions.

Rebasing

Rebasing is usually the last step before pushing changes to Gerrit; this allows you to make the necessary *Change-Ids*. The *Change-Ids* must be kept the same.

- **squash:** mixes two or more commits into a single one.
- **reword:** changes the commit message.
- **edit:** changes the commit content.
- **reorder:** allows you to interchange the order of the commits.
- **rebase:** stacks the commits on top of the master.

Rebasing During a Pull

Before pushing a rebase to your master, ensure that the history has a consecutive order.

For example, your `REMOTE/master` has the list of commits from **a0** to **a4**; Then, your changes **c0...c7** are on top of **a4**; thus:

```
git log --oneline REMOTE/master..master

a0
a1
a2
a3
a4
c0
c1
...
c7
```

If `REMOTE/master` receives commits **a5**, **a6** and **a7**. Pull with a rebase as follows:

```
git pull --rebase REMOTE master
```

This pulls **a5-a7** and re-apply **c0-c7** on top of them:

```
$ git log --oneline REMOTE/master..master
a0
...
a7
c0
c1
...
c7
```

Getting Better Logs from Git

Use these commands to change the configuration of Git in order to produce better logs:

```
git config log.abbrevCommit true
```

The command above sets the log to abbreviate the commits' hash.

```
git config log.abbrev 5
```

The command above sets the abbreviation length to the last 5 characters of the hash.

```
git config format.pretty oneline
```

The command above avoids the insertion of an unnecessary line before the Author line.

To make these configuration changes specifically for the current Git user, you must add the path option `--global` to `config` as follows:

Testing

Unit test

See *building Hyperledger Fabric* for unit testing instructions.

See Unit test coverage reports

To see coverage for a package and all sub-packages, execute the test with the `-cover` switch:

```
go test ./... -cover
```

To see exactly which lines are not covered for a package, generate an html report with source code annotated by coverage:

```
go test -coverprofile=coverage.out
go tool cover -html=coverage.out -o coverage.html
```

System test

[WIP] ...coming soon

This topic is intended to contain recommended test scenarios, as well as current performance numbers against a variety of configurations.

Coding guidelines

Coding Golang

We code in Go™ and strictly follow the best practices and will not accept any deviations. You must run the following tools against your Go code and fix all errors and warnings: - golint - go vet - goimports

Generating gRPC code

If you modify any `.proto` files, run the following command to generate/update the respective `.pb.go` files.

```
cd $GOPATH/src/github.com/hyperledger/fabric
make protos
```

Adding or updating Go packages

Hyperledger Fabric uses Govendor for package management. 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 Govendor, which is installed in the Vagrant environment. The following commands can be used for package management:

```
# Add external packages.
govendor add +external

# Add a specific package.
govendor add github.com/kardianos/osext

# Update vendor packages.
govendor update +vendor

# Revert back to normal GOPATH packages.
govendor remove +vendor

# List package.
govendor list
```

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.

Getting a Linux Foundation account

In order to participate in the development of the Hyperledger Fabric project, you will need a *Linux Foundation account*. You will need to use your LF ID to access to all the Hyperledger community development tools, including Gerrit, Jira and the Wiki (for editing, only).

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.

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 <https://wiki.hyperledger.org/security/bug-handling-process>*.

If it has not been previously reported, 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 Rocket Chat. For example, a doc bug should be broadcast to `#fabric-documentation`, a database bug to `#fabric-ledger`, and so on...

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 you can submit a change request (CR).

Note: If you need help with submitting your first CR, we have created a brief tutorial for you.

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.

Reviewing submitted Change Requests (CRs)

Another way to contribute and learn about Hyperledger Fabric is to help the maintainers with the review of the CRs that are open. Indeed maintainers have the difficult role of having to review all the CRs 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 CR 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 CRs faster and everybody will gain from it.

Just browse through the open CRs on Gerrit to get started.

Making Feature/Enhancement Proposals

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, Story or Improvement, 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 hyperledger-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 CRs will be merged.

Setting up development environment

Next, try *building the project* in your local development environment to ensure that everything is set up correctly.

What makes a good change 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 CR. 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 changess. 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 change requests, e.g. those with more than 300 LOC are more likely than not going to receive a -2, and you'll be asked to refactor the change to conform with this guidance.

- Do not stack change requests (e.g. submit a CR from the same local branch as your previous CR) unless they are related. This will minimize merge conflicts and allow changes to be merged more quickly. If you stack requests your subsequent requests may be held up because of review comments in the preceding requests.
- Write a meaningful commit message. Include a meaningful 50 (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. See the *complete requirements* for an acceptable change request.

Note: That Gerrit will automatically create a hyperlink to the JIRA item. e.g.

```
[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 change 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.

Communication

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.

Maintainers

The project's *maintainers* are responsible for reviewing and merging all patches submitted for review and they guide the over-all technical direction of the project within the guidelines established by the Hyperledger Technical Steering Committee (TSC).

Becoming a maintainer

This project is managed under an open governance model as described in our charter. Projects or sub-projects will be 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. 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.rst* 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.

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`.

Needs Review

CHAPTER 10

Glossary

Terminology is important, so that all Hyperledger Fabric users and developers agree on what we mean by each specific term. What is chaincode, 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!

Anchor Peer

A peer node on a channel that all other peers can discover and communicate with. Each *Member* on a channel has an anchor peer (or multiple anchor peers to prevent single point of failure), allowing for peers belonging to different Members to discover all existing peers on a channel.

Block

An ordered set of transactions that is cryptographically linked to the preceding block(s) on a channel.

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.

Chaincode

Chaincode is software, running on a ledger, to encode assets and the transaction instructions (business logic) for modifying the assets.

Channel

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 properly authenticated to a channel in order to interact with it. Channels are defined by a *Configuration-Block*.

Commitment

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.

Concurrency Control Version Check

Concurrency Control Version Check is a method of keeping state in sync across peers on a channel. Peers execute transactions in parallel, and before commitment to the ledger, peers check that the data read at execution time has not changed. If the data read for the transaction has changed between execution time and commitment 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.

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.

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.

Current State

The current state of the ledger represents the latest values for all keys ever included in its chain transaction log. Peers commit the latest values to ledger current state for each valid transaction included in a processed block. Since current state represents all latest key values known to the channel, it is sometimes referred to as World State. Chaincode executes transaction proposals against current state data.

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.

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.

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. A distinct endorsement policy for install and instantiate transactions is also required.

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.

Genesis Block

The configuration block that initializes a blockchain network or channel, and also serves as the first block on a chain.

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.

Initialize

A method to initialize a chaincode application.

Install

The process of placing a chaincode on a peer's file system.

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.

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.

Leading Peer

Each *Member* can own multiple peers on each channel that it subscribes to. One of these peers is serves as the leading peer for the channel, in order to communicate with the network ordering service on behalf of the member. The ordering service "delivers" blocks to the leading peer(s) on a channel, who then distribute them to other peers within the same member cluster.

Ledger

A ledger is a channel's chain and current state data which is maintained by each peer on the channel.

Member

A legally separate entity that owns a unique root certificate for the network. Network components such as peer nodes and application clients will be linked to a member.

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 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.

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.

Ordering Service

A defined collective of nodes that orders transactions into a block. The ordering service exists independent of the peer processes and orders transactions on a first-come-first-serve basis for all channel's on the network. The ordering service is designed to support pluggable implementations beyond the out-of-the-box SOLO and Kafka varieties. The ordering service is a common binding for the overall network; it contains the cryptographic identity material tied to each *Member*.

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.

Policy

There are policies for endorsement, validation, chaincode management and network/channel management.

Proposal

A request for endorsement that is aimed at specific peers on a channel. Each proposal is either an instantiate or an invoke (read/write) request.

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.

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 three more – Python, Go and REST – are not yet official but can still be downloaded and tested.

State Database

Current state data is stored in a state database for efficient reads and queries from chaincode. Supported databases include levelDB and couchDB.

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.

Transaction

Invoke or instantiate results that are submitted for ordering, validation, and commit. Invokes are requests to read/write data from the ledger. Instantiate is a request to start and initialize a chaincode on a channel. Application clients gather invoke or instantiate responses from endorsing peers and package the results and endorsements into a transaction that is submitted for ordering, validation, and commit.

CHAPTER 11

Release Notes

v1.1.1 - July 5, 2018

Bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Known Vulnerabilities

none

Resolved Vulnerabilities

https://jira.hyperledger.org/browse/FAB-10537 https://jira.hyperledger.org/browse/FAB-10577

Known Issues & Workarounds

The fabric-ccenv image which is used to build chaincode, currently includes the github.com/hyperledger/fabric/core/chaincode/shim ("shim") package. This is convenient, as it provides the ability to package chaincode without the need to include the "shim". However, this may cause issues in future releases (and/or when trying to use packages which are included by the "shim").

In order to avoid any issues, users are advised to manually vendor the "shim" package with their chaincode prior to using the peer CLI for packaging and/or for installing chaincode.

Please refer to https://jira.hyperledger.org/browse/FAB-5177 for more details, and kindly be aware that given the above, we may end up changing the fabric-ccenv in the future.

Change Log

v1.1.0 - March 15, 2018

The v1.1 release includes all of the features delivered in v1.1.0-preview and v1.1.0-alpha.

Additionally, there are feature improvements, bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Updated to Go version 1.9.2. Updated baseimage version to 0.4.6.

Known Vulnerabilities

none

Resolved Vulnerabilities

https://jira.hyperledger.org/browse/FAB-4824 https://jira.hyperledger.org/browse/FAB-5406

Known Issues & Workarounds

The fabric-ccenv image which is used to build chaincode, currently includes the github.com/hyperledger/fabric/core/chaincode/shim ("shim") package. This is convenient, as it provides the ability to package chaincode without the need to include the "shim". However, this may cause issues in future releases (and/or when trying to use packages which are included by the "shim").

In order to avoid any issues, users are advised to manually vendor the "shim" package with their chaincode prior to using the peer CLI for packaging and/or for installing chaincode.

Please refer to FAB-5177 for more details, and kindly be aware that given the above, we may end up changing the fabric-ccenv in the future.

Change Log

v1.1.0-rc1 - March 1, 2018

The v1.1 release candidate 1 (rc1) includes all of the features delivered in v1.1.0-preview and v1.1.0-alpha.

> Additionally, there are feature improvements, bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Known Vulnerabilities

none

Resolved Vulnerabilities

none

Known Issues & Workarounds

The fabric-ccenv image which is used to build chaincode, currently includes the github.com/hyperledger/fabric/core/chaincode/shim ("shim") package. This is convenient, as it provides the ability to package chaincode without the need to include the "shim". However, this may cause issues in future releases (and/or when trying to use packages which are included by the "shim").

In order to avoid any issues, users are advised to manually vendor the "shim" package with their chaincode prior to using the peer CLI for packaging and/or for installing chaincode.

Please refer to FAB-5177 for more details, and kindly be aware that given the above, we may end up changing the fabric-ccenv in the future.

Change Log

v1.1.0-alpha - January 25, 2018

This is a feature-complete *alpha* release of the up-coming 1.1 release. The 1.1 release includes the following new major features:

- FAB-6911 - Event service for blocks
- FAB-5481 - Event service for block transaction events
- FAB-5300 - Certificate Revocation List from CA
- FAB-3067 - Peer management of CouchDB indexes
- FAB-6715 - Mutual TLS between all components
- FAB-5556 - Rolling Upgrade via configured capabilities
- FAB-2331 - Node.js Chaincode support
- FAB-5363 - Node.js SDK Connection Profile
- FAB-830 - Encryption library for chaincode
- FAB-5346 - Attribute-based Access Control
- FAB-6089 - Chaincode APIs for creator identity
- FAB-6421 - Performance improvements

Additionally, there are feature improvements, bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Known Vulnerabilities

none

Resolved Vulnerabilities

none

Known Issues & Workarounds

The fabric-ccenv image which is used to build chaincode, currently includes the github.com/hyperledger/fabric/core/chaincode/shim ("shim") package. This is convenient, as it provides the ability to package chaincode without the need to include the "shim". However, this may cause issues in future releases (and/or when trying to use packages which are included by the "shim").

In order to avoid any issues, users are advised to manually vendor the "shim" package with their chaincode prior to using the peer CLI for packaging and/or for installing chaincode.

Please refer to FAB-5177 for more details, and kindly be aware that given the above, we may end up changing the fabric-ccenv in the future.

Change Log

v1.1.0-preview - November 1, 2017

This is a *preview* release of the up-coming 1.1 release. We are not feature complete for 1.1 just yet, but we wanted to get the following functionality published to gain some early community feedback on the following features:

- FAB-2331 - Node.js Chaincode
- FAB-5363 - Node.js SDK Connection Profile
- FAB-830 - Encryption library for chaincode
- FAB-5346 - Attribute-based Access Control
- FAB-6089 - Chaincode APIs to retrieve creator cert info
- FAB-6421 - Performance improvements

Additionally, there are the usual bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Known Vulnerabilities

none

Resolved Vulnerabilities

none

Known Issues & Workarounds

The fabric-ccenv image which is used to build chaincode, currently includes the github.com/hyperledger/fabric/core/chaincode/shim ("shim") package. This is convenient, as it provides the ability to package chaincode without the need to include the "shim". However, this may cause issues in future releases (and/or when trying to use packages which are included by the "shim").

In order to avoid any issues, users are advised to manually vendor the "shim" package with their chaincode prior to using the peer CLI for packaging and/or for installing chaincode.

Please refer to FAB-5177 for more details, and kindly be aware that given the above, we may end up changing the fabric-ccenv in the future.

Change Log

v1.0.4 - October 31, 2017

Bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Known Vulnerabilities

none

Resolved Vulnerabilities

none

Known Issues & Workarounds

The fabric-ccenv image which is used to build chaincode, currently includes the github.com/hyperledger/fabric/core/chaincode/shim ("shim") package. This is convenient, as it provides the ability to package chaincode without the need to include the "shim". However, this may cause issues in future releases (and/or when trying to use packages which are included by the "shim").

In order to avoid any issues, users are advised to manually vendor the "shim" package with their chaincode prior to using the peer CLI for packaging and/or for installing chaincode.

Please refer to https://jira.hyperledger.org/browse/FAB-5177 for more details, and kindly be aware that given the above, we may end up changing the fabric-ccenv in the future.

Change Log

v1.0.3 - October 3, 2017

Bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Known Vulnerabilities none

Resolved Vulnerabilities none

Known Issues & Workarounds The fabric-ccenv image which is used to build chaincode, currently includes the github.com/hyperledger/fabric/core/chaincode/shim ("shim") package. This is convenient, as it provides the ability to package chaincode without the need to include the "shim". However, this may cause issues in future releases (and/or when trying to use packages which are included by the "shim").

In order to avoid any issues, users are advised to manually vendor the "shim" package with their chaincode prior to using the peer CLI for packaging and/or for installing chaincode.

Please refer to https://jira.hyperledger.org/browse/FAB-5177 for more details, and kindly be aware that given the above, we may end up changing the fabric-ccenv in the future.

Change Log

v1.0.2 - August 31, 2017

Bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Known Vulnerabilities none

Resolved Vulnerabilities https://jira.hyperledger.org/browse/FAB-5753 https://jira.hyperledger.org/browse/FAB-5899

Known Issues & Workarounds The fabric-ccenv image which is used to build chaincode, currently includes the github.com/hyperledger/fabric/core/chaincode/shim ("shim") package. This is convenient, as it provides the ability to package chaincode without the need to include the "shim". However, this may cause issues in future releases (and/or when trying to use packages which are included by the "shim").

In order to avoid any issues, users are advised to manually vendor the "shim" package with their chaincode prior to using the peer CLI for packaging and/or for installing chaincode.

Please refer to https://jira.hyperledger.org/browse/FAB-5177 for more details, and kindly be aware that given the above, we may end up changing the fabric-ccenv in the future.

Change Log

v1.0.1 - August 5, 2017

Bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Known Vulnerabilities none

Resolved Vulnerabilities https://jira.hyperledger.org/browse/FAB-5329 https://jira.hyperledger.org/browse/FAB-5330 https://jira.hyperledger.org/browse/FAB-5353 https://jira.hyperledger.org/browse/FAB-5529 https://jira.hyperledger.org/browse/FAB-5606 https://jira.hyperledger.org/browse/FAB-5627

Known Issues & Workarounds The fabric-ccenv image which is used to build chaincode, currently includes the github.com/hyperledger/fabric/core/chaincode/shim ("shim") package. This is convenient, as it provides the ability to package chaincode without the need to include the "shim". However, this may cause issues in future releases (and/or when trying to use packages which are included by the "shim").

In order to avoid any issues, users are advised to manually vendor the "shim" package with their chaincode prior to using the peer CLI for packaging and/or for installing chaincode.

Please refer to https://jira.hyperledger.org/browse/FAB-5177 for more details, and kindly be aware that given the above, we may end up changing the fabric-ccenv in the future.

Change Log

v1.0.0 - July 11, 2017

Bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (removal of unused code, static security scanning, spelling, linting and more).

Known Vulnerabilities none

Resolved Vulnerabilities https://jira.hyperledger.org/browse/FAB-5207

Known Issues & Workarounds The fabric-ccenv image which is used to build chaincode, currently includes the github.com/hyperledger/fabric/core/chaincode/shim ("shim") package. This is convenient, as it provides the ability to package chaincode without the need to include the "shim". However, this may cause issues in future releases (and/or when trying to use packages which are included by the "shim").

In order to avoid any issues, users are advised to manually vendor the "shim" package with their chaincode prior to using the peer CLI for packaging and/or for installing chaincode.

Please refer to https://jira.hyperledger.org/browse/FAB-5177 for more details, and kindly be aware that given the above, we may end up changing the fabric-ccenv in the future.

Change Log

v1.0.0-rc1 - June 23, 2017

Bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Known Vulnerabilities none

Resolved Vulnerabilities https://jira.hyperledger.org/browse/FAB-4856 https://jira.hyperledger.org/browse/FAB-4848 https://jira.hyperledger.org/browse/FAB-4751 https://jira.hyperledger.org/browse/FAB-4626 https://jira.hyperledger.org/browse/FAB-4567 https://jira.hyperledger.org/browse/FAB-3715

Known Issues & Workarounds none

Change Log

v1.0.0-beta - June 8, 2017

Bug fixes, documentation and test coverage improvements, UX improvements based on user feedback and changes to address a variety of static scan findings (unused code, static security scanning, spelling, linting and more).

Upgraded to latest version (a precursor to 1.4.0) of gRPC-go and implemented keep-alive feature for improved resiliency.

Added a new tool *configtxlator* to enable users to translate the contents of a channel configuration transaction into a human readable form.

Known Vulnerabilities

none

Resolved Vulnerabilities

none

Known Issues & Workarounds

BCCSP content in the configtx.yaml has been removed. This change will cause a panic when running *configtxgen* tool with a configtx.yaml file that includes the removed BCCSP content.

Java Chaincode support has been disabled until post 1.0.0 as it is not yet fully mature. It may be re-enabled for experimentation by cloning the hyperledger/fabric repository, reversing this commit and building your own fork.

Change Log

v1.0.0-alpha2

The second alpha release of the v1.0.0 Hyperledger Fabric. The code is now feature complete. From now until the v1.0.0 release, the community is focused on documentation improvements, testing, hardening, bug fixing and tooling. We will be releasing successive release candidates periodically as the release firms up.

Change Log

v1.0.0-alpha - March 16, 2017

The first alpha release of the v1.0.0 Hyperledger Fabric. The code is being made available to developers to begin exploring the v1.0 architecture.

Change Log

v0.6-preview September 16, 2016

A developer preview release of the Hyperledger Fabric intended to exercise the release logistics and stabilize a set of capabilities for developers to try out. This will be the last release under the original architecture. All subsequent releases will deliver on the v1.0 architecture.

Change Log

v0.5-developer-preview - June 17, 2016

A developer preview release of the Hyperledger Fabric intended to exercise the release logistics and stabilize a set of capabilities for developers to try out.

Key features:

Permissioned blockchain with immediate finality Chaincode (aka smart contract) execution environments Docker container (user chaincode) In-process with peer (system chaincode) Pluggable consensus with PBFT, NOOPS (development mode), SIEVE (prototype) Event framework supports pre-defined and custom events Client SDK (Node.js), basic REST APIs and CLIs Known Key Bugs and work in progress

- 1895 - Client SDK interfaces may crash if wrong parameter specified
- 1901 - Slow response after a few hours of stress testing
- 1911 - Missing peer event listener on the client SDK
- 889 - The attributes in the TCert are not encrypted. This work is still on-going

CHAPTER 12

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 (hyperledger-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 13

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.