hyperledger-fabricdocs Documentation

Release master

hyperledger

Mar 16, 2017

Hyperledger Fabric is a social innovation that is about to free innovators in startups, enterprises and government to transform and radically reduce the cost of working together across organizations. By the end of this section, you should have the essential understanding of Fabric you need to start *knitting* together a great business network.

Fabric is a network of networks, like the Internet itself. An application can use one or more networks, each managing different *Assets*, Agreements and Transactions between different sets of *Member* nodes. In Fabric, the Ordering Service is the foundation of each network. The founder of a network selects an Ordering Service (or creates a new one) and passes in a config file with the rules (usually called Policies) that govern it. Examples of these rules include setting/defining which Members can join the network, how Members can be added or removed, and configuration details like block size. While it is possible for one company to set and control these rules as a "dictator," typically these rules will also include policies that make changing the rules a matter of consensus among the members of the network. Fabric also requires some level of "endorsement" in order to transact. Check out the power and intricacy of *Endorsement policies* , which are used across the Fabric landscape - from a consortium's network configuration to a simple read operation.

We mentioned that the Ordering Service (OS) is the foundation of the network, and you're probably thinking, "It must do something beyond just ordering." Well you're right! All members and entities in the network will be tied to a higher level certificate authority, and this authority is defined within the configuration of the Ordering Service. As a result, the OS can verify and authenticate transactions arriving from any corner of the network. The OS plays a central and critical role in the functionality and integrity of the network, and skeptics might fear too much centralization of power and responsibility. After all, that's a principal feature of shared ledger technology - to decentralize the control and provide a foundation of trust with entities who you CAN'T wholeheartedly trust. Well let's assuage that fear. The OS is agnostic to transaction details; it simply orders on a first-come-first-serve basis and returns blocks to their corresponding channels. Perhaps more importantly though, control of the ordering service can be shared and co-administered by the participating members in the network. OR, if even that solution is untenable, then the OS can be hosted and maintained by a trusted third-party. Fabric is built upon a modular and pluggable architecture, so the only real decision for business networks is how to configure an OS to meet their requirements.

(This notion of the OS as a pluggable component also opens the door to exciting opportunities for innovative teams and individuals. Currently there are only a few OS orchestrations - Solo and Kafka. However, other options such as Intel's PoET or certain BFT flavors could be powerful supplementaries to Fabric, and help solve challenging use cases.)

To participate in the Network, each Organization maintains a runtime called a *Peer*, which will allow an application to participate in transactions, interact with the Ordering Service, and maintain a set of ledgers. Notice we said a set of ledgers. One of Fabric's key innovations is the ability to run multiple *Channel* s on each network. This is how a network can conduct both highly confidential bilateral transactions and multilateral, or even public, transactions in the same solution without everyone having a copy of every transaction or run the code in every agreement.

Watch how Fabric is Building a Blockchain for Business .

If you're still reading, you clearly have some knowledge and an interest in distributed ledger technology, AND you probably think a key piece is missing. Where is consensus in all of this? Well, it's embedded in the entire life cycle of a transaction. Transactions come into the network, and the submitting client's identity is verified and consented upon. Transactions then get executed and endorsed, and these endorsements are consented upon. Transactions get ordered, and the validity of this order is consented upon. Finally, transactions get committed to a shared ledger, and each transaction's subsequent impact on the state of the involved asset(s) is consented upon. Consensus isn't pigeonholed into one module or one function. It lives and exists throughout the entire DNA of Fabric. Fabric is built with security at the forefront, not as an afterthought. Members and participating entities operate with known identities, and no action on the network circumvents the sign/verify/authenticate mandate. Requirements such as security, privacy and confidentiality are paramount in some manner to nearly all business dealings, and they, like consensus, are stitched into the very essence of Fabric.

So what problem do you want to solve? What assets are at stake? Who are the players? What levels of security and encryption do you need? Fabric is designed to provide an answer and solution to this challenging collective of questions and beyond. Just like fabric - in the literal sense of the word - is used in everything from airplane seats to bespoke suits, solutions built on Hyperledger Fabric can range from diamond provenance to equities trading. Explore

the documentation and see how you can leverage Fabric to craft a PoC for your own business network.

Note: This build of the docs is from the "master" branch

CHAPTER 1

What's Offered?

The getting started example uses Docker images to generate the Fabric network components. The scenario includes a consortium of three members, each managing and maintaing a peer node, as well as a "SOLO" *Ordering Service* and a Certificate Authority (CA). The cryptographic identity material, based on standard PKI implementation, has been pre-generated and is used for signing + verification on both the server (peer + ordering service) and client (SDK) sides. The CA is the network entity responsible for issuing and maintaing this identity material, which is necessary for authentication by all components and participants on the network. This sample uses a single CA. However, in enterprise scenarios each *Member* would likely have their own CA, with more complex security/identity measures implemented - e.g. cross-signing certificates, etc.

The members will transact on a private channel, with a shared ledger maintained by each peer node. Requests to read and write data to/from the ledger are sent as "proposals" to the peers. These proposals are in fact a request for endorsement from the peer, which will execute the transaction and return a response to the submitting client.

The sample demonstrates two methods for interacting with the network - a programmatical approach exercising the Node.js SDK APIs and a CLI requiring manual commands.

It's recommended to follow the sample in the order laid forth - application first, followed by the optional CLI route.

CHAPTER 2

Getting Setup

- Go - most recent version
- Docker - v1.13 or higher
- Docker Compose - v1.8 or higher
- Node.js & npm - node v6.9.5 and npm v3.10.10
- xcode - only required for OS X users
- nvm - if you want to use `nvm install` command If you already have node on your machine, use the node website to install v6.9.5 or issue the following command in your terminal:

```
nvm install v6.9.5
```

then execute the following to see your versions:

```
# should be 6.9.5
node -v
```

AND

```
# should be 3.10.10
npm -v
```

Curl the source code to create network entities

- Download the cURL tool if not already installed.
- Determine a location on your local machine where you want to place the Fabric artifacts and application code.

```
mkdir -p <my_dev_workspace>/hackfest
cd <my_dev_workspace>/hackfest
```

Next, execute the following command:

```
curl -L https://raw.githubusercontent.com/hyperledger/fabric/master/examples/
↪sfhackfest/sfhackfest.tar.gz -o sfhackfest.tar.gz 2> /dev/null;  tar -xvf␣
↪sfhackfest.tar.gz
```

This command pulls and extracts all of the necessary artifacts to set up your network - Docker Compose script, channel generate/join script, crypto material for identity attestation, etc. In the `/src/github.com/example_cc` directory you will find the chaincode that will be deployed.

Your directory should contain the following:

```
JDoe-mbp: JohnDoe$ pwd
/Users/JohnDoe
JDoe-mbp: JohnDoe$ ls
sfhackfest.tar.gz   channel_test.sh   src
ccenv     docker-compose-gettingstarted.yml  tmp
```

Using Docker

You do not need to manually pull any images. The images for - `fabric-peer` , `fabric-orderer` , `fabric-ca` , and `cli` are specified in the .yml file and will automatically download, extract, and run when you execute the `docker-compose` command.

Commands

The channel commands are:

- `create` - create and name a channel in the `orderer` and get back a genesis block for the channel. The genesis block is named in accordance with the channel name.
- `join` - use the genesis block from the `create` command to issue a join request to a peer.

Use Docker to spawn network

Ensure the hyperledger/fabric-ccenv image is tagged as latest:

```
docker-compose -f docker-compose-gettingstarted.yml build
```

Create network entities, create channel, join peers to channel:

```
docker-compose -f docker-compose-gettingstarted.yml up -d
```

Behind the scenes this started six containers (3 peers, a "solo" orderer, cli and CA) in detached mode. A script - `channel_test.sh` - embedded within the `docker-compose-gettingstarted.yml` issued the create channel and join channel commands within the CLI container. In the end, you are left with a network and a channel containing three peers - peer0, peer1, peer2.

View your containers:

```
# if you have no other containers running, you will see six
docker ps
```

Ensure the channel has been created and peers have successfully joined:

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

You should see the following in your terminal:

```
/opt/gopath/src/github.com/hyperledger/fabric/peer #
```

To view results for channel creation/join:

```
more results.txt
```

You're looking for:

```
SUCCESSFUL CHANNEL CREATION
SUCCESSFUL JOIN CHANNEL on PEER0
SUCCESSFUL JOIN CHANNEL on PEER1
SUCCESSFUL JOIN CHANNEL on PEER2
```

To view genesis block:

```
more myc1.block
```

Exit the cli container:

```
exit
```

Curl the application source code and SDK modules

- Prior to issuing the command, make sure you are in the same working directory where you curled the network code. AND make sure you have exited the cli container.
- Execute the following command:

```
curl -OOOOOO https://raw.githubusercontent.com/hyperledger/fabric-sdk-node/v1.0-
↪alpha/examples/balance-transfer/{config.json,deploy.js,helper.js,invoke.
↪js,query.js,package.json}
```

This command pulls the javascript code for issuing your deploy, invoke and query calls. It also retrieves dependencies for the node SDK modules.

- Install the node modules:

```
# You may be prompted for your root password at one or more times during this␣
↪process.
npm install
```

You now have all of the necessary prerequisites and Fabric artifacts.

Asset Transfer with SDK

The individual javascript programs will exercise the SDK APIs to register and enroll the client with the provisioned Certificate Authority. Once the client is properly authenticated, the programs will demonstrate basic chaincode functionalities - deploy, invoke, and query. Make sure you are in the working directory where you pulled the source code before proceeding.

Upon success of each node program, you will receive a "200" response in the terminal.

Register/enroll & deploy chaincode (Linux or OSX):

```
# Deploy initializes key value pairs of "a","100" & "b","200".
GOPATH=$PWD node deploy.js
```

Register/enroll & deploy chaincode (Windows):

```
# Deploy initializes key value pairs of "a","100" & "b","200".
SET GOPATH=%cd%
node deploy.js
```

Issue an invoke. Move units 100 from "a" to "b":

```
node invoke.js
```

Query against key value "b":

```
# this should return a value of 300
node query.js
```

Explore the various node.js programs, along with `example_cc.go` to better understand the SDK and APIs.

Asset transfer with CLI

Use the cli container to manually exercise the create channel and join channel APIs.

Channel - `myc1` already exists, so let's create a new channel named `myc2` .

Exec into the cli container:

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

If successful, you should see the following in your terminal:

```
/opt/gopath/src/github.com/hyperledger/fabric/peer #
```

Send createChannel API to Ordering Service:

```
CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050 peer channel create -c myc2
```

This will return a genesis block - `myc2.block` - that you can issue join commands with. Next, send a joinChannel API to `peer0` and pass in the genesis block as an argument. The channel is defined within the genesis block:

```
CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050 CORE_PEER_ADDRESS=peer0:7051 peer␣
↪channel join -b myc2.block
```

To join the other peers to the channel, simply reissue the above command with `peer1` or `peer2` specified. For example:

```
CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050 CORE_PEER_ADDRESS=peer1:7051 peer␣
↪channel join -b myc2.block
```

Once the peers have all joined the channel, you are able to issues queries against any peer without having to deploy chaincode to each of them.

Deploy, invoke and query

Run the deploy command. This command is deploying a chaincode named `mycc` to `peer0` on the Channel ID `myc2` . The constructor message is initializing `a` and `b` with values of 100 and 200 respectively.

```
CORE_PEER_ADDRESS=peer0:7051 CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050 peer␣
↪chaincode deploy -C myc2 -n mycc -p github.com/hyperledger/fabric/examples -c '{
↪"Args":["init","a","100","b","200"]}'
```

Run the invoke command. This invocation is moving 10 units from `a` to `b` .

```
CORE_PEER_ADDRESS=peer0:7051 CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050 peer␣
↪chaincode invoke -C myc2 -n mycc -c '{"function":"invoke","Args":["move","a","b","10
↪"]}'
```

Run the query command. The invocation transferred 10 units from `a` to `b` , therefore a query against `a` should return the value 90.

```
CORE_PEER_ADDRESS=peer0:7051 CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050 peer␣
↪chaincode query -C myc2 -n mycc -c '{"function":"invoke","Args":["query","a"]}'
```

You can issue an `exit` command at any time to exit the cli container.

Create the initial channel

If you want to manually create the initial channel through the cli container, you will need to edit the Docker Compose file. Use an editor to open `docker-compose-gettingstarted.yml` and comment out the `channel_test.sh` command in your cli image. Simply place a `#` to the left of the command. (Recall that this script is executing the create and join channel APIs when you run `docker-compose up`) For example:

```
cli:
  container_name: cli
  <CONTENT REMOVED FOR BREVITY>
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
#  command: sh -c './channel_test.sh; sleep 1000'
#  command: /bin/sh
```

Then use the cli commands from above.

Troubleshooting

If you have existing containers running, you may receive an error indicating that a port is already occupied. If this occurs, you will need to kill the container that is using said port.

If a file cannot be located, make sure your curl commands executed successfully and make sure you are in the directory where you pulled the source code.

If you are receiving timeout or GRPC communication errors, make sure you have the correct version of Docker installed - v1.13.0. Then try restarting your failing docker process. For example:

```
docker stop peer0
```

Then:

```
docker start peer0
```

Another approach to GRPC and DNS errors (peer failing to resolve with orderer and vice versa) is to hardcode the IP addresses for each. You will know if there is a DNS issue, because a `more results.txt` command within the cli container will display something similar to:

```
ERROR CREATING CHANNEL
PEER0 ERROR JOINING CHANNEL
```

Issue a `docker inspect <container_name>` to ascertain the IP address. For example:

```
docker inspect peer0 | grep IPAddress
```

AND

```
docker inspect orderer | grep IPAddress
```

Take these values and hard code them into your cli commands. For example:

```
CORE_PEER_COMMITTER_LEDGER_ORDERER=172.21.0.2:7050 peer channel create -c myc1
```

AND THEN

```
CORE_PEER_COMMITTER_LEDGER_ORDERER=<IP_ADDRESS> CORE_PEER_ADDRESS=<IP_ADDRESS> peer␣
↪channel join -b myc1.block
```

If you are seeing errors while using the node SDK, make sure you have the correct versions of node.js and npm installed on your machine. You want node v6.9.5 and npm v3.10.10.

If you ran through the automated channel create/join process (i.e. did not comment out `channel_test.sh` in the `docker-compose-gettingstarted.yml`), then channel - `myc1` - and genesis block - `myc1.block` - have already been created and exist on your machine. As a result, if you proceed to execute the manual steps in your cli container:

```
CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050 peer channel create -c myc1
```

Then you will run into an error similar to:

```
<EXACT_TIMESTAMP>       UTC [msp] Sign -> DEBU 064 Sign: digest:␣
↪5ABA6805B3CDBAF16C6D0DCD6DC439F92793D55C82DB130206E35791BCF18E5F
Error: Got unexpected status: BAD_REQUEST
Usage:
  peer channel create [flags]
```

This occurs because you are attempting to create a channel named `myc1` , and this channel already exists! There are two options. Try issuing the peer channel create command with a different channel name - `myc2` . For example:

```
CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050 peer channel create -c myc2
```

Then join:

```
CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050 CORE_PEER_ADDRESS=peer0:7051 peer␣
↪channel join -b myc2.block
```

If you do choose to create a new channel, and want to run deploy/invoke/query with the node.js programs, you also need to edit the "channelID" parameter in the `config.json` file to match the new channel's name. For example:

```
{
   "chainName":"fabric-client1",
   "chaincodeID":"mycc",
   "channelID":"myc2",
   "goPath":"../../test/fixtures",
   "chaincodePath":"github.com/example_cc",
```

OR, if you want your channel called - `myc1` -, remove your docker containers and then follow the same commands in the **Manually create and join peers to a new channel** topic.

Clean up

Shut down your containers:

```
docker-compose -f docker-compose-gettingstarted.yml down
```

Helpful Docker tips

Remove a specific docker container:

```
docker rm <containerID>
```

Force removal:

```
docker rm -f <containerID>
```

Remove all docker containers:

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

This will merely kill docker containers (i.e. stop the process). You will not lose any images.

Remove an image:

```
docker rmi <imageID>
```

Forcibly remove:

```
docker rmi -f <imageID>
```

Remove all images:

```
docker rmi -f $(docker images -q)
```

CHAPTER 3

Overview

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 accomodate the complexity and intricacies that exist across the economic ecosystem.

This section provides a high-level overview of key Fabric concepts:

- *Assets*
- *Chaincode*
- *Ledger Features*
- *Privacy through Channels*
- *Security & Membership Services*
- *Consensus*

Needs Review

CHAPTER 4

The Fabric Model

Assets

Assets can range from the tangible (real estate and hardware) to the intangible (contracts and intellectual property). You can easily define Assets in client-side javascript and use them in your Fabric application using the included Fabric Composer tool.

Fabric supports the ability to exchange assets using unspent transaction outputs as the inputs for subsequent transactions. Assets (and asset registries) live in Fabric as a collection of key-value pairs, with state changes recorded as transactions on a *Channel* ledger. Fabric allows for any asset to be represented in binary or JSON format.

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 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 :ref:'MSP's allowing 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

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. Fabric even solves scenarios that want to bridge the gap between total transparency and privacy. Chaincode gets 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 SHA0-256, etc. before appending to the ledger.

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 Fabric, coupled with the existence and capabilities of channels, helps address scenarios where privacy and confidentiality are paramount concerns.

See the Fabric CA section to better understand cryptographic implementations, and the sign, verify, authenticate approach used in 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.

CHAPTER 5

Use Cases

B2B Contract

Business contracts can be codified to allow two or more parties to automate contractual agreements in a trusted way. Although information on blockchain is naturally "public", B2B contracts may require privacy control to protect sensitive business information from being disclosed to outside parties that also have access to the ledger.

While confidential agreements are a key business case, there are many scenarios where contracts can and should be easily discoverable by all parties on a ledger. For example, a ledger used to create offers (asks) seeking bids, by definition, requires access without restriction. This type of contract may need to be standardized so that bidders can easily find them, effectively creating an electronic trading platform with smart contracts (aka chaincode).

Persona

- Contract participant – Contract counter parties
- Third party participant – A third party stakeholder guaranteeing the integrity of the contract.

Key Components

- Multi-sig contract activation - When a contract is first deployed by one of the counter parties, it will be in the pending activation state. To activate a contract, signatures from other counterparties and/or third party participants are required.
- Multi-sig contract execution - Some contracts will require one of many signatures to execute. For example, in trade finance, a payment instruction can only be executed if either the recipient or an authorized third party (e.g. UPS) confirms the shipment of the good.
- Discoverability - If a contract is a business offer seeking bids, it must be easily searchable. In addition, such contracts must have the built-in intelligence to evaluate, select and honor bids.
- Atomicity of contract execution - Atomicity of the contract is needed to guarantee that asset transfers can only occur when payment is received (Delivery vs. Payment). If any step in the execution process fails, the entire transaction must be rolled back.
- Contract to chain-code communication - Contracts must be able to communicate with chaincodes that are deployed on the same ledger.
- Longer Duration contract - Timer is required to support B2B contracts that have long execution windows.
- Reuseable contracts - Often-used contracts can be standardized for reuse.

- Auditable contractual agreement - Any contract can be made auditable to third parties.
- Contract life-cycle management - B2B contracts are unique and cannot always be standardized. An efficient contract management system is needed to enhance the scalability of the ledger network.
- Validation access – Only nodes with validation rights are allowed to validate transactions of a B2B contract.
- View access – B2B contracts may include confidential information, so only accounts with predefined access rights are allowed to view and interrogate them.

Manufacturing Supply Chain

Final assemblers, such as automobile manufacturers, can create a supply chain network managed by its peers and suppliers so that a final assembler can better manage its suppliers and be more responsive to events that would require vehicle recalls (possibly triggered by faulty parts provided by a supplier). The blockchain fabric must provide a standard protocol to allow every participant on a supply chain network to input and track numbered parts that are produced and used on a specific vehicle.

Why is this specific example an abstract use case? Because while all blockchain cases store immutable information, and some add the need for transfer of assets between parties, this case emphasizes the need to provide deep searchability backwards through as many as 5-10 transaction layers. This backwards search capability is the core of establishing provenance of any manufactured good that is made up of other component goods and supplies.

Persona

- Final Assembler – The business entity that performs the final assembly of a product.
- Part supplier – Supplier of parts. Suppliers can also be assemblers by assembling parts that they receive from their sub-suppliers, and then sending their finished product to the final (root) assembler.

Key Components

- Payment upon delivery of goods - Integration with off-chain payment systems is required, so that payment instructions can be sent when parts are received.
- Third party Audit - All supplied parts must be auditable by third parties. For example, regulators might need to track the total number of parts supplied by a specific supplier, for tax accounting purposes.
- Obfuscation of shipments - Balances must be obfuscated so that no supplier can deduce the business activities of any other supplier.
- Obfuscation of market size - Total balances must be obfuscated so that part suppliers cannot deduce their own market share to use as leverage when negotiating contractual terms.
- Validation Access – Only nodes with validation rights are allowed to validate transactions (shipment of parts).
- View access – Only accounts with view access rights are allowed to interrogate balances of shipped parts and available parts.

Asset Depository

Assets such as financial securities must be able to be dematerialized on a blockchain network so that all stakeholders of an asset type will have direct access to that asset, allowing them to initiate trades and acquire information on an asset without going through layers of intermediaries. Trades should be settled in near real time and all stakeholders

must be able to access asset information in near real time. A stakeholder should be able to add business rules on any given asset type, as one example of using automation logic to further reduce operating costs.

Persona

- Investor – Beneficial and legal owner of an asset.
- Issuer – Business entity that issued the asset which is now dematerialized on the ledger network.
- Custodian – Hired by investors to manage their assets, and offer other value-add services on top of the assets being managed.
- Securities Depository – Depository of dematerialized assets.

Key Components

- Asset to cash - Integration with off-chain payment systems is necessary so that issuers can make payments to and receive payments from investors.
- Reference Rate - Some types of assets (such as floating rate notes) may have attributes linked to external data (such as reference rate), and such information must be fed into the ledger network.
- Asset Timer - Many types of financial assets have predefined life spans and are required to make periodic payments to their owners, so a timer is required to automate the operation management of these assets.
- Asset Auditor - Asset transactions must be made auditable to third parties. For example, regulators may want to audit transactions and movements of assets to measure market risks.
- Obfuscation of account balances - Individual account balances must be obfuscated so that no one can deduce the exact amount that an investor owns.
- Validation Access – Only nodes with validation rights are allowed to validate transactions that update the balances of an asset type (this could be restricted to CSD and/or the issuer).
- View access – Only accounts with view access rights are allowed to interrogate the chaincode that defines an asset type. If an asset represents shares of publicly traded companies, then the view access right must be granted to every entity on the network.

One Trade, One Contract

From the time that a trade is captured by the front office until the trade is finally settled, only one contract that specifies the trade will be created and used by all participants. The middle office will enrich the same electronic contract submitted by the front office, and that same contract will then be used by counter parties to confirm and affirm the trade. Finally, securities depository will settle the trade by executing the trading instructions specified on the contract. When dealing with bulk trades, the original contract can be broken down into sub-contracts that are always linked to the original parent contract.

Direct Communication

Company A announces its intention to raise 2 Billion USD by way of rights issue. Because this is a voluntary action, Company A needs to ensure that complete details of the offer are sent to shareholders in real time, regardless of how many intermediaries are involved in the process (such as receiving/paying agents, CSD, ICSD, local/global custodian banks, asset management firms, etc). Once a shareholder has made a decision, that decision will also be processed and

settled (including the new issuance of shares) in real time. If a shareholder sold its rights to a third party, the securities depository must be able to record the new shares under the name of their new rightful owner.

Separation of Asset Ownership and Custodian's Duties

Assets should always be owned by their actual owners, and asset owners must be able to allow third-party professionals to manage their assets without having to pass legal ownership of assets to third parties (such as nominee or street name entities). If issuers need to send messages or payments to asset owners (for example, listed share holders), issuers send them directly to asset owners. Third-party asset managers and/or custodians can always buy, sell, and lend assets on behalf of their owners. Under this arrangement, asset custodians can focus on providing value-add services to shareowners, without worrying about asset ownership duties such as managing and redirecting payments from issuers to shareowners.

Interoperability of Assets

If an organization requires 20,000 units of asset B, but instead owns 10,000 units of asset A, it needs a way to exchange asset A for asset B. Though the current market might not offer enough liquidity to fulfill this trade quickly, there might be plenty of liquidity available between asset A and asset C, and also between asset C and asset B. Instead of settling for market limits on direct trading (A for B) in this case, a chain network connects buyers with "buried" sellers, finds the best match (which could be buried under several layers of assets), and executes the transaction.

CHAPTER 6

Demos

Art Auction

[WIP] ...coming soon

Shows the provenance, attestation, and ownership of a piece of artwork and the ensuing interaction of the various stakeholders. Not yet stable with v1 codebase.

Learn more about the components here

Learn more about the client-side application here

Marbles

[WIP] ...coming soon

The marbles chaincode application demonstrates the ability to create assets (marbles) with unique attributes - size, color, owner, etc... and trade these assets with fellow participants in a blockchain network. It is not yet stable with v1 codebase.

Learn more about the marbles chaincode and client-side application here

CHAPTER 7

Docker Compose

[WIP] ...coming soon

This section will explain how to use Docker Compose to stand up the necessary components for a blockchain network. The various environment variables correlated to each image will be explained, and different configurations will be outlined.

CHAPTER 8

Learn to write chaincode

[WIP] ...coming soon

Teaches a developer how to write chaincode functions and implement the necessary interfaces to create generic assets.

In the meantime, visit the learn chaincode repo here to familiarize yourself with high level concepts and go code.

CHAPTER 9

Videos

Refer to the Hyperledger Fabric libary on youtube. The collection contains developers demonstrating various v1 features and components such as: ledger, channels, gossip, SDK, chaincode, MSP, and more...

CHAPTER 10

Best Practices

Coming soon...

Intended to contain best practices and configurations for MSP, networks, ordering service, channels, ACL, stress, policies, chaincode development, functions, etc...

CHAPTER 11

Starting a network

[WIP] ...coming soon

Intended to contain the recommended steps for generating prerequisite cryptographic material and then bootstrapping an ordering service (i.e. overall network) with participating organizations, ordering node certificates, load balancing, configuration, policies, etc...

CHAPTER 12

Administration and operations

[WIP] ...coming soon

CHAPTER 13

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)
    ...
}
```

CHAPTER 14

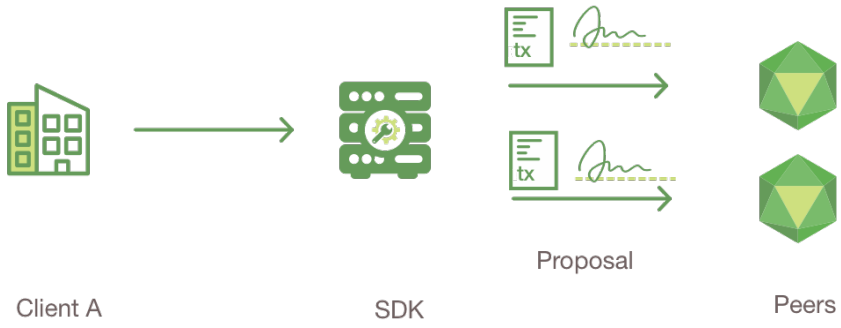
Transaction Flow

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.

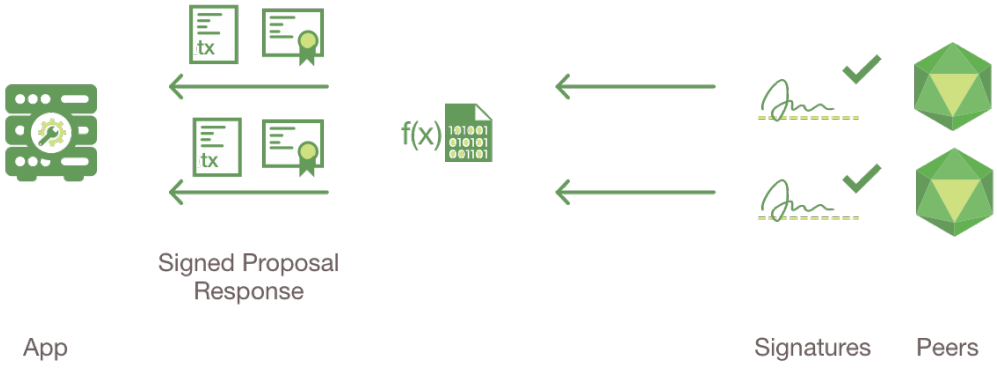


1. **Client A initiates a transaction**

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.



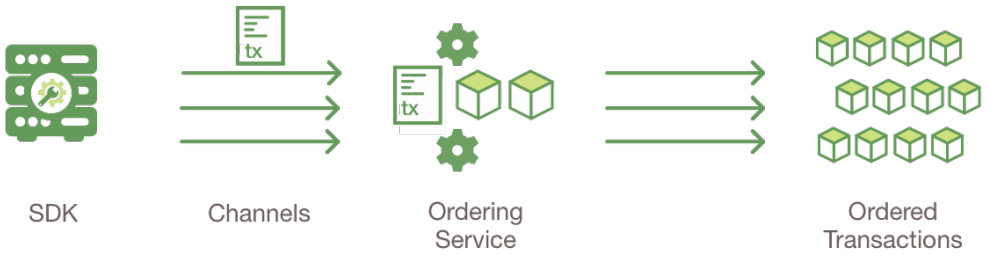
2. **Endorsing peers verify signature & execute the transaction**

The endorsing peers verify the signature (using MSP) and determine if the submitter is properly authorized to perform the proposed operation (using the channel's ACL). The endorsing peers take the transaction proposal arguments as inputs and execute them 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 and a YES/NO endorsement statement is passed back as a "proposal response" to the SDK which parses the payload for the application to consume.

{The MSP is a local process running on the peers which allows them to verify transaction requests arriving from clients and to sign transaction results(endorsements). The ACL (Access Control List) is defined at channel creation time, and determines which peers and end users are permitted to perform certain actions.}

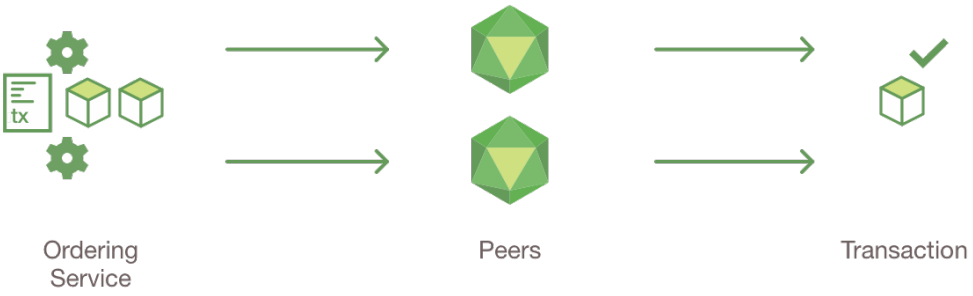
3. **Proposal responses are inspected**

The application verifies the endorsing peer signatures and compares the proposal responses (link to glossary term which will contain a representation of the payload) to determine if the proposal responses are the same and if the specified endorsement policy has been fulfilled (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 policy will still be enforced by peers and upheld at the commit validation phase.



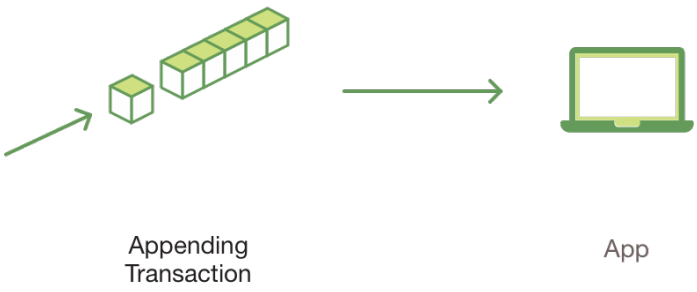
4. **Client assembles endorsements into a transaction**

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 read the transaction details, 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 *Chaincode Swimlanes* diagram to better understand the server side flow and the protobuffers.

CHAPTER 15

Architecture Explained

The v1 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 `\bot`, which is in case of `N` the lowest element. Initially all keys are mapped to `(\bot,\bot)`. 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\in K` and `v\in 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 connects 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 v1 will 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 (k/v) pairs. All k/v entries are versioned, that is, every entry contains ordered version information, which is incremented every time when the value stored under a key is updated. The peer that interprets the transaction records all k/v 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.

Fig. 15.1: Illustration of the transaction flow (common-case path).

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.

CHAPTER 16

Fabric CA User's Guide

Fabric CA is a Certificate Authority for Hyperledger Fabric.

It provides features such as:
1) registration of identities, or connects to LDAP as the user registry;
2) issuance of Enrollment Certificates (ECerts);
3) issuance of Transaction Certificates (TCerts), providing both anonymity and unlinkability when transacting on a Hyperledger Fabric blockchain;
4) certificate renewal and revocation.

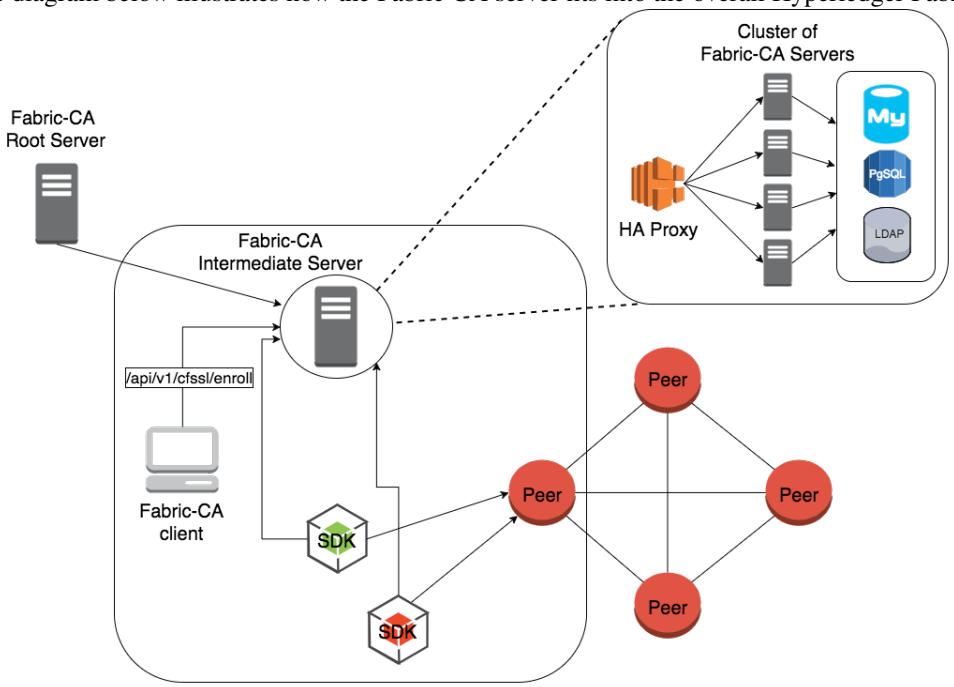
Fabric CA consists of both a server and a client component as described later in this document.

For developers interested in contributing to Fabric CA, see the Fabric CA repository for more information.

Table of Contents

Overview

The diagram below illustrates how the Fabric CA server fits into the overall Hyperledger Fabric architecture.



Text

There are two ways of interacting with a Fabric CA server: via the Fabric CA client or through one of the Fabric SDKs. All communication to the Fabric CA server is via REST APIs. See *fabric-ca/swagger/swagger-fabric-ca.json* for the swagger documentation for these REST APIs.

The Fabric CA client or SDK may connect to a server in a cluster of Fabric CA servers. This is illustrated in the top right section of the diagram. The client routes to an HA Proxy endpoint which load balances traffic to one of the fabric-ca-server cluster members. All Fabric CA servers in a cluster share the same database for keeping track of users and certificates. If LDAP is configured, the user information is kept in LDAP rather than the database.

Getting Started

Prerequisites

- Go 1.7+ installation or later
- **GOPATH** environment variable is set correctly
- libtool and libtdhl-dev packages are installed

The following installs the libtool dependencies.

```
# sudo apt install libtool libltdl-dev
```

For more information on libtool, see https://www.gnu.org/software/libtool.

For more information on libtdhr-dev, see https://www.gnu.org/software/libtool/manual/html_node/Using-libltdl.html.

Install

The following installs both the *fabric-ca-server* and *fabric-ca-client* commands.

```
# go get -u github.com/hyperledger/fabric-ca/cmd/...
```

Start Server Natively

The following starts the *fabric-ca-server* with default settings.

```
# fabric-ca-server start -b admin:adminpw
```

The *-b* option provides the enrollment ID and secret for a bootstrap administrator. A default configuration file named *fabric-ca-server-config.yaml* is created in the local directory which can be customized.

Start Server via Docker

The hyperledger/fabric-ca docker image is not currently being published, but you can build and start the server via docker-compose as shown below.

```
# cd $GOPATH/src/github.com/hyperledger/fabric-ca
# make docker
# cd docker/server
# docker-compose up -d
```

The hyperledger/fabric-ca docker image contains both the fabric-ca-server and the fabric-ca-client.

Explore the Fabric CA CLI

The following shows the Fabric CA server usage message:

```
Hyperledger Fabric Certificate Authority Server

Usage:
  fabric-ca-server [command]
```

```
Available Commands:
  init        Initialize the fabric-ca server
  start       Start the fabric-ca server

Flags:
      --address string                  Listening address of fabric-ca-server␣
↪(default "0.0.0.0")
  -b, --boot string                     The user:pass for bootstrap admin which is␣
↪required to build default config file
      --ca.certfile string              PEM-encoded CA certificate file (default "ca-
↪cert.pem")
      --ca.keyfile string               PEM-encoded CA key file (default "ca-key.pem")
  -c, --config string                   Configuration file (default "fabric-ca-server-
↪config.yaml")
      --csr.cn string                   The common name field of the certificate␣
↪signing request to a parent fabric-ca-server
      --csr.serialnumber string         The serial number in a certificate signing␣
↪request to a parent fabric-ca-server
      --db.datasource string            Data source which is database specific␣
↪(default "fabric-ca-server.db")
      --db.tls.certfiles string         PEM-encoded comma separated list of trusted␣
↪certificate files (e.g. root1.pem, root2.pem)
      --db.tls.client.certfile string   PEM-encoded certificate file when mutual␣
↪authenticate is enabled
      --db.tls.client.keyfile string    PEM-encoded key file when mutual␣
↪authentication is enabled
      --db.tls.enabled                  Enable TLS for client connection
      --db.type string                  Type of database; one of: sqlite3, postgres,␣
↪mysql (default "sqlite3")
  -d, --debug                           Enable debug level logging
      --ldap.enabled                    Enable the LDAP client for authentication and␣
↪attributes
      --ldap.groupfilter string         The LDAP group filter for a single␣
↪affiliation group (default "(memberUid=%s)")
      --ldap.url string                 LDAP client URL of form ldap://adminDN:
↪adminPassword@host[:port]/base
      --ldap.userfilter string          The LDAP user filter to use when searching␣
↪for users (default "(uid=%s)")
  -p, --port int                        Listening port of fabric-ca-server (default␣
↪7054)
      --registry.maxenrollments int     Maximum number of enrollments; valid if LDAP␣
↪not enabled
      --tls.certfile string             PEM-encoded TLS certificate file for server's␣
↪listening port (default "ca-cert.pem")
      --tls.enabled                     Enable TLS on the listening port
      --tls.keyfile string              PEM-encoded TLS key for server's listening␣
↪port (default "ca-key.pem")
  -u, --url string                      URL of the parent fabric-ca-server

Use "fabric-ca-server [command] --help" for more information about a command.
```

The following shows the Fabric CA client usage message:

```
# fabric-ca-client
Hyperledger Fabric Certificate Authority Client

Usage:
```

```
  fabric-ca-client [command]

Available Commands:
  enroll      Enroll user
  reenroll    Reenroll user
  register    Register user
  revoke      Revoke user

Flags:
  -c, --config string                Configuration file (default "/Users/saadkarim/.
↪fabric-ca-client/fabric-ca-client-config.yaml")
      --csr.cn string                The common name field of the certificate signing␣
↪request to a parent fabric-ca-server
      --csr.serialnumber string      The serial number in a certificate signing␣
↪request to a parent fabric-ca-server
  -d, --debug                        Enable debug logging
      --enrollment.hosts string      Comma-separated host list
      --enrollment.label string      Label to use in HSM operations
      --enrollment.profile string    Name of the signing profile to use in issuing␣
↪the certificate
      --id.affiliation string        Name associated with the identity
      --id.attr string               Attributes associated with this identity (e.g.␣
↪hf.revoker=true)
      --id.maxenrollments int        MaxEnrollments is the maximum number of times␣
↪the secret can be reused to enroll.
      --id.name string               Unique name of the identity
      --id.secret string             Secret is an optional password. If not␣
↪specified, a random secret is generated.
      --id.type string               Type of identity being registered (e.g. 'peer,␣
↪app, user')
  -m, --myhost string                Hostname to include in the certificate signing␣
↪request during enrollment (default "saads-mbp.raleigh.ibm.com")
      --tls.certfiles string         PEM-encoded comma separated list of trusted␣
↪certificate files (e.g. root1.pem, root2.pem)
      --tls.client.certfile string   PEM-encoded certificate file when mutual␣
↪authenticate is enabled
      --tls.client.keyfile string    PEM-encoded key file when mutual authentication␣
↪is enabled
      --tls.enabled                  Enable TLS for client connection
  -u, --url string                   URL of fabric-ca-server (default "http://
↪localhost:7054")

Use "fabric-ca-client [command] --help" for more information about a command.
```

Back to Top

File Formats

Fabric CA server's configuration file format

If no configuration file is provided to the server or no file exists, the server will generate a default configuration file like the one below. The location of the default configuration will depend on whether the `-c` or `--config` option was used or not. If the config option was used and the file did not exist it will be created in the specified location. However, if no config option was used, it will be create in the server home directory (see *Fabric CA Server* section more info).

```
# Server's listening port (default: 7054)
port: 7054

# Enables debug logging (default: false)
debug: false

#############################################################################
#  TLS section for the server's listening port
#############################################################################
tls:
  # Enable TLS (default: false)
  enabled: false
  # TLS for the server's listening port (default: false)
  certfile: ca-cert.pem
  keyfile: ca-key.pem

#############################################################################
#  The CA section contains the key and certificate files used when
#  issuing enrollment certificates (ECerts) and transaction
#  certificates (TCerts).
#############################################################################
ca:
  # Certificate file (default: ca-cert.pem)
  certfile: ca-cert.pem
  # Key file (default: ca-key.pem)
  keyfile: ca-key.pem

#############################################################################
#  The registry section controls how the fabric-ca-server does two things:
#  1) authenticates enrollment requests which contain a username and password
#     (also known as an enrollment ID and secret).
#  2) once authenticated, retrieves the identity's attribute names and
#     values which the fabric-ca-server optionally puts into TCerts
#     which it issues for transacting on the Hyperledger Fabric blockchain.
#     These attributes are useful for making access control decisions in
#     chaincode.
#  There are two main configuration options:
#  1) The fabric-ca-server is the registry
#  2) An LDAP server is the registry, in which case the fabric-ca-server
#     calls the LDAP server to perform these tasks.
#############################################################################
registry:
  # Maximum number of times a password/secret can be reused for enrollment
  # (default: 0, which means there is no limit)
  maxEnrollments: 0

  # Contains user information which is used when LDAP is disabled
  identities:
     - name: <<<ADMIN>>>
       pass: <<<ADMINPW>>>
       type: client
       affiliation: ""
       attrs:
          hf.Registrar.Roles: "client,user,peer,validator,auditor,ca"
          hf.Registrar.DelegateRoles: "client,user,validator,auditor"
          hf.Revoker: true
          hf.IntermediateCA: true
```

```
############################################################################
#  Database section
#  Supported types are: "sqlite3", "postgres", and "mysql".
#  The datasource value depends on the type.
#  If the type is "sqlite3", the datasource value is a file name to use
#  as the database store.  Since "sqlite3" is an embedded database, it
#  may not be used if you want to run the fabric-ca-server in a cluster.
#  To run the fabric-ca-server in a cluster, you must choose "postgres"
#  or "mysql".
############################################################################
database:
  type: sqlite3
  datasource: fabric-ca-server.db
  tls:
      enabled: false
      certfiles: db-server-cert.pem
      client:
        certfile: db-client-cert.pem
        keyfile: db-client-key.pem

############################################################################
#  LDAP section
#  If LDAP is enabled, the fabric-ca-server calls LDAP to:
#  1) authenticate enrollment ID and secret (i.e. username and password)
#     for enrollment requests;
#  2) To retrieve identity attributes
############################################################################
ldap:
   # Enables or disables the LDAP client (default: false)
   enabled: false
   # The URL of the LDAP server
   url: ldap://<adminDN>:<adminPassword>@<host>:<port>/<base>
   tls:
      certfiles: ldap-server-cert.pem
      client:
         certfile: ldap-client-cert.pem
         keyfile: ldap-client-key.pem

############################################################################
#  Affiliation section
############################################################################
affiliations:
   org1:
      - department1
      - department2
   org2:
      - department1

############################################################################
#  Signing section
############################################################################
signing:
    profiles:
      ca:
         usage:
           - cert sign
         expiry: 8000h
         caconstraint:
```

```
          isca: true
    default:
      usage:
        - cert sign
      expiry: 8000h

###############################################################################
#  Certificate Signing Request section for generating the CA certificate
###############################################################################
csr:
   cn: fabric-ca-server
   names:
      - C: US
        ST: "North Carolina"
        L:
        O: Hyperledger
        OU: Fabric
   hosts:
     - <<<MYHOST>>>
   ca:
      pathlen:
      pathlenzero:
      expiry:

###############################################################################
#  Crypto section configures the crypto primitives used for all
###############################################################################
crypto:
  software:
     hash_family: SHA2
     security_level: 256
     ephemeral: false
     key_store_dir: keys
```

Fabric CA client's configuration file format

If no configuration file is provided to the client, it will generate a default configuration file like the one below. The location of the default configuration file will depend on whether or not the `-c` or `--config` option was used. If the config option was used and the file did not exist, it will be created in the specified location. However, if no config option was used, it will be created in the in the Fabric CA client home directory (see *Fabric CA Client* section for more info)

```
###############################################################################
# Client Configuration
###############################################################################

# URL of the fabric-ca-server (default: http://localhost:7054)
URL: http://localhost:7054

###############################################################################
#    TLS section for the client's listenting port
###############################################################################
tls:
  # Enable TLS (default: false)
  enabled: false
```

```
  # TLS for the client's listenting port (default: false)
  certfiles:   # Comma Separated (e.g. root.pem, root2.pem)
  client:
    certfile:
    keyfile:

#############################################################################
#  Certificate Signing Request section for generating the CSR for
#  an enrollment certificate (ECert)
#############################################################################
csr:
  cn: <<<ENROLLMENT_ID>>>
  names:
    - C: US
      ST: "North Carolina"
      L:
      O: Hyperledger
      OU: Fabric
  hosts:
   - <<<MYHOST>>>
  ca:
    pathlen:
    pathlenzero:
    expiry:

#############################################################################
#  Registration section used to register a new user with fabric-ca server
#############################################################################
id:
  name:
  type:
  affiliation:
  attributes:
    - name:
      value:

#############################################################################
#  Enrollment section used to enroll a user with fabric-ca server
#############################################################################
enrollment:
  hosts:
  profile:
  label:
```

Back to Top

Configuration Settings Precedence

The Fabric CA provides 3 way to configure settings on the fabric-ca-server and fabric-ca-client. The precedence order is defined below:

1. CLI flags
2. Environment variables
3. Configuration file

In the remainder of this document, we refer to making changes to configuration files. However, configuration file changes can be overridden through environment variables or CLI flags.

For example, if we have the following in the client configuration file:

```
tls:
  # Enable TLS (default: false)
  enabled: false

  # TLS for the client's listenting port (default: false)
  certfiles:   # Comma Separated (e.g. root.pem, root2.pem)
  client:
    certfile: cert.pem
    keyfile:
```

The following environment variable may be used to override the `cert.pem` setting in the configuration file:

```
export FABRIC_CA_CLIENT_TLS_CLIENT_CERTFILE=cert2.pem
```

If we wanted to override both the environment variable and configuration file, we can use a command line flag.

```
fabric-ca-client enroll --tls.client.certfile cert3.pem
```

The same approach applies to fabric-ca-server, except instead of using `FABIRC_CA_CLIENT` as the prefix to environment variables, `FABRIC_CA_SERVER` is used.

Fabric CA Server

This section describes the Fabric CA server.

You may initialize the Fabric CA server before starting it if you prefer. This provides an opportunity for you to generate a default configuration file but to review and customize its settings before starting it.

The fabric-ca-server's home directory is determined as follows:
- if the `FABRIC_CA_SERVER_HOME` environment variable is set, use its value;
- otherwise, if `FABRIC_CA_HOME` environment variable is set, use its value;
- otherwise, if the `CA_CFG_PATH` environment variable is set, use its value;
- otherwise, use current working directory.

For the remainder of this server section, we assume that you have set the `FABRIC_CA_HOME` environment variable to `$HOME/fabric-ca/server`.

The instructions below assume that the server configuration file exists in the server's home directory.

Initializing the server

Initialize the Fabric CA server as follows:

```
# fabric-ca-server init -b admin:adminpw
```

The `-b` (bootstrap user) option is required for initialization. At least one bootstrap user is required to start the fabric-ca-server. The server configuration file contains a Certificate Signing Request (CSR) section that can be configured. The following is a sample CSR.

If you are going to connect to the fabric-ca-server remotely over TLS, replace "localhost" in the CSR section below with the hostname where you will be running your fabric-ca-server.

```
cn: localhost
key:
    algo: ecdsa
    size: 256
names:
  - C: US
    ST: "North Carolina"
    L:
    O: Hyperledger
    OU: Fabric
```

All of the fields above pertain to the X.509 signing key and certificate which is generated by the `fabric-ca-server init` . This corresponds to the `ca.certfile` and `ca.keyfile` files in the server's configuration file. The fields are as follows:

- **cn** is the Common Name
- **key** specifies the algorithm and key size as described below
- **O** is the organization name
- **OU** is the organizational unit
- **L** is the location or city
- **ST** is the state
- **C** is the country

If custom values for the CSR are required, you may customize the configuration file, delete the files specified by the `ca.certfile` and `ca-keyfile` configuration items, and then run the `fabric-ca-server init -b admin:adminpw` command again.

The `fabric-ca-server init` command generates a self-signed CA certificate unless the `-u <parent-fabric-ca-server-URL>` option is specified. If the `-u` is specified, the server's CA certificate is signed by the parent fabric-ca-server. The `fabric-ca-server init` command also generates a default configuration file named **fabric-ca-server-config.yaml** in the server's home directory.

Algorithms and key sizes

The CSR can be customized to generate X.509 certificates and keys that support both RSA and Elliptic Curve (ECDSA). The following setting is an example of the implementation of Elliptic Curve Digital Signature Algorithm (ECDSA) with curve `prime256v1` and signature algorithm `ecdsa-with-SHA256` :

```
key:
   algo: ecdsa
   size: 256
```

The choice of algorithm and key size are based on security needs.

Elliptic Curve (ECDSA) offers the following key size options:

size	ASN1 OID	Signature Algorithm
256	prime256v1	ecdsa-with-SHA256
384	secp384r1	ecdsa-with-SHA384
521	secp521r1	ecdsa-with-SHA512

RSA offers the following key size options:

size	Modulus (bits)	Signature Algorithm
2048	2048	sha256WithRSAEncryption
4096	4096	sha512WithRSAEncryption

Starting the server

Start the Fabric CA server as follows:

```
# fabric-ca-server start -b <admin>:<adminpw>
```

If the server has not been previously initialized, it will initialize itself as it starts for the first time. During this initialization, the server will generate the ca-cert.pem and ca-key.pem files if they don't yet exist and will also create a default configuration file if it does not exist. See the *Initialize the Fabric CA server* section.

Unless the fabric-ca-server is configured to use LDAP, it must be configured with at least one pre-registered bootstrap user to enable you to register and enroll other identities. The `-b` option specifies the name and password for a bootstrap user.

A different configuration file may be specified with the `-c` option as shown below.

```
# fabric-ca-server start -c <path-to-config-file> -b <admin>:<adminpw>
```

To cause the fabric-ca-server to listen on `http` rather than `https`, set `tls.enabled` to `true`.

To limit the number of times that the same secret (or password) can be used for enrollment, set the `registry.maxEnrollments` in the configuration file to the appropriate value. If you set the value to 1, the fabric-ca server allows passwords to only be used once for a particular enrollment ID. If you set the value to 0, the fabric-ca-server places no limit on the number of times that a secret can be reused for enrollment. The default value is 0.

The fabric-ca-server should now be listening on port 7054.

You may skip to the *Fabric CA Client* section if you do not want to configure the fabric-ca-server to run in a cluster or to use LDAP.

Configuring the database

This section describes how to configure the fabric-ca-server to connect to Postgres or MySQL databases. The default database is SQLite and the default database file is `fabric-ca-server.db` in the Fabric CA server's home directory.

If you don't care about running the fabric-ca-server in a cluster, you may skip this section; otherwise, you must configure either Postgres or MySQL as described below.

Postgres

The following sample may be added to the server's configuration file in order to connect to a Postgres database. Be sure to customize the various values appropriately.

```
db:
  type: postgres
  datasource: host=localhost port=5432 user=Username password=Password dbname=fabric-
↪ca-server sslmode=verify-full
```

Specifying *sslmode* configures the type of SSL authentication. Valid values for sslmode are:

Mode	Description
disable	No SSL
require	Always SSL (skip verification)
verify-ca	Always SSL (verify that the certificate presented by the server was signed by a trusted CA)
verify-full	Same as verify-ca AND verify that the certification presented by the server was signed by a trusted CA and the server host name matches the one in the certificate

If TLS would like to be used, we also need configure the TLS section in the fabric-ca-server config file. If the database server requires client authentication, then a client cert and key file needs to be provided. The following should be present in the fabric-ca-server config:

```
db:
  ...
  tls:
      enabled: false
      certfiles: db-server-cert.pem
      client:
            certfile: db-client-cert.pem
            keyfile: db-client-key.pem
```

certfiles - PEM-encoded trusted root certificate files.
certfile - PEM-encoded client certificate file.
keyfile - PEM-encoded client key file containing private key associated with client certificate file.

MySQL

The following sample may be added to the fabric-ca-server config file in order to connect to a MySQL database. Be sure to customize the various values appropriately.

```
db:
  type: mysql
  datasource: root:rootpw@tcp(localhost:3306)/fabric-ca?parseTime=true&tls=custom
```

If connecting over TLS to the MySQL server, the `db.tls.client` section is also required as described in the **Postgres** section above.

Configuring LDAP

The fabric-ca-server can be configured to read from an LDAP server.

In particular, the fabric-ca-server may connect to an LDAP server to do the following:

- authenticate a user prior to enrollment, and
- retrieve a user's attribute values which are used for authorization.

Modify the LDAP section of the server's configuration file to configure the fabric-ca-server to connect to an LDAP server.

```
ldap:
   # Enables or disables the LDAP client (default: false)
   enabled: false
   # The URL of the LDAP server
   url: scheme://<adminDN>:<adminPassword>@<host>:<port>/<base>
   userfilter: filter
```

where:
* `scheme` is one of *ldap* or *ldaps*;
* `adminDN` is the distinquished name of the admin user;
* `pass` is the password of the admin user;
* `host` is the hostname or IP address of the LDAP server;
* `port` is the optional port number, where default 389 for *ldap* and 636 for *ldaps*;
* `base` is the optional root of the LDAP tree to use for searches;
* `filter` is a filter to use when searching to convert a login user name to a distinquished name. For example, a value of `(uid=%s)` searches for LDAP entries with the value of a `uid` attribute whose value is the login user name. Similarly, `(email=%s)` may be used to login with an email address.

The following is a sample configuration section for the default settings for the OpenLDAP server whose docker image is at `https://github.com/osixia/docker-openldap`.

```
ldap:
   enabled: true
   url: ldap://cn=admin,dc=example,dc=org:admin@localhost:10389/dc=example,dc=org
   userfilter: (uid=%s)
```

See `FABRIC_CA/scripts/run-ldap-tests` for a script which starts an OpenLDAP docker image, configures it, runs the LDAP tests in `FABRIC_CA/cli/server/ldap/ldap_test.go`, and stops the OpenLDAP server.

When LDAP is configured, enrollment works as follows:

- The fabric-ca-client or client SDK sends an enrollment request with a basic authorization header.
- The fabric-ca-server receives the enrollment request, decodes the user name and password in the authorization header, looks up the DN (Distinquished Name) associated with the user name using the "userfilter" from the configuration file, and then attempts an LDAP bind with the user's password. If the LDAP bind is successful, the enrollment processing is authorized and can proceed.

When LDAP is configured, attribute retrieval works as follows:

- A client SDK sends a request for a batch of tcerts **with one or more attributes** to the fabric-ca-server.
- The fabric-ca-server receives the tcert request and does as follows:
  - extracts the enrollment ID from the token in the authorization header (after validating the token);
  - does an LDAP search/query to the LDAP server, requesting all of the attribute names received in the tcert request;
  - the attribute values are placed in the tcert as normal.

Setting up a cluster

You may use any IP sprayer to load balance to a cluster of fabric-ca servers. This section provides an example of how to set up Haproxy to route to a fabric-ca-server cluster. Be sure to change hostname and port to reflect the settings of

your fabric-ca servers.

haproxy.conf

```
global
      maxconn 4096
      daemon

defaults
      mode http
      maxconn 2000
      timeout connect 5000
      timeout client 50000
      timeout server 50000

listen http-in
      bind *:7054
      balance roundrobin
      server server1 hostname1:port
      server server2 hostname2:port
      server server3 hostname3:port
```

Back to Top

Fabric CA Client

This section describes how to use the fabric-ca-client command.

The fabric-ca-client's home directory is determined as follows:
- if the `FABRIC_CA_CLIENT_HOME` environment variable is set, use its value;
- otherwise, if the `FABRIC_CA_HOME` environment variable is set, use its value;
- otherwise, if the `CA_CFG_PATH` environment variable is set, use its value;
- otherwise, use `$HOME/.fabric-ca-client`.

The default fabric-ca-client's home directory is `$HOME/.fabric-ca-client`, but this can be changed by setting the `FABRIC_CA_HOME` or `FABRIC_CA_CLIENT_HOME` environment variable.

The instructions below assume that the client configuration file exists in the client's home directory.

Enrolling the bootstrap user

First, if desired, customize the CSR (Certificate Signing Request) in the client configuration file. If custom values for the CSR are required, you must create the client config file before triggering the `enroll` command and place it in the client's home directory. If no client configuration file is provided, default values will be used for CSR.

```
csr:
  key:
    algo: ecdsa
    size: 256
  names:
    - C: US
      ST: North Carolina
```

```
      L: Raleigh
      O: Hyperledger Fabric
      OU: Fabric CA
  hosts:
   - hostname
  ca:
    pathlen:
    pathlenzero:
    expiry:
```

See *CSR fields* for a description of the fields in this file. When enrolling, the CN (Common Name) field is automatically set to the enrollment ID which is *admin* in this example.

The following command enrolls the admin user and stores an enrollment certificate (ECert) in the fabric-ca-client's home directory.

```
# export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
# fabric-ca-client enroll -u http://admin:adminpw@localhost:7054
```

You should see a message similar to `[INFO] enrollment information was successfully stored in` which indicates where the certificate and key files were stored.

The enrollment certificate is stored at `$FABRIC_CA_ENROLLMENT_DIR/cert.pem` by default, but a different path can be specified by setting the `FABRIC_CA_CERT_FILE` environment variable.

The enrollment key is stored at `$FABRIC_CA_ENROLLMENT_DIR/key.pem` by default, but a different path can be specified by setting the `FABRIC_CA_KEY_FILE` environment variable.

If `FABRIC_CA_ENROLLMENT_DIR` is not set, the value of the fabric client home directory is used in its place.

Registering a new identity

The user performing the register request must be currently enrolled, and must also have the proper authority to register the type of user being registered.

In particular, two authorization checks are made by the fabric-ca-server during registration as follows.

1. The invoker's identity must have the "hf.Registrar.Roles" attribute with a comma-separated list of values where one of the value equals the type of identity being registered; for example, if the invoker's identity has the "hf.Registrar.Roles" attribute with a value of "peer,app,user", the invoker can register identities of type peer, app, and user, but not orderer.
2. The affiliation of the invoker's identity must be equal to or a prefix of the affiliation of the identity being registered. For example, an invoker with an affiliation of "a.b" may register an identity with an affiliation of "a.b.c" but may not register an identity with an affiliation of "a.c".

To register a new identity, you must first edit the `id` section in the client configuration file similar to the one below. This information describes the identity being registered.

```
id:
  name: MyPeer1
  type: peer
  affiliation: org1.department1
  attributes:
    - name: SomeAttrName
      value: SomeAttrValue
    - name: foo
      value: bar
```

The **id** field is the enrollment ID of the identity.

The **type** field is the type of the identity: orderer, peer, app, or user.

The **affiliation** field must be a valid group name as found in the server configuration file.

The **attributes** field is optional and is not required for a peer, but is shown here as example of how you associate attributes with any identity. Note that attribute names beginning with "hf." are reserved for Hyperledger Fabric usage (e.g. "hf.Revoker")

The following command uses the **admin** user's credentials to register the **peer1** identity.

```
# export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
# fabric-ca-client register
```

The output of a successful *fabric-ca-client register* command is a password similar to `Password: gHIexUckKpHz` . Make a note of your password to use in the following section to enroll a peer.

Suppose further than you wanted to register another peer and also want to provide your own password (or secret). You may do so as follows:

```
# export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
# fabric-ca-client register --id.name MyPeer2 --id.secret mypassword
```

Enrolling a Peer Identity

Now that you have successfully registered a peer identity, you may now enroll the peer given the enrollment ID and secret (i.e. the *password* from the previous section).

First, create a CSR (Certificate Signing Request) request file similar to the one described in the *Enrolling the bootstrap user* section.

This is similar to enrolling the bootstrap user except that we also demonstrate how to use environment variables to place the key and certificate files in a specific location. The following example shows how to place them into a Hyperledger Fabric MSP (Membership Service Provider) directory structure. The *MSP_DIR* environment variable refers to the root directory of MSP in Hyperledger Fabric and the `$MSP_DIR/signcerts` and `$MSP_DIR/keystore` directories must exist.

```
# export FABRIC_CA_CERT_FILE=$MSP_DIR/signcerts/peer.pem
# export FABRIC_CA_KEY_FILE=$MSP_DIR/keystore/key.pem
# fabric-ca client enroll -u http://peer1:<password>@localhost:7054
```

The cert.pem and key.pem files should now exist at the locations specified by the environment variables.

Reenrolling an Identity

Suppose your enrollment certificate is about to expire. You can issue the reenroll command to renew your enrollment certificate as follows. Note that this is identical to the enroll command except that no username or password is required. Instead, your previously stored private key is used to authenticate to the Fabric CA server.

```
# export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
# cd $FABRIC_CA_HOME
# fabric-ca-client reenroll
```

The enrollment certificate and enrollment key are stored in the same location as described in the previous section for the `enroll` command.

Revoking a certificate or identity

In order to revoke a certificate or user, the calling identity must have the `hf.Revoker` attribute. The revoking identity can only revoke a certificate or user that has an affiliation that is equal to or prefixed by the revoking identity's affiliation.

For example, a revoker with affiliation bank.bank_1 can revoke user with bank.bank1.dep1 but can't revoke bank.bank2.

You may revoke a specific certificate by specifying its AKI (Authority Key Identifier) and its serial number as follows:

```
fabric-ca-client revoke -a xxx -s yyy -r <reason>
```

The following command disables a user's identity and also revokes all of the certificates associated with the identity. All future requests received by the fabric-ca-server from this identity will be rejected.

```
fabric-ca-client revoke -e <enrollment_id> -r <reason>
```

The following are the supported reasons for revoking that can be specified using `-r` flag.

Reasons:
- unspecified
- keycompromise
- cacompromise
- affiliationchange
- superseded
- cessationofoperation
- certificatehold
- removefromcrl
- privilegewithdrawn
- aacompromise

Enabling TLS

This section describes in more detail how to configure TLS for a fabric-ca-client.

The following sections may be configured in the `fabric-ca-client-config.yaml`.

```
tls:
  # Enable TLS (default: false)
  enabled: true

  # TLS for the client's listenting port (default: false)
  certfiles: root.pem   # Comma Separated (e.g. root.pem,root2.pem)
  client:
    certfile: tls_client-cert.pem
    keyfile: tls_client-key.pem
```

The **certfiles** option is the set of root certificates trusted by the client. This will typically just be the root fabric-ca-server's certificate found in the server's home directory in the **ca-cert.pem** file.

The **client** option is required only if mutual TLS is configured on the server.

Back to Top

Appendix

Postgres SSL Configuration

Basic instructions for configuring SSL on Postgres server: 1. In postgresql.conf, uncomment SSL and set to "on" (SSL=on) 2. Place Certificate and Key files Postgress data directory.

Instructions for generating self-signed certificates for: https://www.postgresql.org/docs/9.1/static/ssl-tcp.html

Note: Self-signed certificates are for testing purposes and should not be used in a production environment

Postgres Server - Require Client Certificates 1. Place certificates of the certificate authorities (CAs) you trust in the file root.crt in the Postgres data directory 2. In postgresql.conf, set "ssl_ca_file" to point to the root cert of client (CA cert) 3. Set the clientcert parameter to 1 on the appropriate hostssl line(s) in pg_hba.conf.

For more details on configuring SSL on the Postgres server, please refer to the following Postgres documentation: https://www.postgresql.org/docs/9.4/static/libpq-ssl.html

MySQL SSL Configuration

On MySQL 5.7, strict mode affects whether the server permits '0000-00-00' as a valid date: If strict mode is not enabled, '0000-00-00' is permitted and inserts produce no warning. If strict mode is enabled, '0000-00-00' is not permitted and inserts produce an error.

Disabling STRICT_TRANS_TABLES mode

However to allow the format 0000-00-00 00:00:00, you have to disable STRICT_TRANS_TABLES mode in mysql config file or by command

Command: SET sql_mode = '';

File: Go to /etc/mysql/my.cnf and comment out STRICT_TRANS_TABLES

Basic instructions for configuring SSL on MySQL server:

1. Open or create my.cnf file for the server. Add or un-comment the lines below in [mysqld] section. These should point to the key and certificates for the server, and the root CA cert.

   Instruction on creating server and client side certs: http://dev.mysql.com/doc/refman/5.7/en/creating-ssl-files-using-openssl.html

   [mysqld] ssl-ca=ca-cert.pem ssl-cert=server-cert.pem ssl-key=server-key.pem

   Can run the following query to confirm SSL has been enabled.

   mysql> SHOW GLOBAL VARIABLES LIKE 'have_%ssl';

   Should see:

Variable_name	Value
have_openssl	YES
have_ssl	YES

2. After the server-side SSL configuration is finished, the next step is to create a user who has a privilege to access the MySQL server over SSL. For that, log in to the MySQL server, and type:

   mysql> GRANT ALL PRIVILEGES ON . TO 'ssluser'@'%' IDENTIFIED BY 'password' REQUIRE SSL; mysql> FLUSH PRIVILEGES;

   If you want to give a specific ip address from which the user will access the server change the '%' to the specific ip address.

MySQL Server - Require Client Certificates Options for secure connections are similar to those used on the server side.

- ssl-ca identifies the Certificate Authority (CA) certificate. This option, if used, must specify the same certificate used by the server.
- ssl-cert identifies the client public key certificate.
- ssl-key identifies the client private key.

Suppose that you want to connect using an account that has no special encryption requirements or was created using a GRANT statement that includes the REQUIRE SSL option. As a recommended set of secure-connection options, start the MySQL server with at least –ssl-cert and –ssl-key, and invoke the fabric-ca-server with **ca_certfiles** option set in the fabric-ca-server file.

To require that a client certificate also be specified, create the account using the REQUIRE X509 option. Then the client must also specify the proper client key and certificate files or the MySQL server will reject the connection. CA cert, client cert, and client key are all required for the fabric-ca-server.

Back to Top

CHAPTER 17

Node SDK

[WIP] ...coming soon

In the meantime, refer to the Hyperledger Fabric SDK design doc for more details on the APIs and specifications.

OR

Refer to the fabric-sdk-node repository in the Hyperledger community. The README will take you through a simple setup to build HTML output for the API classes and methods.

CHAPTER 18

What is chaincode?

[WIP]

coming soon ... end-to-end examples of chaincode demonstrating the available APIs.

Chaincode is a piece of code that is written in one of the supported languages such as Go or Java. It is installed and instantiated through an SDK or CLI onto a network of Hyperledger Fabric peer nodes, enabling interaction with that network's shared ledger.

There are three aspects to chaincode development:

- Chaincode Interfaces
- APIs
- Chaincode Responses

Chaincode interfaces

A chaincode implements the Chaincode Interface that supports two methods:

- `Init`
- `Invoke`

Init()

Init is called when you first deploy your chaincode. As the name implies, this function is used to do any initialization your chaincode needs.

Invoke()

Invoke is called when you want to call chaincode functions to do real work (i.e. read and write to the ledger). Invocations are captured as transactions, which get grouped into blocks on the chain. When you need to update or query the ledger, you do so by invoking your chaincode.

Dependencies

The import statement lists a few dependencies for the chaincode to compile successfully.

- fmt – contains `Println` for debugging/logging.
- errors – standard go error format.
- shim – contains the definitions for the chaincode interface and the chaincode stub, which are required to interact with the ledger.

Chaincode APIs

When the Init or Invoke function of a chaincode is called, the fabric passes the `stub shim.ChaincodeStubInterface` parameter and the chaincode returns a `pb.Response` . This stub can be used to call APIs to access to the ledger services, transaction context, or to invoke other chaincodes.

The current APIs are defined in the shim package, and can be generated with the following command:

```
godoc github.com/hyperledger/fabric/core/chaincode/shim
```

However, it also includes functions from chaincode.pb.go (protobuffer functions) that are not intended as public APIs. The best practice is to look at the function definitions in chaincode.go and and the examples directory.

Response

The chaincode response comes in the form of a protobuffer.

```
message Response {

    // A status code that should follow the HTTP status codes.
    int32 status = 1;

    // A message associated with the response code.
    string message = 2;

    // A payload that can be used to include metadata with this response.
    bytes payload = 3;

}
```

The chaincode will also return events. Message events and chaincode events.

```
messageEvent {

    oneof Event {

    //Register consumer sent event
    Register register = 1;

    //producer events common.
    Block block = 2;
    ChaincodeEvent chaincodeEvent = 3;
    Rejection rejection = 4;

    //Unregister consumer sent events
    Unregister unregister = 5;

    }
```

```
}
```

```
messageChaincodeEvent {

    string chaincodeID = 1;
    string txID = 2;
    string eventName = 3;
    bytes payload = 4;

}
```

Once developed and deployed, there are two ways to interact with the chaincode - through an SDK or the CLI. The steps for CLI are described below. For SDK interaction, refer to the balance transfer samples. **Note**: This SDK interaction is covered in the **Getting Started** section.

Command Line Interfaces

To view the currently available CLI commands, execute the following:

```
# this assumes that you have correctly set the GOPATH variable and cloned the Fabric␣
↪codebase into that path
cd /opt/gopath/src/github.com/hyperledger/fabric
build /bin/peer
```

You will see output similar to the example below. (**NOTE**: rootcommand below is hardcoded in main.go. Currently, the build will create a *peer* executable file).

```
Usage:
      peer [flags]
      peer [command]

    Available Commands:
      version     Print fabric peer version.
      node        node specific commands.
      channel     channel specific commands.
      chaincode   chaincode specific commands.
      logging     logging specific commands


    Flags:
      --logging-level string: Default logging level and overrides, see core.yaml for␣
↪full syntax
      --test.coverprofile string: Done (default "coverage.cov)
      -v, --version: Display current version of fabric peer server
    Use "peer [command] --help" for more information about a command.
```

The `peer` command supports several subcommands and flags, as shown above. To facilitate its use in scripted applications, the `peer` command always produces a non-zero return code in the event of command failure. Upon success, many of the subcommands produce a result on stdout as shown in the table below:

Command

stdout result in the event of success

version

String form of peer.version defined in core.yaml

node start

N/A

node status

String form of StatusCode

node stop

String form of StatusCode

chaincode deploy

The chaincode container name (hash) required for subsequent chaincode invoke and chaincode query commands

chaincode invoke

The transaction ID (UUID)

chaincode query

By default, the query result is formatted as a printable

channel create

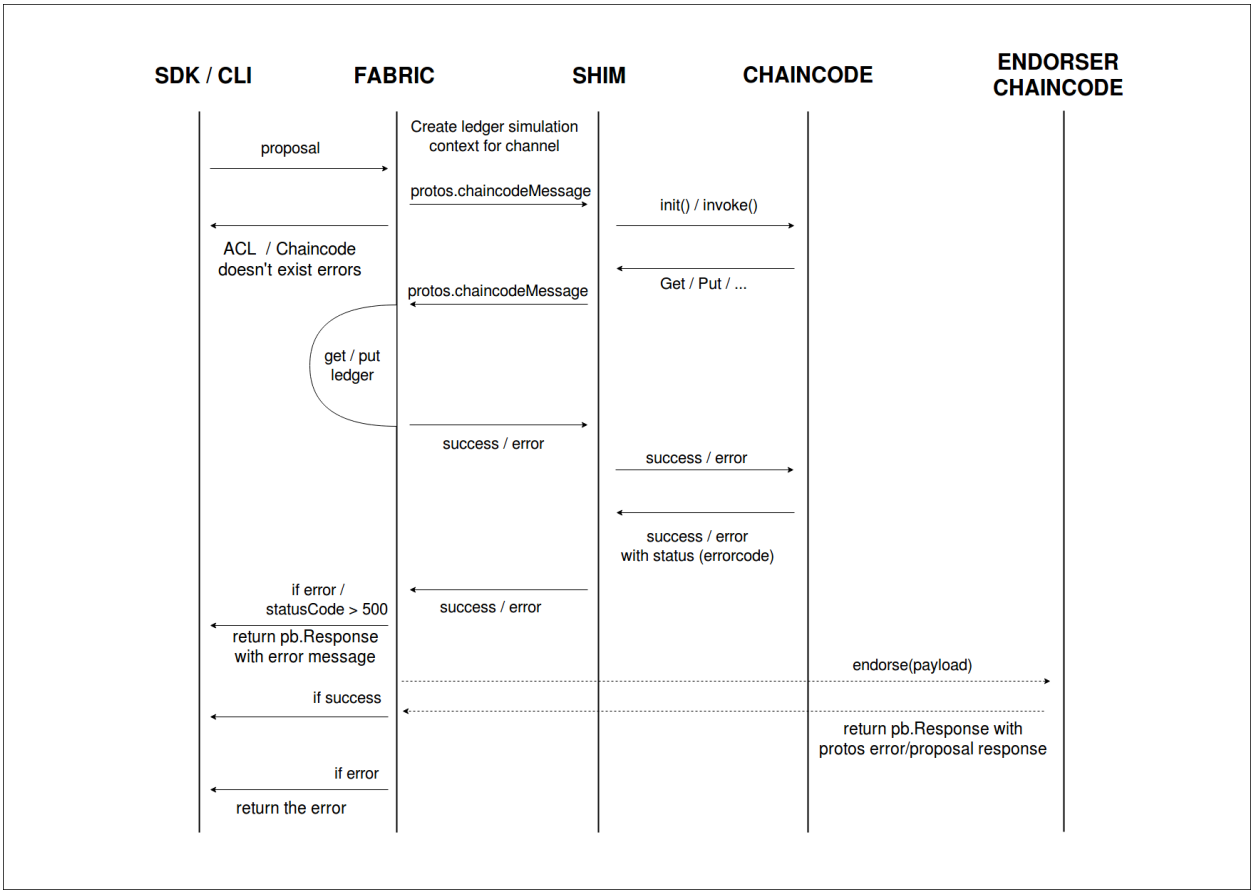
Create a chain

channel join

Adds a peer to the chain

Command line options support writing this value as raw bytes (-r, –raw) or formatted as the hexadecimal representation of the raw bytes (-x, –hex). If the query response is empty then nothing is output.

Chaincode Swimlanes



Deploy a chaincode

[WIP] - the CLI commands need to be refactored based on the new deployment model. Channel Create and Channel Join will remain the same.

CHAPTER 19

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 design

Endorsement policies have two main components: - a principal - a threshold gate

A principal `P` identifies the entity whose signature is expected.

A threshold gate `T` takes two inputs: an integer `t` (the threshold) and a list of `n` principals or gates; this gate essentially captures the expectation that out of those `n` principals or gates, `t` are requested to be satisfied.

For example: - `T(2,'A','B','C')` requests a signature from any 2 principals out of 'A', 'B' or 'C'; - `T(1,'A',T(2,'B','C'))` requests either one signature from principal `A` or 1 signature from `B` and `C` each.

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. Currently, two roles are supported: **member** and **admin**. Principals are described as `MSP.ROLE`, where `MSP` is the MSP ID that is required, and `ROLE` is either one of the two strings `member` and `admin`. Examples of valid principals are `'Org0.admin'` (any administrator of the `Org0` MSP) or `'Org1.member'` (any member 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 - the default policy requires one signature from a member of the `DEFAULT` MSP). This is used if a policy is not specified in the CLI.

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

For example:

```
peer chaincode deploy -C testchainid -n mycc -p github.com/hyperledger/fabric/
↪examples/chaincode/go/chaincode_example02 -c '{"Args":["init","a","100","b","200"]}
↪' -P "AND('Org1.member', 'Org2.member')"
```

This command deploys chaincode `mycc` on chain `testchainid` with the policy `AND('Org1.member','Org2.member')`.

Future enhancements

In this section we list future enhancements for endorsement policies: - alongside the existing way of identifying principals by their relationship with an MSP, we plan to identify principals in terms of the *Organization Unit (OU)* expected in their certificates; this is useful to express policies where we request signatures from any identity displaying a valid certificate with an OU matching the one requested in the definition of the principal. - instead of the syntax `AND(.,.)` we plan to move to a more intuitive syntax `.AND.` - we plan to expose generalized threshold gates in the language as well alongside `AND` (which is the special `n`-out-of-`n` gate) and `OR` (which is the special `1`-out-of-`n` gate)

CHAPTER 20

Ordering Service

[WIP] ...coming soon

This topic will outline the role and functionalities of the ordering service, and explain its place in the broader network and in the lifecycle of a transaction. The v1 architecture has been designed such that the ordering service is the centralized point of trust in a decentralized network, but also such that the specific implementation of "ordering" (solo, kafka, BFT) becomes a pluggable component.

Refer to the design document on a Kafka-based Ordering Service for more information on the default v1 implementation.

CHAPTER 21

Ledger

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.

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.

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. The 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* and *Read-Write set semantics* topics for a deeper dive on transaction structure, concurrency control, and the state DB.

State Database options

State database options include LevelDB and CouchDB (beta). 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 the fabric does not 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.

To enable CouchDB as the state database, configure the /fabric/peer/core.yaml `stateDatabase` section.

CHAPTER 22

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.

CHAPTER 23

Gossip data dissemination protocol

...coming soon

CHAPTER 24

Troubleshooting

[WIP] ...coming soon

This topic is intended to solve high level bugs and then direct users to more granular FAQ topics based on their errors.

CHAPTER 25

V1 Architecture

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 a way to share the source data outside of fabric. If you encrypt the data then you will need a way to share the decryption keys outside of fabric. 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 a way to share the source data outside of fabric. If you encrypt the data then you will need a way to share the decryption keys outside of fabric.

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?

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.

CHAPTER 26

Chaincode (Smart Contracts and Digital Assets)

- Does the fabric implementation support smart contract logic?

Yes. Chaincode is the fabric's 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.

- How do I create a business contract using the fabric?

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.

- How do I create assets using the fabric?

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 fabric does not advocate either one over the other. Instead, one of our first requirements was to ensure that both approaches can be easily implemented with tools available in the fabric.

- Which languages are supported for writing chaincode?

Chaincode can be written in any programming language and executed in containers inside the fabric context layer. We are also looking into developing a templating language (such as Apache Velocity) that can either get compiled into chaincode or have its interpreter embedded into a chaincode container.

The fabric's first fully supported chaincode language is Golang, and support for JavaScript and Java is planned for 2016. Support for additional languages and the development of a fabric-specific templating language have been discussed, and more details will be released in the near future.

- Does the fabric have native currency?

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.

CHAPTER 27

Confidentiality

- How is the confidentiality of transactions and business logic achieved?

The security module works in conjunction with the membership service module to provide access control service to any data recorded and business logic deployed on a chain network.

When a code is deployed on a chain network, whether it is used to define a business contract or an asset, its creator can put access control on it so that only transactions issued by authorized entities will be processed and validated by chain validators.

Raw transaction records are permanently stored in the ledger. While the contents of non-confidential transactions are open to all participants, the contents of confidential transactions are encrypted with secret keys known only to their originators, validators, and authorized auditors. Only holders of the secret keys can interpret transaction contents.

- What if none of the stakeholders of a business contract are

validators?

In some business scenarios, full confidentiality of contract logic may be required – such that only contract counterparties and auditors can access and interpret their chaincode. Under these scenarios, counter parties would need to spin off a new child chain with only themselves as validators.

CHAPTER 28

Identity Management (Membership Service)

- What is unique about the fabric's Membership Service module?

One of the things that makes the Membership Service module stand out from the pack is our implementation of the latest advances in cryptography.

In addition to ensuring private, auditable transactions, our Membership Service module introduces the concept of enrollment and transaction certificates. This innovation ensures that only verified owners can create asset tokens, allowing an infinite number of transaction certificates to be issued through parent enrollment certificates while guaranteeing the private keys of asset tokens can be regenerated if lost.

Issuers also have the ability revoke transaction certificates or designate them to expire within a certain timeframe, allowing greater control over the asset tokens they have issued.

Like most other modules on Fabric, you can always replace the default module with another membership service option should the need arise.

- Does its Membership Service make Fabric a centralized

solution?

No. The only role of the Membership Service module is to issue digital certificates to validated entities that want to participate in the network. It does not execute transactions nor is it aware of how or when these certificates are used in any particular network.

However, because certificates are the way networks regulate and manage their users, the module serves a central regulatory and organizational role.

Needs Review

CHAPTER 29

Glossary

Terminology is important, so that all Fabric users and developers agree on what we mean by each specific term. What is chaincode, for example. So we'll point you there, whenever you want to reassure yourself. Of course, 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 on a Fabric network, allowing 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

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 transaction and return a `YES/NO` response to the client application that generated the transaction proposal. 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 distinct endorsement policy for deploy transactions, which install new chaincode, is also required.

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 a chaincode container.

Invoke

Used to call chaincode functions. Invocations are captured as transaction proposals, which then pass through a modular flow of endorsement, ordering, validation, committal. The structure of invoke is a function 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.

The `fabric-ca` component is an implementation of membership services to manage identities. In particular, it handles the issuance and revocation of enrollment certificates and transaction certificates.

An enrollment certificate is a long-term identity credential; a transaction certificate is a short-term identity credential which is both anonymous and un-linkable.

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, block committal, 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 requests the value of a key(s) against the current state.

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 API uses protocol buffers over gRPC for transaction processing, membership services, node traversal and event handling applications to communicate across the fabric. The SDK comes in multiple flavors - Node.js, Java. and Python.

State Database

Current state data is stored in a state database for efficient reads and queries from chaincode. These 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

An invoke or instantiate operation. Invokes are requests to read/write data from the ledger. Instantiate is a request to start a chaincode container on a peer.

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.

Key enhancements:

- 8de58ed - NodeSDK doc changes – FAB-146
- 62d866d - Add flow control to SYNC_STATE_SNAPSHOT
- 4d97069 - Adding TLS changes to SDK
- e9d3ac2 - Node-SDK: add support for fabric events(block, chaincode, transactional)
- 7ed9533 - Allow deploying Java chaincode from remote git repositories
- 4bf9b93 - Move Docker-Compose files into their own folder
- ce9fcdc - Print ChaincodeName when deploy with CLI
- 4fa1360 - Upgrade go protobuf from 3-beta to 3
- 4b13232 - Table implementation in java shim with example
- df741bc - Add support for dynamically registering a user with attributes
- 4203ea8 - Check for duplicates when adding peers to the chain
- 518f3c9 - Update docker openjdk image
- 47053cd - Add GetTxID function to Stub interface (FAB-306)
- ac182fa - Remove deprecated devops REST API
- ad4645d - Support hyperledger fabric build on ppc64le platform
- 21a4a8a - SDK now properly adding a peer with an invalid URL
- 1d8114f - Fix setting of watermark on restore from crash
- a98c59a - Upgrade go protobuff from 3-beta to 3
- 937039c - DEVENV: Provide strong feedback when provisioning fails
- d74b1c5 - Make pbft broadcast timeout configurable
- 97ed71f - Java shim/chaincode project reorg, separate java docker env
- a76dd3d - Start container with HostConfig was deprecated since v1.10 and removed since v1.12
- 8b63a26 - Add ability to unregister for events
- 3f5b2fa - Add automatic peer command detection
- 6daedfd - Re-enable sending of chaincode events
- b39c93a - Update Cobra and pflag vendor libraries
- dad7a9d - Reassign port numbers to 7050-7060 range

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 30

Contributions Welcome!

We welcome contributions to the Hyperledger Project in many forms, and there's always plenty to do!

First things first, please review the Hyperledger Project's Code of Conduct before participating. It is important that we keep things civil.

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).

Setting up your SSH key

For Gerrit, before you can submit any change set for review, you will need to register your public SSH key. Login to Gerrit with your LFID, and click on your name in the upper right-hand corner of your browser window and then click 'Settings'. In the left-hand margin, you should see a link for 'SSH Public Keys'. Copy-n-paste your public SSH key into the window and press 'Add'.

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.

Requirements and Use Cases

We have a Requirements WG that is documenting use cases and from those use cases deriving requirements. If you are interested in contributing to this effort, please feel free to join the discussion in chat.

Reporting bugs

If you are a user and you find a bug, please submit an issue using 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 `#fabric-maintainers` channel in chat.

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.

Working with a local clone and Gerrit

We are using Gerrit to manage code contributions. If you are unfamiliar, please review this document before proceeding.

After you have familiarized yourself with `Gerrit` , and maybe played around with the `lf-sandbox` project, you should be ready to set up your local development environment.

Next, try building the project in your local development environment to ensure that everything is set up correctly.

Logging control describes how to tweak the logging levels of various components within the Fabric. Finally, every source file needs to include a license header: modified to include a copyright statement for the principle author(s).

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.

- 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.
- Write a meaningful commit message. Include a meaningful 50 (or less) character title, followed by a blank line, followed my a more comprehensive description of the change. Be sure to 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.

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.

Coding guidelines

Be sure to check out the language-specific style guides before making any changes. This will ensure a smoother review.

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 Project's 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.md file. If there are less than eight maintainers, a majority of the existing maintainers on that project are required to merge the change set. If there are more than eight existing maintainers, then if five or more of the maintainers concur with the proposal, the change set is then merged and the individual is added to (or alternatively,

removed from) the maintainers group. explicit resignation, some infraction of the code of conduct or consistently demonstrating poor judgement.

Legal stuff

Note: Each source file must include a license header for the Apache Software License 2.0. A template of that header can be found here.

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@hisdomain.com>
```

You can include this automatically when you commit a change to your local git repository using `git commit -s`.

CHAPTER 31

Maintainers

Name	Gerrit	GitHub	Slack	email
Binh Nguyen	binhn	binhn	binhn	binhn@us.ibm.com
Chris Ferris	ChristopherFerris	christo4ferris	cbf	chris.ferris@gmail.com
Gabor Hosszu	hgabre	gabre	hgabor	gabor@digitalasset.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	jzhang	jim_the_matrix@hotmail.com
Jonathan Levi	JonathanLevi	JonathanLevi	JonathanLevi	jonathan@hacera.com
Sheehan Anderson	sheehan	srderson	sheehan	sranderson@gmail.com
Srinivasan Muralidharan	muralisr	muralisrini	muralisr	muralisr@us.ibm.com
Tamas Blummer	TamasBlummer	tamasblummer	tamas	tamas@digitalasset.com
Yacov Manevich	yacovm	yacovm	yacovm	yacovm@il.ibm.com

CHAPTER 32

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 off 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**:

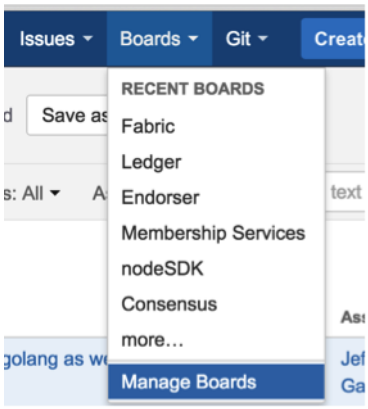


Fig. 32.1: Jira boards

Now on the left side of the screen click on **All boards**:

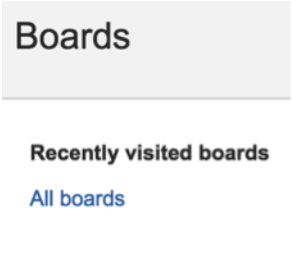


Fig. 32.2: Jira boards

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:

Board name	Board type	Administrators	Saved Filter	Visibility
Consensus	Scrum	Clayton Sims	Consensus	ALL USERS

Fig. 32.3: Jira boards

When you click on Consensus under **Board name** you will be directed to a page that contains the following columns:

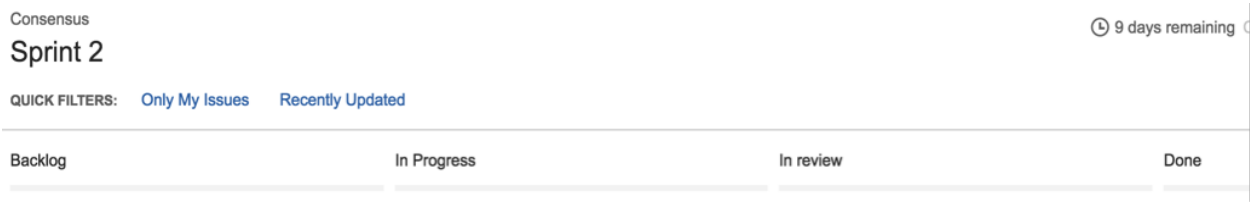


Fig. 32.4: Jira boards

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 – are items currently being worked by someone in the community.
- In Review – waiting to be reviewed and merged in Gerritt
- Done – 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:

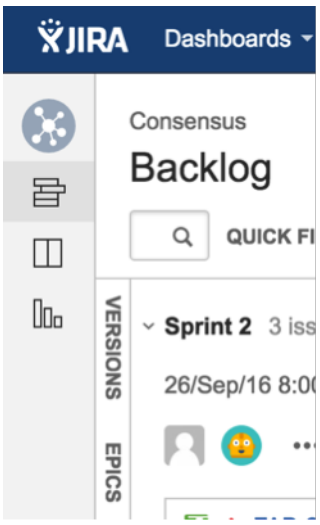


Fig. 32.5: Jira boards

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.

CHAPTER 33

Setting up the development environment

Overview

Through the v0.6 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 MacOSX, Windows, Linux, or whatever. Advances in Docker have enabled native support on the most popular development platforms: MacOSX 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 MacOSX 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 - 1.7 or later (for releases before v1.0, 1.6 or later)
- For MacOSX, Xcode must be installed
- Docker - 1.12 or later
- Pip
- (MacOSX) you may need to install gnutar, as MacOSX 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
```

- (only if using Vagrant) - Vagrant - 1.7.4 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, behave and docker-compose

```
pip install --upgrade pip
pip install behave nose docker-compose
pip install -I flask==0.10.1 python-dateutil==2.2 pytz==2014.3 pyyaml==3.10␣
↪couchdb==1.0 flask-cors==2.0.1 requests==2.4.3 pyOpenSSL==16.2.0 pysha3==1.0b1␣
↪grpcio==1.0.4

#PIP packages required for some behave tests
pip install urllib3 ndg-httpsclient pyasn1 ecdsa python-slugify grpcio-tools jinja2␣
↪b3j0f.aop six
```

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.

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 Fabric project

Since the Fabric project is a `Go` project, you'll need to clone the Fabric repo 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.

Boostrapping 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 MacOSX 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 peer project under `$GOPATH/src/github.com/hyperledger/fabric` . It is also mounted as `/hyperledger` .

Building the fabric

Once you have all the dependencies installed, and have cloned the repository, you can proceed to build and test the 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

CHAPTER 34

Building the fabric

The following instructions assume that you have already set up your development environment.

To build the 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 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/excecute

```
go test -v -run=TestGetFoo
```

Running Node.js Unit Tests

You must also run the Node.js unit tests to insure 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 devenv 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
```

CHAPTER 35

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
```

CHAPTER 36

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
```

CHAPTER 37

Logging

Logging utilizes the go-logging library.

The available log levels in order of increasing verbosity are: *CRITICAL* | *ERROR* | *WARNING* | *NOTICE* | *INFO* | *DEBUG*

See specific logging control instructions when running the peer process.

CHAPTER 38

Testing

[WIP] ...coming soon

This topic is intended to contain recommended test scenarios, as well as current performance numbers against a variety of configurations.

CHAPTER 39

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

CHAPTER 40

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
```

CHAPTER 41

Adding or updating Go packages

The Hyperledger Fabric Project uses Go 1.6 vendoring for package management. This means that all required packages reside in the `vendor` folder within the fabric project. 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
```

CHAPTER 42

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 the Hyperledger Fabric project not answered in this documentation, please use StackOverflow. If you need help finding things, please don't hesitate to send a note to the mailing list, or ask on RocketChat (an alternative to Slack).

CHAPTER 43

Quality

[WIP] ...coming soon

CHAPTER 44

Status

This project is an *Active* Hyperledger project. For more information on the history of this project see the Fabric wiki page. Information on what *Active* entails can be found in the Hyperledger Project Lifecycle document.

CHAPTER 45

License

The Hyperledger Project uses the Apache License Version 2.0 software license.