fabricdocs Documentation

Release 1.0

rameshthoomu

Feb 15, 2017

Warning: This build of the docs is from the “master” branch and Release “1.0“

CHAPTER 1

Overview

Hyperledger Fabric is a robust and flexible blockchain network architecture that provides enterprise-ready security, scalability, confidentiality and performance. Its unique implementation of distributed ledger technology ensures data integrity and consistency, while delivering accountability, transparency and efficiency. As a permissioned network, the fabric delivers a trusted blockchain network, where members are assured that all transactions can be detected and traced by authorized regulators and auditors.

Hyperledger Fabric separates chaincode execution from transaction ordering, which limits the required levels of trust and verification across nodes, optimizing network scalability and performance. Private channels provide multi- lateral transactions with the high degree of privacy and confidentiality required for competing businesses and regulated industries to coexist on a common network. The fabric incorporates a modular approach to blockchain, enabling network designers to plug in their preferred implementations for components such as ordering, identity management and encryption.

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

Attention: The Hyperledger Fabric project team is continually working to improve the security, performance and robustness of the released software, and frequently publishes updates. To stay current as the project progresses, please see the **Communication** and **Still Have Questions?** topics. Your participation in Linux Foundation projects is welcomed and encouraged!

CHAPTER 2

Why Hyperledger Fabric?

The Hyperledger Fabric project is delivering a blockchain platform designed to allow the exchange of an asset or the state of an asset to be consented upon, maintained, and viewed by all parties in a permissioned group. A key characteristic of Hyperledger Fabric is that the asset is defined digitally, with all participants simply agreeing on its representation/characterization. As such, Hyperledger Fabric can support a broad range of asset types; ranging from the tangible (real estate and hardware) to the intangible (contracts and intellectual property).

The technology is based on a standard blockchain concept - a shared, replicated ledger. However, Hyperledger Fabric is based on a *Hyperledger Fabric Glossary*, meaning all participants are required to be authenticated in order to participate and transact on the blockchain. Moreover, these identities can be used to govern certain levels of access control (e.g. this user can read the ledger, but cannot exchange or transfer assets). This dependence on identity is a great advantage in that varying consensus algorithms (e.g. byzantine or crash fault tolerant) can be implemented in place of the more compute-intensive Proof-of-Work and Proof-of-Stake varieties. As a result, permissioned networks tend to provide higher transaction throughput rates and performance.

Once an organization is granted access to the *blockchain network* *Hyperledger Fabric Glossary*, it then has the ability to create and maintain a private *channel* *Hyperledger Fabric Glossary* with other specified members. For example, let's assume there are four organizations trading jewels. They may decide to use Hyperledger Fabric because they trust each other, but not to an unconditional extent. They can all agree on the business logic for trading the jewels, and can all maintain a global ledger to view the current state of their jewel market (call this the consortium channel). Additionally, two or more of these organizations might decide to form an alternate private blockchain for a certain exchange that they want to keep confidential (e.g. price X for quantity Y of asset Z). They can perform this trade without affecting their broader consortium channel, or, if desired, this private channel can broadcast some level of reference data to their consortium channel.

This is powerful! This provides for great flexibility and potent capabilities, along with the interoperability of multiple blockchain ledgers within one consortium. This is the first of its kind and allows organizations to curate Hyperledger Fabric to support the myriad use cases for different businesses and industries. Hyperledger Fabric has already been successfully implemented in the banking, finance, and retail industries.

We welcome you to the Hyperledger Fabric community and are keen to learn of your architectural and business requirements, and help determine how Hyperledger Fabric can be leveraged to support your use cases.

CHAPTER 3

Hyperledger Fabric Glossary

Note: This glossary is structured to prioritize new terms and features specific to architecture. It makes the assumption that one already possesses a working familiarity with the basic tenets of blockchain.

3.1 Blockchain Network

A blockchain network consists of, at minimum, one peer (responsible for endorsing and committing transactions) leveraging an ordering service, and a membership services component (certificate authority) that distributes and revokes cryptographic certificates representative of user identities and permissions.

3.2 Permissioned Network

A blockchain network where any entity (node) is required to maintain a member identity on the network. End users must be authorized and authenticated in order to use the network.

3.3 Peer

Peer is a component that executes, and maintains a ledger of, transactions. There are two roles for a peer – endorser and committer. The architecture has been designed such that a peer is always a committer, but not necessarily always an endorser. Peers play no role in the ordering of transactions.

3.4 Member

A Member is a participant (such as a company or organization) that operates components - Peers, Orderers, and applications - in the blockchain network. A member is identified by its CA certificate (i.e. a unique enrollment). A Member's peer will be leveraged by end users in order to perform transaction operations on specific channels.

3.5 Transaction

Refers to an operation in which an authorized end user performs read/write operations against the ledger. There are three unique types of transactions - deploy, invoke, and query.

3.6 End User

An end user is someone who would interact with the blockchain through a set of published APIs (i.e. the hfc SDK). You can have an admin user who will typically grant permissions to the Member's components, and a client user, who, upon proper authentication through the admin user, will drive chaincode applications (deploy, invoke, query) on various channels. In the case of self-executing transactions, the application itself can also be thought of as the end user.

3.7 Ordering Service

A centralized or decentralized service that orders transactions in a block. You can select different implementations of the "ordering" function - e.g "solo" for simplicity and testing, Kafka for crash fault tolerance, or sBFT/PBFT for byzantine fault tolerance. You can also develop your own protocol to plug into the service.

3.8 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.

3.9 Orderer

One of the network entities that form the ordering service. A collection of ordering service nodes (OSNs) will order transactions into blocks according to the network's chosen ordering implementation. In the case of "solo", only one OSN is required. Transactions are "broadcast" to orderers, and then "delivered" as blocks to the appropriate channel.

3.10 Endorser

A specific peer role, where the Endorser peer is responsible for simulating transactions, and in turn preventing unstable or non-deterministic transactions from passing through the network. A transaction is sent to an endorser in the form of a transaction proposal. All endorsing peers are also committing peers (i.e. they write to the ledger).

3.11 Committer

A specific peer role, where the Committing peer appends the validated transactions to the channel-specific ledger. A peer can act as both an endorser and committer, but in more regulated circumstances might only serve as a committer.

3.12 Bootstrap

The initial setup of a network. There is the bootstrap of a peer network, during which policies, system chaincodes, and cryptographic materials (certs) are disseminated amongst participants, and the bootstrap of an ordering network. The bootstrap of the ordering network must precede the bootstrap of the peer network, as a peer network is contingent upon the presence of an ordering service. A network need only be "bootstrapped" once.

3.13 Block

A batch of ordered transactions, potentially containing ones of an invalid nature, that is delivered to the peers for validation and committal.

3.14 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: root certificates for participating organizations and ordering service nodes, policies, listening address for OSN, and configuration details. Any change to the overall network (e.g. a new org joining or a new OSN 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.

3.15 Channel

A Channel is formed as an offshoot of the system chain; and best thought of as a "topic" for peers to subscribe to, or rather, a subset of a broader blockchain network. A peer may subscribe on various channels and can only access the transactions on the subscribed channels. Each channel will have a unique ledger, thus accommodating confidentiality and execution of multilateral contracts.

3.16 Multi-channel

The fabric will allow for multiple channels with a designated ledger per channel. This capability allows for multilateral contracts where only the restricted participants on the channel will submit, endorse, order, or commit transactions on that channel. As such, a single peer can maintain multiple ledgers without compromising privacy and confidentiality.

3.17 Configuration Block

Contains the configuration data defining members and policies for a system chain or channel(s). Any changes to the channel(s) or overall network (e.g. a new member successfully 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. The policy to alter or edit a channel-level configuration block is defined through the Configuration System Chaincode (CSCC).

3.18 Genesis Block

The configuration block that initializes a blockchain network or channel, and also serves as the first block on a chain.

3.19 Ledger

An append-only transaction log managed by peers. Ledger keeps the log of ordered transaction batches. There are two denotations for ledger; peer and validated. The peer ledger contains all batched transactions coming out of the ordering service, some of which may in fact be invalid. The validated ledger will contain fully endorsed and validated transaction blocks. In other words, transactions in the validated ledger have passed the entire gamut of "consensus" - i.e. they have been endorsed, ordered, and validated.

3.20 Dynamic membership

he fabric will allow for endorsers and committers to come and go based on membership, and the blockchain network will continue to operate. Dynamic membership is critical when businesses grow and members need to be added or removed for various reasons.

3.21 Query/Non-Key Value Query

using couchDB 2.0 you now have the capability to leverage an API to perform more complex queries against combinations of variables, including time ranges, transaction types, users, etc. This feature allows for auditors and regulators to aggregate and mine large chunks of data.

3.22 Gossip Protocol

A communication protocol used among peers in a channel, to maintain their network and to elect Leaders, through which funnels all communications with the Ordering Service. Gossip allows for data dissemination, therein providing support for scalability due to the fact that not all peers are required to execute transactions and communicate with the ordering service.

3.23 System Chaincode

System Chaincode (SCC) is a chaincode built with the peer and run in the same process as the peer. SCC is responsible for broader configurations of fabric behavior, such as timing and naming services.

3.24 Lifecycle System Chaincode

Lifecycle System Chaincode (LSCC) is a system chaincode that handles deployment, upgrade and termination transactions for user chaincodes.

3.25 Configuration System Chaincode

Configuration System Chaincode (CSCC) is a "management" system chaincode that handles configuration requests to alter an aspect of a channel (e.g. add a new member). The CSCC will interrogate the channel's policies to determine if a new configuration block can be created.

3.26 Endorsement System Chaincode

Endorsement System Chaincode (ESCC) is a system chaincode that andles the endorsement policy for specific pieces of chaincode deployed on a network, and defines the necessary parameters (percentage or combination of signatures from endorsing peers) for a transaction proposal to receive a successful proposal response (i.e. endorsement). Deployments and invocations of user chaincodes both require a corresponding ESCC, which is defined at the time of the deployment transaction proposal for the user chaincode.

3.27 Validation System Chaincode

Validation System Chaincode (VSCC) Handles the validation policy for specific pieces of chaincode deployed on a network. Deployments and invocations of user chaincodes both require a corresponding VSCC, which is defined at the time of the deployment transaction proposal for the user chaincode. VSCC validates the specified level of "endorsement" (i.e. endorsement policy) in order to prevent malicious or faulty behavior from the client.

3.28 Policy

There are policies for endorsement, validation, block committal, chaincode management and network/channel management. Policies are defined through system chaincodes, and contain the requisite specifications for a network action to succeed. For example, an endorsement policy may require that 100% of endorsers achieve the same result upon transaction simulation.

3.29 Endorsement policy

A blockchain network must establish rules that govern the endorsement (or not) of proposed, simulated transactions. This endorsement 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.

3.30 Proposal

A transaction request sent from a client or admin user to one or more peers in a network; examples include deploy, invoke, query, or configuration request.

3.31 Deploy

Refers to the function through which chaincode applications are deployed on `chain`. A deploy is first sent from the client SDK or CLI to a Lifecycle System Chaincode in the form of a proposal.

3.32 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.

3.33 Membership Services

Membership Services manages user identities on a permissioned blockchain network; this function is implemented through the `fabric-ca` component. `fabric-ca` is comprised of a client and server, and handles the distribution and revocation of enrollment materials (certificates), which serve to identify and authenticate users on a network.

The in-line `MembershipSrvc` code (MSP) runs on the peers themselves, and is used by the peer when authenticating transaction processing results, and by the client to verify/authenticate transactions. Membership Services provides a distinction of roles by combining elements of Public Key Infrastructure (PKI) and decentralization (consensus). By contrast, non-permissioned networks do not provide member-specific authority or a distinction of roles.

A permissioned blockchain requires entities to register for long-term identity credentials (Enrollment Certificates), which can be distinguished according to entity type. For users, an Enrollment Certificate authorizes the Transaction Certificate Authority (TCA) to issue pseudonymous credentials; these certificates authorize transactions submitted by the user. Transaction certificates persist on the blockchain, and enable authorized auditors to associate, and identify the transacting parties for otherwise un-linkable transactions.

3.34 Membership Service Provider

The Membership Service Provider (MSP) refers to an abstract component of the system that provides (anonymous) 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.

3.35 Initialize

A chaincode method to define the assets and parameters in a piece of chaincode prior to issuing deploys and invocations. As the name implies, this function should be used to do any initialization to the chaincode, such as configure the initial state of a key/value pair on the ledger.

3.36 appshim

An application client used by ordering service nodes to process "broadcast" messages arriving from clients or peers. This shim allows the ordering service to perform membership-related functionality checks. In other words, is a peer or client properly authorized to perform the requested function (e.g. upgrade chaincode or reconfigure channel settings).

3.37 osshim

An ordering service client used by the application to process ordering service messages (i.e. "deliver" messages) that are advertised within a channel.

3.38 Hyperledger Fabric Client SDK

Provides a powerful set of APIs and contains myriad "methods" or "calls" that expose the capabilities and functionalities in the Hyperledger Fabric code base. For example, `addMember` , `removeMember` . The Fabric SDK comes in multiple flavors - Node.js, Java, and Python, for starters - thus, allowing developers to write application code in any of those programming languages.

3.39 Chaincode

Embedded logic that encodes the rules for specific types of network transactions. Developers write chaincode applications, which are then deployed onto a chain by an appropriately authorized member. End users then invoke chaincode through a client-side application that interfaces with a network peer. Chaincode runs network transactions, which if validated, are appended to the shared ledger and modify world state.

CHAPTER 4

Transaction Data Model

...coming soon

CHAPTER 5

Security Model

[WIP] Hyperledger Fabric allows for different organizations and participants in a common network to utilize their own certificate authority, and as a byproduct, implement varying cryptographic algorithms for signing/verifying/identity attestation. This is done through an MSP process running on both the ordering service and channel levels.

Membership service provider (MSP): A set of cryptographic mechanisms and protocols for issuing and validating certificates and identities throughout the blockchain network. Identities issued in the scope of a membership service provider can be evaluated within that membership service provider's rules validation policies.

CHAPTER 6

Multichannel

The fabric will allow for multiple channels with a designated ledger per channel (data segregation). This capability allows for multilateral contracts where only the restricted participants on the channel will submit, endorse, order, or commit transactions on that channel. As such, a single peer can maintain multiple ledgers without compromising privacy and confidentiality.

Refer to the multichannel design document for more detailed explanation on the mechanics and architecture.

CHAPTER 7

Smart Contracts

[WIP] Referred to as "chaincode" in Hyperledger Fabric.

Self-executing logic that encodes the rules for specific types of network transactions. Chaincode (currently written in Go or Java) is installed and instantiated onto a channel's peers by an appropriately authorized member. End users then invoke chaincode through a client-side application that interfaces with a network peer. Chaincode runs network transactions, which if validated, are appended to the shared ledger and modify world state.

CHAPTER 8

Consensus

[WIP] Not to be conflated with the ordering process. Consensus in v1 architecture is 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.

It is achieved as a byproduct of the various steps and verifications that occur during a transaction's lifecycle from proposal to commitment. More information on the high-level data flows is available here.

CHAPTER 9

Getting Started with v1.0 Hyperledger Fabric - App Developers

This document demonstrates an example using the Hyperledger Fabric V1.0 architecture. The scenario will include the creation and joining of channels, client side authentication, and the deployment and invocation of chaincode. CLI will be used for the creation and joining of the channel and the node SDK will be used for the client authentication, and chaincode functions utilizing the channel.

Docker Compose will be used to create a consortium of three organizations, each running an endorsing/committing peer, as well as a "solo" orderer and a Certificate Authority (CA). The cryptographic material, based on standard PKI implementation, has been pre-generated and is included in the `sfhackfest.tar.gz` in order to expedite the flow. The CA, responsible for issuing, revoking and maintaining the crypto material represents one of the organizations and is needed by the client (node SDK) for authentication. In an enterprise scenario, each organization might have their own CA, with more complex security measures implemented - e.g. cross-signing certificates, etc.

The network will be generated automatically upon execution of `docker-compose up` , and the APIs for create channel and join channel will be explained and demonstrated; as such, a user can go through the steps to manually generate their own network and channel, or quickly jump to the application development phase.

9.1 Prerequisites and setup

- Docker - v1.13 or higher
- Docker Compose - v1.8 or higher
- Node.js & npm - node v6.9.5 and npm v3.10.10 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
```

9.2 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
```

9.3 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.

9.4 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.

9.5 Use Docker to spawn network entities & create/join a channel

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
```

9.6 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.

9.7 Use node SDK to register/enroll user and deploy/invoke/query

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.

9.8 Manually create and join peers to a new channel

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.

9.9 Use cli to 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.

9.10 Creating your initial channel through the cli

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 the prior two sections.

9.11 Troubleshooting (optional)

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** section.

9.12 Clean up

Shut down your containers:

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

9.13 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 10

What's Included?

This section demonstrates an example using the Hyperledger Fabric V1.0 architecture. The scenario will include the creation and joining of channels, client side authentication, and the deployment and invocation of chaincode. CLI will be used for the creation and joining of the channel and the node SDK will be used for the client authentication, and chaincode functions utilizing the channel.

Docker Compose will be used to create a consortium of three organizations, each running an endorsing/committing peer, as well as a "solo" orderer and a Certificate Authority (CA). The cryptographic material, based on standard PKI implementation, has been pre-generated and is included in the `sfhackfest.tar.gz` in order to expedite the flow. The CA, responsible for issuing, revoking and maintaining the crypto material, represents one of the organizations and is needed by the client (node SDK) for authentication. In an enterprise scenario, each organization might have their own CA, with more complex security measures implemented - e.g. cross-signing certificates, etc.

The network will be generated automatically upon execution of `docker-compose up` , and the APIs for create channel and join channel will be explained and demonstrated; as such, a user can go through the steps to manually generate their own network and channel, or quickly jump to the application development phase.

It is recommended to run through this section in the order it is laid out - node program first, followed by the CLI approach.

CHAPTER 11

Prerequisites and setup

- Docker - v1.13 or higher
- Docker Compose - v1.8 or higher
- Node.js & npm - node v6.9.5 and npm v3.10.10 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
```

CHAPTER 13

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.

CHAPTER 14

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 entities & create/join a channel

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.

Use node SDK to register/enroll user, followed by deploy/invoke

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.

Manually create and join peers to a new channel

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.

Use cli to 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.

CHAPTER 20

Creating your initial channel through the cli

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.

CHAPTER 21

Troubleshooting (optional)

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** section.

CHAPTER 22

Clean up

Shut down your containers:

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

CHAPTER 23

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 24

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.

CHAPTER 25

Java 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-java repository in the Hyperledger community.

CHAPTER 26

Python 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-py repository in the Hyperledger community.

CHAPTER 27

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 28

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

CHAPTER 29

Commercial Paper

[WIP] ...coming soon

Web application demonstrating a commercial trading network and the issuance and maturation of trades. Not yet stable with v1 codebase.

Learn more about the application and underlying chaincode here

CHAPTER 30

Car Lease

[WIP] ...coming soon

Uses the blockchain to record the lifecycle of a vehicle from materials provenance, manufacture, buyer, all the way to scrap yard. Not yet stable with v1 codebase.

Learn more about the application and underlying chaincode here

CHAPTER 31

What is chaincode?

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: * The interfaces that the chaincode should implement * APIs the chaincode can use to interact with the Fabric * A chaincode response

31.1 Chaincode interfaces

A chaincode implements the Chaincode Interface that supports two methods: * `Init` * `Invoke`

31.1.1 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.

31.1.2 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.

31.2 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 you are required to interact with the ledger.

31.3 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.

31.4 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.

31.5 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.

31.6 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 32

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 33

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 34

Sample Application

[WIP] ...coming soon

In the meantime, refer to the Asset transfer through SDK topic.

CHAPTER 35

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 36

Administration and operations

[WIP] ...coming soon

CHAPTER 37

Debugging & Logging

[WIP] ...coming soon

CHAPTER 38

Logging Control

38.1 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.

38.2 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
```

38.3 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.

38.3.1 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 39

Recipe Book

[WIP] ...coming soon

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

CHAPTER 40

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 41

Architecture

Hyperledger Fabric is a unique implementation of distributed ledger technology (DLT) that ensures data integrity and consistency while delivering accountability, transparency, and efficiencies unmatched by other blockchain or DLT technology.

Hyperledger Fabric implements a specific type of permissioned blockchain network on which members can track, exchange and interact with digitized assets using transactions that are governed by smart contracts - what we call chaincode - in a secure and robust manner while enabling participants in the network to interact in a manner that ensures that their transactions and data can be restricted to an identified subset of network participants - something we call a channel.

The blockchain network supports the ability for members to establish shared ledgers that contain the source of truth about those digitized assets, and recorded transactions, that is replicated in a secure manner only to the set of nodes participating in that channel.

The Hyperledger Fabric architecture is comprised of the following components: peer nodes, ordering nodes and the clients applications that are likely leveraging one of the language-specific Fabric SDKs. These components have identities derived from certificate authorities. Hyperledger Fabric also offers a certificate authority service, *fabric-ca* but, you may substitute that with your own.

All peer nodes maintain the ledger/state by committing transactions. In that role, the peer is called a committer. Some peers are also responsible for simulating transactions by executing chaincodes (smart contracts) and endorsing the result. In that role the peer is called an endorser. A peer may be an endorser for certain types of transactions and just a ledger maintainer (committer) for others.

The orderers consent on the order of transactions in a block to be committed to the ledger. In common blockchain architectures (including earlier versions of the Hyperledger Fabric) the roles played by the peer and orderer nodes were unified (cf. validating peer in Hyperledger Fabric v0.6). The orderers also play a fundamental role in the creation and management of channels.

Two or more participants may create and join a channel, and begin to interact. Among other things, the policies governing the channel membership and chaincode lifecycle are specified at the time of channel creation. Initially, the members in a channel agree on the terms of the chaincode that will govern the transactions. When consensus is reached on the proposal to deploy a given chaincode (as governed by the life cycle policy for the channel), it is committed to the ledger.

Once the chaincode is deployed to the peer nodes in the channel, end users with the right privileges can propose transactions on the channel by using one of the language-specific client SDKs to invoke functions on the deployed chaincode.

The proposed transactions are sent to endorsers that execute the chaincode (also called "simulated the transaction"). On successful execution, endorse the result using the peer's identity and return the result to the client that initiated the proposal.

The client application ensures that the results from the endorsers are consistent and signed by the appropriate endorsers, according to the endorsement policy for that chaincode and, if so, the application then sends the transaction, comprised of the result and endorsements, to the ordering service.

Ordering nodes order the transactions - the result and endorsements received from the clients - into a block which is then sent to the peer nodes to be committed to the ledger. The peers then validate the transaction using the endorsement policy for the transaction's chaincode and against the ledger for consistency of result.

Some key capabilities of Hyperledger Fabric include:

- Allows for complex query for applications that need ability to handle complex data structures.
- Implements a permissioned network, also known as a consortia network, where all members are known to each other.
- Incorporates a modular approach to various capabilities, enabling network designers to plug in their preferred implementations for various capabilities such as consensus (ordering), identity management, and encryption.
- Provides a flexible approach for specifying policies and pluggable mechanisms to enforce them.
- Ability to have multiple channels, isolated from one another, that allows for multi-lateral transactions amongst select peer nodes, thereby ensuring high degrees of privacy and confidentiality required by competing businesses and highly regulated industries on a common network.
- Network scalability and performance are achieved through separation of chaincode execution from transaction ordering, which limits the required levels of trust and verification across nodes for optimization.

For a deeper dive into the details, please visit this document.

CHAPTER 42

Architecture Deep Dive

This page documents the architecture of a blockchain infrastructure with the roles of a blockchain node separated into roles of *peers* (who maintain state/ledger) and *orderers* (who consent on the order of transactions included in the ledger). In common blockchain architectures (including Hyperledger Fabric v0.6 and earlier) these roles are unified (cf. *validating peer* in Hyperledger Fabric v0.6). The architecture also introduces *endorsing peers* (endorsers), as special type of peers responsible for simulating execution and *endorsing* transactions (roughly corresponding to executing transactions in HL Fabric 0.6).

The architecture has the following advantages compared to the design in which peers/orderers/endorsers are unified (e.g., HL Fabric v0.6).

- **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.

This architecture drives the development of Hyperledger Fabric post-v0.6. As detailed below, some of its aspects are to be included in Hyperledger Fabric v1, whereas others are postponed to post-v1 versions of Hyperledger Fabric.

42.1 Table of contents

42.2 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*.

42.2.1 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).**

42.2.2 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.

42.2.3 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.

42.3 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.*

42.3.1 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.

42.3.2 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.).

42.3.3 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!

42.3.4 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` .

42.3.5 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. 42.1: Illustration of the transaction flow (common-case path).

Figure 1. Illustration of one possible transaction flow (common-case path).

42.4 3. Endorsement policies

42.4.1 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.

42.4.2 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.

42.4.3 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.

42.5 4 (post-v1). Validated ledger and `PeerLedger` checkpointing (pruning)

42.5.1 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.

42.5.2 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 43

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.

43.1 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.

43.2 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.

43.3 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')`.

43.4 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 44

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 45

Pluggable Ordering implementations

[WIP] ...coming soon

This topic is intended to explain how to configure an ordering service such that it implements a alternate protocol from the default kafka-based method.

This JIRA issue outlines the proposal for a Simplified Byzantine Fault Tolerant consensus protocol - https://jira.hyperledger.org/browse/FAB-378

CHAPTER 46

Ledger

[WIP] ...coming soon

The ledger exists as a peer process utilizing levelDB. It supports the high level transaction flow - read-write-set simulation, endorsement, MVCC check, file-based blockchain transaction log, and state database.

v1 architecture has been designed to support various ledger implementations such as couchDB, where more complexity with rich queries, pruning, archiving, etc... becomes possible.

For more information on the current state of ledger development, explore the corresponding JIRA issue - https://jira.hyperledger.org/browse/FAB-758

CHAPTER 47

Gossip protocol

[WIP] ...coming soon

v1 architecture utilizes the well-known concept of gossip protocol. See the design doc on Gossip-based data dissemination for more details on this.

CHAPTER 48

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.

CHAPTER 49

Getting Started

49.1 Prerequisites

- Go 1.7+ installation or later
- **GOPATH** environment variable is set correctly

49.2 Install

To install the fabric-ca command:

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

49.3 The Fabric CA CLI

The following shows the fabric-ca CLI usage:

```
# fabric-ca
fabric-ca client       - client related commands
fabric-ca server       - server related commands
fabric-ca cfssl        - all cfssl commands

For help, type "fabric-ca client", "fabric-ca server", or "fabric-ca cfssl".
```

The `fabric-ca server` and `fabric-ca client` commands are discussed below.

If you would like to enable debug-level logging (for server or client), set the `FABRIC_CA_DEBUG` environment variable to `true`.

Since fabric-ca is built on top of CFSSL, the `fabric-ca cfssl` commands are available but are not discussed in this document. See CFSSL for more information.

CHAPTER 50

Fabric CA Server

This section describes the fabric-ca server.

You must initialize the Fabric CA server before starting it.

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

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

Initialize the Fabric CA server as follows:

```
# fabric-ca server init CSR-JSON-FILE
```

The following is a sample `CSR-JSON-FILE` which you can customize as desired. The "CSR" stands for "Certificate Signing Request".

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

```
{
  "CN": "localhost",
  "key": { "algo": "ecdsa", "size": 256 },
  "names": [
      {
        "O": "Hyperledger Fabric",
        "OU": "Fabric CA",
        "L": "Raleigh",
        "ST": "North Carolina",
        "C": "US"
      }
  ]
}
```

All of the fields above pertain to the X.509 certificate which is generated by the `fabric server init` command as follows:

CSR fields

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

The `fabric-ca server init` command generates a self-signed X.509 certificate. It stores the certificate in the `server-cert.pem` file and the key in the `server-key.pem` file in the Fabric CA server's home directory.

Algorithms and key sizes

The CSR-JSON-FILE 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

Create a file named `server-config.json` as shown below in your fabric-ca server's home directory (e.g. *$HOME/fabric-ca/server*).

```
{
 "tls_disable": false,
 "ca_cert": "server-cert.pem",
 "ca_key": "server-key.pem",
 "driver":"sqlite3",
 "data_source":"fabric-ca.db",
 "user_registry": { "max_enrollments": 0 },
 "tls": {
     "tls_cert": "server-cert.pem",
     "tls_key": "server-key.pem"
 },
 "users": {
    "admin": {
      "pass": "adminpw",
      "type": "client",
      "group": "bank_a",
      "attrs": [
```

```
          {"name":"hf.Registrar.Roles","value":"client,peer,validator,auditor"},
          {"name":"hf.Registrar.DelegateRoles", "value": "client"}
       ]
    }
 },
 "groups": {
   "banks_and_institutions": {
     "banks": ["bank_a", "bank_b", "bank_c"],
     "institutions": ["institution_a"]
   }
 },
 "signing": {
    "default": {
       "usages": ["cert sign"],
       "expiry": "8000h",
       "ca_constraint": {"is_ca": true}
    }
 }
}
```

Now you may start the Fabric CA server as follows:

```
# cd $FABRIC_CA_HOME
# fabric-ca server start -address '0.0.0.0' -config server-config.json
```

To cause the fabric-ca server to listen on `http` rather than `https`, set `tls_disable` to `true` in the `server-config.json` file.

To limit the number of times that the same secret (or password) can be used for enrollment, set the `max_enrollments` in the `server-config.json` 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.

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.db` in the Fabric CA'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-config.json` file in order to connect to a Postgres database. Be sure to customize the various values appropriately.

```
"driver":"postgres",
"data_source":"host=localhost port=5432 user=Username password=Password dbname=fabric-
↪ca sslmode=verify-full",
```

Specifying *sslmode* enables SSL, and a value of *verify-full* means to verify that the certificate presented by the postgres server was signed by a trusted CA and that the postgres server's host name matches the one in the certificate.

We also need to set the TLS configuration 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:

```
"tls":{
  ...
  "db_client":{
    "ca_certfiles":["CA.pem"],
    "client":[{"keyfile":"client-key.pem","certfile":"client-cert.pem"}]
  }
},
```

ca_certfiles - The names of the trusted root certificate files.

certfile - Client certificate file.

keyfile - Client key file.

MySQL

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

```
...
"driver":"mysql",
"data_source":"root:rootpw@tcp(localhost:3306)/fabric-ca?parseTime=true&tls=custom",
...
```

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

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.

In order to configure the fabric-ca server to connect to an LDAP server, add a section of the following form to your fabric-ca server's configuration file:

```
{
   "ldap": {
       "url": "scheme://adminDN:pass@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": {
   "url": "ldap://cn=admin,dc=example,dc=org:admin@localhost:10389/dc=example,dc=org",
```

```
   "userfilter": "(uid=%s)"
},
```

See `FABRIC_CA/testdata/testconfig-ldap.json` for the complete configuration file with this section. Also 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:

- A 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/pass in the authorization header, looks up the DN (Distinquished Name) associated with the user using the "userfilter" from the configuration file, and then attempts an LDAP bind with the user's password. If 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

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

## Fabric CA Client
```

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

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

You must create a file named **client-config.json** in the fabric-ca client's home directory. The following is a sample client-config.json file:

```
{
  "ca_certfiles":["server-cert.pem"],
  "signing": {
    "default": {
       "usages": ["cert sign"],
       "expiry": "8000h"
    }
 }
}
```

You must also copy the server's certificate into the client's home directory. In the examples in this document, the server's certificate is at `$HOME/fabric-ca/server/server-cert.pem`. The file name must match the name in the *client-config.json* file.

Enroll the bootstrap user

Unless the fabric-ca server is configured to use LDAP, it must be configured with at least one pre-registered bootstrap user. In the previous server-config.json in this document, that user has an enrollment ID of *admin* with an enrollment secret of *adminpw*.

First, create a CSR (Certificate Signing Request) JSON file similar to

the following. Customize it as desired.

```
{
  "key": { "algo": "ecdsa", "size": 256 },
  "names": [
      {
        "O": "Hyperledger Fabric",
        "OU": "Fabric CA",
        "L": "Raleigh",
        "ST": "North Carolina",
        "C": "US"
      }
  ]
}
```

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, so it can be omitted from the csr.json file.

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

```
# export FABRIC_CA_HOME=$HOME/fabric-ca/clients/admin
# fabric-ca client enroll -config client-config.json admin adminpw http://localhost:
↪7054 csr.json
```

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_CA_HOME` environment variable is used in its place.

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, the invoker's identity must have been registered with the attribute "hf.Registrar.Roles". This attribute specifies the types of identities that the registrar is allowed to register.

For example, the attributes for a registrar might be as follows, indicating that this registrar identity can register peer, application, and user identities.

```
"attrs": [{"name":"hf.Registrar.Roles", "value":"peer,app,user"}]
```

To register a new identity, you must first create a JSON file similar to the one below which defines information for the identity being registered. This is a sample of registration information for a peer.

```
{
  "id": "peer1",
  "type": "peer",
  "group": "bank_a",
  "attrs": [{"name":"SomeAttrName","value":"SomeAttrValue"}]
}
```

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 **group** field must be a valid group name as found in the *server-config.json* file.

The **attrs** field is optional and is not required for a peer, but is shown here as example of how you associate attributes with any identity.

Assuming you store the information above in a file named **register.json**, the following command uses the **admin** user's credentials to register the **peer1** identity.

```
# export FABRIC_CA_HOME=$HOME/fabric-ca/clients/admin
# cd $FABRIC_CA_HOME
# fabric-ca client register -config client-config.json register.json http://localhost:
↪7054
```

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

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) JSON file similar to the one described in the *Enrolling the bootstrap user* section. Name the file *csr.json* for the following example.

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.

Also note that you must replace <*secret*> with the secret which was returned from the registration in the previous section.

```
# export FABRIC_CA_CERT_FILE=$MSP_DIR/signcerts/peer.pem
# export FABRIC_CA_KEY_FILE=$MSP_DIR/keystore/key.pem
# fabric-ca client enroll -config client-config.json peer1 <secret> https://localhost:
↪7054 csr.json
```

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

In order to revoke a certificate or user, the calling identity must have the `hf.Revoker` attribute.

You may revoke a specific certificate by specifying its AKI (Authority Key Identifier) and its serial number, as shown below.

```
fabric-ca client revoke -config client-config.json -aki xxx -serial yyy -reason "you
↪'re bad" https://localhost:7054
```

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 -config client-config.json https://localhost:7054 ENROLLMENT-
↪ID -reason "you're really bad"
```

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

The following sections may be configured in the `client-config.json`.

```
{
"ca_certfiles":["CA_root_cert.pem"],
"client":[{"keyfile":"client-key.pem","certfile":"client-cert.pem"}]
}
```

The **ca_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 **server-cert.pem** file.

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

CHAPTER 51

Appendix

51.1 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

51.2 MySQL SSL Configuration

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.

CHAPTER 52

Components

[WIP] ...coming soon

This topic will contain a diagram explaining the various components of a blockchain network and their corresponding roles.

CHAPTER 53

Transaction Flow

[WIP] ...coming soon

This topic will contain a diagram (currently in progress) outlining at a high level the basic flow of a transaction(s) from Application/SDK -> Endorsing Peers -> Back to SDK with proposal responses -> "Broadcast" to ordering service -> "Delivered" as a block to a channel's peers for validation and commitment (i.e. written to the shared ledger).

In the meantime, view the high level data flows and familiarize yourself with the concepts, components, and roles of system chaincodes.

CHAPTER 54

Endorsing Peer

[WIP] ...coming soon

This topic will explain the peer's runtime and role as an endorser for a certain piece of chaincode. In the meantime, refer to the high-level data flow.

CHAPTER 55

Committing Peer

[WIP] ...coming soon

This topic will explain the peer's runtime and role as a committer for transactions on a channel. In the meantime, refer to the high-level data flow.

CHAPTER 56

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 57

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 58

Confidentiality

58.1 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 59

Consensus Algorithm

Which Consensus Algorithm is used in the fabric? The fabric is built on a pluggable architecture such that developers can configure their deployment with the consensus module that best suits their needs. The initial release package will offer three consensus implementations for users to select from: 1) No-op (consensus ignored); and 2) Batch PBFT.

CHAPTER 60

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 the fabric, you can always replace the default module with another membership service option should the need arise.

Does its Membership Service make the 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.

CHAPTER 61

Usage

#####What are the expected performance figures for the fabric? The performance of any chain network depends on several factors: proximity of the validating nodes, number of validators, encryption method, transaction message size, security level set, business logic running, and the consensus algorithm deployed, among others.

The current performance goal for the fabric is to achieve 100,000 transactions per second in a standard production environment of about 15 validating nodes running in close proximity. The team is committed to continuously improving the performance and the scalability of the system.

Do I have to own a validating node to transact on a chain network? No. You can still transact on a chain network by owning a non-validating node (NV-node).

Although transactions initiated by NV-nodes will eventually be forwarded to their validating peers for consensus processing, NV-nodes establish their own connections to the membership service module and can therefore package transactions independently. This allows NV-node owners to independently register and manage certificates, a powerful feature that empowers NV-node owners to create custom-built applications for their clients while managing their client certificates.

In addition, NV-nodes retain full copies of the ledger, enabling local queries of the ledger data.

What does the error string "state may be inconsistent, cannot query" as a query result mean? Sometimes, a validating peer will be out of sync with the rest of the network. Although determining this condition is not always possible, validating peers make a best effort determination to detect it, and internally mark themselves as out of date.

When under this condition, rather than reply with out of date or potentially incorrect data, the peer will reply to chaincode queries with the error string "state may be inconsistent, cannot query".

In the future, more sophisticated reporting mechanisms may be introduced such as returning the stale value and a flag that the value is stale.

CHAPTER 62

Releases

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 63

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.

63.1 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).

63.1.1 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'.

63.2 Getting help

If you are looking for something to work on, or need some expert assistance in debugging a problem or working out a fix to an issue, our community is always eager to help. We hang out on Chat, IRC (#hyperledger on freenode.net) and the mailing lists. Most of us don't bite :grin: and will be glad to help. The only silly question is the one you don't ask. Questions are in fact a great way to help improve the project as they highlight where our documentation could be clearer.

63.3 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.

63.4 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.

63.5 Fixing issues and working stories

Review the issues list and find something that interests you. You could also check the "help-wanted" list. It is wise to start with something relatively straight forward and achievable, and that no one is already assigned. If no one is assigned, then assign the issue to yourself. Please be considerate and rescind the assignment if you cannot finish in a reasonable time, or add a comment saying that you are still actively working the issue if you need a little more time.

63.6 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).

63.7 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.

63.8 Coding guidelines

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

63.9 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.

63.10 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).

63.10.1 Becoming a maintainer

This project is managed under an open governance model as described in our charter. Projects or sub-projects will be lead by a set of maintainers. New sub-projects can designate an initial set of maintainers that will be approved by the top-level project's existing maintainers when the project is first approved. The project's maintainers will, from time-to-time, consider adding or removing a maintainer. An existing maintainer can submit a change set to the MAINTAINERS.rst file. 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.

63.11 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 64

Requesting a Linux Foundation Account

Contributions to the Fabric code base require a Linux Foundation account. Follow the steps below to create a Linux Foundation account.

64.1 Creating a Linux Foundation ID

1. Go to the Linux Foundation ID website.
2. Select the option `I need to create a Linux Foundation ID.`
3. Fill out the form that appears:
4. Open your email account and look for a message with the subject line: "Validate your Linux Foundation ID email".
5. Open the received URL to validate your email address.
6. Verify the browser displays the message `You have successfully validated your e-mail address.`
7. Access `Gerrit` by selecting `Sign In`:
8. Use your Linux Foundation ID to Sign In:

64.2 Configuring Gerrit to Use SSH

Gerrit uses SSH to interact with your Git client. A SSH private key needs to be generated on the development machine with a matching public key on the Gerrit server.

If you already have a SSH key-pair, skip this section.

As an example, we provide the steps to generate the SSH key-pair on a Linux environment. Follow the equivalent steps on your OS.

1. Create a key-pair, enter:

```
ssh-keygen -t rsa -C "John Doe john.doe@example.com"
```

Note: This will ask you for a password to protect the private key as it generates a unique key. Please keep this password private, and DO NOT enter a blank password.

The generated key-pair is found in: `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub`.

1. Add the private key in the `id_rsa` file in your key ring, e.g.:

```
ssh-add ~/.ssh/id_rsa
```

Once the key-pair has been generated, the public key must be added to Gerrit.

Follow these steps to add your public key `id_rsa.pub` to the Gerrit account:

1. Go to Gerrit.
2. Click on your account name in the upper right corner.
3. From the pop-up menu, select `Settings` .
4. On the left side menu, click on `SSH Public Keys` .
5. Paste the contents of your public key `~/.ssh/id_rsa.pub` and click `Add key` .

Note: The `id_rsa.pub` file can be opened with any text editor. Ensure that all the contents of the file are selected, copied and pasted into the `Add SSH key` window in Gerrit.

Note: The ssh key generation instructions operate on the assumtion that you are using the default naming. It is possible to generate multiple ssh Keys and to name the resulting files differently. See the ssh-keygen documentation for details on how to do that. Once you have generated non-default keys, you need to configure ssh to use the correct key for Gerrit. In that case, you need to create a `~/.ssh/config` file modeled after the one below.

```
host gerrit.hyperledger.org
 HostName gerrit.hyperledger.org
 IdentityFile ~/.ssh/id_rsa_hyperledger_gerrit
 User <LFID>
```

where is your Linux Foundation ID and the value of IdentityFile is the name of the public key file you generated.

Warning: Potential Security Risk! Do not copy your private key `~/.ssh/id_rsa` Use only the public `~/.ssh/id_rsa.pub` .

64.3 Checking Out the Source Code

1. Ensure that SSH has been set up properly. See `Configuring Gerrit to Use SSH` for details.
2. Clone the repository with your Linux Foundation ID ():

```
git clone ssh://<LFID>@gerrit.hyperledger.org:29418/fabric fabric
```

You have successfully checked out a copy of the source code to your local machine.

CHAPTER 65

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 66

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**:

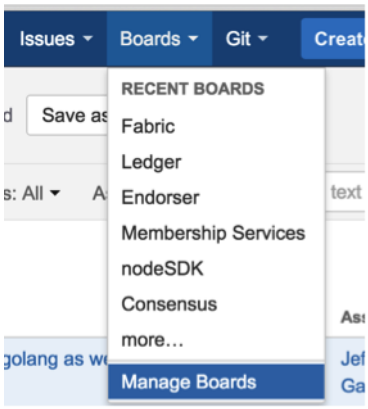

Fig. 66.1: Jira boards

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

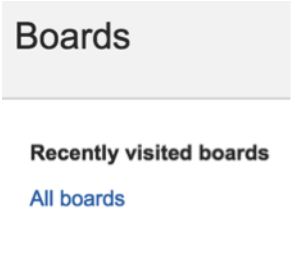

Fig. 66.2: Jira boards

On this page you will see all the public (and restricted) boards that have been created. If you want to see the items with current sprint focus, click on the boards where the column labeled **Visibility** is **All Users** and the column **Board type** is labeled **Scrum**. For example the **Board Name** Consensus:

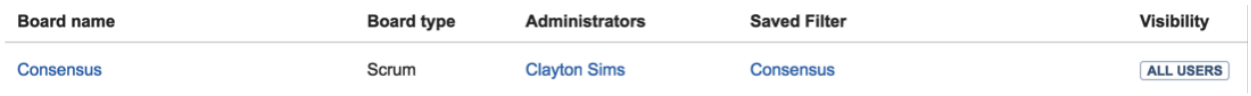

Fig. 66.3: Jira boards

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

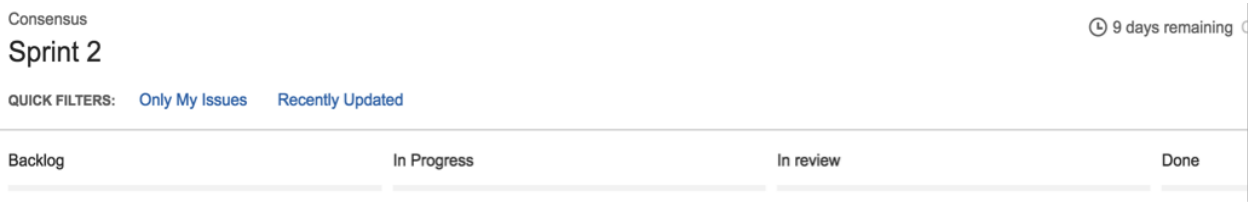

Fig. 66.4: Jira boards

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:

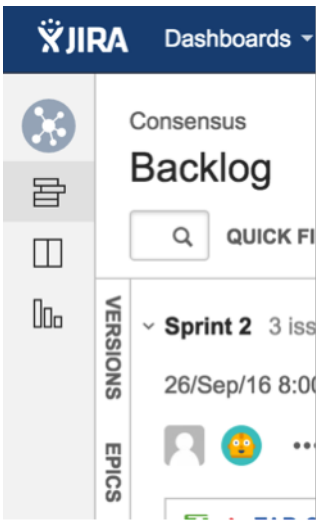

Fig. 66.5: Jira boards

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 67

Setting up the development environment

67.1 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.

67.2 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

67.3 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 sha3==0.2.1
```

67.4 Steps

67.4.1 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.

67.4.2 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
```

67.4.3 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.

67.4.4 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` .

67.5 Building the fabric

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

67.6 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 68

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
```

68.1 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
```

68.2 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.

68.3 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 69

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.

69.1 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
```

69.2 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 70

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 71

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 72

Working with Gerrit

Follow these instructions to collaborate on the Hyperledger Fabric Project through the Gerrit review system.

Please be sure that you are subscribed to the mailing list and of course, you can reach out on chat if you need help.

Gerrit assigns the following roles to users:

- **Submitters**: May submit changes for consideration, review other code changes, and make recommendations for acceptance or rejection by voting +1 or -1, respectively.
- **Maintainers**: May approve or reject changes based upon feedback from reviewers voting +2 or -2, respectively.
- **Builders**: (e.g. Jenkins) May use the build automation infrastructure to verify the change.

Maintainers should be familiar with the review process. However, anyone is welcome to (and encouraged!) review changes, and hence may find that document of value.

72.1 Git-review

There's a **very** useful tool for working with Gerrit called git-review. This command-line tool can automate most of the ensuing sections for you. Of course, reading the information below is also highly recommended so that you understand what's going on behind the scenes.

72.2 Sandbox project

We have created a sandbox project to allow developers to familiarize themselves with Gerrit and our workflows. Please do feel free to use this project to experiment with the commands and tools, below.

72.3 Getting deeper into Gerrit

A comprehensive walk-through of Gerrit is beyond the scope of this document. There are plenty of resources available on the Internet. A good summary can be found here. We have also provided a set of Best Practices that you may find helpful.

72.4 Working with a local clone of the repository

To work on something, whether a new feature or a bugfix:

1. Open the Gerrit Projects page
2. Select the project you wish to work on.
3. Open a terminal window and clone the project locally using the `Clone with git hook` URL. Be sure that `ssh` is also selected, as this will make authentication much simpler:

```
git clone ssh://LFID@gerrit.hyperledger.org:29418/fabric && scp -p -P 29418␣
↪LFID@gerrit.hyperledger.org:hooks/commit-msg fabric/.git/hooks/
```

Note: if you are cloning the fabric project repository, you will want to clone it to the `$GOPATH/src/github.com/hyperledger` directory so that it will build, and so that you can use it with the Vagrant development environment.

4. Create a descriptively-named branch off of your cloned repository

```
cd fabric
git checkout -b issue-nnnn
```

5. Commit your code. For an in-depth discussion of creating an effective commit, please read this document.

```
git commit -s -a
```

Then input precise and readable commit msg and submit.

6. Any code changes that affect documentation should be accompanied by corresponding changes (or additions) to the documentation and tests. This will ensure that if the merged PR is reversed, all traces of the change will be reversed as well.

72.5 Submitting a Change

Currently, Gerrit is the only method to submit a change for review. **Please review the 'guidelines <changes.md>'__ for making and submitting a change**.

72.5.1 Use git review

Note: if you prefer, you can use the *git-review* tool instead of the following. e.g.

Add the following section to `.git/config` , and replace `<USERNAME>` with your gerrit id.

```
[remote "gerrit"]
    url = ssh://<USERNAME>@gerrit.hyperledger.org:29418/fabric.git
    fetch = +refs/heads/*:refs/remotes/gerrit/*
```

Then submit your change with `git review` .

```
$ cd <your code dir>
$ git review
```

When you update your patch, you can commit with `git commit --amend` , and then repeat the `git review` command.

72.5.2 Not Use git review

Directions for building the source code can be found here.

When a change is ready for submission, Gerrit requires that the change be pushed to a special branch. The name of this special branch contains a reference to the final branch where the code should reside, once accepted.

For the Hyperledger Fabric Project, the special branch is called `refs/for/master`.

To push the current local development branch to the gerrit server, open a terminal window at the root of your cloned repository:

```
cd <your clone dir>
git push origin HEAD:refs/for/master
```

If the command executes correctly, the output should look similar to this:

```
Counting objects: 3, done.
Writing objects: 100% (3/3), 306 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote: Processing changes: new: 1, refs: 1, done
remote:
remote: New Changes:
remote:   https://gerrit.hyperledger.org/r/6 Test commit
remote:
To ssh://LFID@gerrit.hyperledger.org:29418/fabric
* [new branch]      HEAD -> refs/for/master
```

The gerrit server generates a link where the change can be tracked.

72.6 Adding reviewers

Optionally, you can add reviewers to your change.

To specify a list of reviewers via the command line, add `%r=reviewer@project.org` to your push command. For example:

```
git push origin HEAD:refs/for/master%r=rev1@email.com,r=rev2@notemail.com
```

Alternatively, you can auto-configure GIT to add a set of reviewers if your commits will have the same reviewers all at the time.

To add a list of default reviewers, open the :file:`.git/config` file in the project directory and add the following line in the `[ branch "master" ]` section:

```
[branch "master"] #.... push =
HEAD:refs/for/master%r=rev1@email.com,r=rev2@notemail.com`
```

Make sure to use actual email addresses instead of the `@email.com and @notemail.com` addressses. Don't forget to replace `origin` with your git remote name.

72.7 Reviewing Using Gerrit

- **Add**: This button allows the change submitter to manually add names of people who should review a change; start typing a name and the system will auto-complete based on the list of people registered and with access to

the system. They will be notified by email that you are requesting their input.

- **Abandon**: This button is available to the submitter only; it allows a committer to abandon a change and remove it from the merge queue.
- **Change-ID**: This ID is generated by Gerrit (or system). It becomes useful when the review process determines that your commit(s) have to be amended. You may submit a new version; and if the same Change-ID header (and value) are present, Gerrit will remember it and present it as another version of the same change.
- **Status**: Currently, the example change is in review status, as indicated by "Needs Verified" in the upper-left corner. The list of Reviewers will all emit their opinion, voting +1 if they agree to the merge, -1 if they disagree. Gerrit users with a Maintainer role can agree to the merge or refuse it by voting +2 or -2 respectively.

Notifications are sent to the email address in your commit message's Signed-off-by line. Visit your Gerrit dashboard, to check the progress of your requests.

The history tab in Gerrit will show you the in-line comments and the author of the review.

72.8 Viewing Pending Changes

Find all pending changes by clicking on the `All --> Changes` link in the upper-left corner, or open this link.

If you collaborate in multiple projects, you may wish to limit searching to the specific branch through the search bar in the upper-right side.

Add the filter *project:fabric* to limit the visible changes to only those from the Hyperledger Fabric Project.

List all current changes you submitted, or list just those changes in need of your input by clicking on `My --> Changes` or open this link

CHAPTER 73

Submitting a Change to Gerrit

Carefully review the following before submitting a change. These guidelines apply to developers that are new to open source, as well as to experienced open source developers.

73.1 Change Requirements

This section contains guidelines for submitting code changes for review. For more information on how to submit a change using Gerrit, please see Gerrit.

Changes are submitted as Git commits. Each commit must contain:

- a short and descriptive subject line that is 72 characters or fewer, followed by a blank line.
- a change description with your logic or reasoning for the changes, followed by a blank line
- a Signed-off-by line, followed by a colon (Signed-off-by:)
- a Change-Id identifier line, followed by a colon (Change-Id:). Gerrit won't accept patches without this identifier.

A commit with the above details is considered well-formed.

All changes and topics sent to Gerrit must be well-formed. Informationally, `commit messages` must include:

- **what** the change does,
- **why** you chose that approach, and
- **how** you know it works – for example, which tests you ran.

Commits must build cleanly when applied in top of each other, thus avoiding breaking bisectability. Each commit must address a single identifiable issue and must be logically self-contained.

For example: One commit fixes whitespace issues, another renames a function and a third one changes the code's functionality. An example commit file is illustrated below in detail:

```
A short description of your change with no period at the end

You can add more details here in several paragraphs, but please keep each line
width less than 80 characters. A bug fix should include the issue number.

Fix Issue # 7050.

Change-Id: IF7b6ac513b2eca5f2bab9728ebd8b7e504d3cebe1
Signed-off-by: Your Name <commit-sender@email.address>
```

Each commit must contain the following line at the bottom of the commit message:

```
Signed-off-by: Your Name <your@email.address>
```

The name in the Signed-off-by line and your email must match the change authorship information. Make sure your :file:`.git/config` is set up correctly. Always submit the full set of changes via Gerrit.

When a change is included in the set to enable other changes, but it will not be part of the final set, please let the reviewers know this.

CHAPTER 74

Reviewing a Change

1. Click on a link for incoming or outgoing review.
2. The details of the change and its current status are loaded:

- **Status:** Displays the current status of the change. In the example below, the status reads: Needs Verified.
- **Reply:** Click on this button after reviewing to add a final review message and a score, -1, 0 or +1.
- **Patch Sets:** If multiple revisions of a patch exist, this button enables navigation among revisions to see the changes. By default, the most recent revision is presented.
- **Download:** This button brings up another window with multiple options to download or checkout the current changeset. The button on the right copies the line to your clipboard. You can easily paste it into your git interface to work with the patch as you prefer.

Underneath the commit information, the files that have been changed by this patch are displayed.

3. Click on a filename to review it. Select the code base to differentiate against. The default is `Base` and it will generally be what is needed.
4. The review page presents the changes made to the file. At the top of the review, the presentation shows some general navigation options. Navigate through the patch set using the arrows on the top right corner. It is possible to go to the previous or next file in the set or to return to the main change screen. Click on the yellow sticky pad to add comments to the whole file.

The focus of the page is on the comparison window. The changes made are presented in green on the right versus the base version on the left. Double click to highlight the text within the actual change to provide feedback on a specific section of the code. Press *c* once the code is highlighted to add comments to that section.

5. After adding the comment, it is saved as a *Draft*.
6. Once you have reviewed all files and provided feedback, click the *green up arrow* at the top right to return to the main change page. Click the `Reply` button, write some final comments, and submit your score for the patch set. Click `Post` to submit the review of each reviewed file, as well as your final comment and score. Gerrit sends an email to the change-submitter and all listed reviewers. Finally, it logs the review for future reference. All individual comments are saved as *Draft* until the `Post` button is clicked.

CHAPTER 75

Gerrit Recommended Practices

This document presents some best practices to help you use Gerrit more effectively. The intent is to show how content can be submitted easily. Use the recommended practices to reduce your troubleshooting time and improve participation in the community.

75.1 Browsing the Git Tree

Visit Gerrit then select `Projects --> List --> SELECT-PROJECT --> Branches` . Select the branch that interests you, click on `gitweb` located on the right-hand side. Now, `gitweb` loads your selection on the Git web interface and redirects appropriately.

75.2 Watching a Project

Visit Gerrit, then select `Settings` , located on the top right corner. Select `Watched Projects` and then add any projects that interest you.

75.3 Commit Messages

Gerrit follows the Git commit message format. Ensure the headers are at the bottom and don't contain blank lines between one another. The following example shows the format and content expected in a commit message:

Brief (no more than 50 chars) one line description.

Elaborate summary of the changes made referencing why (motivation), what was changed and how it was tested. Note also any changes to documentation made to remain consistent with the code changes, wrapping text at 72 chars/line.

Jira: FAB-100
Change-Id: LONGHEXHASH
Signed-off-by: Your Name your.email@example.org
AnotherExampleHeader: An Example of another Value

The Gerrit server provides a precommit hook to autogenerate the Change-Id which is one time use.

Recommended reading: How to Write a Git Commit Message

75.4 Avoid Pushing Untested Work to a Gerrit Server

To avoid pushing untested work to Gerrit.

Check your work at least three times before pushing your change to Gerrit. Be mindful of what information you are publishing.

75.5 Keeping Track of Changes

- Set Gerrit to send you emails:
- Gerrit will add you to the email distribution list for a change if a developer adds you as a reviewer, or if you comment on a specific Patch Set.
- Opening a change in Gerrit's review interface is a quick way to follow that change.
- Watch projects in the Gerrit projects section at `Gerrit` , select at least *New Changes, New Patch Sets, All Comments* and *Submitted Changes*.

Always track the projects you are working on; also see the feedback/comments mailing list to learn and help others ramp up.

75.6 Topic branches

Topic branches are temporary branches that you push to commit a set of logically-grouped dependent commits:

To push changes from `REMOTE/master` tree to Gerrit for being reviewed as a topic in **TopicName** use the following command as an example:

$ git push REMOTE HEAD:refs/for/master/TopicName

The topic will show up in the review :abbr:`UI` and in the `Open Changes List` . Topic branches will disappear from the master tree when its content is merged.

75.7 Creating a Cover Letter for a Topic

You may decide whether or not you'd like the cover letter to appear in the history.

1. To make a cover letter that appears in the history, use this command:

```
git commit --allow-empty
```

Edit the commit message, this message then becomes the cover letter. The command used doesn't change any files in the source tree.

2. To make a cover letter that doesn't appear in the history follow these steps:

- Put the empty commit at the end of your commits list so it can be ignored without having to rebase.
- Now add your commits

```
git commit ...
git commit ...
git commit ...
```

- Finally, push the commits to a topic branch. The following command is an example:

```
git push REMOTE HEAD:refs/for/master/TopicName
```

If you already have commits but you want to set a cover letter, create an empty commit for the cover letter and move the commit so it becomes the last commit on the list. Use the following command as an example:

```
git rebase -i HEAD~#Commits
```

Be careful to uncomment the commit before moving it. `#Commits` is the sum of the commits plus your new cover letter.

75.8 Finding Available Topics

```
$ ssh -p 29418 gerrit.hyperledger.org gerrit query \ status:open project:fabric␣
↪branch:master \
| grep topic: | sort -u
```

- **'gerrit.hyperledger.org <>'__** Is the current URL where the project is hosted.
- *status* Indicates the topic's current status: open , merged, abandoned, draft, merge conflict.
- *project* Refers to the current name of the project, in this case fabric.
- *branch* The topic is searched at this branch.
- *topic* The name of an specific topic, leave it blank to include them all.
- *sort* Sorts the found topics, in this case by update (-u).

75.9 Downloading or Checking Out a Change

In the review UI, on the top right corner, the **Download** link provides a list of commands and hyperlinks to checkout or download diffs or files.

We recommend the use of the *git review* plugin. The steps to install git review are beyond the scope of this document. Refer to the git review documentation for the installation process.

To check out a specific change using Git, the following command usually works:

```
git review -d CHANGEID
```

If you don't have Git-review installed, the following commands will do the same thing:

```
git fetch REMOTE refs/changes/NN/CHANGEIDNN/VERSION \ && git checkout FETCH_HEAD
```

For example, for the 4th version of change 2464, NN is the first two digits (24):

```
git fetch REMOTE refs/changes/24/2464/4 \ && git checkout FETCH_HEAD
```

75.10 Using Draft Branches

You can use draft branches to add specific reviewers before you publishing your change. The Draft Branches are pushed to `refs/drafts/master/TopicName`

The next command ensures a local branch is created:

```
git checkout -b BRANCHNAME
```

The next command pushes your change to the drafts branch under **TopicName**:

```
git push REMOTE HEAD:refs/drafts/master/TopicName
```

75.11 Using Sandbox Branches

You can create your own branches to develop features. The branches are pushed to the `refs/sandbox/USERNAME/BRANCHNAME` location.

These commands ensure the branch is created in Gerrit's server.

```
git checkout -b sandbox/USERNAME/BRANCHNAME
git push --set-upstream REMOTE HEAD:refs/heads/sandbox/USERNAME/BRANCHNAME
```

Usually, the process to create content is:

- develop the code,
- break the information into small commits,
- submit changes,
- apply feedback,
- rebase.

The next command pushes forcibly without review:

```
git push REMOTE sandbox/USERNAME/BRANCHNAME
```

You can also push forcibly with review:

```
git push REMOTE HEAD:ref/for/sandbox/USERNAME/BRANCHNAME
```

75.12 Updating the Version of a Change

During the review process, you might be asked to update your change. It is possible to submit multiple versions of the same change. Each version of the change is called a patch set.

Always maintain the **Change-Id** that was assigned. For example, there is a list of commits, **c0...c7**, which were submitted as a topic branch:

```
git log REMOTE/master..master

c0
...
c7

git push REMOTE HEAD:refs/for/master/SOMETOPIC
```

After you get reviewers' feedback, there are changes in **c3** and **c4** that must be fixed. If the fix requires rebasing, rebasing changes the commit Ids, see the rebasing section for more information. However, you must keep the same Change-Id and push the changes again:

```
git push REMOTE HEAD:refs/for/master/SOMETOPIC
```

This new push creates a patches revision, your local history is then cleared. However you can still access the history of your changes in Gerrit on the `review UI` section, for each change.

It is also permitted to add more commits when pushing new versions.

75.13 Rebasing

Rebasing is usually the last step before pushing changes to Gerrit; this allows you to make the necessary *Change-Ids*. The *Change-Ids* must be kept the same.

- **squash:** mixes two or more commits into a single one.
- **reword:** changes the commit message.
- **edit:** changes the commit content.
- **reorder:** allows you to interchange the order of the commits.
- **rebase:** stacks the commits on top of the master.

75.14 Rebasing During a Pull

Before pushing a rebase to your master, ensure that the history has a consecutive order.

For example, your `REMOTE/master` has the list of commits from **a0** to **a4**; Then, your changes **c0...c7** are on top of **a4**; thus:

```
git log --oneline REMOTE/master..master

a0
a1
a2
a3
a4
c0
c1
...
c7
```

If `REMOTE/master` receives commits **a5**, **a6** and **a7**. Pull with a rebase as follows:

```
git pull --rebase REMOTE master
```

This pulls **a5-a7** and re-apply **c0-c7** on top of them:

```
$ git log --oneline REMOTE/master..master
a0
...
a7
c0
```

```
c1
...
c7
```

75.15 Getting Better Logs from Git

Use these commands to change the configuration of Git in order to produce better logs:

```
git config log.abbrevCommit true
```

The command above sets the log to abbreviate the commits' hash.

```
git config log.abbrev 5
```

The command above sets the abbreviation length to the last 5 characters of the hash.

```
git config format.pretty oneline
```

The command above avoids the insertion of an unnecessary line before the Author line.

To make these configuration changes specifically for the current Git user, you must add the path option `--global` to `config` as follows:

CHAPTER 76

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 77

Coding guidelines

77.1 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 78

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 79

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 80

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 81

Quality

[WIP] ...coming soon

CHAPTER 82

Incubation Notice

This project is a Hyperledger project in *Incubation*. It was proposed to the community and documented here. Information on what *Incubation* entails can be found in the Hyperledger Project Lifecycle document.

CHAPTER 83

License

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