hyperledger-fabricdocs Documentation

Release v0.6

hyperledger

Apr 04, 2017

This build of the docs is from the “v0.6” branch

Hyperledger Fabric

The Hyperledger fabric is an implementation of blockchain technology, that has been collaboratively developed under the Linux Foundation's Hyperledger Project. It leverages familiar and proven technologies, and offers a modular architecture that allows pluggable implementations of various functions including membership services, consensus, and smart contracts (chaincode) execution. It features powerful container technology to host any mainstream language for smart contracts development.

Status

This project is an archived Hyperledger project. It was proposed to the community and documented here. Information on this project's history can be found in the Hyperledger Project Lifecycle document.

Releases

The fabric releases are documented *here*. We have just released our second release under the governance of the Hyperledger Project - v0.6-preview.

Fabric Starter Kit

If you'd like to dive right in and get an operational experience on your local server or laptop to begin development, we have just the thing for you. We have created a standalone Docker-based *starter kit* that leverages the latest published Docker images that you can run on your laptop and be up and running in no time. That should get you going with a sample application and some simple chaincode. From there, you can go deeper by exploring our *Developer guides*.

Contributing to the project

We welcome contributions to the Hyperledger Project in many forms. There's always plenty to do! Full details of how to contribute to this project are documented in the *Fabric developer guide* below.

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).

Communication

We use Rocket Chat 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.

Still Have Questions?

We try to maintain a comprehensive set of documentation (see below) 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 Rocket Chat.

Table of Contents

Below, you'll find the following sections:

Read all about it

If you are new to the project, you can begin by reviewing the following links. If you'd prefer to dive right in, see the Quickstart section, below.

- Whitepaper WG: where the community is developing a whitepaper to explain the motivation and goals for the project.
- Requirements WG: where the community is developing use cases and requirements.
- *Canonical use cases*
- *Glossary*: to understand the terminology that we use throughout the Fabric project's documentation.
- Fabric FAQs

CHAPTER 2

Developer guides

Chaincode developer guide

- *Setting up the development environment* when developing and testing Chaincode, or an application that leverages the fabric API or SDK, you'll probably want to run the fabric locally on your laptop to test. You can use the same setup that Fabric developers use.
- *Setting Up a Network For Development* alternately, you can follow these instructions for setting up a local network for Chaincode development without the entire fabric development environment setup.
- *Writing, Building, and Running Chaincode in a Development Environment* a step-by-step guide to writing and testing Chaincode.
- *Chaincode FAQ* a FAQ for all of your burning questions relating to Chaincode.

Application developer guide

- *APIs - CLI, REST, and Node.js*
- *CLI* - working with the command-line interface.
- *REST* - working with the REST API (*deprecated*).
- *Node.js SDK* - working with the Node.js SDK.

Fabric developer guide

- First, you'll want to familiarize yourself with the project's *code contribution guidelines*
- After that, you will want to *set up the development environment*
- try *building the fabric core* in your local development environment to ensure that everything is set up correctly.
- For the *adventurous*, you might try *building outside of the standard Vagrant development environment*
- *Logging control* describes how to tweak the logging levels of various components within the fabric.
- Every source file must include this license header modified to include a copyright statement for the principle author(s).

CHAPTER 3

Operations guide

- *Setting Up a Network* instructions for setting up a network of fabric peers.
- *Certificate Authority (CA) Setup* setting up a CA to support identity, security (authentication/authorization), privacy and confidentiality.
- *Application ACL* working with access control lists.

CHAPTER 4

License

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

CHAPTER 5

Roles & Personas

Roles

Chain Member

Entities that do not participate in the validation process of a blockchain network, but help to maintain the integrity of a network. Unlike Chain transactors, chain members maintain a local copy of the ledger.

Chain Transactor

Entities that have permission to create transactions and query network data.

Chain Validator

Entities that own a stake of a chain network. Each chain validator has a voice in deciding whether a transaction is valid, therefore chain validators can interrogate all transactions sent to their chain.

Chain Auditor

Entities with the permission to interrogate transactions.

Solution User

End users are agnostic about the details of chain networks, they typically initiate transactions on a chain network through applications made available by solutions providers.

Roles: None

Solution Provider

Organizations that develop mobile and/or browser based applications for end (solution) users to access chain networks. Some application owners may also be network owners.

Roles: Chain Transactor

Network Proprietor

Proprietor(s) setup and define the purpose of a chain network. They are the stakeholders of a network.

Roles: Chain Transactor, Chain Validator

Network Owner

Owners are stakeholders of a network that can validate transactions. After a network is first launched, its proprietor (who then becomes an owner) will invite business partners to co-own the network (by assigning them validating nodes). Any new owner added to a network must be approved by its existing owners.

Roles: Chain Transactor, Chain Validator

Network Member

Members are participants of a blockchain network that cannot validate transactions but has the right to add users to the network.

Roles: Chain Transactor, Chain Member

Network Users

End users of a network are also solution users. Unlike network owners and members, users do not own nodes. They transact with the network through an entry point offered by a member or an owner node.

Roles: Chain Transactor

Network Auditors

Individuals or organizations with the permission to interrogate transactions.

Roles: Chain Auditor

CHAPTER 6

Business Network

Industry Network

A chain network that services solutions built for a particular industry.

Regional Industry Network

A chain network that services applications built for a particular industry and region.

Application Network

A chain network that only services a single solution.

Main Chain

A business network; each main chain operates one or multiple applications/solutions validated by the same group of organizations.

Confidential Chain

A special purpose chain created to run confidential business logic that is only accessible by contract stakeholders.

Network Management

Owner Registration

The process of registering and inviting new owner(s) to a blockchain network. Approval from existing network owners is required when adding or deleting a participant with ownership right

Member Registration

The process of registering and inviting new network members to a blockchain network.

User Registration

The process of registering new users to a blockchain network. Both members and owners can register users on their own behalf as long as they follow the policy of their network.

CHAPTER 8

Transactions

Deployment Transaction

Transactions that deploy a new chaincode to a chain.

Invocation Transaction

Transactions that invoke a function on a chaincode.

Public Transaction

A transaction with its payload in the open. Anyone with access to a chain network can interrogate the details of public transactions.

Confidential Transaction

A transaction with its payload cryptographically hidden such that no one besides the stakeholders of a transaction can interrogate its content.

Confidential Chaincode Transaction

A transaction with its payload encrypted such that only validators can decrypt them. Chaincode confidentiality is determined during deploy time. If a chaincode is deployed as a confidential chaincode, then the payload of all subsequent invocation transactions to that chaincode will be encrypted.

Inter-Network Transaction

Transactions between two business networks (main chains).

Inter-Chain Transaction

Transactions between confidential chains and main chains. Chaincodes in a confidential chain can trigger transactions on one or multiple main chain(s).

Network Entities

Application Backend

Purpose: Backend application service that supports associated mobile and/or browser based applications.

Key Roles:

1.
   ```
   Manages end users and registers them with the membership service
   ```

2.
   ```
   Initiates transactions requests, and sends the requests to a node
   ```

   Owned by: Solution Provider, Network Proprietor

   Non Validating Node (Peer)

   Purpose: Constructs transactions and forwards them to validating nodes. Peer nodes keep a copy of all transaction records so that solution providers can query them locally.

   Key Roles:

3.
   ```
   Manages and maintains user certificates issued by the membership service<p>
   ```

4.
   ```
   Constructs transactions and forwards them to validating nodes <p>
   ```

5.
   ```
   Maintains a local copy of the ledger, and allows application owners to query␣
   ↪information locally.
   ```

   Owned by: Solution Provider, Network Auditor

   Validating Node (Peer)

   Purpose: Creates and validates transactions, and maintains the state of chaincodes

   Key Roles:

6.
   ```
   Manages and maintains user certificates issued by membership service<p>
   ```

7.
   ```
   Creates transactions<p>
   ```

8.
   ```
   Executes and validates transactions with other validating nodes on the network<p>
   ```

9.
```
Maintains a local copy of ledger<p>
```

10.
```
Participates in consensus and updates ledger
```

Owned by: Network Proprietor, Solution Provider (if they belong to the same entity)

Membership Service

Purpose: Issues and manages the identity of end users and organizations

Key Roles:

11.
```
Issues enrollment certificate to each end user and organization<p>
```

12.
```
Issues transaction certificates associated to each end user and organization<p>
```

13.
```
Issues TLS certificates for secured communication between Hyperledger fabric␣
↪entities<p>
```

14.
```
Issues chain specific keys
```

Owned by: Third party service provider

Membership Service Components

Registration Authority

Assigns registration username & registration password pairs to network participants. This username/password pair will be used to acquire enrollment certificate from ECA.

Enrollment Certificate Authority (ECA)

Issues enrollment certificates (ECert) to network participants that have already registered with a membership service. ECerts are long term certificates used to identify individual entities participating in one or more networks.

Transaction Certificate Authority (TCA)

Issues transaction certificates (TCerts) to ECert owners. An infinite number of TCerts can be derived from each ECert. TCerts are used by network participants to send transactions. Depending on the level of security requirements, network participants may choose to use a new TCert for every transaction.

TLS-Certificate Authority (TLS-CA)

Issues TLS certificates to systems that transmit messages in a chain network. TLS certificates are used to secure the communication channel between systems.

CHAPTER 10

Hyperledger Fabric Entities

Public Chaincode

Chaincodes deployed by public transactions, these chaincodes can be invoked by any member of the network.

Confidential Chaincode

Chaincodes deployed by confidential transactions, these chaincodes can only be invoked by validating members (Chain validators) of the network.

Access Controlled Chaincode

Chaincodes deployed by confidential transactions that also embed the tokens of approved invokers. These invokers are also allowed to invoke confidential chaincodes even though they are not validators.

Chaincode-State

HPL provides state support; Chaincodes access internal state storage through state APIs. States are created and updated by transactions calling chaincode functions with state accessing logic.

Transaction List

All processed transactions are kept in the ledger in their original form (with payload encrypted for confidential transactions), so that network participants can interrogate past transactions to which they have access permissions.

Ledger Hash

A hash that captures the present snapshot of the ledger. It is a product of all validated transactions processed by the network since the genesis transaction.

DevOps Service

The frontal module on a node that provides APIs for clients to interact with their node and chain network. This module is also responsible to construct transactions, and work with the membership service component to receive and store all types of certificates and encryption keys in its storage.

Node Service

The main module on a node that is responsible to process transactions, deploy and execute chaincodes, maintain ledger data, and trigger the consensus process.

Consensus

The default consensus algorithm of Hyperledger fabric is an implementation of PBFT.

CHAPTER 11

Protocol Specification

Preface

This document is the protocol specification for a permissioned blockchain implementation for industry use-cases. It is not intended to be a complete explanation of the implementation, but rather a description of the interfaces and relationships between components in the system and the application.

Intended Audience

The intended audience for this specification includes the following groups:

- Blockchain vendors who want to implement blockchain systems that conform to this specification
- Tool developers who want to extend the capabilities of the fabric
- Application developers who want to leverage blockchain technologies to enrich their applications

---

Table of Contents

---

1. Introduction

This document specifies the principles, architecture, and protocol of a blockchain implementation suitable for industrial use-cases.

1.1 What is the fabric?

The fabric is a ledger of digital events, called transactions, shared among different participants, each having a stake in the system. The ledger can only be updated by consensus of the participants, and, once recorded, information can never be altered. Each recorded event is cryptographically verifiable with proof of agreement from the participants.

Transactions are secured, private, and confidential. Each participant registers with proof of identity to the network membership services to gain access to the system. Transactions are issued with derived certificates unlinkable to the individual participant, offering a complete anonymity on the network. Transaction content is encrypted with sophisticated key derivation functions to ensure only intended participants may see the content, protecting the confidentiality of the business transactions.

The ledger allows compliance with regulations as ledger entries are auditable in whole or in part. In collaboration with participants, auditors may obtain time-based certificates to allow viewing the ledger and linking transactions to provide an accurate assessment of the operations.

The fabric is an implementation of blockchain technology, where Bitcoin could be a simple application built on the fabric. It is a modular architecture allowing components to be plug-and-play by implementing this protocol specification. It features powerful container technology to host any main stream language for smart contracts development. Leveraging familiar and proven technologies is the motto of the fabric architecture.

1.2 Why the fabric?

Early blockchain technology serves a set of purposes but is often not well-suited for the needs of specific industries. To meet the demands of modern markets, the fabric is based on an industry-focused design that addresses the multiple and varied requirements of specific industry use cases, extending the learning of the pioneers in this field while also addressing issues such as scalability. The fabric provides a new approach to enable permissioned networks, privacy, and confidentially on multiple blockchain networks.

1.3 Terminology

The following terminology is defined within the limited scope of this specification to help readers understand clearly and precisely the concepts described here.

Transaction is a request to the blockchain to execute a function on the ledger. The function is implemented by a **chaincode**.

Transactor is an entity that issues transactions such as a client application.

Ledger is a sequence of cryptographically linked blocks, containing transactions and current **world state**.

World State is the collection of variables containing the results of executed transactions.

Chaincode is an application-level code (a.k.a. smart contract) stored on the ledger as a part of a transaction. Chaincode runs transactions that may modify the world state.

Validating Peer is a computer node on the network responsible for running consensus, validating transactions, and maintaining the ledger.

Non-validating Peer is a computer node on the network which functions as a proxy connecting transactors to the neighboring validating peers. A non-validating peer doesn't execute transactions but does verify them. It also hosts the event stream server and the REST service.

Permissioned Ledger is a blockchain network where each entity or node is required to be a member of the network. Anonymous nodes are not allowed to connect.

Privacy is required by the chain transactors to conceal their identities on the network. While members of the network may examine the transactions, the transactions can't be linked to the transactor without special privilege.

Confidentiality is the ability to render the transaction content inaccessible to anyone other than the stakeholders of the transaction.

Auditability of the blockchain is required, as business usage of blockchain needs to comply with regulations to make it easy for regulators to investigate transaction records.

2. Fabric

The fabric is made up of the core components described in the subsections below.

2.1 Architecture

The reference architecture is aligned in 3 categories: Membership, Blockchain, and Chaincode services. These categories are logical structures, not a physical depiction of partitioning of components into separate processes, address spaces or (virtual) machines.

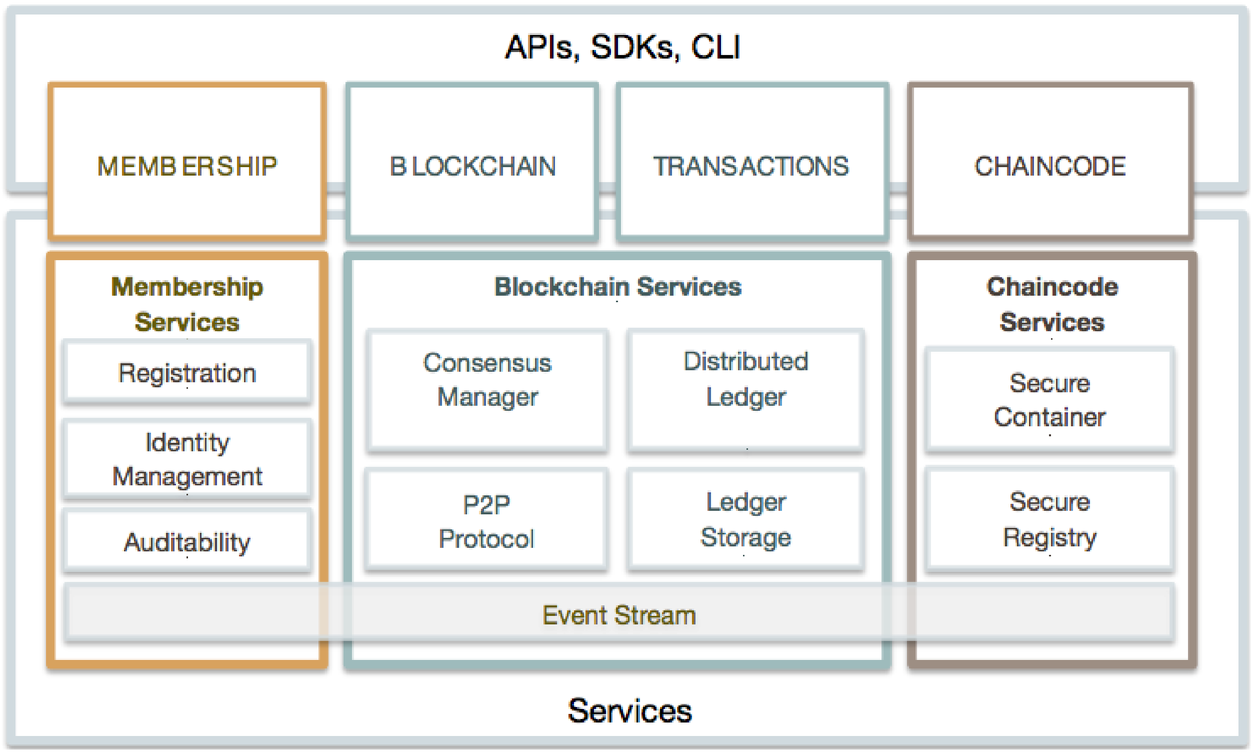


Fig. 11.1: Reference architecture

2.1.1 Membership Services

Membership provides services for managing identity, privacy, confidentiality and auditability on the network. In a non-permissioned blockchain, participation does not require authorization and all nodes can equally submit transactions and/or attempt to accumulate them into acceptable blocks, i.e. there are no distinctions of roles. Membership services combine elements of Public Key Infrastructure (PKI) and decentralization/consensus to transform a non-permissioned blockchain into a permissioned blockchain. In the latter, entities register in order to acquire long-term identity credentials (enrollment certificates), and may be distinguished according to entity type. In the case of users, such credentials enable the Transaction Certificate Authority (TCA) to issue pseudonymous credentials. Such credentials, i.e., transaction certificates, are used to authorize submitted transactions. Transaction certificates persist on the blockchain, and enable authorized auditors to cluster otherwise unlinkable transactions.

2.1.2 Blockchain Services

Blockchain services manage the distributed ledger through a peer-to-peer protocol, built on HTTP/2. The data structures are highly optimized to provide the most efficient hash algorithm for maintaining the world state replication. Different consensus (PBFT, Raft, PoW, PoS) may be plugged in and configured per deployment.

2.1.3 Chaincode Services

Chaincode services provides a secured and lightweight way to sandbox the chaincode execution on the validating nodes. The environment is a "locked down" and secured container along with a set of signed base images containing secure OS and chaincode language, runtime and SDK layers for Go, Java, and Node.js. Other languages can be enabled if required.

2.1.4 Events

Validating peers and chaincodes can emit events on the network that applications may listen for and take actions on. There is a set of pre-defined events, and chaincodes can generate custom events. Events are consumed by 1 or more event adapters. Adapters may further deliver events using other vehicles such as Web hooks or Kafka.

2.1.5 Application Programming Interface (API)

The primary interface to the fabric is a REST API and its variations over Swagger 2.0. The API allows applications to register users, query the blockchain, and to issue transactions. There is a set of APIs specifically for chaincode to interact with the stack to execute transactions and query transaction results.

2.1.6 Command Line Interface (CLI)

CLI includes a subset of the REST API to enable developers to quickly test chaincodes or query for status of transactions. CLI is implemented in Golang and operable on multiple OS platforms.

2.2 Topology

A deployment of the fabric can consist of a membership service, many validating peers, non-validating peers, and 1 or more applications. All of these components make up a chain. There can be multiple chains; each one having its own operating parameters and security requirements.

2.2.1 Single Validating Peer

Functionally, a non-validating peer is a subset of a validating peer; that is, every capability on a non-validating peer may be enabled on a validating peer, so the simplest network may consist of a single validating peer node. This configuration is most appropriate for a development environment, where a single validating peer may be started up during the edit-compile-debug cycle.

A single validating peer doesn't require consensus, and by default uses the `noops` plugin, which executes transactions as they arrive. This gives the developer an immediate feedback during development.

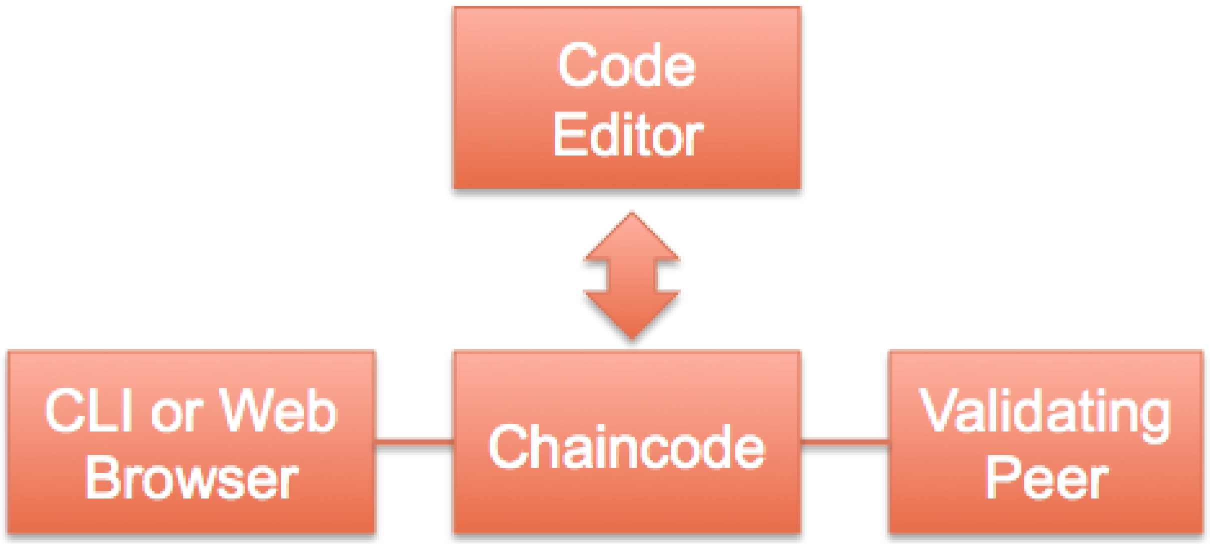


Fig. 11.2: Single Validating Peer

2.2.2 Multiple Validating Peers

Production or test networks should be made up of multiple validating and non-validating peers as necessary. Non-validating peers can take workload off the validating peers, such as handling API requests and processing events.

The validating peers form a mesh-network (every validating peer connects to every other validating peer) to disseminate information. A non-validating peer connects to a neighboring validating peer that it is allowed to connect to. Non-validating peers are optional since applications may communicate directly with validating peers.

2.2.3 Multichain

Each network of validating and non-validating peers makes up a chain. Many chains may be created to address different needs, similar to having multiple Web sites, each serving a different purpose.

3. Protocol

The fabric's peer-to-peer communication is built on gRPC, which allows bi-directional stream-based messaging. It uses Protocol Buffers to serialize data structures for data transfer between peers. Protocol buffers are a language-neutral, platform-neutral and extensible mechanism for serializing structured data. Data structures, messages, and services are described using proto3 language notation.

3.1 Message

Messages passed between nodes are encapsulated by `Message` proto structure, which consists of 4 types: Discovery, Transaction, Synchronization, and Consensus. Each type may define more subtypes embedded in the `payload`.

```
message Message {
   enum Type {
```

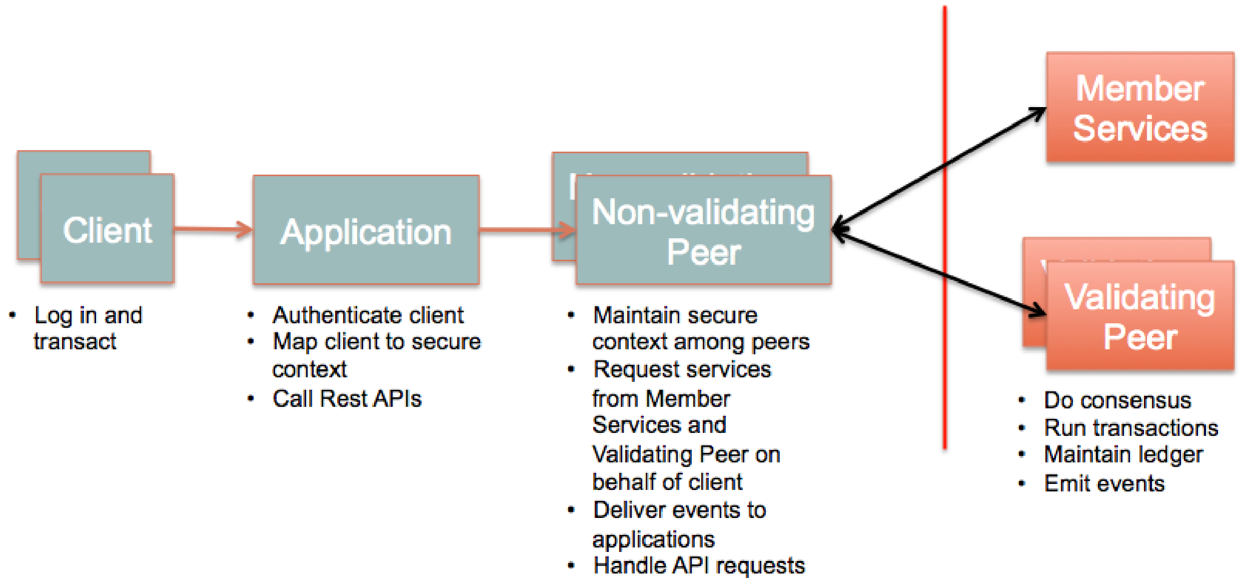


Fig. 11.3: Multiple Validating Peers

```
        UNDEFINED = 0;

        DISC_HELLO = 1;
        DISC_DISCONNECT = 2;
        DISC_GET_PEERS = 3;
        DISC_PEERS = 4;
        DISC_NEWMSG = 5;

        CHAIN_STATUS = 6;
        CHAIN_TRANSACTION = 7;
        CHAIN_GET_TRANSACTIONS = 8;
        CHAIN_QUERY = 9;

        SYNC_GET_BLOCKS = 11;
        SYNC_BLOCKS = 12;
        SYNC_BLOCK_ADDED = 13;

        SYNC_STATE_GET_SNAPSHOT = 14;
        SYNC_STATE_SNAPSHOT = 15;
        SYNC_STATE_GET_DELTAS = 16;
        SYNC_STATE_DELTAS = 17;

        RESPONSE = 20;
        CONSENSUS = 21;
    }
    Type type = 1;
    bytes payload = 2;
    google.protobuf.Timestamp timestamp = 3;
}
```

The `payload` is an opaque byte array containing other objects such as `Transaction` or `Response` depending on the type of the message. For example, if the `type` is `CHAIN_TRANSACTION`, the `payload` is a `Transaction` object.

3.1.1 Discovery Messages

Upon start up, a peer runs discovery protocol if `CORE_PEER_DISCOVERY_ROOTNODE` is specified. `CORE_PEER_DISCOVERY_ROOTNODE` is the IP address of another peer on the network (any peer) that serves as the starting point for discovering all the peers on the network. The protocol sequence begins with `DISC_HELLO`, whose `payload` is a `HelloMessage` object, containing its endpoint:

```
message HelloMessage {
  PeerEndpoint peerEndpoint = 1;
  uint64 blockNumber = 2;
}
message PeerEndpoint {
    PeerID ID = 1;
    string address = 2;
    enum Type {
      UNDEFINED = 0;
      VALIDATOR = 1;
      NON_VALIDATOR = 2;
    }
    Type type = 3;
    bytes pkiID = 4;
}

message PeerID {
    string name = 1;
}
```

Definition of fields:

- `PeerID` is any name given to the peer at start up or defined in the config file
- `PeerEndpoint` describes the endpoint and whether it's a validating or a non-validating peer
- `pkiID` is the cryptographic ID of the peer
- `address` is host or IP address and port of the peer in the format `ip:port`
- `blockNumber` is the height of the blockchain the peer currently has

If the block height received upon `DISC_HELLO` is higher than the current block height of the peer, it immediately initiates the synchronization protocol to catch up with the network.

After `DISC_HELLO`, peer sends `DISC_GET_PEERS` periodically to discover any additional peers joining the network. In response to `DISC_GET_PEERS`, a peer sends `DISC_PEERS` with `payload` containing an array of `PeerEndpoint`. Other discovery message types are not used at this point.

3.1.2 Transaction Messages

There are 3 types of transactions: Deploy, Invoke and Query. A deploy transaction installs the specified chaincode on the chain, while invoke and query transactions call a function of a deployed chaincode. Another type in consideration is Create transaction, where a deployed chaincode may be instantiated on the chain and is addressable. This type has not been implemented as of this writing.

3.1.2.1 Transaction Data Structure

Messages with type `CHAIN_TRANSACTION` or `CHAIN_QUERY` carry a `Transaction` object in the `payload`:

```
message Transaction {
    enum Type {
        UNDEFINED = 0;
        CHAINCODE_DEPLOY = 1;
        CHAINCODE_INVOKE = 2;
        CHAINCODE_QUERY = 3;
        CHAINCODE_TERMINATE = 4;
    }
    Type type = 1;
    string uuid = 5;
    bytes chaincodeID = 2;
    bytes payloadHash = 3;

    ConfidentialityLevel confidentialityLevel = 7;
    bytes nonce = 8;
    bytes cert = 9;
    bytes signature = 10;

    bytes metadata = 4;
    google.protobuf.Timestamp timestamp = 6;
}

message TransactionPayload {
    bytes payload = 1;
}

enum ConfidentialityLevel {
    PUBLIC = 0;
    CONFIDENTIAL = 1;
}
```

Definition of fields: - `type` - The type of the transaction, which is 1 of the following: - `UNDEFINED` - Reserved for future use. - `CHAINCODE_DEPLOY` - Represents the deployment of a new chaincode. - `CHAINCODE_INVOKE` - Represents a chaincode function execution that may read and modify the world state. - `CHAINCODE_QUERY` - Represents a chaincode function execution that may only read the world state. - `CHAINCODE_TERMINATE` - Marks a chaincode as inactive so that future functions of the chaincode can no longer be invoked. - `chaincodeID` - The ID of a chaincode which is a hash of the chaincode source, path to the source code, constructor function, and parameters. - `payloadHash` - Bytes defining the hash of `TransactionPayload.payload`. - `metadata` - Bytes defining any associated transaction metadata that the application may use. - `uuid` - A unique ID for the transaction. - `timestamp` - A timestamp of when the transaction request was received by the peer. - `confidentialityLevel` - Level of data confidentiality. There are currently 2 levels. Future releases may define more levels. - `nonce` - Used for security. - `cert` - Certificate of the transactor. - `signature` - Signature of the transactor. - `TransactionPayload.payload` - Bytes defining the payload of the transaction. As the payload can be large, only the payload hash is included directly in the transaction message.

More detail on transaction security can be found in section 4.

3.1.2.2 Transaction Specification

A transaction is always associated with a chaincode specification which defines the chaincode and the execution environment such as language and security context. Currently there is an implementation that uses Golang for writing chaincode. Other languages may be added in the future.

```
message ChaincodeSpec {
    enum Type {
        UNDEFINED = 0;
        GOLANG = 1;
        NODE = 2;
    }
    Type type = 1;
    ChaincodeID chaincodeID = 2;
    ChaincodeInput ctorMsg = 3;
    int32 timeout = 4;
    string secureContext = 5;
    ConfidentialityLevel confidentialityLevel = 6;
    bytes metadata = 7;
}

message ChaincodeID {
    string path = 1;
    string name = 2;
}

message ChaincodeInput {
    string function = 1;
    repeated string args  = 2;
}
```

Definition of fields: - `chaincodeID` - The chaincode source code path and name. - `ctorMsg` - Function name and argument parameters to call. - `timeout` - Time in milliseconds to execute the transaction. - `confidentialityLevel` - Confidentiality level of this transaction. - `secureContext` - Security context of the transactor. - `metadata` - Any data the application wants to pass along.

The peer, receiving the `chaincodeSpec`, wraps it in an appropriate transaction message and broadcasts to the network.

3.1.2.3 Deploy Transaction

Transaction `type` of a deploy transaction is `CHAINCODE_DEPLOY` and the payload contains an object of `ChaincodeDeploymentSpec`.

```
message ChaincodeDeploymentSpec {
    ChaincodeSpec chaincodeSpec = 1;
    google.protobuf.Timestamp effectiveDate = 2;
    bytes codePackage = 3;
}
```

Definition of fields: - `chaincodeSpec` - See section 3.1.2.2, above. - `effectiveDate` - Time when the chaincode is ready to accept invocations. - `codePackage` - gzip of the chaincode source.

The validating peers always verify the hash of the `codePackage` when they deploy the chaincode to make sure the package has not been tampered with since the deploy transaction entered the network.

3.1.2.4 Invoke Transaction

Transaction `type` of an invoke transaction is `CHAINCODE_INVOKE` and the `payload` contains an object of `ChaincodeInvocationSpec`.

```
message ChaincodeInvocationSpec {
    ChaincodeSpec chaincodeSpec = 1;
}
```

3.1.2.5 Query Transaction

A query transaction is similar to an invoke transaction, but the message `type` is `CHAINCODE_QUERY`.

3.1.3 Synchronization Messages

Synchronization protocol starts with discovery, described above in section 3.1.1, when a peer realizes that it's behind or its current block is not the same with others. A peer broadcasts either `SYNC_GET_BLOCKS`, `SYNC_STATE_GET_SNAPSHOT`, or `SYNC_STATE_GET_DELTAS` and receives `SYNC_BLOCKS`, `SYNC_STATE_SNAPSHOT`, or `SYNC_STATE_DELTAS` respectively.

The installed consensus plugin (e.g. pbft) dictates how synchronization protocol is being applied. Each message is designed for a specific situation:

SYNC_GET_BLOCKS requests for a range of contiguous blocks expressed in the message `payload`, which is an object of `SyncBlockRange`. The correlationId specified is included in the `SyncBlockRange` of any replies to this message.

```
message SyncBlockRange {
    uint64 correlationId = 1;
    uint64 start = 2;
    uint64 end = 3;
}
```

A receiving peer responds with a `SYNC_BLOCKS` message whose `payload` contains an object of `SyncBlocks`

```
message SyncBlocks {
    SyncBlockRange range = 1;
    repeated Block blocks = 2;
}
```

The `start` and `end` indicate the starting and ending blocks inclusively. The order in which blocks are returned is defined by the `start` and `end` values. For example, if `start` =3 and `end` =5, the order of blocks will be 3, 4, 5. If `start` =5 and `end` =3, the order will be 5, 4, 3.

SYNC_STATE_GET_SNAPSHOT requests for the snapshot of the current world state. The `payload` is an object of `SyncStateSnapshotRequest`

```
message SyncStateSnapshotRequest {
  uint64 correlationId = 1;
}
```

The `correlationId` is used by the requesting peer to keep track of the response messages. A receiving peer replies with `SYNC_STATE_SNAPSHOT` message whose `payload` is an instance of `SyncStateSnapshot`

```
message SyncStateSnapshot {
    bytes delta = 1;
    uint64 sequence = 2;
    uint64 blockNumber = 3;
    SyncStateSnapshotRequest request = 4;
}
```

This message contains the snapshot or a chunk of the snapshot on the stream, and in which case, the sequence indicate the order starting at 0. The terminating message will have len(delta) == 0.

SYNC_STATE_GET_DELTAS requests for the state deltas of a range of contiguous blocks. By default, the Ledger maintains 500 transition deltas. A delta(j) is a state transition between block(i) and block(j) where i = j-1. The message `payload` contains an instance of `SyncStateDeltasRequest`

```
message SyncStateDeltasRequest {
    SyncBlockRange range = 1;
}
```

A receiving peer responds with `SYNC_STATE_DELTAS` , whose `payload` is an instance of `SyncStateDeltas`

```
message SyncStateDeltas {
    SyncBlockRange range = 1;
    repeated bytes deltas = 2;
}
```

A delta may be applied forward (from i to j) or backward (from j to i) in the state transition.

3.1.4 Consensus Messages

Consensus deals with transactions, so a `CONSENSUS` message is initiated internally by the consensus framework when it receives a `CHAIN_TRANSACTION` message. The framework converts `CHAIN_TRANSACTION` into `CONSENSUS` then broadcasts to the validating nodes with the same `payload` . The consensus plugin receives this message and process according to its internal algorithm. The plugin may create custom subtypes to manage consensus finite state machine. See section 3.4 for more details.

3.2 Ledger

The ledger consists of two primary pieces, the blockchain and the world state. The blockchain is a series of linked blocks that is used to record transactions within the ledger. The world state is a key-value database that chaincodes may use to store state when executed by a transaction.

3.2.1 Blockchain

3.2.1.1 Block

The blockchain is defined as a linked list of blocks as each block contains the hash of the previous block in the chain. The two other important pieces of information that a block contains are the list of transactions contained within the block and the hash of the world state after executing all transactions in the block.

```
message Block {
  version = 1;
  google.protobuf.Timestamp timestamp = 2;
  bytes transactionsHash = 3;
  bytes stateHash = 4;
  bytes previousBlockHash = 5;
  bytes consensusMetadata = 6;
  NonHashData nonHashData = 7;
}

message BlockTransactions {
  repeated Transaction transactions = 1;
}
```

- `version` - Version used to track any protocol changes.
- `timestamp` - The timestamp to be filled in by the block proposer.

- `transactionsHash` - The merkle root hash of the block's transactions.
- `stateHash` - The merkle root hash of the world state.
- `previousBlockHash` - The hash of the previous block.
- `consensusMetadata` - Optional metadata that the consensus may include in a block.
- `nonHashData` - A `NonHashData` message that is set to nil before computing the hash of the block, but stored as part of the block in the database.
- `BlockTransactions.transactions` - An array of Transaction messages. Transactions are not included in the block directly due to their size.

3.2.1.2 Block Hashing

- The `previousBlockHash` hash is calculated using the following algorithm.

1. Serialize the Block message to bytes using the protocol buffer library.
2. Hash the serialized block message to 512 bits of output using the SHA3 SHAKE256 algorithm as described in FIPS 202.

- The `transactionHash` is the root of the transaction merkle tree. Defining the merkle tree implementation is a TODO.
- The `stateHash` is defined in section 3.2.2.1.

3.2.1.3 NonHashData

The NonHashData message is used to store block metadata that is not required to be the same value on all peers. These are suggested values.

```
message NonHashData {
  google.protobuf.Timestamp localLedgerCommitTimestamp = 1;
  repeated TransactionResult transactionResults = 2;
}

message TransactionResult {
  string uuid = 1;
  bytes result = 2;
  uint32 errorCode = 3;
  string error = 4;
}
```

- `localLedgerCommitTimestamp` - A timestamp indicating when the block was commited to the local ledger.
- `TransactionResult` - An array of transaction results.
- `TransactionResult.uuid` - The ID of the transaction.
- `TransactionResult.result` - The return value of the transaction.
- `TransactionResult.errorCode` - A code that can be used to log errors associated with the transaction.
- `TransactionResult.error` - A string that can be used to log errors associated with the transaction.

3.2.1.4 Transaction Execution

A transaction defines either the deployment of a chaincode or the execution of a chaincode. All transactions within a block are run before recording a block in the ledger. When chaincodes execute, they may modify the world state. The hash of the world state is then recorded in the block.

3.2.2 World State

The *world state* of a peer refers to the collection of the *states* of all the deployed chaincodes. Further, the state of a chaincode is represented as a collection of key-value pairs. Thus, logically, the world state of a peer is also a collection of key-value pairs where key consists of a tuple `{chaincodeID,ckey}` . Here, we use the term `key` to represent a key in the world state i.e., a tuple `{chaincodeID,ckey}` and we use the term `cKey` to represent a unique key within a chaincode.

For the purpose of the description below, `chaincodeID` is assumed to be a valid utf8 string and `ckey` and the `value` can be a sequence of one or more arbitrary bytes.

3.2.2.1 Hashing the world state

During the functioning of a network, many occasions such as committing transactions and synchronizing peers may require computing a crypto-hash of the world state observed by a peer. For instance, the consensus protocol may require to ensure that a *minimum* number of peers in the network observe the same world state.

Since, computing the crypto-hash of the world state could be an expensive operation, this is highly desirable to organize the world state such that it enables an efficient crypto-hash computation of the world state when a change occurs in the world state. Further, different organization designs may be suitable under different workloads conditions.

Because the fabric is expected to function under a variety of scenarios leading to different workloads conditions, a pluggable mechanism is supported for organizing the world state.

3.2.2.1.1 Bucket-tree

Bucket-tree is one of the implementations for organizing the world state. For the purpose of the description below, a key in the world state is represented as a concatenation of the two components (`chaincodeID` and `ckey`) separated by a `nil` byte i.e., `key` = `chaincodeID` +`nil` +`cKey` .

This method models a *merkle-tree* on top of buckets of a *hash table* in order to compute the crypto-hash of the *world state*.

At the core of this method, the *key-values* of the world state are assumed to be stored in a hash-table that consists of a pre-decided number of buckets (`numBuckets`). A hash function (`hashFunction`) is employed to determine the bucket number that should contain a given key. Please note that the `hashFunction` does not represent a crypto-hash method such as SHA3, rather this is a regular programming language hash function that decides the bucket number for a given key.

For modeling the merkle-tree, the ordered buckets act as leaf nodes of the tree - lowest numbered bucket being the left most leaf node in the tree. For constructing the second-last level of the tree, a pre-decided number of leaf nodes (`maxGroupingAtEachLevel`), starting from left, are grouped together and for each such group, a node is inserted at the second-last level that acts as a common parent for all the leaf nodes in the group. Note that the number of children for the last parent node may be less than `maxGroupingAtEachLevel` . This grouping method of constructing the next higher level is repeated until the root node of the tree is constructed.

An example setup with configuration `{numBuckets=10009 and maxGroupingAtEachLevel=10}` will result in a tree with number of nodes at different level as depicted in the following table.

Level	Number of nodes
0	1
1	2
2	11
3	101
4	1001
5	10009

For computing the crypto-hash of the world state, the crypto-hash of each bucket is computed and is assumed to be the crypto-hash of leaf-nodes of the merkle-tree. In order to compute crypto-hash of a bucket, the key-values present in the bucket are first serialized and crypto-hash function is applied on the serialized bytes. For serializing the key-values of a bucket, all the key-values with a common chaincodeID prefix are serialized separately and then appending together, in the ascending order of chaincodeIDs. For serializing the key-values of a chaincodeID, the following information is concatenated: 1. Length of chaincodeID (number of bytes in the chaincodeID) - The utf8 bytes of the chaincodeID - Number of key-values for the chaincodeID - For each key-value (in sorted order of the ckey) - Length of the ckey - ckey bytes - Length of the value - value bytes

For all the numeric types in the above list of items (e.g., Length of chaincodeID), protobuf's varint encoding is assumed to be used. The purpose of the above encoding is to achieve a byte representation of the key-values within a bucket that can not be arrived at by any other combination of key-values and also to reduce the overall size of the serialized bytes.

For example, consider a bucket that contains three key-values namely, `chaincodeID1_key1:value1,chaincodeID1_key2:value2,and chaincodeID2_key1:value1` . The serialized bytes for the bucket would logically look as - `12 + chaincodeID1 + 2 + 4 + key1 + 6 + value1 + 4 + key2 + 6 + value2 + 12 + chaincodeID2 + 1 + 4 + key1 + 6 + value1`

If a bucket has no key-value present, the crypto-hash is considered as `nil` .

The crypto-hash of an intermediate node and root node are computed just like in a standard merkle-tree i.e., applying a crypto-hash function on the bytes obtained by concatenating the crypto-hash of all the children nodes, from left to right. Further, if a child has a crypto-hash as `nil` , the crypto-hash of the child is omitted when concatenating the children crypto-hashes. If the node has a single child, the crypto-hash of the child is assumed to be the crypto-hash of the node. Finally, the crypto-hash of the root node is considered as the crypto-hash of the world state.

The above method offers performance benefits for computing crypto-hash when a few key-values change in the state. The major benefits include - Computation of crypto-hashes of the unchanged buckets can be skipped - The depth and breadth of the merkle-tree can be controlled by configuring the parameters `numBuckets` and `maxGroupingAtEachLevel` . Both depth and breadth of the tree has different implication on the performance cost incurred by and resource demand of different resources (namely - disk I/O, storage, and memory)

In a particular deployment, all the peer nodes are expected to use same values for the configurations `numBuckets,maxGroupingAtEachLevel,and hashFunction` . Further, if any of these configurations are to be changed at a later stage, the configurations should be changed on all the peer nodes so that the comparison of crypto-hashes across peer nodes is meaningful. Also, this may require to migrate the existing data based on the implementation. For example, an implementation is expected to store the last computed crypto-hashes for all the nodes in the tree which would need to be recalculated.

3.3 Chaincode

Chaincode is an application-level code deployed as a transaction (see section 3.1.2) to be distributed to the network and managed by each validating peer as isolated sandbox. Though any virtualization technology can support the sandbox, currently Docker container is utilized to run the chaincode. The protocol described in this section enables different virtualization support implementation to plug and play.

3.3.1 Virtual Machine Instantiation

A virtual machine implements the VM interface:

```
type VM interface {
    build(ctxt context.Context, id string, args []string, env []string, attachstdin␣
↪bool, attachstdout bool, reader io.Reader) error
    start(ctxt context.Context, id string, args []string, env []string, attachstdin␣
↪bool, attachstdout bool) error
    stop(ctxt context.Context, id string, timeout uint, dontkill bool, dontremove␣
↪bool) error
}
```

The fabric instantiates the VM when it processes a Deploy transaction or other transactions on the chaincode while the VM for that chaincode is not running (either crashed or previously brought down due to inactivity). Each chaincode image is built by the `build` function, started by `start` and stopped by `stop` function.

Once the chaincode container is up, it makes a gRPC connection back to the validating peer that started the chaincode, and that establishes the channel for Invoke and Query transactions on the chaincode.

3.3.2 Chaincode Protocol

Communication between a validating peer and its chaincodes is based on a bidirectional gRPC stream. There is a shim layer on the chaincode container to handle the message protocol between the chaincode and the validating peer using protobuf message.

```
message ChaincodeMessage {

    enum Type {
        UNDEFINED = 0;
        REGISTER = 1;
        REGISTERED = 2;
        INIT = 3;
        READY = 4;
        TRANSACTION = 5;
        COMPLETED = 6;
        ERROR = 7;
        GET_STATE = 8;
        PUT_STATE = 9;
        DEL_STATE = 10;
        INVOKE_CHAINCODE = 11;
        INVOKE_QUERY = 12;
        RESPONSE = 13;
        QUERY = 14;
        QUERY_COMPLETED = 15;
        QUERY_ERROR = 16;
        RANGE_QUERY_STATE = 17;
    }

    Type type = 1;
    google.protobuf.Timestamp timestamp = 2;
    bytes payload = 3;
    string uuid = 4;
}
```

Definition of fields: - `Type` is the type of the message. - `payload` is the payload of the message. Each payload depends on the `Type`. - `uuid` is a unique identifier of the message.

The message types are described in the following sub-sections.

A chaincode implements the `Chaincode` interface, which is called by the validating peer when it processes Deploy, Invoke or Query transactions.

```
type Chaincode interface {
i   Init(stub *ChaincodeStub, function string, args []string) ([]byte, error)
    Invoke(stub *ChaincodeStub, function string, args []string) ([]byte, error)
    Query(stub *ChaincodeStub, function string, args []string) ([]byte, error)
}
```

`Init` , `Invoke` and `Query` functions take `function` and `args` as parameters to be used by those methods to support a variety of transactions. `Init` is a constructor function, which will only be invoked by the Deploy transaction. The `Query` function is not allowed to modify the state of the chaincode; it can only read and calculate the return value as a byte array.

3.3.2.1 Chaincode Deploy

Upon deploy (chaincode container is started), the shim layer sends a one time `REGISTER` message to the validating peer with the `payload` containing the `ChaincodeID` . The validating peer responds with `REGISTERED` or `ERROR` on success or failure respectively. The shim closes the connection and exits if it receives an `ERROR` .

After registration, the validating peer sends `INIT` with the `payload` containing a `ChaincodeInput` object. The shim calls the `Init` function with the parameters from the `ChaincodeInput` , enabling the chaincode to perform any initialization, such as setting up the persistent state.

The shim responds with `RESPONSE` or `ERROR` message depending on the returned value from the chaincode `Init` function. If there are no errors, the chaincode initialization is complete and is ready to receive Invoke and Query transactions.

3.3.2.2 Chaincode Invoke

When processing an invoke transaction, the validating peer sends a `TRANSACTION` message to the chaincode container shim, which in turn calls the chaincode `Invoke` function, passing the parameters from the `ChaincodeInput` object. The shim responds to the validating peer with `RESPONSE` or `ERROR` message, indicating the completion of the function. If `ERROR` is received, the `payload` contains the error message generated by the chaincode.

3.3.2.3 Chaincode Query

Similar to an invoke transaction, when processing a query, the validating peer sends a `QUERY` message to the chaincode container shim, which in turn calls the chaincode `Query` function, passing the parameters from the `ChaincodeInput` object. The `Query` function may return a state value or an error, which the shim forwards to the validating peer using `RESPONSE` or `ERROR` messages respectively.

3.3.2.4 Chaincode State

Each chaincode may define its own persistent state variables. For example, a chaincode may create assets such as TVs, cars, or stocks using state variables to hold the assets attributes. During `Invoke` function processing, the chaincode may update the state variables, for example, changing an asset owner. A chaincode manipulates the state variables by using the following message types:

PUT_STATE

Chaincode sends a `PUT_STATE` message to persist a key-value pair, with the `payload` containing `PutStateInfo` object.

```
message PutStateInfo {
    string key = 1;
    bytes value = 2;
}
```

GET_STATE

Chaincode sends a `GET_STATE` message to retrieve the value whose key is specified in the `payload`.

DEL_STATE

Chaincode sends a `DEL_STATE` message to delete the value whose key is specified in the `payload`.

RANGE_QUERY_STATE

Chaincode sends a `RANGE_QUERY_STATE` message to get a range of values. The message `payload` contains a `RangeQueryStateInfo` object.

```
message RangeQueryState {
    string startKey = 1;
    string endKey = 2;
}
```

The `startKey` and `endKey` are inclusive and assumed to be in lexical order. The validating peer responds with `RESPONSE` message whose `payload` is a `RangeQueryStateResponse` object.

```
message RangeQueryStateResponse {
    repeated RangeQueryStateKeyValue keysAndValues = 1;
    bool hasMore = 2;
    string ID = 3;
}
message RangeQueryStateKeyValue {
    string key = 1;
    bytes value = 2;
}
```

If `hasMore=true` in the response, this indicates that additional keys are available in the requested range. The chaincode can request the next set of keys and values by sending a `RangeQueryStateNext` message with an ID that matches the ID returned in the response.

```
message RangeQueryStateNext {
    string ID = 1;
}
```

When the chaincode is finished reading from the range, it should send a `RangeQueryStateClose` message with the ID it wishes to close.

```
message RangeQueryStateClose {
  string ID = 1;
}
```

INVOKE_CHAINCODE

Chaincode may call another chaincode in the same transaction context by sending an `INVOKE_CHAINCODE` message to the validating peer with the `payload` containing a `ChaincodeSpec` object.

QUERY_CHAINCODE

Chaincode may query another chaincode in the same transaction context by sending a `QUERY_CHAINCODE` message with the `payload` containing a `ChaincodeSpec` object.

3.4 Pluggable Consensus Framework

The consensus framework defines the interfaces that every consensus *plugin* implements:

- `consensus.Consenter` : interface that allows consensus plugin to receive messages from the network.
- `consensus.CPI` : *Consensus Programming Interface* (`CPI`) is used by consensus plugin to interact with rest of the stack. This interface is split in two parts:
  - `consensus.Communicator` : used to send (broadcast and unicast) messages to other validating peers.
  - `consensus.LedgerStack` : which is used as an interface to the execution framework as well as the ledger.

As described below in more details, `consensus.LedgerStack` encapsulates, among other interfaces, the `consensus.Executor` interface, which is the key part of the consensus framework. Namely, `consensus.Executor` interface allows for a (batch of) transaction to be started, executed, rolled back if necessary, previewed, and potentially committed. A particular property that every consensus plugin needs to satisfy is that batches (blocks) of transactions are committed to the ledger (via `consensus.Executor.CommitTxBatch`) in total order across all validating peers (see `consensus.Executor` interface description below for more details).

Currently, consensus framework consists of 3 packages `consensus` , `controller` , and `helper` . The primary reason for `controller` and `helper` packages is to avoid "import cycle" in Go (golang) and minimize code changes for plugin to update.

- `controller` package specifies the consensus plugin used by a validating peer.
- `helper` package is a shim around a consensus plugin that helps it interact with the rest of the stack, such as maintaining message handlers to other peers.

There are 2 consensus plugins provided: `pbft` and `noops` :

- `pbft` package contains consensus plugin that implements the *PBFT* [1] consensus protocol. See section 5 for more detail.
- `noops` is a ''dummy" consensus plugin for development and test purposes. It doesn't perform consensus but processes all consensus messages. It also serves as a good simple sample to start learning how to code a consensus plugin.

3.4.1 `Consenter` interface

Definition:

```
type Consenter interface {
    RecvMsg(msg *pb.Message) error
}
```

The plugin's entry point for (external) client requests, and consensus messages generated internally (i.e. from the consensus module) during the consensus process. The `controller.NewConsenter` creates the plugin `Consenter` . `RecvMsg` processes the incoming transactions in order to reach consensus.

See `helper.HandleMessage` below to understand how the peer interacts with this interface.

3.4.2 `CPI` interface

Definition:

```
type CPI interface {
    Inquirer
    Communicator
    SecurityUtils
    LedgerStack
}
```

`CPI` allows the plugin to interact with the stack. It is implemented by the `helper.Helper` object. Recall that this object:

1. Is instantiated when the `helper.NewConsensusHandler` is called.
2. Is accessible to the plugin author when they construct their plugin's `consensus.Consenter` object.

3.4.3 `Inquirer` interface

Definition:

```
type Inquirer interface {
        GetNetworkInfo() (self *pb.PeerEndpoint, network []*pb.PeerEndpoint, err␣
↪error)
        GetNetworkHandles() (self *pb.PeerID, network []*pb.PeerID, err error)
}
```

This interface is a part of the `consensus.CPI` interface. It is used to get the handles of the validating peers in the network (`GetNetworkHandles`) as well as details about the those validating peers (`GetNetworkInfo`):

Note that the peers are identified by a `pb.PeerID` object. This is a protobuf message (in the `protos` package), currently defined as (notice that this definition will likely be modified):

```
message PeerID {
    string name = 1;
}
```

3.4.4 `Communicator` interface

Definition:

```
type Communicator interface {
    Broadcast(msg *pb.Message) error
    Unicast(msg *pb.Message, receiverHandle *pb.PeerID) error
}
```

This interface is a part of the `consensus.CPI` interface. It is used to communicate with other peers on the network (`helper.Broadcast`, `helper.Unicast`):

3.4.5 `SecurityUtils` interface

Definition:

```
type SecurityUtils interface {
        Sign(msg []byte) ([]byte, error)
        Verify(peerID *pb.PeerID, signature []byte, message []byte) error
}
```

This interface is a part of the `consensus.CPI` interface. It is used to handle the cryptographic operations of message signing (`Sign`) and verifying signatures (`Verify`)

3.4.6 `LedgerStack` interface

Definition:

```
type LedgerStack interface {
    Executor
    Ledger
    RemoteLedgers
}
```

A key member of the `CPI` interface, `LedgerStack` groups interaction of consensus with the rest of the fabric, such as the execution of transactions, querying, and updating the ledger. This interface supports querying the local blockchain and state, updating the local blockchain and state, and querying the blockchain and state of other nodes in the consensus network. It consists of three parts: `Executor`, `Ledger` and `RemoteLedgers` interfaces. These are described in the following.

3.4.7 `Executor` interface

Definition:

```
type Executor interface {
    BeginTxBatch(id interface{}) error
    ExecTXs(id interface{}, txs []*pb.Transaction) ([]byte, []error)
    CommitTxBatch(id interface{}, transactions []*pb.Transaction, transactionsResults␣
↪[]*pb.TransactionResult, metadata []byte) error
    RollbackTxBatch(id interface{}) error
    PreviewCommitTxBatchBlock(id interface{}, transactions []*pb.Transaction,␣
↪metadata []byte) (*pb.Block, error)
}
```

The executor interface is the most frequently utilized portion of the `LedgerStack` interface, and is the only piece which is strictly necessary for a consensus network to make progress. The interface allows for a transaction to be started, executed, rolled back if necessary, previewed, and potentially committed. This interface is comprised of the following methods.

3.4.7.1 Beginning a transaction batch

```
BeginTxBatch(id interface{}) error
```

This call accepts an arbitrary `id`, deliberately opaque, as a way for the consensus plugin to ensure only the transactions associated with this particular batch are executed. For instance, in the pbft implementation, this `id` is the an encoded hash of the transactions to be executed.

3.4.7.2 Executing transactions

```
ExecTXs(id interface{}, txs []*pb.Transaction) ([]byte, []error)
```

This call accepts an array of transactions to execute against the current state of the ledger and returns the current state hash in addition to an array of errors corresponding to the array of transactions. Note that a transaction resulting in an error has no effect on whether a transaction batch is safe to commit. It is up to the consensus plugin to determine the behavior which should occur when failing transactions are encountered. This call is safe to invoke multiple times.

3.4.7.3 Committing and rolling-back transactions

```
RollbackTxBatch(id interface{}) error
```

This call aborts an execution batch. This will undo the changes to the current state, and restore the ledger to its previous state. It concludes the batch begun with `BeginBatchTx` and a new one must be created before executing any transactions.

```
PreviewCommitTxBatchBlock(id interface{}, transactions []*pb.Transaction, metadata␣
↪[]byte) (*pb.Block, error)
```

This call is most useful for consensus plugins which wish to test for non-deterministic transaction execution. The hashable portions of the block returned are guaranteed to be identical to the block which would be committed if `CommitTxBatch` were immediately invoked. This guarantee is violated if any new transactions are executed.

```
CommitTxBatch(id interface{}, transactions []*pb.Transaction, transactionsResults␣
↪[]*pb.TransactionResult, metadata []byte) error
```

This call commits a block to the blockchain. Blocks must be committed to a blockchain in total order. `CommitTxBatch` concludes the transaction batch, and a new call to `BeginTxBatch` must be made before any new transactions are executed and committed.

3.4.8 `Ledger` interface

Definition:

```
type Ledger interface {
    ReadOnlyLedger
    UtilLedger
    WritableLedger
}
```

`Ledger` interface is intended to allow the consensus plugin to interrogate and possibly update the current state and blockchain. It is comprised of the three interfaces described below.

3.4.8.1 `ReadOnlyLedger` interface

Definition:

```
type ReadOnlyLedger interface {
    GetBlock(id uint64) (block *pb.Block, err error)
    GetCurrentStateHash() (stateHash []byte, err error)
    GetBlockchainSize() (uint64, error)
}
```

```
```

`ReadOnlyLedger` interface is intended to query the local copy of the ledger without the possibility of modifying it. It is comprised of the following functions.

```
GetBlockchainSize() (uint64, error)
```

This call returns the current length of the blockchain ledger. In general, this function should never fail, though in the unlikely event that this occurs, the error is passed to the caller to decide what if any recovery is necessary. The block with the highest number will have block number `GetBlockchainSize()-1` .

Note that in the event that the local copy of the blockchain ledger is corrupt or incomplete, this call will return the highest block number in the chain, plus one. This allows for a node to continue operating from the current state/block even when older blocks are corrupt or missing.

```
GetBlock(id uint64) (block *pb.Block, err error)
```

This call returns the block from the blockchain with block number `id` . In general, this call should not fail, except when the block queried exceeds the current blocklength, or when the underlying blockchain has somehow become corrupt. A failure of `GetBlock` has a possible resolution of using the state transfer mechanism to retrieve it.

```
GetCurrentStateHash() (stateHash []byte, err error)
```

This call returns the current state hash for the ledger. In general, this function should never fail, though in the unlikely event that this occurs, the error is passed to the caller to decide what if any recovery is necessary.

3.4.8.2 `UtilLedger` interface

Definition:

```
type UtilLedger interface {
    HashBlock(block *pb.Block) ([]byte, error)
    VerifyBlockchain(start, finish uint64) (uint64, error)
}
```

`UtilLedger` interface defines some useful utility functions which are provided by the local ledger. Overriding these functions in a mock interface can be useful for testing purposes. This interface is comprised of two functions.

```
HashBlock(block *pb.Block) ([]byte, error)
```

Although `*pb.Block` has a `GetHash` method defined, for mock testing, overriding this method can be very useful. Therefore, it is recommended that the `GetHash` method never be directly invoked, but instead invoked via this `UtilLedger.HashBlock` interface. In general, this method should never fail, but the error is still passed to the caller to decide what if any recovery is appropriate.

```
VerifyBlockchain(start, finish uint64) (uint64, error)
```

This utility method is intended for verifying large sections of the blockchain. It proceeds from a high block `start` to a lower block `finish` , returning the block number of the first block whose `PreviousBlockHash` does not match the block hash of the previous block as well as an error. Note, this generally indicates the last good block number, not the first bad block number.

3.4.8.3 `WritableLedger` interface

Definition:

```
type WritableLedger interface {
    PutBlock(blockNumber uint64, block *pb.Block) error
    ApplyStateDelta(id interface{}, delta *statemgmt.StateDelta) error
    CommitStateDelta(id interface{}) error
    RollbackStateDelta(id interface{}) error
    EmptyState() error
}
```

`WritableLedger` interface allows for the caller to update the blockchain. Note that this is *NOT* intended for use in normal operation of a consensus plugin. The current state should be modified by executing transactions using the `Executor` interface, and new blocks will be generated when transactions are committed. This interface is instead intended primarily for state transfer or corruption recovery. In particular, functions in this interface should *NEVER* be exposed directly via consensus messages, as this could result in violating the immutability promises of the blockchain concept. This interface is comprised of the following functions.

-
  ```
  PutBlock(blockNumber uint64, block *pb.Block) error
  ```

  This function takes a provided, raw block, and inserts it into the blockchain at the given blockNumber. Note that this intended to be an unsafe interface, so no error or sanity checking is performed. Inserting a block with a number higher than the current block height is permitted, similarly overwriting existing already committed blocks is also permitted. Remember, this does not affect the auditability or immutability of the chain, as the hashing techniques make it computationally infeasible to forge a block earlier in the chain. Any attempt to rewrite the blockchain history is therefore easily detectable. This is generally only useful to the state transfer API.

-
  ```
  ApplyStateDelta(id interface{}, delta *statemgmt.StateDelta) error
  ```

  This function takes a state delta, and applies it to the current state. The delta will be applied to transition a state forward or backwards depending on the construction of the state delta. Like the `Executor` methods, `ApplyStateDelta` accepts an opaque interface `id` which should also be passed into `CommitStateDelta` or `RollbackStateDelta` as appropriate.

-
  ```
  CommitStateDelta(id interface{}) error
  ```

  This function commits the state delta which was applied in `ApplyStateDelta` . This is intended to be invoked after the caller to `ApplyStateDelta` has verified the state via the state hash obtained via `GetCurrentStateHash()` . This call takes the same `id` which was passed into `ApplyStateDelta` .

-
  ```
  RollbackStateDelta(id interface{}) error
  ```

  This function unapplies a state delta which was applied in `ApplyStateDelta` . This is intended to be invoked after the caller to `ApplyStateDelta` has detected the state hash obtained via `GetCurrentStateHash()` is incorrect. This call takes the same `id` which was passed into `ApplyStateDelta` .

-
  ```
  EmptyState() error
  ```

  This function will delete the entire current state, resulting in a pristine empty state. It is intended to be called before loading an entirely new state via deltas. This is generally only useful to the state transfer API.

3.4.9 `RemoteLedgers` interface

Definition:

```
type RemoteLedgers interface {
    GetRemoteBlocks(peerID uint64, start, finish uint64) (<-chan *pb.SyncBlocks,␣
↪error)
    GetRemoteStateSnapshot(peerID uint64) (<-chan *pb.SyncStateSnapshot, error)
    GetRemoteStateDeltas(peerID uint64, start, finish uint64) (<-chan *pb.
↪SyncStateDeltas, error)
}
```

The `RemoteLedgers` interface exists primarily to enable state transfer and to interrogate the blockchain state at other replicas. Just like the `WritableLedger` interface, it is not intended to be used in normal operation and is designed to be used for catchup, error recovery, etc. For all functions in this interface it is the caller's responsibility to enforce timeouts. This interface contains the following functions.

-
  ```
  GetRemoteBlocks(peerID uint64, start, finish uint64) (<-chan *pb.SyncBlocks,␣
  ↪error)
  ```

  This function attempts to retrieve a stream of `*pb.SyncBlocks` from the peer designated by `peerID` for the range from `start` to `finish`. In general, `start` should be specified with a higher block number than `finish`, as the blockchain must be validated from end to beginning. The caller must validate that the desired block is being returned, as it is possible that slow results from another request could appear on this channel. Invoking this call for the same `peerID` a second time will cause the first channel to close.

-
  ```
  GetRemoteStateSnapshot(peerID uint64) (<-chan *pb.SyncStateSnapshot, error)
  ```

  This function attempts to retrieve a stream of `*pb.SyncStateSnapshot` from the peer designated by `peerID`. To apply the result, the existing state should first be emptied via the `WritableLedger EmptyState` call, then the contained deltas in the stream should be applied sequentially.

-
  ```
  GetRemoteStateDeltas(peerID uint64, start, finish uint64) (<-chan *pb.
  ↪SyncStateDeltas, error)
  ```

  This function attempts to retrieve a stream of `*pb.SyncStateDeltas` from the peer designated by `peerID` for the range from `start` to `finish`. The caller must validated that the desired block delta is being returned, as it is possible that slow results from another request could appear on this channel. Invoking this call for the same `peerID` a second time will cause the first channel to close.

3.4.10 `controller` package

3.4.10.1 controller.NewConsenter

Signature:

```
func NewConsenter(cpi consensus.CPI) (consenter consensus.Consenter)
```

This function reads the `peer.validator.consensus` value in `core.yaml` configuration file, which is the configuration file for the `peer` process. The value of the `peer.validator.consensus` key defines whether the validating peer will run with the `noops` consensus plugin or the `pbft` one. (Notice that this should eventually be changed to either `noops` or `custom`. In case of `custom`, the validating peer will run with the consensus plugin defined in `consensus/config.yaml`.)

The plugin author needs to edit the function's body so that it routes to the right constructor for their package. For example, for `pbft` we point to the `pbft.GetPlugin` constructor.

This function is called by `helper.NewConsensusHandler` when setting the `consenter` field of the returned message handler. The input argument `cpi` is the output of the `helper.NewHelper` constructor and implements the `consensus.CPI` interface.

3.4.11 `helper` package

3.4.11.1 High-level overview

A validating peer establishes a message handler (`helper.ConsensusHandler`) for every connected peer, via the `helper.NewConsesusHandler` function (a handler factory). Every incoming message is inspected on its type (`helper.HandleMessage`); if it's a message for which consensus needs to be reached, it's passed on to the peer's consenter object (`consensus.Consenter`). Otherwise it's passed on to the next message handler in the stack.

3.4.11.2 helper.ConsensusHandler

Definition:

```
type ConsensusHandler struct {
    chatStream  peer.ChatStream
    consenter   consensus.Consenter
    coordinator peer.MessageHandlerCoordinator
    done        chan struct{}
    peerHandler peer.MessageHandler
}
```

Within the context of consensus, we focus only on the `coordinator` and `consenter` fields. The `coordinator` , as the name implies, is used to coordinate between the peer's message handlers. This is, for instance, the object that is accessed when the peer wishes to `Broadcast` . The `consenter` receives the messages for which consensus needs to be reached and processes them.

Notice that `fabric/peer/peer.go` defines the `peer.MessageHandler` (interface), and `peer.MessageHandlerCoordinator` (interface) types.

3.4.11.3 helper.NewConsensusHandler

Signature:

```
func NewConsensusHandler(coord peer.MessageHandlerCoordinator, stream peer.
↪ChatStream, initiatedStream bool, next peer.MessageHandler) (peer.MessageHandler,␣
↪error)
```

Creates a `helper.ConsensusHandler` object. Sets the same `coordinator` for every message handler. Also sets the `consenter` equal to: `controller.NewConsenter(NewHelper(coord))`

3.4.11.4 helper.Helper

Definition:

```
type Helper struct {
    coordinator peer.MessageHandlerCoordinator
}
```

Contains the reference to the validating peer's `coordinator` . Is the object that implements the `consensus.CPI` interface for the peer.

3.4.11.5 helper.NewHelper

Signature:

```
func NewHelper(mhc peer.MessageHandlerCoordinator) consensus.CPI
```

Returns a `helper.Helper` object whose `coordinator` is set to the input argument `mhc` (the `coordinator` field of the `helper.ConsensusHandler` message handler). This object implements the `consensus.CPI` interface, thus allowing the plugin to interact with the stack.

3.4.11.6 helper.HandleMessage

Recall that the `helper.ConsesusHandler` object returned by `helper.NewConsensusHandler` implements the `peer.MessageHandler` interface:

```
type MessageHandler interface {
    RemoteLedger
    HandleMessage(msg *pb.Message) error
    SendMessage(msg *pb.Message) error
    To() (pb.PeerEndpoint, error)
    Stop() error
}
```

Within the context of consensus, we focus only on the `HandleMessage` method. Signature:

```
func (handler *ConsensusHandler) HandleMessage(msg *pb.Message) error
```

The function inspects the `Type` of the incoming `Message`. There are four cases:

1. Equal to `pb.Message_CONSENSUS`: passed to the handler's `consenter.RecvMsg` function.
2. Equal to `pb.Message_CHAIN_TRANSACTION` (i.e. an external deployment request): a response message is sent to the user first, then the message is passed to the `consenter.RecvMsg` function.
3. Equal to `pb.Message_CHAIN_QUERY` (i.e. a query): passed to the `helper.doChainQuery` method so as to get executed locally.
4. Otherwise: passed to the `HandleMessage` method of the next handler down the stack.

3.5 Events

The event framework provides the ability to generate and consume predefined and custom events. There are 3 basic components: - Event stream - Event adapters - Event structures

3.5.1 Event Stream

An event stream is a gRPC channel capable of sending and receiving events. Each consumer establishes an event stream to the event framework and expresses the events that it is interested in. the event producer only sends appropriate events to the consumers who have connected to the producer over the event stream.

The event stream initializes the buffer and timeout parameters. The buffer holds the number of events waiting for delivery, and the timeout has 3 options when the buffer is full:

- If timeout is less than 0, drop the newly arriving events
- If timeout is 0, block on the event until the buffer becomes available

- If timeout is greater than 0, wait for the specified timeout and drop the event if the buffer remains full after the timeout

3.5.1.1 Event Producer

The event producer exposes a function to send an event, `Send(e *pb.Event)`, where `Event` is either a pre-defined `Block` or a `Generic` event. More events will be defined in the future to include other elements of the fabric.

```
message Generic {
    string eventType = 1;
    bytes payload = 2;
}
```

The `eventType` and `payload` are freely defined by the event producer. For example, JSON data may be used in the `payload`. The `Generic` event may also be emitted by the chaincode or plugins to communicate with consumers.

3.5.1.2 Event Consumer

The event consumer enables external applications to listen to events. Each event consumer registers an event adapter with the event stream. The consumer framework can be viewed as a bridge between the event stream and the adapter. A typical use of the event consumer framework is:

```
adapter = <adapter supplied by the client application to register and receive events>
consumerClient = NewEventsClient(<event consumer address>, adapter)
consumerClient.Start()
...
...
consumerClient.Stop()
```

3.5.2 Event Adapters

The event adapter encapsulates three facets of event stream interaction: - an interface that returns the list of all events of interest - an interface called by the event consumer framework on receipt of an event - an interface called by the event consumer framework when the event bus terminates

The reference implementation provides Golang specific language binding.

```
EventAdapter interface {
   GetInterestedEvents() ([]*ehpb.Interest, error)
   Recv(msg *ehpb.Event) (bool,error)
   Disconnected(err error)
}
```

Using gRPC as the event bus protocol allows the event consumer framework to be ported to different language bindings without affecting the event producer framework.

3.5.3 Event Structure

This section details the message structures of the event system. Messages are described directly in Golang for simplicity.

The core message used for communication between the event consumer and producer is the Event.

```
message Event {
    oneof Event {
        //consumer events
        Register register = 1;

        //producer events
        Block block = 2;
        Generic generic = 3;
   }
}
```

Per the above definition, an event has to be one of `Register`, `Block` or `Generic`.

As mentioned in the previous sections, a consumer creates an event bus by establishing a connection with the producer and sending a `Register` event. The `Register` event is essentially an array of `Interest` messages declaring the events of interest to the consumer.

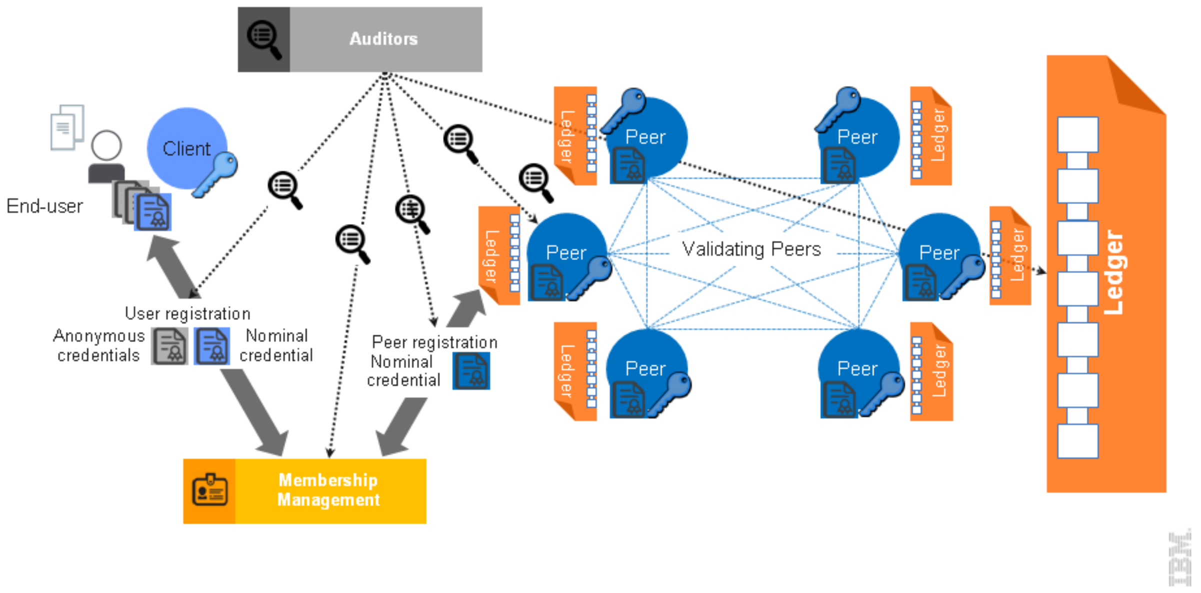
```
message Interest {
    enum ResponseType {
        //don't send events (used to cancel interest)
        DONTSEND = 0;
        //send protobuf objects
        PROTOBUF = 1;
        //marshall into JSON structure
        JSON = 2;
    }
    string eventType = 1;
    ResponseType responseType = 2;
}
```

Events can be sent directly as protobuf structures or can be sent as JSON structures by specifying the `responseType` appropriately.

Currently, the producer framework can generate a `Block` or a `Generic` event. A `Block` is a message used for encapsulating properties of a block in the blockchain.

4. Security

This section discusses the setting depicted in the figure below. In particular, the system consists of the following entities: membership management infrastructure, i.e., a set of entities that are responsible for identifying an individual user (using any form of identification considered in the system, e.g., credit cards, id-cards), open an account for that user to be able to register, and issue the necessary credentials to successfully create transactions and deploy or invoke chaincode successfully through the fabric.



* Peers, that are classified as validating peers, and non-validating peers. Validating peers (also known as validators) order and process (check validity, execute, and add to the blockchain) user-messages (transactions) submitted to the network. Non validating peers (also known as peers) receive user transactions on behalf of users, and after some fundamental validity checks, they forward the transactions to their neighboring validating peers. Peers maintain an up-to-date copy of the blockchain, but in contradiction to validators, they do not execute transactions (a process also known as *transaction validation*). * End users of the system, that have registered to our membership service administration, after having demonstrated ownership of what is considered *identity* in the system, and have obtained credentials to install the client-software and submit transactions to the system. * Client-software, the software that needs to be installed at the client side for the latter to be able to complete his registration to our membership service and submit transactions to the system. * Online wallets, entities that are trusted by a user to maintain that user's credentials, and submit transactions solely upon user request to the network. Online wallets come with their own software at the client-side, that is usually light-weight, as the client only needs to authenticate himself and his requests to the wallet. While it can be the case that peers can play the role of *online wallet* for a set of users, in the following sessions the security of online wallets is detailed separately.

Users who wish to make use of the fabric, open an account at the membership management administration, by proving ownership of identity as discussed in previous sections, new chaincodes are announced to the blockchain network by the chaincode creator (developer) through the means of a deployment transaction that the client-software would construct on behalf of the developer. Such transaction is first received by a peer or validator, and afterwards circulated in the entire network of validators, this transaction is executed and finds its place to the blockchain network. Users can also invoke a function of an already deployed chain-code through an invocation transaction.

The next section provides a summary of the business goals of the system that drive the security requirements. We then overview the security components and their operation and show how this design fulfills the security requirements.

4.1 Business security requirements

This section presents business security requirements that are relevant to the context of the fabric. **Incorporation of identity and role management.**

In order to adequately support real business applications it is necessary to progress beyond ensuring cryptographic continuity. A workable B2B system must consequently move towards addressing proven/demonstrated identities or other attributes relevant to conducting business. Business transactions and consumer interactions with financial institutions need to be unambiguously mapped to account holders. Business contracts typically require demonstrable affiliation with specific institutions and/or possession of other specific properties of transacting parties. Accountability and non-frameability are two reasons that identity management is a critical component of such systems.

Accountability means that users of the system, individuals, or corporations, who misbehave can be traced back and be set accountable for their actions. In many cases, members of a B2B system are required to use their identities (in some form) to participate in the system, in a way such that accountability is guaranteed. Accountability and non-frameability are both essential security requirements in B2B systems and they are closely related. That is, a B2B system should guarantee that an honest user of such system cannot be framed to be accused as responsible for transactions originated by other users.

In addition a B2B system should be renewable and flexible in order to accommodate changes of participants's roles and/or affiliations.

Transactional privacy.

In B2B relationships there is a strong need for transactional privacy, i.e., allowing the end-user of a system to control the degree to which it interacts and shares information with its environment. For example, a corporation doing business through a transactional B2B system requires that its transactions are not visible to other corporations or industrial partners that are not authorized to share classified information with.

Transactional privacy in the fabric is offered by the mechanisms to achieve two properties with respect to non authorized users:

- Transaction anonymity, where the owner of a transaction is hidden among the so called *anonymity set*, which in the fabric, is the set of users.
- Transaction unlinkability, where two or more transactions of the same user should not be linked as such.

Clearly depending on the context, non-authorized users can be anyone outside the system, or a subset of users.

Transactional privacy is strongly associated to the confidentiality of the content of a contractual agreement between two or more members of a B2B system, as well as to the anonymity and unlinkability of any authentication mechanism that should be in place within transactions.

Reconciling transactional privacy with identity management.

As described later in this document, the approach taken here to reconcile identity management with user privacy and to enable competitive institutions to transact effectively on a common blockchain (for both intra- and inter-institutional transactions) is as follows:

1. add certificates to transactions to implement a "permissioned" blockchain
2. utilize a two-level system:
3. (relatively) static enrollment certificates (ECerts), acquired via registration with an enrollment certificate authority (CA).
4. transaction certificates (TCerts) that faithfully but pseudonymously represent enrolled users, acquired via a transaction CA.
5. offer mechanisms to conceal the content of transactions to unauthorized members of the system.

Audit support. Commercial systems are occasionally subjected to audits. Auditors in such cases should be given the means to check a certain transaction, or a certain group of transactions, the activity of a particular user of the system, or the operation of the system itself. Thus, such capabilities should be offered by any system featuring transactions containing contractual agreements between business partners.

4.2 User Privacy through Membership Services

Membership Services consists of an infrastructure of several entities that together manage the identity and privacy of users on the network. These services validate user's identity, register the user in the system, and provide all the credentials needed for him/her to be an active and compliant participant able to create and/or invoke transactions. A Public Key Infrastructure (PKI) is a framework based on public key cryptography that ensures not only the secure exchange of data over public networks but also affirms the identity of the other party. A PKI manages the generation, distribution and revocation of keys and digital certificates. Digital certificates are used to establish user credentials and to sign messages. Signing messages with a certificate ensures that the message has not been altered. Typically a PKI has a Certificate Authority (CA), a Registration Authority (RA), a certificate database, and a certificate storage. The RA is a trusted party that authenticates users and vets the legitimacy of data, certificates or other evidence submitted to support the user's request for one or more certificates that reflect that user's identity or other properties. A CA, upon advice from an RA, issues digital certificates for specific uses and is certified directly or hierarchically by a root CA. Alternatively, the user-facing communications and due diligence responsibilities of the RA can be subsumed as part of the CA. Membership Services is composed of the entities shown in the following figure. Introduction of such full PKI reinforces the strength of this system for B2B (over, e.g. Bitcoin).

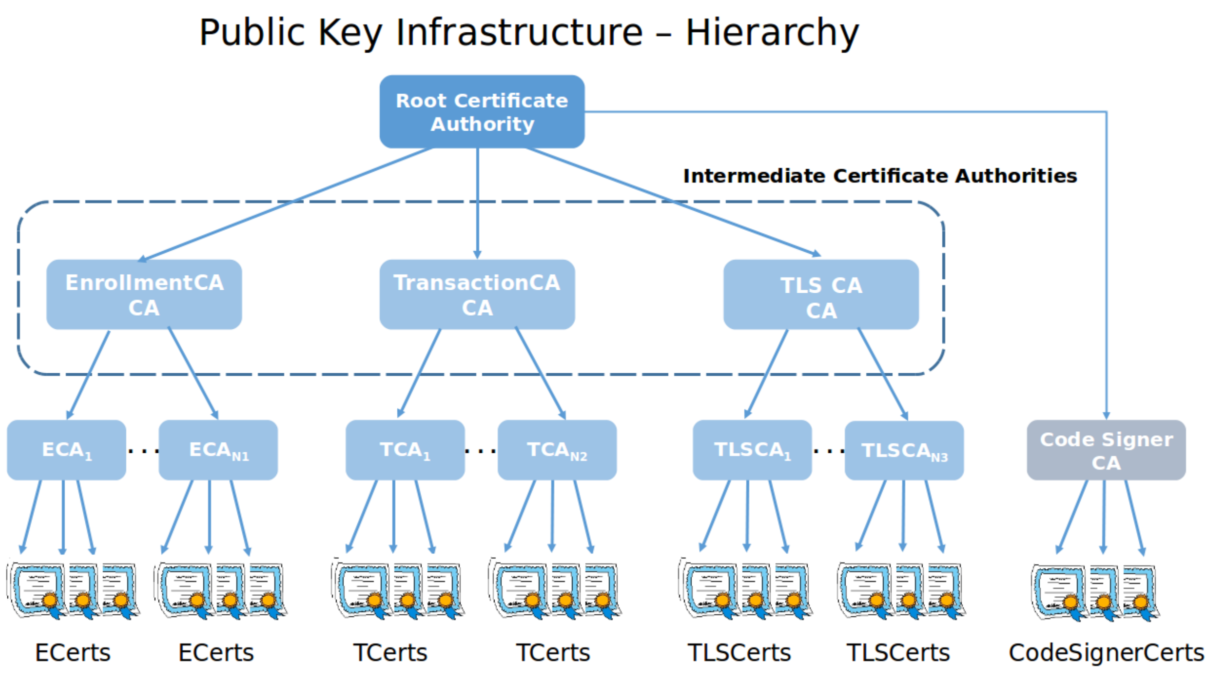


Fig. 11.4: Figure 1

Root Certificate Authority (Root CA): entity that represents the trust anchor for the PKI scheme. Digital certificates verification follows a chain of trust. The Root CA is the top-most CA in the PKI hierarchy.

Registration Authority (RA): a trusted entity that can ascertain the validity and identity of users who want to participate in the permissioned blockchain. It is responsible for out-of-band communication with the user to validate his/her identity and role. It creates registration credentials needed for enrollment and information on root of trust.

Enrollment Certificate Authority (ECA): responsible for issuing Enrollment Certificates (ECerts) after validating the registration credentials provided by the user.

Transaction Certificate Authority (TCA): responsible for issuing Transaction Certificates (TCerts) after validating the enrollment credentials provided by the user.

TLS Certificate Authority (TLS-CA): responsible for issuing TLS certificates and credentials that allow the user to make use of its network. It validates the credential(s) or evidence provided by the user that justifies issuance of a TLS certificate that includes specific information pertaining to the user.

In this specification, membership services is expressed through the following associated certificates issued by the PKI:

Enrollment Certificates (ECerts) ECerts are long-term certificates. They are issued for all roles, i.e. users, non-validating peers, and validating peers. In the case of users, who submit transactions for candidate incorporation into the blockchain and who also own TCerts (discussed below), there are two possible structure and usage models for ECerts:

- Model A: ECerts contain the identity/enrollmentID of their owner and can be used to offer only nominal entity-authentication for TCert requests and/or within transactions. They contain the public part of two key pairs – a signature key-pair and an encryption/key agreement key-pair. ECerts are accessible to everyone.
- Model B: ECerts contain the identity/enrollmentID of their owner and can be used to offer only nominal entity-authentication for TCert requests. They contain the public part of a signature key-pair, i.e., a signature verification public key. ECerts are preferably accessible to only TCA and auditors, as relying parties. They are invisible to transactions, and thus (unlike TCerts) their signature key pairs do not play a non-repudiation role at that level.

Transaction Certificates (TCerts) TCerts are short-term certificates for each transaction. They are issued by the TCA upon authenticated user-request. They securely authorize a transaction and may be configured to not reveal the identities of who is involved in the transaction or to selectively reveal such identity/enrollmentID information. They include the public part of a signature key-pair, and may be configured to also include the public part of a key agreement key pair. They are issued only to users. They are uniquely associated to the owner – they may be configured so that this association is known only by the TCA (and to authorized auditors). TCerts may be configured to not carry information of the identity of the user. They enable the user not only to anonymously participate in the system but also prevent linkability of transactions.

However, auditability and accountability requirements assume that the TCA is able to retrieve TCerts of a given identity, or retrieve the owner of a specific TCert. For details on how TCerts are used in deployment and invocation transactions see Section 4.3, Transaction Security offerings at the infrastructure level.

TCerts can accommodate encryption or key agreement public keys (as well as digital signature verification public keys). If TCerts are thus equipped, then enrollment certificates need not also contain encryption or key agreement public keys.

Such a key agreement public key, Key_Agreement_TCertPub_Key, can be generated by the transaction certificate authority (TCA) using a method that is the same as that used to generate the Signature_Verification_TCertPub_Key, but using an index value of TCertIndex + 1 rather than TCertIndex, where TCertIndex is hidden within the TCert by the TCA for recovery by the TCert owner.

The structure of a Transaction Certificate (TCert) is as follows: * TCertID – transaction certificate ID (preferably generated by TCA randomly in order to avoid unintended linkability via the Hidden Enrollment ID field). * Hidden Enrollment ID: AES_EncryptK(enrollmentID), where key K = [HMAC(Pre-K, TCertID)]256-bit truncation and where three distinct key distribution scenarios for Pre-K are defined below as (a), (b) and (c). * Hidden Private Keys Extraction: AES_EncryptTCertOwner_EncryptKey(TCertIndex || known padding/parity check vector) where || denotes concatenation, and where each batch has a unique (per batch) time-stamp/random offset that is added to a counter (initialized at 1 in this implementation) in order to generate TCertIndex. The counter can be incremented by 2 each time in order to accommodate generation by the TCA of the public keys and recovery by the TCert owner of the private keys of both types, i.e., signature key pairs and key agreement key pairs. * Sign Verification Public Key – TCert signature verification public key. * Key Agreement Public Key – TCert key agreement public key. * Validity period – the time window during which the transaction certificate can be used for the outer/external signature of a transaction.

There are at least three useful ways to consider configuring the key distribution scenario for the Hidden Enrollment ID field: *(a)* Pre-K is distributed during enrollment to user clients, peers and auditors, and is available to the TCA and authorized auditors. It may, for example, be derived from Kchain (described subsequently in this specification) or be independent of key(s) used for chaincode confidentiality.

(b) Pre-K is available to validators, the TCA and authorized auditors. K is made available by a validator to a user (under TLS) in response to a successful query transaction. The query transaction can have the same format as the invocation transaction. Corresponding to Example 1 below, the querying user would learn the enrollmentID of the user who created the Deployment Transaction if the querying user owns one of the TCerts in the ACL of the Deployment Transaction. Corresponding to Example 2 below, the querying user would learn the enrollmentID of the user who created the Deployment Transaction if the enrollmentID of the TCert used to query matches one of the affiliations/roles in the Access Control field of the Deployment Transaction.

Example 1:

Example 2:

(c) Pre-K is available to the TCA and authorized auditors. The TCert-specific K can be distributed the TCert owner (under TLS) along with the TCert, for each TCert in the batch. This enables targeted release by the TCert owner of K (and thus trusted notification of the TCert owner's enrollmentID). Such targeted release can use key agreement public keys of the intended recipients and/or PKchain where SKchain is available to validators as described subsequently in this specification. Such targeted release to other contract participants can be incorporated into a transaction or done out-of-band.

If the TCerts are used in conjunction with ECert Model A above, then using (c) where K is not distributed to the TCert owner may suffice. If the TCerts are used in conjunction with ECert Model A above, then the Key Agreement Public Key field of the TCert may not be necessary.

The Transaction Certificate Authority (TCA) returns TCerts in batches, each batch contains the KeyDF_Key (Key-Derivation-Function Key) which is not included within every TCert but delivered to the client with the batch of TCerts (using TLS). The KeyDF_Key allows the TCert owner to derive TCertOwner_EncryptKey which in turn enables recovery of TCertIndex from AES_EncryptTCertOwner_EncryptKey(TCertIndex || known padding/parity check vector).

TLS-Certificates (TLS-Certs) TLS-Certs are certificates used for system/component-to-system/component communications. They carry the identity of their owner and are used for network level security.

This implementation of membership services provides the following basic functionality: there is no expiration/revocation of ECerts; expiration of TCerts is provided via the validity period time window; there is no revocation of TCerts. The ECA, TCA, and TLS CA certificates are self-signed, where the TLS CA is provisioned as a trust anchor.

4.2.1 User/Client Enrollment Process

The next figure has a high-level description of the user enrollment process. It has an offline and an online phase.

Offline Process: in Step 1, each user/non-validating peer/validating peer has to present strong identification credentials (proof of ID) to a Registration Authority (RA) offline. This has to be done out-of-band to provide the evidence needed by the RA to create (and store) an account for the user. In Step 2, the RA returns the associated username/password and trust anchor (TLS-CA Cert in this implementation) to the user. If the user has access to a local client then this is one way the client can be securely provisioned with the TLS-CA certificate as trust anchor.

Online Phase: In Step 3, the user connects to the client to request to be enrolled in the system. The user sends his username and password to the client. On behalf of the user, the client sends the request to the PKI framework, Step 4, and receives a package, Step 5, containing several certificates, some of which should correspond to private/secret keys held by the client. Once the client verifies that the all the crypto material in the package is correct/valid, it stores the certificates in local storage and notifies the user. At this point the user enrollment has been completed.

Figure 4 shows a detailed description of the enrollment process. The PKI framework has the following entities – RA, ECA, TCA and TLS-CA. After Step 1, the RA calls the function "AddEntry" to enter the (username/password) in its database. At this point the user has been formally registered into the system database. The client needs the TLS-CA certificate (as trust anchor) to verify that the TLS handshake is set up appropriately with the server. In Step 4, the client sends the registration request to the ECA along with its enrollment public key and additional identity information such as username and password (under the TLS record layer protocol). The ECA verifies that such user really exists in

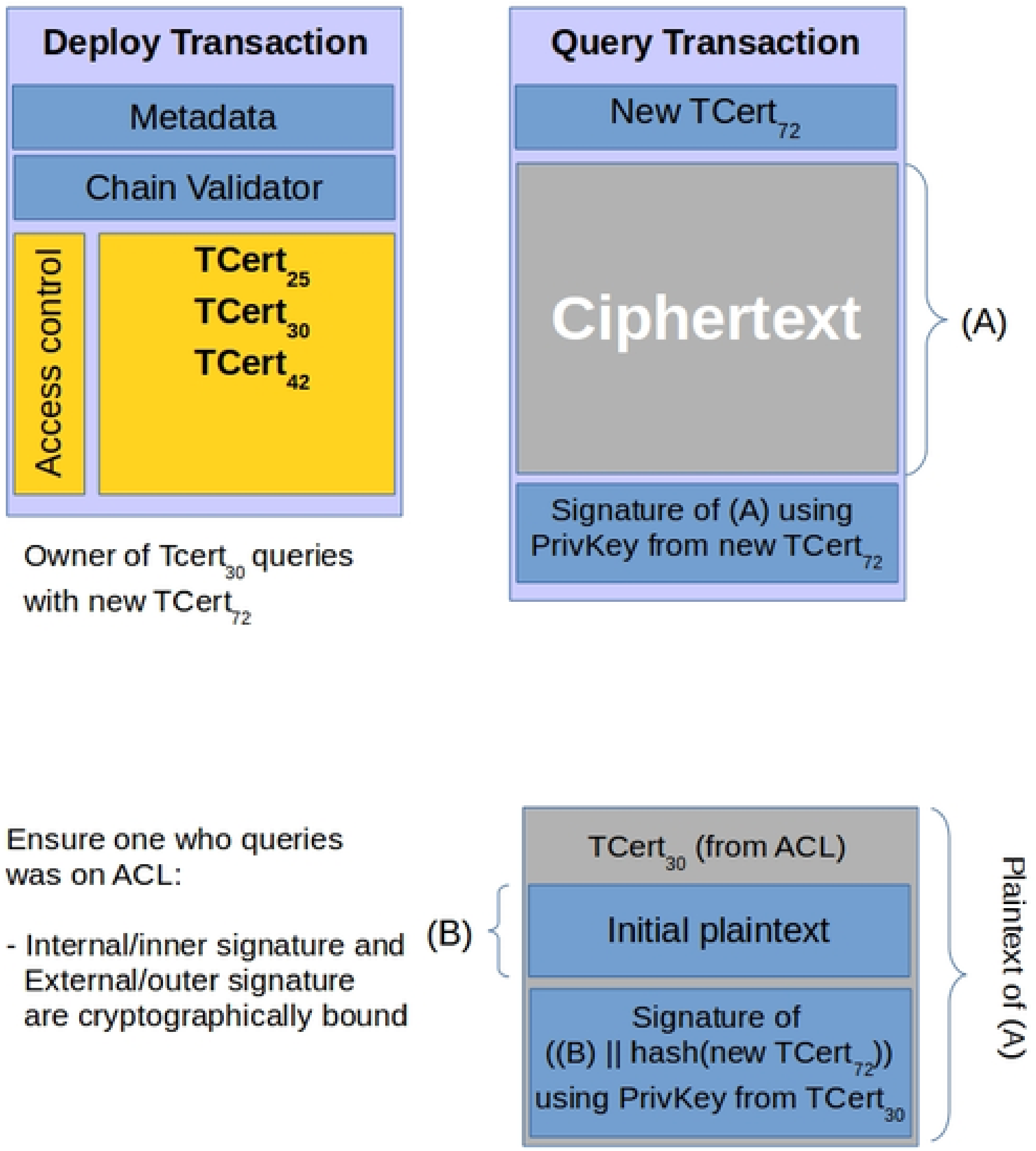
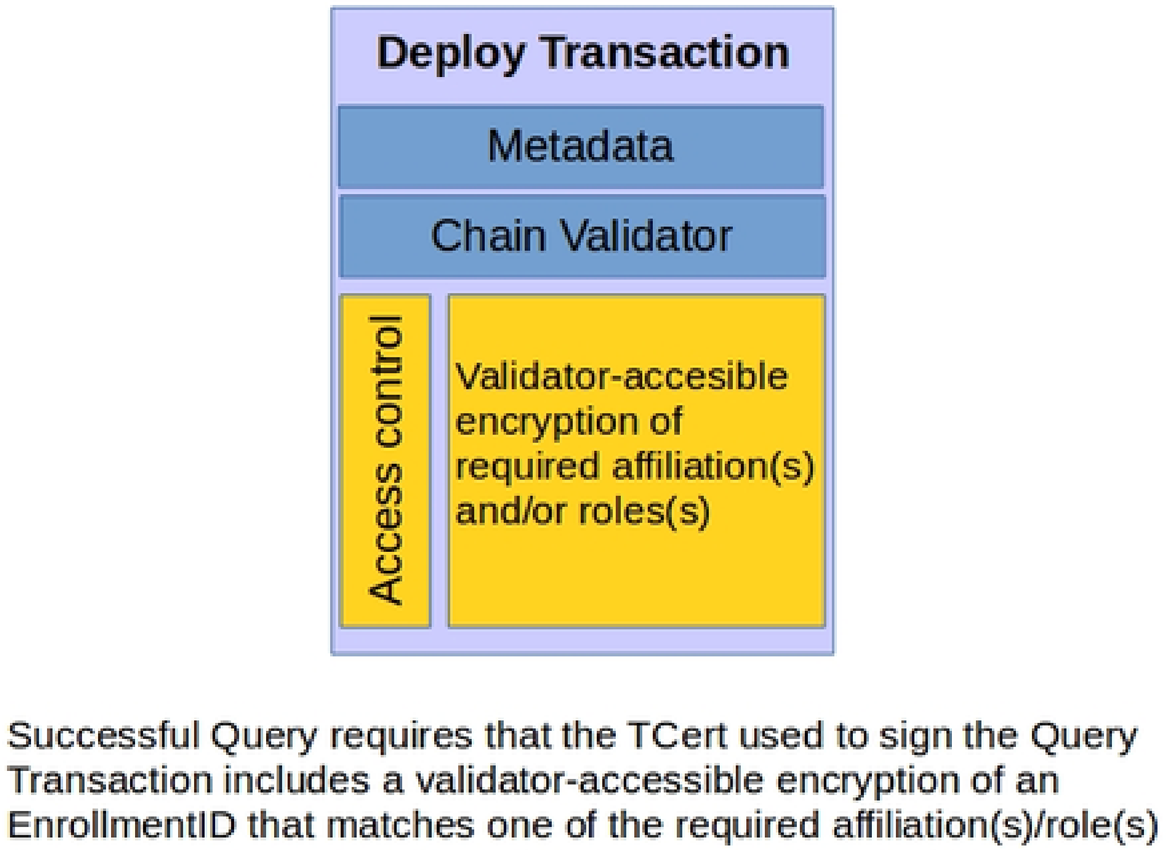

Deploy Transaction
Metadata
Chain Validator
Access control
TCert$_{25}$
TCert$_{30}$
TCert$_{42}$
Owner of Tcert$_{30}$ queries with new TCert$_{72}$
Query Transaction
New TCert$_{72}$
Ciphertext
(A)
Signature of (A) using PrivKey from new TCert$_{72}$
Ensure one who queries was on ACL:
- Internal/inner signature and External/outer signature are cryptographically bound
(B)
TCert$_{30}$ (from ACL)
Initial plaintext
Signature of ((B) || hash(new TCert$_{72}$)) using PrivKey from TCert$_{30}$
Plaintext of (A)

Fig. 11.5: Example 1


Deploy Transaction
Metadata
Chain Validator
Access control
Validator-accesible encryption of required affiliation(s) and/or roles(s)

Successful Query requires that the TCert used to sign the Query Transaction includes a validator-accessible encryption of an EnrollmentID that matches one of the required affiliation(s)/role(s)

Fig. 11.6: Example 2

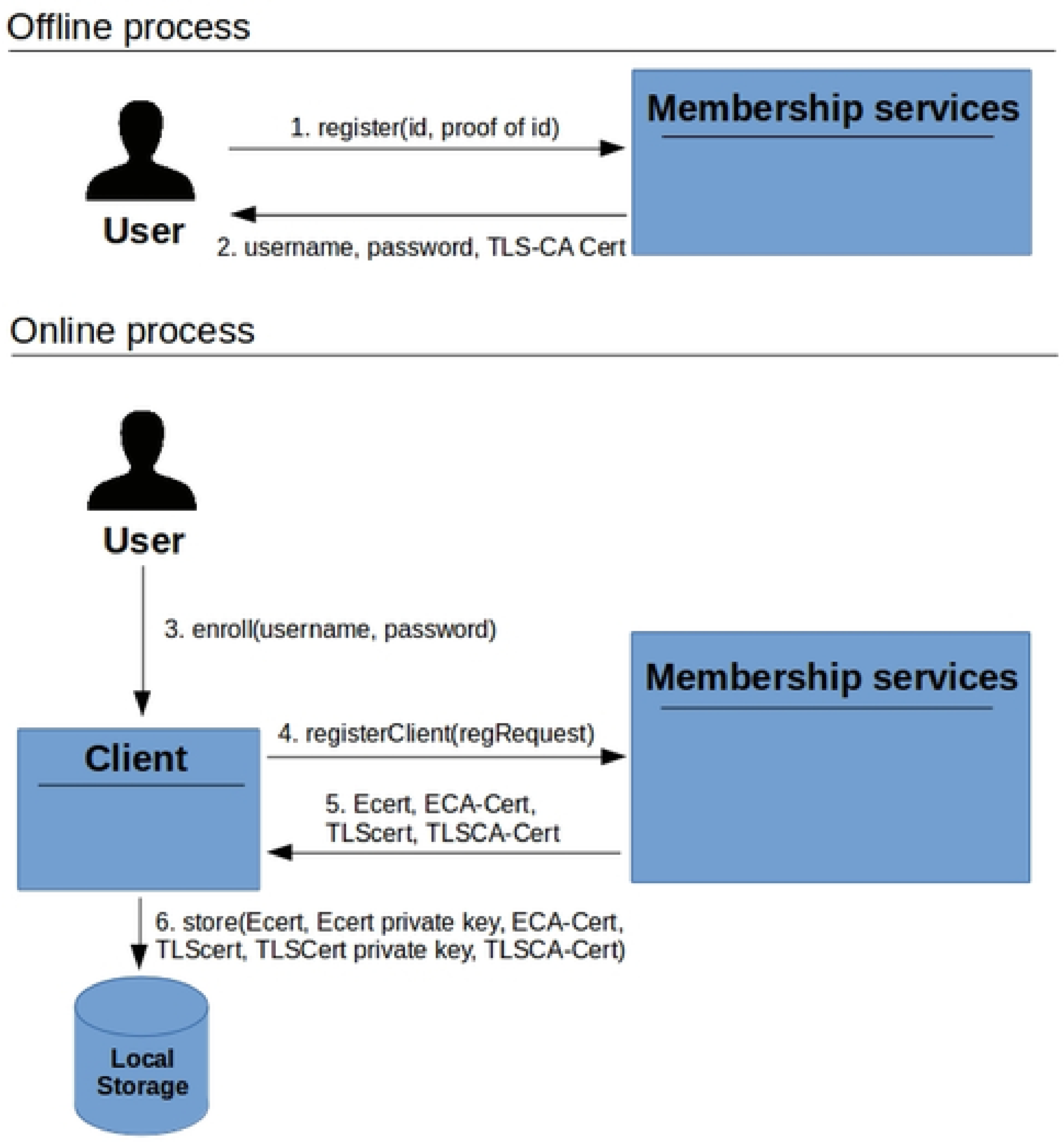

Offline process
User
1. register(id, proof of id)
2. username, password, TLS-CA Cert
Membership services
Online process
User
3. enroll(username, password)
Client
4. registerClient(regRequest)
5. Ecert, ECA-Cert, TLScert, TLSCA-Cert
Membership services
6. store(Ecert, Ecert private key, ECA-Cert, TLScert, TLSCert private key, TLSCA-Cert)
Local Storage

Fig. 11.7: Registration

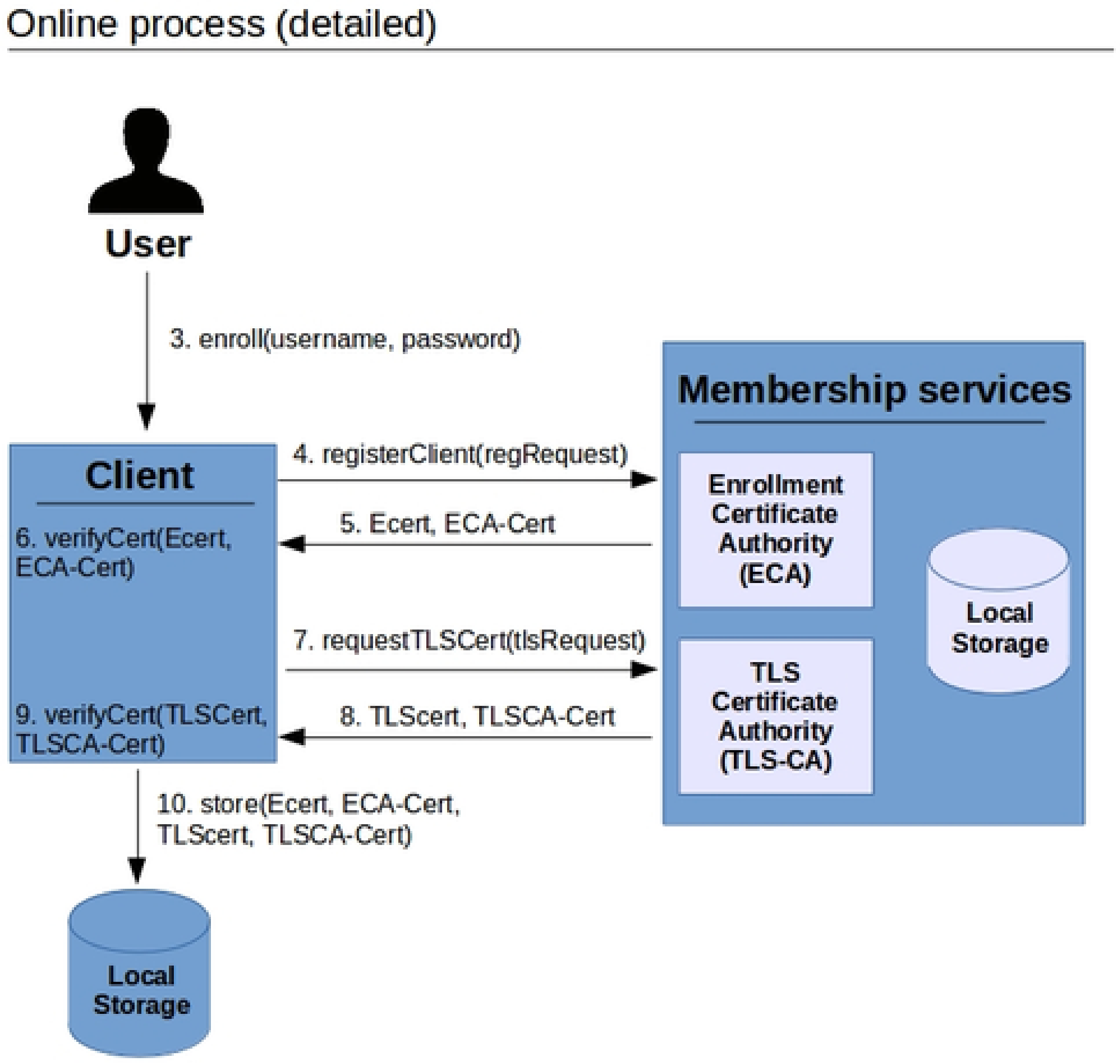
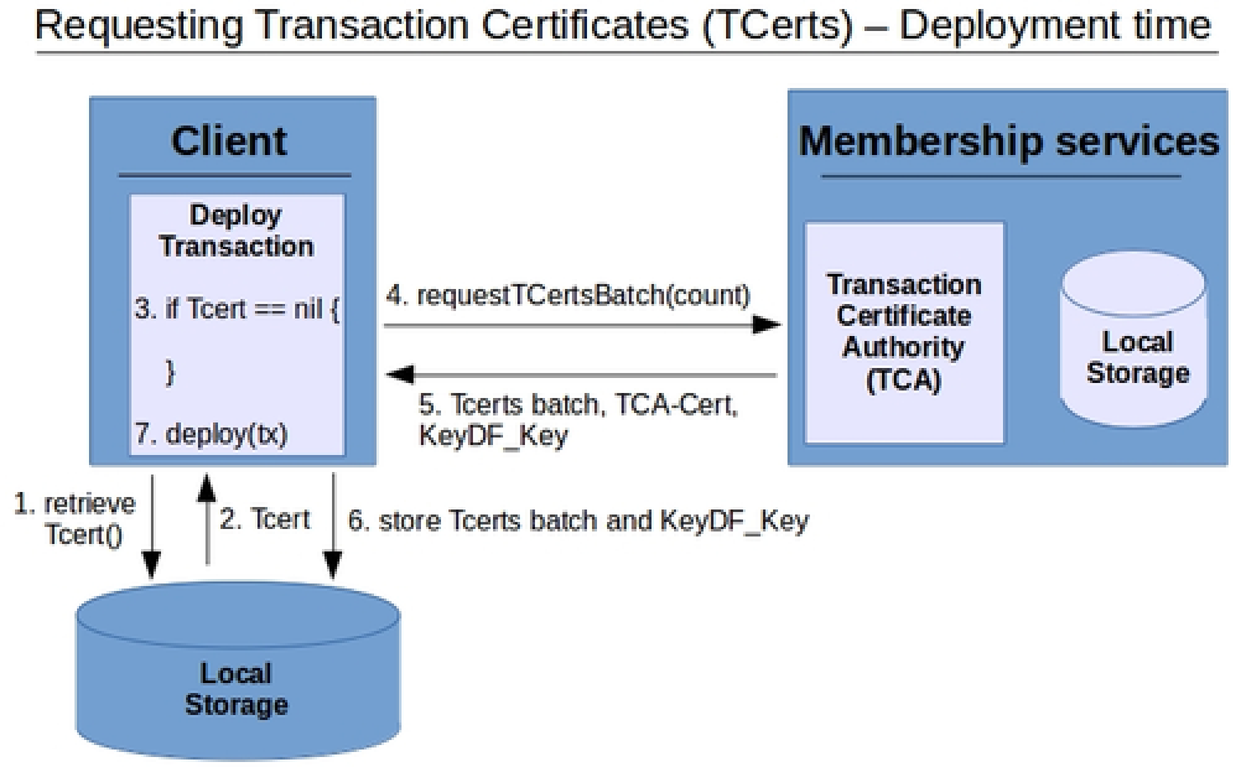

Online process (detailed)
User
3. enroll(username, password)
Client
6. verifyCert(Ecert, ECA-Cert)
9. verifyCert(TLSCert, TLSCA-Cert)
4. registerClient(regRequest)
5. Ecert, ECA-Cert
7. requestTLSCert(tlsRequest)
8. TLScert, TLSCA-Cert
Membership services
Enrollment Certificate Authority (ECA)
TLS Certificate Authority (TLS-CA)
Local Storage
10. store(Ecert, ECA-Cert, TLScert, TLSCA-Cert)
Local Storage

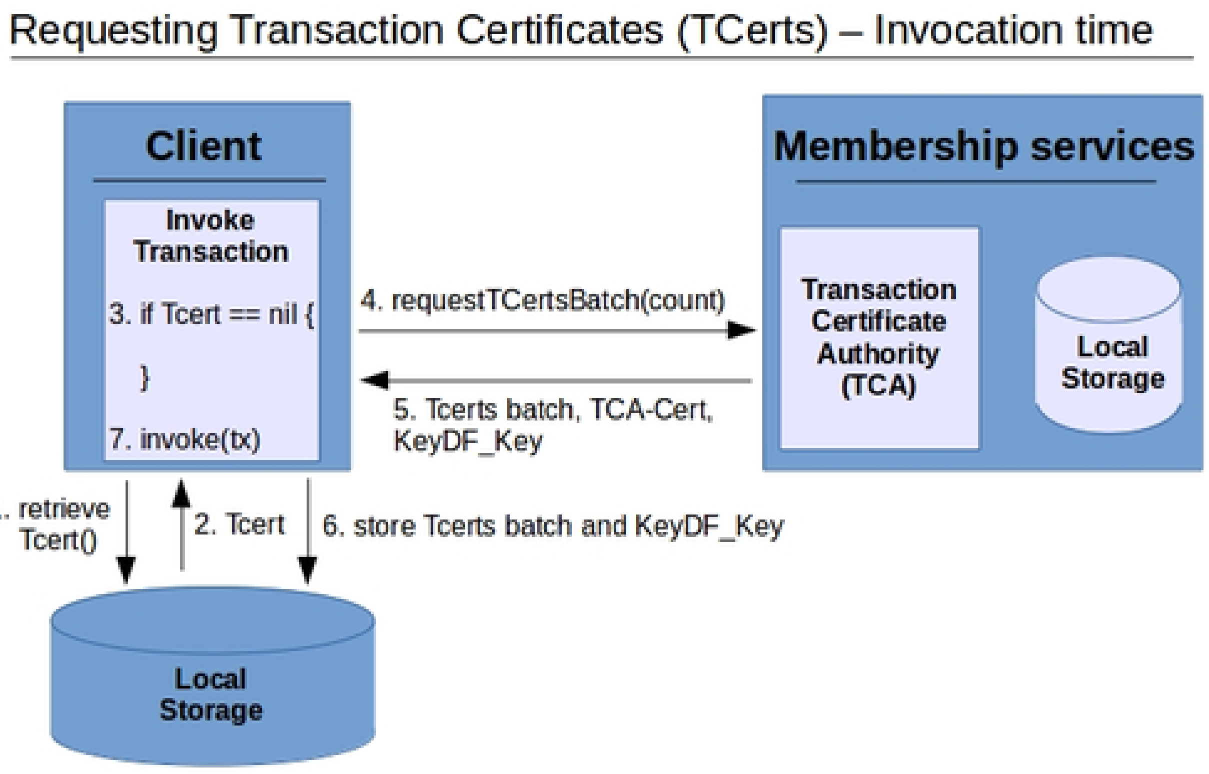
Fig. 11.8: Figure 4

the database. Once it establishes this assurance the user has the right to submit his/her enrollment public key and the ECA will certify it. This enrollment information is of a one-time use. The ECA updates the database marking that this registration request information (username/password) cannot be used again. The ECA constructs, signs and sends back to the client an enrollment certificate (ECert) that contains the user's enrollment public key (Step 5). It also sends the ECA Certificate (ECA-Cert) needed in future steps (client will need to prove to the TCA that his/her ECert was created by the proper ECA). (Although the ECA-Cert is self-signed in the initial implementation, the TCA and TLS-CA and ECA are co-located.) The client verifies, in Step 6, that the public key inside the ECert is the one originally submitted by the client (i.e. that the ECA is not cheating). It also verifies that all the expected information within the ECert is present and properly formed.

Similarly, In Step 7, the client sends a registration request to the TLS-CA along with its public key and identity information. The TLS-CA verifies that such user is in the database. The TLS-CA generates, and signs a TLS-Cert that contains the user's TLS public key (Step 8). TLS-CA sends the TLS-Cert and its certificate (TLS-CA Cert). Step 9 is analogous to Step 6, the client verifies that the public key inside the TLS Cert is the one originally submitted by the client and that the information in the TLS Cert is complete and properly formed. In Step 10, the client saves all certificates in local storage for both certificates. At this point the user enrollment has been completed.

In this implementation the enrollment process for validators is the same as that for peers. However, it is possible that a different implementation would have validators enroll directly through an on-line process.


Requesting Transaction Certificates (TCerts) – Deployment time
Client
Deploy Transaction
3. if Tcert == nil {
}
7. deploy(tx)
4. requestTCertsBatch(count)
5. Tcerts batch, TCA-Cert, KeyDF_Key
Membership services
Transaction Certificate Authority (TCA)
Local Storage
1. retrieve Tcert()
2. Tcert
6. store Tcerts batch and KeyDF_Key
Local Storage


Requesting Transaction Certificates (TCerts) – Invocation time
Client
Invoke Transaction
3. if Tcert == nil {
}
7. invoke(tx)
4. requestTCertsBatch(count)
5. Tcerts batch, TCA-Cert, KeyDF_Key
Membership services
Transaction Certificate Authority (TCA)
Local Storage
1. retrieve Tcert()
2. Tcert
6. store Tcerts batch and KeyDF_Key
Local Storage

Client: Request for TCerts batch needs to include (in addition to count), ECert and signature of request using ECert private key (where Ecert private key is pulled from Local Storage).

TCA generates TCerts for batch: Generates key derivation function key, KeyDF_Key, as HMAC(TCA_KDF_Key, EnrollPub_Key). Generates each TCert public key (using TCertPub_Key = EnrollPub_Key + ExpansionValue G, where 384-bit ExpansionValue = HMAC(Expansion_Key, TCertIndex) and 384-bit Expansion_Key = HMAC(KeyDF_Key, "2")). Generates each AES_EncryptTCertOwner_EncryptKey(TCertIndex || known padding/parity check vector), where || denotes concatenation and where TCertOwner_EncryptKey is derived as [HMAC(KeyDF_Key, "1")]256-bit truncation.

Client: Deriving TCert private key from a TCert in order to be able to deploy or invoke or query: KeyDF_Key and ECert private key need to be pulled from Local Storage. KeyDF_Key is used to derive TCertOwner_EncryptKey as [HMAC(KeyDF_Key, "1")]256-bit truncation; then TCertOwner_EncryptKey is used to decrypt the TCert field AES_EncryptTCertOwner_EncryptKey(TCertIndex || known padding/parity check vector); then TCertIndex is used to derive TCert private key: TCertPriv_Key = (EnrollPriv_Key + ExpansionValue) modulo n, where 384-bit ExpansionValue = HMAC(Expansion_Key, TCertIndex) and 384-bit Expansion_Key = HMAC(KeyDF_Key, "2").

4.2.2 Expiration and revocation of certificates

It is practical to support expiration of transaction certificates. The time window during which a transaction certificate can be used is expressed by a 'validity period' field. The challenge regarding support of expiration lies in the distributed nature of the system. That is, all validating entities must share the same information; i.e. be consistent with respect to the expiration of the validity period associated with the transactions to be executed and validated. To guarantee that the expiration of validity periods is done in a consistent manner across all validators, the concept of validity period identifier is introduced. This identifier acts as a logical clock enabling the system to uniquely identify a validity period. At genesis time the "current validity period" of the chain gets initialized by the TCA. It is essential that this validity period identifier is given monotonically increasing values over time, such that it imposes a total order among validity periods.

A special type of transactions, system transactions, and the validity period identified are used together to announce the expiration of a validity period to the Blockchain. System transactions refer to contracts that have been defined in the genesis block and are part of the infrastructure. The validity period identified is updated periodically by the TCA invoking a system chaincode. Note that only the TCA should be allowed to update the validity period. The TCA sets the validity period for each transaction certificate by setting the appropriate integer values in the following two fields that define a range: 'not-before' and 'not-after' fields.

TCert Expiration: At the time of processing a TCert, validators read from the state table associated with the ledger the value of 'current validity period' to check if the outer certificate associated with the transaction being evaluated is currently valid. That is, the current value in the state table has to be within the range defined by TCert sub-fields 'not-before' and 'not-after'. If this is the case, the validator continues processing the transaction. In the case that the current value is not within range, the TCert has expired or is not yet valid and the validator should stop processing the transaction.

ECert Expiration: Enrollment certificates have different validity period length(s) than those in transaction certificates.

Revocation is supported in the form of Certificate Revocation Lists (CRLs). CRLs identify revoked certificates. Changes to the CRLs, incremental differences, are announced through the Blockchain.

4.3 Transaction security offerings at the infrastructure level

Transactions in the fabric are user-messages submitted to be included in the ledger. As discussed in previous sections, these messages have a specific structure, and enable users to deploy new chaincodes, invoke existing chaincodes, or query the state of existing chaincodes. Therefore, the way transactions are formed, announced and processed plays an important role to the privacy and security offerings of the entire system.

On one hand our membership service provides the means to authenticate transactions as having originated by valid users of the system, to disassociate transactions with user identities, but while efficiently tracing the transactions a particular individual under certain conditions (law enforcement, auditing). In other words, membership services offer to transactions authentication mechanisms that marry user-privacy with accountability and non-repudiation.

On the other hand, membership services alone cannot offer full privacy of user-activities within the fabric. First of all, for privacy provisions offered by the fabric to be complete, privacy-preserving authentication mechanisms need to be accompanied by transaction confidentiality. This becomes clear if one considers that the content of a chaincode, may leak information on who may have created it, and thus break the privacy of that chaincode's creator. The first subsection discusses transaction confidentiality.

Enforcing access control for the invocation of chaincode is an important security requirement. The fabric exposes to the application (e.g., chaincode creator) the means for the application to perform its own invocation access control, while leveraging the fabric's membership services. Section 4.4 elaborates on this.

Replay attacks is another crucial aspect of the security of the chaincode, as a malicious user may copy a transaction that was added to the Blockchain in the past, and replay it in the network to distort its operation. This is the topic of Section 4.3.3.

The rest of this Section presents an overview of how security mechanisms in the infrastructure are incorporated in the transactions' lifecycle, and details each security mechanism separately.

4.3.1 Security Lifecycle of Transactions

Transactions are created on the client side. The client can be either plain client, or a more specialized application, i.e., piece of software that handles (server) or invokes (client) specific chaincodes through the blockchain. Such applications are built on top of the platform (client) and are detailed in Section 4.4.

Developers of new chaincodes create a new deploy transaction by passing to the fabric infrastructure: * the confidentiality/security version or type they want the transaction to conform with, * the set of users who wish to be given access to parts of the chaincode and a proper representation of their (read) access rights * the chaincode specification,

* code metadata, containing information that should be passed to the chaincode at the time of its execution (e.g., configuration parameters), and * transaction metadata, that is attached to the transaction structure, and is only used by the application that deployed the chaincode.

Invoke and query transactions corresponding to chaincodes with confidentiality restrictions are created using a similar approach. The transactor provides the identifier of the chaincode to be executed, the name of the function to be invoked and its arguments. Optionally, the invoker can pass to the transaction creation function, code invocation metadata, that will be provided to the chaincode at the time of its execution. Transaction metadata is another field that the application of the invoker or the invoker himself can leverage for their own purposes.

Finally transactions at the client side, are signed by a certificate of their creator and released to the network of validators. Validators receive the confidential transactions, and pass them through the following phases: * *pre-validation* phase, where validators validate the transaction certificate against the accepted root certificate authority, verify transaction certificate signature included in the transaction (statically), and check whether the transaction is a replay (see, later section for details on replay attack protection). * *consensus* phase, where the validators add this transaction to the total order of transactions (ultimately included in the ledger) * *pre-execution* phase, where validators verify the validity of the transaction / enrollment certificate against the current validity period, decrypt the transaction (if the transaction is encrypted), and check that the transaction's plaintext is correctly formed(e.g., invocation access control is respected, included TCerts are correctly formed); mini replay-attack check is also performed here within the transactions of the currently processed block. * *execution* phase, where the (decrypted) chaincode is passed to a container, along with the associated code metadata, and is executed * *commit* phase, where (encrypted) updates of that chaincodes state is committed to the ledger with the transaction itself.

4.3.2 Transaction confidentiality

Transaction confidentiality requires that under the request of the developer, the plain-text of a chaincode, i.e., code, description, is not accessible or inferable (assuming a computational attacker) by any unauthorized entities(i.e., user or peer not authorized by the developer). For the latter, it is important that for chaincodes with confidentiality requirements the content of both *deploy* and *invoke* transactions remains concealed. In the same spirit, non-authorized parties, should not be able to associate invocations (invoke transactions) of a chaincode to the chaincode itself (deploy transaction) or these invocations to each other.

Additional requirements for any candidate solution is that it respects and supports the privacy and security provisions of the underlying membership service. In addition, it should not prevent the enforcement of any invocation access control of the chain-code functions in the fabric, or the implementation of enforcement of access-control mechanisms on the application (See Subsection 4.4).

In the following is provided the specification of transaction confidentiality mechanisms at the granularity of users. The last subsection provides some guidelines on how to extend this functionality at the level of validators. Information on the features supported in current release and its security provisions, you can find in Section 4.7.

The goal is to achieve a design that will allow for granting or restricting access to an entity to any subset of the following parts of a chain-code: 1. chaincode content, i.e., complete (source) code of the chaincode, 2. chaincode function headers, i.e., the prototypes of the functions included in a chaincode, 3. chaincode [invocations &] state, i.e., successive updates to the state of a specific chaincode, when one or more functions of its are invoked 4. all the above

Notice, that this design offers the application the capability to leverage the fabric's membership service infrastructure and its public key infrastructure to build their own access control policies and enforcement mechanisms.

4.3.2.1 Confidentiality against users

To support fine-grained confidentiality control, i.e., restrict read-access to the plain-text of a chaincode to a subset of users that the chaincode creator defines, a chain is bound to a single long-term encryption key-pair (PKchain, SKchain). Though initially this key-pair is to be stored and maintained by each chain's PKI, in later releases, however, this restriction will be moved away, as chains (and the associated key-pairs) can be triggered through the Blockchain by any user with *special* (admin) privileges (See, Section 4.3.2.2).

Setup. At enrollment phase, users obtain (as before) an enrollment certificate, denoted by Certui for user ui, while each validator vj obtain its enrollment certificate denoted by Certvj. Enrollment would grant users and validators the following credentials:

1. Users:

1. claim and grant themselves signing key-pair (spku, ssku),
2. claim and grant themselves encryption key-pair (epku, esku),
3. obtain the encryption (public) key of the chain PKchain

2. Validators:

1. claim and grant themselves signing key-pair (spkv, sskv),
2. claim and grant themselves an encryption key-pair (epkv, eskv),
3. obtain the decryption (secret) key of the chain SKchain

Thus, enrollment certificates contain the public part of two key-pairs: * one signature key-pair [denoted by (spkvj,sskvj) for validators and by (spkui, sskui) for users], and * an encryption key-pair [denoted by (epkvj,eskvj) for validators and (epkui, eskui) for users]

Chain, validator and user enrollment public keys are accessible to everyone.

In addition to enrollment certificates, users who wish to anonymously participate in transactions issue transaction certificates. For simplicity transaction certificates of a user ui are denoted by TCertui. Transaction certificates include the public part of a signature key-pair denoted by
(tpkui,tskui).

The following section provides a high level description of how transaction format accommodates read-access restrictions at the granularity of users.

Structure of deploy transaction. The following figure depicts the structure of a typical deploy transaction with confidentiality enabled.

One can notice that a deployment transaction consists of several sections: * Section *general-info*: contains the administration details of the transaction, i.e., which chain this transaction corresponds to (chained), the type of transaction (that is set to ‘’deplTrans‘‘), the version number of confidentiality policy implemented, its creator identifier (expressed by means of transaction certificate TCert of enrollment certificate Cert), and a Nonce, that facilitates primarily replay-attack resistance techniques. * Section *code-info*: contains information on the chain-code source code, and function headers. As shown in the figure below, there is a symmetric key used for the source-code of the chaincode (KC), and another symmetric key used for the function prototypes (KH). A signature of the creator of the chaincode is included on the plain-text code such that the latter cannot be detached from the transaction and replayed by another party. * Section *chain-validators*: where appropriate key material is passed to the validators for the latter to be able to (i) decrypt the chain-code source (KC), (ii) decrypt the headers, and (iii) encrypt the state when the chain-code has been invoked accordingly(KS). In particular, the chain-code creator generates an encryption key-pair for the chain-code it deploys (PKC, SKC). It then uses PKC to encrypt all the keys associated to the chain-code:

[(‘’code‘‘,KC) ,(‘’headr‘‘,KH),(‘’code-state‘‘,KS), SigTCertuc(*)]PKc,

and passes the secret key SKC to the validators using the chain-specific public key:

[(‘’chaincode‘‘,SKC), SigTCertuc(*)]PKchain.

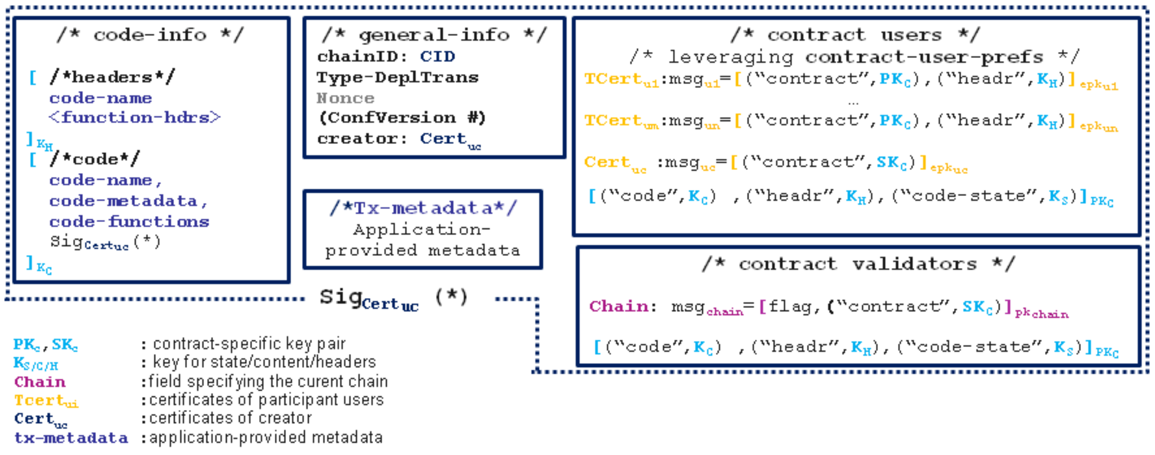

/* code-info */
[/*headers*/
code-name
<function-hdrs>
]KH
[/*code*/
code-name,
code-metadata,
code-functions
SigCertuc(*)
]KC
/* general-info */
chainID: CID
Type-DeplTrans
Nonce
(ConfVersion #)
creator: Certuc
/*Tx-metadata*/
Application-provided metadata
SigCertuc (*)
/* contract users */
/* leveraging contract-user-prefs */
TCertu1:msgu1=[("contract",PKC),("headr",KH)]epku1
...
TCertun:msgun=[("contract",PKC),("headr",KH)]epkun
Certuc :msguc=[("contract",SKC)]epkuc
[("code",KC) ,("headr",KH),("code-state",KS)]PKC
/* contract validators */
Chain: msgchain=[flag,("contract",SKC)]pkchain
[("code",KC) ,("headr",KH),("code-state",KS)]PKC
PKc,SKc : contract-specific key pair
KS/C/H : key for state/content/headers
Chain : field specifying the curent chain
Tcertui : certificates of participant users
Certuc : certificates of creator
tx-metadata : application-provided metadata

Fig. 11.9: FirstRelease-deploy

- Section *contract-users*: where the public encryption keys of the contract users, i.e., users who are given read-access to parts of the chaincode, are used to encrypt the keys associated to their access rights:

1. SKc for the users to be able to read any message associated to that chain-code (invocation, state, etc),
2. KC for the user to be able to read only the contract code,
3. KH for the user to only be able to read the headers,
4. KS for the user to be able to read the state associated to that contract.

Finally users are given the contract's public key PKc, for them to be able to encrypt information related to that contract for the validators (or any in possession of SKc) to be able to read it. Transaction certificate of each contract user is appended to the transaction and follows that user's message. This is done for users to be able to easily search the blockchain for transactions they have been part of. Notice that the deployment transaction also appends a message to the creator uc of the chain-code, for the latter to be able to retrieve this transaction through parsing the ledger and without keeping any state locally.

The entire transaction is signed by a certificate of the chaincode creator, i.e., enrollment or transaction certificate as decided by the latter. Two noteworthy points: * Messages that are included in a transaction in an encrypted format, i.e., code-functions, code-hdrs, are signed before they are encrypted using the same TCert the entire transaction is signed with, or even with a different TCert or the ECert of the user (if the transaction deployment should carry the identity of its owner. A binding to the underlying transaction carrier should be included in the signed message, e.g., the hash of the TCert the transaction is signed, such that mix&match attacks are not possible. Though we detail such attacks in Section 4.4, in these cases an attacker who sees a transaction should not be able to isolate the ciphertext corresponding to, e.g., code-info, and use it for another transaction of her own. Clearly, such an ability would disrupt the operation of the system, as a chaincode that was first created by user A, will now also belong to malicious user B (who is not even able to read it). * To offer the ability to the users to cross-verify they are given access to the correct key, i.e., to the same key as the other contract users, transaction ciphertexts that are encrypted with a key K are accompanied by a commitment to K, while the opening of this commitment value is passed to all users who are entitled access to K in contract-users, and chain-validator sections. In this way, anyone who is entitled access to that key can verify that the key has been properly passed to it. This part is omitted in the figure above to avoid confusion.

Structure of invoke transaction. A transaction invoking the chain-code triggering the execution of a function of the chain-code with user-specified arguments is structured as depicted in the figure below.

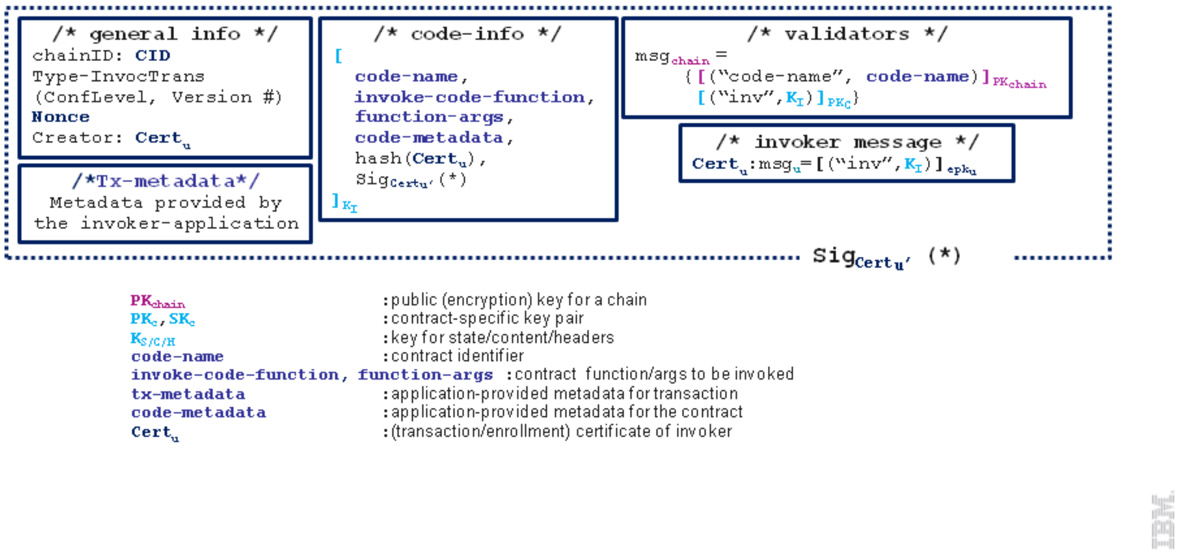

/* general info */
chainID: CID
Type-InvocTrans
(ConfLevel, Version #)
Nonce
Creator: Cert_u
/*Tx-metadata*/
Metadata provided by the invoker-application
/* code-info */
[
code-name,
invoke-code-function,
function-args,
code-metadata,
hash(Cert_u),
Sig_Cert_u' (*)
]_KI
/* validators */
msg_chain =
{[("code-name", code-name)]_PKchain
[("inv", KI)]_PKC}
/* invoker message */
Cert_u:msg_u=[("inv", KI)]_epk_u
Sig_Cert_u' (*)
PKchain : public (encryption) key for a chain
PKc, SKc : contract-specific key pair
KS/C/H : key for state/content/headers
code-name : contract identifier
invoke-code-function, function-args : contract function/args to be invoked
tx-metadata : application-provided metadata for transaction
code-metadata : application-provided metadata for the contract
Cert_u : (transaction/enrollment) certificate of invoker
IBM

Fig. 11.10: FirstRelease-deploy

Invocation transaction as in the case of deployment transaction consists of a *general-info* section, a *code-info* section, a section for the *chain-validators*, and one for the *contract users*, signed altogether with one of the invoker's transaction certificates.

- General-info follows the same structure as the corresponding section of the deployment transaction. The only difference relates to the transaction type that is now set to ''InvocTx'', and the chain-code identifier or name that is now encrypted under the chain-specific encryption (public) key.
- Code-info exhibits the same structure as the one of the deployment transaction. Code payload, as in the case of deployment transaction, consists of function invocation details (the name of the function invoked, and associated arguments), code-metadata provided by the application and the transaction's creator (invoker's u) certificate, TCertu. Code payload is signed by the transaction certificate TCertu of the invoker u, as in the case of deploy transactions. As in the case of deploy transactions, code-metadata, and tx-metadata, are fields that are provided by the application and can be used (as described in Section 4.4), for the latter to implement their own access control mechanisms and roles.
- Finally, contract-users and chain-validator sections provide the key the payload is encrypted with, the invoker's key, and the chain encryption key respectively. Upon receiving such transactions, the validators decrypt [code-name]PKchain using the chain-specific secret key SKchain and obtain the invoked chain-code identifier. Given the latter, validators retrieve from their local storage the chaincode's decryption key SKc, and use it to decrypt chain-validators' message, that would equip them with the symmetric key KI the invocation transaction's payload was encrypted with. Given the latter, validators decrypt code-info, and execute the chain-code function with the specified arguments, and the code-metadata attached(See, Section 4.4 for more details on the use of code-metadata). While the chain-code is executed, updates of the state of that chain-code are possible. These are encrypted using the state-specific key Ks that was defined during that chain-code's deployment. In particular, Ks is used the same way KiTx is used in the design of our current release (See, Section 4.7).

Structure of query transaction. Query transactions have the same format as invoke transactions. The only difference is that Query transactions do not affect the state of the chaincode, and thus there is no need for the state to be retrieved (decrypted) and/or updated (encrypted) after the execution of the chaincode completes.

4.3.2.2 Confidentiality against validators

This section deals with ways of how to support execution of certain transactions under a different (or subset) sets of validators in the current chain. This section inhibits IP restrictions and will be expanded in the following few weeks.

4.3.3 Replay attack resistance

In replay attacks the attacker "replays" a message it "eavesdropped" on the network or ''saw" on the Blockchain. Replay attacks are a big problem here, as they can incur into the validating entities re-doing a computationally intensive process (chaincode invocation) and/or affect the state of the corresponding chaincode, while it requires minimal or no power from the attacker side. To make matters worse, if a transaction was a payment transaction, replays could potentially incur into the payment being performed more than once, without this being the original intention of the payer. Existing systems resist replay attacks as follows: * Record hashes of transactions in the system. This solution would require that validators maintain a log of the hash of each transaction that has ever been announced through the network, and compare a new transaction against their locally stored transaction record. Clearly such approach cannot scale for large networks, and could easily result into validators spending a lot of time to do the check of whether a transaction has been replayed, than executing the actual transaction. * Leverage state that is maintained per user identity (Ethereum). Ethereum keeps some state, e.g., counter (initially set to 1) for each identity/pseudonym in the system. Users also maintain their own counter (initially set to 0) for each identity/pseudonym of theirs. Each time a user sends a transaction using an identity/pseudonym of his, he increases his local counter by one and adds the resulting value to the transaction. The transaction is subsequently signed by that user identity and released to the network. When picking up this transaction, validators check the counter value included within and compare it with the one they have stored locally; if the value is the same, they increase the local value of that identity's counter and accept the transaction. Otherwise, they reject the transaction as invalid or replay. Although this would work well in cases where we have limited number of user identities/pseudonyms (e.g., not too large), it would ultimately not scale in a system where users use a different identifier (transaction certificate) per transaction, and thus have a number of user pseudonyms proportional to the number of transactions.

Other asset management systems, e.g., Bitcoin, though not directly dealing with replay attacks, they resist them. In systems that manage (digital) assets, state is maintained on a per asset basis, i.e., validators only keep a record of who owns what. Resistance to replay attacks come as a direct result from this, as replays of transactions would be immediately be deemed as invalid by the protocol (since can only be shown to be derived from older owners of an asset/coin). While this would be appropriate for asset management systems, this does not abide with the needs of a Blockchain systems with more generic use than asset management.

In the fabric, replay attack protection uses a hybrid approach. That is, users add in the transaction a nonce that is generated in a different manner depending on whether the transaction is anonymous (followed and signed by a transaction certificate) or not (followed and signed by a long term enrollment certificate). More specifically:

- Users submitting a transaction with their enrollment certificate should include in that transaction a nonce that is a function of the nonce they used in the previous transaction they issued with the same certificate (e.g., a counter function or a hash). The nonce included in the first transaction of each enrollment certificate can be either pre-fixed by the system (e.g., included in the genesis block) or chosen by the user. In the first case, the genesis block would need to include nonceall , i.e., a fixed number and the nonce used by user with identity IDA for his first enrollment certificate signed transaction would be

  nonceround0IDA <- hash(IDA, nonceall),

  where IDA appears in the enrollment certificate. From that point onward successive transactions of that user with enrollment certificate would include a nonce as follows nonceroundiIDA <- hash(nonceround{i-1}IDA),

that is the nonce of the ith transaction would be using the hash of the nonce used in the {i-1}th transaction of that certificate. Validators here continue to process a transaction they receive, as long as it satisfies the condition mentioned above. Upon successful validation of transaction's format, the validators update their database with that nonce.

Storage overhead:

1. on the user side: only the most recently used nonce,
2. on validator side: O(n), where n is the number of users.

- Users submitting a transaction with a transaction certificate should include in the transaction a random nonce, that would guarantee that two transactions do not result into the same hash. Validators add the hash of this transaction in their local database if the transaction certificate used within it has not expired. To avoid storing large amounts of hashes, validity periods of transaction certificates are leveraged. In particular validators maintain an updated record of received transactions' hashes within the current or future validity period.

Storage overhead (only makes sense for validators here): O(m), where m is the approximate number of transactions within a validity period and corresponding validity period identifier (see below).

4.4 Access control features on the application

An application, is a piece of software that runs on top of a Blockchain client software, and, performs a special task over the Blockchain, i.e., restaurant table reservation. Application software have a version of developer, enabling the latter to generate and manage a couple of chaincodes that are necessary for the business this application serves, and a client-version that would allow the application's end-users to make use of the application, by invoking these chain-codes. The use of the Blockchain can be transparent to the application end-users or not.

This section describes how an application leveraging chaincodes can implement its own access control policies, and guidelines on how our Membership services PKI can be leveraged for the same purpose.

The presentation is divided into enforcement of invocation access control, and enforcement of read-access control by the application.

4.4.1 Invocation access control

To allow the application to implement its own invocation access control at the application layer securely, special support by the fabric must be provided. In the following we elaborate on the tools exposed by the fabric to the application for this purpose, and provide guidelines on how these should be used by the application for the latter to enforce access control securely.

Support from the infrastructure. For the chaincode creator, let it be, *uc*, to be able to implement its own invocation access control at the application layer securely, special support by the fabric must be provided. More specifically fabric layer gives access to following capabilities:

1. The client-application can request the fabric to sign and verify any message with specific transaction certificates or enrollment certificate the client owns; this is expressed via the Certificate Handler interface
2. The client-application can request the fabric a unique *binding* to be used to bind authentication data of the application to the underlying transaction transporting it; this is expressed via the Transaction Handler interface
3. Support for a transaction format, that allows for the application to specify metadata, that are passed to the chain-code at deployment, and invocation time; the latter denoted by code-metadata.

The **Certificate Handler** interface allows to sign and verify any message using signing key-pair underlying the associated certificate. The certificate can be a TCert or an ECert.

```
// CertificateHandler exposes methods to deal with an ECert/TCert
type CertificateHandler interface {

    // GetCertificate returns the certificate's DER
    GetCertificate() []byte

    // Sign signs msg using the signing key corresponding to the certificate
    Sign(msg []byte) ([]byte, error)

    // Verify verifies msg using the verifying key corresponding to the certificate
    Verify(signature []byte, msg []byte) error

    // GetTransactionHandler returns a new transaction handler relative to this␣
↪certificate
    GetTransactionHandler() (TransactionHandler, error)
}
```

The **Transaction Handler** interface allows to create transactions and give access to the underlying *binding* that can be leveraged to link application data to the underlying transaction. Bindings are a concept that have been introduced in network transport protocols (See, https://tools.ietf.org/html/rfc5056), known as *channel bindings*, that *allows applications to establish that the two end-points of a secure channel at one network layer are the same as at a higher layer by binding authentication at the higher layer to the channel at the lower layer. This allows applications to delegate session protection to lower layers, which has various performance benefits.* Transaction bindings offer the ability to uniquely identify the fabric layer of the transaction that serves as the container that application data uses to be added to the ledger.

```
// TransactionHandler represents a single transaction that can be uniquely determined␣
↪or identified by the output of the GetBinding method.
// This transaction is linked to a single Certificate (TCert or ECert).
type TransactionHandler interface {

    // GetCertificateHandler returns the certificate handler relative to the␣
↪certificate mapped to this transaction
    GetCertificateHandler() (CertificateHandler, error)

    // GetBinding returns a binding to the underlying transaction (container)
    GetBinding() ([]byte, error)

    // NewChaincodeDeployTransaction is used to deploy chaincode
    NewChaincodeDeployTransaction(chaincodeDeploymentSpec *obc.
↪ChaincodeDeploymentSpec, uuid string) (*obc.Transaction, error)

    // NewChaincodeExecute is used to execute chaincode's functions
    NewChaincodeExecute(chaincodeInvocation *obc.ChaincodeInvocationSpec, uuid␣
↪string) (*obc.Transaction, error)

    // NewChaincodeQuery is used to query chaincode's functions
    NewChaincodeQuery(chaincodeInvocation *obc.ChaincodeInvocationSpec, uuid string)␣
↪(*obc.Transaction, error)
}
```

For version 1, *binding* consists of the *hash*(TCert, Nonce), where TCert, is the transaction certificate used to sign the entire transaction, while Nonce, is the nonce number used within.

The **Client** interface is more generic, and offers a mean to get instances of the previous interfaces.

```
type Client interface {

    ...

    // GetEnrollmentCertHandler returns a CertificateHandler whose certificate is the␣
↪enrollment certificate
    GetEnrollmentCertificateHandler() (CertificateHandler, error)

    // GetTCertHandlerNext returns a CertificateHandler whose certificate is the next␣
↪available TCert
    GetTCertificateHandlerNext() (CertificateHandler, error)

    // GetTCertHandlerFromDER returns a CertificateHandler whose certificate is the␣
↪one passed
    GetTCertificateHandlerFromDER(der []byte) (CertificateHandler, error)

}
```

To support application-level access control lists for controlling chaincode invocation, the fabric's transaction and chaincode specification format have an additional field to store application-specific metadata. This field is depicted in both figures 1, by code-metadata. The content of this field is decided by the application, at the transaction creation time. The fabric layer treats it as an unstructured stream of bytes.

```
message ChaincodeSpec {

    ...

    ConfidentialityLevel confidentialityLevel;
    bytes metadata;

    ...
}


message Transaction {
    ...

    bytes payload;
    bytes metadata;

    ...
}
```

To assist chaincode execution, at the chain-code invocation time, the validators provide the chaincode with additional information, like the metadata and the binding.

Application invocation access control. This section describes how the application can leverage the means provided by the fabric to implement its own access control on its chain-code functions. In the scenario considered here, the following entities are identified:

1. **C**: is a chaincode that contains a single function, e.g., called *hello*;
2. **uc**: is the **C** deployer;
3. **ui**: is a user who is authorized to invoke **C**'s functions. User uc wants to ensure that only ui can invoke the function *hello*.

Deployment of a Chaincode: At deployment time, uc has full control on the deployment transaction's metadata, and can be used to store a list of ACLs (one per function), or a list of roles that are needed by the application. The format which is used to store these ACLs is up to the deployer’s application, as the chain-code is the one who would need to parse the metadata at execution time. To define each of these lists/roles, uc can use any TCerts/Certs of the ui (or, if applicable, or other users who have been assigned that privilege or role). Let this be TCertui. The exchange of TCerts or Certs among the developer and authorized users is done through an out-of-band channel.

Assume that the application of uc’s requires that to invoke the *hello* function, a certain message *M* has to be authenticated by an authorized invoker (ui, in our example). One can distinguish the following two cases:

1. *M* is one of the chaincode’s function arguments;
2. *M* is the invocation message itself, i.e., function-name, function-arguments.

Chaincode invocation: To invoke C, ui’s application needs to sign *M* using the TCert/ECert, that was used to identify ui’s participation in the chain-code at the associated deployment transaction’s metadata, i.e., TCertui. More specifically, ui’s client application does the following:

1. Retrieves a CertificateHandler for Certui, *cHandler*;
2. obtains a new TransactionHandler to issue the execute transaction, *txHandler* relative to his next available TCert or his ECert;
3. gets *txHandler*‘s *binding* by invoking *txHandler.getBinding()*;
4. signs ‘M* || txBinding’* by invoking *cHandler.Sign(* ‘M* || txBinding’)*, let *sigma* be the output of the signing function;
5. issues a new execute transaction by invoking, *txHandler.NewChaincodeExecute(...)*. Now, *sigma* can be included in the transaction as one of the arguments that are passed to the function (case 1) or as part of the code-metadata section of the payload(case 2).

Chaincode processing: The validators, who receive the execute transaction issued ui, will provide to *hello* the following information:

1. The *binding* of the execute transaction, that can be independently computed at the validator side;
2. The *metadata* of the execute transaction (code-metadata section of the transaction);
3. The *metadata* of the deploy transaction (code-metadata component of the corresponding deployment transaction).

Notice that *sigma* is either part of the arguments of the invoked function, or stored inside the code-metadata of the invocation transaction (properly formatted by the client-application). Application ACLs are included in the code-metadata section, that is also passed to the chain-code at execution time. Function *hello* is responsible for checking that *sigma* is indeed a valid signature issued by TCertui, on ‘*M || txBinding’*.

4.4.2 Read access control

This section describes how the fabric’s infrastructure offers support to the application to enforce its own read-access control policies at the level of users. As in the case of invocation access control, the first part describes the infrastructure features that can be leveraged by the application for this purpose, and the last part details on the way applications should use these tools.

For the purpose of this discussion, we leverage a similar example as before, i.e.,

1. **C**: is a chaincode that contains a single function, e.g., called *hello*;
2. **uA**: is the **C**‘s deployer, also known as application;
3. **ur**: is a user who is authorized to read **C**‘s functions. User uA wants to ensure that only ur can read the function *hello*.

Support from the infrastructure. For **uA** to be able to implement its own read access control at the application layer securely, our infrastructure is required to support the transaction format for code deployment and invocation, as depicted in the two figures below.

More specifically fabric layer is required to provide the following functionality:

1. Provide minimal encryption capability such that data is only decryptable by a validator's (infrastructure) side; this means that the infrastructure should move closer to our future version, where an asymmetric encryption scheme is used for encrypting transactions. More specifically, an asymmetric key-pair is used for the chain, denoted by Kchain in the Figures above, but detailed in Section Transaction Confidentiality.
2. The client-application can request the infrastructure sitting on the client-side to encrypt/decrypt information using a specific public encryption key, or that client's long-term decryption key.
3. The transaction format offers the ability to the application to store additional transaction metadata, that can be passed to the client-application after the latter's request. Transaction metadata, as opposed to code-metadata, is not encrypted or provided to the chain-code at execution time. Validators treat these metadata as a list of bytes they are not responsible for checking validity of.

Application read-access control. For this reason the application may request and obtain access to the public encryption key of the user **ur**; let that be **PKur**. Optionally, **ur** may be providing **uA** with a certificate of its, that would be leveraged by the application, say, TCertur; given the latter, the application would, e.g., be able to trace that user's transactions w.r.t. the application's chain-codes. TCertur, and PKur, are exchanged in an out-of-band channel.

At deployment time, application **uA** performs the following steps:

1. Uses the underlying infrastructure to encrypt the information of **C**, the application would like to make accessible to **ur**, using PKur. Let Cur be the resulting ciphertext.
2. (optional) Cur can be concatenated with TCertur
3. Passes the overall string as ''Tx-metadata" of the confidential transaction to be constructed.

At invocation time, the client-application on ur's node, would be able, by obtaining the deployment transaction to retrieve the content of **C**. It just needs to retrieve the **tx-metadata** field of the associated deployment transaction, and trigger the decryption functionality offered by our Blockchain infrastrucure's client, for Cur. Notice that it is the application's responsibility to encrypt the correct **C** for ur. Also, the use of **tx-metadata** field can be generalized to accommodate application-needs. E.g., it can be that invokers leverage the same field of invocation transactions to pass information to the developer of the application, etc.

Important Note: It is essential to note that validators **do not provide** any decryption oracle to the chain-code throughout its execution. Its infrastructure is though responsible for decrypting the payload of the chain-code itself (as well as the code-metadata fields near it), and provide those to containers for deployment/execution.

4.5 Online wallet service

This section describes the security design of a wallet service, which in this case is a node with which end-users can register, store their key material and through which they can perform transactions. Because the wallet service is in possession of the user's key material, it is clear that without a secure authorization mechanism in place a malicious wallet service could successfully impersonate the user. We thus emphasize that this design corresponds to a wallet service that is **trusted** to only perform transactions on behalf of its clients, with the consent of the latter. There are two cases for the registration of an end-user to an online wallet service:

1. When the user has registered with the registration authority and acquired his/her `<enrollID,enrollPWD>`, but has not installed the client to trigger and complete the enrollment process;
2. When the user has already installed the client, and completed the enrollment phase.

Initially, the user interacts with the online wallet service to issue credentials that would allow him to authenticate to the wallet service. That is, the user is given a username, and password, where username identifies the user in the membership service, denoted by AccPub, and password is the associated secret, denoted by AccSec, that is **shared** by both user and service.

To enroll through the online wallet service, a user must provide the following request object to the wallet service:

```
AccountRequest /* account request of u \*/
{
    OBCSecCtx ,           /* credentials associated to network \*/
    AccPub<sub>u</sub>,   /* account identifier of u \*/
    AccSecProof<sub>u</sub>  /* proof of AccSec<sub>u</sub>\*/
 }
```

OBCSecCtx refers to user credentials, which depending on the stage of his enrollment process, can be either his enrollment ID and password, `<enrollID,enrollPWD>` or his enrollment certificate and associated secret key(s) (ECertu, sku), where sku denotes for simplicity signing and decryption secret of the user. The content of AccSecProofu is an HMAC on the rest fields of request using the shared secret. Nonce-based methods similar to what we have in the fabric can be used to protect against replays. OBCSecCtx would give the online wallet service the necessary information to enroll the user or issue required TCerts.

For subsequent requests, the user u should provide to the wallet service a request of similar format.

```
TransactionRequest /* account request of u \*/
{
     TxDetails,            /* specifications for the new transaction \*/
     AccPub<sub>u</sub>,       /* account identifier of u \*/
     AccSecProof<sub>u</sub>   /* proof of AccSec<sub>u</sub> \*/
}
```

Here, TxDetails refer to the information needed by the online service to construct a transaction on behalf of the user, i.e., the type, and user-specified content of the transaction.

AccSecProofu is again an HMAC on the rest fields of request using the shared secret. Nonce-based methods similar to what we have in the fabric can be used to protect against replays.

TLS connections can be used in each case with server side authentication to secure the request at the network layer (confidentiality, replay attack protection, etc)

4.6 Network security (TLS)

The TLS CA should be capable of issuing TLS certificates to (non-validating) peers, validators, and individual clients (or browsers capable of storing a private key). Preferably, these certificates are distinguished by type, per above. TLS certificates for CAs of the various types (such as TLS CA, ECA, TCA) could be issued by an intermediate CA (i.e., a CA that is subordinate to the root CA). Where there is not a particular traffic analysis issue, any given TLS connection can be mutually authenticated, except for requests to the TLS CA for TLS certificates.

In the current implementation the only trust anchor is the TLS CA self-signed certificate in order to accommodate the limitation of a single port to communicate with all three (co-located) servers, i.e., the TLS CA, the TCA and the ECA. Consequently, the TLS handshake is established with the TLS CA, which passes the resultant session keys to the co-located TCA and ECA. The trust in validity of the TCA and ECA self-signed certificates is therefore inherited from trust in the TLS CA. In an implementation that does not thus elevate the TLS CA above other CAs, the trust anchor should be replaced with a root CA under which the TLS CA and all other CAs are certified.

4.7 Restrictions in the current release

This section lists the restrictions of the current release of the fabric. A particular focus is given on client operations and the design of transaction confidentiality, as depicted in Sections 4.7.1 and 4.7.2.

- Client side enrollment and transaction creation is performed entirely by a non-validating peer that is trusted not to impersonate the user. See, Section 4.7.1 for more information.
- A minimal set of confidentiality properties where a chaincode is accessible by any entity that is member of the system, i.e., validators and users who have registered through Hyperledger Fabric's Membership Services and is not accessible by anyone else. The latter include any party that has access to the storage area where the ledger is maintained, or other entities that are able to see the transactions that are announced in the validator network. The design of the first release is detailed in subsection 4.7.2
- The code utilizes self-signed certificates for entities such as the enrollment CA (ECA) and the transaction CA (TCA)
- Replay attack resistance mechanism is not available
- Invocation access control can be enforced at the application layer: it is up to the application to leverage the infrastructure's tools properly for security to be guaranteed. This means, that if the application fails to *bind* the transaction binding offered by the fabric, secure transaction processing may be at risk.

4.7.1 Simplified client

Client-side enrollment and transaction creation are performed entirely by a non-validating peer that plays the role of an online wallet. In particular, the end-user leverages their registration credentials to open an account to a non-validating peer and uses these credentials to further authorize the peer to build transactions on the user's behalf. It needs to be noted, that such a design does not provide secure **authorization** for the peer to submit transactions on behalf of the user, as a malicious peer could impersonate the user. Details on the specifications of a design that deals with the security issues of online wallet can be found is Section 4.5. Currently the maximum number of peers a user can register to and perform transactions through is one.

4.7.2 Simplified transaction confidentiality

Disclaimer: The current version of transaction confidentiality is minimal, and will be used as an intermediate step to reach a design that allows for fine grained (invocation) access control enforcement in a subsequent release.

In its current form, confidentiality of transactions is offered solely at the chain-level, i.e., that the content of a transaction included in a ledger, is readable by all members of that chain, i.e., validators and users. At the same time, application auditors who are not members of the system can be given the means to perform auditing by passively observing the blockchain data, while guaranteeing that they are given access solely to the transactions related to the application under audit. State is encrypted in a way that such auditing requirements are satisfied, while not disrupting the proper operation of the underlying consensus network.

More specifically, currently symmetric key encryption is supported in the process of offering transaction confidentiality. In this setting, one of the main challenges that is specific to the blockchain setting, is that validators need to run consensus over the state of the blockchain, that, aside from the transactions themselves, also includes the state updates of individual contracts or chaincode. Though this is trivial to do for non-confidential chaincode, for confidential chaincode, one needs to design the state encryption mechanism such that the resulting ciphertexts are semantically secure, and yet, identical if the plaintext state is the same.

To overcome this challenge, the fabric utilizes a key hierarchy that reduces the number of ciphertexts that are encrypted under the same key. At the same time, as some of these keys are used for the generation of IVs, this allows the validating parties to generate exactly the same ciphertext when executing the same transaction (this is necessary to remain agnostic to the underlying consensus algorithm) and offers the possibility of controlling audit by disclosing to auditing entities only the most relevant keys.

Method description: Membership service generates a symmetric key for the ledger (Kchain) that is distributed at registration time to all the entities of the blockchain system, i.e., the clients and the validating entities that have issued credentials through the membership service of the chain. At enrollment phase, user obtain (as before) an enrollment certificate, denoted by Certui for user ui , while each validator vj obtains its enrollment certificate denoted by Certvj.

Entity enrollment would be enhanced, as follows. In addition to enrollment certificates, users who wish to anonymously participate in transactions issue transaction certificates. For simplicity transaction certificates of a user ui are denoted by TCertui. Transaction certificates include the public part of a signature key-pair denoted by (tpkui,tskui).

In order to defeat crypto-analysis and enforce confidentiality, the following key hierarchy is considered for generation and validation of confidential transactions: To submit a confidential transaction (Tx) to the ledger, a client first samples a nonce (N), which is required to be unique among all the transactions submitted to the blockchain, and derive a transaction symmetric key (KTx) by applying the HMAC function keyed with Kchain and on input the nonce, KTx= HMAC(Kchain, N). From KTx, the client derives two AES keys: KTxCID as HMAC(KTx, c1), KTxP as HMAC(KTx, c2)) to encrypt respectively the chain-code name or identifier CID and code (or payload) P. c1, c2 are public constants. The nonce, the Encrypted Chaincode ID (ECID) and the Encrypted Payload (EP) are added in the transaction Tx structure, that is finally signed and so authenticated. Figure below shows how encryption keys for the client's transaction are generated. Arrows in this figure denote application of an HMAC, keyed by the key at the source of the arrow and using the number in the arrow as argument. Deployment/Invocation transactions' keys are indicated by d/i respectively.

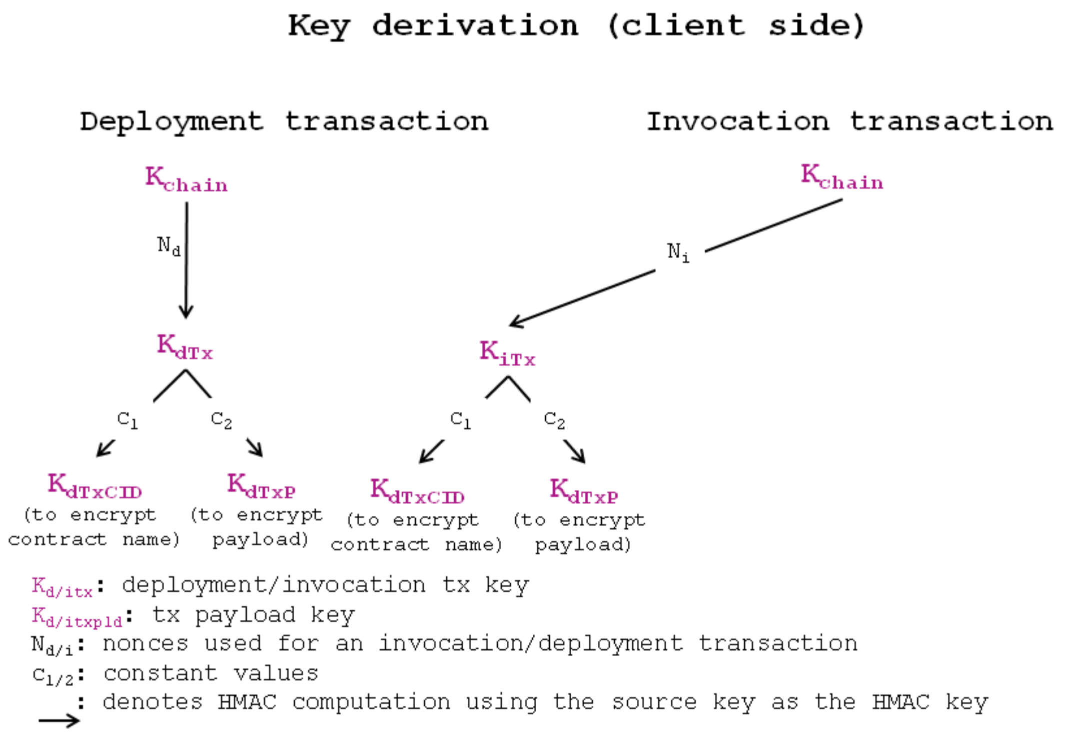

Key derivation (client side)
Deployment transaction
Invocation transaction
K_chain
N_d
N_i
K_dTx
K_iTx
c_1
c_2
K_dTxCID
(to encrypt contract name)
K_dTxP
(to encrypt payload)
K_d/itx: deployment/invocation tx key
K_d/itxpld: tx payload key
N_d/i: nonces used for an invocation/deployment transaction
c_1/2: constant values
: denotes HMAC computation using the source key as the HMAC key

Fig. 11.11: FirstRelease-clientSide

To validate a confidential transaction Tx submitted to the blockchain by a client, a validating entity first decrypts ECID and EP by re-deriving KTxCID and KTxP from Kchain and Tx.Nonce as done before. Once the Chaincode ID and the Payload are recovered the transaction can be processed.

When V validates a confidential transaction, the corresponding chaincode can access and modify the chaincode's state. V keeps the chaincode's state encrypted. In order to do so, V generates symmetric keys as depicted in the figure above.

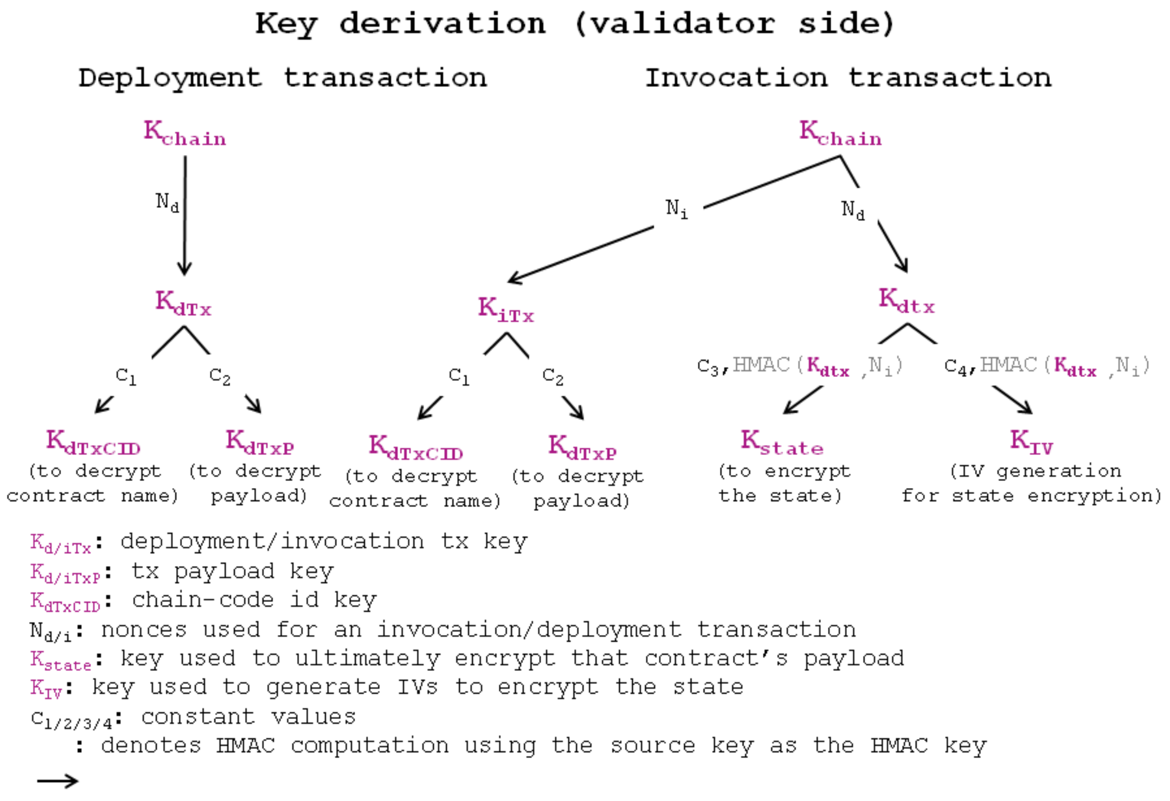

Key derivation (validator side)
Deployment transaction
Invocation transaction
K_{chain}
N_d
K_{dTx}
c_1
c_2
K_{dTxCID}
(to decrypt contract name)
K_{dTxP}
(to decrypt payload)
K_{chain}
N_i
N_d
K_{iTx}
c_1
c_2
K_{dTxCID}
(to decrypt contract name)
K_{dTxP}
(to decrypt payload)
K_{dtx}
c_3, HMAC(K_{dtx}, N_i)
c_4, HMAC(K_{dtx}, N_i)
K_{state}
(to encrypt the state)
K_{IV}
(IV generation for state encryption)
$K_{d/iTx}$: deployment/invocation tx key
$K_{d/iTxP}$: tx payload key
K_{dTxCID}: chain-code id key
$N_{d/i}$: nonces used for an invocation/deployment transaction
K_{state}: key used to ultimately encrypt that contract's payload
K_{IV}: key used to generate IVs to encrypt the state
$c_{1/2/3/4}$: constant values
$\longrightarrow$: denotes HMAC computation using the source key as the HMAC key

Fig. 11.12: FirstRelease-validatorSide

Let iTx be a confidential transaction invoking a function deployed at an early stage by the confidential transaction dTx (notice that iTx can be dTx itself in the case, for example, that dTx has a setup function that initializes the chaincode's state). Then, V generates two symmetric keys KIV and Kstate as follows:

1. It computes as KdTx , i.e., the transaction key of the corresponding deployment transaction, and then Nstate = HMAC(Kdtx ,hash(Ni)), where Ni is the nonce appearing in the invocation transaction, and *hash* a hash function.
2. It sets Kstate = HMAC(KdTx, c3 || Nstate), truncated opportunely deeding on the underlying cipher used to encrypt; c3 is a constant number
3. It sets KIV = HMAC(KdTx, c4 || Nstate); c4 is a constant number

In order to encrypt a state variable S, a validator first generates the IV as HMAC(KIV, crtstate) properly truncated, where crtstate is a counter value that increases each time a state update is requested for the same chaincode invocation. The counter is discarded after the execution of the chaincode terminates. After IV has been generated, V encrypts with authentication (i.e., GSM mode) the value of S concatenated with Nstate(Actually, Nstate doesn't need to be encrypted but only authenticated). To the resulting ciphertext (CT), Nstate and the IV used is appended. In order to decrypt an encrypted state CT|| Nstate' , a validator first generates the symmetric keys KdTX' ,Kstate' using Nstate' and then decrypts CT.

Generation of IVs: In order to be agnostic to any underlying consensus algorithm, all the validating parties need a method to produce the same exact ciphertexts. In order to do so, the validators need to use the same IVs. Reusing the same IV with the same symmetric key completely breaks the security of the underlying cipher. Therefore, the process described before is followed. In particular, V first derives an IV generation key KIV by computing HMAC(KdTX, c4 || Nstate), where c4 is a constant number, and keeps a counter crtstate for the pair (dTx, iTx) with is initially set to 0. Then, each time a new ciphertext has to be generated, the validator generates a new IV by computing it as the output of HMAC(KIV, crtstate) and then increments the crtstate by one.

Another benefit that comes with the above key hierarchy is the ability to enable controlled auditing. For example, while by releasing Kchain one would provide read access to the whole chain, by releasing only Kstate for a given pair of transactions (dTx,iTx) access would be granted to a state updated by iTx, and so on.

The following figures demonstrate the format of a deployment and invocation transaction currently available in the code.

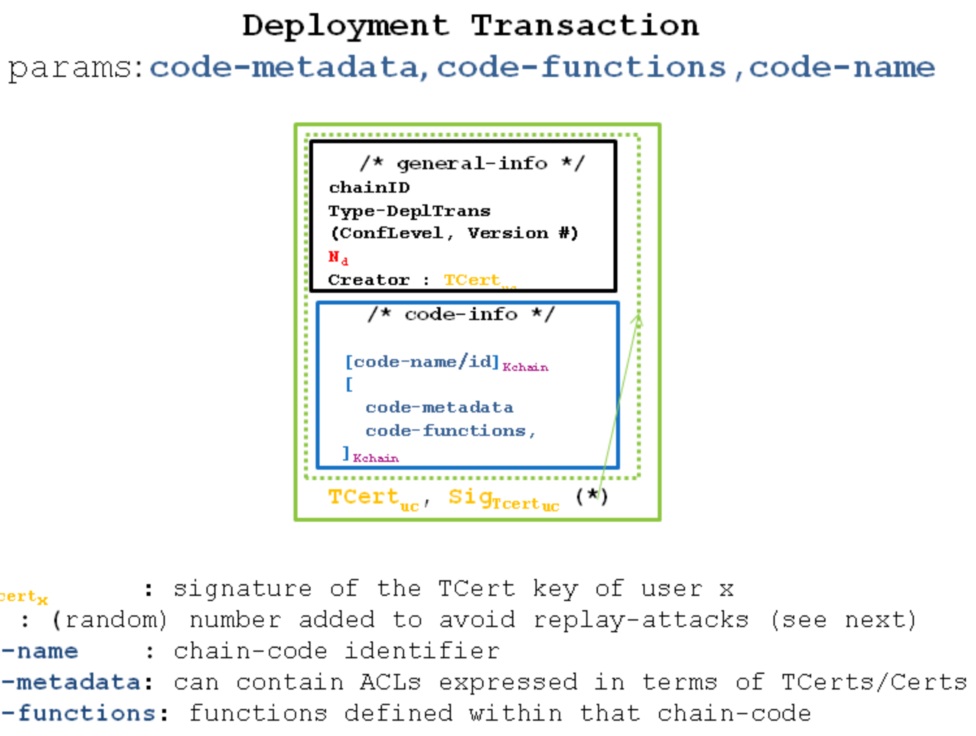
One can notice that both deployment and invocation transactions consist of two sections:

- Section *general-info*: contains the administration details of the transaction, i.e., which chain this transaction corresponds to (is chained to), the type of transaction (that is set to ''deploymTx'' or ''invocTx''), the version number of confidentiality policy implemented, its creator identifier (expressed by means of TCert of Cert) and a nonce (facilitates primarily replay-attack resistance techniques).
- Section *code-info*: contains information on the chain-code source code. For deployment transaction this is essentially the chain-code identifier/name and source code, while for invocation chain-code is the name of the function invoked and its arguments. As shown in the two figures code-info in both transactions are encrypted ultimately using the chain-specific symmetric key Kchain.

5. Byzantine Consensus

The `pbft` package is an implementation of the seminal PBFT consensus protocol [1], which provides consensus among validators despite a threshold of validators acting as *Byzantine*, i.e., being malicious or failing in an unpredictable manner. In the default configuration, PBFT tolerates up to t<n/3 Byzantine validators.

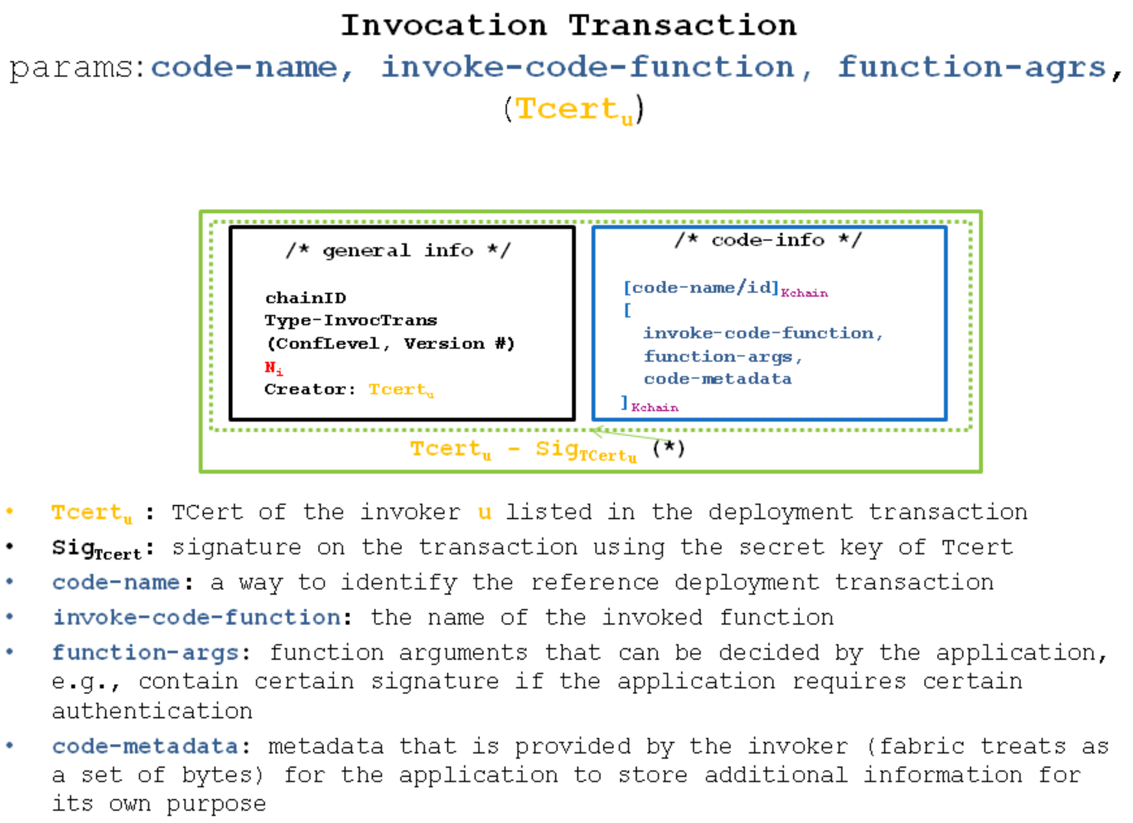
In the default configuration, PBFT is designed to run on at least *3t+1* validators (replicas), tolerating up to *t* potentially faulty (including malicious, or *Byzantine*) replicas.


Deployment Transaction

params: code-metadata, code-functions, code-name

- Sig_{Tcert_x} : signature of the TCert key of user x
- N_d : (random) number added to avoid replay-attacks (see next)
- code-name : chain-code identifier
- code-metadata: can contain ACLs expressed in terms of TCerts/Certs
- code-functions: functions defined within that chain-code

Fig. 11.13: FirstRelease-deploy



- Tcert$_u$: TCert of the invoker u listed in the deployment transaction
- Sig$_{Tcert}$: signature on the transaction using the secret key of Tcert
- code-name: a way to identify the reference deployment transaction
- invoke-code-function: the name of the invoked function
- function-args: function arguments that can be decided by the application, e.g., contain certain signature if the application requires certain authentication
- code-metadata: metadata that is provided by the invoker (fabric treats as a set of bytes) for the application to store additional information for its own purpose

Fig. 11.14: FirstRelease-deploy

5.1 Overview

The `pbft` plugin provides an implementation of the PBFT consensus protocol.

5.2 Core PBFT Functions

The following functions control for parallelism using a non-recursive lock and can therefore be invoked from multiple threads in parallel. However, the functions typically run to completion and may invoke functions from the CPI passed in. Care must be taken to prevent livelocks.

5.2.1 newPbftCore

Signature:

```
func newPbftCore(id uint64, config *viper.Viper, consumer innerCPI, ledger consensus.
↪Ledger) *pbftCore
```

The `newPbftCore` constructor instantiates a new PBFT box instance, with the specified `id`. The `config` argument defines operating parameters of the PBFT network: number replicas *N*, checkpoint period *K*, and the timeouts for request completion and view change duration.

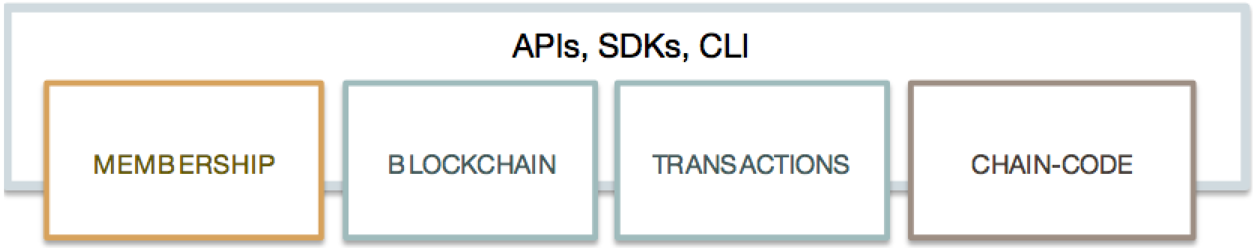
configuration key	type	example value	description
`general.N`	*integer*	4	Number of replicas
`general.K`	*integer*	10	Checkpoint period
`general.timeout.request`	*duration*	2s	Max delay between request reception and execution
`general.timeout.viewchange`	*duration*	2s	Max delay between view-change start and next request execution

The arguments `consumer` and `ledger` pass in interfaces that are used to query the application state and invoke application requests once they have been totally ordered. See the respective sections below for these interfaces.

6. Application Programming Interface

The primary interface to the fabric is a REST API. The REST API allows applications to register users, query the blockchain, and to issue transactions. A CLI is also provided to cover a subset of the available APIs for development purposes. The CLI enables developers to quickly test chaincodes or query for status of transactions.

Applications interact with a non-validating peer node through the REST API, which will require some form of authentication to ensure the entity has proper privileges. The application is responsible for implementing the appropriate authentication mechanism and the peer node will subsequently sign the outgoing messages with the client identity.



The fabric API design covers the categories below, though the implementation is incomplete for some of them in the current release. The *REST API* section will describe the APIs currently supported.

- Identity - Enrollment to acquire or to revoke a certificate
- Address - Target and source of a transaction
- Transaction - Unit of execution on the ledger
- Chaincode - Program running on the ledger
- Blockchain - Contents of the ledger
- Network - Information about the blockchain peer network
- Storage - External store for files or documents
- Event Stream - Sub/pub events on the blockchain

6.1 REST Service

The REST service can be enabled (via configuration) on either validating or non-validating peers, but it is recommended to only enable the REST service on non-validating peers on production networks.

```
func StartOpenchainRESTServer(server *oc.ServerOpenchain, devops *oc.Devops)
```

This function reads the `rest.address` value in the `core.yaml` configuration file, which is the configuration file for the `peer` process. The value of the `rest.address` key defines the default address and port on which the peer will listen for HTTP REST requests.

It is assumed that the REST service receives requests from applications which have already authenticated the end user.

6.2 REST API

You can work with the REST API through any tool of your choice. For example, the curl command line utility or a browser based client such as the Firefox Rest Client or Chrome Postman. You can likewise trigger REST requests directly through Swagger. To obtain the REST API Swagger description, click here. The currently available APIs are summarized in the following section.

6.2.1 REST Endpoints

- *Block*
- GET /chain/blocks/{block-id}
- *Blockchain*
- GET /chain
- *Chaincode*
- POST /chaincode
- *Network*
- GET /network/peers
- *Registrar*
- POST /registrar

- GET /registrar/{enrollmentID}
- DELETE /registrar/{enrollmentID}
- GET /registrar/{enrollmentID}/ecert
- GET /registrar/{enrollmentID}/tcert
- *Transactions*
- GET /transactions/{UUID}

6.2.1.1 Block API

- **GET /chain/blocks/{block-id}**

Use the Block API to retrieve the contents of various blocks from the blockchain. The returned Block message structure is defined in section *3.2.1.1*.

Block Retrieval Request:

```
GET host:port/chain/blocks/173
```

Block Retrieval Response:

```
{
    "transactions": [
        {
            "type": 3,
            "chaincodeID": "EgRteWNj",
            "payload": "Ch4IARIGEgRteWNjGhIKBmludm9rZRIBYRIBYhICMTA=",
            "uuid": "f5978e82-6d8c-47d1-adec-f18b794f570e",
            "timestamp": {
                "seconds": 1453758316,
                "nanos": 206716775
            },
            "cert": "MIIB/
↪zCCAYWgAwIBAgIBATAKBggqhkjOPQQDAzApMQswCQYDVQQGEwJVUzEMMAoGA1UEChMDSUJNMQwwCgYDVQQDEwN0Y2EwHhcNMTY
↪BBkt8izf6Ew8UDd62EdWFikJhyCPY5VO9Wxq9JVzt3D6nubx2jO5JdfWt49q8V1Aythia50MZEDpmKhtM6z7LHOU1RxuxdjcYD
↪D3A+97qZpKN/MH0wDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/
↪wQCMAAwDQYDVR0OBAYEBAECAwQwDwYDVR0jBAgwBoAEAQIDBDA9BgYqAwQFBgcBAf8EMABNbPHZ0e/
↪2EToi0H8mkouuUDwurgBYuUB+vZfeMewBre3wXG0irzMtfwHlfECRDDAKBggqhkjOPQQDAwNoADBlAjAoote5zYFv91lHzpbEw
↪+r+CG7oMVFUFuoSlvBSCObK2bDIbNkW4VQ+ZC9GTsCMQC5GCgy2oZdHw/
↪x7XYzG2BiqmRkLRTiCS7vYCVJXLivU65P984HopxW0cEqeFM9co0=",
            "signature": "MGUCMCIJaCT3YRsjXt4TzwfmD9hg9pxYnV13kWgf7e1hAW5Nar//
↪05kFtpVlq83X+YtcmAIxAK0IQlCgS6nqQzZEGCLd9r7cg1AkQOT/
↪RgoWB8zcaVjh3bCmgYHsoPAPgMsi3TJktg=="
        }
    ],
    "stateHash": "7ftCvPeHIpsvSavxUoZM0u7o67MPU81ImOJIO7ZdMoH2mjnAaAAafYy9MIH3HjrWM1/
↪Zla/Q6LsLzIjuYdYdlQ==",
    "previousBlockHash":
↪"lT0InRg4Cvk4cKykWpCRKWDZ9YNYMzuHdUzsaeTeAcH3HdfriLEcTuxrFJ76W4jrWVvTBdI1etxuIV9AO6UF4Q==
↪",
    "nonHashData": {
        "localLedgerCommitTimestamp": {
            "seconds": 1453758316,
            "nanos": 250834782
        }
    }
}
```

6.2.1.2 Blockchain API

- **GET /chain**

Use the Chain API to retrieve the current state of the blockchain. The returned BlockchainInfo message is defined below.

```
message BlockchainInfo {
    uint64 height = 1;
    bytes currentBlockHash = 2;
    bytes previousBlockHash = 3;
}
```

- `height` - Number of blocks in the blockchain, including the genesis block.
- `currentBlockHash` - The hash of the current or last block.
- `previousBlockHash` - The hash of the previous block.

Blockchain Retrieval Request:

```
GET host:port/chain
```

Blockchain Retrieval Response:

```
{
    "height": 174,
    "currentBlockHash":
↪"lIfbDax2NZMU3rG3cDR11OGicPLp1yebIkia33Zte9AnfqvffK6tsHRyKwsw0hZFZkCGIa9wHVkOGyFTcFxM5w==
↪",
    "previousBlockHash": "Vlz6Dv5OSy0OZpJvijrU1cmY2cNS5Ar3xX5DxAi/
↪seaHHRPdssrljDeppDLzGx6ZVyayt8Ru6jO+E68IwMrXLQ=="
}
```

6.2.1.3 Chaincode API

- **POST /chaincode**

Use the Chaincode API to deploy, invoke, and query chaincodes. The deploy request requires the client to supply a `path` parameter, pointing to the directory containing the chaincode in the file system. The response to a deploy request is either a message containing a confirmation of successful chaincode deployment or an error, containing a reason for the failure. It also contains the generated chaincode `name` in the `message` field, which is to be used in subsequent invocation and query transactions to uniquely identify the deployed chaincode.

To deploy a chaincode, supply the required ChaincodeSpec payload, defined in section *3.1.2.2*.

Deploy Request:

```
POST host:port/chaincode

{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": "GOLANG",
    "chaincodeID":{
        "path":"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_
↪example02"
    },
    "ctorMsg": {
        "function":"init",
        "args":["a", "1000", "b", "2000"]
    }
  },
  "id": "1"
}
```

Deploy Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message":
↪"52b0d803fc395b5e34d8d4a7cd69fb6aa00099b8fabed83504ac1c5d61a425aca5b3ad3bf96643ea4fdaac132c417c37b
↪"
    },
    "id": 1
}
```

With security enabled, modify the required payload to include the `secureContext` element passing the enrollment ID of a logged in user as follows:

Deploy Request with security enabled:

```
POST host:port/chaincode

{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": "GOLANG",
    "chaincodeID":{
        "path":"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_
↪example02"
    },
    "ctorMsg": {
        "function":"init",
        "args":["a", "1000", "b", "2000"]
    },
    "secureContext": "lukas"
  },
  "id": "1"
}
```

The invoke request requires the client to supply a `name` parameter, which was previously returned in the response from the deploy transaction. The response to an invocation request is either a message containing a confirmation of successful execution or an error, containing a reason for the failure.

To invoke a function within a chaincode, supply the required ChaincodeSpec payload, defined in section *3.1.2.2*.

Invoke Request:

```
POST host:port/chaincode

{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
    "type": "GOLANG",
    "chaincodeID":{
      "name":
↪"52b0d803fc395b5e34d8d4a7cd69fb6aa00099b8fabed83504ac1c5d61a425aca5b3ad3bf96643ea4fdaac132c417c37b
↪"
    },
    "ctorMsg": {
        "function":"invoke",
        "args":["a", "b", "100"]
    }
  },
  "id": "3"
}
```

Invoke Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "5a4540e5-902b-422d-a6ab-e70ab36a2e6d"
    },
    "id": 3
}
```

With security enabled, modify the required payload to include the `secureContext` element passing the enrollment ID of a logged in user as follows:

Invoke Request with security enabled:

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
    "type": "GOLANG",
    "chaincodeID":{
      "name":
↪"52b0d803fc395b5e34d8d4a7cd69fb6aa00099b8fabed83504ac1c5d61a425aca5b3ad3bf96643ea4fdaac132c417c37b
↪"
    },
    "ctorMsg": {
        "function":"invoke",
        "args":["a", "b", "100"]
    },
    "secureContext": "lukas"
  },
  "id": "3"
}
```

The query request requires the client to supply a `name` parameter, which was previously returned in the response from the deploy transaction. The response to a query request depends on the chaincode implementation. The response will contain a message containing a confirmation of successful execution or an error, containing a reason for the failure. In the case of successful execution, the response will also contain values of requested state variables within the chaincode.

To invoke a query function within a chaincode, supply the required ChaincodeSpec payload, defined in section *3.1.2.2*.

Query Request:

```
POST host:port/chaincode/

{
  "jsonrpc": "2.0",
  "method": "query",
  "params": {
    "type": "GOLANG",
    "chaincodeID":{
      "name":
↪"52b0d803fc395b5e34d8d4a7cd69fb6aa00099b8fabed83504ac1c5d61a425aca5b3ad3bf96643ea4fdaac132c417c37b
↪"
    },
    "ctorMsg": {
        "function":"query",
        "args":["a"]
    }
  },
  "id": "5"
}
```

Query Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "-400"
    },
    "id": 5
}
```

With security enabled, modify the required payload to include the `secureContext` element passing the enrollment ID of a logged in user as follows:

Query Request with security enabled:

```
{
  "jsonrpc": "2.0",
  "method": "query",
  "params": {
    "type": "GOLANG",
    "chaincodeID":{
      "name":
↪"52b0d803fc395b5e34d8d4a7cd69fb6aa00099b8fabed83504ac1c5d61a425aca5b3ad3bf96643ea4fdaac132c417c37b
↪"
    },
    "ctorMsg": {
        "function":"query",
        "args":["a"]
    },
    "secureContext": "lukas"
  },
  "id": "5"
}
```

6.2.1.4 Network API

Use the Network API to retrieve information about the network of peer nodes comprising the blockchain fabric.

The /network/peers endpoint returns a list of all existing network connections for the target peer node. The list includes both validating and non-validating peers. The list of peers is returned as type `PeersMessage` , containing an array of `PeerEndpoint` , defined in section *3.1.1*.

```
message PeersMessage {
    repeated PeerEndpoint peers = 1;
}
```

Network Request:

```
GET host:port/network/peers
```

Network Response:

```
{
    "peers": [
        {
            "ID": {
                "name": "vp1"
            },
            "address": "172.17.0.4:7051",
            "type": 1,
            "pkiID": "rUA+vX2jVCXev6JsXDNgNBMX03IV9mHRPWo6h6SI0KLMypBJLd+JoGGlqFgi+eq/
↪"
        },
        {
            "ID": {
                "name": "vp3"
            },
            "address": "172.17.0.5:7051",
            "type": 1,
            "pkiID": "OBduaZJ72gmM+B9wp3aErQlofE0ulQfXfTHh377ruJjOpsUn0MyvsJELUTHpAbHI
↪"
        },
        {
            "ID": {
                "name": "vp2"
            },
            "address": "172.17.0.6:7051",
            "type": 1,
            "pkiID": "GhtP0Y+o/XVmRNXGF6pcm9KLNTfCZp+XahTBqVRmaIumJZnBpom4ACayVbg4Q/Eb
↪"
        }
    ]
}
```

6.2.1.5 Registrar API (member services)

- **POST /registrar**

- **GET /registrar/{enrollmentID}**
- **DELETE /registrar/{enrollmentID}**
- **GET /registrar/{enrollmentID}/ecert**
- **GET /registrar/{enrollmentID}/tcert**

Use the Registrar APIs to manage end user registration with the certificate authority (CA). These API endpoints are used to register a user with the CA, determine whether a given user is registered, and to remove any login tokens for a target user from local storage, preventing them from executing any further transactions. The Registrar APIs are also used to retrieve user enrollment and transaction certificates from the system.

The `/registrar` endpoint is used to register a user with the CA. The required Secret payload is defined below. The response to the registration request is either a confirmation of successful registration or an error, containing a reason for the failure.

```
message Secret {
    string enrollId = 1;
    string enrollSecret = 2;
}
```

- `enrollId` - Enrollment ID with the certificate authority.
- `enrollSecret` - Enrollment password with the certificate authority.

Enrollment Request:

```
POST host:port/registrar

{
  "enrollId": "lukas",
  "enrollSecret": "NPKYL39uKbkj"
}
```

Enrollment Response:

```
{
    "OK": "Login successful for user 'lukas'."
}
```

The `GET /registrar/{enrollmentID}` endpoint is used to confirm whether a given user is registered with the CA. If so, a confirmation will be returned. Otherwise, an authorization error will result.

Verify Enrollment Request:

```
GET host:port/registrar/jim
```

Verify Enrollment Response:

```
{
    "OK": "User jim is already logged in."
}
```

Verify Enrollment Request:

```
GET host:port/registrar/alex
```

Verify Enrollment Response:

```
{
    "Error": "User alex must log in."
}
```

The `DELETE /registrar/{enrollmentID}` endpoint is used to delete login tokens for a target user. If the login tokens are deleted successfully, a confirmation will be returned. Otherwise, an authorization error will result. No payload is required for this endpoint.

Remove Enrollment Request:

```
DELETE host:port/registrar/lukas
```

Remove Enrollment Response:

```
{
    "OK": "Deleted login token and directory for user lukas."
}
```

The `GET /registrar/{enrollmentID}/ecert` endpoint is used to retrieve the enrollment certificate of a given user from local storage. If the target user has already registered with the CA, the response will include a URL-encoded version of the enrollment certificate. If the target user has not yet registered, an error will be returned. If the client wishes to use the returned enrollment certificate after retrieval, keep in mind that it must be URL-decoded.

Enrollment Certificate Retrieval Request:

```
GET host:port/registrar/jim/ecert
```

Enrollment Certificate Retrieval Response:

```
{
    "OK": "-----BEGIN+CERTIFICATE-----
↪%0AMIIBzTCCAVSgAwIBAgIBATAKBggqhkjOPQQDAzApMQswCQYDVQQGEwJVUzEMMAoG
↪%0AA1UEChMDSUJNMQwwCgYDVQQDEwNPQkMwHhcNMTYwMTIxMDYzNjEwWhcNMTYwNDIw
↪%0AMDYzNjEwWjApMQswCQYDVQQGEwJVUzEMMAoGA1UEChMDSUJNMQwwCgYDVQQDEwNP
↪%0AQkMwdjAQBgcqhkjOPQIBBgUrgQQAIgNiAARSLgjGD0omuJKYrJF5ClyYb3sGEGTU
↪%0AH1mombSAOJ6GAOKEULt4L919sbSSChs0AEvTX7UDf4KNaKTrKrqo4khCoboMg1VS%0AXVTTPrJ
↪%2BOxSJTXFZCohVgbhWh6ZZX2tfb7%2BjUDBOMA4GA1UdDwEB%2FwQEAwIHgDAM
↪%0ABgNVHRMBAf8EAjAAMA0GA1UdDgQGBAQBAgMEMA8GA1UdIwQIMAaABAECAwQwDgYG%0AUQMEBQYHAQH
↪%2FBAE0MAoGCCqGSM49BAMDA2cAMGQCMGz2RR0NsJOhxbo0CeVts2C5%0A
↪%2BsAkKQ7v1Llbg78A1pyC5uBmoBvSnv5Dd0w2yOmj7QIwY%2Bn5pkLiwisxWurkHfiD
↪%0AxizmN6vWQ8uhTd3PTdJiEEckjHKiq9pwD%2FGMt%2BWjP7zF%0A-----END+CERTIFICATE-----%0A"
}
```

The `/registrar/{enrollmentID}/tcert` endpoint retrieves the transaction certificates for a given user that has registered with the certificate authority. If the user has registered, a confirmation message will be returned containing an array of URL-encoded transaction certificates. Otherwise, an error will result. The desired number of transaction certificates is specified with the optional 'count' query parameter. The default number of returned transaction certificates is 1; and 500 is the maximum number of certificates that can be retrieved with a single request. If the client wishes to use the returned transaction certificates after retrieval, keep in mind that they must be URL-decoded.

Transaction Certificate Retrieval Request:

```
GET host:port/registrar/jim/tcert
```

Transaction Certificate Retrieval Response:

```
{
    "OK": [
        "-----BEGIN+CERTIFICATE-----
↪%0AMIIBwDCCAWagAwIBAgIBATAKBggqhkjOPQQDAzApMQswCQYDVQQGEwJVUzEMMAoG
↪%0AA1UEChMDSUJNMQwwCgYDVQQDEwN0Y2EwHhcNMTYwMzExMjEwMTI2WhcNMTYwNjA5
↪%0AMjEwMTI2WjApMQswCQYDVQQGEwJVUzEMMAoGA1UEChMDSUJNMQwwCgYDVQQDEwNq
↪%0AaW0wWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAAQfwJORRED9RAsmSl%2FEowq1STBb%0A
↪%2FoFteymZ96RUr%2BsKmF9PNrrUNvFZFhvukxZZjqhEcGiQqFyRf%2FBnVN%2BbtRzMo38w
↪%0AfTAOBgNVHQ8BAf8EBAMCB4AwDAYDVR0TAQH%2FBAIwADANBgNVHQ4EBgQEAQIDBDAP
↪%0ABgNVHSMECDAGgAQBAgMEMD0GBioDBAUGBwEB%2FwQwSRWQFmErr0SmQO9AFP4GJYzQ%0APQMmcsCjKiJf
↪%2Bw1df%2FLnXunCsCUlf%2FalIUaeSrT7MAoGCCqGSM49BAMDA0gAMEUC%0AIQC
↪%2FnE71FBJd0hwNTLXWmlCJff4Yi0J%2BnDi%2BYnujp%2Fn9nQIgYWg0m0QFzddyJ0%2FF
↪%0AKzIZEJlKgZTt8ZTlGg3BBrgl7qY%3D%0A-----END+CERTIFICATE-----%0A"
    ]
}
```

Transaction Certificate Retrieval Request:

```
GET host:port/registrar/jim/tcert?count=5
```

Transaction Certificate Retrieval Response:

```
{
    "OK": [
        "-----BEGIN+CERTIFICATE-----
↪%0AMIIBwDCCAWagAwIBAgIBATAKBggqhkjOPQQDAzApMQswCQYDVQQGEwJVUzEMMAoG
↪%0AA1UEChMDSUJNMQwwCgYDVQQDEwN0Y2EwHhcNMTYwMzExMjEwMTI2WhcNMTYwNjA5
↪%0AMjEwMTI2WjApMQswCQYDVQQGEwJVUzEMMAoGA1UEChMDSUJNMQwwCgYDVQQDEwNq
↪%0AaW0wWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAARwJxVezgDcTAgj2LtTKVm65qft%0AhRTYnIOQhhOx
↪%2B%2B2NRu5r3Kn%2FXTf1php3NXOFY8ZQbY%2FQbFAwn%2FB0O68wlHiro38w
↪%0AfTAOBgNVHQ8BAf8EBAMCB4AwDAYDVR0TAQH%2FBAIwADANBgNVHQ4EBgQEAQIDBDAP
↪%0ABgNVHSMECDAGgAQBAgMEMD0GBioDBAUGBwEB%2FwQwRVPMSKVcHsk4aGHxBWc8PGKj
↪%0AqtTVTtuXnN45BynIx6lP6urpqkSuILgB1YOdRNefMAoGCCqGSM49BAMDA0gAMEUC%0AIAIjESYDp
↪%2FXePKANGpsY3Tu%2F4A2IfeczbC3uB%2BpziltWAiEA6Stp%2FX4DmbJGgZe8
↪%0APMNBgRKeoU6UbgTmed0ZEALLZP8%3D%0A-----END+CERTIFICATE-----%0A",
        "-----BEGIN+CERTIFICATE-----
↪%0AMIIBwDCCAWagAwIBAgIBATAKBggqhkjOPQQDAzApMQswCQYDVQQGEwJVUzEMMAoG
↪%0AA1UEChMDSUJNMQwwCgYDVQQDEwN0Y2EwHhcNMTYwMzExMjEwMTI2WhcNMTYwNjA5
↪%0AMjEwMTI2WjApMQswCQYDVQQGEwJVUzEMMAoGA1UEChMDSUJNMQwwCgYDVQQDEwNq
↪%0AaW0wWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAARwJxVezgDcTAgj2LtTKVm65qft%0AhRTYnIOQhhOx
↪%2B%2B2NRu5r3Kn%2FXTf1php3NXOFY8ZQbY%2FQbFAwn%2FB0O68wlHiro38w
↪%0AfTAOBgNVHQ8BAf8EBAMCB4AwDAYDVR0TAQH%2FBAIwADANBgNVHQ4EBgQEAQIDBDAP
↪%0ABgNVHSMECDAGgAQBAgMEMD0GBioDBAUGBwEB%2FwQwRVPMSKVcHsk4aGHxBWc8PGKj
↪%0AqtTVTtuXnN45BynIx6lP6urpqkSuILgB1YOdRNefMAoGCCqGSM49BAMDA0gAMEUC%0AIAIjESYDp
↪%2FXePKANGpsY3Tu%2F4A2IfeczbC3uB%2BpziltWAiEA6Stp%2FX4DmbJGgZe8
↪%0APMNBgRKeoU6UbgTmed0ZEALLZP8%3D%0A-----END+CERTIFICATE-----%0A",
        "-----BEGIN+CERTIFICATE-----
↪%0AMIIBwDCCAWagAwIBAgIBATAKBggqhkjOPQQDAzApMQswCQYDVQQGEwJVUzEMMAoG
↪%0AA1UEChMDSUJNMQwwCgYDVQQDEwN0Y2EwHhcNMTYwMzExMjEwMTI2WhcNMTYwNjA5
↪%0AMjEwMTI2WjApMQswCQYDVQQGEwJVUzEMMAoGA1UEChMDSUJNMQwwCgYDVQQDEwNq
↪%0AaW0wWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAARwJxVezgDcTAgj2LtTKVm65qft%0AhRTYnIOQhhOx
↪%2B%2B2NRu5r3Kn%2FXTf1php3NXOFY8ZQbY%2FQbFAwn%2FB0O68wlHiro38w
↪%0AfTAOBgNVHQ8BAf8EBAMCB4AwDAYDVR0TAQH%2FBAIwADANBgNVHQ4EBgQEAQIDBDAP
↪%0ABgNVHSMECDAGgAQBAgMEMD0GBioDBAUGBwEB%2FwQwRVPMSKVcHsk4aGHxBWc8PGKj
↪%0AqtTVTtuXnN45BynIx6lP6urpqkSuILgB1YOdRNefMAoGCCqGSM49BAMDA0gAMEUC%0AIAIjESYDp
↪%2FXePKANGpsY3Tu%2F4A2IfeczbC3uB%2BpziltWAiEA6Stp%2FX4DmbJGgZe8
↪%0APMNBgRKeoU6UbgTmed0ZEALLZP8%3D%0A-----END+CERTIFICATE-----%0A",
        "-----BEGIN+CERTIFICATE-----
↪%0AMIIBwDCCAWagAwIBAgIBATAKBggqhkjOPQQDAzApMQswCQYDVQQGEwJVUzEMMAoG
↪%0AA1UEChMDSUJNMQwwCgYDVQQDEwN0Y2EwHhcNMTYwMzExMjEwMTI2WhcNMTYwNjA5
↪%0AMjEwMTI2WjApMQswCQYDVQQGEwJVUzEMMAoGA1UEChMDSUJNMQwwCgYDVQQDEwNq
↪%0AaW0wWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAARwJxVezgDcTAgj2LtTKVm65qft%0AhRTYnIOQhhOx
↪%2B%2B2NRu5r3Kn%2FXTf1php3NXOFY8ZQbY%2FQbFAwn%2FB0O68wlHiro38w
↪%0AfTAOBgNVHQ8BAf8EBAMCB4AwDAYDVR0TAQH%2FBAIwADANBgNVHQ4EBgQEAQIDBDAP
↪%0ABgNVHSMECDAGgAQBAgMEMD0GBioDBAUGBwEB%2FwQwRVPMSKVcHsk4aGHxBWc8PGKj
↪%0AqtTVTtuXnN45BynIx6lP6urpqkSuILgB1YOdRNefMAoGCCqGSM49BAMDA0gAMEUC%0AIAIjESYDp
↪%2FXePKANGpsY3Tu%2F4A2IfeczbC3uB%2BpziltWAiEA6Stp%2FX4DmbJGgZe8
```

```
        "-----BEGIN+CERTIFICATE-----
↪%0AMIIBwDCCAWagAwIBAgIBATAKBggqhkjOPQQDAzApMQswCQYDVQQGEwJVUzEMMAoG
↪%0AA1UEChMDSUJNMQwwCgYDVQQDEwN0Y2EwHhcNMTYwMzExMjEwMTI2WhcNMTYwNjA5
↪%0AMjEwMTI2WjApMQswCQYDVQQGEwJVUzEMMAoGA1UEChMDSUJNMQwwCgYDVQQDEwNq
↪%0AaW0wWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAARwJxVezgDcTAgj2LtTKVm65qft%0AhRTYnIOQhhOx
↪%2B%2B2NRu5r3Kn%2FXTf1php3NXOFY8ZQbY%2FQbFAwn%2FB0O68wlHiro38w
↪%0AfTAOBgNVHQ8BAf8EBAMCB4AwDAYDVR0TAQH%2FBAIwADANBgNVHQ4EBgQEAQIDBDAP
↪%0ABgNVHSMECDAGgAQBAgMEMD0GBioDBAUGBwEB%2FwQwRVPMSKVcHsk4aGHxBWc8PGKj
↪%0AqtTVTtuXnN45BynIx6lP6urpqkSuILgB1YOdRNefMAoGCCqGSM49BAMDA0gAMEUC%0AIAIjESYDp
↪%2FXePKANGpsY3Tu%2F4A2IfeczbC3uB%2BpziltWAiEA6Stp%2FX4DmbJGgZe8
↪%0APMNBgRKeoU6UbgTmed0ZEALLZP8%3D%0A-----END+CERTIFICATE-----%0A"
    ]
}
```

6.2.1.6 Transactions API

- **GET /transactions/{UUID}**

Use the Transaction API to retrieve an individual transaction matching the UUID from the blockchain. The returned transaction message is defined in section *3.1.2.1*.

Transaction Retrieval Request:

```
GET host:port/transactions/f5978e82-6d8c-47d1-adec-f18b794f570e
```

Transaction Retrieval Response:

```
{
    "type": 3,
    "chaincodeID": "EgRteWNj",
    "payload": "Ch4IARIGEgRteWNjGhIKBmludm9rZRIBYRIBYhICMTA=",
    "uuid": "f5978e82-6d8c-47d1-adec-f18b794f570e",
    "timestamp": {
        "seconds": 1453758316,
        "nanos": 206716775
    },
    "cert": "MIIB/
↪zCCAYWgAwIBAgIBATAKBggqhkjOPQQDAzApMQswCQYDVQQGEwJVUzEMMAoGA1UEChMDSUJNMQwwCgYDVQQDEwN0Y2EwHhcNMTY
↪BBkt8izf6Ew8UDd62EdWFikJhyCPY5VO9Wxq9JVzt3D6nubx2jO5JdfWt49q8V1Aythia50MZEDpmKhtM6z7LHOU1RxuxdjcYD
↪D3A+97qZpKN/MH0wDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/
↪wQCMAAwDQYDVR0OBAYEBAECAwQwDwYDVR0jBAgwBoAEAQIDBDA9BgYqAwQFBgcBAf8EMABNbPHZ0e/
↪2ETOi0H8mkouuUDwurgBYuUB+vZfeMewBre3wXG0irzMtfwHlfECRDDAKBggqhkjOPQQDAwNoADBlAjAoote5zYFv91lHzpbEw
↪+r+CG7oMVFUFuoSlvBSCObK2bDIbNkW4VQ+ZC9GTsCMQC5GCgy2oZdHw/
↪x7XYzG2BiqmRkLRTiCS7vYCVJXLivU65P984HopxW0cEqeFM9co0=",
    "signature": "MGUCMCIJaCT3YRsjXt4TzwfmD9hg9pxYnV13kWgf7e1hAW5Nar//
↪05kFtpVlq83X+YtcmAIxAK0IQlCgS6nqQzZEGCLd9r7cg1AkQOT/
↪RgoWB8zcaVjh3bCmgYHsoPAPgMsi3TJktg=="
}
```

6.3 CLI

The CLI includes a subset of the available APIs to enable developers to quickly test and debug chaincodes or query for status of transactions. CLI is implemented in Golang and operable on multiple OS platforms. The currently available CLI commands are summarized in the following section.

6.3.1 CLI Commands

To see what CLI commands are currently available in the implementation, execute the following:

```
$ peer
```

You will receive a response similar to below:

```
Usage:
  peer [command]

Available Commands:
  node        node specific commands.
  network     network specific commands.
  chaincode   chaincode specific commands.
  help        Help about any command

Flags:
  -h, --help[=false]: help for peer
      --logging-level="": Default logging level and overrides, see core.yaml for full␣
↪syntax

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

Some of the available command line arguments for the `peer` command are listed below:

- `-c` - constructor: function to trigger in order to initialize the chaincode state upon deployment.
- `-l` - language: specifies the implementation language of the chaincode. Currently, only Golang is supported.
- `-n` - name: chaincode identifier returned from the deployment transaction. Must be used in subsequent invoke and query transactions.
- `-p` - path: identifies chaincode location in the local file system. Must be used as a parameter in the deployment transaction.
- `-u` - username: enrollment ID of a logged in user invoking the transaction.

Not all of the above commands are fully implemented in the current release. The fully supported commands that are helpful for chaincode development and debugging are described below.

Note, that any configuration settings for the peer node listed in the `core.yaml` configuration file, which is the configuration file for the `peer` process, may be modified on the command line with an environment variable. For example, to set the `peer.id` or the `peer.addressAutoDetect` settings, one may pass the `CORE_PEER_ID=vp1` and `CORE_PEER_ADDRESSAUTODETECT=true` on the command line.

6.3.1.1 node start

The CLI `node start` command will execute the peer process in either the development or production mode. The development mode is meant for running a single peer node locally, together with a local chaincode deployment. This allows a chaincode developer to modify and debug their code without standing up a complete network. An example for starting the peer in development mode follows:

```
peer node start --peer-chaincodedev
```

To start the peer process in production mode, modify the above command as follows:

```
peer node start
```

6.3.1.2 network login

The CLI `network login` command will login a user, that is already registered with the CA, through the CLI. To login through the CLI, issue the following command, where `username` is the enrollment ID of a registered user.

```
peer network login <username>
```

The example below demonstrates the login process for user `jim`.

```
peer network login jim
```

The command will prompt for a password, which must match the enrollment password for this user registered with the certificate authority. If the password entered does not match the registered password, an error will result.

```
22:21:31.246 [main] login -> INFO 001 CLI client login...
22:21:31.247 [main] login -> INFO 002 Local data store for client loginToken: /var/
↪hyperledger/production/client/
Enter password for user 'jim': ************
22:21:40.183 [main] login -> INFO 003 Logging in user 'jim' on CLI interface...
22:21:40.623 [main] login -> INFO 004 Storing login token for user 'jim'.
22:21:40.624 [main] login -> INFO 005 Login successful for user 'jim'.
```

You can also pass a password for the user with `-p` parameter. An example is below.

```
peer network login jim -p 123456
```

6.3.1.3 chaincode deploy

The CLI `deploy` command creates the docker image for the chaincode and subsequently deploys the package to the validating peer. An example is below.

```
peer chaincode deploy -p github.com/hyperledger/fabric/examples/chaincode/go/
↪chaincode_example02 -c '{"Function":"init", "Args": ["a","100", "b", "200"]}'
```

With security enabled, the command must be modified to pass an enrollment id of a logged in user with the `-u` parameter. An example is below.

```
peer chaincode deploy -u jim -p github.com/hyperledger/fabric/examples/chaincode/go/
↪chaincode_example02 -c '{"Function":"init", "Args": ["a","100", "b", "200"]}'
```

Note: If your GOPATH environment variable contains more than one element, the chaincode must be found in the first one or deployment will fail.

6.3.1.4 chaincode invoke

The CLI `invoke` command executes a specified function within the target chaincode. An example is below.

```
peer chaincode invoke -n <name_value_returned_from_deploy_command> -c '{"Function":
↪"invoke", "Args": ["a", "b", "10"]}'
```

With security enabled, the command must be modified to pass an enrollment id of a logged in user with the `-u` parameter. An example is below.

```
peer chaincode invoke -u jim -n <name_value_returned_from_deploy_command> -c '{
↪"Function": "invoke", "Args": ["a", "b", "10"]}'
```

6.3.1.5 chaincode query

The CLI `query` command triggers a specified query method within the target chaincode. The response that is returned depends on the chaincode implementation. An example is below.

```
peer chaincode query -l golang -n <name_value_returned_from_deploy_command> -c '{
↪"Function": "query", "Args": ["a"]}'
```

With security enabled, the command must be modified to pass an enrollment id of a logged in user with the `-u` parameter. An example is below.

```
peer chaincode query -u jim -l golang -n <name_value_returned_from_deploy_command> -c
↪'{"Function": "query", "Args": ["a"]}'
```

7. Application Model

7.1 Composition of an Application

An application follows a MVC-B architecture – Model, View, Control, BlockChain.

VIEW LOGIC – Mobile or Web UI interacting with control logic.

CONTROL LOGIC – Coordinates between UI, Data Model and APIs to drive transitions and chain-code.

DATA MODEL – Application Data Model – manages off-chain data, including Documents and large files.

BLOCKCHAIN LOGIC – Blockchain logic are extensions of the Controller Logic and Data Model, into the Blockchain realm. Controller logic is enhanced by chaincode, and the data model is enhanced with transactions on the blockchain.

For example, a Bluemix PaaS application using Node.js might have a Web front-end user interface or a native mobile app with backend model on Cloudant data service. The control logic may interact with 1 or more chaincodes to process transactions on the blockchain.

7.2 Sample Application

8. Future Directions

8.1 Enterprise Integration

8.2 Performance and Scalability

8.3 Additional Consensus Plugins

8.4 Additional Languages

9.1 Authors

The following authors have written sections of this document: Binh Q Nguyen, Elli Androulaki, Angelo De Caro, Sheehan Anderson, Manish Sethi, Thorsten Kramp, Alessandro Sorniotti, Marko Vukolic, Florian Simon Schubert, Jason K Yellick, Konstantinos Christidis, Srinivasan Muralidharan, Anna D Derbakova, Dulce Ponceleon, David Kravitz, Diego Masini.

9.2 Reviewers

The following reviewers have contributed to this document: Frank Lu, John Wolpert, Bishop Brock, Nitin Gaur, Sharon Weed, Konrad Pabjan.

9.3 Acknowledgements

The following contributors have provided invaluable technical input to this specification: Gennaro Cuomo, Joseph A Latone, Christian Cachin

10. References

- [1] Miguel Castro, Barbara Liskov: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20(4): 398-461 (2002)
- [2] Christian Cachin, Rachid Guerraoui, Luís E. T. Rodrigues: Introduction to Reliable and Secure Distributed Programming (2. ed.). Springer 2011, ISBN 978-3-642-15259-7, pp. I-XIX, 1-367
- [3] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg: The Weakest Failure Detector for Solving Consensus. J. ACM 43(4): 685-722 (1996)
- [4] Cynthia Dwork, Nancy A. Lynch, Larry J. Stockmeyer: Consensus in the presence of partial synchrony. J. ACM 35(2): 288-323 (1988)
- [5] Manos Kapritsos, Yang Wang, Vivien Quéma, Allen Clement, Lorenzo Alvisi, Mike Dahlin: All about Eve: Execute-Verify Replication for Multi-Core Servers. OSDI 2012: 237-250
- [6] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, Marko Vukolic: The Next 700 BFT Protocols. ACM Trans. Comput. Syst. 32(4): 12:1-12:45 (2015)
- [7] Christian Cachin, Simon Schubert, Marko Vukolić: Non-determinism in Byzantine Fault-Tolerant Replication

CHAPTER 12

Canonical Use Cases

B2B Contract

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

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

Persona

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

Key Components

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

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

Manufacturing Supply Chain

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

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

Persona

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

Key Components

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

Asset Depository

Assets such as financial securities must be able to be dematerialized on a blockchain network so that all stakeholders of an asset type will have direct access to that asset, allowing them to initiate trades and acquire information on an asset without going through layers of intermediaries. Trades should be settled in near real time and all stakeholders must be able to access asset information in near real time. A stakeholder should be able to add business rules on any given asset type, as one example of using automation logic to further reduce operating costs.

Persona

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

Key Components

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

CHAPTER 13

Extended Use Cases

The following extended use cases examine additional requirements and scenarios.

One Trade, One Contract

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

Direct Communication

Company A announces its intention to raise 2 Billion USD by way of rights issue. Because this is a voluntary action, Company A needs to ensure that complete details of the offer are sent to shareholders in real time, regardless of how many intermediaries are involved in the process (such as receiving/paying agents, CSD, ICSD, local/global custodian banks, asset management firms, etc). Once a shareholder has made a decision, that decision will also be processed and settled (including the new issuance of shares) in real time. If a shareholder sold its rights to a third party, the securities depository must be able to record the new shares under the name of their new rightful owner.

Separation of Asset Ownership and Custodian’s Duties

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

Interoperability of Assets

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

This document serves as an overview of the new features present in fabric v0.6.1-preview release, and outlines the changes you will need to make to successfully migrate code originally written for the v0.5-developer-preview release that you now wish to run on fabric v0.6.1-preview.

CHAPTER 14

Migrating chaincode to fabric v0.6.1-preview

1. The chaincode shim interface changes for compatibility with the latest Hyperledger shim:

The chaincode interface has changed from `shim.ChaincodeStub` to `shim.ChaincodeStubInterface`. See the interfaces.go file for the shim source code. The following code snippet from line 74 of chaincode_example02 will highlight this alteration.

This change applies to all transaction types: Deploy, Invoke, and Query.

```
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStub, function string, args␣
↪[]string) ([]byte, error) {

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface, function string,␣
↪args []string) ([]byte, error) {
```

2. Chaincode calling chaincode is included in the shim package.

There are two functions available - **InvokeChaincode** and **QueryChaincode**. First, these functions no longer accept the function name as a parameter; instead the function must be passed in as an argument. Second, the arguments are passed in as a byte array, not a string. A utility is provided to convert your strings to a byte array.

The following code snippets from chaincode_example04 demonstrate the difference. Make note of `f`, representing the function invoke. It is removed from the **InvokeChaincode** parameters, and instead passed as an argument to the **invokeArgs** element. The arguments are then converted to a byte array before being passed to **InvokeChaincode**. This change is not optional and must be implemented in your code.

```
// fabric v0.5-developer-preview
f := "invoke"
invokeArgs := []string{"a", "b", "10"}
response, err := stub.InvokeChaincode(chainCodeToCall, f, invokeArgs)
```

```
// fabric v0.6.1-preview code
f := "invoke"
// function is removed from InvokeChaincode, now passed as an argument within
// invokeArgs.  invokeArgs is converted to byte array and then passed along.
invokeArgs := util.ToChaincodeArgs(f, "a", "b", "10")
response, err := stub.InvokeChaincode(chainCodeToCall, invokeArgs)
```

3. Chaincode APIs have changed when constructing REST API payloads and CLI commands.

The function can now be passed within the “args” element as the first argument. The following code snippets will demonstrate the changes to a basic chaincode invoking transaction from the CLI and through the REST API.

```
peer chaincode invoke -l golang -n mycc -c '{"Function": "invoke", "Args": ["a", "b",
↪"10"]}'

peer chaincode invoke -l golang -n mycc -c '{"Args": ["invoke", "a", "b", "10"]}'
```

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"mycc"
      },
      "ctorMsg": {
         "function":"invoke",
         "args":["a", "b", "10"]
      }
  },
  "id": 3
}
```

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"mycc"
      },
      "ctorMsg": {
         "args":["invoke", "a", "b", "10"]
      }
  },
  "id": 3
}
```

Note: REST API and CLI developed in fabric v0.5-developer-preview will work with fabric v0.6.1-preview. However, when using the Java SDK you must implement the new format, where the function is passed within the "args" element.

CHAPTER 15

New Features

1. Custom Events & Event handler:

The fabric now has the ability to create custom events and emit this information to a client-side node application leveraging the hfc SDK. This is done by implementing the EventHub Service in your node program. The EventHub Service listens for events.

You can customize the eventsender.go code to determine which events you want sent. In the example, only the invoke transaction type is coded to send outgoing events. See the following code snippet in eventsender.go which demonstrates invocations being broadcast by the event sender:

```
func (t *EventSender) Invoke(stub shim.ChaincodeStubInterface, function string, args␣
↪[]string) ([]byte, error) {
    b, err := stub.GetState("noevents")
    if err != nil {
        return nil, errors.New("Failed to get state")
    }
  // define the construct for the event
    noevts, _ := strconv.Atoi(string(b))

    tosend := "Event " + string(b)
    for _, s := range args {
        tosend = tosend + "," + s
    }
  // create the event based on the construct
    err = stub.PutState("noevents", []byte(strconv.Itoa(noevts+1)))
    if err != nil {
        return nil, err
    }
  // pass the event along for Event Listener service
    err = stub.SetEvent("evtsender", []byte(tosend))
    if err != nil {
        return nil, err
    }
    return nil, nil
}
```

Enable the event service in your node program with the following steps:

```
// set the port where the event service will listen
chain.eventHubConnect("localhost:7053");

// Get the eventHub service associated with the chain
var eventHub = chain.getEventHub();
```

```
// Register on a specific chaincode for a specific event - syntax example only
eventHub.registerChaincodeEvent(<chaincode ID>, <event name>, <callback>);
// actual code example
var registrationId = eh.registerChaincodeEvent(
↪"b16cec7aa4466f57dd18f3c159b85d2962e741824c702136fdfcf616addcec01", "evtsender",␣
↪function(event) {
        console.log(util.format("Custom event received, payload: %j\n", event.payload.
↪toString()));
});

//Unregister events or a specific chaincode
eventHub.unregisterChaincodeEvent(registrationID);

// disconnect when done listening for events
process.on('exit', function() {
    chain.eventHubDisconnect();
});
```

Explore the full library of the sample event application for the application source code and deeper documentation.

2. Java chaincode shim - new shim library to support java chaincode interacting with Hyperledger fabric. See the java shim library for the source code.

3. Ability to call chaincode using a 64encoded string. A custom UnmarshalJSON method for ChaincodeInput allows for string-based REST/JSON input, which is then converted to []byte-based. This allows browsers to pass in string or binary arguments to the application driving the chaincode.

4. Docker client upgrade to Docker 1.12

5. `fabric-peer` and `fabric-membersrvc` images for multiple platforms are available on Hyperledger Dockerhub. The images are part of the continuous integration process and built with every new code change.

6. New warnings for chaincode development. The following practices can lead to malfunctioning and/or non-deterministic chaincode and should be avoided:

- Iterating using GetRows
- Using associative arrays with iteration (the order is randomized in Go)
- Reading list of items from KVS table (the order is not guaranteed). Use ordering
- Writing thread-unsafe chaincode where invoke and query may be called in parallel
- Substituting global memory or cache storage for ledger state variables in the chaincode
- Accessing external services (e.g. databases) directly from the chaincode
- Using libraries or globabl variables that could introduce non-determinism (e.g. “random” or “time”)
- For templates of deterministic and properly-written chaincode, see the examples library. This directory contains samples written in Go and Java.

7. Fabric Starter Kit - This section describes how to set up a self-contained environment for application development with the Hyperledger fabric. The setup uses **Docker** to provide a controlled environment with all the necessary Hyperledger fabric components to support a Node.js application built with the fabric’s Node.js SDK, and chaincode written in Go.

There are three Docker images that, when run, will provide a basic network environment. There is an image to run a single `peer`, one to run the `membersrvc`, and one to run both your Node.js application and your chaincode.

8. Fabric boilerplate - The public IBM-Blockchain repo now contains a boilerplate application to help application developers quickly create a network and deploy and app. The network can be spun up locally using Docker containers or through a Bluemix instance of the blockchain service.

9. Fabric v0.6 provides the ability to dynamically register and enroll users with attributes through the hfc SDK. See asset-mgmt-with-dynamic-roles.js as an example. The hfc SDK previously allowed you to dynamically enroll users, but these users were already registered and aligned with attributes/affiliations hardcoded in the membersrvc.yml. Now a user with `registrar` authority can register and enroll unique users to the network. See the following code snippets as an example of dynamic registration:

```
// call the registerAndEnroll function to add a unique user to the network
// (i.e. a user not present in the membersrvc.yml)
// below we see "assigner2" being registered and enrolled with two unique
// attributes - a 'role' with the value of 'client' and an 'account' with the
// value of 'aliceAccount'
console.log("enrolling alice2 ...");
registerAndEnroll("alice2",[{name:'role',value:'client'},{name:'account',value:
↪aliceAccount}], function(err,user) {
  if (err) return cb(err);
  alice = user;
```

```
// define the funtion
function registerAndEnroll(name, attrs, cb) {
    console.log("registerAndEnroll name=%s attrs=%j",name,attrs);
    var registrationRequest = {
         roles: [ 'client' ],
         enrollmentID: name,
         affiliation: "bank_a",
         attributes: attrs
    };
    chain.registerAndEnroll(registrationRequest,cb);
}
```

CHAPTER 16

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 17

Fabric Starter Kit

This section describes how to set up a self-contained environment for application development with the Hyperledger fabric. The setup uses **Docker** to provide a controlled environment with all the necessary Hyperledger fabric components to support a Node.js application built with the fabric's Node.js SDK, and chaincode written in Go.

There are three Docker images that, when run, will provide a basic network environment. There is an image to run a single `peer` , one to run the `membersrvc` , and one to run both your Node.js application and your chaincode. See *Application Developer's Overview* on how the components running within the containers will communicate.

The starter kit comes with a sample Node.js application ready to execute and sample chaincode. The starter kit will be running in chaincode developer mode. In this mode, the chaincode is built and started prior to the application making a call to deploy it.

Note: The deployment of chaincode in network mode requires that the Hyperledger fabric Node.js SDK has access to the chaincode source code and all of its dependencies, in order to properly build a deploy request. It also requires that the `peer` have access to the Docker daemon to be able to build and deploy the new Docker image that will run the chaincode. *This is a more complicated configuration and not suitable to an introduction to the Hyperledger fabric.* We recommend first running in chaincode development mode.

Further exploration

If you wish, there are a number of chaincode examples near by.

```
cd ../../chaincode
```

Getting started

Note: This sample was prepared using Docker for Mac 1.12.0. First, install the Docker Engine.

Next, copy our docker-compose.yml file to a local directory:

```
curl -o docker-compose.yml https://raw.githubusercontent.com/hyperledger/fabric/v0.6/
↪examples/sdk/node/docker-compose.yml
```

The docker-compose environment uses three Docker images. Two are published to DockerHub. However, with the third, we provide you the source to build your own, so that you can customize it to inject your application code for development. The following Dockerfile is used to build the base **fabric-starter-kit** image and may be used as a starting point for your own customizations.

```
curl -o Dockerfile https://raw.githubusercontent.com/hyperledger/fabric/v0.6/examples/
↪sdk/node/Dockerfile
docker build -t hyperledger/fabric-starter-kit:latest .
```

- Start the fabric network environment using docker-compose. From a terminal session that has the working directory of where the above *docker-compose.yml* is located, execute one of following `docker-compose` commands.
- to run as detached containers:

  `docker-compose up -d`

  note: to see the logs for the `peer` container use the `docker logs peer` command
- to run in the foreground and see the log output in the current terminal session:

  `docker-compose up`

Both commands will start three Docker containers. To view the container status use the `docker ps` command. The first time this is run, the Docker images will be downloaded. This may take 10 minutes or more depending on the network connections of the system running the command.

```
docker ps
```

You should see something like the following:

```
CONTAINER ID        IMAGE                               COMMAND                  CREATED␣
↪                   STATUS              PORTS               NAMES
b0ae19e4184c        hyperledger/fabric-starter-kit      "sh -c 'sleep 20; ..."   About a␣
↪minute ago    Up About a minute                           starter
cbed45603157        hyperledger/fabric-peer             "sh -c 'sleep 10; ..."   About a␣
↪minute ago    Up About a minute                           peer
49490fa0e109        hyperledger/fabric-membersrvc       "membersrvc"             About a␣
↪minute ago    Up About a minute                           membersrvc
```

- Start a terminal session in the **starter** container. This is where the Node.js application is located.

note: Be sure to wait 20 seconds after starting the network using the `docker-compose up` command before executing the following command to allow the network to initialize:

```
docker exec -it starter /bin/bash
```

- From the terminal session in the **starter** container execute the standalone Node.js application. The Docker terminal session should be in the working directory of the sample application called **app.js** (*/opt/gopath/src/github.com/hyperledger/fabric/examples/sdk/node*). Execute the following Node.js command to run the application:

```
node app
```

In another terminal session on the host you can view the logs for the peer by executing the following command (not in the docker shell above, in a new terminal session of the real system):

```
docker logs peer
```

- If you wish to run your own Node.js application using the pre-built Docker images:
- use the directories in the `volumes` tag under **starter** in the `docker-compose.yml` file as a place to store your programs from the host system into the docker container. The first path is the top level system (host system) and the second is created in the Docker container. If you wish to use a host location that is not under the `/Users` directory (~ is under '/Users') then you must add that to the Docker file sharing under Docker preferences.

```
volumes:
  - ~/mytest:/user/mytest
```

- copy or create and edit your application in the `~/mytest` directory as stated in the `docker-compose.yml` `volumes` tag under **starter** container.
- run npm to install Hyperledger fabric Node.js SDK in the `mytest` directory:

```
npm install /opt/gopath/src/github.com/hyperledger/fabric/sdk/node
```

- run the application from within the **starter** Docker container using the following commands:

```
docker exec -it starter /bin/bash
```

once in the shell, and assuming your Node.js application is called `app.js` :

```
cd /user/mytest
node app
```

- To shutdown the environment, execute the following **docker-compose** command in the directory where the *docker-compose.yml* is located. Any changes you made to the sample application or deployment of a chaincode will be lost. Only changes made to the shared area defined in the 'volumes' tag of the **starter** container will persist. This will shutdown each of the containers and remove the containers from Docker:

```
docker-compose down
```

or if you wish to keep your changes and just stop the containers, which will be restarted on the next `up` command:

```
docker-compose kill
```

CHAPTER 18

Writing, Building, and Running Chaincode in a Development Environment

Chaincode developers need a way to test and debug their chaincode without having to set up a complete peer network. By default, when you want to interact with chaincode, you need to first `Deploy` it using the CLI, REST API, gRPC API, or SDK. Upon receiving this request, the peer node would typically spin up a Docker container with the relevant chaincode. This can make things rather complicated for debugging chaincode under development, because of the turnaround time with the `launch chaincode -debug docker container -fix problem -launch chaincode -lather -rinse -repeat` cycle. As such, the fabric peer has a `--peer-chaincodedev` flag that can be passed on start-up to instruct the peer node not to deploy the chaincode as a Docker container.

The following instructions apply to *developing* chaincode in Go or Java. They do not apply to running in a production environment. However, if *developing* chaincode in Java, please see the Java chaincode setup instructions first, to be sure your environment is properly configured.

Note: We have added support for System chaincode

Choices

Once again, you have the choice of using one of the following approaches:

- *Option 1* using Docker for Mac or Windows
- *Option 2* using Docker toolbox
- *Option 3* using the **Vagrant** development environment that is used for developing the fabric itself

A Docker approach provides several advantages, highlighted through its simplicity.

By using options *1* or *2*, from above, you avoid having to build everything from scratch, and there's no need to keep a synchronized clone of the Hyperledger fabric codebase. Instead, you can simply pull and run the `fabric-peer` and `fabric-membersrvc` images directly from DockerHub. There is no need to manually start the peer and member service nodes, rather a single `docker-compose up` command will spin up a live network on your machine. Additionally, you are able to operate from a single terminal, and you avoid the extra layer of abstraction and virtualization which arises when using the Vagrant environment.

For more information on using Docker Compose, and customizing your Docker environment, see the Docker Setup Guide If you are not familiar with Docker and/or chaincode development, it's recommended to go through this section first.

Option 1 Docker for Mac or Windows

The Docker images for `fabric-peer` and `fabric-membersrvc` are continuously built and tested through the Hyperledger fabric CI (continuous integration). To run these fabric components on your Mac or Windows laptop/server using the Docker for Mac or Windows platform, follow these steps. If using Docker Toolbox, please skip to *Option 2*, below.

Pull images from DockerHub

You DO NOT need to manually pull the `fabric-peer` , `fabric-membersrvc` or `fabric-baseimage` images published by the Hyperledger Fabric project from DockerHub. These images are specified in the docker-compose.yaml and will be automatically downloaded and extracted when you run `docker-compose up` . However, you do need to ensure that the image tags correspond correctly to your platform.

Identify your platform and check the image tags. Use the **Tags** tab in the `hyperledger/fabric-baseimage` repository on DockerHub to browse the available images. For example, if you are running Docker natively on Linux or OSX then you will want:

```
hyperledger/fabric-baseimage:x86_64-0.2.0
```

Retrieve the docker-compose file

Now you need to retrieve a Docker Compose file to spin up your network. There are two standard Docker Compose files available. One is for a single node + CA network, and the second is for a four node + CA network. Identify or create a working directory where you want the Docker Compose file(s) to reside; this can be anywhere on your machine (the below directory is simply an example). Then execute a `cURL` to retrieve the .yaml file. For example, to retrieve the .yaml file for a single node + CA network:

```
mkdir -p $HOME/hyperledger/docker-compose
cd $HOME/hyperledger/docker-compose
curl https://raw.githubusercontent.com/hyperledger/fabric/v0.6/examples/docker-
↪compose/single-peer-ca.yaml -o single-peer-ca.yaml 2>/dev/null
```

OR to retrieve the .yaml file for a four node + CA network:

```
mkdir -p $HOME/hyperledger/docker-compose
cd $HOME/hyperledger/docker-compose
curl https://raw.githubusercontent.com/hyperledger/fabric/v0.6/examples/docker-
↪compose/four-peer-ca.yaml -o four-peer-ca.yaml 2>/dev/null
```

If you want to configure your network to use specific `fabric-peer` or `fabric-membersrvc` images from Hyperledger Docker Hub, use the **Tags** tab in the corresponding image repository to browse the available versions. Then add the tag in your Docker Compose .yaml file. For example, in the `single-peer-ca.yaml` you might alter the `hyperledger/fabric-peer` image from:

```
vp0:
    image: hyperledger/fabric-peer
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

to

```
vp0:
    image: hyperledger/fabric-peer:x86_64-0.6.1-preview
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

Running the Peer and CA

To run the `fabric-peer` and `fabric-membersrvc` images, you will use Docker Compose against one of your .yaml files. You specify the file after the `-f` argument on the command line. Therefore, to spin up the single node + CA network you first navigate to the working directory where your compose file(s) reside, and then execute `docker-compose up` from the command line:

```
cd $HOME/hyperledger/docker-compose
docker-compose -f single-peer-ca.yaml up
```

OR for a four node + CA network:

```
cd $HOME/hyperledger/docker-compose
docker-compose -f four-node-ca.yaml up
```

Now, you are ready to start *running the chaincode*.

Option 2 Docker Toolbox

If you are using Docker Toolbox, please follow these instructions.

Note: Docker will not run natively on older versions of macOS or any Windows versions prior to Windows 10. If either scenario describes your OS, you must use Docker Toolbox.

Docker Toolbox bundles Docker Engine, Docker Machine and Docker Compose, and by means of a VirtualBox, provides you with an environment to run Docker processes. You initialize the Docker host simply by launching the Docker Quick Start Terminal. Once the host is initialized, you can run all of the Docker commands and Docker Compose commands from the toolbox as if you were running them on the command line. Once you are in the toolbox, it is the same experience as if you were running on a Linux machine with Docker & Docker Compose installed.

Start up the default Docker host by clicking on the Docker Quick Start Terminal. It will open a new terminal window and initialize the Docker host. Once the startup process is complete, you will see the Docker whale together with the IP address of the Docker host, as shown below. In this example the IP address of the Docker host is 192.168.99.100. Take note of this IP address as you will need it later to connect to your Docker containers.

If you need to retrieve an IP address for one of your peers, use the `docker inspect` command. For more information on useful Docker commands, refer to the Docker documentation.

```
                        ##         .
                  ## ## ##        ==
               ## ## ## ## ##    ===
           /"""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
           \______ o           __/
             \    \         __/
              \____\_______/

docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com
```

Pull images from DockerHub

You DO NOT need to manually pull the `fabric-peer`, `fabric-membersrvc` or `fabric-baseimage` images published by the Hyperledger Fabric project from DockerHub.

These images are specified in the docker-compose.yaml and will be automatically downloaded and extracted when you run `docker-compose up`. However, you do need to ensure that the image tags correspond correctly to your platform.

Identify your platform and check the image tags. Use the **Tags** tab in the `hyperledger/fabric-baseimage` repository on DockerHub to browse the available images. If you are using Docker toolbox, then you will want:

```
hyperledger/fabric-baseimage:x86_64-0.2.0
```

Retrieve the docker-compose file

Now you need to retrieve a Docker Compose file to spin up your network. There are two standard Docker Compose files available. One is for a single node + CA network, and the second is for a four node + CA network. Identify or create a working directory where you want the Docker Compose file(s) to reside. Then execute a `cURL` to retrieve the .yaml file. For example, to retrieve the .yaml file for a single node + CA network:

```
mkdir -p $HOME/hyperledger/docker-compose
cd $HOME/hyperledger/docker-compose
curl https://raw.githubusercontent.com/hyperledger/fabric/master/examples/docker-
↪compose/single-peer-ca.yaml -o single-peer-ca.yaml 2>/dev/null
```

OR to retrieve the .yaml file for a four node + CA network:

```
mkdir -p $HOME/hyperledger/docker-compose
cd $HOME/hyperledger/docker-compose
curl https://raw.githubusercontent.com/hyperledger/fabric/master/examples/docker-
↪compose/four-peer-ca.yaml
-o four-peer-ca.yaml 2>dev/null
```

If you want to configure your network to use specific `fabric-peer` or `fabric-membersrvc` images from Hyperledger Docker Hub, use the **Tags** tab in the corresponding image repository to browse the available versions. Then add the tag in your Docker Compose .yaml file. For example, in the `single-peer-ca.yaml` you might alter the `hyperledger/fabric-peer` image from:

```
vp0:
    image: hyperledger/fabric-peer
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

to

```
vp0:
    image: hyperledger/fabric-peer:x86_64-0.6.1-preview
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

Running the Peer and CA

To run the `fabric-peer` and `fabric-membersrvc` images, you will use Docker Compose against one of your .yaml files. You specify the file through the `-f` argument on the command line. Therefore, to spin up the single node + CA network you first navigate to the working directory where your compose file(s) reside, and then execute `docker-compose up` from the command line:

```
cd $HOME/hyperledger/docker-compose
docker-compose -f single-peer-ca.yaml up
```

OR for a four node + CA network:

```
cd $HOME/hyperledger/docker-compose
docker-compose -f four-node-ca.yaml up
```

Now, you are ready to start *running the chaincode*.

Option 3 Vagrant development environment

You will need multiple terminal windows - essentially one for each component. One runs the validating peer; another runs the chaincode; the third runs the CLI or REST API commands to execute transactions. Finally, when running with security enabled, an additional fourth window is required to run the **Certificate Authority (CA)** server. Detailed instructions are provided in the sections below.

Note: Using the Vagrant environment results in a more complicated scenario due to an extra layer of virtualization and the need for multiple terminals.
Running Docker natively or using Docker Toolbox are the recommended approaches.

Security Setup (optional)

From the `devenv` subdirectory of your fabric workspace environment, `ssh` into Vagrant:

```
cd $GOPATH/src/github.com/hyperledger/fabric/devenv
vagrant ssh
```

To set up the local development environment with security enabled, you must first build and run the **Certificate Authority (CA)** server:

```
cd $GOPATH/src/github.com/hyperledger/fabric
make membersrvc && membersrvc
```

Running the above commands builds and runs the CA server with the default setup, which is defined in the membersrvc.yaml configuration file. The default configuration includes multiple users who are already registered with the CA; these users are listed in the `eca.users` section of the configuration file. To register additional users with the CA for testing, modify the `eca.users` section of the membersrvc.yaml file to include additional `enrollmentID` and `enrollmentPW` pairs. Note the integer that precedes the `enrollmentPW`. That integer indicates the role of the user, where 1 = client, 2 = non-validating peer, 4 = validating peer, and 8 = auditor.

Running the validating peer

Note: To run with security enabled, first modify the core.yaml configuration file to set the `security.enabled` value to `true` before building the peer executable. Alternatively, you can enable security by running the peer with the following environment variable: `CORE_SECURITY_ENABLED=true`. To enable privacy and confidentiality of transactions (which requires security to also be enabled), modify the core.yaml configuration file to set the `security.privacy` value to `true` as well. Alternatively, you can enable privacy by running the peer with the following environment variable: `CORE_SECURITY_PRIVACY=true`. If you are enabling security and privacy on the peer process with environment variables, it is important to include these environment variables in the command when executing all subsequent peer operations (e.g. deploy, invoke, or query).

In a **new** terminal window, from the `devenv` subdirectory of your fabric workspace environment, `ssh` into Vagrant:

```
cd $GOPATH/src/github.com/hyperledger/fabric/devenv
vagrant ssh
```

Build and run the peer process to enable security and privacy after setting `security.enabled` and `security.privacy` settings to `true`.

```
cd $GOPATH/src/github.com/hyperledger/fabric
make peer
peer node start --peer-chaincodedev
```

Alternatively, rather than tweaking the `core.yaml` and rebuilding, you can enable security and privacy on the peer with environment variables:

```
CORE_SECURITY_ENABLED=true CORE_SECURITY_PRIVACY=true peer node start --peer-
↪chaincodedev
```

Now, you are ready to start *running the chaincode*.

Running the chaincode

Docker or Docker Toolbox

Start a **new** terminal window. If you ran spun up your Docker containers in detached mode - `docker-compose up -d` - you can remain in the same terminal.

If you are using either *Option 1* or *Option 2*, you'll need to download the sample chaincode. The chaincode project must be placed somewhere under the `src` directory in your local `$GOPATH` as shown below.

```
mkdir -p $GOPATH/src/github.com/chaincode_example02/
cd $GOPATH/src/github.com/chaincode_example02
curl GET https://raw.githubusercontent.com/hyperledger/fabric/master/examples/
↪chaincode/go/chaincode_example02/chaincode_example02.go > chaincode_example02.go
```

Next, you'll need to clone the Hyperledger fabric to your local $GOPATH, so that you can build your chaincode. **Note:** this is a temporary stop-gap until we can provide an independent package for the chaincode shim.

```
mkdir -p $GOPATH/src/github.com/hyperledger
cd $GOPATH/src/github.com/hyperledger
git clone http://gerrit.hyperledger.org/r/fabric
```

Now, you should be able to build your chaincode.

```
cd $GOPATH/src/github.com/chaincode_example02
go build
```

When you are ready to start creating your own Go chaincode, create a new subdirectory under $GOPATH/src. You can copy the **chaincode_example02** file to the new directory and modify it.

Vagrant

Start a **new** terminal window.

If you are using *Option 3*, you'll need to `ssh` to Vagrant.

```
cd $GOPATH/src/github.com/hyperledger/fabric/devenv
vagrant ssh
```

Next, we'll build the **chaincode_example02** code, which is provided in the Hyperledger fabric source code repository. If you are using *Option 3*, then you can do this from your clone of the fabric repository.

```
cd $GOPATH/src/github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02
go build
```

Starting and registering the chaincode

Run the following chaincode command to start and register the chaincode with the validating peer:

```
CORE_CHAINCODE_ID_NAME=mycc CORE_PEER_ADDRESS=0.0.0.0:7051 ./chaincode_example02
```

The chaincode console will display the message "Received REGISTERED, ready for invocations", which indicates that the chaincode is ready to receive requests. Follow the steps below to send a chaincode deploy, invoke or query transaction. If the "Received REGISTERED" message is not displayed, then an error has occurred during the deployment; revisit the previous steps to resolve the issue.

Note: These instructions relate to writing, building, and running chaincode in "development" mode. This means that if you are using Docker, you will not see additional Docker containers after you have deployed your chaincode. Rather, the chaincode is directly registered with the peer as outlined in the above command.
See the Docker Setup Guide

Running the CLI or REST API

- *chaincode deploy via CLI and REST*
- *chaincode invoke via CLI and REST*
- *chaincode query via CLI and REST*

If you were running with security enabled, see *Removing temporary files when security is enabled* to learn how to clean up the temporary files.

See the logging control reference for information on controllinglogging output from the `peer` and chaincodes.

Terminal 3 (CLI or REST API)

Note on REST API port

The default REST interface port is `7050`. It can be configured in core.yaml using the `rest.address` property. If using Vagrant, the REST port mapping is defined in Vagrantfile.

Note on security functionality

Current security implementation assumes that end user authentication takes place at the application layer and is not handled by the fabric. Authentication may be performed through any means considered appropriate for the target application. Upon successful user authentication, the application will perform user registration with the CA exactly once. If registration is attempted a second time for the same user, an error will result. During registration, the application sends a request to the certificate authority to verify the user registration and if successful, the CA responds with the user certificates and keys. The enrollment and transaction certificates received from the CA will be stored locally inside `/var/hyperledger/production/crypto/client/` directory. This directory resides on a specific peer node which allows the user to transact only through this specific peer while using the stored crypto material. If the end user needs to perform transactions through more then one peer node, the application is responsible for replicating the crypto material to other peer nodes. This is necessary as registering a given user with the CA a second time will fail.

With security enabled, the CLI commands and REST payloads must be modified to include the `enrollmentID` of a registered user who is logged in; otherwise an error will result. A registered user can be logged in through the CLI or the REST API by following the instructions below. To log in through the CLI, issue the following commands, where `username` is one of the `enrollmentID` values listed in the `eca.users` section of the membersrvc.yaml file.

From your command line terminal, move to the `devenv` subdirectory of your workspace environment. Log into a Vagrant terminal by executing the following command:

```
vagrant ssh
```

Register the user though the CLI, substituting for `<username>` appropriately:

```
cd $GOPATH/src/github.com/hyperledger/fabric/peer
peer network login <username>
```

The command will prompt for a password, which must match the `enrollmentPW` listed for the target user in the `eca.users` section of the membersrvc.yaml file. If the password entered does not match the `enrollmentPW`, an error will result.

To log in through the REST API, send a POST request to the `/registrar` endpoint, containing the `enrollmentID` and `enrollmentPW` listed in the `eca.users` section of the membersrvc.yaml file.

REST Request:

```
POST localhost:7050/registrar

{
  "enrollId": "jim",
  "enrollSecret": "6avZQLwcUe9b"
}
```

REST Response:

```
200 OK
{
    "OK": "Login successful for user 'jim'."
}
```

chaincode deploy via CLI and REST

First, send a chaincode deploy transaction, only once, to the validating peer. The CLI connects to the validating peer using the properties defined in the core.yaml file. **Note:** The deploy transaction typically requires a `path` parameter to locate, build, and deploy the chaincode. However, because these instructions are specific to local development mode and the chaincode is deployed manually, the `name` parameter is used instead.

```
peer chaincode deploy -n mycc -c '{Args": ["init", "a","100", "b", "200"]}'
```

Alternatively, you can run the chaincode deploy transaction through the REST API.

REST Request:

```
POST <host:port>/chaincode

{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": 1,
    "chaincodeID":{
        "name": "mycc"
    },
    "ctorMsg": {
        "args":["init", "a", "100", "b", "200"]
    }
  },
  "id": 1
}
```

REST Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "mycc"
    },
    "id": 1
}
```

Note: When security is enabled, modify the CLI command and the REST API payload to pass the `enrollmentID` of a logged in user. To log in a registered user through the CLI or the REST API, follow the instructions in the *note on security functionality*. On the CLI, the `enrollmentID` is passed with the `-u` parameter; in the REST API, the `enrollmentID` is passed with the `secureContext` element. If you are enabling security and privacy on the peer process with environment variables, it is important to include these environment variables in the command when executing all subsequent peer operations (e.g. deploy, invoke, or query).

```
CORE_SECURITY_ENABLED=true CORE_SECURITY_PRIVACY=true peer chaincode deploy -u
jim -n mycc -c '{"Args": ["init", "a","100", "b", "200"]}'
```

REST Request:

```
POST <host:port>/chaincode

{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": 1,
    "chaincodeID":{
        "name": "mycc"
    },
    "ctorMsg": {
        "args":["init", "a", "100", "b", "200"]
    },
    "secureContext": "jim"
  },
  "id": 1
}
```

The deploy transaction initializes the chaincode by executing a target initializing function. Though the example shows "init", the name could be arbitrarily chosen by the chaincode developer. You should see the following output in the chaincode window:

```
  <TIMESTAMP_SIGNATURE> Received INIT(uuid:005dea42-d57f-4983-803e-3232e551bf61),
initializing chaincode Aval = 100, Bval = 200
```

Chaincode invoke via CLI and REST

Run the chaincode invoking transaction on the CLI as many times as desired. The -n argument should match the value provided in the chaincode window (started in Vagrant terminal 2):

```
peer chaincode invoke -l golang -n mycc -c '{Args": ["invoke", "a", "b", "10"]}'
```

Alternatively, run the chaincode invoking transaction through the REST API.

REST Request:

```
POST <host:port>/chaincode

{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"mycc"
      },
      "ctorMsg": {
         "args":["invoke", "a", "b", "10"]
      }
  },
  "id": 3
}
```

REST Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "5a4540e5-902b-422d-a6ab-e70ab36a2e6d"
    },
    "id": 3
}
```

Note: When security is enabled, modify the CLI command and REST API payload to pass the `enrollmentID` of a logged in user. To log in a registered user through the CLI or the REST API, follow the instructions in the *note on security functionality*. On the CLI, the `enrollmentID` is passed with the `-u` parameter; in the REST API, the `enrollmentID` is passed with the `secureContext` element. If you are enabling security and privacy on the peer process with environment variables, it is important to include these environment variables in the command when executing all subsequent peer operations (e.g. deploy, invoke, or query).

```
  CORE_SECURITY_ENABLED=true CORE_SECURITY_PRIVACY=true peer chaincode invoke
-u jim -l golang -n mycc -c '{"Function": "invoke", "Args": ["a", "b", "10"]}'
```

REST Request:

```
POST <host:port>/chaincode

{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"mycc"
      },
      "ctorMsg": {
         "args":["invoke", "a", "b", "10"]
      },
      "secureContext": "jim"
  },
  "id": 3
}
```

The invoking transaction runs the specified chaincode function name "invoke" with the arguments. This transaction transfers 10 units from A to B. You should see the following output in the chaincode window:

```
  <TIMESTAMP_SIGNATURE> Received RESPONSE. Payload 200,
Uuid 075d72a4-4d1f-4a1d-a735-4f6f60d597a9 Aval = 90, Bval = 210
```

Chaincode query via CLI and REST

Run a query on the chaincode to retrieve the desired values. The `-n` argument should match the value provided in the chaincode window (started in Vagrant terminal 2):

```
peer chaincode query -l golang -n mycc -c '{"Args": ["query", "b"]}'
```

The response should be similar to the following:

```
{"Name":"b","Amount":"210"}
```

If a name other than "a" or "b" is provided in a query sent to `chaincode_example02`, you should see an error response similar to the following:

```
{"Error":"Nil amount for c"}
```

Alternatively, run the chaincode query transaction through the REST API.

REST Request:

```
POST <host:port>/chaincode

{
  "jsonrpc": "2.0",
  "method": "query",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"mycc"
      },
      "ctorMsg": {
         "args":["query", "a"]
      }
  },
  "id": 5
}
```

REST Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "90"
    },
    "id": 5
}
```

Note: When security is enabled, modify the CLI command and REST API payload to pass the `enrollmentID` of a logged in user. To log in a registered user through the CLI or the REST API, follow the instructions in the *note on security functionality*. On the CLI, the `enrollmentID` is passed with the `-u` parameter; in the REST API, the `enrollmentID` is passed with the `secureContext` element. If you are enabling security and privacy on the peer process with environment variables, it is important to include these environment variables in the command when executing all subsequent peer operations (e.g. deploy, invoke, or query).

```
  CORE_SECURITY_ENABLED=true CORE_SECURITY_PRIVACY=true peer chaincode query
-u jim -l golang -n mycc -c '{Args": ["query", "b"]}'
```

REST Request:

```
POST <host:port>/chaincode

{
  "jsonrpc": "2.0",
  "method": "query",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"mycc"
```

```
        },
        "ctorMsg": {
           "args":["query", "a"]
        },
        "secureContext": "jim"
    },
    "id": 5
}
```

Removing temporary files when security is enabled

Note: this step applies **ONLY** if you were using Option 1 above. For Option 2 or 3, the cleanup is handled by Docker.

After the completion of a chaincode test with security enabled, remove the temporary files that were created by the CA server process. To remove the client enrollment certificate, enrollment key, transaction certificate chain, etc., run the following commands. Note, that you must run these commands if you want to register a user who has already been registered previously.

From your command line terminal, `ssh` into Vagrant:

```
cd $GOPATH/src/github.com/hyperledger/fabric/devenv
vagrant ssh
```

And then run:

```
rm -rf /var/hyperledger/production
```

CHAPTER 19

Java chaincode

Note: This guide generally assumes you have followed the Chaincode development environment setup tutorial here.

To get started developing Java chaincode

1. Ensure you have gradle

- Download the binary distribution from http://gradle.org/gradle-download/
- Unpack, move to the desired location, and add gradle's bin directory to your system path
- Ensure `gradle -v` works from the command-line, and shows version 2.12 or greater
- Optionally, enable the gradle daemon for faster builds

2. Ensure you have the Java 1.8 **JDK** installed. Also ensure Java's directory is on your path with `java -version`

- Additionally, you will need to have the ``JAVA HOME` <https://docs.oracle.com/cd/E19182-01/821-0917/6nluh6gq9/index.html>`__ variable set to your **JDK** installation in your system path

3. From your command line terminal, move to the `devenv` subdirectory of your workspace environment. Log into a Vagrant terminal by executing the following command:

```
vagrant ssh
```

4. Build and run the peer process.

```
cd $GOPATH/src/github.com/hyperledger/fabric
make peer
peer node start
```

5. The following steps is for deploying chaincode in non-dev mode.

   - Deploy the chaincode,

```
peer chaincode deploy -l java -p /opt/gopath/src/github.com/hyperledger/fabric/
↪examples/chaincode/java/SimpleSample -c '{"Args": ["init", "a","100", "b", "200"]}'
```

```
6d9a704d95284593fe802a5de89f84e86fb975f00830bc6488713f9441b835cf32d9cd07b087b90e5cb57a88360
```

```
* This command will give the 'name' for this chaincode, and use this value in all the␣
↪further commands with the -n (name) parameter


* PS. This may take a few minutes depending on the environment as it deploys the␣
↪chaincode in the container,
```

- Invoke a transfer transaction,

```
peer chaincode invoke -l java \
-n␣
↪6d9a704d95284593fe802a5de89f84e86fb975f00830bc6488713f9441b835cf32d9cd07b087b90e5cb57a88360f90a4de
↪\
-c '{"Args": ["transfer", "a", "b", "10"]}'
```

```
c7dde1d7-fae5-4b68-9ab1-928d61d1e346
```

- Query the values of a and b after the transfer

```
peer chaincode query -l java \
-n␣
↪6d9a704d95284593fe802a5de89f84e86fb975f00830bc6488713f9441b835cf32d9cd07b087b90e5cb57a88360f90a4de
↪\
-c '{ "Args": ["query", "a"]}'
{"Name":"a","Amount":"80"}


peer chaincode query -l java \
-n␣
↪6d9a704d95284593fe802a5de89f84e86fb975f00830bc6488713f9441b835cf32d9cd07b087b90e5cb57a88360f90a4de
↪\
-c '{ "Args": ["query", "b"]}'
{"Name":"b","Amount":"220"}
```

Java chaincode deployment in DEV Mode

1. Follow the step 1 to 3 as above,
2. Build and run the peer process

   ```
   cd $GOPATH/src/github.com/hyperledger/fabric
   make peer
   peer node start --peer-chaincodedev
   ```

3. Open the second Vagrant terminal and build the Java shim layer and publish it to Local Maven Repo

   ```
   cd $GOPATH/src/github.com/hyperledger/fabric/core/chaincode/shim/java
   gradle -b build.gradle clean
   gradle -b build.gradle build
   ```

4. Change to examples folder to build and run,

   ```
   cd $GOPATH/src/github.com/hyperledger/fabric/examples/chaincode/java/SimpleSample
   gradle -b build.gradle build
   ```

5. Run the SimpleSample chaincode using the `gradle -b build.gradle run`

6. Open the third Vagrant terminal to run init and invoke on the chaincode

   peer chaincode deploy -l java -n SimpleSample -c '{"Args": ["init", "a","100", "b", "200"]}'

   ```
   2016/06/28 19:10:15 Load docker HostConfig: %+v &{[] [] []  [] false map[] []␣
   ↪false [] [] [] [] host    { 0} [] { map[]} false []  0 0 0 false 0    0 0 0 []}
   19:10:15.461 [crypto] main -> INFO 002 Log level recognized 'info', set to INFO
   SimpleSample
   ```

   peer chaincode invoke -l java -n SimpleSample -c '{"Args": ["transfer", "a", "b", "10"]}'

```
2016/06/28 19:11:13 Load docker HostConfig: %+v &{[] [] []  [] false map[] [] false␣
↪[] [] [] [] host    { 0} [] { map[]} false []  0 0 0 false 0    0 0 0 []}
19:11:13.553 [crypto] main -> INFO 002 Log level recognized 'info', set to INFO
978ff89e-e4ef-43da-a9f8-625f2f6f04e5
```

```
peer chaincode query -l java -n SimpleSample -c '{ "Args": ["query", "a"]}'
```

```
2016/06/28 19:12:19 Load docker HostConfig: %+v &{[] [] []  [] false map[] [] false␣
↪[] [] [] [] host    { 0} [] { map[]} false []  0 0 0 false 0    0 0 0 []}
19:12:19.289 [crypto] main -> INFO 002 Log level recognized 'info', set to INFO
{"Name":"a","Amount":"90"}
```

```
peer chaincode query -l java -n SimpleSample -c '{"Args": ["query", "b"]}'
```

```
2016/06/28 19:12:25 Load docker HostConfig: %+v &{[] [] []  [] false map[] [] false␣
↪[] [] [] [] host    { 0} [] { map[]} false []  0 0 0 false 0    0 0 0 []}
19:12:25.667 [crypto] main -> INFO 002 Log level recognized 'info', set to INFO
{"Name":"b","Amount":"210"}
```

Developing new JAVA chaincode

1. Create a new Java project structure.
2. Use existing `build.grade` from any example JAVA Chaincode project like `examples/chaincode/java/SimpleSample`.
3. Make your main class extend ChaincodeBase class and implement the following methods from base class.
4. `public String run(ChaincodeStub stub,String function,String[] args)`
5. `public String query(ChaincodeStub stub,String function,String[] args)`
6. `public String getChaincodeID()`
7. Modify the `mainClassName` in `build.gradle` to point to your new class.
8. Build this project using `gradle -b build.gradle build`
9. Run this chaincode after starting a peer in dev-mode as above using `gradle -b build.gradle run`

Setting Up a Network

This document covers setting up a network on your local machine for various development and testing activities. Unless you are intending to contribute to the development of the Hyperledger Fabric project, you'll probably want to follow the more commonly used approach below - *leveraging published Docker images* for the various Hyperledger Fabric components, directly. Otherwise, skip down to the *secondary approach* below.

Leveraging published Docker images

This approach simply leverages the Docker images that the Hyperledger Fabric project publishes to DockerHub and either Docker commands or Docker Compose descriptions of the network one wishes to create.

Installing Docker

Note: When running Docker *natively* on Mac and Windows, there is no IP forwarding support available. Hence, running more than one fabric-peer image is not advised because you do not want to have multiple processes binding to the same port. For most application and chaincode development/testing running with a single fabric peer should not be an issue unless you are interested in performance and resilience testing the fabric's capabilities, such as consensus. For more advanced testing, we strongly recommend using the fabric's Vagrant *development environment*

With this approach, there are multiple choices as to how to run Docker: using Docker Toolbox or one of the new native Docker runtime environments for Mac OSX or Windows. There are some subtle differences between how Docker runs natively on Mac and Windows versus in a virtualized context on Linux. We'll call those out where appropriate below, when we get to the point of actually running the various components.

Pulling the images from DockerHub

Once you have Docker (1.11 or greater) installed and running, prior to starting any of the fabric components, you will need to first pull the fabric images from DockerHub.

```
docker pull hyperledger/fabric-peer:latest
docker pull hyperledger/fabric-membersrvc:latest
```

Building your own images

Note: *This approach is not necessarily recommended for most users*. If you have pulled images from DockerHub as described in the previous section, you may proceed to the *next step*.

The second approach would be to leverage the *development environment* setup (which we will assume you have already established) to build and deploy your own binaries and/or Docker images from a clone of the hyperledger/fabric GitHub repository. This approach is suitable for developers that might wish to contribute directly to the Hyperledger Fabric project, or that wish to deploy from a fork of the Hyperledger code base.

The following commands should be run from *within* the Vagrant environment described in *Setting Up Development Environment*

To create the Docker image for the `hyperledger/fabric-peer`:

```
cd $GOPATH/src/github.com/hyperledger/fabric
make peer-image
```

To create the Docker image for the `hyperledger/fabric-membersrvc`:

```
make membersrvc-image
```

Starting up validating peers

Check the available images again with `docker images`. You should see `hyperledger/fabric-peer` and `hyperledger/fabric-membersrvc` images. For example,

```
$ docker images
REPOSITORY                      TAG                 IMAGE ID            CREATED             ␣
↪       SIZE
hyperledger/fabric-membersrvc   latest              7d5f6e0bcfac        12 days ago         ␣
↪       1.439 GB
hyperledger/fabric-peer         latest              82ef20d7507c        12 days ago         ␣
↪       1.445 GB
```

If you don't see these, go back to the previous step.

With the relevant Docker images in hand, we can start running the peer and membersrvc services.

Determine value for CORE_VM_ENDPOINT variable

Next, we need to determine the address of your docker daemon for the CORE_VM_ENDPOINT. If you are working within the Vagrant development environment, or a Docker Toolbox environment, you can determine this with the `ip add` command. For example,

```
$ ip add

<<< detail removed >>>

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN␣
↪group default
    link/ether 02:42:ad:be:70:cb brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:adff:febe:70cb/64 scope link
       valid_lft forever preferred_lft forever
```

Your output might contain something like `inet 172.17.0.1/16 scope global docker0`. That means the docker0 interface is on IP address 172.17.0.1. Use that IP address for the `CORE_VM_ENDPOINT` option. For more information on the environment variables, see `core.yaml` configuration file in the `fabric` repository.

If you are using the native Docker for Mac or Windows, the value for `CORE_VM_ENDPOINT` should be set to `unix:///var/run/docker.sock`. [TODO] double check this. I believe that `127.0.0.1:2375` also works.

Assigning a value for CORE_PEER_ID

The ID value of `CORE_PEER_ID` must be unique for each validating peer, and it must be a lowercase string. We often use a convention of naming the validating peers vpN where N is an integer starting with 0 for the root node and incrementing N by 1 for each additional peer node started. e.g. vp0, vp1, vp2, ...

Consensus

By default, we are using a consensus plugin called `NOOPS`, which doesn't really do consensus. If you are running a single peer node, running anything other than `NOOPS` makes little sense. If you want to use some other consensus plugin in the context of multiple peer nodes, please see the *Using a Consensus Plugin* section, below.

Docker Compose

We'll be using Docker Compose to launch our various Fabric component containers, as this is the simplest approach. You should have it installed from the initial setup steps. Installing Docker Toolbox or any of the native Docker runtimes should have installed Compose.

Start up a validating peer:

Let's launch the first validating peer (the root node). We'll set CORE_PEER_ID to vp0 and CORE_VM_ENDPOINT as above. Here's the docker-compose.yml for launching a single container within the **Vagrant** *development environment*

```
vp0:
  image: hyperledger/fabric-peer
  environment:
    - CORE_PEER_ID=vp0
    - CORE_PEER_ADDRESSAUTODETECT=true
    - CORE_VM_ENDPOINT=http://172.17.0.1:2375
    - CORE_LOGGING_LEVEL=DEBUG
  command: peer node start
```

You can launch this Compose file as follows, from the same directory as the docker-compose.yml file:

```
$ docker-compose up
```

Here's the corresponding Docker command:

```
$ docker run --rm -it -e CORE_VM_ENDPOINT=http://172.17.0.1:2375 -e CORE_LOGGING_
↪LEVEL=DEBUG -e CORE_PEER_ID=vp0 -e CORE_PEER_ADDRESSAUTODETECT=true hyperledger/
↪fabric-peer peer node start
```

If you are running Docker for Mac or Windows, we'll need to explicitly map the ports, and we will need a different value for CORE_VM_ENDPOINT as we discussed above.

Here's the docker-compose.yml for Docker on Mac or Windows:

```
vp0:
  image: hyperledger/fabric-peer
  ports:
    - "7050:7050"
    - "7051:7051"
    - "7052:7052"
  environment:
    - CORE_PEER_ADDRESSAUTODETECT=true
    - CORE_VM_ENDPOINT=unix:///var/run/docker.sock
    - CORE_LOGGING_LEVEL=DEBUG
  command: peer node start
```

This single peer configuration, running the `NOOPS` 'consensus' plugin, should satisfy many development/test scenarios. `NOOPS` is not really providing consensus, it is essentially a no-op that simulates consensus. For instance, if you are simply developing and testing chaincode; this should be adequate unless your chaincode is leveraging membership services for identity, access control, confidentiality and privacy.

Running with the CA

If you want to take advantage of security (authentication and authorization), privacy and confidentiality, then you'll need to run the Fabric's certificate authority (CA). Please refer to the *CA Setup* instructions.

Start up additional validating peers:

Following the pattern we established *above* we'll use `vp1` as the ID for the second validating peer. If using Docker Compose, we can simply link the two peer nodes. Here's the docker-compse.yml for a **Vagrant** environment with two peer nodes - vp0 and vp1:

```
vp0:
  image: hyperledger/fabric-peer
  environment:
    - CORE_PEER_ADDRESSAUTODETECT=true
    - CORE_VM_ENDPOINT=http://172.17.0.1:2375
    - CORE_LOGGING_LEVEL=DEBUG
  command: peer node start
vp1:
  extends:
    service: vp0
  environment:
    - CORE_PEER_ID=vp1
    - CORE_PEER_DISCOVERY_ROOTNODE=vp0:7051
  links:
    - vp0
```

If we wanted to use the docker command line to launch another peer, we need to get the IP address of the first validating peer, which will act as the root node to which the new peer(s) will connect. The address is printed out on the terminal window of the first peer (e.g. 172.17.0.2) and should be passed in with the `CORE_PEER_DISCOVERY_ROOTNODE` environment variable.

```
docker run --rm -it -e CORE_VM_ENDPOINT=http://172.17.0.1:2375 -e CORE_PEER_ID=vp1 -e␣
↪CORE_PEER_ADDRESSAUTODETECT=true -e CORE_PEER_DISCOVERY_ROOTNODE=172.17.0.2:7051␣
↪hyperledger/fabric-peer peer node start
```

Using a Consensus Plugin

A consensus plugin might require some specific configuration that you need to set up. For example, to use the Practical Byzantine Fault Tolerant (PBFT) consensus plugin provided as part of the fabric, perform the following configuration:

1. In `core.yaml`, set the `peer.validator.consensus` value to `pbft`
2. In `core.yaml`, make sure the `peer.id` is set sequentially as `vpN` where `N` is an integer that starts from `0` and goes to `N-1`. For example, with 4 validating peers, set the `peer.id` to`vp0`, `vp1`, `vp2`, `vp3`.
3. In `consensus/pbft/config.yaml`, set the `general.mode` value to `batch` and the `general.N` value to the number of validating peers on the network, also set `general.batchsize` to the number of transactions per batch.
4. In `consensus/pbft/config.yaml`, optionally set timer values for the batch period (`general.timeout.batch`), the acceptable delay between request and execution (`general.timeout.request`), and for view-change (`general.timeout.viewchange`)

See `core.yaml` and `consensus/pbft/config.yaml` for more detail.

All of these setting may be overridden via the command line environment variables, e.g. `CORE_PEER_VALIDATOR_CONSENSUS_PLUGIN=pbft` or `CORE_PBFT_GENERAL_MODE=batch`

Logging control

See *Logging Control* for information on controlling logging output from the `peer` and deployed chaincodes.

CHAPTER 21

Hyperledger Fabric Client (HFC) SDK for Node.js

The Hyperledger Fabric Client (HFC) SDK provides a powerful and easy to use API to interact with a Hyperledger Fabric blockchain.

This document assumes that you already have set up a Node.js development environment. If not, go here to download and install Node.js for your OS. You'll also want the latest version of `npm` installed. For that, execute `sudo npm install npm -g` to get the latest version.

Installing the hfc module

We publish the `hfc` node module to `npm`. To install `hfc` from npm simply execute the following command:

```
npm install -g hfc
```

See *Hyperledger fabric Node.js client SDK* for more information.

Hyperledger Fabric network

First, you'll want to have a running peer node and member services. The instructions for setting up a network are *here* You may also use the *Fabric-starter-kit* that provides the network.

CHAPTER 22

Certificate Authority (CA) Setup

The *Certificate Authority* (CA) provides a number of certificate services to users of a blockchain. More specifically, these services relate to *user enrollment*, *transactions* invoked on the blockchain, and *TLS*-secured connections between users or components of the blockchain.

This guide builds on either the *fabric developer's setup* or the prerequisites articulated in the *fabric network setup* guide. If you have not already set up your environment with one of those guides, please do so before continuing.

Enrollment Certificate Authority

The *enrollment certificate authority* (ECA) allows new users to register with the blockchain network and enables registered users to request an *enrollment certificate pair*. One certificate is for data signing, one is for data encryption. The public keys to be embedded in the certificates have to be of type ECDSA, whereby the key for data encryption is then converted by the user to be used in an ECIES (Elliptic Curve Integrated Encryption System) fashion.

Transaction Certificate Authority

Once a user is enrolled, he or she can also request *transaction certificates* from the *transaction certificate authority* (TCA). These certificates are to be used for deploying Chaincode and for invoking Chaincode transactions on the blockchain. Although a single *transaction certificate* can be used for multiple transactions, for privacy reasons it is recommended that a new *transaction certificate* be used for each transaction.

TLS Certificate Authority

In addition to *enrollment certificates* and *transaction certificates*, users will need *TLS certificates* to secure their communication channels. *TLS certificates* can be requested from the *TLS certificate authority* (TLSCA).

Configuration

All CA services are provided by a single process, which can be configured by setting parameters in the CA configuration file `membersrvc.yaml` , which is located in the same directory as the CA binary. More specifically, the following parameters can be set:

- `server.gomaxprocs` : limits the number of operating system threads used by the CA.

- `server.rootpath` : the root path of the directory where the CA stores its state.
- `server.cadir` : the name of the directory where the CA stores its state.
- `server.port` : the port at which all CA services listen (multiplexing of services over the same port is provided by GRPC).

Furthermore, logging levels can be enabled/disabled by adjusting the following settings:

- `logging.trace` (off by default, useful for debugging the code only)
- `logging.info`
- `logging.warning`
- `logging.error`
- `logging.panic`

Alternatively, these fields can be set via environment variables, which—if set—have precedence over entries in the yaml file. The corresponding environment variables are named as follows:

```
MEMBERSRVC_CA_SERVER_GOMAXPROCS
MEMBERSRVC_CA_SERVER_ROOTPATH
MEMBERSRVC_CA_SERVER_CADIR
MEMBERSRVC_CA_SERVER_PORT
```

In addition, the CA may be preloaded with registered users, where each user's name, roles, and password are specified:

```
eca:
    users:
        alice: 2 DRJ20pEql15a
        bob: 4 7avZQLwcUe9q
```

The role value is simply a bitmask of the following:

```
CLIENT = 1;
PEER = 2;
VALIDATOR = 4;
AUDITOR = 8;
```

For example, a peer that is also a validator would have a role value of 6.

When the CA is started for the first time, it will generate all of its required state (e.g., internal databases, CA certificates, blockchain keys, etc.) and writes this state to the directory given in its configuration. The certificates for the CA services (i.e., for the ECA, TCA, and TLSCA) are self-signed as the current default. If those certificates shall be signed by some root CA, this can be done manually by using the `*.priv` and `*.pub` private and public keys in the CA state directory, and replacing the self-signed `*.cert` certificates with root-signed ones. The next time the CA is launched, it will read and use those root-signed certificates.

Operating the CA

You can either *build and run* the CA from source. Or, you can use Docker Compose and work with the published images on DockerHub, or some other Docker registry. Using Docker Compose is by far the simplest approach.

Docker Compose

Here's a sample docker-compose.yml for the CA.

```
membersrvc:
  image: hyperledger/fabric-membersrvc
  command: membersrvc
```

The corresponding docker-compose.yml for running Docker on Mac or Windows natively looks like this:

```
membersrvc:
  image: hyperledger/fabric-membersrvc
  ports:
    - "7054:7054"
  command: membersrvc
```

If you are launching one or more `peer` nodes in the same docker-compose.yml, then you will want to add a delay to the start of the peer to allow sufficient time for the CA to start, before the peer attempts to connect to it.

```
membersrvc:
  image: hyperledger/fabric-membersrvc
  command: membersrvc
vp0:
  image: hyperledger/fabric-peer
  environment:
    - CORE_PEER_ADDRESSAUTODETECT=true
    - CORE_VM_ENDPOINT=http://172.17.0.1:2375
    - CORE_LOGGING_LEVEL=DEBUG
    - CORE_PEER_ID=vp0
    - CORE_SECURITY_ENROLLID=test_vp0
    - CORE_SECURITY_ENROLLSECRET=MwYpmSRjupbT
  links:
    - membersrvc
  command: sh -c "sleep 5; peer node start"
```

The corresponding docker-compose.yml for running Docker on Mac or Windows natively looks like this:

```
membersrvc:
  image: hyperledger/fabric-membersrvc
  ports:
    - "7054:7054"
  command: membersrvc
vp0:
  image: hyperledger/fabric-peer
  ports:
    - "7050:7050"
    - "7051:7051"
    - "7052:7052"
  environment:
    - CORE_PEER_ADDRESSAUTODETECT=true
    - CORE_VM_ENDPOINT=unix:///var/run/docker.sock
    - CORE_LOGGING_LEVEL=DEBUG
    - CORE_PEER_ID=vp0
    - CORE_SECURITY_ENROLLID=test_vp0
    - CORE_SECURITY_ENROLLSECRET=MwYpmSRjupbT
  links:
    - membersrvc
  command: sh -c "sleep 5; peer node start"
```

Build and Run

The CA can be built with the following command executed in the `membersrvc` directory:

```
cd $GOPATH/src/github.com/hyperledger/fabric
make membersrvc
```

The CA can be started with the following command:

```
build/bin/membersrvc
```

Note: the CA must be started before any of the fabric peer nodes, to allow the CA to have initialized before any peer nodes attempt to connect to it.

The CA looks for an `membersrvc.yaml` configuration file in $GOPATH/src/github.com/hyperledger/fabric/membersrvc. If the CA is started for the first time, it creates all its required state (e.g., internal databases, CA certificates, blockchain keys, etc.) and writes that state to the directory given in the CA configuration.

Enabling TLS

Follow this document to enable TLS for all servers (ECA, ACA, TLSCA, TCA) and between ACA client to server communications.

1. Go to **memebersrvc.yaml** file under the fabric/membersrvc directory and edit security section, that is:

```
security:
  serverhostoverride:
  tls_enabled: false
  client:
cert:
 file:
```

   To enable TLS between the ACA client and the rest of the CA Services set the `tls_enbabled` flag to `true`.

2. Next, set **serverhostoverride** field to match **CN** (Common Name) of TLS Server certificate. To extract the Common Name from TLS Server's certificate, for example using OpenSSL, you can use the following command:

```
openssl x509 -in <<certificate.crt -text -noout
```

where `certficate.crt` is the Server Certificate. If you have openssl installed on the machine and everything went well, you should expect an output of the form:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            4f:39:0f:ac:7b:ce:2b:9f:28:57:52:4a:bb:94:a6:e5:9c:69:99:56
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, ST=California, L=San Francisco, O=Internet Widgets, Inc., OU=WWW
        Validity
            Not Before: Aug 24 16:27:00 2016 GMT
            Not After : Aug 24 16:27:00 2017 GMT
        **Subject**: C=US, ST=California, L=San Francisco, O=example.com, **CN=www.
↪example.com**
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
            EC Public Key:
                pub:
                    04:38:d2:62:75:4a:18:d9:f7:fe:6a:e7:df:32:e2:
                    15:0f:01:9c:1b:4f:dc:ff:22:97:5c:2a:d9:5c:c3:
                    a3:ef:e3:90:3b:3c:8a:d2:45:b1:60:11:94:5e:a7:
                    51:e8:e5:5d:be:38:39:da:66:e1:99:46:0c:d3:45:
```

```
                    3d:76:7e:b7:8c
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                E8:9C:86:81:59:D4:D7:76:43:C7:2E:92:88:30:1B:30:A5:B3:A4:5C
            X509v3 Authority Key Identifier:
                keyid:5E:33:AC:E0:9D:B9:F9:71:5F:1F:96:B5:84:85:35:BE:89:8C:35:C2

            X509v3 Subject Alternative Name:
                DNS:www.example.com
    Signature Algorithm: ecdsa-with-SHA256
        30:45:02:21:00:9f:7e:93:93:af:3d:cf:7b:77:f0:55:2d:57:
        9d:a9:bf:b0:8c:9c:2e:cf:b2:b4:d8:de:f3:79:c7:66:7c:e7:
        4d:02:20:7e:9b:36:d1:3a:df:e4:d2:d7:3b:9d:73:c7:61:a8:
        2e:a5:b1:23:10:65:81:96:b1:3b:79:d4:a6:12:fe:f2:69
```

Now you can use that CN value (**www.example.com** above, for example) from the output and use it in the **serverhostoverride** field (under the security section of the membersrvc.yaml file)

3. Last, make sure that path to the corresponding TLS Server Certificate is specified under `security.client.cert.file`

CHAPTER 24

Logging Control

Overview

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

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

All logs are currently directed to `stderr`, and the pretty-printing is currently fixed. However global and module-level control of logging by severity is provided for both users and developers. There are currently no formalized rules for the types of information provided at each severity level, however when submitting bug reports the developers may want to see full logs down to the DEBUG level.

In pretty-printed logs the logging level is indicated both by color and by a 4-character code, e.g, “ERRO” for ERROR, “DEBU” for DEBUG, etc. In the logging context a *module* is an arbitrary name (string) given by developers to groups of related messages. In the pretty-printed example below, the logging modules “peer”, “rest” and “main” are generating logs.

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

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

Peer

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

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

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

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

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

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

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

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

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

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

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

Go chaincodes

As independently executed programs, user-provided chaincodes can use any appropriate technique to create their private logs - from simple print statements to fully-annotated and level-controlled logs. 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.

`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 25

Chaincode APIs

When the `Init` , `Invoke` or `Query` function of a chaincode is called, the fabric passes the `stub *shim.ChaincodeStub` parameter. 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, generated by `godoc` . However, it includes functions from chaincode.pb.go such as `func (*Column) XXX_OneofFuncs` that are not intended as public API. The best is to look at the function definitions in chaincode.go and chaincode samples for usage.

CHAPTER 26

APIs - CLI, REST, and Node.js

Overview

This document covers the available APIs for interacting with a peer node. Three interface choices are provided:

1. *CLI*
2. *REST API*
3. *Node.js Application*

- *Using Swagger JS Plugin*
- *Marbles Demo Application*
- *Commercial Paper Demo Application*

Note: If you are working with APIs with security enabled, please review the security setup instructions before proceeding.

CLI

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

```
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.
  network     network specific commands.
  chaincode   chaincode specific commands.
  help        Help about any command

Flags:
  -h, --help[=false]: help for peer
```

```
      --logging-level="": Default logging level and overrides, see core.yaml for full␣
↪syntax
      --test.coverprofile="coverage.cov": Done
  -v, --version[=false]: Show current version number 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:

Comm and	**st dout ** resu lt in the even t of succ ess
`ve rsio n`	Stri ng form of `pe er.v ersi on` defi ned in **'cor e.ya ml < http s:// gith ub.c om/h yper ledg er/f abri c/bl ob/m aste r/pe er/c ore. yaml >'__**
''no de s tart ''	N/A
`no de s tatu s`	Stri ng form of **'Sta tusC ode <htt ps:/ /git hub. com/ hype rled ger/ fabr ic/b lob/ mast er/p roto s/se rver _adm in.p roto #L36 >'__**
''no de s top' '	Stri ng form of **'Sta tusC ode <htt ps:/ /git hub. com/ hype rled ger/ fabr ic/b lob/ mast er/p roto s/se rver _adm in.p roto #L36 >'__**
''ne twor k lo gin' '	N/A
`ne twor k li st`	The list of netw ork conn ecti ons to the peer node .
`ch ainc ode depl oy`	The chai ncod e cont aine r name (has h) requ ired for subs eque nt `ch ainc ode invo ke` and `ch ainc ode quer y` comm ands
`ch ainc ode invo ke`	The tran sact ion ID (UUI D)
`ch ainc ode quer y`	By defa ult, the quer y resu lt is form atte d as a prin tabl e stri ng. Comm and line opti ons supp ort writ ing this valu e as raw byte s (-r, –ra w), or form atte d as the hexa deci mal repr esen tati on of the raw byte s (-x, –he x). If the quer y resp onse is empt y then noth ing is outp ut.

Deploy a Chaincode

Deploy creates the docker image for the chaincode and subsequently deploys the package to the validating peer. An example is below.

```
peer chaincode deploy -p github.com/hyperledger/fabric/examples/chaincode/go/chaincode_exan
-c '{"Function":"init","Args": ["a","100","b","200"]}'
```

The response to the chaincode deploy command will contain the chaincode identifier (hash) which will be required on subsequent `chaincode invoke` and `chaincode query` commands in order to identify the deployed chaincode.

With security enabled, modify the command to include the -u parameter passing the username of a logged in user as follows:

```
peer chaincode deploy -u jim -p github.com/hyperledger/fabric/examples/chaincode/go/chainco
-c '{"Function":"init","Args": ["a","100","b","200"]}'
```

Note: If your GOPATH environment variable contains more than one element, the chaincode must be found in the first one or deployment will fail.

Verify Results

To verify that the block containing the latest transaction has been added to the blockchain, use the `/chain` REST endpoint from the command line. Target the IP address of either a validating or a non-validating node. In the example below, 172.17.0.2 is the IP address of a validating or a non-validating node and 7050 is the REST interface port defined in core.yaml.

`curl 172.17.0.2:7050/chain`

An example of the response is below.

```
{
    "height":1,
    "currentBlockHash":"4Yc4yCO95wcpWHW2NLFlf76OGURBBxYZMf3yUyvrEXs5TMai9qNKfy9Yn/=="
}
```

The returned BlockchainInfo message is defined inside fabric.proto.

```
message BlockchainInfo {
    uint64 height = 1;
    bytes currentBlockHash = 2;
    bytes previousBlockHash = 3;
}
```

To verify that a specific block is inside the blockchain, use the `/chain/blocks/{Block}` REST endpoint. Likewise, target the IP address of either a validating or a non-validating node on port 7050.

`curl 172.17.0.2:7050/chain/blocks/0`

The returned Block message structure is defined inside fabric.proto.

```
message Block {
    uint32 version = 1;
    google.protobuf.Timestamp timestamp = 2;
    repeated Transaction transactions = 3;
    bytes stateHash = 4;
    bytes previousBlockHash = 5;
    bytes consensusMetadata = 6;
    NonHashData nonHashData = 7;
}
```

An example of a returned Block structure is below.

```
{
    "transactions":[{
        "type":1,
        "chaincodeID": {
            "path":"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_
↪example02"
        },
        "payload":"ClwIARJYCk9naXRod...",
        "uuid":"abdcec99-ae5e-415e-a8be-1fca8e38ba71"
    }],
    "stateHash":
↪"PY5YcQRu2g1vjiAqHHshoAhnq8CFP3MqzMslcEAJbnmXDtD+LopmkrUHrPMOGSF5UD7Kxqhbg1XUjmQAi84paw==
↪"
}
```

For additional information on the available CLI commands, please see the protocol specification section 6.3 on CLI.

REST API

You can work with the REST API through any tool of your choice. For example, the curl command line utility or a browser based client such as the Firefox Rest Client or Chrome Postman. You can likewise trigger REST requests directly through Swagger. You can utilize the Swagger service directly or, if you prefer, you can set up Swagger locally by following the instructions *here*.

Note: The default REST interface port is `7050`. It can be configured in core.yaml using the `rest.address` property. If using Vagrant, the REST port mapping is defined in Vagrantfile.

Note on constructing a test blockchain If you want to test the REST API locally, construct a test blockchain by running the TestServerOpenchain_API_GetBlockCount test implemented inside api_test.go. This test will create a test blockchain with 5 blocks. Subsequently restart the peer process.

```
cd /opt/gopath/src/github.com/hyperledger/fabric/core/rest
go test -v -run TestServerOpenchain_API_GetBlockCount
```

REST Endpoints

To learn about the REST API through Swagger, please take a look at the Swagger document here. You can upload the service description file to the Swagger service directly or, if you prefer, you can set up Swagger locally by following the instructions *here*.

- *Block*
- GET /chain/blocks/{Block}
- *Blockchain*
- GET /chain
- *Chaincode*
  - POST /chaincode
- *Network*
- GET /network/peers
- *Registrar*
- POST /registrar
- DELETE /registrar/{enrollmentID}
- GET /registrar/{enrollmentID}
- GET /registrar/{enrollmentID}/ecert
- GET /registrar/{enrollmentID}/tcert
- *Transactions*
  - GET /transactions/{UUID}

Block

- **GET /chain/blocks/{Block}**

Use the Block API to retrieve the contents of various blocks from the blockchain. The returned Block message structure is defined inside fabric.proto.

```
message Block {
    uint32 version = 1;
    google.protobuf.Timestamp Timestamp = 2;
    repeated Transaction transactions = 3;
    bytes stateHash = 4;
    bytes previousBlockHash = 5;
}
```

Blockchain

- **GET /chain**

Use the Chain API to retrieve the current state of the blockchain. The returned BlockchainInfo message is defined inside fabric.proto.

```
message BlockchainInfo {
    uint64 height = 1;
    bytes currentBlockHash = 2;
    bytes previousBlockHash = 3;
}
```

Chaincode

- **POST /chaincode**

Use the /chaincode endpoint to deploy, invoke, and query a target chaincode. This service endpoint implements the JSON RPC 2.0 specification with the payload identifying the desired chaincode operation within the `method` field. The supported methods are `deploy` , `invoke` , and `query` .

The /chaincode endpoint implements the JSON RPC 2.0 specification and as such, must have the required fields of `jsonrpc` , `method` , and in our case `params` supplied within the payload. The client should also add the `id` element within the payload if they wish to receive a response to the request. If the `id` element is missing from the request payload, the request is assumed to be a notification and the server will not produce a response.

The following sample payloads may be used to deploy, invoke, and query a sample chaincode. To deploy a chaincode, supply the ChaincodeSpec identifying the chaincode to deploy within the request payload.

Chaincode Deployment Request without security enabled:

```
POST host:port/chaincode

{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": 1,
    "chaincodeID":{
        "path":"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_
↪example02"
    },
    "ctorMsg": {
        "args":["init", "a", "1000", "b", "2000"]
    }
  },
  "id": 1
}
```

To deploy a chaincode with security enabled, supply the `secureContext` element containing the registrationID of a registered and logged in user together with the payload from above.

Chaincode Deployment Request with security enabled (add `secureContext` element):

```
POST host:port/chaincode

{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": 1,
    "chaincodeID":{
        "path":"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_
↪example02"
    },
    "ctorMsg": {
        "args":["init", "a", "1000", "b", "2000"]
    },
    "secureContext": "lukas"
  },
  "id": 1
}
```

The response to a chaincode deployment request will contain a `status` element confirming successful completion of the request. The response to a successful deployment request will likewise contain the generated chaincode hash which must be used in subsequent invocation and query requests sent to this chaincode.

Chaincode Deployment Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message":
↪"52b0d803fc395b5e34d8d4a7cd69fb6aa00099b8fabed83504ac1c5d61a425aca5b3ad3bf96643ea4fdaac132c417c37b
↪"
    },
    "id": 1
}
```

To invoke a chaincode, supply the ChaincodeSpec identifying the chaincode to invoke within the request payload. Note the chaincode `name` field, which is the hash returned from the deployment request.

Chaincode Invocation Request without security enabled:

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":
↪"52b0d803fc395b5e34d8d4a7cd69fb6aa00099b8fabed83504ac1c5d61a425aca5b3ad3bf96643ea4fdaac132c417c37b
↪"
      },
      "ctorMsg": {
         "args":["invoke", "a", "b", "100"]
      }
  },
  "id": 3
}
```

To invoke a chaincode with security enabled, supply the `secureContext` element containing the registrationID of a registered and logged in user together with the payload from above.

Chaincode Invocation Request with security enabled (add `secureContext` element):

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":
↪"52b0d803fc395b5e34d8d4a7cd69fb6aa00099b8fabed83504ac1c5d61a425aca5b3ad3bf96643ea4fdaac132c417c37b
↪"
      },
      "ctorMsg": {
         "args":["invoke", "a", "b", "100"]
      },
      "secureContext": "lukas"
  },
  "id": 3
}
```

The response to a chaincode invocation request will contain a `status` element confirming successful completion of the request. The response will likewise contain the transaction id number for that specific transaction. The client may use the returned transaction id number to check on the status of the transaction after it has been submitted to the system, as the transaction execution is asynchronous.

Chaincode Invocation Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "5a4540e5-902b-422d-a6ab-e70ab36a2e6d"
    },
    "id": 3
}
```

To query a chaincode, supply the ChaincodeSpec identifying the chaincode to query within the request payload. Note the chaincode `name` field, which is the hash returned from the deployment request.

Chaincode Query Request without security enabled:

```
{
  "jsonrpc": "2.0",
  "method": "query",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":
↪"52b0d803fc395b5e34d8d4a7cd69fb6aa00099b8fabed83504ac1c5d61a425aca5b3ad3bf96643ea4fdaac132c417c37b
↪"
      },
      "ctorMsg": {
         "args":["query", "a"]
      }
  },
  "id": 5
}
```

To query a chaincode with security enabled, supply the `secureContext` element containing the registrationID of a registered and logged in user together with the payload from above.

Chaincode Query Request with security enabled (add `secureContext` element):

```
{
  "jsonrpc": "2.0",
  "method": "query",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":
↪"52b0d803fc395b5e34d8d4a7cd69fb6aa00099b8fabed83504ac1c5d61a425aca5b3ad3bf96643ea4fdaac132c417c37b
↪"
      },
      "ctorMsg": {
         "args":["query", "a"]
      },
      "secureContext": "lukas"
  },
  "id": 5
}
```

The response to a chaincode query request will contain a `status` element confirming successful completion of the request. The response will likewise contain an appropriate `message`, as defined by the chaincode. The `message` received depends on the chaincode implementation and may be a string or number indicating the value of a specific chaincode variable.

Chaincode Query Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "-400"
    },
    "id": 5
}
```

Network

- **GET /network/peers**

Use the Network APIs to retrieve information about the network of peer nodes comprising the blockchain network.

The /network/peers endpoint returns a list of all existing network connections for the target peer node. The list includes both validating and non-validating peers. The list of peers is returned as type `` `PeersMessage `` <https://github.com/hyperledger/fabric/blob/v0.6/protos/fabric.proto#L138>`__, containing an array of `` `PeerEndpoint `` <https://github.com/hyperledger/fabric/blob/v0.6/protos/fabric.proto#L127>`__.

```
message PeersMessage {
    repeated PeerEndpoint peers = 1;
}
```

```
message PeerEndpoint {
    PeerID ID = 1;
    string address = 2;
    enum Type {
      UNDEFINED = 0;
      VALIDATOR = 1;
      NON_VALIDATOR = 2;
    }
    Type type = 3;
    bytes pkiID = 4;
}
```

```
message PeerID {
    string name = 1;
}
```

Registrar

- **POST /registrar**
- **DELETE /registrar/{enrollmentID}**
- **GET /registrar/{enrollmentID}**
- **GET /registrar/{enrollmentID}/ecert**
- **GET /registrar/{enrollmentID}/tcert**

Use the Registrar APIs to manage end user registration with the CA. These API endpoints are used to register a user with the CA, determine whether a given user is registered, and to remove any login tokens for a target user preventing them from executing any further transactions. The Registrar APIs are also used to retrieve user enrollment and transaction certificates from the system.

The /registrar endpoint is used to register a user with the CA. The required Secret payload is defined in devops.proto.

```
message Secret {
    string enrollId = 1;
    string enrollSecret = 2;
}
```

The response to the registration request is either a confirmation of successful registration or an error, containing a reason for the failure. An example of a valid Secret message to register user 'lukas' is shown below.

```
{
  "enrollId": "lukas",
  "enrollSecret": "NPKYL39uKbkj"
}
```

The GET /registrar/{enrollmentID} endpoint is used to confirm whether a given user is registered with the CA. If so, a confirmation will be returned. Otherwise, an authorization error will result.

The DELETE /registrar/{enrollmentID} endpoint is used to delete login tokens for a target user. If the login tokens are deleted successfully, a confirmation will be returned. Otherwise, an authorization error will result. No payload is required for this endpoint. Note, that registration with the CA is a one time process for a given user, utilizing a single-use registrationID and registrationPW. If the user registration is deleted through this API, the user will not be able to register with the CA a second time.

The GET /registrar/{enrollmentID}/ecert endpoint is used to retrieve the enrollment certificate of a given user from local storage. If the target user has already registered with the CA, the response will include a URL-encoded version of the enrollment certificate. If the target user has not yet registered, an error will be returned. If the client wishes to use the returned enrollment certificate after retrieval, keep in mind that it must be URL-decoded. This can be accomplished with the QueryUnescape method in the "net/url" package.

The /registrar/{enrollmentID}/tcert endpoint retrieves the transaction certificates for a given user that has registered with the certificate authority. If the user has registered, a confirmation message will be returned containing an array of URL-encoded transaction certificates. Otherwise, an error will result. The desired number of transaction certificates is specified with the optional 'count' query parameter. The default number of returned transaction certificates is 1; and 500 is the maximum number of certificates that can be retrieved with a single request. If the client wishes to use the returned transaction certificates after retrieval, keep in mind that they must be URL-decoded. This can be accomplished with the QueryUnescape method in the "net/url" package.

Transactions

- **GET /transactions/{UUID}**

Use the /transactions/{UUID} endpoint to retrieve an individual transaction matching the UUID from the blockchain. The returned transaction message is defined inside fabric.proto.

```
message Transaction {
    enum Type {
        UNDEFINED = 0;
        CHAINCODE_DEPLOY = 1;
        CHAINCODE_INVOKE = 2;
        CHAINCODE_QUERY = 3;
        CHAINCODE_TERMINATE = 4;
    }
    Type type = 1;
    bytes chaincodeID = 2;
    bytes payload = 3;
    string uuid = 4;
    google.protobuf.Timestamp timestamp = 5;

    ConfidentialityLevel confidentialityLevel = 6;
    bytes nonce = 7;

    bytes cert = 8;
    bytes signature = 9;
}
```

For additional information on the REST endpoints and more detailed examples, please see the protocol specification section 6.2 on the REST API.

To set up Swagger-UI

Swagger is a convenient package that allows you to describe and document your REST API in a single file. The REST API is described in rest_api.json. To interact with the peer node directly through the Swagger-UI, you can upload the available Swagger definition to the Swagger service. Alternatively, you may set up a Swagger installation on your machine by following the instructions below.

1. You can use Node.js to serve up the rest_api.json locally. To do so, make sure you have Node.js installed on your local machine. If it is not installed, please download the Node.js package and install it.
2. Install the Node.js http-server package with the command below:

   `npm install http-server -g`
3. Start up an http-server on your local machine to serve up the rest_api.json.

```
cd /opt/gopath/src/github.com/hyperledger/fabric/core/rest
http-server -a 0.0.0.0 -p 5554 --cors
```

4. Make sure that you are successfully able to access the API description document within your browser at this link:

   `http://localhost:5554/rest_api.json`
5. Download the Swagger-UI package with the following command:

   `git clone https://github.com/swagger-api/swagger-ui.git`
6. Navigate to the /swagger-ui/dist directory and click on the index.html file to bring up the Swagger-UI interface inside your browser.
7. Start up the peer node with no connections to a leader or validator as follows.

```
cd /opt/gopath/src/github.com/hyperledger/fabric
build/bin/peer node start
```

8. If you need to construct a test blockchain on the local peer node, run the the TestServerOpenchain_API_GetBlockCount test implemented inside api_test.go. This test will create a blockchain with 5 blocks. Subsequently restart the peer process.

```
cd /opt/gopath/src/github.com/hyperledger/fabric/core/rest
go test -v -run TestServerOpenchain_API_GetBlockCount
```

9. Go back to the Swagger-UI interface inside your browser and load the API description. You should now be able to issue queries against the pre-built blockchain directly from Swagger.

Node.js Application

You can interface with the peer process from a Node.js application. One way to accomplish that is by relying on the Swagger API description document, rest_api.json and the swagger-js plugin. Another way to accomplish that relies upon the IBM Blockchain JS SDK. Use the approach that you find the most convenient.

Using Swagger JS Plugin

- Demonstrates interfacing with a peer node from a Node.js application.
- Utilizes the Node.js swagger-js plugin: https://github.com/swagger-api/swagger-js

To run:

1. Build and install the fabric core.

```
cd /opt/gopath/src/github.com/hyperledger/fabric
make peer
```

2. Run a local peer node only (not a complete network) with:

   `build/bin/peer node start`

3. Set up a test blockchain data structure (with 5 blocks only) by running a test from within Vagrant as follows. Subsequently restart the peer process.

   ```
   cd /opt/gopath/src/github.com/hyperledger/fabric/core/rest
   go test -v -run TestServerOpenchain_API_GetBlockCount
   ```

4. Start up an http-server on your local machine to serve up the rest_api.json.

   ```
   npm install http-server -g
   cd /opt/gopath/src/github.com/hyperledger/fabric/core/rest
   http-server -a 0.0.0.0 -p 5554 --cors
   ```

5. Download and unzip Sample_1.zip

   ```
   unzip Sample_1.zip -d Sample_1
   cd Sample_1
   ```

6. Update the api_url variable within openchain.js to the appropriate URL if it is not already the default

   `var api_url = 'http://localhost:5554/rest_api.json';`

7. Run the Node.js app

   `node ./openchain.js`

You will observe several responses on the console and the program will appear to hang for a few moments at the end. This is expected, as is it waiting for the invocation transaction to complete in order to then execute a query. You can take a look at the sample output of the program inside the 'openchain_test' file located in the Sample_1 directory.

Marbles Demo Application

- Demonstrates an alternative way of interfacing with a peer node from a Node.js app.
- Demonstrates deploying a Blockchain application as a Bluemix service.

Hold on to your hats everyone, this application is going to demonstrate transferring marbles between two users leveraging IBM Blockchain. We are going to do this in Node.js and a bit of GoLang. The backend of this application will be the GoLang code running in our blockchain network. The chaincode itself will create a marble by storing it to the chaincode state. The chaincode itself is able to store data as a string in a key/value pair setup. Thus we will stringify JSON objects to store more complex structures.

For more inforation on the IBM Blockchain marbles demo, set-up, and instructions, please visit this page.

Commercial Paper Demo Application

- Demonstrates an alternative way of interfacing with a peer node from a Node.js app.
- Demonstrates deploying a Blockchain application as a Bluemix service.

This application is a demonstration of how a commercial paper trading network might be implemented on IBM Blockchain. The components of the demo are:

- An interface for creating new users on the network.
- An interface for creating new commercial papers to trade.

- A Trade Center for buying and selling existing trades.
- A special interface just for auditors of the network to examine trades.

For more inforation on the IBM Blockchain commercial paper demo, set-up, and instructions, please visit this page.

CHAPTER 27

Certificate Authority API

Each of the CA services is split into two GRPC interfaces, namely a public one (indicated by a *P* suffix) and an administrator one (indicated by an *A* suffix).

Enrollment Certificate Authority

The administrator interface of the ECA provides the following functions:

```
service ECAA { // admin
    rpc RegisterUser(RegisterUserReq) returns (Token);
    rpc ReadUserSet(ReadUserSetReq) returns (UserSet);
    rpc RevokeCertificate(ECertRevokeReq) returns (CAStatus); // not yet implemented
    rpc PublishCRL(ECertCRLReq) returns (CAStatus); // not yet implemented
}
```

The `RegisterUser` function allows you to register a new user by specifiying their name and roles in the `RegisterUserReq` structure. If the user has not been registered before, the ECA registers the new user and returns a unique one-time password, which can be used by the user to request their enrollment certificate pair via the public interface of the ECA. Otherwise an error is returned.

The `ReadUserSet` function allows only auditors to retrieve the list of users registered with the blockchain.

The public interface of the ECA provides the following functions:

```
service ECAP { // public
    rpc ReadCACertificate(Empty) returns (Cert);
    rpc CreateCertificatePair(ECertCreateReq) returns (ECertCreateResp);
    rpc ReadCertificatePair(ECertReadReq) returns (CertPair);
    rpc ReadCertificateByHash(Hash) returns (Cert);
    rpc RevokeCertificatePair(ECertRevokeReq) returns (CAStatus); // not yet␣
↪implemented
}
```

The `ReadCACertificate` function returns the certificate of the ECA itself.

The `CreateCertificatePair` function allows a user to create and read their enrollment certificate pair. For this, the user has to do two successive invocations of this function. Firstly, both the signature and encryption public keys have to be handed to the ECA together with the one-time password previously returned by the `RegisterUser` function invocation. The request has to be signed by the user's private signature key to demonstrate that the user is in possession of the private signature key. The ECA in return gives the user a challenge encrypted with the user's public encryption key. The user has to decrypt the challenge, thereby demonstrating that they are in possession of the private encryption key, and then re-issue the certificate creation request - this time with the decrypted challenge instead of the one-time password passed in the invocation. If the challenge has been decrypted correctly, the ECA issues and returns the enrollment certificate pair for the user.

The `ReadCertificatePair` function allows any user of the blockchain to read the certificate pair of any other user of the blockchain.

The `ReadCertificatePairByHash` function allows any user of the blockchain to read a certificate from the ECA matching a given hash.

Transaction Certificate Authority

The administrator interface of the TCA provides the following functions:

```
service TCAA { // admin
    rpc RevokeCertificate(TCertRevokeReq) returns (CAStatus); // not yet implemented
    rpc RevokeCertificateSet(TCertRevokeSetReq) returns (CAStatus); // not yet␣
↪implemented
    rpc PublishCRL(TCertCRLReq) returns (CAStatus); // not yet implemented
}
```

The public interface of the TCA provides the following functions:

```
service TCAP { // public
    rpc ReadCACertificate(Empty) returns (Cert);
    rpc CreateCertificate(TCertCreateReq) returns (TCertCreateResp);
    rpc CreateCertificateSet(TCertCreateSetReq) returns (TCertCreateSetResp);
    rpc RevokeCertificate(TCertRevokeReq) returns (CAStatus); // not yet implemented
    rpc RevokeCertificateSet(TCertRevokeSetReq) returns (CAStatus); // not yet␣
↪implemented
}
```

The `ReadCACertificate` function returns the certificate of the TCA itself.

The `CreateCertificate` function allows a user to create and retrieve a new transaction certificate.

The `CreateCertificateSet` function allows a user to create and retrieve a set of transaction certificates in a single call.

TLS Certificate Authority

The administrator interface of the TLSCA provides the following functions:

```
service TLSCAA { // admin
    rpc RevokeCertificate(TLSCertRevokeReq) returns (CAStatus); not yet implemented
}
```

The public interface of the TLSCA provides the following functions:

```
service TLSCAP { // public
    rpc ReadCACertificate(Empty) returns (Cert);
    rpc CreateCertificate(TLSCertCreateReq) returns (TLSCertCreateResp);
    rpc ReadCertificate(TLSCertReadReq) returns (Cert);
    rpc RevokeCertificate(TLSCertRevokeReq) returns (CAStatus); // not yet implemented
}
```

The `ReadCACertificate` function returns the certificate of the TLSCA itself.

The `CreateCertificate` function allows a user to create and retrieve a new TLS certificate.

The `ReadCertificate` function allows a user to retrieve a previously created TLS certificate.

CHAPTER 28

Attributes usage

Overview

The Attributes feature allows chaincode to make use of extended data in a transaction certificate. These attributes are certified by the Attributes Certificate Authority (ACA) so the chaincode can trust in the authenticity of the attributes' values.

To view complete documentation about attributes design please read *'Attributes support'*

Use case: Authorizable counter

A common use case for the Attributes feature is Attributes Based Access Control (ABAC) which allows specific permissions to be granted to a chaincode invoker based on attribute values carried in the invoker's certificate.

'Authorizable counter' is a simple example of ABAC, in this case only invokers whose "position" attribute has the value 'Software Engineer' will be able to increment the counter. On the other hand any invoker will be able to read the counter value.

In order to implement this example we used 'VerifyAttribyte' function to check the attribute value from the chaincode code.

```
isOk, _ := stub.VerifyAttribute("position", []byte("Software Engineer")) // Here the␣
↪ABAC API is called to verify the attribute, just if the value is verified the␣
↪counter will be incremented.
if isOk {
    // Increment counter code
}
```

The same behavior can be achieved by making use of 'Attribute support' API, in this case an attribute handler must be instantiated.

```
attributesHandler, _ := attr.NewAttributesHandlerImpl(stub)
isOk, _ := attributesHandler.VerifyAttribute("position", []byte("Software Engineer"))
if isOk {
    // Increment counter code
}
```

If attributes are accessed more than once, using `attributeHandler` is more efficient since the handler makes use of a cache to store values and keys.

In order to get the attribute value, in place of just verifying it, the following code can be used:

```
attributesHandler, _ := attr.NewAttributesHandlerImpl(stub)
value, _ := attributesHandler.GetValue("position")
```

Enabling attributes

To make use of this feature the following property has to be set in the membersrvc.yaml file:

- aca.enabled = true

Another way is using environment variables:

```
MEMBERSRVC_CA_ACA_ENABLED=true ./membersrvc
```

Enabling attributes encryption*

In order to make use of attributes encryption the following property has to be set in the membersrvc.yaml file:

- tca.attribute-encryption.enabled = true

Or using environment variables:

```
MEMBERSRVC_CA_ACA_ENABLED=true MEMBERSRVC_CA_TCA_ATTRIBUTE-ENCRYPTION_ENABLED=true ./
↪membersrvc
```

Deploy API making use of attributes

CLI

```
$ ./peer chaincode deploy --help
Deploy the specified chaincode to the network.

Usage:
  peer chaincode deploy [flags]

Global Flags:
  -a, --attributes="[]": User attributes for the chaincode in JSON format
  -c, --ctor="{}": Constructor message for the chaincode in JSON format
  -l, --lang="golang": Language the chaincode is written in
      --logging-level="": Default logging level and overrides, see core.yaml for full␣
↪syntax
  -n, --name="": Name of the chaincode returned by the deploy transaction
  -p, --path="": Path to chaincode
      --test.coverprofile="coverage.cov": Done
  -t, --tid="": Name of a custom ID generation algorithm (hashing and decoding) e.g.␣
↪sha256base64
  -u, --username="": Username for chaincode operations when security is enabled
  -v, --version[=false]: Display current version of fabric peer server
```

To deploy a chaincode with attributes “company” and “position” it should be written in the following way:

```
./peer chaincode deploy -u userName -n mycc -c '{"Function":"init", "Args": []}' -a '[
↪"position", "company"]'
```

REST

```
POST host:port/chaincode

{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": 1,
    "chaincodeID":{
        "name": "mycc"
    },
    "ctorMsg": {
        "function":"init",
        "args":[]
    }
    "attributes": ["position", "company"]
  },
  "id": 1
}
```

Invoke API making use of attributes

CLI

```
$ ./peer chaincode invoke --help
Invoke the specified chaincode.

Usage:
  peer chaincode invoke [flags]

Global Flags:
  -a, --attributes="[]": User attributes for the chaincode in JSON format
  -c, --ctor="{}": Constructor message for the chaincode in JSON format
  -l, --lang="golang": Language the chaincode is written in
      --logging-level="": Default logging level and overrides, see core.yaml for full␣
↪syntax
  -n, --name="": Name of the chaincode returned by the deploy transaction
  -p, --path="": Path to chaincode
      --test.coverprofile="coverage.cov": Done
  -t, --tid="": Name of a custom ID generation algorithm (hashing and decoding) e.g.␣
↪sha256base64
  -u, --username="": Username for chaincode operations when security is enabled
  -v, --version[=false]: Display current version of fabric peer server
```

To invoke “autorizable counter” with attributes “company” and “position” it should be written as follows:

```
./peer chaincode invoke -u userName -n mycc -c '{"Function":"increment", "Args": []}'␣
↪-a '["position", "company"]'
```

REST

```
POST host:port/chaincode
```

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
    "type": 1,
    "chaincodeID":{
        "name": "mycc"
    },
    "ctorMsg": {
        "function":"increment",
        "args":[]
    }
    "attributes": ["position", "company"]
  },
  "id": 1
}
```

Query API making use of attributes

CLI

```
$ ./peer chaincode query --help
Query using the specified chaincode.

Usage:
  peer chaincode query [flags]

Flags:
  -x, --hex[=false]: If true, output the query value byte array in hexadecimal.␣
↪Incompatible with --raw
  -r, --raw[=false]: If true, output the query value as raw bytes, otherwise format␣
↪as a printable string


Global Flags:
  -a, --attributes="[]": User attributes for the chaincode in JSON format
  -c, --ctor="{}": Constructor message for the chaincode in JSON format
  -l, --lang="golang": Language the chaincode is written in
      --logging-level="": Default logging level and overrides, see core.yaml for full␣
↪syntax
  -n, --name="": Name of the chaincode returned by the deploy transaction
  -p, --path="": Path to chaincode
      --test.coverprofile="coverage.cov": Done
  -t, --tid="": Name of a custom ID generation algorithm (hashing and decoding) e.g.␣
↪sha256base64
  -u, --username="": Username for chaincode operations when security is enabled
  -v, --version[=false]: Display current version of fabric peer server
```

To query “autorizable counter” with attributes “company” and “position” it should be written in this way:

```
./peer chaincode query -u userName -n mycc -c '{"Function":"read", "Args": []}' -a '[
↪"position", "company"]'
```

REST

```
POST host:port/chaincode

{
  "jsonrpc": "2.0",
  "method": "query",
  "params": {
    "type": 1,
    "chaincodeID":{
        "name": "mycc"
    },
    "ctorMsg": {
        "function":"read",
        "args":[]
    }
    "attributes": ["position", "company"]
  },
  "id": 1
}
```

- Attributes encryption is not yet available.

CHAPTER 29

NO-OP system chaincode

NO-OP is a system chaincode that does nothing when invoked. The parameters of the invoke transaction are stored on the ledger so it is possible to encode arbitrary data into them.

Functions and valid options

- Invoke transactions have to be called with *'execute'* as function name and at least one argument. Only the **first argument** is used. Note that it should be **encoded with BASE64**.
- Only one type of query is supported: *'getTran'* (passed as a function name). GetTran has to get a transaction ID as argument in hexadecimal format. The function looks up the corresponding transaction's (if any) **first argument** and tries to **decode it as a BASE64 encoded string**.

Testing

NO-OP has unit tests checking invocation and queries using proper/improper arguments. The chaincode implementation provides a facility for mocking the ledger under the chaincode (*mockLedgerH* in struct *chaincode.SystemChaincode*). This should only be used for testing as it is dangerous to rely on global variables in memory that can hold state across invokes.

CHAPTER 30

HYPERLEDGER FABRIC v1.0

Hyperledger fabric is a platform that enables the delivery of a secure, robust, permissioned blockchain for the enterprise that incorporates a byzantine fault tolerant consensus. We have learned much as we progressed through the v0.6-preview release. In particular, that in order to provide for the scalability and confidentiality needs of many use cases, a refactoring of the architecture was needed. The v0.6-preview release will be the final (barring any bug fixes) release based upon the original architecture.

Hyperledger fabric's v1.0 architecture has been designed to address two vital enterprise-grade requirements – **security** and **scalability**. Businesses and organizations can leverage this new architecture to execute confidential transactions on networks with shared or common assets – e.g. supply chain, FOREX market, healthcare, etc. The progression to v1.0 will be incremental, with myriad windows for community members to contribute code and start curating the fabric to fit specific business needs.

WHERE WE ARE:

The current implementation involves every validating peer shouldering the responsibility for the full gauntlet of network functionality. They execute transactions, perform consensus, and maintain the shared ledger. Not only does this configuration lay a huge computational burden on each peer, hindering scalability, but it also constricts important facets of privacy and confidentiality. Namely, there is no mechanism to "channel" or "silo" confidential transactions. Every peer can see the logic for every transaction.

WHERE WE'RE GOING

The new architecture introduces a clear functional separation of peer roles, and allows a transaction to pass through the network in a structured and modularized fashion. The peers are diverged into two distinct roles – Endorser & Committer. As an endorser, the peer will simulate the transaction and ensure that the outcome is both deterministic and stable. As a committer, the peer will validate the integrity of a transaction and then append to the ledger. Now confidential transactions can be sent to specific endorsers and their correlating committers, without the network being made cognizant of the transaction. Additionally, policies can be set to determine what levels of "endorsement" and "validation" are acceptable for a specific class of transactions. A failure to meet these thresholds would simply result in a transaction being withdrawn, rather than imploding or stagnating the entire network. This new model also introduces the possibility for more elaborate networks, such as a foreign exchange market. Entities may need to only participate as endorsers for their transactions, while leaving consensus and commitment (i.e. settlement in this scenario) to a trusted third party such as a clearing house.

The consensus process (i.e. algorithmic computation) is entirely abstracted from the peer. This modularity not only provides a powerful security layer – the consenting nodes are agnostic to the transaction logic – but it also generates a framework where consensus can become pluggable and scalability can truly occur. There is no longer a parallel relationship between the number of peers in a network and the number of consenters. Now networks can grow dynamically (i.e. add endorsers and committers) without having to add corresponding consenters, and exist in a modular infrastructure designed to support high transaction throughput. Moreover, networks now have the capability to completely liberate themselves from the computational and legal burden of consensus by tapping into a pre-existing consensus cloud.

As v1.0 manifests, we will see the foundation for interoperable blockchain networks that have the ability to scale and transact in a manner adherent with regulatory and industry standards. Watch how fabric v1.0 and the Hyperledger Project are building a true blockchain for business -

v1.0 in motion

HOW TO CONTRIBUTE

Use the following links to explore upcoming additions to fabric's codebase that will spawn the capabilities in v1.0:

- Familiarize yourself with the *guidelines for code contributions* to this project. **Note**: In order to participate in the development of the Hyperledger fabric project, you will need an *LF account* This will give you single sign-on to JIRA and Gerrit.
- Explore the design document for the new architecture
- Explore JIRA for open Hyperledger fabric issues.
- Explore the JIRA backlog for upcoming Hyperledger fabric issues.
- Explore JIRA for Hyperledger fabric issues tagged with "help wanted."
- Explore the source code
- Explore the documentation

CHAPTER 31

Contributions Welcome!

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

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

Getting a Linux Foundation account

In order to participate in the development of the Hyperledger Fabric project, you will need an *LF account* You will need to use your LF ID to grant you access to all the Hyperledger community tools, including Gerrit and Jira.

Setting up your SSH key

For Gerrit, you will want to register your public SSH key. Login to Gerrit with your LF account, and click on your name in the upper right-hand corner and then click 'Settings'. In the left-hand margin, you should see a link for 'SSH Public Keys'. Copy-n-paste your public SSH key into the window and press 'Add'.

Getting help

If you are looking for something to work on, or need some expert assistance in debugging a problem or working out a fix to an issue, our community is always eager to help. We hang out on Slack, IRC (#hyperledger on freenode.net) and the mailing lists. Most of us don't bite ;-) and will be glad to help.

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 slack.

Reporting bugs

If you are a user and you find a bug, please submit an issue. 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 and request that it be reviewed.

Fixing issues and working stories

Review the issues list and find something that interests you. You could also check the “help wanted” and “good first bug” lists. It is wise to start with something relatively straight forward and achievable. Usually there will be a comment in the issue that indicates whether someone has already self-assigned the issue. If no one has already taken it, then add a comment assigning the issue to yourself, eg.: `I'll work on this issue.` . Please be considerate and rescind the offer in comments if you cannot finish in a reasonable time, or add a comment saying that you are still actively working the issue if you need a little more time.

Working with a local clone and Gerrit

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

After you have familiarized yourself with `Gerrit` , and maybe played around with the `lf-sandbox` project, you should be ready to set up your local *development environment* We use a Vagrant-based approach to development that simplifies things greatly.

Coding guidelines

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

Becoming a maintainer

This project is managed under open governance model as described in our charter. Projects or sub-projects will be lead by a set of maintainers. New projects can designate an initial set of maintainers that will be approved by the Technical Steering Committee when the project is first approved. The project’s maintainers will, from time-to-time, consider adding or removing a maintainer. An existing maintainer will post a patchset to the *MAINTAINERS* file. If a majority of the maintainers concur in the comments, the pull request is then merged and the individual becomes a (or is removed as a) maintainer. Note that removing a maintainer should not be taken lightly, but occasionally, people do move on - hence the bar should be some period of inactivity, an explicit resignation, some infraction of the code of conduct or consistently demonstrating poor judgement.

Legal stuff

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

We have tried to make it as easy as possible to make contributions. This applies to how we handle the legal aspects of contribution. We use the same approach the Developer’s Certificate of Origin 1.1 (DCO) that the Linux® Kernel community uses to manage code contributions.

We simply ask that when submitting a patch for review, the developer must include a sign-off statement in the commit message.

Here is an example Signed-off-by line, which indicates that the submitter accepts the DCO:

```
Signed-off-by: John Doe <john.doe@hisdomain.com>
```

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

CHAPTER 32

Maintainers

Name	GitHub	Gerrit	email
Binh Nguyen	binhn		binhn@us.ibm.com
Sheehan Anderson	srderson	sheehan	sranderson@gmail.com
Tamas Blummer	tamasblummer		tamas@digitalasset.com
Robert Fajta	rfajta		robert@digitalasset.com
Greg Haskins	ghaskins		ghaskins@lseg.com
Jonathan Levi	JonathanLevi		jonathan@levi.name
Gabor Hosszu	gabre		gabor@digitalasset.com
Simon Schubert	corecode		sis@zurich.ibm.com
Chris Ferris	christo4ferris	ChristopherFerris	chrisfer@us.ibm.com
Srinivasan Muralidharan	muralisrini	muralisr	muralisr@us.ibm.com
Gari Singh	mastersingh24	mastersingh24	gari.r.singh@gmail.com

CHAPTER 33

Setting up the development environment

Overview

The current development environment utilizes Vagrant running an Ubuntu image, which in turn launches Docker containers. Conceptually, the Host launches a VM, which in turn launches Docker containers.

Host -> VM -> Docker

This model allows developers to leverage their favorite OS/editors and execute the system in a controlled environment that is consistent amongst the development team.

- Note that your Host should not run within a VM. If you attempt this, the VM within your Host may fail to boot with a message indicating that VT-x is not available.

Prerequisites

- Git client
- Go - 1.6 or later
- Vagrant - 1.7.4 or later
- VirtualBox - 5.0 or later
- BIOS Enabled Virtualization - Varies based on hardware
- Note: The BIOS Enabled Virtualization may be within the CPU or Security settings of the BIOS

Steps

Set your GOPATH

Make sure you have properly setup your Host's GOPATH environment variable. This allows for both building within the Host and the VM.

Note to Windows users

If you are running Windows, before running any `git clone` commands, run the following command.

```
git config --get core.autocrlf
```

If `core.autocrlf` is set to `true`, you must set it to `false` by running

```
git config --global core.autocrlf false
```

If you continue with `core.autocrlf` set to `true`, the `vagrant up` command will fail with the error `./setup.sh: /bin/bash^M: bad interpreter: No such file or directory`

Cloning the Fabric project

Since the Fabric project is a `Go` project, you'll need to clone the Fabric repo to your $GOPATH/src directory. If your $GOPATH has multiple path components, then you will want to use the first one. There's a little bit of setup needed:

```
cd $GOPATH/src
mkdir -p github.com/hyperledger
cd github.com/hyperledger
```

Recall that we are using `Gerrit` for source control, which has its own internal git repositories. Hence, we will need to *clone from Gerrit* For brevity, the command is as follows:

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

Note: of course, you would want to replace `LFID` with your *Linux Foundation ID*

Boostrapping the VM using Vagrant

Now you're ready to launch Vagrant.

```
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
```

Building the fabric

Once you have your vagrant development environment established, you can proceed to *build and test* the fabric. Once inside the VM, you can find the peer project under `$GOPATH/src/github.com/hyperledger/fabric`. It is also mounted as `/hyperledger`.

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 Microsoft Visual C++ Redistributable installed. You can download the missing package at the following address: http://www.microsoft.com/en-us/download/details.aspx?id=8328

CHAPTER 34

Building the fabric

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

To access your VM, run `vagrant ssh` from within the devenv directory of your locally cloned fabric repository.

```
cd $GOPATH/src/github.com/hyperledger/fabric/devenv
vagrant ssh
```

From within the VM, you can build, run, and test your environment.

```
cd $GOPATH/src/github.com/hyperledger/fabric
make peer
```

To see what commands are available, simply execute the following commands:

```
peer help
```

You should see the following output:

```
Usage:
  peer [command]

Available Commands:
  node        node specific commands.
  network     network specific commands.
  chaincode   chaincode specific commands.
  help        Help about any command

Flags:
  -h, --help[=false]: help for peer
      --logging-level="": Default logging level and overrides, see core.yaml for full
 ↪syntax


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

The `peer node start` command will initiate a peer process, with which one can interact by executing other commands. For example, the `peer node status` command will return the status of the running peer. The full list of commands is the following:

```
node
  start       Starts the node.
  status      Returns status of the node.
  stop        Stops the running node.
```

```
network
  login       Logs in user to CLI.
  list        Lists all network peers.
chaincode
  deploy      Deploy the specified chaincode to the network.
  invoke      Invoke the specified chaincode.
  query       Query using the specified chaincode.
help        Help about any command
```

Note: If your GOPATH environment variable contains more than one element, the chaincode must be found in the first one or deployment will fail.

Running the unit tests

Use the following sequence to run all unit tests

```
cd $GOPATH/src/github.com/hyperledger/fabric
make unit-test
```

To run a specific test use the `-run RE` flag where RE is a regular expression that matches the test case name. To run tests with verbose output use the `-v` flag. For example, to run the `TestGetFoo` test case, change to the directory containing the `foo_test.go` and call/excecute

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

Running Node.js Unit Tests

You must also run the Node.js unit tests to insure that the Node.js client SDK is not broken by your changes. To run the Node.js unit tests, follow the instructions here.

Running Behave BDD Tests

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

```
behave -D logs=Y
```

Note, in order to run behave directly, you must run 'make images' first to build the necessary `peer` and `member services` docker images. These images can also be individually built when `go test` is called with the following parameters:

```
go test github.com/hyperledger/fabric/core/container -run=BuildImage_Peer
go test github.com/hyperledger/fabric/core/container -run=BuildImage_Obcca
```

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.

Prerequisites

- Git client
- Go - 1.6 or later
- RocksDB version 4.1 and its dependencies
- Docker
- Pip
- Set the maximum number of open files to 10000 or greater for your OS

Docker

Make sure that the Docker daemon initialization includes the options

```
-H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock
```

Typically, docker runs as a `service` task, with configuration file at `/etc/default/docker`.

Be aware that the Docker bridge (the `CORE_VM_ENDPOINT`) may not come up at the IP address currently assumed by the test environment (`172.17.0.1`). Use `ifconfig` or `ip addr` to find the docker bridge.

Building RocksDB

```
apt-get install -y libsnappy-dev zlib1g-dev libbz2-dev
cd /tmp
git clone https://github.com/facebook/rocksdb.git
cd rocksdb
git checkout v4.1
PORTABLE=1 make shared_lib
INSTALL_PATH=/usr/local make install-shared
```

`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
```

Building on Z

To make building on Z easier and faster, this script is provided (which is similar to the setup file provided for vagrant). This script has been tested only on RHEL 7.2 and has some assumptions one might want to re-visit (firewall settings, development as root user, etc.). It is however sufficient for development in a personally-assigned VM instance.

To get started, from a freshly installed OS:

```
sudo su
yum install git
mkdir -p $HOME/git/src/github.com/hyperledger
cd $HOME/git/src/github.com/hyperledger
git clone http://gerrit.hyperledger.org/r/fabric
source fabric/devenv/setupRHELonZ.sh
```

From this point, you can proceed as described above for the Vagrant development environment.

```
cd $GOPATH/src/github.com/hyperledger/fabric
make peer unit-test behave
```

Building on Power Platform

Development and build on Power (ppc64le) systems is done outside of vagrant as outlined *here*. For ease of setting up the dev environment on Ubuntu, invoke this script as root. This script has been validated on Ubuntu 16.04 and assumes certain things (like, development system has OS repositories in place, firewall setting etc) and in general can be improvised further.

To get started on Power server installed with Ubuntu, first ensure you have properly setup your Host's GOPATH environment variable. Then, execute the following commands to build the fabric code:

```
mkdir -p $GOPATH/src/github.com/hyperledger
cd $GOPATH/src/github.com/hyperledger
git clone http://gerrit.hyperledger.org/r/fabric
sudo ./fabric/devenv/setupUbuntuOnPPC64le.sh
cd $GOPATH/src/github.com/hyperledger/fabric
make dist-clean all
```

Building natively on OSX

First, install Docker, as described here. The database by default writes to /var/hyperledger. You can override this in the `core.yaml` configuration file, under `peer.fileSystemPath`.

```
brew install go rocksdb snappy gnu-tar     # For RocksDB version 4.1, you can compile␣
↪your own, as described earlier

# You will need the following two for every shell you want to use
eval $(docker-machine env)
export PATH="/usr/local/opt/gnu-tar/libexec/gnubin:$PATH"

cd $GOPATH/src/github.com/hyperledger/fabric
make peer
```

CHAPTER 36

Configuration

Configuration utilizes the viper and cobra libraries.

There is a **core.yaml** file that contains the configuration for the peer process. Many of the configuration settings can be overridden on the command line by setting ENV variables that match the configuration setting, but by prefixing with *'CORE_'*. For example, logging level manipulation through the environment is shown below:

```
CORE_PEER_LOGGING_LEVEL=CRITICAL peer
```

CHAPTER 37

Logging

Logging utilizes the go-logging library.

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

See specific logging control instructions when running the peer process.

CHAPTER 38

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.

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:

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`.

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 39

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 Slack if you need help.

Gerrit assigns the following roles to users:

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

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

Git-review

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

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.

Getting deeper into Gerrit

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

Working with a local clone of the repository

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

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

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

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

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

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

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

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

Then input precise and readable commit msg and submit.

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

Submitting a Change

Currently, Gerrit is the only method to submit a change for review. Please review the *guidelines* for making and submitting a change.

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.

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.

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.

Reviewing Using Gerrit

- **Add**: This button allows the change submitter to manually add names of people who should review a change; start typing a name and the system will auto-complete based on the list of people registered and with access to the system. They will be notified by email that you are requesting their input.

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

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

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

Viewing Pending Changes

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

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

Add the filter *project:fabric* to limit the visible changes to only those from 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 40

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 41

Submitting a Change to Gerrit

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

Change Requirements

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

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

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

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

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

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

Commits must *build cleanly* when applied 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 42

Gerrit Recommended Practices

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

Browsing the Git Tree

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

Watching a Project

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

Commit Messages

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

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

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

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

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

Recommended reading: How to Write a Git Commit Message

Avoid Pushing Untested Work to a Gerrit Server

To avoid pushing untested work to Gerrit.

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

Keeping Track of Changes

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

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

Topic branches

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

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

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

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

Creating a Cover Letter for a Topic

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

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

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

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

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

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

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

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

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

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

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

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

Finding Available Topics

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

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

Downloading or Checking Out a Change

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

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

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

```
git review -d CHANGEID
```

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

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

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

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

Using Draft Branches

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

The next command ensures a local branch is created:

```
git checkout -b BRANCHNAME
```

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

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

Using Sandbox Branches

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

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

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

Usually, the process to create content is:

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

The next command pushes forcibly without review:

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

You can also push forcibly with review:

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

Updating the Version of a Change

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

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

```
git log REMOTE/master..master

c0
...
c7

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

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

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

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

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

Rebasing

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

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

Rebasing During a Pull

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

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

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

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

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

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

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

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

```
c1
...
c7
```

Getting Better Logs from Git

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

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

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

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

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

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

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

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

CHAPTER 43

Coding guidelines

Coding Golang

We code in Go™ and strictly follow the best practices and will not accept any deviations. You must run the following tools against your Go code and fix all errors and warnings: - golint - go vet - goimports

Generating gRPC code

If you modify any `.proto` files, run the following command to generate/update the respective `.pb.go` files.

```
cd $GOPATH/src/github.com/hyperledger/fabric
make protos
```

Adding or updating Go packages

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
```

Chaincode (Smart Contracts and Digital Assets)

17. Does the fabric implementation support smart contract logic?

A. 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.

17. How do I create a business contract using the fabric?

A. There are generally two ways to develop business contracts: the first way is to code individual contracts into standalone instances of chaincode; the second way, and probably the more efficient way, is to use chaincode to create decentralized applications that manage the life cycle of one or multiple types of business contracts, and let end users instantiate instances of contracts within these applications.

17. How do I create assets using the fabric?

A. Users can use chaincode (for business rules) and membership service (for digital tokens) to design assets, as well as the logic that manages them.

There are two popular approaches to defining assets in most blockchain solutions: the stateless UTXO model, where account balances are encoded into past transaction records; and the account model, where account balances are kept in state storage space on the ledger.

Each approach carries its own benefits and drawbacks. This blockchain 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.

17. Which languages are supported for writing chaincode?

A. 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.

17. Does the fabric have native currency?

A. No. However, if you really need a native currency for your chain network, you can develop your own native currency with chaincode. One common attribute of native currency is that some amount will get transacted (the chaincode defining that currency will get called) every time a transaction is processed on its chain.

CHAPTER 45

Confidentiality

17. How is the confidentiality of transactions and business logic achieved?

A. 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.

Q. What if none of the stakeholders of a business contract are validators?

A. 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 46

Consensus Algorithm

17. Which Consensus Algorithm is used in the fabric?

A. 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 two consensus implementations for users to select from: 1) No-op (consensus ignored); and 2) Batch PBFT.

CHAPTER 47

Identity Management (Membership Service)

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

A. 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.

Q. Does its Membership Service make the fabric a centralized solution?

A. 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 48

Usage

17. What are the expected performance figures for the fabric?

A. 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.

Q. Do I have to own a validating node to transact on a chain network?

A. 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.

Q. What does the error string “state may be inconsistent, cannot query” as a query result mean?

A. 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.

Hyperledger Fabric - Application Access Control Lists

Overview

We consider the following entities:

1. *HelloWorld*: is a chaincode that contains a single function called *hello*;
2. *Alice*: is the *HelloWorld* deployer;
3. *Bob*: is the *HelloWorld*‘s functions invoker.

Alice wants to ensure that only Bob can invoke the function *hello*.

Fabric Support

To allow Alice to specify her own access control lists and Bob to gain access, the fabric layer gives access to following capabilities:

1. Alice and Bob can sign and verify any message with specific transaction certificates or enrollment certificate they own;
2. The fabric allows to *name* each transaction by means of a unique *binding* to be used to bind application data to the underlying transaction transporting it;
3. Extended transaction format.

The fabric layer exposes the following interfaces and functions to allow the application layer to define its own ACLS.

Certificate Handler

The following interface allows to sign and verify any message using signing key-pair underlying the associated certificate. The certificate can be a TCert or an ECert.

```
// CertificateHandler exposes methods to deal with an ECert/TCert
type CertificateHandler interface {

    // GetCertificate returns the certificate's DER
    GetCertificate() []byte

    // Sign signs msg using the signing key corresponding to the certificate
    Sign(msg []byte) ([]byte, error)
    // Verify verifies msg using the verifying key corresponding to the certificate
    Verify(signature []byte, msg []byte) error

    // GetTransactionHandler returns a new transaction handler relative to this␣
↪certificate
    GetTransactionHandler() (TransactionHandler, error)
}
```

Transaction Handler

The following interface allows to create transactions and give access to the underlying *binding* that can be leveraged to link application data to the underlying transaction.

```
// TransactionHandler represents a single transaction that can be named by the output␣
↪of the GetBinding method.
// This transaction is linked to a single Certificate (TCert or ECert).
type TransactionHandler interface {

    // GetCertificateHandler returns the certificate handler relative to the␣
↪certificate mapped to this transaction
    GetCertificateHandler() (CertificateHandler, error)

    // GetBinding returns a binding to the underlying transaction
    GetBinding() ([]byte, error)

    // NewChaincodeDeployTransaction is used to deploy chaincode
    NewChaincodeDeployTransaction(chaincodeDeploymentSpec *obc.
↪ChaincodeDeploymentSpec, uuid string) (*obc.Transaction, error)

    // NewChaincodeExecute is used to execute chaincode's functions
    NewChaincodeExecute(chaincodeInvocation *obc.ChaincodeInvocationSpec, uuid␣
↪string) (*obc.Transaction, error)

    // NewChaincodeQuery is used to query chaincode's functions
    NewChaincodeQuery(chaincodeInvocation *obc.ChaincodeInvocationSpec, uuid string)␣
↪(*obc.Transaction, error)
}
```

Client

The following interface offers a mean to get instances of the previous interfaces.

```
type Client interface {

    ...

    // GetEnrollmentCertHandler returns a CertificateHandler whose certificate is the␣
↪enrollment certificate
    GetEnrollmentCertificateHandler() (CertificateHandler, error)

    // GetTCertHandlerNext returns a CertificateHandler whose certificate is the next␣
↪available TCert
    GetTCertificateHandlerNext() (CertificateHandler, error)

    // GetTCertHandlerFromDER returns a CertificateHandler whose certificate is the␣
↪one passed
    GetTCertificateHandlerFromDER(der []byte) (CertificateHandler, error)

}
```

Transaction Format

To support application-level ACLs, the fabric's transaction and chaincode specification format have an additional field to store application-specific metadata. The content of this field is decided by the application. The fabric layer treats it as an unstructured stream of bytes.

```
message ChaincodeSpec {

    ...

    ConfidentialityLevel confidentialityLevel;
    bytes metadata;

    ...
}


message Transaction {
    ...

    bytes payload;
    bytes metadata;

    ...
}
```

Another way to achieve this is to have the payload contain the metadata itself.

Validators

To assist chaincode execution, the validators provide the chaincode additional information, such as the metadata and the binding.

Application-level access control

Deploy Transaction

Alice has full control over the deployment transaction's metadata. In particular, the metadata can be used to store a list of ACLs (one per function), or a list of roles. To define each of these lists/roles, Alice can use any TCerts/ECerts of the users who have been granted that (access control) privilege or have been assigned that role. The latter is done offline.

Now, Alice requires that in order to invoke the *hello* function, a certain message *M* has to be authenticated by an authorized invoker (Bob, in our case). We distinguish the following two cases:

1. *M* is one of the chaincode's function arguments;

2. *M* is the invocation message itself, i.e., function-name, arguments.

Execute Transaction

To invoke *hello*, Bob needs to sign *M* using the TCert/ECert Alice has used to name him in the deployment transaction's metadata. Let's call this certificate CertBob. At this point Bob does the following:

1. Bob obtains a *CertificateHandler* for CertBob, *cHandlerBob*;
2. Bob obtains a new *TransactionHandler* to issue the execute transaction, *txHandler* relative to his next available TCert or his ECert;
3. Bob obtains *txHandler*'s *binding* by invoking *txHandler.getBinding()*;
4. Bob signs *'M || txBinding'* by invoking *cHandlerBob.Sign('M || txBinding')*, let *signature* be the output of the signing function;
5. Bob issues a new execute transaction by invoking, *txHandler.NewChaincodeExecute(...)*. Now, *signature* can be included in the transaction as one of the argument to be passed to the function or as transaction metadata.

Chaincode Execution

The validators, who receive the execute transaction issued by Bob, will provide to *hello* the following information:

1. The *binding* of the execute transaction;
2. The *metadata* of the execute transaction;
3. The *metadata* of the deploy transaction.

Then, *hello* is responsible for checking that *signature* is indeed a valid signature issued by Bob.

CHAPTER 50

Attributes support

To support attributes the user has to pass them during TCert creation, these attributes can be used during transaction deployment, execution or query for Attribute Based Access Control (ABAC) to determine whether the user can or cannot execute a specific chaincode or used attributes' values for other purposes. A mechanism to validate the ownership of attributes is required in order to prove if the attributes passed by the user are correct. The Attribute Certificate Authority (ACA) has the responsibility of validate attributes and to return an Attribute Certificate (ACert) with the valid attribute values. Attributes values are encrypted using the keys defined below (section Attributes keys).

Attribute Keys

The attributes are encrypted using a key derived from a hierarchy called PreKey tree. This approach consists in deriving keys from a parent key, allowing the parent key owner, get access to derived keys. This way keys used to encrypt attributes are different among attributes and TCerts avoiding linkability while allowing an authorized auditor who owns a parent key to derive the keys in the lower levels.

Example of prekey tree

```
Pre3K_BI
        |_Pre2K_B = HMAC(Pre3K_BI, “banks”)
        |    |_Pre1K_BankA = HMAC(Pre2K_B, “Bank A”)
        |    |    |_Pre0K_BankA = HMAC(Pre1K_BankA, TCertID)
        |    |        |_PositionKey_BankA_TIdx = HMAC(Pre0K_BankA, "position")
        |    |        |_CompanyKey_BankA_TIdx = HMAC(Pre0K_BankA, "company")
        |    |
        |    |_Pre1K_BankB = HMAC(Pre2K_B, “BanKB”)
        |        |_Pre0K_BankB = HMAC(Pre1K_BankB, TCertID)
        |             |_PositionKey_BankB_TIdx = HMAC(Pre0K_BankB, "position")
        |             |_CompanyKey_BankB_TIdx = HMAC(Pre0K_BankB, "company")
        |
        |_Pre2K_I = HMAC(Pre3K_BI, "institutions")
            |_Pre1K_InstitutionA= HMAC(Pre2K_I, "Institution A”)
               |_Pre0K_InstitutionA = HMAC(_Pre1K_InstitutionA, TCertID)
                    |_PositionKey_InstA_TIdx = HMAC(Pre0K_InstitutionA, "position")
                    |_CompanyKey_InstA_TIdx = HMAC(Pre0K_InstitutionA, "company")
```

- Pre3K_BI: is available to TCA and auditors for banks and institutions.
- Pre2K_B: is available to auditors for banks
- Pre1K_BankA: is available to auditors for Bank A.

- Pre1K_BankB: is available to auditors for Bank B.
- Pre2K_I: is available to auditors for institutions.
- Pre1K_InstitutionA: is available to auditors for Institution A.

Each TCert has a different PreK0 (for example Pre0K_BankA) and each TCert attribute has a different attribute key (for example PositionKey_BankA_TIdx).

Attribute Certificate Authority

Attribute Certificate Authority (ACA) has the responsibility of certify the ownership of the attributes. ACA has a database to hold attributes for each user and affiliation.

1. id: The id passed by the user during enrollment
2. affiliation: The entity which the user is affiliated to
3. attributeName: The name used to look for the attribute, e.g. 'position'
4. attributeValue: The value of the attribute, e.g. 'software engineer'
5. validFrom: The start of the attribute's validity period
6. validTo: The end of the attribute's validity period

gRPC ACA API

1. FetchAttributes

```
rpc FetchAttributes(ACAFetchAttrReq) returns (ACAFetchAttrResp);

message ACAFetchAttrReq {
    google.protobuf.Timestamp ts = 1;
    Cert eCert = 2;                     // ECert of involved user.
    Signature signature = 3;            // Signed using the ECA private key.
}

message ACAFetchAttrResp {
    enum StatusCode {
        SUCCESS = 000;
        FAILURE = 100;
    }
    StatusCode status = 1;
}
```

2. RequestAttributes

```
rpc RequestAttributes(ACAAttrReq) returns (ACAAttrResp);

message ACAAttrReq {
    google.protobuf.Timestamp ts = 1;
    Identity id = 2;
    Cert eCert = 3;                                  // ECert of involved user.
    repeated TCertAttributeHash attributes = 4;     // Pairs attribute-key, attribute-
↪value-hash
    Signature signature = 5;                         // Signed using the TCA private␣
↪key.
}

message ACAAttrResp {
    enum StatusCode {
        FULL_SUCCESSFUL     = 000;
        PARTIAL_SUCCESSFUL  = 001;
        NO_ATTRIBUTES_FOUND = 010;
        FAILURE             = 100;
    }
    StatusCode status = 1;
    Cert cert = 2;                  // ACert with the owned attributes.
    Signature signature = 3;        // Signed using the ACA private key.
}
```

3. RefreshAttributes

```
rpc RefreshAttributes(ACARefreshReq) returns (ACARefreshResp);

message ACARefreshAttrReq {
    google.protobuf.Timestamp ts = 1;
    Cert eCert = 2;                             // ECert of the involved user.
    Signature signature = 3;                    // Signed using enrollPrivKey
}

message ACARefreshAttrResp {
    enum StatusCode {
        SUCCESS = 000;
        FAILURE = 100;
    }
    StatusCode status = 1;
}
```

FLOW

During enrollment

1. The user requests an Enrollment Certificate (ECert) to ECA
2. ECA creates the ECert and responds to the user with it.
3. ECA issues a fetch request under TLS to the ACA passing the newly generated ECert as a parameter. This request is signed with the ECA's private key.
4. The request triggers ACA asynchronous mechanism that fetches attributes' values from external sources and populates the attributes database (in the current implementation attributes are loaded from an internal configuration file).

During TCert generation

1. When the user needs TCerts to create a new transaction it requests a batch of TCerts to the TCA, and provides the following:

- The batch size (i.e. how many TCerts the user is expecting)
- Its ECert

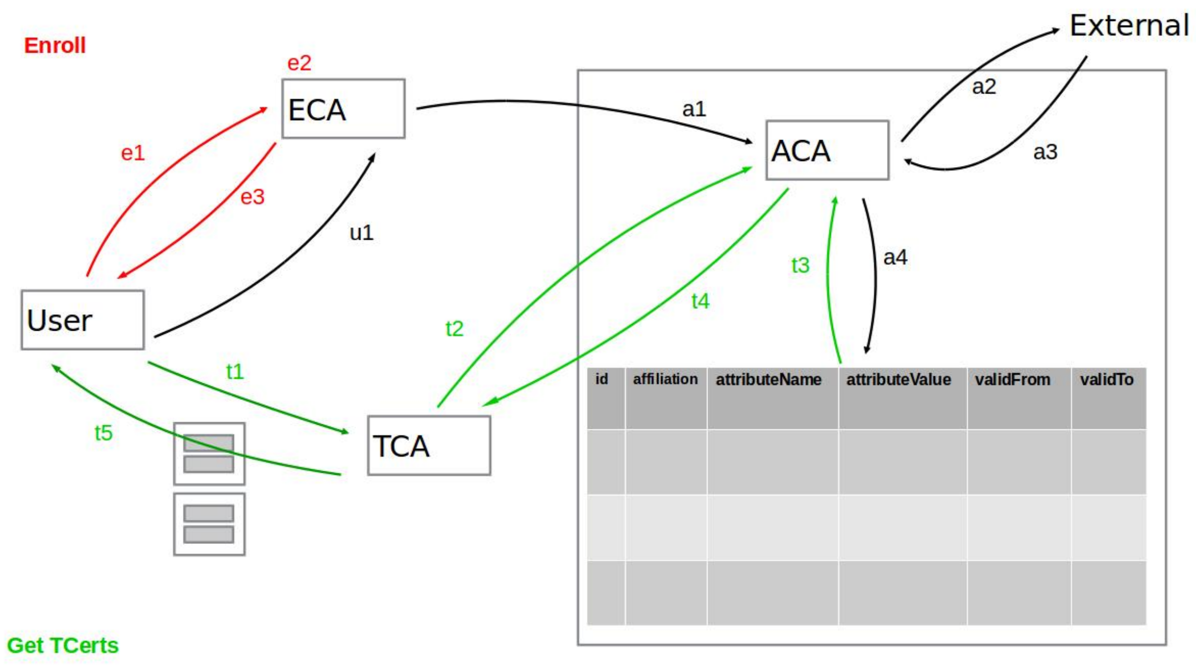


Fig. 50.1: ACA flow

- A list of attributes (e.g. Company, Position)

2. Under TLS TCA sends a RequestAttributes() to ACA to verify if the user is in possession of those attributes. This request is signed with TCA's private key and it contains:

- User's ECert
- A list of attribute names "company, position, ..."

3. The ACA performs a query to the internal attributes database and there are three possible scenarios***:
   (a) The user does not have any of the specified attributes – An error is returned.
   (b) The user has all the specified attributes – An X.509 certificate (ACert) with all the specified attributes and the ECert public key is returned.
   (c) The user has a subset of the requested attributes – An X.509 certificate (ACert) with just the subset of the specified attributes and the ECert public key is returned.
4. The TCA checks the validity period of the ACert's attributes and updates the list by eliminating those that are expired. Then for scenarios b and c from the previous item it checks how many (and which ones) of the attributes the user will actually receive inside each TCert. This information needs to be returned to the user in order to decide whether the TCerts are useful or if further actions needs to be performed (i.e. issue a RefreshAttributes command and request a new batch, throw an error or make use of the TCerts as they are).
5. The TCA could have other criteria to update the valid list of attributes.
6. The TCA creates the batch of TCerts. Each TCert contains the valid attributes encrypted with keys derived from the Prekey tree (each key is unique per attribute, per TCert and per user).
7. The TCA returns the batch of TCerts to the user along with a root key (Prek0) from which each attribute encryption key was derived. There is a Prek0 per TCert. All the TCerts in the batch have the same attributes and the validity period of the TCerts is the same for the entire batch.

*** *In the current implementation an attributes refresh is executed automatically before this step, but once the refresh service is implemented the user will have the responsibility of keeping his/her attributes updated by invoking this method.*

Assumptions

1. An Attribute Certificate Authority (ACA) has been incorporated to the Membership Services internally to provide a trusted source for attribute values.
2. In the current implementation attributes are loaded from a configuration file (membersrvc.yml).
3. Refresh attributes service is not implemented yet, instead, attributes are refreshed in each RequestAttribute invocation.

CHAPTER 51

Setting up the Full Hyperledger fabric Developer's Environment

- See *Setting Up The Development Environment* to set up your development environment.
- The following commands are all issued from the vagrant environment. The following will open a terminal session:

```
cd <your cloned location>/fabric/devenv
vagrant up
vagrant ssh
```

- Issue the following commands to build the Hyperledger fabric client (HFC) Node.js SDK including the API reference documentation

```
cd /opt/gopath/src/github.com/hyperledger/fabric/sdk/node
make all
```

- Issue the following command where your Node.js application is located if you wish to use the `require("hfc")`, this will install the HFC locally.

```
npm install /opt/gopath/src/github.com/hyperledger/fabric/sdk/node
```

Or point to the HFC directly by using the following `require()` in your code:

```
require("/opt/gopath/src/github.com/hyperledger/fabric/sdk/node");
```

- To build the API reference documentation:

```
cd /opt/gopath/src/github.com/hyperledger/fabric/sdk/node
make doc
```

- To build the reference documentation in the *Fabric-starter-kit*

```
docker exec -it nodesdk /bin/bash
cd /opt/gopath/src/github.com/hyperledger/fabric/sdk/node
make doc
```

- The the API reference documentation will be available in: `/opt/gopath/src/github.com/hyperledger/fabric/sd`

CHAPTER 52

Application Overview

Hyperledger fabric supports two types of applications:

- A standalone application that interacts directly with a blockchain. See the *Standalone Application* section.
- A web application that interacts with a blockchain on behalf of its web application users. See the *Web Application* section.

  ## Standalone Application

The following diagram provides an overview of the major components of Hyperledger fabric for the standalone application developer.

In the diagram above, the blue boxes are Hyperledger fabric components and the green boxes are application developer components. Each outer box represents a separate process.

The **Standalone Application** may be developed in Node.js by using the Hyperledger fabric Client (HFC) SDK for Node.js. This SDK handles all interactions with other Hyperledger fabric components.

#### SDK interactions

The **SDK** interacts with a **Peer** process. If the Peer process fails, the Node.js Client SDK can fail-over to another Peer as shown by the dotted line from the Node.js Client SDK to another Peer.

This interaction with the Peer consists of submitting transactions to the blockchain. There are three types of transactions:

- deploy - to deploy developer's chaincode as depicted by the green **CC1** boxes in the diagram;
- invoke - to execute a chaincode function which changes the state of the *blockchain database*;
- query - to execute a chaincode function which may return state information related to the *blockchain database*.

The **SDK** also interacts with a **Membership Services** process. In fact, if security is enabled in the Peer (strongly recommended), the Node.js client SDK must interact with Membership Services first in order to retrieve credential certificates which are then used to interact with the Peer.

The interaction with the Membership Services consists of:

- register - to invite a new user to the blockchain
- enroll - to get access to the blockchain

  ## Web Application

The following diagram provides an overview of the major components of Hyperledger fabric for the web application developer.

At a high-level, you can think of the blockchain as a database with which the web application interacts; therefore, it is similar to the following topology.

```
browser --> web tier --> database tier
```

In the diagram above, the blue boxes are Hyperledger fabric components and the green boxes are application developer components. Each outer box represents a separate process.

The browser interacts with the developer's Node.js web application using the Hyperledger fabric's Node.js client SDK. The SDK handles all interactions with other Hyperledger fabric components as described in the *SDK interactions* section of *Standalone Application*.

CHAPTER 53

Hyperledger fabric Client (HFC) SDK for Node.js

The Hyperledger fabric Client (HFC) SDK for Node.js provides a powerful and easy to use API to interact with a Hyperledger fabric blockchain. The HFC is designed to be used in the Node.js JavaScript runtime.

Overview and Deep Dive

- *Application Developer's Overview* for a topological overview of applications and a blockchain.
- *Hyperledger fabric Client (HFC) SDK for Node.js* the Node.js client SDK in more depth

Development Environment Choices

- *Recommended: Fabric-starter-kit* uses pre-built docker images for the Node.js client application interacting with Hyperledger fabric blockchain. This environment may suffice for a majority of Node.js application developers. The environment contains a built-in standalone sample ready to go.
- *Full Hyperledger fabric development environment* on how to set up an environment for developing chaincode and applications.

Note: Only recommended for development of the Hyperledger fabric Client SDK itself.

Sample Code

- *Node.js Standalone Application in Vagrant* for a sample standalone Node.js application running in the full development environment within Vagrant.
- *Node.js Web Application* for a sample web application and to see how to use the Node.js client SDK for a sample web app leveraging the client SDK to interact with a blockchain network.

Related information

- To build the reference documentation for the Node.js client SDK APIs follow the instructions *here*
- To learn more about chaincode, see *Writing, Building, and Running Chaincode in a Development Environment*

Hyperledger fabric Client (HFC) SDK for Node.js

The Hyperledger fabric Client (HFC) SDK provides a powerful and easy to use API to interact with a Hyperledger fabric blockchain.

Below, you'll find the following sections:

Installing only the SDK

If you are an experienced node.js developer and already have a blockchain environment set up and running elsewhere, you can set up a client-only environment to run the node.js client by installing the HFC node module as shown below. This assumes a minimum of npm version 2.11.3 and node.js version 0.12.7 are already installed.

- To install the latest HFC module of Hyperledger fabric

```
npm install hfc
```

Terminology

In order to transact on a Hyperledger fabric blockchain, you must first have an identity which has been both **registered** and **enrolled** with Membership Services. For a topological overview of how the components interact, see *Application Developer's Overview*

Think of **registration** as *issuing a user invitation* to join a blockchain. It consists of adding a new user name (also called an *enrollment ID*) to the membership service configuration. This can be done programatically with the `Member.register` method, or by adding the enrollment ID directly to the membersrvc.yaml configuration file.

Think of **enrollment** as *accepting a user invitation* to join a blockchain. This is always done by the entity that will transact on the blockchain. This can be done programatically via the `Member.enroll` method.

HFC Objects

HFC is written primarily in typescript. The source can be found in the `fabric/sdk/node/src` directory. The reference documentation is generated automatically from this source code and can be found in`fabric/sdk/node/doc` after building the project.

The following is a high-level description of the HFC objects (classes and interfaces) to help guide you through the object hierarchy.

- **Chain**

This is the main top-level class which is the client's representation of a chain. HFC allows you to interact with multiple chains and to share a single `KeyValStore` and `MemberServices` object with multiple chains if needed. For each chain, you add one or more `Peer` objects which represents the endpoint(s) to which HFC connects to transact on the chain. The second peer is used only if the first peer fails. The third peer is used only if both the first and second peers fail, etc.

- **KeyValStore**

This is a very simple interface which HFC uses to store and retrieve all persistent data. This data includes private keys, so it is very important to keep this storage secure. The default implementation is a simple file-based version found in the `FileKeyValStore` class. If running in a clustered web application, you will need to either ensure that a shared file system is used, or you must implement your own `KeyValStore` that can be shared among all cluster members.

- **MemberServices**

This is an interface representing Membership Services and is implemented by the `MemberServicesImpl` class. It provides security and identity related features such as privacy, unlinkability, and confidentiality. This implementation issues *ECerts* (enrollment certificates) and *TCerts* (transaction certificates). ECerts are for enrollment identity and TCerts are for transactions.

- **Member** or **User**

The Member class most often represents an end User who transacts on the chain, but it may also represent other types of members such as peers. From the Member class, you can *register* and *enroll* members or users. This interacts with the `MemberServices` object. You can also deploy, query, and invoke chaincode directly, which interacts with the `Peer`. The implementation for deploy, query and invoke simply creates a temporary `TransactionContext` object and delegates the work to it.

- **TransactionContext**

This class implements the bulk of the deploy, invoke, and query logic. It interacts with Membership Services to get a TCert to perform these operations. Note that there is a one-to-one relationship between TCert and TransactionContext. In other words, a single TransactionContext will always use the same TCert. If you want to issue multiple transactions with the same TCert, then you can get a `TransactionContext` object from a `Member` object directly and issue multiple deploy, invoke, or query operations on it. Note however that if you do this, these transactions are linkable, which means someone could tell that they came from the same user, though not know which user. For this reason, you will typically just call deploy, invoke, and query on the Member or User object.

Pluggability

HFC was designed to support two pluggable components:

1. Pluggable `KeyValStore` key value store which is used to retrieve and store keys associated with a member. The key value store is used to store sensitive private keys, so care must be taken to properly protect access.

IMPORTANT NOTE: The default KeyValStore is file-based. If multiple instances of a web application run in a cluster, you must provide an implementation of the KeyValStore which is used by all members of the cluster.

2. Pluggable `MemberServices` which is used to register and enroll members. Member services enables hyperledger to be a permissioned blockchain, providing security services such as anonymity, unlinkability of transactions, and confidentiality

Chaincode Deployment

'net' mode

To have the chaincode deployment succeed in network mode, you must properly set up the chaincode project outside of your Hyperledger fabric source tree to include all the **golang** dependencies such that when tarred up and sent to the peer, the peer will be able to build the chain code and then deploy it. The following instructions will demonstrate how to properly set up the directory structure to deploy *chaincode_example02* in network mode.

The chaincode project must be placed under the `$GOPATH/src` directory. For example, the chaincode_example02 project should be placed under `$GOPATH/src/` as shown below.

```
mkdir -p $GOPATH/src/github.com/chaincode_example02/
cd $GOPATH/src/github.com/chaincode_example02
curl GET https://raw.githubusercontent.com/hyperledger/fabric/v0.6/examples/chaincode/
↪go/chaincode_example02/chaincode_example02.go > chaincode_example02.go
```

After you have placed your chaincode project under the `$GOPATH/src`, you will need to vendor the dependencies. From the directory containing your chaincode source, run the following commands:

```
go get -u github.com/kardianos/govendor
cd $GOPATH/src/github.com/chaincode_example02
govendor init
govendor fetch github.com/hyperledger/fabric
```

Now, execute `go build` to verify that all of the chaincode dependencies are present.

```
go build
```

'dev' mode

For deploying chaincode in development mode see *Writing, Building, and Running Chaincode in a Development Environment* The chaincode must be running and connected to the peer before issuing the `deploy()` from the Node.js application. The hfc `chain` object must be set to dev mode.

```
chain.setDevMode(true);
```

The deploy request must include the `chaincodeName` that the chaincode registered with the peer. The built-in chaincode example checks an environment variable `CORE_CHAINCODE_ID_NAME=mycc` when it starts.

```
var deployRequest = {
    chaincodeName: 'mycc',
    fcn: "init",
    args: ["a", "100", "b", "200"]
};
```

Enabling TLS

If you wish to configure TLS with the Membership Services server, the following steps are required:

- Modify `$GOPATH/src/github.com/hyperledger/fabric/membersrvc/membersrvc.yaml` as follows:

```
server:
    tls:
       cert:
          file: "/var/hyperledger/production/.membersrvc/tlsca.cert"
       key:
          file: "/var/hyperledger/production/.membersrvc/tlsca.priv"
```

To specify to the Membership Services (TLS) Certificate Authority (TLSCA) what X.509 v3 Certificate (with a corresponding Private Key) to use:

- Modify `$GOPATH/src/github.com/hyperledger/fabric/peer/core.yaml` as follows:

```
peer:
   pki:
       tls:
          enabled: true
          rootcert:
              file: "/var/hyperledger/production/.membersrvc/tlsca.cert"
```

To configure the peer to connect to the Membership Services server over TLS (otherwise, the connection will fail).

- Bootstrap your Membership Services and the peer. This is needed in order to have the file *tlsca.cert* generated by the member services.
- Copy `/var/hyperledger/production/.membersrvc/tlsca.cert` to `$GOPATH/src/github.com/hyperledger/fabric/sdk/node`.

Note: If you cleanup the folder `/var/hyperledger/production` then don't forget to copy again the *tlsca.cert* file as described above.

Troubleshooting

If you see errors stating that the client has already been registered/enrolled, keep in mind that you can perform the enrollment process only once, as the enrollmentSecret is a one-time-use password. You will see these errors if you have performed a user registration/enrollment and subsequently deleted the cryptographic tokens stored on the client side. The next time you try to enroll, errors similar to the ones below will be seen.

```
Error: identity or token do not match
```

```
Error: user is already registered
```

To address this, remove any stored cryptographic material from the CA server by following the instructions here. You will also need to remove any of the cryptographic tokens stored on the client side by deleting the KeyValStore directory. That directory is configurable and is set to `/tmp/keyValStore` within the unit tests.

Node.js Standalone Application in Vagrant

This section describes how to run a sample standalone Node.js application which interacts with a Hyperledger fabric blockchain.

- If you haven't already done so, see *Setting Up The Application Development Environment* to get your environment set up. The remaining steps assume that you are running **inside the vagrant environment**.
- Issue the following commands to build the Node.js Client SDK:

```
cd /opt/gopath/src/github.com/hyperledger/fabric
make node-sdk
```

- Start the membership services and peer processes. We run the peer in dev mode for simplicity.

```
cd /opt/gopath/src/github.com/hyperledger/fabric/build/bin
membersrvc > membersrvc.log 2>&1&
peer node start --peer-chaincodedev > peer.log 2>&1&
```

- Build and run chaincode example 2:

```
cd /opt/gopath/src/github.com/hyperledger/fabric/examples/chaincode/go/chaincode_
↪example02
go build
CORE_CHAINCODE_ID_NAME=mycc CORE_PEER_ADDRESS=0.0.0.0:7051 ./chaincode_example02 >␣
↪log 2>&1&
```

- Put the following sample app in a file named **app.js** in the */tmp* directory. Take a moment (now or later) to read the comments and code to begin to learn the Node.js Client SDK APIs.

You may retrieve the sample application file:

```
cd /tmp
curl -o app.js https://raw.githubusercontent.com/hyperledger/fabric/v0.6/examples/sdk/
↪node/app.js
```

- Run **npm** to install Hyperledger Fabric Node.js SDK in the `/tmp` directory

```
npm install /opt/gopath/src/github.com/hyperledger/fabric/sdk/node
```

- To run the application :

```
CORE_CHAINCODE_ID_NAME=mycc CORE_PEER_ADDRESS=0.0.0.0:7051 MEMBERSRVC_ADDRESS=0.0.0.0:
↪7054 DEPLOY_MODE=dev node app
```

Congratulations! You've successfully run your first Hyperledger fabric application.

CHAPTER 56

Node.js Web Application

The following is a web application template. It is NOT a complete program.

This template code demonstrates how to communicate with a blockchain from a Node.js web application. It does not show how to listen for incoming requests nor how to authenticate your web users as this is application specific. The code below is intended to demonstrate how to interact with a Hyperledger fabric blockchain from an existing web application.

You may retrieve the sample application file:

```
curl -o app.js https://raw.githubusercontent.com/hyperledger/fabric/v0.6/examples/sdk/
↪node/web-app.js
```