	Module-5
	Hyperledger, Hyperledger as a protocol, Fabric, Hyperledger Fabric, Sawtooth lake, Corda.
	Chapter 9

Hyperledger

Hyperledger is not a blockchain, but it is a project that was initiated by Linux foundation in December 2015 to advance blockchain technology. This project is a collaborative effort by its members to build an open-source distributed ledger framework that can be used to develop and implement cross-industry blockchain applications and systems. The key focus is to build and run platforms that support global business transactions. The project also focuses on improving the reliability and performance of blockchain systems. Projects under Hyperledger undergo various stages of development, starting from proposal to incubation and graduating to an active state. Projects can also be deprecated or in End-of-Life state where they are no longer actively developed. In order for a project to be able to move into incubation stage, it must have a fully working code base along with an active community of developers.

Projects

Currently, there are six projects under the Hyperledger umbrella: Fabric, Iroha, Sawtooth Lake, blockchain explorer, Fabric chain tool, and Fabric SDK Py. Corda is the most recent addition that is expected to be added to the Hyperledger project. The Hyperledger project currently has 100 members and is very active with more than 120 contributors, with regular meet-ups and talks being organized around the globe. A brief introduction of all these projects follows, after which we will provide more details around the design, architecture, and implementation of Fabric and Sawtooth Lake.

Sawtooth lake

Sawtooth lake is a blockchain project proposed by Intel in April 2016 with some key innovations focusing on decoupling of ledgers from transactions, flexible usage across multiple business areas using transaction families, and pluggable consensus. Decoupling can be explained more precisely by saying that the transactions are decoupled from the consensus layer by making use of a new concept called Transaction families. Instead of transactions being individually coupled with the ledger, transaction families are used, which allows for more flexibility, rich semantics and unrestricted design of business logic. Transactions follow the patterns and structures defined in the transaction families. Intel has also introduced a novel consensus algorithm abbreviated as PoET, proof of elapsed time, which makes use of Intel Software Guard Extensions (Intel's SGX) architecture's trusted execution environment (TEE) in order to provide a safe and random leader election process. It also supports permissioned and permission less setups. This project is available at https://github.com/hyperledger/sawtooth-core.

Iroha

Iroha was proposed by Soramitsu, Hitachi, NTT Data, and Colu in September 2016. Iroha is aiming to build a library of reusable components that users can choose to run on their Hyperledger-based distributed ledgers. Iroha's main goal is to complement other Hyperledger projects by providing reusable components written in C++ with an emphasis on mobile development. This project has also proposed a novel consensus algorithm called Sumeragi, which is a chain based Byzantine fault tolerant consensus algorithm. Iroha is available at https://github.com/hyperledger/iroha. Various libraries have been proposed and are being worked on by Iroha, including but not limited to a digital signature library (ed25519), an SHA-3 hashing library, a transaction serialization library, a P2P library, an API server library, an iOS library, an Android library, and a JavaScript library.

Blockchain explorer

This project aims to build a blockchain explorer for Hyperledger that can be used to view and query the transactions, blocks, and associated data from the blockchain. It also provides network information and the ability to interact with chain code. Currently there are two other projects that are in incubation: Fabric chaintool, and Fabric SDK Py. These projects are aimed at supporting Hyperledger Fabric. Fabric chaintool Hyperledger chaincode compiler is being developed to support Fabric chaincode development. The aim is to build a tool that reads in a high-level Google protocol buffer structure and produces a chaincode. Additionally, it packages the chaincode so that it can be deployed directly. It is envisaged that this tool will help developers in various stages of development, such as compiling, testing, packaging, and deployment. It is available at https://github.com/hyperledger/fabric-chaintool.

Fabric SDK Py

The aim of this project is to build a python based SDK library that can be used to interact with the blockchain (Fabric). It is available at https://github.com/hyperledger/fabric-sdk-py. Corda Corda is the latest project that has been contributed by R3 to the Hyperledger project. It was open sourced on November 30, 2016. Corda is heavily oriented towards the financial services industry and has been developed in collaboration with major banks and organizations in the financial industry. At the time of writing it is not yet in incubation under the Hyperledger project. Technically, Corda is not a blockchain but has key features similar to those of a blockchain, such as consensus, validity, uniqueness, immutability, and authentication.

Hyperledger as a protocol

Hyperledger is aiming to build a new blockchain platform that is driven by industry use cases. As there have been number of contributions made to the Hyperledger project by the community, Hyperledger blockchain platform is evolving into a protocol for business transactions. Hyperledger is also evolving into a specification that can be used as a reference to build blockchain platforms as compared to earlier blockchain solutions that address only a specific type of industry or requirement. In the following section, a reference architecture is presented that has been published by the Hyperledger project. As this work is under

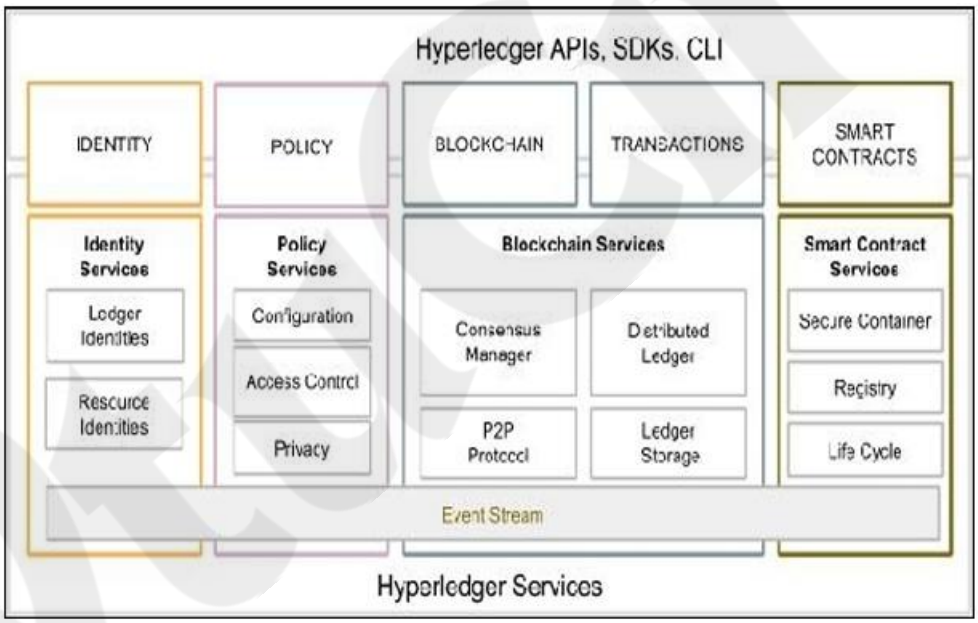
continuous and rigorous development some changes are expected in this, but core services are expected to remain unchanged.

Reference architecture

Hyperledger has published a white paper with reference architecture that can serve as a guideline to build permissioned distributed ledgers.

The reference architecture consists of two main components: Hyperledger services and Hyperledger APIs, SDKs, and CLI.

Hyperledger services provide various services such as identity services, policy services, blockchain services, and smart contract services. On the other hand, Hyperledger APIs, SDKs, and CLIs provide an interface into blockchain services via appropriate application programming interfaces, software development kits, or command line interfaces. Moreover, an event stream, which is basically a gRPC channel, runs across all services. It can receive and send events. Events are either pre-defined or custom. Validating peers or chaincode can emit events to which external application can respond or listen to. The reference architecture that has been published in the Hyperledger white paper at the time of writing is shown in the following diagram. Hyperledger is a rapidly changing and evolving project, and the architecture shown here is expected to change somewhat



Hyperledger architecture, as proposed in the latest draft V2.0.0 of Hyperledger white paper. (Source: Hyperledger white paper)

Requirements

There are certain requirements of a blockchain service. The reference architecture is driven by the needs and requirements raised by the participants of the Hyperledger project and after

studying the industry use cases. There are several categories of requirements that have been deduced from the study of industrial use cases and are discussed in the following sections.

MODULAR APPROACH

The main requirement of Hyperledger is a modular structure. It is expected that, as a cross-industry fabric (blockchain), it will be used in many business scenarios. As such, functions related to storage, policy, chaincode, access control, consensus and many other blockchain services should be pluggable. The modules should be plug and play and users should be able to easily remove and add a different module that meets the requirements of the business. For example, if a business blockchain needs to be run only between already trusted parties and performs very basic business operations, then perhaps there is no need to have advanced cryptographic support for confidentiality and privacy, and therefore users should be able to remove that functionality (module) or replace that with a more appropriate module that suits their needs. Similarly, if users need to run a cross-industry blockchain, then confidentiality and privacy can be of paramount importance. In this case, users should be able to plug an advanced cryptographic and access control mechanism (module) into the blockchain (fabric).

PRIVACY AND CONFIDENTIALITY

Privacy and confidentiality of transactions and contracts is of utmost importance in a business blockchain. As such, Hyperledger's vision is to provide a wide range of cryptographic protocols and algorithms and it is expected that users will be able to choose appropriate modules according to their business requirements. The fabric should be able to handle complex cryptographic algorithms without compromising performance

IDENTITY

In order to provide privacy and confidentiality services, a flexible PKI model that can be used to handle the access control functionality is also required. The strength and type of cryptographic mechanisms is also expected to vary according to the needs and requirements of the users. In certain scenarios it might be required for a user to hide their identity, and as such the Hyperledger is expected to provide this functionality.

AUDITABILITY

Auditability is another requirement of a Hyperledger Fabric. It is expected that an immutable audit trail of all identities, related operations and any changes is kept.

INTEROPERABILITY

Currently there are many blockchain solutions available, but they cannot communicate with each other and this can be a limiting factor in the growth of a blockchain based global business ecosystem. It is envisaged that many blockchain networks will operate in the business world for specific needs, but it is important that they are able to communicate with each other. There should be a common set of standards that all blockchains can follow in order to allow communication between different ledgers. It is expected that a protocol will be developed that will allow the exchange of information between many Fabrics.

PORTABILITY

The portability requirement is concerned with the ability to run across multiple platforms and environments without the need to change anything at code level. Hyperledger is envisaged to be portable, not only at infrastructure level but also at code, libraries, and API levels so that it can support uniform development across various implementations of Hyperledger.

FABRIC

- Fabric can be defined as a collection of components providing a foundation layer that can be used to deliver a blockchain network.
- There are various types and capabilities of a fabric network, but all fabrics share common attributes such as immutability and are consensus driven. Some fabrics can provide modular approach towards building blockchain networks.
- In this case the blockchain network can have multiple pluggable modules to perform various function on the network. For example, consensus algorithms can be a pluggable module in a blockchain network where, depending on the requirements of the network, an appropriate consensus algorithm can be chosen and plugged into the network.
- The modules can be based on some particular specification of the fabric and can include APIs, access control, and various other components. Fabrics can also be designed either to be private or public and can allow the creation of multiple business networks. As an example, bitcoin is an application that runs on top of its fabric (blockchain network).
- blockchain can either be permissioned or permission less and the same is true for fabric in Hyperledger terminology. Fabric is also the name given to the code contribution made by IBM to the Hyperledger foundation and is formally called Hyperledger Fabric. IBM also offers blockchain as a service (IBM Blockchain) via its Bluemix cloud service.

Hyperledger Fabric

- Fabric is the contribution originally made by IBM to the Hyperledger project. The aim of this contribution is to enable a modular, open and flexible approach towards building blockchain networks.
- Various functions in the fabric are pluggable, and it also allows use of any language to develop smart contracts. This is possible because it is based on container technology which can host any language.
- Chaincode (smart contract) is sandboxed into a secure container which includes a secure operating system, chaincode language, runtime environment and SDKs for Go, Java, and Node.js. Other languages can be supported too if required.
- Smart contracts are called chaincode in the Fabric. This is a very powerful feature compared to domain specific languages in Ethereum, or the very limited scripted language in bitcoin.
- It is a permissioned network that aims to address issues such as scalability, privacy, and confidentiality. The key idea behind this is modular technology, which would allow

for flexibility in design and implementation. This can then result in achieving scalability, privacy and other desired attributes.

- Transactions in fabric are private, confidential and anonymous for general users, but they can still be traced and linked to the users by authorized auditors. As a permissioned network, all participants are required to be registered with the membership services in order to access the blockchain network. This ledger also provided auditability functionality in order to meet the regulatory and compliance needs

Fabric architecture

- The Fabric is logically organized into three main categories based on the type of service provided. These include membership services, blockchain services, and chaincode services. The current stable version of Hyperledger Fabric is v0.6, however the latest version v1.0 is available but is not yet stable.

MEMBERSHIP SERVICES

These services are used to provide access control capability for the users of the fabric network. The following list shows the functions that membership services perform

1. User identity validation.

2. User registration.

3. Assign appropriate permissions to the users depending on their roles.

Membership services makes use of Public Key Infrastructure (PKI) in order to support identity management and authorization operations. **Membership services are made up of various components:**

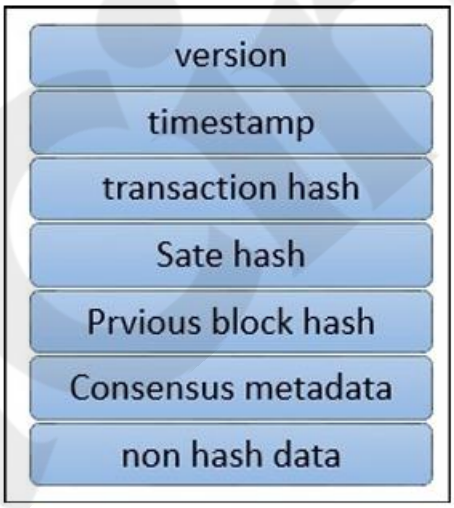
I. **Registration authority (RA):** A service that authenticates the users and assesses the identity of the fabric participants for issuance of certificates.
II. **Enrolment certificate authority:** Enrolment certificates (Ecerts) are long term certificates issued by ECA to registered participants in order to provide identification to the entities participating on the network.
III. **Transaction certificate authority:** In order to send transactions on the networks, participants are required to hold a transaction certificate. TCA is responsible for issuing transaction certificates to holders of Enrolment certificates and is derived from Ecerts.
IV. **TLS certificate authority:** In order to secure the network level communication between nodes on the Fabric, TLS certificates are used. TLS certificate authority issues TLS certificates in order to ensure security of the messages being passed between various systems on the blockchain network.

BLOCKCHAIN SERVICES

Blockchain services are at the core of the Hyperledger Fabric. Components within this category are as follows.

i. **Consensus manager** Consensus manager is responsible for providing the interface to the consensus algorithm. This serves as an adapter that receives the transaction from other Hyperledger entities and executes them under criteria according to the type of algorithm chosen. Consensus is pluggable and currently there are three types of consensus algorithm available in Fabric, namely the batch PBFT protocol, SIEVE algorithm, and NOOPS.

ii. **Distributed ledger** Blockchain and world state are two main elements of the distributed ledger. Blockchain is simply a linked list of blocks (as introduced in earlier chapters) and world ledger is a key-value database. This database is used by smart contracts to store relevant states during execution by the transactions. The blockchain consists of blocks that contain transactions. These transactions contain chaincode, which runs transactions that can result in updating the world state. Each node saves the world state on disk in RocksDB. The following diagram shows a typical block in the Hyperledger Fabric with the relevant fields



Block structure

The fields shown in the preceding diagram are as follows:

- Version: Used for keeping track of changes in the protocol.
- Timestamp: Timestamp in UTC epoch time, updated by block proposer.
- Transaction hash: This field contains the Merkle root hash of the transactions in the block.
- State hash: This is the Merkle root hash of the world state.
- Previous hash: This is the previous block's hash, which is calculated after serializing the block message and then creating the message digest by applying the SHA3 SHAKE256 algorithm.

- **Consensus metadata**: This is an optional field that can be used by the consensus protocol to provide some relevant information about the consensus.
- **Non-Hash data**: This is some metadata that is stored with the block but is not hashed. This feature makes it possible to have different data on different peers. It also provides the ability to discard data without any impact on the blockchain

Peer to Peer protocol

- P2P protocol in the Hyperledger Fabric is built using google RPC (gRPC). It uses protocol buffers to define the structure of the messages. Messages are passed between nodes in order to perform various functions.
- There are four main types of messages in Hyperledger Fabric: **Discovery, transaction, synchronization and consensus.**
- Discovery messages are exchanged between nodes when starting up in order to discover other peers on the network.
- Transaction messages can be divided into **two** types: **Deployment transactions and Invocation transactions**
- The former is used to deploy new chaincode to the ledger, and the latter is used to call functions from the smart contract. Transactions can be public, confidential, and confidential chaincode transactions. Public transactions are open and available to all participants. Confidential transactions are allowed to be queried only by transaction owners and participants. Confidential chaincode transactions have encrypted chaincode and can only be decrypted by validating nodes. Validating nodes run consensus, validate the transactions and maintain the blockchain. Non-validating nodes on the other hand, provide transaction verification, stream server, and REST services. They also act as a proxy between the transactors and the validating nodes.
- Synchronization messages are used by peers to keep the blockchain updated and in synch with other nodes. Consensus messages are used in consensus management and broadcasting payloads to validating peers. These are generated internally by the consensus framework. Ledger storage In order to save the state of the ledger, RocksDB is used, and it is stored at each peer. RocksDB is a high performance database available at http://rocksdb.org/.

CHAINCODE SERVICES

These services allow the creation of secure containers that are used to execute the chaincode. Components in this category are as follows:

i. **Secure container:** Chaincode is deployed in Docker containers that provide a locked down sandboxed environment for smart contract execution. Currently Golang is supported as the main smart contract language, but any other main stream language can be added and enabled if required.

ii. **Secure registry:** This provides a record of all images containing smart contracts

EVENTS

- Events on the blockchain can be triggered by validator nodes and smart contracts. External applications can listen to these events and react to them if required via event adapters.

APIS AND CLIS

- An application programming interface provides an interface into the fabric by exposing various REST APIs. Additionally, command line interfaces that provide a subset of REST APIs and allow for quick testing and limited interaction with the blockchain are also available.

Components of the Fabric

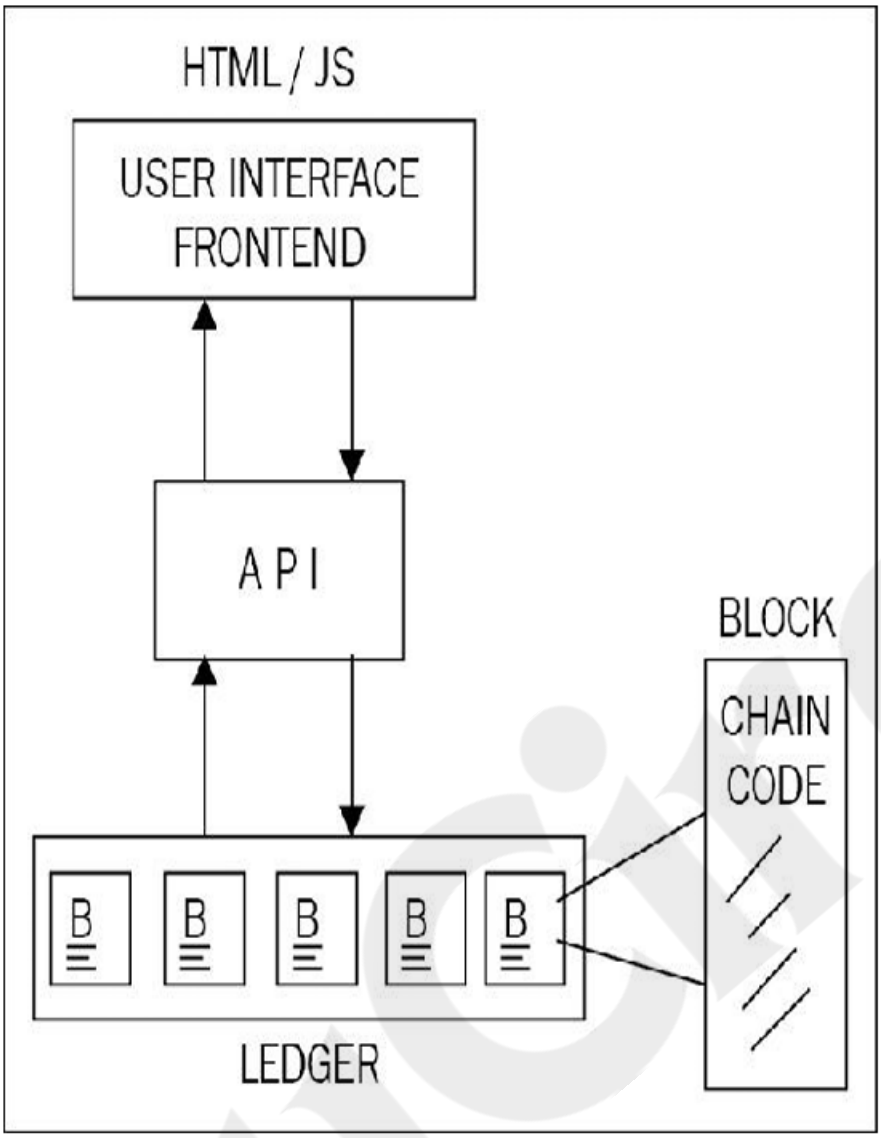
There are various components that can be part of the blockchain. These components include but are not limited to the ledger, chaincode, consensus mechanism, access control, events, system monitoring and management, wallets and system integration components.

PEERS OR NODES

- There are two main types of peers that can be run on a fabric network: Validating and non-validating. Simply put, a validating node runs consensus, creates and validates a transaction, and contributes towards updating the ledger and maintaining the chaincode.
- A non-validating peer does not execute transactions and only constructs transactions that are then forwarded to validating nodes. Both nodes manage and maintain user certificates that have been issued by membership services.

APPLICATIONS ON BLOCKCHAIN

A typical application on Fabric is simply composed of a user interface, usually written in JavaScript/HTML, that interacts with the backend chaincode (smart contract) stored on the ledger via an API layer.



Typical blockchain application

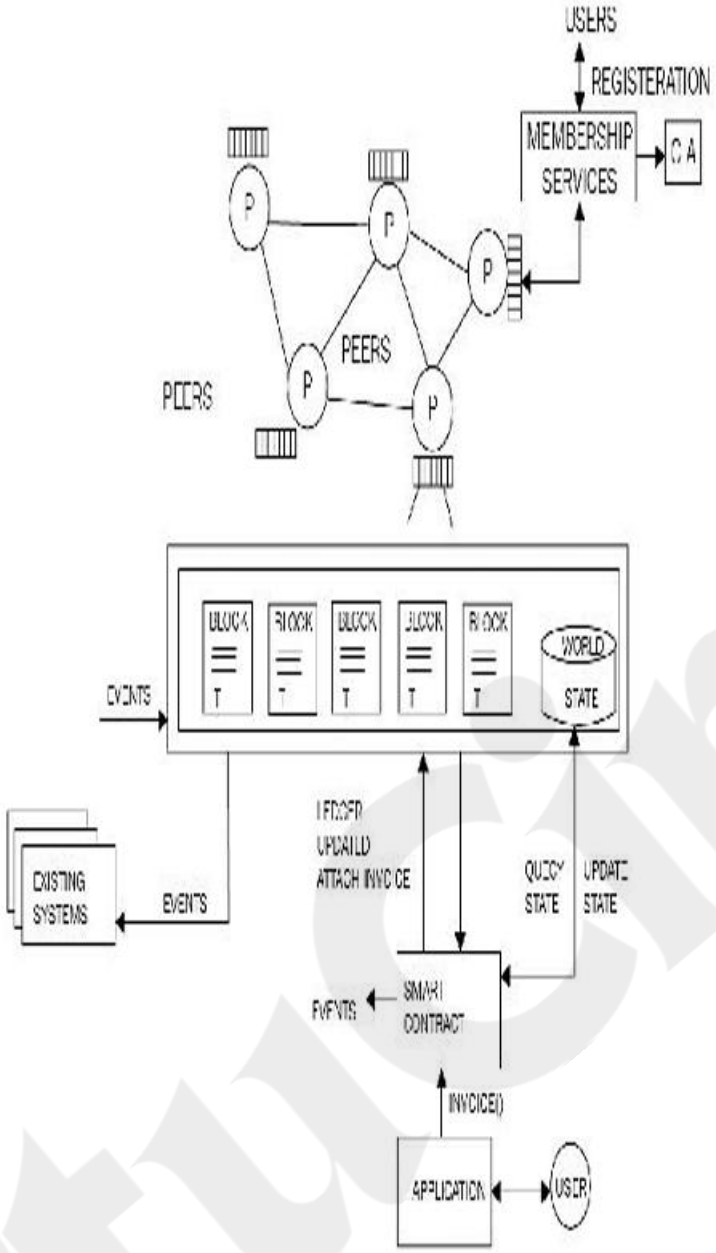
Hyperledger provides various APIs and command line interfaces to enable interaction with the ledger. These APIs include interfaces for identity, transactions, chaincode, ledger, network, storage, and events.

Chaincode implementation

- Chaincode is usually written in Golang or Java. Chaincode can be public, confidential or access controlled. These codes serve as a smart contract that users can interact with via APIs.
- Users can call functions in the chaincode that result in a state change, and consequently updates the ledger. There are also functions that are only used to query the ledger and do not result in any state change.

- Chaincode implementation is performed by first creating the chaincode shim interface in the code. It can either be in Java or Golang code. The following four functions are required in order to implement the chaincode:
    i. **Init():** This function is invoked when chaincode is deployed onto the ledger. This initializes the chaincode and results in making a state change, which accordingly updates the ledger.
    ii. **Invoke():** This function is used when contracts are executed. It takes a function name as parameters along with an array of arguments. This function results in a state change and writes to the ledger.
    iii. **Query():** This function is used to query the current state of a deployed chaincode. This function does not make any changes to the ledger.
    iv. **Main():** This function is executed when a peer deploys its own copy of the chaincode. The chaincode is registered with the peer using this function.

The following diagram illustrates **the general overview of Hyperledger Fabric:**



HIGH LEVEL OVERVIEW OF HYPERLEDGER FABRIC

Application model

Any blockchain application for Hyperledger Fabric follows MVC-B architecture. This is based on the popular MVC design pattern. Components in this model are Model, View, Control, and Blockchain:

View logic: This is concerned with the user interface. It can be a desktop, web application or mobile frontend.

Control logic: This is the orchestrator between user interface, data model, and APIs. Data model: This model is used to manage the off-chain data.

Blockchain logic: This is used to manage the blockchain via the controller and the data model via transactions.

Sawtooth lake

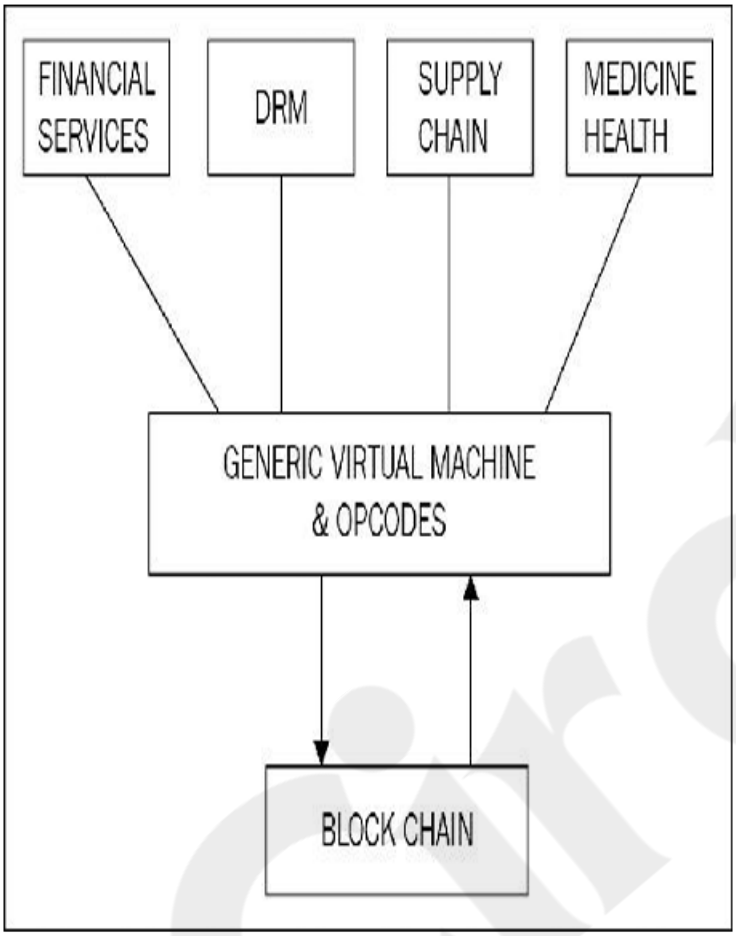
Sawtooth lake can run in both permissioned and non-permissioned modes. It is a distributed ledger that proposes two novel concepts: The first is the introduction of a new consensus algorithm called **Proof of Elapsed Time (PoET);** and the second is **the idea of transaction families**.

PoET-> Proof of Elapsed Time

- PoET is a novel consensus algorithm that allows a node to be selected randomly based on the time that the node has waited before proposing a block. This is in contrast to other leader election and lottery based proof of work algorithms, where an enormous amount of electricity and computer resources are used in order be elected as a block proposer, for example in the case of bitcoin.
- PoET is a type of Proof of Work algorithm but, instead of spending computer resources, it uses a trusted computing model to provide a mechanism to fulfill Proof of Work requirements.
- PoET makes use of Intel's SGX architecture to provide a trusted execution environment to ensure randomness and cryptographic security of the process. It should be noted that the current implementation of Sawtooth lake does not require real hardware SGX based TEE, as it is simulated for experimental purposes only and as such should not be used in production environments.

Transaction families

- A traditional smart contract paradigm provides a solution that is based on a general purpose instruction set for all domains. For example, in the case of Ethereum, a set of opcodes has been developed for the Ethereum virtual machine (EVM) that can be used to build smart contracts to address any type of requirements for any industry. Whilst this model has its merits, it is becoming clear that this approach is not very secure as it provides a single interface into the ledger with a powerful and expressive language, which potentially offers a larger attack surface for malicious code. This complexity and generic virtual machine paradigm has resulted in several vulnerabilities that were found and exploited recently by hackers. A recent example is the DAO hack and further Denial of Services (DoS) attacks that exploited limitations in some EVM opcodes.
- A model shown in the following figure describes the traditional smart contract model, where a generic virtual machine has been used to provide the interface into the blockchain for all domains.

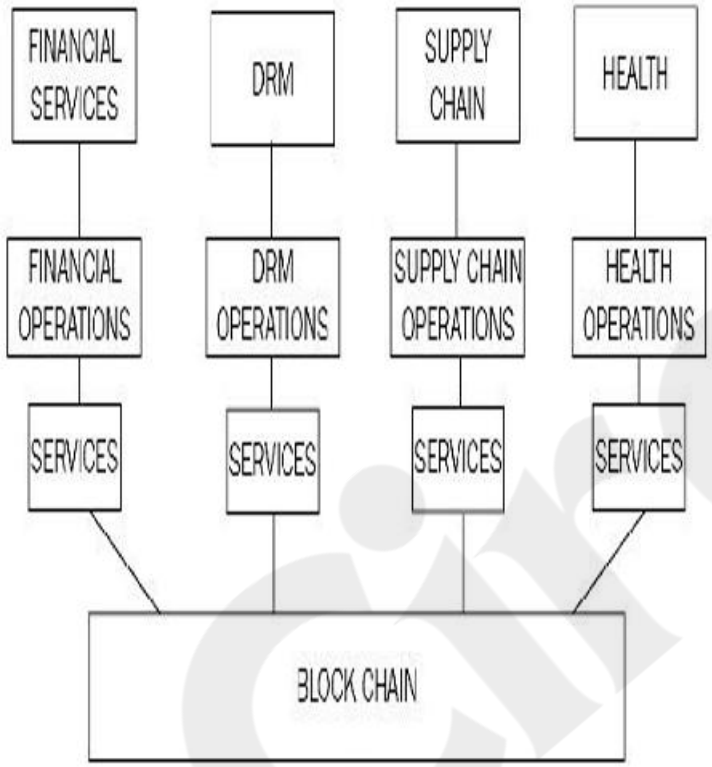


Traditional smart contract paradigm

- In order to address this issue, Sawtooth lake has proposed the idea of transaction families.
- A transaction family is created by decomposing the logic layer into a set of rules and a composition layer for a specific domain.
- The key idea is that business logic is composed within transaction families, which provides a more secure and powerful way to build smart contracts. Transaction families contain the domain-specific rules and another layer that allows for creating transactions for that domain.
- Another way of looking at it is that transaction families are a combination of a data model and a transaction language that implements a logic layer for a specific domain. The data model represents the current state of the blockchain (ledger) whereas the transaction language modifies the state of the ledger. It is expected that users will build their own transaction families according to their business requirements.
- The following diagram represents this model, where each specific domain, like financial services, digital rights management (DRM), supply chain, and the health industry, has its own logic layer comprised of operations and services specific to

that domain. This makes the logic layer both restrictive and powerful at the same time.

- Transaction families ensure that operations related to only the required domain are present in the control logic, thus removing the possibility of executing needless, arbitrary and potentially harmful operations.



Sawtooth (transaction families) smart contract paradigm

- Intel has provided three transaction families with Sawtooth: Endpoint registry, Integerkey, and MarketPlace.
  1. Endpoint registry is used for registering ledger services.
  2. Integerkey is used for testing deployed ledgers.
  3. MarketPlace is used for selling, buying and trading operations and services. Sawtooth_bond has been developed as a proof of concept to demonstrate a bond trading platform. It is available at https://github.com/hyperledger/sawtooth core/tree/master/extensions/bond.

Consensus in Sawtooth

- Sawtooth has two types of consensus mechanisms based on the choice of network. **PoET,** is a trusted executed environment-based lottery function that elects a leader randomly based on the time a node has waited for block proposal.
- There is another consensus type called **quorum voting,** which is an adaptation of consensus protocols built by Ripple and Stellar. This consensus algorithm allows instant transaction finality, which is usually desirable in permissioned networks.

Development environment

In this section, a quick introduction is given on how to set up a development environment for Sawtooth lake.

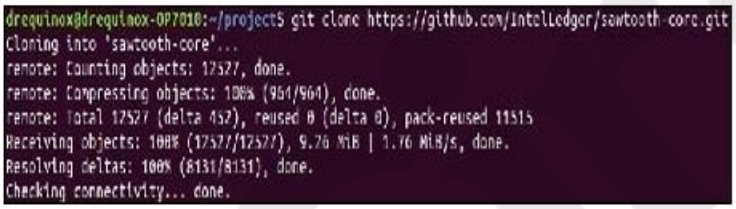
There are few pre-requisites that are required in order to set up the development environment. Examples in this section assume a running Ubuntu system and the following:

1. vagrant, at least version 1.9.0, available at https://www.vagrantup.com/downloads.html.
2. Virtual box, at least 5.0.10 r104061, available at https://www.virtualbox.org/wiki/Downloads.

Once both of the above pre-requisites are downloaded and installed successfully, the next step is to clone the repository.

```
$ git clone
https://github.com/IntelLedger/sawtooth-
core.git
```

This will produce an output similar to the one shown in the following screenshot:



GitHub Sawtooth clone Once Sawtooth is cloned correctly, the next step is to start up the environment. First, run the following command to change the directory to the correct location and then start the vagrant box.

If at any point Vagrant needs to be stopped, the following command can be used

```
$ vagrant halt
```

Or

```
$ vagrant destroy
```

Halt will stop the `vagrant` machine, whereas `destroy` will stop and delete `vagrant` machines.

Finally, the transaction validator can be started by using the following commands. First `ssh` into the `vagrant` Sawtooth box.

```
$ vagrant ssh
```

When the vagrant prompt is available, run the following commands. First build the sawtooth lake core using following command:
When the vagrant prompt is available, run the following commands. First build the sawtooth lake core using following command.

```
$ /project/sawtooth-core/bin/build_all
```

When the build has completed successfully, in order to run transaction validator issue the following commands:

```
$ /project/sawtooth-
core/docs/source/tutorial/genesis.sh
```

This will create the genesis block and clear any existing data files and keys. This should show an output similar to the following screenshot:

Genesis block and keys generation

The next step is to run the transaction validator, and change the directory as shown follows:

```
$ cd /project/saw-toothcore
```

Run the transaction validator:

```
$ ./bin/txnvalidator -v -F
ledger.transaction.integer_key --config
/home/ubuntu/sawtooth/v0.json
```

Running transaction validator

The validator node can be stopped by pressing Ctrl + C. Once the validator is up and running, various clients can be started up in another terminal window to communicate with the transaction validator and submit transactions.

For example, in the following screenshot the market client is started up to communicate with the transaction validator. Note that keys under /keys/mkt.wif are created by using the following command

```
./bin/sawtooth keygen --key-dir
validator/keys mkt
```

This demonstration is just a basic example derived from Sawtooth lake documentation. However, development using Sawtooth lake is quite an involved process and a full chapter could be dedicated to that.

mktclient for marketplace transaction family

CORDA

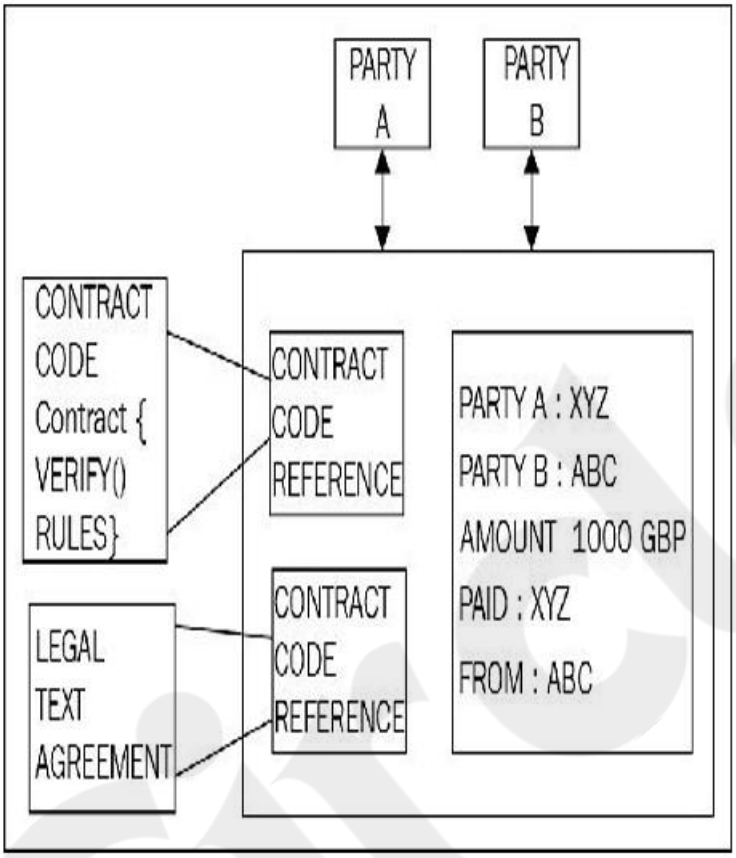
- Corda is not a blockchain. Traditional blockchain solutions, as discussed before, have the concept of transactions that are bundled together in a block and each block is linked back cryptographically to its parent block, which provides an immutable record of transactions. This is not the case with Corda:
- Corda has been designed entirely from scratch with a new model for providing all blockchain benefits, but without a traditional blockchain. It has been developed purely for the financial industry to solve issues arising from the fact that each organization manages their own ledgers and thus have their own view of truth, which leads to contradictions and operational risk. Moreover, data is also duplicated at each organization which results in an increased cost of managing individual infrastructures and complexity. These are the types of problems within the financial industry that Corda aims to resolve by building a decentralized database platform. Corda source code is available at https://github.com/corda/corda. It is written in a language called Kotlin, which is a statically typed language targeting the Java Virtual Machine (JVM).

ARCHITECTURE

The main components of the Corda platform include **state objects, contract code, legal prose, transactions, consensus, and flows**.

i. STATE OBJECTS

- State objects represent the smallest unit of data that represent a financial agreement. They are created or deleted as a result of a transaction execution. They refer to contract code and legal prose.
- Legal prose is optional and provides legal binding to the contract. However, contract code is mandatory in order to manage the state of the object. It is required in order to provide a state transition mechanism for the node according to the business logic defined in the contract code.
- State objects contain a data structure that represent the current state of the object. For example, in the following diagram, a state object represents the current state of the object.
- In this case, it is a simple mock agreement between Party A and Party B where Party ABC has paid Party XYZ 1,000 GBP. This represents the current state of the object; however the referred contract code can change the state via transactions.
- State objects can be thought of as a state machine, which are consumed by transactions in order to create updated state objects



An example state object

TRANSACTIONS

- Transactions are used to perform transitions between different states. For example, the state object shown in the preceding diagram is created as a result of a transaction.
- Corda uses a bitcoin-style UTXO based model for its transaction processing. The concept of state transition by transactions is same as in bitcoin. Similar to bitcoin, transactions can have none, single or multiple inputs, and single or multiple outputs. All transactions are digitally signed. Moreover, Corda has no concept of mining because it does not use blocks to arrange transactions in a blockchain. Instead, notary services are used in order to provide temporal ordering of transactions. In Corda, new transaction types can be developed using JVM bytecode, which makes it very flexible and powerful.

CONSENSUS

- The consensus model in Corda is quite simple and is based on **notary services**. The general idea is that the transactions are evaluated for their uniqueness by the notary service and, if they are

unique, they are signed as valid. There can be single or multiple clustered notary services running on a Corda network.

- Various consensus algorithms like PBFT or Raft can be used by notaries to reach consensus. There are two main concepts regarding consensus in Corda: Consensus over state validity, and consensus over state uniqueness. ***The first concept is concerned with the validation of the transaction, ensuring that all required signatures are available and states are appropriate***. ***The second concept is a means to detect double--spend attack and ensures that a transaction has not been already been spent and is unique.***

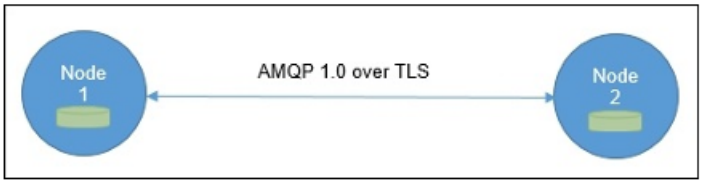
FLOWS

- Flows in Corda are a novel idea that allow the development of decentralized workflows. All communication on the Corda network is handled by these flows.
- These are transaction-building protocols that can be used to define any financial flow of any complexity using code. Flows run as an asynchronous state machine and they interact with other nodes and users. During the execution, they can be suspended or resumed as required.

Components The Corda network has multiple components.

NODES

- Nodes in a Corda network operated under a trust-less model and run by different organizations.
- Nodes run as part of an authenticated peer-to-peer network. Nodes communicate directly with each other using the Advanced Message Queuing Protocol (AMQP), which is an approved international standard (ISO/IEC 19464) and ensures that messages across different nodes are transferred safely and securely.
- AMQP works over Transport Layer Security (TLS) in Corda, thus ensuring privacy and integrity of data communicated between nodes. Nodes also make use of a local relational database for storage. Messages on the network are encoded in a compact binary format. They are delivered and managed by using the Apache Artemis message broker (Active MQ).
- A node can serve as a network map service, notary, Oracle, or a regular node.
- The following diagram shows a high-level view of two nodes communicating with each other.



Two nodes communicating in a Corda network

In the preceding diagram, Node 1 is communicating with Node 2 over a TLS communication channel using the AMQP protocol, and the nodes have a local relational database for storage.

PERMISSIONING SERVICE

- A Permissioning service is used to provision TLS certificates for security. In order to participate on the network, participants are required to have a signed identity issued by a root certificate authority. Identities are required to be unique on the network and the Permissioning service is used to sign these identities. The naming convention used to recognise participants is based on the X.500 standard. This ensures the uniqueness of the name.

NETWORK MAP SERVICE

- This service is used to provide a network map in the form of a document of all nodes on the network. This service publishes IP addresses, identity certificates and a list of services offered by nodes. All nodes announce their presence by registering to this service when they first start up, and when a connection request is received by a node, the presence of the requesting node is checked on the network map first. Put another way, this service resolves the identities of the participants to physical nodes.

NOTARY SERVICE

- In a traditional blockchain, mining is used to ascertain the order of blocks that contain transactions. In Corda, notary services are used to provide transaction ordering and timestamping services. There can be multiple notaries in a network and they are identified by composite public keys. Notaries can use different consensus algorithms like BFT or Raft depending on the requirements of the applications. Notary services sign the transactions to indicate validity and finality of the transaction which is then persisted to the database. Notaries can be run in a load-balanced configuration in order to spread the load across the nodes for performance reasons; and, in order to reduce latency, the nodes are recommended to be run physically closer to the transaction participants.

ORACLE SERVICE

Oracle services either sign a transaction containing a fact, if it is true, or can themselves provide factual data. They allow real world feed into the distributed ledgers.

TRANSACTIONS

Transactions in a Corda network are never transmitted globally, but in a semi-private network. They are shared only between a subset of participants who are related to the transaction. This is in contrast to traditional blockchain solutions like Ethereum and bitcoin, where all transactions are broadcasted to the entire network globally. Transactions are digitally signed and either consume state(s) or create new state(s).

Transactions on a Corda network are composed of the following elements:

a. Input references: This is a reference to the states the transaction is going to consume and use as an input.
b. Output states: These are new states created by the transaction.
c. Attachments: This is a list of hashes of attached zip files. Zip files can contain code and other relevant documentation related to the transaction. Files themselves are not made part of the transaction, instead, they are transferred and stored separately.
d. Commands: A command represents the information about the intended operation of the transaction as a parameter to the contract. Each command has a list of public keys which represents all parties that are required to sign a transaction.
e. Signatures: This represents the signature required by the transaction. The total number of signatures required is directly proportional to the number of public keys for commands.
f. Type: There are two types of transactions namely, Normal or Notary changing. Notary changing transactions are used for reassigning a notary for a state.
g. Timestamp: This field represents a bracket of time during which the transaction has taken place. These are verified and enforced by notary services. Also, it is expected that if strict timings are required, which is desirable in many financial services scenarios, notaries should be synched with an atomic clock.
h. Summaries: This is a text description that describes the operations of the transaction.

VAULTS:

Vaults run on a node and are akin to the concept of wallets in bitcoin. As the transactions are not globally broadcast, each node will have only that part of data in their vaults that is considered relevant to them. Vaults store their data in a standard relational database and as such can be queried by using standard SQL. Vaults can contain both on ledger and off ledger data, meaning that it can also have some part of data that is not on ledger.

CORDAPP

The core model of Corda consists of state objects, transactions and transaction protocols, which when combined with contract code, APIs, wallet plugins, and user interface components results in constructing a Corda distributed application (CorDapp). Smart contracts in Corda are written using Kotlin or Java. The code is targeted for JVM. JVM has been modified slightly in order to achieve deterministic results of execution of JVM bytecode.

- There are three main components in a Corda smart contract as follows:
  1. Executable code that defines the validation logic to validate changes to the state objects.
  2. State objects represent the current state of a contract and either can be consumed by a transaction or produced (created) by a transaction.
  3. Commands are used to describe the operational and verification data that defines how a transaction can be verified.

  DEVELOPMENT ENVIRONMENT

  The development environment for Corda can be set up easily using the following steps.

  Required software includes the following:

  1.**JDK8** which is available at http://www.oracle.com/technetwork/java/javase/downloads/index.ht m l.

  2. **IntelliJ IDEA** community edition which is free and available at https://www.jetbrains.com/idea/download .

  3. **H2 database** platform independent zip, and is available at http://www.h2database.com/html/download.html.

  4.**Git,** available at https://git-scm.com/downloads.

  5. **Kotlin language**, which is available for IntelliJ, and more information can be found at https://kotlinlang.org/.

6. **Gradle** is another component that is used to build Corda. Once all these tools are installed, smart contract development can be started. CorDapps can be developed by utilizing an example template available at https://github.com/corda/cordapp-template. Detailed documentation on how to develop contract code is available at https://docs.corda.net/

Corda can be cloned locally from GitHub using the following command

```
$ git clone
https://github.com/corda/corda.git
```

When the cloning is successful, you should see output similar to the following:

```
Cloning into 'corda'...
remote: Counting objects: 74695, done.
remote: Compressing objects: 100% (67/67),
done.
remote: Total 74695 (delta 17), reused 0
(delta 0), pack-reused 74591
Receiving objects: 100% (74695/74695),
51.27 MiB | 1.72 MiB/s, done.
Resolving deltas: 100% (42863/42863),
done.
Checking connectivity... done.
```

Once the repository is cloned, it can be opened in IntelliJ for further development. There are multiple samples available in the repository, such as a bank of Corda, interest rate swaps, demo, and traders demo. Readers can find them under the /samples directory under corda and they can be explored using IntelliJ IDEA IDE.