Block Chain Technology

20CS4601C

UNIT_5

Hyperledger: Overview, Hyperledger Reference Architecture, Hyperledger fabric. Ripple, Storj, multichain, BigchainDB, Quorum

Blockchain-Outside of Currencies: Internet of Things, Government, Health, Finance, Media, aviation, voting, identity management, stock trading, agriculture.

15
Hyperledger

Hyperledger is not a blockchain, but it is a project that was initiated by the 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 principal focus is to develop 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. For a project to be able to move into the incubation stage, it must have a fully working code base along with an active community of developers.

Projects under Hyperledger

There are two categories of projects under Hyperledger. The first is **blockchain projects** and the second category is **relevant tools or modules that support these blockchains**.

Currently, there are five blockchain framework projects under the Hyperledger umbrella: **Fabric**, **Sawtooth Lake**, **Iroha**, **Burrow**, and **Indy**. Under modules, there are the **Hyperledger Cello**, **Hyperledger Composer**, **Hyperledger Explorer**, and **Hyperledger Quilt**. The Hyperledger project currently has more than 200-member organizations and is very active with many contributors, with regular meet-ups and talks organized around the globe.

A brief introduction of all these projects follows, after which we will see more details around the design, architecture, and implementation of Fabric and Sawtooth Lake.

Fabric

The fabric is a blockchain project that was proposed by **IBM** and **DAH** (**Digital Asset Holdings**). This blockchain framework implementation is intended to provide a foundation for the development of blockchain solutions with a modular architecture. It is based on a pluggable architecture where various components, such as consensus engine and membership services, can be plugged into the system as required. It also makes use of container technology which is used to run smart contracts in an isolated contained environment. Currently, its status is *active* and it's the first project to graduate from incubation to active state.

The source code is available at `https://github.com/hyperledger/fabric`.

Sawtooth Lake

The Sawtooth Lake is a blockchain project proposed by Intel in April 2016 with some key innovations focusing on the 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 open design of business logic. Transactions follow the patterns and structures defined in the transaction families.

Some of the innovative elements Intel has introduced include a novel consensus algorithm abbreviated as **PoET**, **Proof of Elapsed Time**, which makes use of **Trusted Execution Environment** (**TEE**) provided by **Intel Software Guard Extensions** (**Intel's SGX**) 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 contributed 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 own Hyperledger-based distributed ledgers.

Iroha's primary 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), a 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.

Burrow

This project is currently in the incubation state. Hyperledger Burrow was contributed by Monax, who develop blockchain development and deployment platforms for business. Hyperledger Burrow introduces a modular blockchain platform and an **Ethereum Virtual Machine** (**EVM**) based smart contract execution environment. Burrow uses proof of stake, Byzantine fault tolerant Tendermint consensus mechanism. As a result, Burrow provides high throughput and transaction finality.

The source code is available at `https://github.com/hyperledger/burrow`.

Indy

This project is under incubation under Hyperledger. Indy is a distributed ledger developed for building a decentralized identity. It provides tools, utility libraries, and modules which can be used to build blockchain-based digital identities. These identities can be used across multiple blockchains, domains, and applications. Indy has its own distributed ledger and uses **Redundant Byzantine Fault Tolerance** (**RBFT**) for consensus.

The source code is available at `https://github.com/hyperledger/indy-node`.

Explorer

This project aims to build a blockchain explorer for Hyperledger Fabric 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 few other projects that are in incubation under Hyperledger. These projects are aimed to provide tools and utilities to support blockchain networks. These projects are introduced in the following section.

The source code is available at `https://github.com/hyperledger/blockchain-explorer`.

Cello

The aim behind Cello is to allow easy deployment of blockchains. This will provide an ability to allow "as a service" deployments of blockchain service. Currently, this project is in the incubation stage.

The source code of Cello is available at `https://github.com/hyperledger/cello`.

Composer

This utility makes the development of blockchain solutions easier by allowing business processes to be described in a business language, while abstracting away the low-level smart contract development details.

Hyperledger composer is available at `https://hyperledger.github.io/composer/`.

Quilt

This utility implements the Interledger protocol, which facilitates interoperability across different distributed and non-distributed ledger networks.

Quilt is available at `https://github.com/hyperledger/quilt`.

Currently, all the mentioned projects are in various stages of development.

This list is expected to grow as more and more members are joining Hyperledger project and contributing to the development of blockchain technology. Now in the next section, we will see the reference architecture of Hyperledger, which provides general principles and design philosophy which can be followed to build new Hyperledger projects.

Hyperledger as a protocol

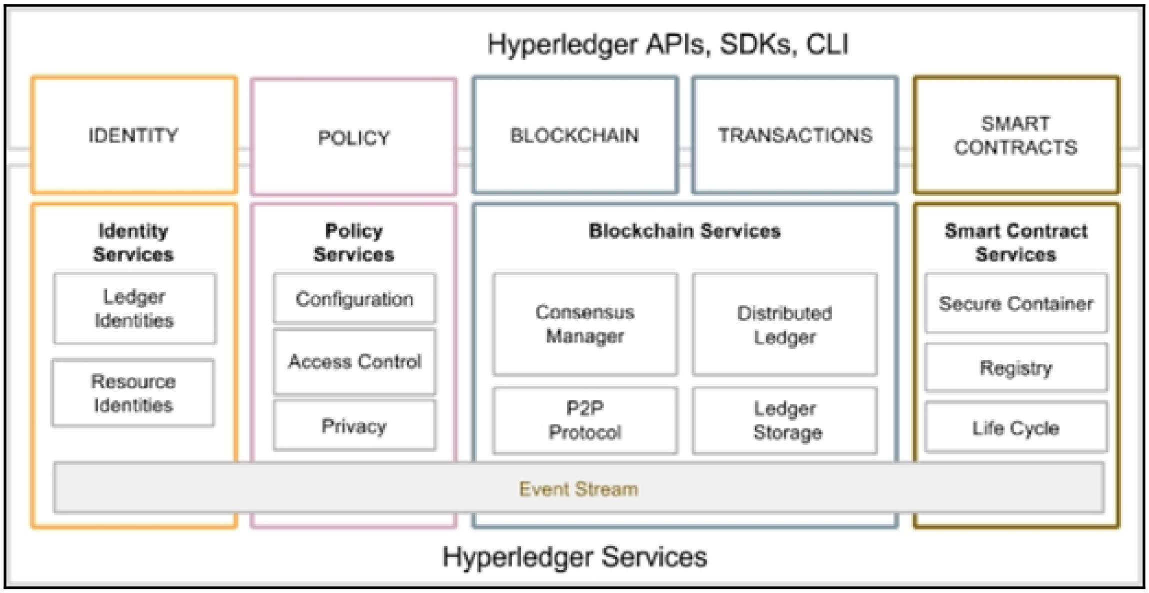
Hyperledger is aiming to build new blockchain platforms that are driven by industry use cases. As there have been many contributions made to the Hyperledger project by the community, Hyperledger blockchain platform is evolving into a protocol for business transactions. Hyperledger is also evolving into 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.

The reference architecture

Hyperledger has published a white paper which is available at `https://docs.google.com/document/d/1Z4M_qwILLRehPbVRUsJ3OF8Iir-gqS-ZYe7W-LE9gnE/edit#heading=h.m6iml6hqrnm2`.

This document presents a reference architecture that can serve as a guideline to build permissioned distributed ledgers. The reference architecture consists of various components that form a business blockchain. These high-level components are shown in the reference architecture diagram shown here:



Reference architecture - source: Hyperledger whitepaper

Starting from the left we see that we have five top-level components which provide various services. We will explore all these components in detail.

First is identity, that provides authorization, identification, and authentication services under membership services.

Then is the policy component, which provides policy services.

After this, ledger and transactions come, which consists of the distributed ledger, ordering service, network protocols, and endorsement and validation services. This ledger is updateable only via consensus among the participants of the blockchain network.

Finally, we have the smart contracts layer, which provides chaincode services in Hyperledger and makes use of secure container technology to host smart contracts. We will see all these in more detail in the *Hyperledger Fabric* section shortly.

Generally, from a components point of view Hyperledger contains various elements described here:

- **Consensus layer**: These services are responsible for facilitating the agreement process between the participants on the blockchain network. The consensus is required to make sure that the order and state of transactions is validated and agreed upon in the blockchain network.
- **Smart contract layer**: These services are responsible for implementing business logic as per the requirements of the users. Transaction are processed based on the logic defined in the smart contracts that reside on the blockchain.
- **Communication layer**: This layer is responsible for message transmission and exchange between the nodes on the blockchain network.
- **Security and crypto layer**: These services are responsible for providing a capability to allow various cryptographic algorithms or modules to provide privacy, confidentiality and non-repudiations services.
- **Data stores**: This layer provides an ability to use different data stores for storing state of the ledger. This means that data stores are also pluggable and allows usage of any database backend.
- **Policy services**: This set of services provide the ability to manage different policies required for the blockchain network. This includes endorsement policy and consensus policy.
- **APIs and SDKs**: This layer allows clients and applications to interact with the blockchain. An SDK is used to provide mechanisms to deploy and execute chaincode, query blocks and monitor events on the blockchain.

There are certain requirements of a blockchain service. In the next section, we are going to discuss the design goals of Hyperledger Fabric.

Requirements and design goals of Hyperledger Fabric

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.

The 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 modular and pluggable. The specification suggests that 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.

Privacy and confidentiality

This requirement is one of the most critical factors. As traditional blockchains are permissionless, in the permissioned model like Hyperledger Fabric, it is of utmost importance that transactions on the network are visible to only those who are allowed to view it.

Privacy and confidentiality of transactions and contracts are of absolute importance in a business blockchain. As such, Hyperledger's vision is to provide support for a full range of cryptographic protocols and algorithms. We discussed cryptography in `Chapter 3`, *Symmetric Cryptography* and `Chapter 4`, *Public Key Cryptography*.

It is expected that users will be able to choose appropriate modules according to their business requirements. 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. 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), which can even allow usage of **hardware of security modules** (**HSMs**).

Also, the blockchain should be able to handle sophisticated cryptographic algorithms without compromising performance. In addition to the previously mentioned scenarios, due to regulatory requirements in business, there should also be a provision to allow implementation of privacy and confidentiality policies in conformance with regulatory and compliance requirements.

Scalability

This is another major requirement which once met will allow reasonable transaction throughput, which will be sufficient for all business requirements and also a large number of users.

Deterministic transactions

This is a core requirement in any blockchain because if transactions do not produce the same result every time they are executed regardless of who and where the transaction is executed, then achieving consensus is impossible. Therefore, deterministic transactions become a key requirement in any blockchain network. We discussed these concepts in `Chapter 9`, *Smart Contracts*.

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 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 platforms 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 Fabric 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.

Rich data queries

The blockchain network should allow rich queries to be run on the network. This can be used to query the current state of the ledger using traditional query languages, which will allow for wider adoption and ease of use.

All aforementioned points describe the requirements, which need to be met to develop blockchain solutions that are in line with the Hyperledger design philosophy. In the next section, we will have a look at Hyperledger Fabric, which is the first project to graduate to active status under Hyperledger.

Fabric

To understand various projects that are under development in the Hyperledger project, it is essential to understand the foundations of Hyperledger first. A few terminologies that are specific to Hyperledger need some clarification before readers are introduced to the more in-depth material.

First, there is the concept of 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 a modular approach towards building blockchain networks. In this case, the blockchain network can have multiple pluggable modules to perform a 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). As discussed earlier in `Chapter 1`, *Blockchain 101*, blockchain can either be permissioned or permission-less. However, the aim of Hyperledger Fabric is to develop a permissioned distributed ledger.

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 *IBM Cloud service*.

It is available at `https://www.ibm.com/cloud/`.

Now let's have a detailed look at Hyperledger Fabric.

Hyperledger Fabric

The fabric is the contribution made initially by IBM and Digital Assets to the Hyperledger project. This contribution aims to enable a modular, open, and flexible approach towards building blockchain networks.

Various functions in the fabric are pluggable, and it also allows the use of any language to develop smart contracts. This functionality is possible because it is based on container technology (Docker), which can host any language.

Chaincode is sandboxed in 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 in future, if required, but needs some development work. Smart contracts are called chaincode in the fabric. This ability is a compelling feature compared to domain-specific languages in Ethereum, or the limited scripted language in Bitcoin. It is a permissioned network that aims to address issues such as scalability, privacy, and confidentiality. The fundamental idea behind this is modularization, which would allow for flexibility in design and implementation of the business blockchain. This can then result in achieving scalability, privacy, and other desired attributes and fine tune them according to the requirements.

Transactions in the 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 to access the blockchain network. This ledger also provided auditability functionality to meet the regulatory and compliance needs required by the user.

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:

- User identity verification
- User registration
- Assign appropriate permissions to the users depending on their roles

Membership services make use of **a certificate authority** in order to support identity management and authorization operations. This CA can be internal (Fabric CA), which is a default interface in Hyperledger Fabric or organization can opt to use an external certificate authority. Fabric CA issues **enrollment certificates** (**E-Certs**), which are produced by **enrollment certificate authority** (**E-CA**). Once peers are issued with an identity, they are allowed to join the blockchain network. There are also temporary certificates issued called T-Certs, which are used for one-time transactions.

All peers and applications are identified using certificate authority. Authentication service is provided by the certificate authority. MSPs can also interface with existing identity services like LDAP.

Blockchain services

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

Consensus services

A consensus service is responsible for providing the interface to the consensus mechanism. This serves as a module that is pluggable and receives the transaction from other Hyperledger entities and executes them under criteria according to the type of mechanism chosen.

Consensus in Hyperledger V1 is implemented as a peer called **orderer**, which is responsible for ordering the transactions in sequence into a block. Orderer does not hold smart contracts or ledgers. Consensus is pluggable and currently, there are two types of ordering services available in Hyperledger Fabric:

- **SOLO**: This is a basic ordering service intended to be used for development and testing purposes.
- **Kafka**: This is an implementation of Apache Kafka, which provides ordering service. It should be noted that currently Kafka only provides crash fault tolerance but does not provide byzantine fault tolerance. This is acceptable in a permissioned network where chances of malicious actors are almost none.

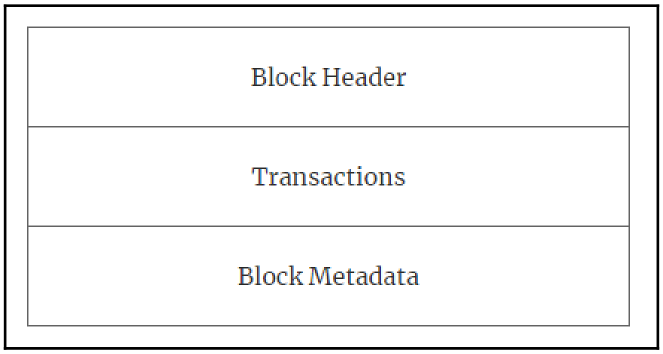
In addition to these mechanisms, the **Simple Byzantine Fault Tolerance** (**SBFT**) based mechanism is also under development, which will become available in the later releases of Hyperledger Fabric.

Distributed ledger

Blockchain and world state are two main elements of the distributed ledger. Blockchain is simply a cryptographically linked list of blocks (as introduced in `Chapter 1`, *Blockchain 101*) and world state 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 LevelDB or CouchDB depending on the implementation. As Fabric allows pluggable data store, you can choose any data store for storage.

A block consists of three main components called Block header, Transactions (Data) and block metadata.

The following diagram shows a typical block in the Hyperledger Fabric 1.0 with the relevant fields:



Block structure

Block Header consists of three fields, namely Number, Previous hash, and Data hash.

Transaction is made up of multiple fields such as transaction type, version, timestamp, channel ID, transaction ID, epoch, payload visibility, chaincode path, chaincode name, chaincode version, creator identity, signature, chaincode type, input, timeout, endorser identities and signatures, proposal hash, chaincode events, response status, namespace, read set, write set, start key, end key, list of read, and Merkle tree query summary.

Block Metadata consists of creator identity, relevant signatures, last configuration block number, flag for each transaction included in the block, and last offset persisted (kafka).

The peer to peer protocol

The P2P protocol in the Hyperledger Fabric is built using **google RPC** (**gRPC**). It uses protocol buffers to define the structure of the messages.

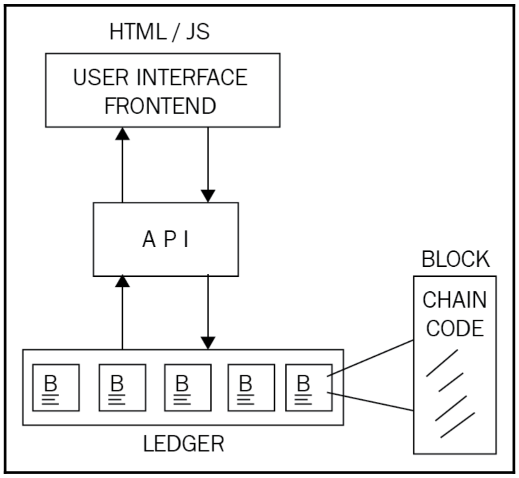
Messages are passed between nodes in order to perform various functions. There are four main types of messages in Hyperledger Fabric: **discovery**, **transaction**, **synchronization**, and **consensus**. Discovery messages are exchanged between nodes when starting up in order to discover other peers on the network. Transaction messages are used to deploy, invoke, and query transactions, and consensus messages are exchanged during consensus. Synchronization messages are passed between nodes to synchronize and keep the blockchain updated on all nodes.

Ledger storage

In order to save the state of the ledger, by default, LevelDB is used which is available at each peer. An alternative is to use CouchDB which provides the ability to run rich queries.

Chaincode services

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



- **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 mainstream languages can be added and enabled if required.
- **Secure registry:** This provides a record of all images containing smart contracts.

Events

Events on the blockchain can be triggered by endorsers and smart contracts. External applications can listen to these events and react to them if required via event adapters. They are similar to the concept of events introduced in solidity in `Chapter 14`, *Introducing Web3*.

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 Hyperledger Fabric 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

Peers participate in maintaining the state of the distributed ledger. They also hold a local copy of the distributed ledger. Peers communicate via gossip protocol. There are three types of peers in the Hyperledger Fabric network:

- **Endorsing peers** or endorsers which simulate the transaction execution and generate a read-write set. Read is a simulation of transaction's reading of data from the ledger and write is the set of updates that would be made to the ledger if and when the transaction is executed and committed to the ledger. Endorses execute and endorse transactions. It should be noted that an endorser is also a committer too. Endorsement policies are implemented with chaincode and specify the rules for transaction endorsement.
- **Committing peers** or committers which receives transaction endorsed by endorsers, verify them and then update the ledger with the read-write set. A committer verifies the read-write set generated by the endorsers along with transaction validation.
- **Submitters** is the third type of peers which has not been implemented yet. It is on the development roadmap and will be implemented

Orderer nodes

Ordering nodes receive transactions from endorsers along with read-write sets, arrange them in a sequence, and send those to committing peers. Committing peers then perform validation and committing to the ledger.

All peers make use of certificates issued by membership services.

Clients

Clients are software that makes use of APIs to interact with the Hyperledger Fabric and propose transactions.

Channels

Channels allow the flow of confidential transactions between different parties on the network. They allow using the same blockchain network but with separate blockchains. Channels allow only members of the channel to view the transaction related to them, all other members of the network will not be able to view the transactions.

World state database

World state reflects all committed transaction on the blockchain. This is basically a key-value store which is updated as a result of transactions and chaincode execution. For this purpose, either LevelDB or CouchDB is used. LevelDB is a key-value store whereas CouchDB stores data as JSON objects which allows rich queries to run against the database.

Transactions

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 either public or confidential. Public transactions are open and available to all participants whilst confidential transactions are visible only in a channel open to its participants.

Membership Service Provider (MSP)

MSP is a modular component that is used to manage identities on the blockchain network. This provider is used to authenticate clients who want to join the blockchain network. We have discussed certificate authority is some detail earlier in this chapter. CA is used in MSP to provide identity verification and binding service.

Smart contracts

We discussed smart contracts in good detail in `Chapter 9`, *Smart Contracts*. In Hyperledger Fabric same concept of smart contracts is implemented but they are called chain code instead of smart contracts. They contain conditions and parameters to execute transactions and update the ledger. Chaincode is usually written in Golang and Java.

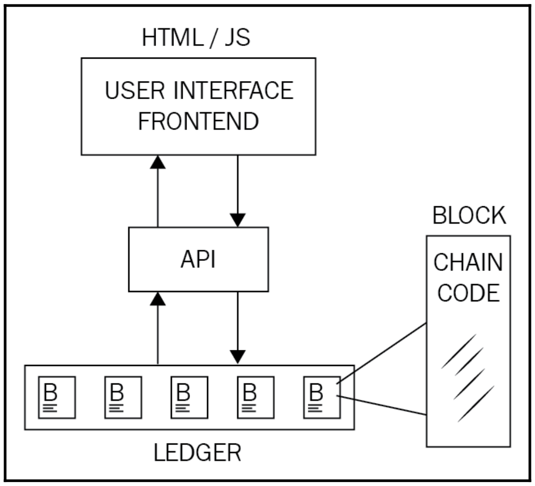
Crypto service provider

As the name suggests this is a service that provides cryptographic algorithms and standards for usage in the blockchain network. This service provides key management, signature and verification operations, and encryption-decryption mechanisms. This service is used with the membership service to provide support for cryptographic operations for elements of blockchain such as endorsers, clients, and other nodes and peers.

After this introduction to this component of Hyperledger Fabric, in the next section, we will see what an application looks like when on a Hyperledger network.

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:



A typical Fabric 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

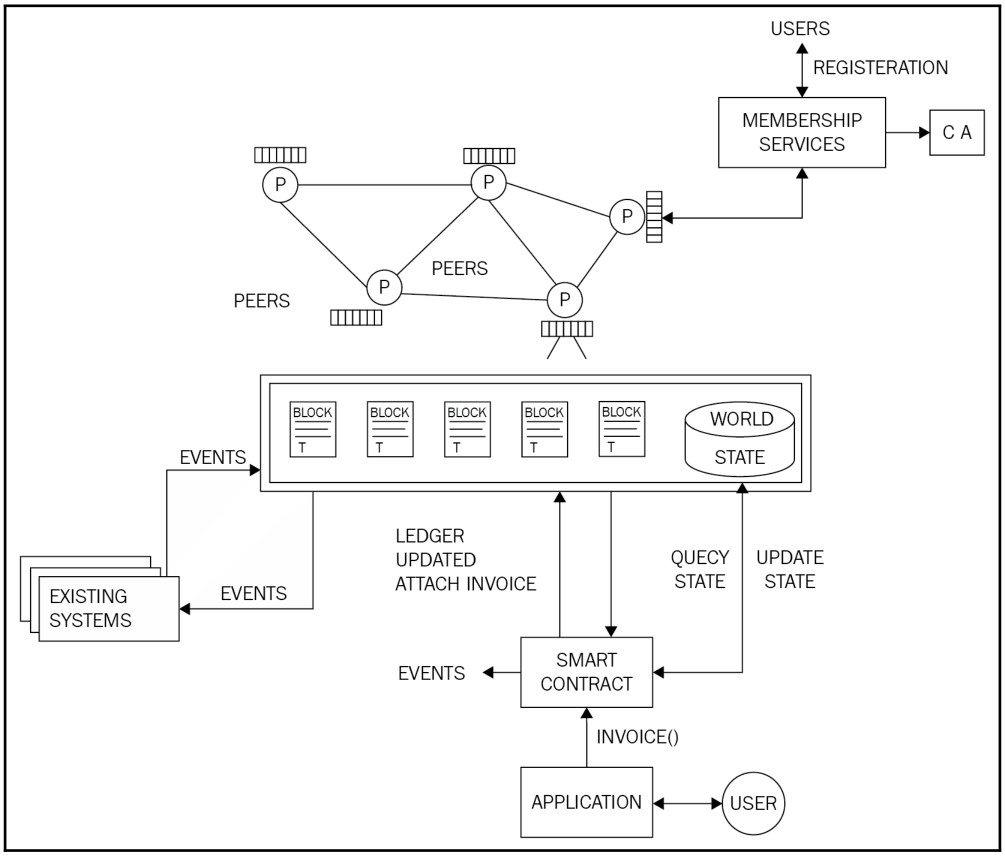
Chaincode is usually written in Golang or Java. Chaincode can be public (visible to all on the network), confidential, or access controlled. These code files serve as a smart contract that users can interact with via APIs. Users can call functions in the chaincode that result in a state change, and consequently 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. Shim provides APIs for accessing state variables and transaction context of chain code. It can either be in Java or Golang code.

The following four functions are required in order to implement the chaincode:

- `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.
- `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.
- `Query()`: This function is used to query the current state of a deployed chaincode. This function does not make any changes to the ledger.
- `4()`: 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, note that peers cluster at the top includes all types of nodes such as endorsers, committers, Orderers, and so on.



A high-level overview of Hyperledger Fabric

The preceding diagram shows that peers shown at the top middle communicate with each and each node has a copy of blockchain. On the top-right corner, the membership services are shown which validate and authenticate peers on the network by using a **certificate authority** (**CA**). At the bottom of the image, a magnified view of blockchain is shown where by existing systems can produce events for the blockchain and also can listen for the blockchain events, which then can optionally trigger an action. At the bottom right-hand side, a user's interaction is shown with the application which talks to the smart contract via the `invoice()` method, and smart contracts can query or update the state of the blockchain.

The application model

Any blockchain application for Hyperledger Fabric follows the 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 the 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.

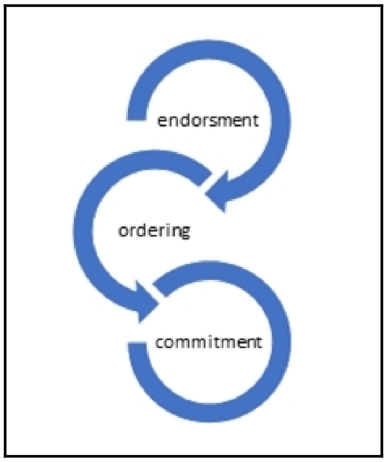
The IBM cloud service offers sample applications for blockchain under its blockchain as a service offering. It is available at `https://www.ibm.com/blockchain/platform/.This service allows users to create their own blockchain networks in an easy-to-use environment.`

Consensus in Hyperledger Fabric

The consensus mechanism in Hyperledger Fabric consists of three steps:

1. **Transaction endorsement**: This process endorses the transactions by simulating the transaction execution process.
2. **Ordering**: This is a service provided by the cluster of orderers which takes endorsed transactions and decide on a sequence in which the transactions will be written to the ledger.
3. **Validation and commitment**: This process is executed by committing peers which first validates the transactions received from the orderers and then commit that transaction to the ledger.

These steps are shown in the following flowchart:

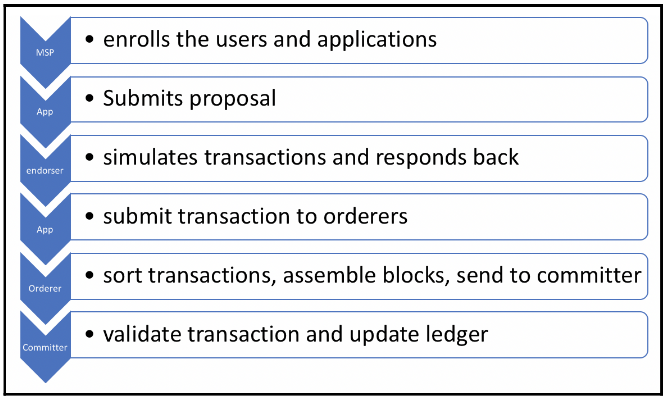


The consensus flow

The transaction life cycle in Hyperledger Fabric

There are several steps that are involved in a transaction flow in Hyperledger Fabric. These steps are shown in the following diagram below

A quick summary of the process can be visualized in the following diagram:

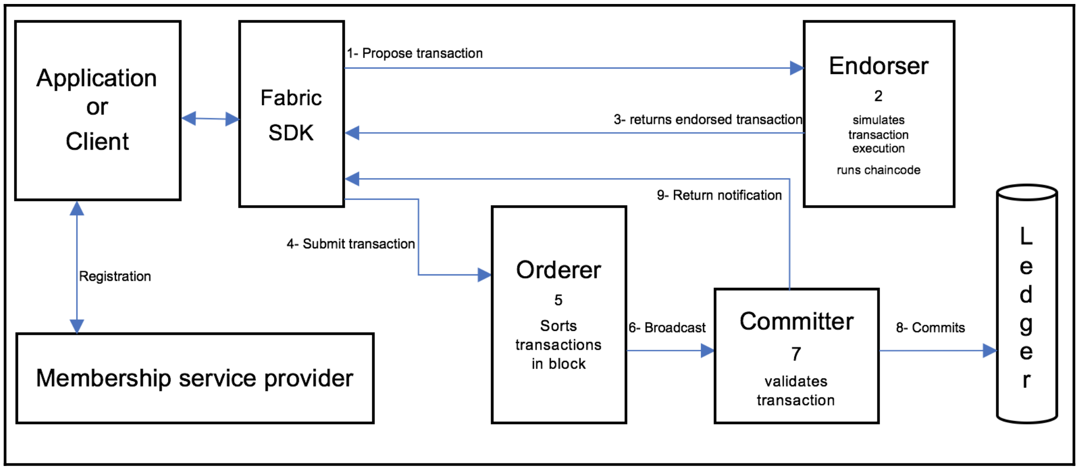

MSP
• enrolls the users and applications
App
• Submits proposal
endorser
• simulates transactions and responds back
App
• submit transaction to orderers
Orderer
• sort transactions, assemble blocks, send to committer
Committer
• validate transaction and update ledger

The transaction life cycle

The steps are described below in detail:

1. Transaction proposal by clients. This is the first step where a transaction is proposed by the clients and sent to endorsing peers on the distributed ledger network. All clients need to be enrolled via membership services before they can propose transactions.
2. The transaction is simulated by endorsers which generates a read-write (RW) set. This is achieved by executing the chaincode but instead of updating the ledger, only a read-write set depicting any reads or updates to the ledger is created.
3. The endorsed transaction is sent back to the application.
4. Submission of endorsed transactions and read-write (RW) sets to the ordering service by the application.
5. The ordering service assembles all endorsed transactions and read-write sets in order into a block, and sorts them by channel ID.
6. Ordering service broadcasts the assembled block to all committing peers.
7. Committing peers validate the transactions.
8. Committing peers update the ledger.
9. Finally, notification of success or failure of the transaction by committing peers is sent back to the clients/applications.

The following diagram represents the above-mentioned steps and Fabric architecture from transaction flow point of view:



The transaction flow architecture

As seen in the preceding diagram, the first step is to propose transactions which a client does via an SDK. Before this, it is assumed that all clients and peers are registered with the Membership service provider.

With this topic, our introduction to Hyperledger Fabric is complete. In the next section, we will see another Hyperledger project named Sawtooth Lake.

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**.

A description of these novel proposals is given in the following sections.

PoET

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 concept is in contrast to other leader election and lottery-based proof of work algorithms, such as the PoW used in Bitcoin 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 the Proof of Work requirements. PoET makes use of Intel's SGX architecture (Software Guard Extensions) to provide a trusted execution environment (TEE) 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. The fundamental idea in PoET is to provide a mechanism of leader election by waiting randomly to be elected as a leader for proposing new transactions.

PoET, however, has a limitation which has been highlighted by Ittay Eyal. This limitation is called the *stale chips* problem.

The research paper is available at `https://eprint.iacr.org/2017/179.pdf`.

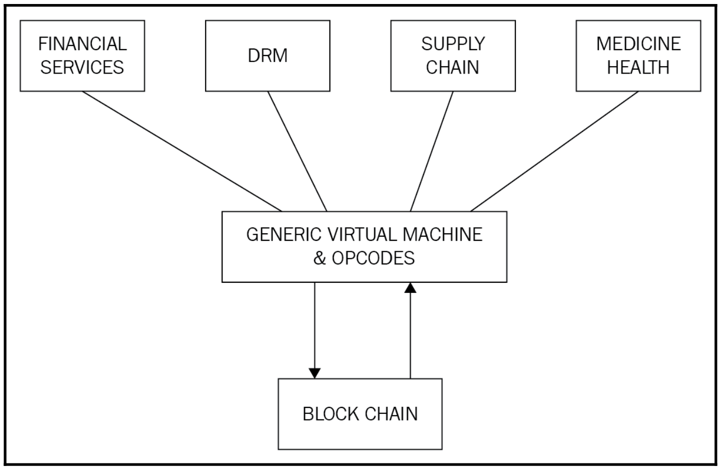
This limitation results in hardware wastage, which can result in the waste of resources. There is also a possibility of hacking the chip's hardware, which could result in system compromise and undue incentivizing to miners.

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 EVM that can be used to build smart contracts to address any type of requirements for any industry.

While 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 have 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. The DAO hack was discussed in `Chapter 9`, *Smart Contracts*.

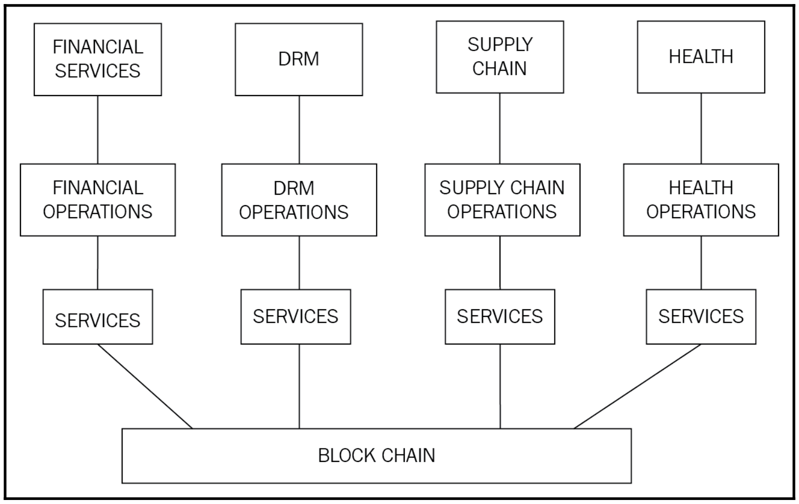
A model shown in the following figure describes the traditional smart contract model, where a generic virtual machine has been used to provide the interface into the blockchain for all domains:



The 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:



The Sawtooth (transaction families) smart contract paradigm

Intel has provided three transaction families with Sawtooth: Endpoint registry, Integerkey, and MarketPlace.

- **Endpoint registry** is used for registering ledger services
- **Integerkey** is used for testing deployed ledgers
- **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, as discussed previously, 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.

The development environment – Sawtooth Lake

In this section, a quick introduction is given on how to set up a development environment for Sawtooth Lake. There are a few prerequisites that are required in order to set up the development environment.

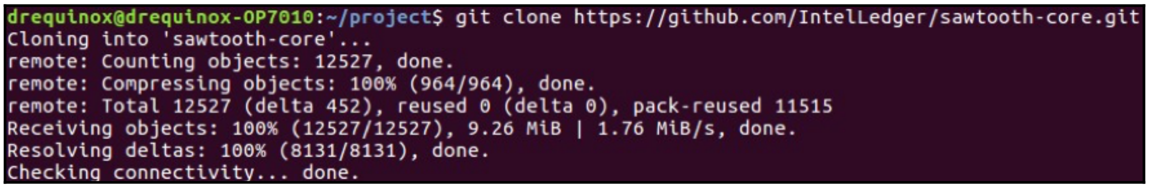
Examples in this section assume a running Ubuntu system and the following:

- Vagrant, at least version 1.9.0, available at `https://www.vagrantup.com/downloads.html`.
- VirtualBox, at least 5.0.10 r104061, available at `https://www.virtualbox.org/wiki/Downloads`.

Once both of the prerequisites 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:



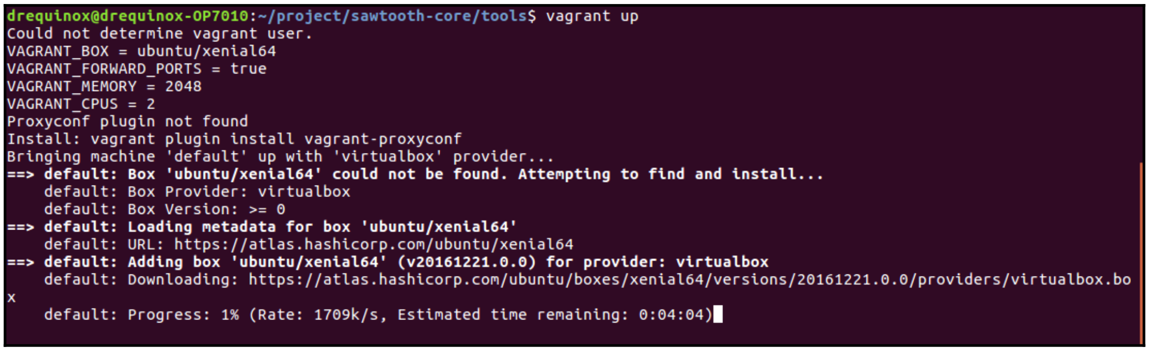
```
drequinox@drequinox-OP7010:~/project$ git clone https://github.com/IntelLedger/sawtooth-core.git
Cloning into 'sawtooth-core'...
remote: Counting objects: 12527, done.
remote: Compressing objects: 100% (964/964), done.
remote: Total 12527 (delta 452), reused 0 (delta 0), pack-reused 11515
Receiving objects: 100% (12527/12527), 9.26 MiB | 1.76 MiB/s, done.
Resolving deltas: 100% (8131/8131), done.
Checking connectivity... done.
```

The 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:

```
$ cd sawtooth-core/tools
$ vagrant up
```

This will produce an output similar to the following screenshot:



```
drequinox@drequinox-OP7010:~/project/sawtooth-core/tools$ vagrant up
Could not determine vagrant user.
VAGRANT_BOX = ubuntu/xenial64
VAGRANT_FORWARD_PORTS = true
VAGRANT_MEMORY = 2048
VAGRANT_CPUS = 2
Proxyconf plugin not found
Install: vagrant plugin install vagrant-proxyconf
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Box 'ubuntu/xenial64' could not be found. Attempting to find and install...
    default: Box Provider: virtualbox
    default: Box Version: >= 0
==> default: Loading metadata for box 'ubuntu/xenial64'
    default: URL: https://atlas.hashicorp.com/ubuntu/xenial64
==> default: Adding box 'ubuntu/xenial64' (v20161221.0.0) for provider: virtualbox
    default: Downloading: https://atlas.hashicorp.com/ubuntu/boxes/xenial64/versions/20161221.0.0/providers/virtualbox.bo
x
    default: Progress: 1% (Rate: 1709k/s, Estimated time remaining: 0:04:04)
```

The vagrant up command

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
```

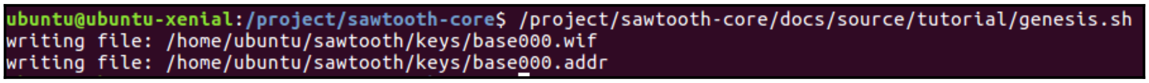
When the vagrant prompt is available, run the following commands. First build the Sawtooth Lake core using following command:

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

When the build has completed successfully, in order to run the 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 command should show an output similar to the following screenshot:



```
ubuntu@ubuntu-xenial:/project/sawtooth-core$ /project/sawtooth-core/docs/source/tutorial/genesis.sh
writing file: /home/ubuntu/sawtooth/keys/base000.wif
writing file: /home/ubuntu/sawtooth/keys/base000.addr
```

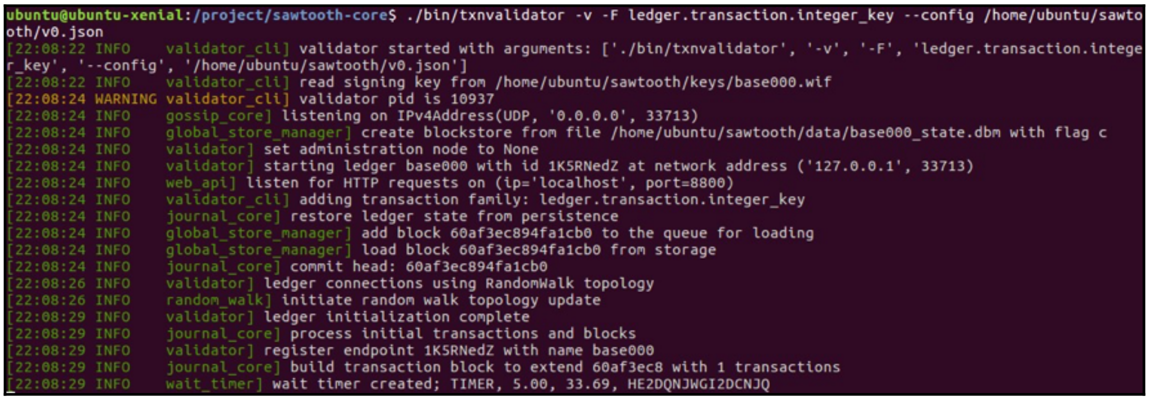
Genesis block and keys generation

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
```



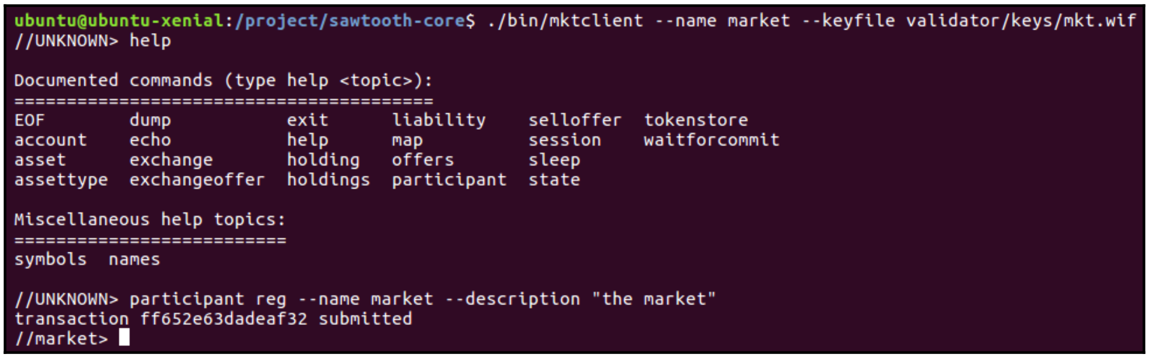
```
ubuntu@ubuntu-xenial:/project/sawtooth-core$ ./bin/txnvalidator -v -F ledger.transaction.integer_key --config /home/ubuntu/sawto
oth/v0.json
[22:08:22 INFO    validator_cli] validator started with arguments: ['./bin/txnvalidator', '-v', '-F', 'ledger.transaction.intege
r_key', '--config', '/home/ubuntu/sawtooth/v0.json']
[22:08:22 INFO    validator_cli] read signing key from /home/ubuntu/sawtooth/keys/base000.wif
[22:08:24 WARNING validator_cli] validator pid is 10937
[22:08:24 INFO    gossip_core] listening on IPv4Address(UDP, '0.0.0.0', 33713)
[22:08:24 INFO    global_store_manager] create blockstore from file /home/ubuntu/sawtooth/data/base000_state.dbm with flag c
[22:08:24 INFO    validator] set administration node to None
[22:08:24 INFO    validator] starting ledger base000 with id 1K5RNedZ at network address ('127.0.0.1', 33713)
[22:08:24 INFO    web_api] listen for HTTP requests on (ip='localhost', port=8800)
[22:08:24 INFO    validator_cli] adding transaction family: ledger.transaction.integer_key
[22:08:24 INFO    journal_core] restore ledger state from persistence
[22:08:24 INFO    global_store_manager] add block 60af3ec894fa1cb0 to the queue for loading
[22:08:24 INFO    global_store_manager] load block 60af3ec894fa1cb0 from storage
[22:08:24 INFO    journal_core] commit head: 60af3ec894fa1cb0
[22:08:26 INFO    validator] ledger connections using RandomWalk topology
[22:08:26 INFO    random_walk] initiate random walk topology update
[22:08:29 INFO    validator] ledger initialization complete
[22:08:29 INFO    journal_core] process initial transactions and blocks
[22:08:29 INFO    validator] register endpoint 1K5RNedZ with name base000
[22:08:29 INFO    journal_core] build transaction block to extend 60af3ec8 with 1 transactions
[22:08:29 INFO    wait_timer] wait timer created; TIMER, 5.00, 33.69, HE2DQNJWGI2DCNJQ
```

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
```



```
ubuntu@ubuntu-xenial:/project/sawtooth-core$ ./bin/mktclient --name market --keyfile validator/keys/mkt.wif
//UNKNOWN> help

Documented commands (type help <topic>):
========================================
EOF        dump           exit      liability    selloffer  tokenstore
account    echo           help      map          session    waitforcommit
asset      exchange       holding   offers       sleep
assettype  exchangeoffer  holdings  participant  state

Miscellaneous help topics:
==========================
symbols  names

//UNKNOWN> participant reg --name market --description "the market"
transaction ff652e63dadeaf32 submitted
//market>
```

mktclient for marketplace transaction family

This completes our basic introduction to Sawtooth. The example shown above is also quite basic but demonstrates that how Sawtooth Lake works.

Sawtooth Lake is also under continuous development and therefore, it is recommended that readers keep an eye on documentation available at `http://intelledger.github.io/` to keep up with the latest developments.

There is an excellent online page where official Sawtooth lake examples are provided. The page is available at `https://sawtooth.hyperledger.org/examples/`. Readers are encouraged to visit this page and explore these sample projects.

Now in the next section we will see an introduction to Corda. It should be noted that Corda is not yet an official project under Hyperledger; however, it may become a member very soon. Therefore, for now, this is being discussed under Hyperledger, but in the future, it may not become part of Hyperledger.

Corda

Corda is not a blockchain by definition because it does not contain blocks of bundled transactions, but it falls under the category of distributed ledgers. It provides all benefits that a blockchain can. 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.

The 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**).

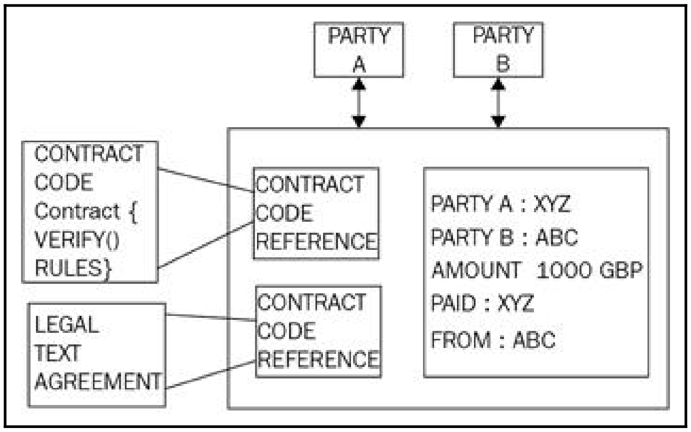
Architecture

The main components of the Corda platform include state objects, contract code, legal prose, transactions, consensus, and flows. We will now explore them in more detail.

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 represents the current state of the object. A state object can be either current (live) or historic (no longer valid).

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 that are discussed in a later section of this chapter. The general idea is that the transactions are evaluated for their uniqueness by the notary service and, if they are unique (that is, unique transaction inputs), they are signed by consensus services 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 attacks and ensures that a transaction has not already been spent and is unique.

Flows

Flows in Corda are a novel idea that allows 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.

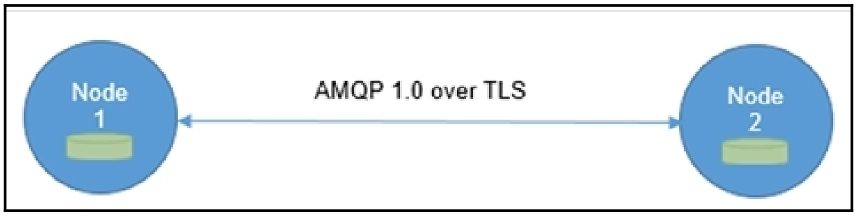
Components

The Corda network has multiple components. All these components are described in the upcoming sections.

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.

The permissioning service

A permissioning service is used to provision TLS certificates for security. In order to participate in 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 recognize 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 startup, 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. Oracles were discussed in `Chapter 9`, *Smart Contracts*.

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:

- **Input references**: This is a reference to the states the transaction is going to consume and use as an input.
- **Output states**: These are new states created by the transaction.
- **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.
- **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.
- **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.
- **Type**: There are two types of transactions namely, normal or notary changing. Notary changing transactions are used for reassigning a notary for a state.
- **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 synced with an atomic clock.
- **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 broadcasted, 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:

- Executable code that defines the validation logic to validate changes to the state objects.
- State objects represent the current state of a contract and either can be consumed by a transaction or produced (created) by a transaction.
- Commands are used to describe the operational and verification data that defines how a transaction can be verified.

The development environment – Corda

The development environment for Corda can be set up easily using the following steps. Required software includes the following:

- JDK 8 (8u131), which is available at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.
- IntelliJ IDEA Community edition, which is free and available at `https://www.jetbrains.com/idea/download`.
- H2 database platform independent ZIP, and is available at `http://www.h2database.com/html/download.html`.
- Git, which is available at `https://git-scm.com/downloads`.
- Kotlin language, which is available for IntelliJ, and more information can be found at `https://kotlinlang.org/`.

Gradle is another component that is used to build Corda. It is available at `https://gradle.org`.

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.

Summary

In this chapter, we have gone through an introduction to the Hyperledger project. Firstly, the core ideas behind the Hyperledger project were discussed, and a brief introduction to all projects under Hyperledger was provided. Three main Hyperledger projects were discussed in detail, namely Hyperledger Fabric, Sawtooth lake, and Corda. All these projects are continuously improving and changes are expected in the next releases. However, the core concepts of all the projects mentioned above are expected to remain unchanged or change only slightly. Readers are encouraged to visit the relevant links provided within the chapter to see the latest updates.

It is evident that a lot is going on in this space and projects like Hyperledger from the Linux Foundation are playing a pivotal role in the advancement of blockchain technology. Each of the projects discussed in this chapter has novel approaches towards solving the issues faced in various industries, and any current limitations within the blockchain technology are also being addressed, such as scalability and privacy. It is expected that more projects will soon be proposed to the Hyperledger project, and it is envisaged that with this collaborative and open effort blockchain technology will advance tremendously and will benefit the community as a whole.

In the next chapter, alternative blockchain solutions and platforms will be introduced. As blockchain technology is growing very fast and has attracted lot of research interest there are many new projects that have emerged recently. We will discuss those projects in the next chapter.

16
Alternative Blockchains

This chapter is intended to provide an introduction to alternative blockchain solutions. With the success of Bitcoin and subsequent realization of the potential of blockchain technology, a Cambrian explosion started that resulted in the development of various blockchain protocols, applications, and platforms. Some projects did not gain much traction, for example as an estimate 46% of ICOs have failed this year, but many have succeeded in creating a solid place in this space.

In this chapter, readers will be introduced to alternative blockchains and platforms such as Kadena, Ripple, and Stellar. We will explore the projects that either are new blockchains on their own or provide support to other existing blockchains by providing SDKs, frameworks, and tools to make development and deployment of blockchain solutions easier. The success of Ethereum and Bitcoin has resulted in various projects that spawned into existence by leveraging the underlying technologies and concepts introduced by them. These new projects add value by addressing the limitations in the current blockchains such as scalability and or enhancing the existing solutions by providing an additional layer of user-friendly tools on top of them.

Blockchains

In this section, an introduction to new blockchain solutions will be given, and later sections will cover various platforms and development kits that complement existing blockchains. For example, Kadena is a new private blockchain with novel ideas such as Scalable BFT. Various concepts such as sidechains, drivechains, and pegging have also been introduced with this growth of blockchain technologies. This chapter will cover all these technologies and related concepts in detail. Of course, it's not possible to cover all **alternative chains** (**altchains**) and platforms, but all those platforms have been included in this chapter that is related to blockchains, covered in the previous chapters, or are expected to gain traction soon.

We will explore Kadena, Ripple, Stellar, Quorum and various other blockchains in the section.

Kadena

Kadena is a private blockchain that has successfully addressed scalability and privacy issues in blockchain systems. A new Turing incomplete language, called Pact, has also been introduced with Kadena that allows the development of smart contracts. A key innovation in Kadena is its Scalable BFT consensus algorithm, which has the potential to scale to thousands of nodes without performance degradation.

Scalable BFT is based on the original Raft algorithm and is a successor of Tangaroa and Juno. Tangaroa, which is a name given to an implementation of Raft with fault tolerance (a BFT Raft), was developed to address the availability and safety issues that arose from the behavior of Byzantine nodes in the Raft algorithm, and Juno was a fork of Tangaroa that was developed by JPMorgan. Consensus algorithms are discussed in `Chapter 1`, *Blockchain 101* in more detail.

Both of these proposals have a fundamental limitation—they cannot scale while maintaining a high level of high performance. As such, Juno could not gain much traction. Private blockchains have the more desirable property of maintaining high performance as the number of nodes increase, but the aforementioned proposals lack this feature. Kadena solves this issue with its proprietary Scalable BFT algorithm, which is expected to scale up to thousands of nodes without any performance degradation.

Moreover, confidentiality is another significant aspect of Kadena that enables privacy of transactions on the blockchain. This security service is achieved by using a combination of key rotation, symmetric on-chain encryption, incremental hashing, and Double Ratchet protocol.

Key rotation is used as a standard mechanism to ensure the security of the private blockchain. It is used as a best practice to thwart any attacks if the keys have been compromised, by periodically changing the encryption keys. There is native support for key rotation in Pact smart contract language.

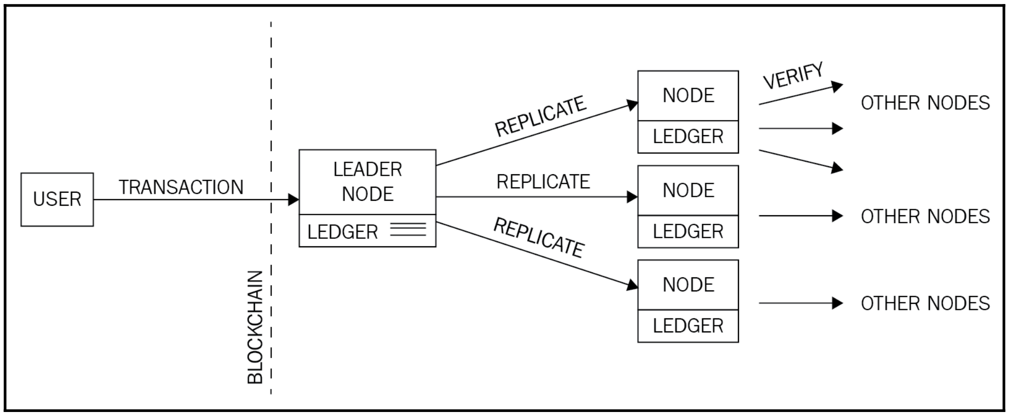
Symmetric on-chain encryption allows encryption of transaction data on the blockchain. These transactions can be automatically decrypted by the participants of a particular private transaction. Double Ratchet protocol is used to provide key management and encryption functions.

Scalable BFT consensus protocol ensures that adequate replication and consensus has been achieved before smart contract execution. The consensus is achieved by following the process described here.

This is how a transaction originates and flows in the network:

1. First, a new transaction is signed by the user and broadcasted over the blockchain network, which is picked up by a leader node that adds it to its immutable log. At this point, an incremental hash is also calculated for the log. Incremental hash is a type of hash function that allows computation of hash messages in the scenario where, if a previous original message which is already hashed is slightly changed, then the new hash message is computed from the already existing hash. This scheme is quicker and less resource intensive compared to a conventional hash function where an altogether new hash message is required to be generated even if the original message has only changed very slightly.
2. Once the transaction is written to the log by the leader node, it signs the replication and incremental hash and broadcasts it to other nodes.
3. Other nodes after receiving the transaction, verify the signature of the leader node, add the transaction into their own logs, and broadcast their own calculated incremental hashes (quorum proofs) to other nodes. Finally, the transaction is committed to the ledger permanently after an adequate number of proofs are received from other nodes.

A simplified version of this process is shown in the following diagram, where the leader node is recording the new transactions and then replicating them to the follower nodes:



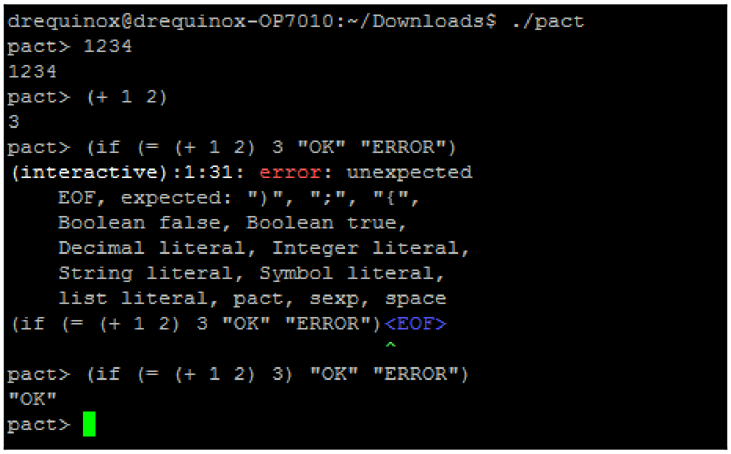
Consensus mechanism in Kadena

Once the consensus is achieved, a smart contract execution can start and takes a number of steps, as follows:

1. First, the signature of the message is verified.
2. Pact smart contract layer takes over.
3. Pact code is compiled.
4. The transaction is initiated and executes any business logic embedded within the smart contract. In case of any failures, an immediate rollback is initiated that reverts that state back to what it was before the execution started.
5. Finally, the transaction completes and relevant logs are updated.

Pact has been open sourced by Kadena and is available for download at `http://kadena.io/pact/downloads.html`.

This can be downloaded as a standalone binary that provides a REPL for Pact language. An example is shown here where Pact is run by issuing the `./pact` command in Linux console:



```
drequinox@drequinox-OP7010:~/Downloads$ ./pact
pact> 1234
1234
pact> (+ 1 2)
3
pact> (if (= (+ 1 2) 3 "OK" "ERROR")
(interactive):1:31: error: unexpected
    EOF, expected: ")", ";", "{",
    Boolean false, Boolean true,
    Decimal literal, Integer literal,
    String literal, Symbol literal,
    list literal, pact, sexp, space
(if (= (+ 1 2) 3 "OK" "ERROR")<EOF>
                               ^

pact> (if (= (+ 1 2) 3) "OK" "ERROR")
"OK"
pact>
```

Pact REPL, showing sample commands and error output

A smart contract in Pact language is usually composed of three sections: keysets, modules, and tables. These sections are described here:

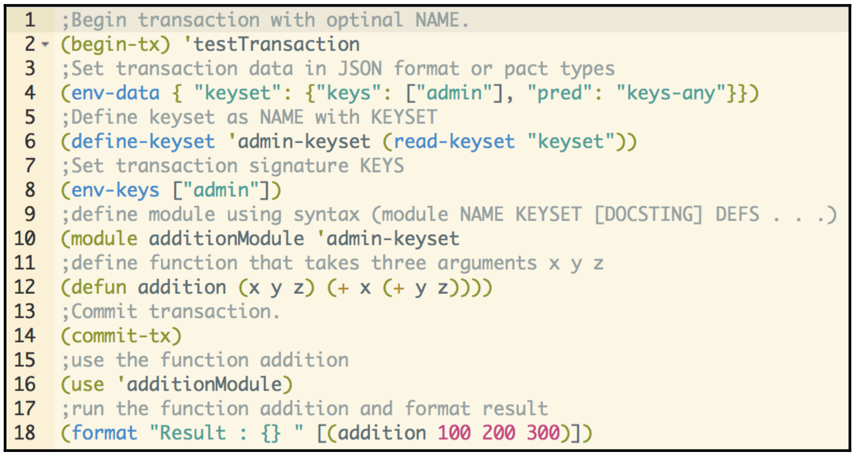
- **Keysets**: This section defines relevant authorization schemes for tables and modules.
- **Modules**: This section defines the smart contract code encompassing the business logic in the form of functions and pacts. Pacts within modules are composed of multiple steps and are executed sequentially.
- **Tables:** This section is an access-controlled construct defined within modules. Only administrators defined in the admin keyset have direct access to this table. Code within the module is granted full access, by default to the tables.

Pact also allows several execution modes. These modes include contract definition, transaction execution, and querying. These execution modes are described here:

- **Contract definition:** This mode allows a contract to be created on the blockchain via a single transaction message.
- **Transaction execution:** This mode entails the execution of modules of smart contract code that represent business logic.
- **Querying**: This mode is concerned with simply probing the contract for data and is executed locally on the nodes for performance reason. Pact uses LISP-like syntax and represents in the code exactly what will be executed on the blockchain, as it is stored on the blockchain in human-readable format. This is in contrast to Ethereum's EVM, which compiles into bytecode for execution, which makes it difficult to verify what code is in execution on the blockchain. Moreover, it is Turing incomplete, supports immutable variables, and does not allow null values, which improves the overall safety of the transaction code execution.

It is not possible to cover the complete syntax and functions of Pact in this limited length chapter; however, a small example is shown here, that shows the general structure of a smart contract written in Pact. This example shows a simple addition module that defines a function named `addition` that takes three parameters. When the code is executed it adds all three values and displays the result.

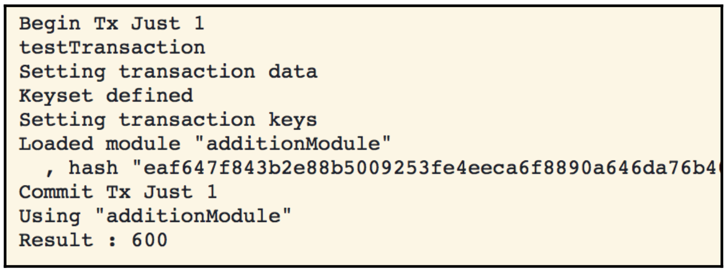
The following example has been developed using the online Pact compiler available at `http://kadena.io/try-pact/`.



```
;Begin transaction with optinal NAME.
(begin-tx) 'testTransaction
;Set transaction data in JSON format or pact types
(env-data { "keyset": {"keys": ["admin"], "pred": "keys-any"}})
;Define keyset as NAME with KEYSET
(define-keyset 'admin-keyset (read-keyset "keyset"))
;Set transaction signature KEYS
(env-keys ["admin"])
;define module using syntax (module NAME KEYSET [DOCSTING] DEFS . . .)
(module additionModule 'admin-keyset
;define function that takes three arguments x y z
(defun addition (x y z) (+ x (+ y z))))
;Commit transaction.
(commit-tx)
;use the function addition
(use 'additionModule)
;run the function addition and format result
(format "Result : {} " [(addition 100 200 300)])
```

Sample Pact code

When the code is run, it produces the output shown as follows:



```
Begin Tx Just 1
testTransaction
Setting transaction data
Keyset defined
Setting transaction keys
Loaded module "additionModule"
  , hash "eaf647f843b2e88b5009253fe4eeca6f8890a646da76b4
Commit Tx Just 1
Using "additionModule"
Result : 600
```

The output of the code

As shown in the preceding example, the execution output matches exactly with the code layout and structure, which allows for greater transparency and limits the possibility of malicious code execution.

Kadena is a new class of blockchains introducing the novel concept of **pervasive determinism** where, in addition to standard public/private key-based data origin security, an additional layer of fully deterministic consensus is also provided. It provides cryptographic security at all layers of the blockchain including transactions and consensus layer.

Relevant documentation and source code for Pact can be found here `https://github.com/kadena-io/pact`.

Kadena has also introduced a public blockchain in January, 2018 which is another leap forward in building blockchains with massive throughput. The novel idea in this proposal is to build a PoW parallel chain architecture. This scheme works by combining individually mined chains on peers into a single network. The result is massive throughput capable of processing more than 10,000 transactions per second.

The original research paper is available at `http://kadena.io/docs/chainweb-v15.pdf`.

Ripple

Introduced in 2012, Ripple is a currency exchange and real-time gross settlement system. In Ripple, the payments are settled without any waiting as opposed to traditional settlement networks, where it can take days for settlement.

It has a native currency called **Ripples** (**XRP**). It also supports non-XRP payments. This system is considered similar to an old traditional money transfer mechanism known as *Hawala*. This system works by making use of agents who take the money and a password from the sender, then contact the payee's agent and instruct them to release funds to the person who can provide the password. The payee then contacts the local agent, tells them the password and collects the funds. An analogy to the agent is gateway in Ripple. This is just a very simple analogy; the actual protocol is rather complex but principally it is the same.

The Ripple network is composed of various nodes that can perform different functions based on their type:

- **User nodes**: These nodes use in payment transactions and can pay or receive payments.
- **Validator nodes**: These nodes participate in the consensus mechanism. Each server maintains a set of unique nodes, which it needs to query while achieving consensus. Nodes in the **Unique Node List** (**UNL**) are trusted by the server involved in the consensus mechanism and will accept votes only from this list of unique nodes.

Ripple is sometimes not considered a truly decentralized network as there are network operators and regulators involved. However, it can be considered decentralized due to the fact that anyone can become part of the network by running a validator node. Moreover, the consensus process is also decentralized because any changes proposed to make on the ledger have to be decided by following a scheme of super majority voting. However, this is a hot topic among researchers and enthusiasts and there are arguments against and in favor of each school of thought. There are some discussions online that readers can refer to for further exploration of these ideas.

You can find these online discussions at the following links:

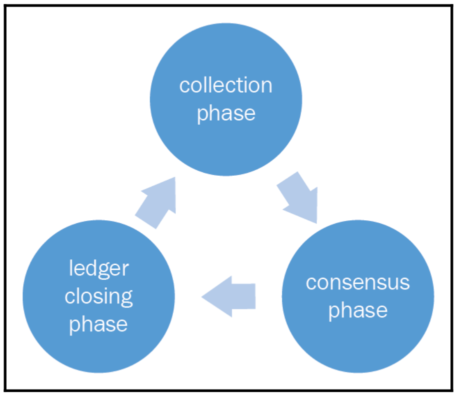
- `https://www.quora.com/Why-is-Ripple-centralized`
- `https://thenextweb.com/hardfork/2018/02/06/ripple-report-bitmex-centralized/`
- `https://www.reddit.com/r/Ripple/comments/6c8j7b/is_ripple_centralized_and_other_related_questions/?st=jewkor7bamp;sh=e39bc635`

Ripple maintains a globally distributed ledger of all transactions that are governed by a novel low-latency consensus algorithm called **Ripple Protocol Consensus Algorithm** (**RPCA**). The consensus process works by achieving an agreement on the state of an open ledger containing transactions by seeking verification and acceptance from validating servers in an iterative manner until an adequate number of votes are achieved. Once enough votes are received (a super majority, initially 50% and gradually increasing with each iteration up to at least 80%) the changes are validated and the ledger is closed. At this point, an alert is sent to the whole network indicating that the ledger is closed.

Original research paper for RPCA is available at `https://ripple.com/files/ripple_consensus_whitepaper.pdf`.

In summary, the consensus protocol is a three-phase process:

- **Collection phase**: In this phase validating nodes gather all transactions broadcasted on the network by account owners and validate them. Transactions, once accepted, are called candidate transactions and can be accepted or rejected based on the validation criteria.
- **Consensus phase**: After the collection phase the consensus process starts, and after achieving it the ledger is **closed**.
- **Ledger closing phase**: This process runs asynchronously every few seconds in rounds and, as result, the ledger is opened and closed (updated) accordingly:



Ripple consensus protocol phases

In a Ripple network, there are a number of components that work together in order to achieve consensus and form a payment network. These components are discussed individually here:

- **Server**: This component serves as a participant in the consensus protocol. Ripple server software is required in order to be able to participate in consensus protocol.
- **Ledger**: This is the main record of balances of all accounts on the network. A ledger contains various elements such as ledger number, account settings, transactions, timestamp, and a flag that indicates the validity of the ledger.

- **Last closed ledger**: A ledger is closed once consensus is achieved by validating nodes.
- **Open ledger**: This is a ledger that has not been validated yet and no consensus has been reached about its state. Each node has its own open ledger, which contains proposed transactions.
- **Unique Node List:** This is a list of unique trusted nodes that a validating server uses in order to seek votes and subsequent consensus.
- **Proposer**: As the name suggests, this component proposes new transactions to be included in the consensus process. It is usually a subset of nodes (UNL defined in the previous point) that can propose transactions to the validating server.

Transactions

Transactions are created by the network users in order to update the ledger. A transaction is expected to be digitally signed and valid in order for it to be considered as a candidate in the consensus process. Each transaction costs a small amount of XRP, which serves as a protection mechanism against denial of service attacks caused by spamming.

There are different types of transaction in the Ripple network. A single field within the Ripple transaction data structure called `TransactionType` is used to represent the type of the transaction. Transactions are executed by using a four-step process:

1. First, transactions are prepared whereby an unsigned transaction is created by following the standards
2. The second step is signing, where the transaction is digitally signed to authorize it
3. After this, the actual submission to the network occurs via the connected server
4. Finally, the verification is performed to ensure that the transaction is validated successfully

Roughly, the transactions can be categorized into three types, namely payments related, order related, and account and security related. All these types are described in the following section.

Payments related

There are several fields in this category that result in certain actions. All these fields are described as follows:

- `Payment`: This transaction is most commonly used and allows one user to send funds to another.
- `PaymentChannelClaim`: This is used to claim Ripples (XRP) from a payment channel. A payment channel is a mechanism that allows recurring and unidirectional payments between parties. This can also be used to set the expiration time of the payment channel.
- `PaymentChannelCreate`: This transaction creates a new payment channel and adds XRP to it in *drops*. A single drop is equivalent to 0.000001 of an XRP.
- `PaymentChannelFund`: This transaction is used to add more funds to an existing channel. Similar to the `PaymentChannelClaim` transaction, this can also be used to modify the expiration time of the payment channel.

Order related

This type of transaction includes following two fields:

- `OfferCreate`: This transaction represents a limit order, which represents an intent for the exchange of currency. It results in creating an offer node in the consensus ledger if it cannot be completely fulfilled.
- `OfferCancel`: This is used to remove a previously created offer node from the consensus ledger, indicating withdrawal of the order.

Account and security-related

This type of transaction includes the fields listed as follows. Each field is responsible for performing a certain function:

- `AccountSet`: This transaction is used to modify the attributes of an account in the Ripple consensus ledger.
- `SetRegularKey`: This is used to change or set the transaction signing key for an account. An account is identified using a base-58 Ripple address derived from the account's master public key.
- `SignerListSet`: This can be used to create a set of signers for use in multisignature transactions.
- `TrustSet`: This is used to create or modify a trust line between accounts.

A transaction in Ripple is composed of various fields that are common to all transaction types. These fields are listed as follows with a description:

- `Account`: This is the address of the initiator of the transaction.
- `AccountTxnID`: This is an optional field which contains the hash of another transaction. It is used to chain the transaction together.
- `Fee`: This is the amount of XRP.
- `Flags`: This is an optional field specifying the flags for the transaction.
- `LastLedgerSequence`: This is the highest sequence number of the ledger in which the transaction can appear.
- `Memos`: This represents optional arbitrary information.
- `SigningPubKey`: This represents the public key.
- `Signers`: This represent signers in a multisig transaction.
- `SourceTag`: This represents either sender or reason of the transaction.
- `SourceTag`: This represents either sender or reason of the transaction.
- `TxnSignature`: This is the verification digital signature for the transaction.

Interledger

Interledger is a simple protocol that is composed of four layers: Application, Transport, Interledger, and Ledger. Each layer is responsible for performing various functions under certain protocols. These functions and protocols are described in the following section.

The specifications of this protocol are available at: `https://interledger.org/rfcs/0003-interledger-protocol/draft-9.html`

Application layer

Protocols running on this layer govern the key attributes of a payment transaction. Examples of application layer protocols include **Simple Payment Setup Protocol** (**SPSP**) and **Open Web Payment Scheme** (**OWPS**). SPSP is an Interledger protocol that allows secure payment across different ledgers by creating connectors between them. OWPS is another scheme that allows consumer payments across different networks.

Once the protocols on this layer have run successfully, protocols from the transport layer will be invoked in order to start the payment process.

Transport layer

This layer is responsible for managing payment transactions. Protocols such as **Optimistic Transport Protocol** (**OTP**), **Universal Transport Protocol** (**UTP**) and **Atomic Transport Protocol** (**ATP**) are available currently for this layer. OTP is the simplest protocol, which manages payment transfers without any escrow protection, whereas UTP provides escrow protection. ATP is the most advanced protocol, which not only provides an escrowed transfer mechanism but in addition, makes use of trusted notaries to further secure the payment transactions.

Interledger layer

This layer provides interoperability and routing services. This layer contains protocols such as **Interledger Protocol** (**ILP**), **Interledger Quoting Protocol** (**ILQP**), and **Interledger Control Protocol** (**ILCP**). ILP packet provides the final target (destination) of the transaction in a transfer. ILQP is used in making quote requests by the senders before the actual transfer. ILCP is used to exchange data related to routing information and payment errors between connectors on the payment network.

Ledger layer

This layer contains protocols that enable communication and execution of payment transactions between connectors. **Connectors** are basically objects that implement the protocol for forwarding payments between different ledgers. It can support various protocols such as simple ledger protocol, various blockchain protocols, legacy protocols, and different proprietary protocols.

Ripple connect consists of various Plug and Play modules that allow connectivity between ledgers by using the ILP. It enables the exchange of required data between parties before the transaction, visibility, fee management, delivery confirmation, and secure communication using transport layer security. A third-party application can connect to the Ripple network via various connectors that forward payments between different ledgers.

All layers described in the preceding sections make up the architecture of Interledger Protocol. Overall, Ripple is a solution that is targeted for the financial industry and makes real-time payments possible without any settlement risk. As this is a very feature-rich platform, covering all aspects of it are not possible in this chapter.

Ripple documentation for the platform are available at `https://ripple.com/`.

Stellar

Stellar is a payment network based on blockchain technology and a novel consensus model called **Federated Byzantine Agreement** (**FBA**). FBA works by creating quorums of trusted parties. **Stellar Consensus Protocol** (**SCP**) is an implementation of FBA.

Key issues identified in the Stellar whitepaper are the cost and complexity of current financial infrastructure. This limitation warrants the need for a global financial network that addresses these issues without compromising the integrity and security of the financial transaction. This requirement has resulted in the invention of SCP which is a provably safe consensus mechanism.

Original research paper for SCP is available at `https://www.stellar.org/papers/stellar-consensus-protocol.pdf`.

It has four main properties, which are described here:

- **Decentralized control**: This allows participation by anyone without any central party
- **Low latency**: This addresses the much-desired requirement of fast transaction processing
- **Flexible trust**: This allows users to choose which parties they trust for a specific purpose
- **Asymptotic security**: This makes use of digital signatures and hash functions for providing the required level of security on the network

The Stellar network allows transfer and representation of the value of an asset by its native digital currency, called **Lumens,** abbreviated as **XLM**. Lumens are consumed when a transaction is broadcasted on the network, which also serves as a deterrent against denial of service attacks.

At its core, the Stellar network maintains a distributed ledger that records every transaction and is replicated on each Stellar server (node). The consensus is achieved by verifying transactions between servers and updating the ledger with updates. The Stellar ledger can also act as a distributed exchange order book by allowing users to store their offers to buy or sell currencies.

There are various tools, SDKs, and software that make up the Stellar network.

The core software is available at `https://github.com/stellar/stellar-core`.

Rootstock

Before discussing **Rootstock** (**RSK**) in detail, it's important to define and introduce some concepts that are fundamental to the design of Rootstock. These concepts include sidechains, drivechains, and two-way pegging. The concept of the sidechain was originally developed by Blockstream.

Blockstream's online presence is at `https://blockstream.com`.

Two-way pegging is a mechanism by which value (coins) can transfer between one blockchain to another and vice versa. There is no real transfer of coin between chains. The idea revolves around the concept of locking the same amount and value of coins in a bitcoin blockchain (main chain) and unlocking the equivalent number of tokens in the secondary chain.

Bearing this definition in mind, sidechains can be defined as described in the following section.

Sidechain

This is a blockchain that runs in parallel with a main blockchain and allows transfer of value between them. This means that tokens from one blockchain can be used in the sidechain and vice versa. This is also called a pegged sidechain because it supports two-way pegged assets.

Drivechain

This is a relatively new concept, where control on unlocking the locked bitcoins (in main chain) is given to the miners who can vote when to unlock them. This is in contrast to sidechains, where consensus is validated though simple payment verification mechanism in order to transfer the coins back to the main chain.

Rootstock is a smart contract platform which has a two-way peg into bitcoin blockchain. The core idea is to increase the scalability and performance of the bitcoin system and enable it to work with smart contracts. Rootstock runs a Turing complete deterministic virtual machine called **Rootstock Virtual Machine** (**RVM**). It is also compatible with the EVM and allows solidity-compiled contracts to run on Rootstock. Smart contracts can also run under the time-tested security of bitcoin blockchain. The Rootstock blockchain works by merge mining with bitcoins. This allows Rootstock blockchain to achieve the same security level as Bitcoin. This is especially true for preventing double spends and achieving settlement finality. It allows scalability, up to 400 transactions per second due to faster block times and other design considerations.

The research paper is available at, should you want to explore it further `https://uploads.strikinglycdn.com/files/ec5278f8-218c-407a-af3c-ab71a910246d/RSK%20White%20Paper%20-%20Overview.pdf`.

RSK has released the main network called Bamboo, RSK MainNet which is a beta currently.

It is available at `http://www.rsk.co/`.

Quorum

This is a blockchain solution built by enhancing the existing Ethereum blockchain. There are several enhancements such as transaction privacy and a new consensus mechanism that has been introduced in Quorum. Quorum has introduced a new consensus model known as QuorumChain, which is based on a majority voting and time-based mechanism. Another feature called Constellation is also introduced which is a general-purpose mechanism for submitting information and allows encrypted communication between peers. Furthermore, permissions at node level is governed by smart contracts. It also provides a higher level of performance compared to public Ethereum blockchains.

Several components make up the Quorum blockchain ecosystem. These are listed in the following subsections.

Transaction manager

This component enables access to encrypted transaction data. It also manages local storage on nodes and communication with other transaction managers on the network.

Crypto Enclave

As the name suggests, this component is responsible for providing cryptographic services to ensure transaction privacy. It is also responsible for performing key management functions.

QuorumChain

This is the key innovation in Quorum. It is a BFT consensus mechanism which allows verification and circulation of votes via transactions on the blockchain network. In this scheme, a smart contract is used to manage the consensus process and nodes can be given voting rights to vote on which new block should be accepted. Once an appropriate number of votes is received by the voters, the block is considered valid. Nodes can have two roles, namely Voter or Maker. The **Voter** node is allowed to vote, whereas the **Maker** node is the one that creates a new block. By design, a node can have either right, none, or only one.

Network manager

This component provides an access control layer for the permissioned network.

A node in the quorum network can take several roles, for example, a Maker node that is allowed to create new blocks. Transaction privacy is provided using cryptography and the concept that certain transactions are meant to be viewable only by their relevant participants. This idea is similar to Corda's idea of private transactions that was discussed in `Chapter 15`, *Hyperledger*. As it allows both public and private transactions on the blockchain, the state database has been divided into two databases representing private and public transactions. As such, there are two separate Patricia-Merkle trees that represent the private and public state of the network. A private contract state hash is used to provide consensus evidence in private transactions between transacting parties.

Transaction in a Quorum network consists of various elements such as the recipient, the digital signature of the sender, which is used to identify the transaction originator, optional Ether amount, the optional list of participants that are allowed to see the transaction, and a field that contains a hash in case of private transactions.

A transaction goes through several steps before it can reach its destination. These steps are described as follows in detail:

1. User applications (DApps) send the transaction to the Quorum node via an API exposed by the blockchain network. This also contains the recipient address and transaction data.
2. The API then encrypts the payload and applies any other necessary cryptographic algorithm in order to ensure the privacy of the transaction and is sent to the transaction manager. The hash of the encrypted payload is also calculated at this step.
3. After receiving the transaction, the transaction manager validates the signature of the transaction sender and stores the message.
4. The hash of the previously encrypted payload is sent to the Quorum node.
5. Once the Quorum node starts to validate a block that contains the private transaction, it requests more relevant data from the transaction manager.
6. Once this request is received by the transaction manager, it sends the encrypted payload and relevant symmetric keys to the requestor Quorum node.
7. Once the Quorum node has all the data, it decrypts the payload and sends it to the EVM for execution. This is how Quorum achieves privacy with symmetric encryption on the blockchain, while it is able to use native Ethereum protocol and EVM for message transfer and execution respectively.

A similar concept, but quite different in a few aspects, has been proposed before in the form of **HydraChain**, which is based on Ethereum blockchain and allows the creation of permissioned distributed ledgers.

Quorum is available for download at `https://github.com/jpmorganchase/quorum`.

Tezos

Tezos is a generic self-amending cryptographic ledger, which means that it not only allows decentralized consensus on the state of the blockchain but also allows consensus on how the protocol and nodes are evolved over time. Tezos has been developed to address limitations in the Bitcoin protocol such as issues arising from hard forks, cost, and mining power centralization due to PoW, limited scripting ability, and security issues. It has been developed in a purely functional language called OCaml.

The original research paper is available at `https://www.tezos.com/static/papers/white_paper.pdf`.

The architecture of Tezos distributed ledger is divided into three layers: the network layer, consensus layer, and transaction layer. This decomposition allows the protocol to be evolved in a decentralized fashion. For this purpose, a generic network shell is implemented in Tezos that is responsible for maintaining the blockchain, which is represented by a combination of consensus and transaction layer. This shell provides an interface layer between the network and the protocol.

A concept of seed protocol has also been introduced, which is used as a mechanism to allow stakeholders on the network to approve any changes to the protocol.

Tezos blockchain starts from a seed protocol compared to a traditional blockchain that starts from a genesis block.

This seed protocol is responsible for defining procedures for amendments in the blockchain and even the amendment protocol itself. The reward mechanism in Tezos is based on a PoS algorithm, therefore there is no mining requirement.

Contract script language has been developed in Tezos for writing smart contracts, which is a stack-based Turing complete language. Smart contracts in Tezos are formally verifiable, which allows the code to be mathematically proven for its correctness.

Tezos has recently completed crowdfunding via ICO of 232 million USD. Their public network is due to be released in Q1 2018.

Tezos code is available at `https://github.com/tezos/tezos`.

Storj

Existing models for cloud-based storage are all centralized solutions, which may or may not be as secure as users expect them to be. There is a need to have a cloud storage system that is secure, highly available, and above all decentralized. Storj aims to provide blockchain based, decentralized, and distributed storage. It is a cloud shared by the community instead of a central organization. It allows execution of storage contracts between nodes that act as autonomous agents. These agents (nodes) execute various functions such as data transfer, validation, and perform data integrity checks.

The core concept is based on **Distributed Hash Tables** (**DHTs**) called **Kademlia**, however this protocol has been enhanced by adding new message types and functionalities in Storj. It also implements a peer to peer **publish/subscribe** (**pub/sub**) mechanism known as **Quasar**, which ensures that messages successfully reach the nodes that are interested in storage contracts. This is achieved via a bloom filter-based storage contract parameters selection mechanism called **topics**.

Storj stores files in an encrypted format spread across the network. Before the file is stored on the network, it is encrypted using AES-256-CTR symmetric encryption and is then stored piece by piece in a distributed manner on the network. This process of dissecting the file into pieces is called **sharding** and results in increased availability, security, performance, and privacy of the network. Also, if a node fails the shard is still available because by default a single shard is stored at three different locations on the network.

It maintains a blockchain, which serves as a shared ledger and implements standard security features such as public/private key cryptography and hash functions similar to any other blockchain. As the system is based on hard drive sharing between peers, anyone can contribute by sharing their extra space on the drive and get paid with Storj's own cryptocurrency called **Storjcoin X** (**SJCX**). SJCX was developed as a *Counterparty* asset and makes use of Counterparty (Bitcoin blockchain based) for transactions. This has been migrated to Ethereum now.

A detailed discussion is available at `https://blog.storj.io/post/158740607128/migration-from-counterparty-to-ethereum`.
Storj code is available at `https://github.com/Storj/`.

MaidSafe

This is another distributed storage system similar to Storj. Users are paid in Safecoin for their storage space contribution to the network. This mechanism of payment is governed by *proof of resource*, which ensures that the disk space committed by a user to the network is available, if not then the payment of Safecoin will drop accordingly. The files are encrypted and divided into small portions before being transmitted on to the network for storage.

Another concept of **opportunistic caching** has been introduced with MaidSafe, which is a mechanism to create copies of frequently accessed data physically closer to where the access requests are coming from, which results in high performance of the network. Another novel feature of the SAFE network is that it automatically removes any duplicate data on the network, thus resulting in reduced storage requirements.

Moreover, the concept of **churning** has also been introduced, which basically means that data is constantly moved across the network so that the data cannot be targeted by malicious adversaries. It also keeps multiple copies of data across the network to provide redundancy in case a node goes offline or fails.

BigchainDB

This is a scalable blockchain database. It is not strictly a blockchain itself but complements blockchain technology by providing a decentralized database. At its core it's a distributed database but with the added attributes of a blockchain such as decentralization, immutability, and handling of digital assets. It also allows usage of NoSQL for querying the database.

It is intended to provide a database in a decentralized ecosystem where not only processing is decentralized (blockchain) or the filesystem is decentralized (for example, IPFS) but the database is also decentralized. This makes the whole application ecosystem decentralized.

This is available at `https://www.bigchaindb.com/`.

MultiChain

MultiChain has been developed as a platform for the development and deployment of private blockchains. It is based on bitcoin code and addresses security, scalability, and privacy issues. It is a highly configurable blockchain platform that allows users to set different blockchain parameters. It supports control and privacy via a granular permissioning layer. Installation of MultiChain is very quick.

Link to installation files are available at `http://www.multichain.com/download-install/`.

Tendermint

Tendermint is a software that provides a BFT consensus mechanism and state machine replication functionality to an application. Its main motivation is to develop a general purpose, secure, and high-performance replicated state machine.

There are two components in Tendermint, which are described in the following section.

Tendermint Core

This is a consensus engine that enables secure replication of transactions on each node in the network.

Tendermint Socket Protocol (TMSP)

This is an application interface protocol that allows interfacing with any programming language to process transactions.

Tendermint allows decoupling of the application process and consensus process, which allows any application to benefit from the consensus mechanism.

The Tendermint consensus algorithm is a round-based mechanism where validator nodes propose new blocks in each round. A locking mechanism is used to ensure protection against a scenario where two different blocks are selected for committing at the same height of the blockchain. Each validator node maintains a full local replicated ledger of blocks that contain transactions. Each block contains a header, which consists of the previous block hash, timestamp of the proposal of block, the current block height, and the Merkle root hash of all transactions present in the block.

Tendermint has recently been used in **Cosmos** (`https://cosmos.network`) which is a network of blockchains that allows interoperability between different chains running on BFT consensus algorithm. Blockchains on this network are called zones. The first zone in Cosmos is called Cosmos hub, which is, in fact, a public blockchain and is responsible for providing connectivity service to other blockchains. For this purpose, the hub makes use of **Inter Blockchain Communication** (**IBC**) protocol. IBC protocol supports two types of transactions called `IBCBlockCimmitTx` and `IBCPacketTx`. The first type is used to provide proof of the most recent block hash in a blockchain to any party, whereas the latter type is used to provide data origin authentication. A packet from one blockchain to another is published by first posting a proof to the target chain. The receiving (target) chain checks this proof in order to verify that the sending chain has indeed published the packet. In addition, it has its own native currency called Atom. This scheme addresses scalability and interoperability issues by allowing multiple blockchains to connect to the hub.

Tendermint is available at `https://tendermint.com/`.

Platforms and frameworks

This section covers various platforms that have been developed to enhance the experience of existing blockchain solutions.

Eris

Eris is not a single blockchain, it is an open modular platform developed by Monax for development of blockchain-based ecosystem applications. It offers various frameworks, SDKs, and tools that allow accelerated development and deployment of blockchain applications.

The core idea behind the Eris application platform is to enable development and management of ecosystem applications with a blockchain backend. It allows integration with multiple blockchains and enables various third-party systems to interact with various other systems.

This platform makes use of smart contracts written in Solidity language. It can interact with blockchains such as Ethereum or Bitcoin. The interaction can include connectivity commands, start, stop, disconnection, and creation of new blockchains. Complexity related to setup and interaction with blockchains have been abstracted away in Eris. All commands are standardized for different blockchains, and the same commands can be used across the platform regardless of the blockchain type being targeted.

An ecosystem application can consist the Eris platform, enabling the API gateway to allow legacy applications to connect to key management systems, consensus engines, and application engines. The Eris platform provides various toolkits that are used to provide various services to the developers. These modules are described as follows:

- **Chains**: This allows the creation of and interaction with blockchains.
- **Packages**: This allows the development of smart contracts.
- **Keys**: This is used for key management and signing operations.
- **Files**: This allows working with distributed data management systems. It can be used to interact with filesystems such as IPFS and data lakes.
- **Services**: This exposes a set of services that allows the management and integration of ecosystem applications.

Several SDKs has also been developed by Eris that allow the development and management of ecosystem applications. These SDKs contain smart contracts that have been fully tested and address specific needs and requirements of business. For example, a finance SDK, insurance SDK, and logistics SDK. There is also a base SDK that serves as a basic development kit to manage the life cycle of an ecosystem application.

Monax has developed its own permissioned blockchain client called `eris:db`. It is a PoS-based blockchain system that allows integration with a number of different blockchain networks. The `eris:db` client consists of four components:

- **Consensus**: This is based on the Tendermint consensus mechanism, discussed before
- **Virtual machine**: Eris uses EVM, as such it supports Solidity compiled contracts
- **Permissions layer**: Being a permissioned ledger, Eris provides an access control mechanism that can be used to assign specific roles to different entities on the network

- **Interface**: This provides various command-line tools and RPC interfaces to enable interaction with the backend blockchain network

The key difference between Ethereum blockchain and `eris:db` is that `eris:db` makes use of a **Practical Byzantine Fault-Tolerance** (**PBFT**) algorithm, which is implemented as a deposit-based Proof of Stake (DPOS system) whereas Ethereum uses PoW. Moreover, `eris:db` uses the ECDSA `ed22519` curve scheme whereas Ethereum uses the `secp256k1` algorithm. Finally, it is permissioned with an access control layer on top whereas Ethereum is a public blockchain.

Eris is a feature-rich application platform that offers a large selection of toolkits and services to develop blockchain-based applications.

It is available at `https://monax.io/`.

Summary

This chapter started with the introduction of alternative blockchains and is divided into two main sections discussing blockchains and platforms. Blockchain technology is a very thriving area, as such changes are quite rapid in existing solutions and new relevant technologies or tools are being introduced almost every day. In this chapter, a careful selection of platforms and blockchains was introduced. Several solutions were considered that complement material covered in previous chapters, for example, Eris, which supports blockchain development. New blockchains such as Kadena, various new protocols such as Ripple, and concepts such as sidechains and drivechains were also discussed.

The material covered in this chapter is intended to provide a strong foundation for more in-depth research into areas that readers are interested in. As said before, blockchain is a very fast-moving field, and there are many other blockchain proposals projects such as Tau-Chain, HydraChain, Elements, CREDITS, and many more that have not been discussed in this chapter. Readers are encouraged to keep an eye on the developments in this field to keep themselves up to date with advancement in this rapidly growing area.

In the next chapter, we will explore that how blockchain can be used out of its original usage, that is, cryptocurrencies. We will cover various use cases and especially usage of blockchain in IoT.

17
Blockchain – Outside of Currencies

Digital currencies were the first-ever application of blockchain technology, arguably without realizing its real potential. With the invention of Bitcoin, the concept of blockchain was introduced for the very first time, but it was not until 2013 that the true potential of blockchain technology was realized with its possible application in many different industries, other than cryptocurrencies. Since then many use cases of blockchain technology in various industries have been proposed, including but not limited to finance, the Internet of Things, digital rights management, government, and law.

In this chapter, four main industries namely the Internet of Things, government, health, and finance, have been selected, with the aid of use cases, for discussion.

In 2010, discussion started regarding BitDNS, a decentralized naming system for domains on the internet. Then Namecoin (`https://wiki.namecoin.org/index.php?title=History`) started in April 2011 with a different vision as compared to Bitcoin whose sole purpose is to provision electronic cash. This can be considered first example of blockchain usage other than purely cryptocurrencies.
After this by 2013, many ideas emerged. Since 2013 this trend is growing exponentially.

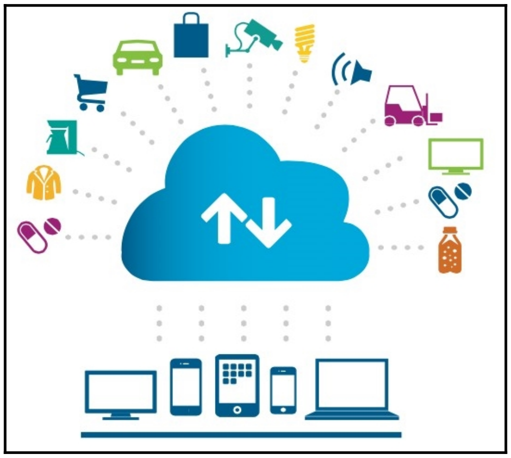
Internet of Things

The **Internet of Things** (**IoT**) for short has recently gained much traction due to its potential for transforming business applications and everyday life. IoT can be defined as a network of computationally intelligent physical objects (any object such as cars, fridges, industrial sensors, and so on) that are capable of connecting to the internet, sensing real-world events or environments, reacting to those events, collecting relevant data, and communicating it over the internet.

This simple definition has enormous implications and has led to exciting concepts, such as wearables, smart homes, smart grids, smart connected cars, and smart cities, that are all based on this basic concept of an IoT device. After dissecting the definition of IoT, four functions come to light as being performed by an IoT device. These include **sensing**, **reacting**, **collecting**, and **communicating**. All these functions are performed by using various components on the IoT device.

Sensing is performed by sensors. Reacting or controlling is performed by actuators, the collection is a function of various sensors, and communication is performed by chips that provide network connectivity. One thing to note is that all these components are accessible and controllable via the internet in the IoT. An IoT device on its own is perhaps useful to some extent, but if it is part of a broader IoT ecosystem, it is more valuable.

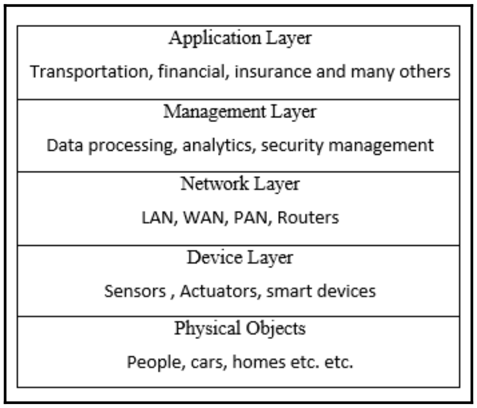
A typical IoT can consist of many physical objects connecting with each other and to a centralized cloud server. This is shown in the following diagram:



A typical IoT network

Source: IBM

Elements of IoT are spread across multiple layers, and various reference architectures exist that can be used to develop IoT systems. A five-layer model can be used to describe IoT, which contains a physical object layer, device layer, network layer, services layer, and application layer. Each layer or level is responsible for various functions and includes multiple components. These are shown in the following diagram:


Application Layer
Transportation, financial, insurance and many others
Management Layer
Data processing, analytics, security management
Network Layer
LAN, WAN, PAN, Routers
Device Layer
Sensors , Actuators, smart devices
Physical Objects
People, cars, homes etc. etc.

IoT five-layer model

Now we will examine each layer in detail.

Physical object layer

These include any real-world physical objects. It includes people, animals, cars, trees, fridges, trains, factories, homes, and in fact anything that is required to be monitored and controlled can be connected to the IoT.

Device layer

This layer contains things that make up the IoT such as sensors, transducers, actuators, smartphones, smart devices, and **Radio-Frequency Identification** (**RFID**) tags. There can be many categories of sensors such as body sensors, home sensors, and environmental sensors based on the type of work they perform. This layer is the core of an IoT ecosystem where various sensors are used to sense real-world environments. This layer includes sensors that can monitor temperature, humidity, liquid flow, chemicals, air, pressure, and much more. Usually, an **Analog to Digital Converter** (**ADC**) is required on a device to turn the real-world analog signal into a digital signal that a microprocessor can understand.

Actuators in this layer provide the means to enable control of external environments, for example, starting a motor or opening a door. These components also require digital to analog converters to convert a digital signal into analog. This method is especially relevant when control of a mechanical component is required by the IoT device.

Network layer

This layer is composed of various network devices that are used to provide Internet connectivity between devices and to the cloud or servers that are part of the IoT ecosystem. These devices can include gateways, routers, hubs, and switches. This layer can include two types of communication.

First there is the horizontal means of communication, which includes radio, Bluetooth, Wi-Fi, Ethernet, LAN, Zigbee, and PAN and can be used to provide communication between IoT devices. Second, we have communication to the next layer, which is usually through the internet and provides communication between machines and people or other upper layers. The first layer can optionally be included in the device layer as it physically is residing on the device layer where devices can communicate with each other at the same layer.

Management layer

This layer provides the management layer for the IoT ecosystem. This includes platforms that enable processing of data gathered from the IoT devices and turn that into meaningful insights. Also, device management, security management, and data flow management are included in this layer. It also manages communication between the device and application layers.

Application layer

This layer includes applications running on top of the IoT network. This layer can consist of many applications depending on the requirements such as transportation, healthcare, financial, insurance, or supply chain management. This list, of course, is not an exhaustive list by any stretch of the imagination; there is a myriad of IoT applications that can fall into this layer.

With the availability of cheap sensors, hardware, and bandwidth, IoT has gained popularity in recent years and currently has applications in many different areas including healthcare, insurance, supply chain management, home automation, industrial automation, and infrastructure management. Moreover, advancements in technology such as the availability of IPv6, smaller and powerful processors, and better internet access have also played a vital role in the popularity of IoT.

The benefits of IoT range from cost saving to enabling businesses to make vital decisions and thus improve performance based on the data provided by the IoT devices. Even in domestic usage IoT equipped home appliances can provide valuable data for cost saving. For example, smart meters for energy monitoring can provide valuable information on how the energy is being used and can convey that back to the service provider. Raw data from millions of things (IoT devices) is analyzed and provides meaningful insights that help in making timely and efficient business decisions.

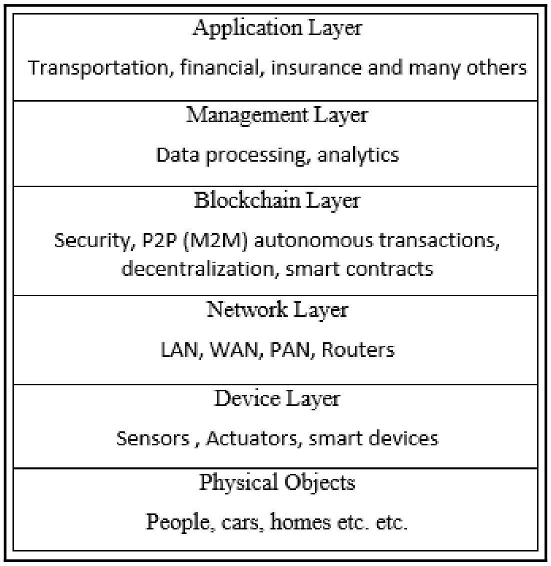
The usual IoT model is based on a centralized paradigm where IoT devices usually connect to a cloud infrastructure or central servers to report and process the relevant data back. This centralization poses certain possibilities of exploitation including hacking and data theft. Moreover, not having control of personal data on a single, centralized service provider also increases the possibility of security and privacy issues. While there are methods and techniques to build a highly secure IoT ecosystem based on the normal IoT model, there are specific much more desirable benefits that blockchain can bring to IoT. A blockchain-based IoT model differs from the traditional IoT network paradigm.

According to IBM, blockchain for IoT can help to build trust, reduce costs, and accelerate transactions. Additionally, decentralization, which is at the very core of blockchain technology, can eliminate single points of failure in an IoT network. For example, a central server perhaps is not able to cope with the amount of data that billions of IoT devices (things) are producing at high frequency. Also, the peer-to-peer communication model provided by blockchain can help to reduce costs because there is no need to build high-cost centralized data centers or implementation of complex public key infrastructure for security. Devices can communicate with each other directly or via routers.

As an estimate of various researchers and companies, by 2020 there will be roughly 22 billion devices connected to the internet. With this explosion of billions of devices connecting to the internet, it is hard to imagine that centralized infrastructures will be able to cope with the high demands of bandwidth, services, and availability without incurring excessive expenditure. Blockchain-based IoT will be able to solve scalability, privacy, and reliability issues in the current IoT model.

Blockchain enables *things* to communicate and transact with each other directly and with the availability of smart contracts, negotiation, and financial transactions can also occur directly between the devices instead of requiring an intermediary, authority, or human intervention. For example, if a room in a hotel is vacant, it can rent itself out, negotiate the rent, and can open the door lock for a human who has paid the right amount of funds. Another example could be that if a washing machine runs out of detergent, it could order it online after finding the best price and value based on the logic programmed in its smart contract.

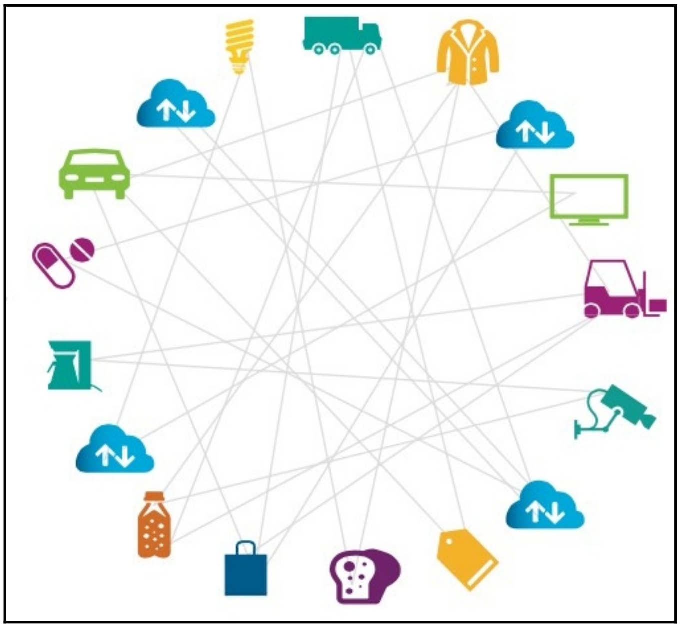
The aforementioned five-layer IoT model can be adapted to a blockchain-based model by adding a blockchain layer on top of the network layer. This layer will run smart contracts, and provide security, privacy, integrity, autonomy, scalability, and decentralization services to the IoT ecosystem. The management layer, in this case, can consist of only software related to analytics and processing, and security and control can be moved to the blockchain layer. This model can be visualized in the following diagram:


Application Layer
Transportation, financial, insurance and many others
Management Layer
Data processing, analytics
Blockchain Layer
Security, P2P (M2M) autonomous transactions, decentralization, smart contracts
Network Layer
LAN, WAN, PAN, Routers
Device Layer
Sensors , Actuators, smart devices
Physical Objects
People, cars, homes etc. etc.

Blockchain-based IoT model

In this model, other layers would perhaps remain the same, but an additional blockchain layer will be introduced as a middleware between all participants of the IoT network.

It can also be visualized as a peer-to-peer IoT network after abstracting away all the layers mentioned earlier. This model is shown in the following diagram where all devices are communicating and negotiating with each other without a central command and control entity:



Blockchain-based direct communication model, source: IBM

It can also result in cost saving which is due to easier device management by using a blockchain based decentralized approach. The IoT network can be optimized for performance by using blockchain. In this case, there will be no need to store IoT data centrally for millions of devices because storage and processing requirements can be distributed to all IoT devices on the blockchain. This can result in completely removing the need for large data centers for processing and storing the IoT data.

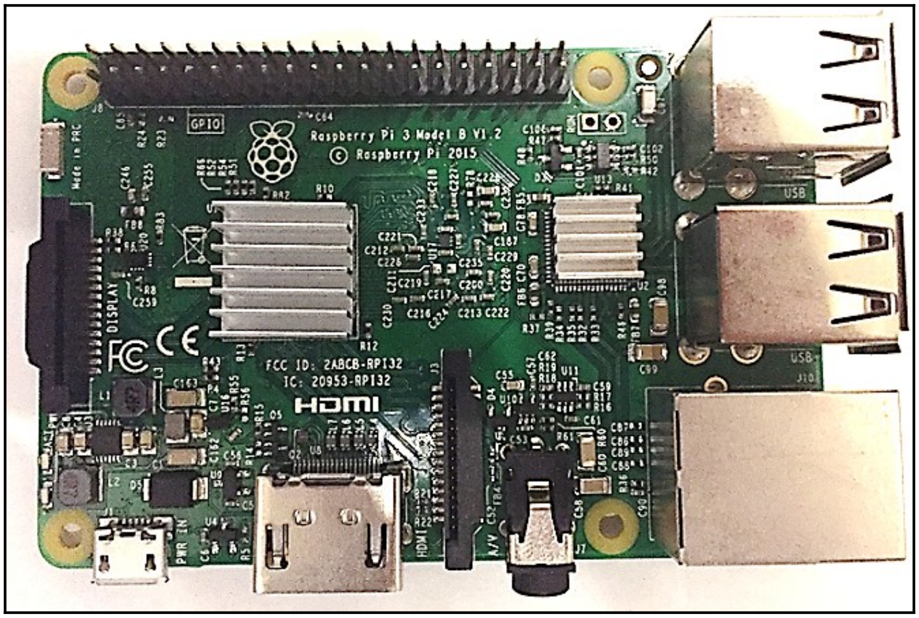
Blockchain-based IoT can also thwart denial of service attacks where hackers can target a centralized server or data center more efficiently, but with blockchain's distributed and decentralized nature, such attacks are no longer possible. Additionally, if as estimated there will be billions of devices connected to the internet soon, it will become almost impossible to manage security and updates of all those devices from traditional centrally-owned servers. Blockchain can provide a solution to this problem by allowing devices to communicate with each other directly in a secure manner and even request firmware and security updates from each other. On a blockchain network, these communications can be recorded immutably and securely which will provide auditability, integrity, and transparency to the system. This mechanism is not possible with traditional peer-to-peer systems.

In summary, there are clear benefits that can be reaped with the convergence of IoT and blockchain and a lot of research and work in academia and industry are already in progress. There are various projects already proposed providing blockchain-based IoT solutions. For example, IBM Blue Horizon and IBM Bluemix are IoT platforms supporting blockchain IoT platforms. Various start-ups such as Filament have already proposed novel ideas on how to build a decentralized network that enables devices on IoT to transact with each other directly and autonomously driven by smart contracts.

In the following section, a practical example is provided on how to build a simple IoT device and connect it to the Ethereum blockchain. This IoT device is connected to the Ethereum blockchain and is used to open a door (in this case the door lock is represented by an LED) when the appropriate amount of funds is sent by a user on the blockchain. This is a simple example and requires a more rigorously-tested version to implement it in production, but it demonstrates how an IoT device can be connected, controlled, and responded to in response to certain events on an Ethereum blockchain.

IoT blockchain experiment

This example makes use of a Raspberry Pi device which is a **Single Board Computer** (**SBC**). The Raspberry Pi is a SBC developed as a low-cost computer to promote computer education but has also gained much more popularity as a tool of choice for building IoT platforms. A Raspberry Pi 3 Model B is shown in the following picture. You may be able to use earlier models too, but those have not been tested:



Raspberry Pi Model B

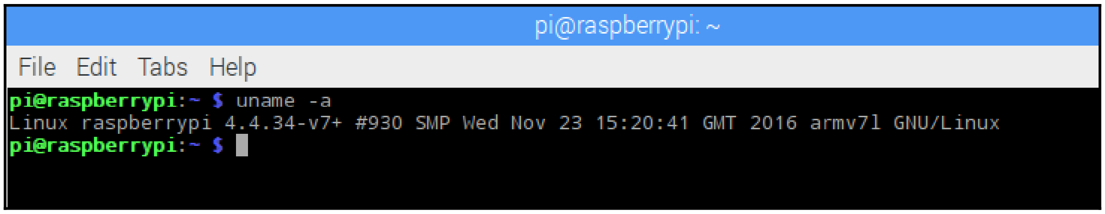
In the following section, an example will be discussed where a Raspberry Pi will be used as an IoT device connected to the Ethereum blockchain and will act in response to a smart contract invocation.

First, the Raspberry Pi needs to be set up. This can be done by using NOOBS which provides an easy method of installing Raspbian or any other operating system.

This can be downloaded and installed from the link `https://www.raspberrypi.org/downloads/noobs/`.
Alternatively, only Raspbian can be installed from the link `https://www.raspberrypi.org/downloads/raspbian/`.
Another alternative available at `https://github.com/debian-pi/raspbian-ua-netinst` can also be used to install a minimal non-GUI version of Raspbian OS.

For this example, NOOBS has been used to install Raspbian, as such the rest of the exercise assumes Raspbian is installed on the SD memory card of the Raspberry Pi. The command output in the following screenshot shows that which architecture the operating system is running on. In this case, it is `armv71`; therefore ARM-compatible binary for Geth will be downloaded.

The platform can be confirmed by running the command `uname -a` in a terminal window in Raspberry Pi Raspbian operating system.



```
pi@raspberrypi:~ $ uname -a
Linux raspberrypi 4.4.34-v7+ #930 SMP Wed Nov 23 15:20:41 GMT 2016 armv7l GNU/Linux
pi@raspberrypi:~ $
```

Raspberry Pi architecture

Once the Raspbian operating system is installed, the next step is to download the appropriate Geth binary for the Raspberry Pi ARM platform.

The download and installation steps are described in detail:

1. Geth download. Note that in the following example a specific version is downloaded however other versions are available which can be downloaded from `https://geth.ethereum.org/downloads/`.

   We can use `wget`, to download the `geth` client images:

```
    $ wget https://gethstore.blob.core.windows.net/builds/geth-linux-
arm7-1.5.6-2a609af5.tar.gz
```

Other versions are also available, but it's recommended that you download this version, as this is the one that has been used in examples in this chapter.

2. Unzip and extract into a directory. The directory named `geth-linux-arm7-1.5.6-2a609af5` will be created automatically with the `tar` command shown next:

   ```
   $ tar -zxvf geth-linux-arm7-1.5.6-2a609af5.tar
   ```

   This command will create a directory named `geth-linux-arm7-1.5.6-2a609af5` and will extract the Geth binary and related files into that directory. The Geth binary can be copied into `/usr/bin` or the appropriate path on Raspbian to make it available from anywhere in the operating system. When the download is finished, the next step is to create the genesis block.

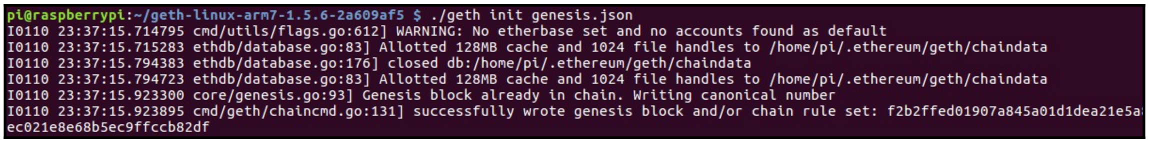
3. The same genesis block needs to be used that was created previously in `Chapter 12`, *Ethereum Development Environment*. The genesis file can be copied from the other node on the network. This is shown in the following screenshot. Alternatively, an entirely new genesis block can be generated.

```
      {
            "nonce": "0x0000000000000042",
            "timestamp": "0x00",
            "parentHash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
            "extraData": "0x00",
            "gasLimit": "0x8000000",
            "difficulty": "0x0400",
            "mixhash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
            "coinbase": "0x3333333333333333333333333333333333333333",
            "alloc": {
            },
            "config": {
                "chainId": 786,
                "homesteadBlock": 0,
                "eip155Block": 0,
                "eip158Block": 0
            }
      }
```

4. Once the `genesis.json` file is copied onto the Raspberry Pi; the following command can be run to generate the genesis block. It is important that the same genesis block is used that was generated previously otherwise the nodes will effectively be running on separate networks:

   ```
   $ ./geth init genesis.json
   ```

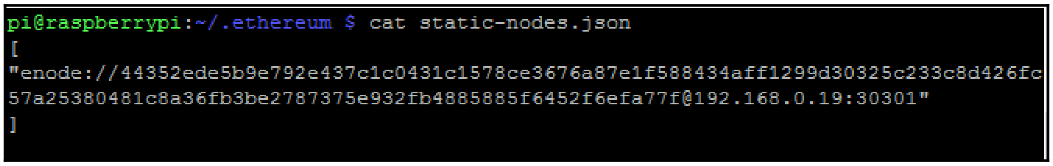
   This will show the output similar to the one shown in the following screenshot:



```
pi@raspberrypi:~/geth-linux-arm7-1.5.6-2a609af5 $ ./geth init genesis.json
I0110 23:37:15.714795 cmd/utils/flags.go:612] WARNING: No etherbase set and no accounts found as default
I0110 23:37:15.715283 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:37:15.794383 ethdb/database.go:176] closed db:/home/pi/.ethereum/geth/chaindata
I0110 23:37:15.794723 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:37:15.923300 core/genesis.go:93] Genesis block already in chain. Writing canonical number
I0110 23:37:15.923895 cmd/geth/chaincmd.go:131] successfully wrote genesis block and/or chain rule set: f2b2ffed01907a845a01d1dea21e5a
ec021e8e68b5ec9ffccb82df
```

Initialize genesis file

5. After genesis block creation, there is a need to add peers to the network. This can be achieved by creating a file named `static-nodes.json`, which contains the enode ID of the peer that `geth` on the Raspberry Pi will connect for syncing:



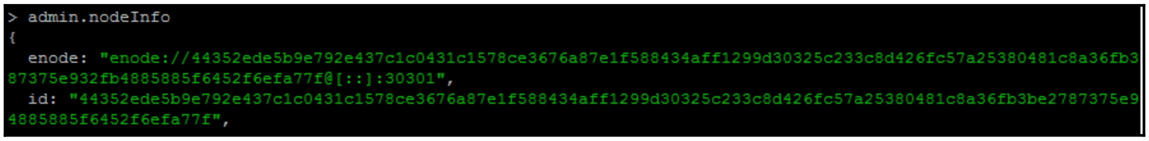
```
pi@raspberrypi:~/.ethereum $ cat static-nodes.json
[
"enode://44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc
57a25380481c8a36fb3be2787375e932fb4885885f6452f6efa77f@192.168.0.19:30301"
]
```

Static nodes configuration

   This information can be obtained from the Geth JavaScript console by running the following command, and this command should be run on the peer to which Raspberry Pi is going to connect:

   ```
   > admin.nodeInfo
   ```

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



```
> admin.nodeInfo
{
  enode: "enode://44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3
87375e932fb4885885f6452f6efa77f@[::]:30301",
  id: "44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3be2787375e9
4885885f6452f6efa77f",
```

geth nodeInfo

After this step, further instructions presented in the following sections can be followed to connect Raspberry Pi to the other node on the private network. In the example, the Raspberry Pi will be connected to the network ID `786` created in `Chapter 12`, *Ethereum Development Environment*. The key is to use the same genesis file created previously and different port numbers. Same genesis file will ensure that clients connect to the same network in which the genesis file originated from.

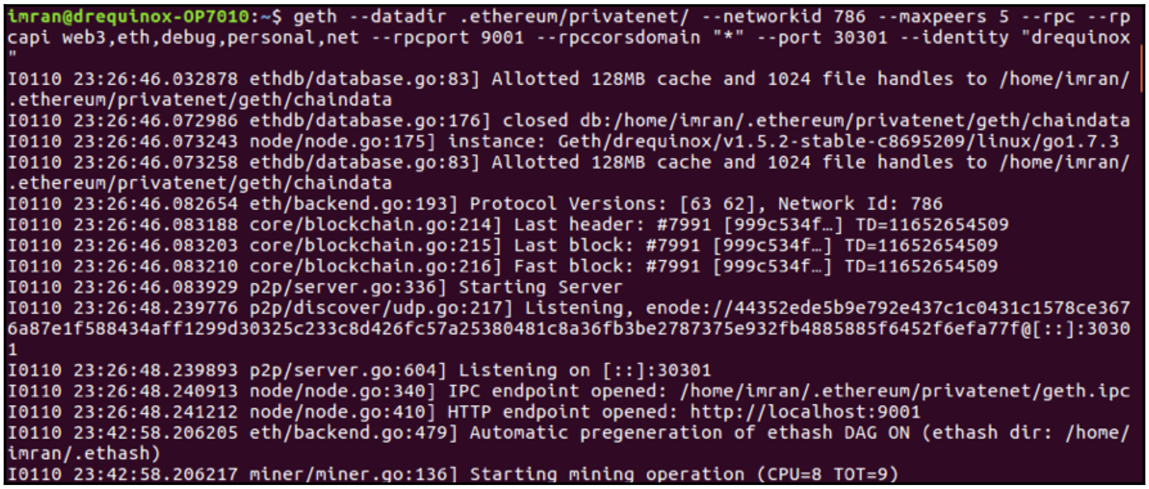
Different ports are not a strict requirement, however, if the two nodes are running under a private network and access from an environment external to the network is required then a combination of DMZ, router and port forwarding will be used. Therefore, it is recommended to use different TCP ports to allow port forwarding to work correctly. The `--identity` switch shown in the following command for first node set up, which hasn't been introduced previously, allows for an identifying name to be specified for the node.

First node setup

First, the `geth` client needs to be started on the first node using the following command:

```
$ geth --datadir .ethereum/privatenet/ --networkid 786 --maxpeers 5 --rpc -
-rpcapi web3,eth,debug,personal,net --rpcport 9001 --rpccorsdomain "*" --
port 30301 --identity "drequinox"
```

This will give the output similar to the following:



```
imran@drequinox-OP7010:~$ geth --datadir .ethereum/privatenet/ --networkid 786 --maxpeers 5 --rpc --rp
capi web3,eth,debug,personal,net --rpcport 9001 --rpccorsdomain "*" --port 30301 --identity "drequinox
"
I0110 23:26:46.032878 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/imran/
.ethereum/privatenet/geth/chaindata
I0110 23:26:46.072986 ethdb/database.go:176] closed db:/home/imran/.ethereum/privatenet/geth/chaindata
I0110 23:26:46.073243 node/node.go:175] instance: Geth/drequinox/v1.5.2-stable-c8695209/linux/go1.7.3
I0110 23:26:46.073258 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/imran/
.ethereum/privatenet/geth/chaindata
I0110 23:26:46.082654 eth/backend.go:193] Protocol Versions: [63 62], Network Id: 786
I0110 23:26:46.083188 core/blockchain.go:214] Last header: #7991 [999c534f…] TD=11652654509
I0110 23:26:46.083203 core/blockchain.go:215] Last block: #7991 [999c534f…] TD=11652654509
I0110 23:26:46.083210 core/blockchain.go:216] Fast block: #7991 [999c534f…] TD=11652654509
I0110 23:26:46.083929 p2p/server.go:336] Starting Server
I0110 23:26:48.239776 p2p/discover/udp.go:217] Listening, enode://44352ede5b9e792e437c1c0431c1578ce367
6a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3be2787375e932fb4885885f6452f6efa77f@[::]:3030
1
I0110 23:26:48.239893 p2p/server.go:604] Listening on [::]:30301
I0110 23:26:48.240913 node/node.go:340] IPC endpoint opened: /home/imran/.ethereum/privatenet/geth.ipc
I0110 23:26:48.241212 node/node.go:410] HTTP endpoint opened: http://localhost:9001
I0110 23:42:58.206205 eth/backend.go:479] Automatic pregeneration of ethash DAG ON (ethash dir: /home/
imran/.ethash)
I0110 23:42:58.206217 miner/miner.go:136] Starting mining operation (CPU=8 TOT=9)
```

geth on first node

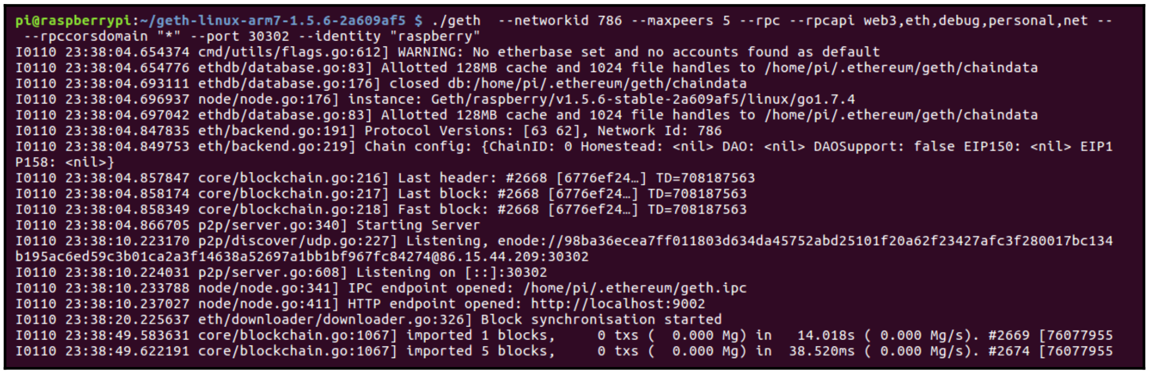
Once this is started up, it should be kept running, and another `geth` instance should be started from the Raspberry Pi node.

Raspberry Pi node setup

On Raspberry Pi, the following command is required to be run to start `geth` and to sync it with other nodes (in this case only one node). The following is the command:

```
$ ./geth --networkid 786 --maxpeers 5 --rpc --rpcapi
web3,eth,debug,personal,net --rpccorsdomain "*" --port 30302 --identity
"raspberry"
```

This should produce the output similar to the one shown in the following screenshot. When the output contains the row displaying `Block synchronisation started` it means that the node has connected successfully to its peer.



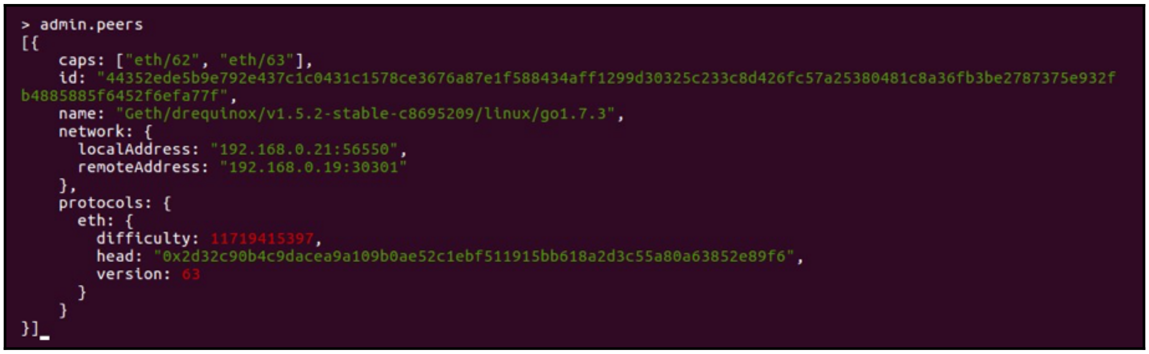
```
pi@raspberrypi:~/geth-linux-arm7-1.5.6-2a609af5 $ ./geth  --networkid 786 --maxpeers 5 --rpc --rpcapi web3,eth,debug,personal,net --
 --rpccorsdomain "*" --port 30302 --identity "raspberry"
I0110 23:38:04.654374 cmd/utils/flags.go:612] WARNING: No etherbase set and no accounts found as default
I0110 23:38:04.654776 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:38:04.693111 ethdb/database.go:176] closed db:/home/pi/.ethereum/geth/chaindata
I0110 23:38:04.696937 node/node.go:176] instance: Geth/raspberry/v1.5.6-stable-2a609af5/linux/go1.7.4
I0110 23:38:04.697042 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:38:04.847835 eth/backend.go:191] Protocol Versions: [63 62], Network Id: 786
I0110 23:38:04.849753 eth/backend.go:219] Chain config: {ChainID: 0 Homestead: <nil> DAO: <nil> DAOSupport: false EIP150: <nil> EIP1
P158: <nil>}
I0110 23:38:04.857847 core/blockchain.go:216] Last header: #2668 [6776ef24…] TD=708187563
I0110 23:38:04.858174 core/blockchain.go:217] Last block: #2668 [6776ef24…] TD=708187563
I0110 23:38:04.858349 core/blockchain.go:218] Fast block: #2668 [6776ef24…] TD=708187563
I0110 23:38:04.866705 p2p/server.go:340] Starting Server
I0110 23:38:10.223170 p2p/discover/udp.go:227] Listening, enode://98ba36ecea7ff011803d634da45752abd25101f20a62f23427afc3f280017bc134
b195ac6ed59c3b01ca2a3f14638a52697a1bb1bf967fc84274@86.15.44.209:30302
I0110 23:38:10.224031 p2p/server.go:608] Listening on [::]:30302
I0110 23:38:10.233788 node/node.go:341] IPC endpoint opened: /home/pi/.ethereum/geth.ipc
I0110 23:38:10.237027 node/node.go:411] HTTP endpoint opened: http://localhost:9002
I0110 23:38:20.225637 eth/downloader/downloader.go:326] Block synchronisation started
I0110 23:38:49.583631 core/blockchain.go:1067] imported 1 blocks,     0 txs (  0.000 Mg) in   14.018s ( 0.000 Mg/s). #2669 [76077955
I0110 23:38:49.622191 core/blockchain.go:1067] imported 5 blocks,     0 txs (  0.000 Mg) in  38.520ms ( 0.000 Mg/s). #2674 [76077955
```

geth on the Raspberry Pi.

This can be further verified by running commands in the `geth` console on both nodes as shown in the following screenshot. The `geth` client can be attached by simply running the following command on the Raspberry Pi:

```
$ geth attach
```

This will open the JavaScript `geth` console for interacting with the `geth` node. We can use `admin.peers` command to see the connected peers:



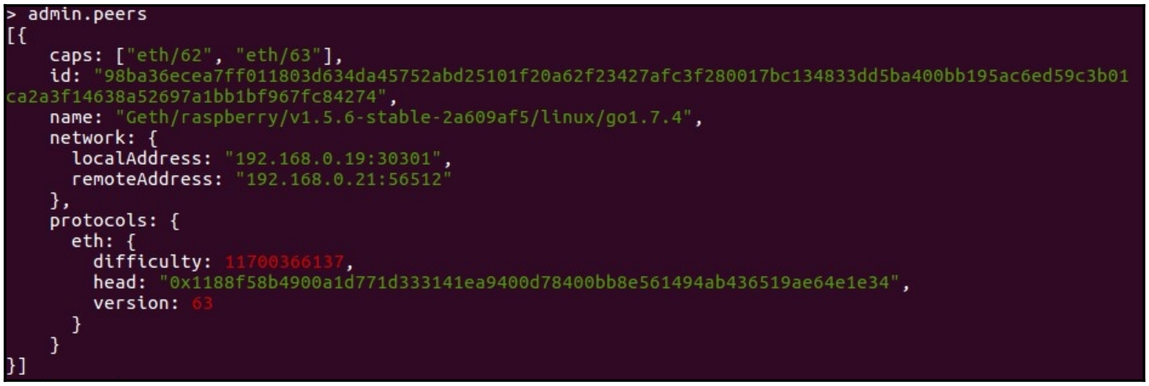
```
> admin.peers
[{
    caps: ["eth/62", "eth/63"],
    id: "44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3be2787375e932f
b4885885f6452f6efa77f",
    name: "Geth/drequinox/v1.5.2-stable-c8695209/linux/go1.7.3",
    network: {
      localAddress: "192.168.0.21:56550",
      remoteAddress: "192.168.0.19:30301"
    },
    protocols: {
      eth: {
        difficulty: 11719415397,
        head: "0x2d32c90b4c9dacea9a109b0ae52c1ebf511915bb618a2d3c55a80a63852e89f6",
        version: 63
      }
    }
}]_
```

geth console admin peers command running on Raspberry Pi

Similarly, we can attach to the `geth` instance by running the following command on the first node:

```
$ geth attach ipc:.ethereum/privatenet/geth.ipc
```

Once the console is available `admin.peers` can be run to reveal the details about other connected nodes as shown in the following screenshot:



```
> admin.peers
[{
    caps: ["eth/62", "eth/63"],
    id: "98ba36ecea7ff011803d634da45752abd25101f20a62f23427afc3f280017bc134833dd5ba400bb195ac6ed59c3b01
ca2a3f14638a52697a1bb1bf967fc84274",
    name: "Geth/raspberry/v1.5.6-stable-2a609af5/linux/go1.7.4",
    network: {
      localAddress: "192.168.0.19:30301",
      remoteAddress: "192.168.0.21:56512"
    },
    protocols: {
      eth: {
        difficulty: 11700366137,
        head: "0x1188f58b4900a1d771d333141ea9400d78400bb8e561494ab436519ae64e1e34",
        version: 63
      }
    }
}]
```

geth console admin peers command running on the other peer

Once both nodes are up-and-running further prerequisites can be installed to set up the experiment. Installation of Node.js and the relevant JavaScript libraries is required.

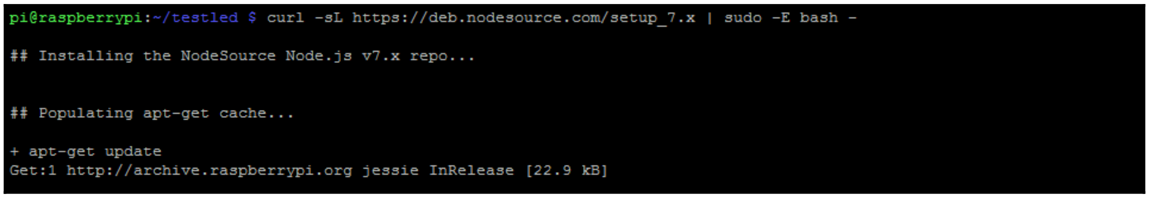
Installing Node.js

The required libraries and dependencies are listed here. First Node.js and npm need to be updated on the Raspberry Pi Raspbian operating system. For this the following steps can be followed:

1. Install latest Node.js on the Raspberry Pi using the following command:

   ```
   $ curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
   ```

   This should display output similar to the following. The output is quite large therefore only the top part of the output is shown in the following screenshot:



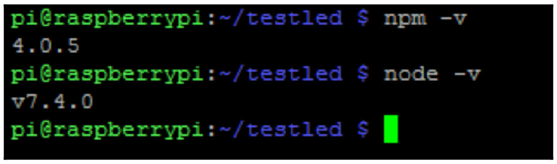
```
pi@raspberrypi:~/testled $ curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -

## Installing the NodeSource Node.js v7.x repo...


## Populating apt-get cache...

+ apt-get update
Get:1 http://archive.raspberrypi.org jessie InRelease [22.9 kB]
```

Node.js installation

2. Run the update via `apt-get`:

   ```
   $ sudo apt-get install nodejs
   ```

   Verification can be performed by running the following command to ensure that the correct versions of Node.js and npm are installed, as shown in the following screenshot:



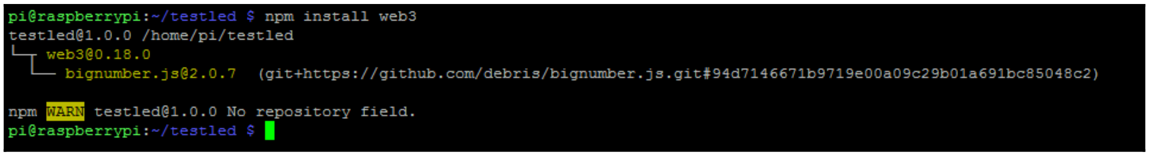
```
pi@raspberrypi:~/testled $ npm -v
4.0.5
pi@raspberrypi:~/testled $ node -v
v7.4.0
pi@raspberrypi:~/testled $
```

npm and node installation verification

It should be noted that these versions are not a necessity; any of the latest version of npm and Node.js should work. However, the examples in this chapter make use of npm 4.0.5 and node v7.4.0, so it is recommended that readers use the same version in order to avoid any compatibility issues.

3. Install Ethereum `web3` npm, which is required to enable JavaScript code to access the Ethereum blockchain:

   Make sure that specific version of `web3` shown in the screenshot is installed or a version similar to this for example 0.20.2. This is important because by default version 1.0.0-beta.26 (at the time of writing) will be installed which is beta and is under development. Therefore `web3` 0.20.2 or 0.18.0 stable version should be used for this example. Readers can install this version by using `$ npm install web3@0.20.2`.

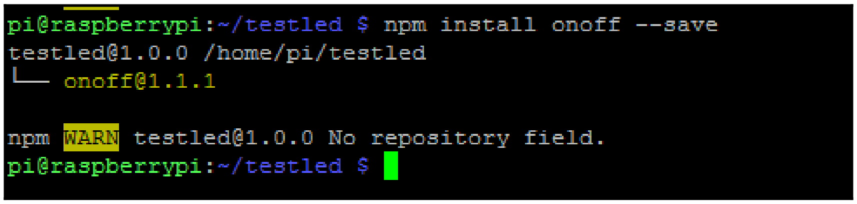


```
pi@raspberrypi:~/testled $ npm install web3
testled@1.0.0 /home/pi/testled
└┬ web3@0.18.0
  └── bignumber.js@2.0.7  (git+https://github.com/debris/bignumber.js.git#94d7146671b9719e00a09c29b01a691bc85048c2)

npm WARN testled@1.0.0 No repository field.
pi@raspberrypi:~/testled $
```

npm install web3

4. Similarly, npm `onoff` can be installed, which is required to communicate with the Raspberry Pi and control GPIO:

   ```
   $ npm install onoff
   ```



```
pi@raspberrypi:~/testled $ npm install onoff --save
testled@1.0.0 /home/pi/testled
└── onoff@1.1.1

npm WARN testled@1.0.0 No repository field.
pi@raspberrypi:~/testled $
```

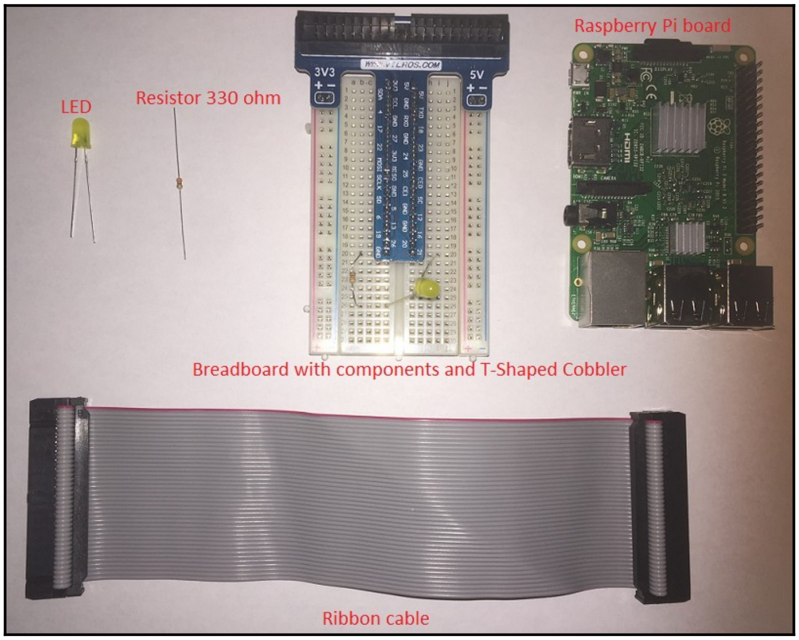
Onoff installation

When all the prerequisites are installed, hardware setup can be performed. For this purpose, a simple circuit is built using a breadboard and a few electronic components.

The hardware components are listed as follows:

- **LED**: The abbreviation of **Light Emitting Diode**, this can be used as a visual indication for an event.
- **Resistor**: A 330 ohm component is required which provides resistance to passing current based on its rating. It is not necessary to understand the theory behind it for this experiment; any standard electronics engineering text covers all these topics in detail.

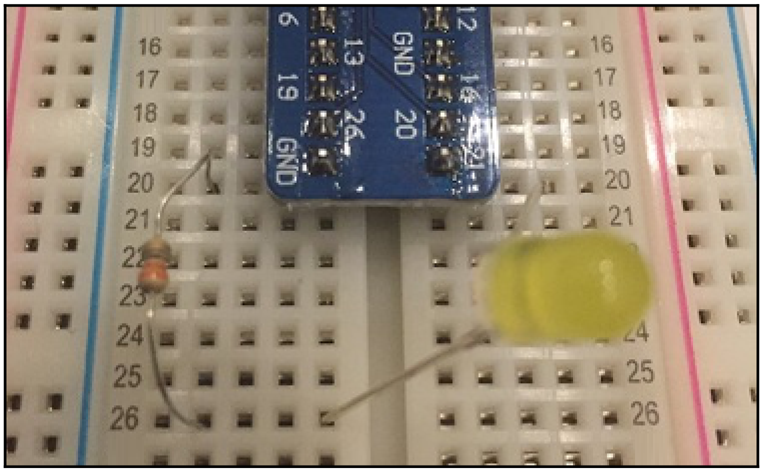
- **Breadboard**: This provides a means of building an electronic circuit without requiring soldering.
- **T-Shaped cobbler**: This is inserted on the breadboard as shown in the following photo and provides a labeled view of all **General Purpose I/O** (**GPIO**) pins for the Raspberry Pi.
- **Ribbon cable connector**: This is simply used to provide connectivity between the Raspberry Pi and the breadboard via GPIO. All these components are shown in the following picture:



Required components

Circuit

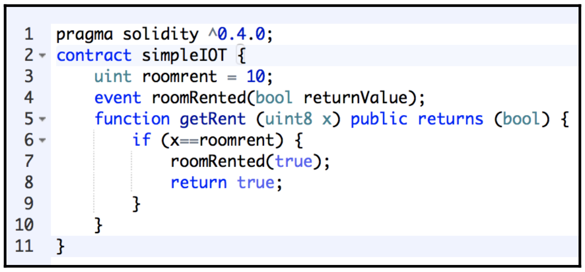
As shown in the following picture, the positive leg (long leg) of the LED is connected to pin number **21** of the GPIO, and the negative (short leg) is connected to the resistor, which is then connected to the **ground** (**GND**) pin of the GPIO. Once the connections are set up the ribbon cable can be used to connect to the GPIO connector on the Raspberry Pi simply.



Connections for components on the breadboard

Once the connections are set up correctly, and the Raspberry Pi has been updated with the appropriate libraries and Geth, the next step is to develop a simple, smart contract that expects a value. If the value provided to it is not what it expects it does not trigger an event; otherwise if the value passed matches the correct value, the event triggers which can be read by the client JavaScript program running via Node.js. Of course, the Solidity contract can be very complicated and can also deal with the Ether sent to it, and if the amount of Ether is equal to the required amount, then the event can trigger. However, in this example, the aim is to demonstrate the usage of smart contracts to trigger events that can then be read by JavaScript program running on Node.js, which then, in turn, can trigger actions on IoT devices using various libraries.

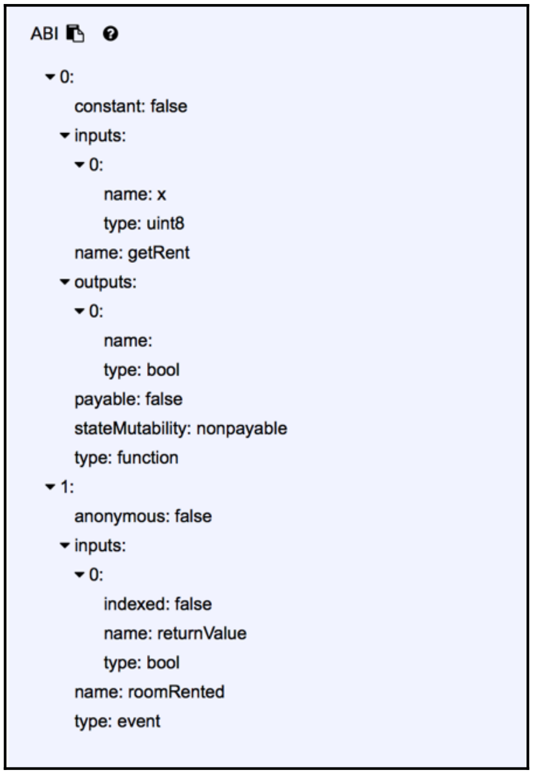
The smart contract source code is shown as follows:



```
pragma solidity ^0.4.0;
contract simpleIOT {
    uint roomrent = 10;
    event roomRented(bool returnValue);
    function getRent (uint8 x) public returns (bool) {
        if (x==roomrent) {
            roomRented(true);
            return true;
        }
    }
}
```

Solidity code for simple IOT

The online Solidity compiler (Remix IDE) can be used to run and test this contract. The **Application Binary Interface** (**ABI**) required for interacting with the contract is also available in the **Details** section as shown in the following screenshot:



```
ABI
▾ 0:
    constant: false
  ▾ inputs:
    ▾ 0:
        name: x
        type: uint8
    name: getRent
  ▾ outputs:
    ▾ 0:
        name:
        type: bool
    payable: false
    stateMutability: nonpayable
    type: function
▾ 1:
    anonymous: false
  ▾ inputs:
    ▾ 0:
        indexed: false
        name: returnValue
        type: bool
    name: roomRented
    type: event
```

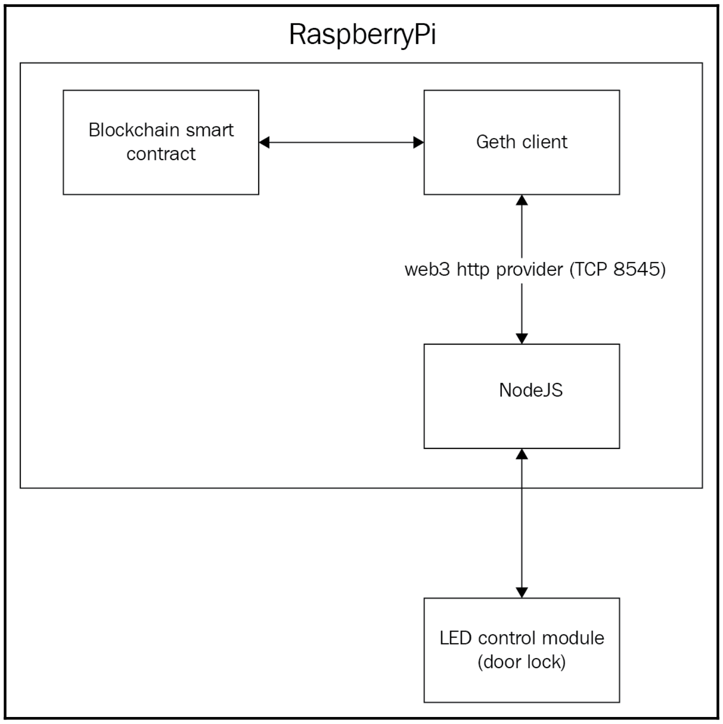
ABI from Remix IDE

The following is the ABI of the contract:

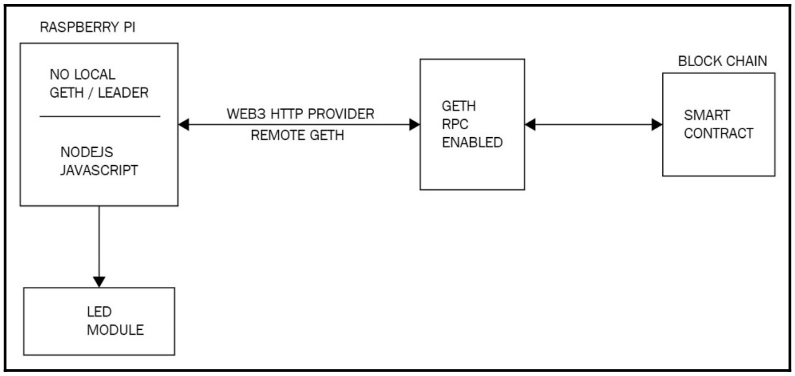
```
[
    {
        "constant": false,
        "inputs": [
            {
                "name": "x",
                "type": "uint8"
            }
        ],
        "name": "getRent",
        "outputs": [
            {
                "name": "",
                "type": "bool"
            }
        ],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": false,
                "name": "returnValue",
                "type": "bool"
            }
        ],
        "name": "roomRented",
        "type": "event"
    }
]
```

There are two methods by which the Raspberry Pi node can connect to the private blockchain via the `web3` interface. The first is where the Raspberry Pi device is running its own `geth` client locally and maintains its ledger, but with resource-constrained devices, it is not possible to run a full `geth` node or even a light node in a few circumstances. In that case, the second method, which uses `web3` provider can be used to connect to the appropriate RPC channel. This will be shown later in the client JavaScript Node.js program.

A comparison of both of these approaches is shown in the following diagram:


RaspberryPi
Blockchain smart contract
Geth client
web3 http provider (TCP 8545)
NodeJS
LED control module (door lock)

Application architecture of room rent IoT application (IoT device with local ledger)



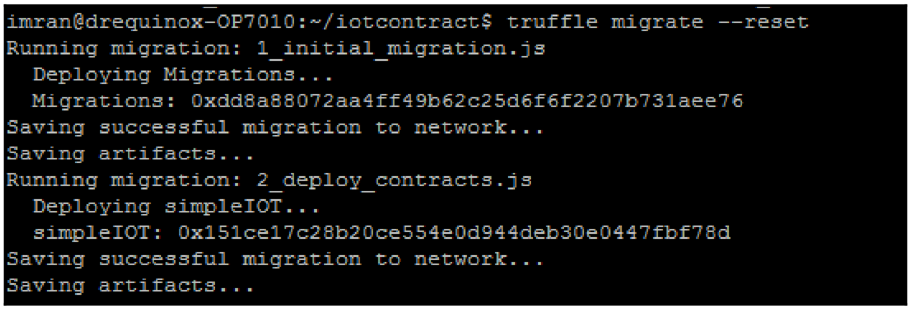
Application architecture of room rent IoT application (IoT device without local ledger)

There are obvious security concerns which arise from exposing RPC interfaces publicly; therefore, it is recommended that this option is used only on private networks and if required to be used on public networks appropriate security measures are put in place, such as allowing only the known IP addresses to connect to the `geth` RPC interface. This can be achieved by a combination of disabling peer discovery mechanisms and HTTP-RPC server listening interfaces.

More information about this can be found using `geth help`. The traditional network security measures such as firewalls, **Transport Layer Security** (**TLS**) and certificates can also be used but have not been discussed in this example. Now Truffle can be used to deploy the contract on the private network ID `786` to which at this point the Raspberry Pi is connected. Truffle deploy can be performed simply by using the following shown command; it is assumed that `truffle init` and other preliminaries discussed in `Chapter 12`, *Ethereum Development Environment* has already been performed:

```
$ truffle migrate
```

It should produce the output similar to the following screenshot:



```
imran@drequinox-OP7010:~/iotcontract$ truffle migrate --reset
Running migration: 1_initial_migration.js
  Deploying Migrations...
  Migrations: 0xdd8a88072aa4ff49b62c25d6f6f2207b731aee76
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying simpleIOT...
  simpleIOT: 0x151ce17c28b20ce554e0d944deb30e0447fbf78d
Saving successful migration to network...
Saving artifacts...
```

Truffle deploy

Once the contract is deployed correctly, JavaScript code can be developed that will connect to the blockchain via `web3`, listen for the events from the smart contract in the blockchain, and turn the LED on via the Raspberry Pi. The JavaScript code of the `index.js` file is shown as follows:

```
var Web3 = require('web3');
if (typeof web3 !== 'undefined')
{
    web3 = new Web3(web3.currentProvider);
}else
{
    web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:9002"));
    //http-rpc-port
}
var Gpio = require('onoff').Gpio;
var led = new Gpio(21,'out');
var coinbase = web3.eth.coinbase;
var ABIString =
'[{"constant":false,"inputs":[{"name":"x","type":"uint8"}],"name":"getRent"
,"outputs":[{"name":"","type":"bool"}],"payable":false,"stateMutability":"n
onpayable","type":"function"},{"anonymous":false,"inputs":[{"indexed":false
,"name":"returnValue","type":"bool"}],"name":"roomRented","type":"event"}]'
;
var ABI = JSON.parse(ABIString);
var ContractAddress = '0x975881c44fbef4573fef33cccec1777a8f76669c';
web3.eth.defaultAccount = web3.eth.accounts[0];
var simpleiot = web3.eth.contract(ABI).at(ContractAddress);
var event = simpleiot.roomRented( {}, function(error, result) { if (!error)
{
    console.log("LED On");
    led.writeSync(1);
}
});
```

Note that in the preceding example the contract address `'0x975881c44fbef4573fef33cccec1777a8f76669c'` for variable `var ContractAddress` is specific to the deployment and it will be different when readers run this example. Simply change the address in the file to what you see after deploying the contract.

Also, note that the HTTP-RPC server listening port on which Geth has been started on Raspberry Pi. By default, it is TCP port `8545`. Remember to change this according to your Raspberry Pi setup and Geth configuration. It is set to `9002` in the preceding example code because Geth running on Raspberry Pi is listening on `9002` in the example. If it's listening to a different port on your Raspberry Pi, then change it to that port:

```
web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:9002"));
```

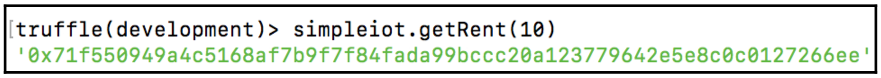
When Geth starts up it shown which port it has HTTP endpoint listening on. This is also configurable with `--rpcport` in `geth` by specifying the port number value as a parameter to the flag.

This JavaScript code can be placed in a file on the Raspberry Pi, for example, `index.js`. It can be run by using the following command:

```
$ node index.js
```

This will start the program, which will run on Node.js and listen for events from the smart contract. Once the program is running correctly, the smart contract can be invoked by using the Truffle console as shown in the following screenshot.

In this case, the `getRent` function is called with parameter `10`, which is the expected value:

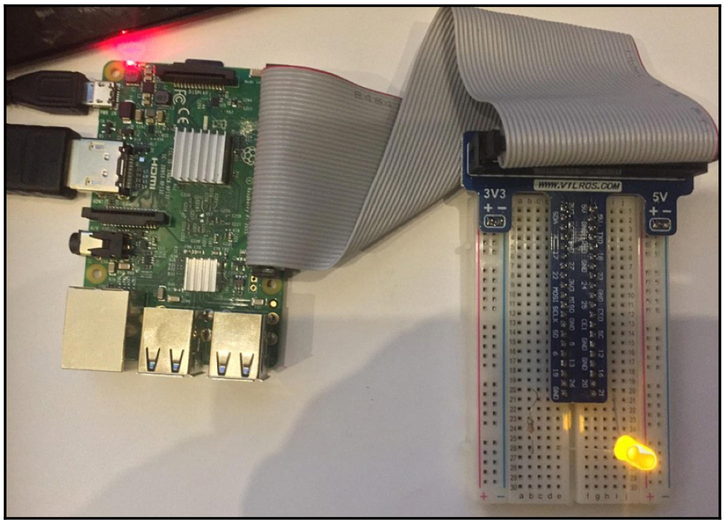


```
truffle(development)> simpleiot.getRent(10)
'0x71f550949a4c5168af7b9f7f84fada99bccc20a123779642e5e8c0c0127266ee'
```

Interaction with the contract

After the contract is mined, `roomRented` will be triggered, which will turn on the LED.

In this example, it is a simple LED, but it can be any physical device such as a room lock that can be controlled via an actuator. If all works well, the LED will be turned on as a result of the smart contract function invocation as shown in the following picture:



Raspberry Pi with LED control

Also, on node side it will display output similar to the one shown here:

```
$ node index.js
LED On
```

As demonstrated in the preceding example, a private network of IoT devices can be built that runs a `geth` client on each of the nodes and can listen for events from smart contracts and trigger an action accordingly. The example shown is simple on purpose but demonstrates the underlying principles of an Ethereum network that can be built using IoT devices along with smart contract-driven control of the physical devices.

In the next section, other applications of the blockchain technology in government, finance, and health will be discussed.

Government

There are various applications of blockchain being researched currently that can support government functions and take the current model of e-government to the next level. First, in this section, some background for e-government will be provided, and then a few use cases such as e-voting, homeland security (border control), and electronic IDs (citizen ID cards) will be discussed.

Government or electronic government is a paradigm where information and communication technology are used to deliver public services to citizens. The concept is not new and has been implemented in various countries around the world, but with blockchain, a new avenue of exploration has opened up. Many governments are researching the possibility of using blockchain technology for managing and delivering public services including but not limited to identity cards, driving licenses, secure data sharing among various government departments and contract management. Transparency, auditability, and integrity are attributes of blockchain that can go a long way in effectively managing various government functions.

Border control

Automated border control systems have been in use for decades now to thwart illegal entry into countries and prevent terrorism and human trafficking.

Machine-readable travel documents and specifically biometric passports have paved the way for automated border control; however current systems are limited to a certain extent and blockchain technology can provide solutions. A **Machine Readable Travel Document** (**MRTD**) standard is defined in document ICAO 9303 (`https://www.icao.int/publications/pages/publication.aspx?docnum=9303`) by the **International Civil Aviation Organization** (**ICAO**) and has been implemented by many countries around the world.

Each passport contains various security and identity attributes that can be used to identify the owner of the passport and also circumvent attempts at tampering with the passports. These include biometric features such as retina scan, fingerprints, facial recognition, and standard ICAO specified features including **Machine Readable Zone** (**MRZ**) and other text attributes that are visible on the first page of the passport.

One key issue with current border control systems is data sharing whereby the systems are controlled by a single entity and data is not readily shared among law enforcement agencies. This lack of the ability to share data makes it challenging to track suspected travel documents or individuals. Another issue is related to the immediate implementation of blacklisting of a travel document, for example, when there is an immediate need to track and control suspected travel documents. Currently, there is no mechanism available to blacklist or revoke a suspected passport immediately and broadcast it to the border control ports worldwide.

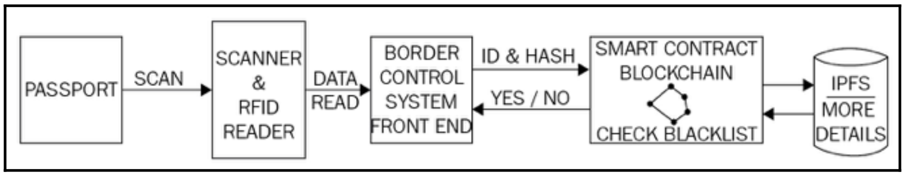
Blockchain can provide a solution to this problem by maintaining a blacklist in a smart contract which can be updated as required and any changes will be immediately visible to all agencies and border control points thus enabling immediate control over the movement of a suspected travel document. It could be argued that traditional mechanisms like PKIs and peer-to-peer networks can also be used for this purpose, but they do not provide the benefits that a blockchain can provide. With blockchain, the whole system can be simplified without the requirement of complex networks and PKI setups which will also result in cost reduction. Moreover, blockchain based systems will provide cryptographically guaranteed immutability which helps with auditing and discourages any fraudulent activity.

The full database of all travel documents perhaps cannot be stored on the blockchain currently due to scalability issues, but a backend distributed database such as BigchainDB, IPFS, or Swarm can be used for that purpose. In this case, a hash of the travel document with the biometric ID of an individual can be stored in a simple smart contract, and a hash of the document can then be used to refer to the detailed data available on the distributed filesystem such as IPFS. This way, when a travel document is blacklisted anywhere on the network, that information will be available immediately with the cryptographic guarantee of its authenticity and integrity throughout the distributed ledger. This functionality can also provide adequate support in anti-terrorism activities, thus playing a vital role in the homeland security function of a government.

A simple contract in Solidity can have an array defined for storing identities and associated biometric records. This array can be used to store the identifying information about a passport. The identity can be a hash of MRZ of the passport or travel document concatenated with the biometric record from the RFID chip. A simple Boolean field can be used to identify blacklisted passports. Once this initial check passes, further detailed biometric verification can be performed by traditional systems and eventually when a decision is made regarding the entry of the passport holder that decision can be propagated back to the blockchain, thus enabling all participants on the network to immediately share the outcome of the decision.

A high-level approach to building a blockchain-based border control system can be visualized as shown in the following diagram. In this scenario, the passport is presented for scanning to an RFID and page scanner which reads the data page and extracts machine-readable information along with a hash of the biometric data stored in the RFID chip. At this stage, a live photo and retina scan of the passport holder is also taken. This information is then passed on to the blockchain where a smart contract is responsible for verifying the legitimacy of the travel document by first checking its list of blacklisted passports and then requesting more data from the backend IPFS database for comparison. Note that the biometric data such as photo or retina scan is not stored on the blockchain, instead only a reference to this data in the backend (IPFS or BigchainDB) is stored in the blockchain.

If the data from the presented passport matches with what is held in the IPFS as files or in BigchainDB and also pass the smart contract logical check, then the border gate can be opened.



Automated border control using blockchain

After verification, this information is propagated throughout the blockchain and is instantly available to all participants on the border control blockchain. These participants can be a worldwide consortium of homeland security departments of various nations.

Voting

Voting in any government is a key function and allows citizens to participate in the democratic election process. While voting has evolved into a much more mature and secure process, it still has limitations that need to be addressed to achieve a desired level of maturity. Usually, the limitations in current voting systems revolve around fraud, weaknesses in operational processes, and especially transparency. Over the years, secure voting mechanisms (machines) have been built which make use of specialized voting machines that promised security and privacy, but they still had vulnerabilities that could be exploited to subvert the security mechanisms of those machines. These vulnerabilities can lead to serious implications for the whole voting process and can result in mistrust in the government by the public.

Blockchain-based voting systems can resolve these issues by introducing end-to-end security and transparency in the process. Security is provided in the form of integrity and authenticity of votes by using public key cryptography which comes as standard in a blockchain. Moreover, immutability guaranteed by blockchain ensures that votes cast once cannot be cast again. This can be achieved through a combination of biometric features and a smart contract maintaining a list of votes already cast. For example, a smart contract can maintain a list of already casted votes with the biometric ID (for example a fingerprint) and can use that to detect and prevent double casting. Secondly, **Zero-Knowledge Proofs** (**ZKPs**) can also be used on the blockchain to protect voters' privacy on the blockchain.

Some companies are already providing such services, one example is `https://polys.me/blockchain/online-voting-system`.
Recently, presidential elections were held in Sierra Leone using blockchain technology, making it the first country to use blockchain technology for elections (`https://www.coindesk.com/sierra-leone-secretly-holds-first-blockchain-powered-presidential-vote/`).

Citizen identification (ID cards)

Electronic IDs or national ID cards are issued by various countries around the world at present. These cards are secure and possess many security features that thwart duplication or tampering attempts. However, with the advent of blockchain technology, several improvements can be made to this process.

Digital identity is not only limited to just government-issued ID cards; it is a concept that applies to online social networks and forums too. There can be multiple identities used for different purposes. A blockchain-based online digital identity allows control over personal information sharing. Users can see who used their data and for what purpose and can control access to it. This is not possible with the current infrastructures which are centrally controlled. The key benefit is that a single identity issued by the government can be used easily and in a transparent manner for multiple services via a single government blockchain. In this case, the blockchain serves as a platform where a government is providing various services such as pensions, taxation, or benefits and a single ID is being used for accessing all these services. Blockchain, in this case, provides a permanent record of every change and transaction made by a digital ID, thus ensuring integrity and transparency of the system. Also, citizens can notarize birth certificates, marriages, deeds, and many other documents on the blockchain tied with their digital ID as a proof of existence.

Currently, there are successful implementations of identity schemes in various countries that work well, and there is an argument that perhaps blockchain is not required in identity management systems. Although there are several benefits such as privacy and control over the usage of identity information due to the current immaturity of blockchain technology, perhaps it is not ready for use in real-world identity systems. However, research is being carried out by various governments to explore the usage of blockchain for identity management.

Moreover, laws such as the right to be forgotten can be quite difficult to incorporate into blockchain due to its immutable nature.

Miscellaneous

Other government functions where blockchain technology can be implemented to improve cost and efficiency include the collection of taxes, benefits management and disbursement, land ownership record management, life event registration (marriages, births), motor vehicle registration, and licenses. This is not an exhaustive list, and over time many functions and processes of a government can be adapted to a blockchain-based model. The key benefits of blockchain such as immutability, transparency, and decentralization can help to bring improvements to most of the traditional government systems.

Health

The health industry has also been identified as another major industry that can benefit by adapting blockchain technology. Blockchain provides an immutable, auditable, and transparent system that traditional peer-to-peer networks cannot. Also, blockchain provides a cost-effective, simpler infrastructure as compared to traditional complex PKI networks. In healthcare, major issues such as privacy compromises, data breaches, high costs, and fraud can arise from lack of interoperability, overly complex processes, transparency, auditability, and control. Another burning issue is counterfeit medicines; especially in developing countries, this is a major cause of concern.

With the adaptability of blockchain in the health sector, several benefits can be realized, ranging from cost saving, increased trust, faster processing of claims, high availability, no operational errors due to complexity in the operational procedures, and preventing the distribution of counterfeit medicines.

From another angle, blockchains that are providing a digital currency as an incentive for mining can be used to provide processing power to solve scientific problems that can help to find cures for certain diseases. Examples include FoldingCoin, which rewards its miners with FLDC tokens for sharing their computer's processing power for solving scientific problems that require unusually large calculations.

FoldingCoin is available at `http://foldingcoin.net/`.

Another similar project is called CureCoin which is available at `https://www.curecoin.net/`. It is yet to be seen that how successful these projects will be in achieving their goals but the idea is very promising.

Finance

Blockchain has many applications in the finance industry. Blockchain in finance is the hottest topic in the industry currently, and major banks and financial organizations are researching to find ways to adapt blockchain technology primarily due to its highly-desired potential to cost-save.

Insurance

In the insurance industry, blockchain technology can help to stop fraudulent claims, increase the speed of claim processing, and enable transparency. Imagine a shared ledger between all insurers that can provide a quick and efficient mechanism for handling intercompany claims. Also, with the convergence of IoT and blockchain, an ecosystem of smart devices can be imagined where all these things can negotiate and manage their insurance policies controlled by smart contracts on the blockchain.

Blockchain can reduce the overall cost and effort required to process claims. Claims can be automatically verified and paid via smart contracts and the associated identity of the insurance policyholder. For example, a smart contract with the help of Oracle and possibly IoT can make sure that when the accident occurred, it can record related telemetry data and based on this information can release payment. It can also withhold payment if the smart contract after evaluating conditions of payment concludes that payment should not be released. For example, in a scenario where an authorized workshop did not repair the vehicle or was used outside a designated area and so on and so forth. There can be many conditions that a smart contract can evaluate to process claims and choice of these rules depend on the insurer, but the general idea is that smart contracts in combination with IoT and Oracle can automate the entire vehicle insurance industry.

Several start-ups such as Dynamis have proposed smart contract-based peer-to-peer insurance platforms that run on Ethereum blockchain. This is initially proposed to be used for unemployment insurance and does not require underwriters in the model.

It is available at `http://dynamisapp.com/`.

Post-trade settlement

This is the most sought-after application of blockchain technology. Currently, many financial institutions are exploring the possibility of using blockchain technology to simplify, automate, and speed up the costly and time-consuming post-trade settlement process.

To understand the problem better, the trade life cycle is described briefly. A trade life cycle contains three steps: execution, clearing, and settlement. Execution is concerned with the commitment of trading between two parties and can be entered into the system via front office order management terminals or exchanges. Clearing is the next step whereby the trade is matched between the seller and buyer based on certain attributes such as price and quantity. At this stage, accounts that are involved in payment are also identified. Finally, the settlement is where eventually the security is exchanged for payment between the buyer and seller.

In the traditional trade life cycle model, a central clearinghouse is required to facilitate trading between parties which bears the credit risk of both parties. The current scheme is somewhat complicated, whereby a seller and buyer have to take a complicated route to trade with each other. This comprises of various firms, brokers, clearing houses, and custodians but with blockchain, a single distributed ledger with appropriate smart contracts can simplify this whole process and can enable buyers and sellers to talk directly to each other.

Notably, the post-trade settlement process usually takes two to three days and has a dependency on central clearing houses and reconciliation systems. With the shared ledger approach, all participants on the blockchain can immediately see a single version of truth regarding the state of the trade. Moreover, the peer-to-peer settlement is possible, which results in the reduction of complexity, cost, risk, and the time it takes to settle the trade. Finally, intermediaries can be eliminated by making use of appropriate smart contracts on the blockchain. Also, regulators can also see view the blockchain for auditing and regulatory requirements.

This can be very useful in implementing MIFID-II regulation requirements (`https://www.fca.org.uk/markets/mifid-ii`).

Financial crime prevention

Know Your Customer (**KYC**), and **Anti Money Laundering** (**AML**) are the key enablers for the prevention of financial crime. In the case of KYC, currently, each institution maintains their own copy of customer data and performs verification via centralized data providers. This can be a time-consuming process and can result in delays in onboarding a new client.

Blockchain can provide a solution to this problem by securely sharing a distributed ledger between all financial institutions that contain verified and true identities of customers. This distributed ledger can only be updated by consensus between the participants thus providing transparency and auditability. This can not only reduce costs but also enable meeting regulatory and compliance requirements in a better and consistent manner.

In the case of AML, due to the immutable, shared, and transparent nature of blockchain, regulators, can easily be granted access to a private blockchain where they can fetch data for relevant regulatory reporting. This will also result in reducing complexity and costs related to the current regulatory reporting paradigm where data is fetched from various legacy and disparate systems and aggregated and formatted together for reporting purposes. Blockchain can provide a single shared view of all financial transactions in the system that are cryptographically secure, authentic, and auditable, thus reducing the costs and complexity associated with the currently employed regulatory reporting methods.

Media

Critical issues in the media industry revolve around content distribution, rights management, and royalty payments to artists. For example, digital music can be copied many times without any restriction and any attempts to apply copy protection have been hacked in some way or other. There is no control over the distribution of the content that a musician or songwriter produces; it can be copied as many times as needed without any restriction and consequently has an impact on the royalty payments. Also, payments are not always guaranteed and are based on traditional airtime figures. All these issues revolving around copy protection and royalty payments can be resolved by connecting consumers, artists, and all players in the industry, allowing transparency and control over the process. Blockchain can provide a network where digital music is cryptographically guaranteed to be owned only by the consumers who pay for it. This payment mechanism is controlled by a smart contract instead of a centralized media agency or authority. The payments will be automatically made based on the logic embedded within the smart contract and number of downloads.

A recent example of such an initiative is Musicoin (`https://musicoin.org`).

Moreover, illegal copying of digital music files can be stopped altogether because everything is recorded and owned immutably in a transparent manner on the blockchain. A music file, for example, can be stored with owner information and timestamp which can be traced throughout the blockchain network. Furthermore, the consumers who own a legal copy of some content are cryptographically tied to the content they have, and it cannot be moved to another owner unless permissioned by the owner. Copyrights and transfers can be managed easily via blockchain once all digital content is immutably recorded on the blockchain. Smart contracts can then control the distribution and payment to all concerned parties.

Summary

There are many applications of blockchain technology, and as discussed in the chapter they can be implemented in various industries to bring about multiple benefits to existing solutions. In this chapter, five main industries that can benefit from blockchain have been discussed. First IoT was discussed, which is another revolutionary technology on its own; and by combining it with the blockchain, several fundamental limitations can be addressed, which brings about tremendous benefits to the IoT industry. More focus has been given to IoT as it is the most prominent and most ready candidate for adapting blockchain technology.

Already, practical use cases and platforms have emerged in the form of **Platform as a Service** (**PaaS**) for blockchain-based IoT such as the IBM Watson IoT blockchain. IBM Blue Horizon is also now available for experimentation, which is a decentralized blockchain-based IoT network. Second, applications in the government sector were discussed whereby various government processes such as homeland security, identification cards, and benefit disbursements can be made transparent, secure, and more robust.

Furthermore, issues in the finance sector were discussed with possible solutions that blockchain technology could provide. Although the finance sector is exploring the possibilities of using blockchain with high energy and enthusiasm, it is still far away from production-ready blockchain-based systems. Finally, some aspects of the health sector and music industry were also discussed. All these use cases and much more in the industry stand on pillars provided by core attributes of blockchain technology such as decentralization, transparency, reliability, and security. However, certain challenges need to be addressed before blockchain technology can be adapted fully; these will be discussed in the next chapter.