DMC8025

MASTER OF COMPUTER APPLICATIONS

BLOCKCHAIN TECHNOLOGIES

CENTREFORDISTANCEEDUCATION ANNA UNIVERSITY
CHENNAI–600025

SYLLABUS

UNIT I INTRODUCTION TO BLOCKCHAIN: History of Blockchain – Types of Blockchain – Consensus – Decentralization using Blockchain – Blockchain and Full Ecosystem Decentralization – Platforms for Decentralization.

UNIT II INTRODUCTION TO CRYPTOCURRENCY: Bitcoin – Digital Keys and Addresses – Transactions – Mining – Bitcoin Networks and Payments – Wallets – Alternative Coins – Theoretical Limitations – Bitcoin Limitations – Name Coin – Prime Coin – Zcash – Smart Contracts – Ricardian Contracts.

UNIT III ETHEREUM: The Ethereum Network – Components of Ethereum Ecosystem – Ethereum Programming Languages: Runtime Byte Code, Blocks and Blockchain, Fee Schedule – Supporting Protocols – Solidity Language.

UNIT IV WEB3 AND HYPERLEDGER: Introduction to Web3 – Contract Deployment – POST Requests – Development frameworks – Hyperledger as a protocol – The Reference Architecture – Hyperledger Fabric – Distributed Ledger – Corda..

UNIT V ALTERNATIVE BLOCKCHAINS AND NEXT EMERGING TRENDS: Kadena – Ripple-Rootstock – Quorum – Tendermint – Scalability – Privacy – Other Challenges – Blockchain Research Notable Projects – Miscellaneous tools.

Table of Contents

UNIT 1 INTRODUCTION TO BLOCKCHAIN

Learning Objective

To learn the components of blockchain system's fundamental concepts and how they fit to examine centralization of blockchain

Keywords: Blockchain, consensus, decentralization, platforms of blockchain, ethereum, maidsafe

1.1 Introduction

In this Unit, History of blockchain is presented first followed by the definition of Blockchain. The associated terms to understand blockchain, are also presented. You will learn generic structure of blockchain. The types of blockchain, consensus and decentralization are also discussed in detail.

1.2 History of Blockchain

Blockchain was introduced with the invention of Bitcoin in 2008. Its practical implementation then occurred in 2009. In 2008, a groundbreaking paper entitled Bitcoin: A Peer-to-Peer Electronic Cash System was written on the topic of peer-to-peer electronic cash under the pseudonym Satoshi Nakamoto. It introduced the term chain of blocks. The term chain of blocks evolved over the years into the word blockchain.

Before proceeding, let us define Blockchain, formally and discuss the terminologies associated with it to understand.

Blockchain is defined as a peer-to-peer, distributed ledger that is cryptographically-secure, append-only, immutable (extremely hard to change), and updateable only via consensus or agreement among peers.

In other words, Blockchain is an ever-growing, secure, shared record keeping system in which each user of the data holds a copy of the records, which can only be updated if all parties involved in a transaction agree to update.

Some of the major terms to be known to understand Blockchain are, peer to peer, distributed ledger, append-only and consensus.

The first keyword in the technical definition is ***peer-to-peer***. This means that there is no central controller in the network, and all participants talk to each other directly. This property allows

for cash transactions to be exchanged directly among the peers without a third-party involvement, such as by a bank.

As already mentioned Blockchain is a ***distributed ledger***, which simply means that a ledger is spread across the network among all peers in the network, and each peer holds a copy of the complete ledger.

Next, we see that this ledger is cryptographically-secure, which means that cryptography has been used to provide security services which make this ledger secure against tampering and misuse. These services include non-repudiation, data integrity, and data origin authentication.

Another property that we encounter is that blockchain is ***append-only***, which means that data can only be added to the blockchain in time-ordered sequential order. This property implies that once data is added to the blockchain, it is almost impossible to change that data and can be considered practically immutable.

Finally, the most critical attribute of a blockchain is that it is updateable only via ***consensus.*** This is what gives it the power of decentralization. In this scenario, no central authority is in control of updating the ledger. Instead, any update made to the blockchain is validated against strict criteria defined by the blockchain protocol and added to the blockchain only after a consensus has been reached among all participating peers/nodes on the network. To achieve consensus, there are various consensus facilitation algorithms which ensure that all parties are in agreement about the final state of the data on the blockchain network and resolutely agree upon it to be true. Consensus algorithms are discussed later.

A block is merely a selection of transactions bundled together and organized logically. A transaction is a record of an event, for example, the event of transferring cash from a sender's account to a beneficiary's account.

A block is made up of transactions, and its size varies depending on the type and design of the blockchain in use.

A reference to a previous block is also included in the block unless it is a genesis block. A genesis block is the first block in the blockchain that is hardcoded at the time the blockchain was first started. The structure of a block is also dependent on the type and design of a blockchain.

Generally, however, there are just a few attributes that are essential to the functionality of a block: the block header, which is composed of pointer to previous block, the timestamp, nonce, Merkle root, and the block body that contains transactions. There are also other attributes in a block, but generally, the aforementioned components are always available in a block.

A ***nonce*** is a number that is generated and used only once. A nonce is used extensively in many cryptographic operations to provide replay protection, authentication, and encryption. In blockchain, it's used in PoW consensus algorithms and for transaction replay protection.
Merkle root is a hash of all of the nodes of a Merkle tree. Merkle trees are widely used to validate the large data structures securely and efficiently. In the blockchain world, Merkle trees are commonly used to allow efficient verification of transactions. Merkle root in a blockchain is present in the block header section of a block, which is the hash of all transactions in a block. This means that verifying only the Merkle root is required to verify all transactions present in the Merkle tree instead of verifying all transactions one by one.

1.2.2 Generic elements of a blockchain

Now, let's walk through the generic elements of a blockchain. You can use this as a handy reference section if you ever need a reminder about the different parts of a blockchain. More precise elements will be discussed in the context of their respective blockchains in later chapters, for example, the Ethereum blockchain:

1.2.3 Generic structure of a blockchain

Elements of a generic blockchain are described here one by one. These are the elements that you will come across in relation to blockchain:

- **Address:** Addresses are unique identifiers used in a blockchain transaction to denote senders and recipients. An address is usually a public key or derived from a public key. While addresses can be reused by the same user, addresses themselves are unique. In practice, however, a single user may not use the same address again and generate a new one for each transaction. This newly-created address will be unique.
- **Transaction:** A transaction is the fundamental unit of a blockchain. A transaction represents a transfer of value from one address to another.
- **Block:** A block is composed of multiple transactions and other elements, such as the previous block hash (hash pointer), timestamp, and nonce.
- **Peer-to-peer network:** As the name implies, a peer-to-peer network is a network topology wherein all peers can communicate with each other and send and receive messages.
- **Scripting or programming language:** Scripts or programs perform various operations on a transaction in order to facilitate various functions. For example, in Bitcoin, transaction scripts are predefined in a language called Script, which consist of sets of

commands that allow nodes to transfer tokens from one address to another. Script is a limited language, however, in the sense that it only allows essential operations that are necessary for executing transactions, but it does not allow for arbitrary program development. and branching capability to perform complex computations. Therefore, Bitcoin's scripting language is not

- **Virtual machine:** A virtual machine allows Turing complete code to be run on a blockchain (as smart contracts); whereas a transaction script is limited in its operation. Various blockchains use virtual machines to run programs such as Ethereum Virtual Machine (EVM) and Chain Virtual Machine (CVM). EVM is used in Ethereum blockchain, while CVM is a virtual machine developed for and used in an enterprise-grade blockchain called Chain Core.
- **State machine:** A blockchain can be viewed as a state transition mechanism whereby a state is modified from its initial form to the next one and eventually to a final form by nodes on the blockchain network as a result of a transaction execution, validation, and finalization process.
- **Node:** A node in a blockchain network performs various functions depending on the role that it takes on. A node can propose and validate transactions and perform mining to facilitate consensus and secure the blockchain. This goal is achieved by following a consensus protocol (most commonly PoW).
- **Smart contract:** These programs run on top of the blockchain and encapsulate the business logic to be executed when certain conditions are met. These programs are enforceable and automatically executable. Smart contracts have many use cases, including but not limited to identity management, capital markets, trade finance, record management, insurance, and e-governance.

1.2.4 Functioning of Blockchain

We have now defined and described blockchain. Now let's see how a blockchain actually works. Nodes are either miners who create new blocks and mint cryptocurrency (coins) or block signers who validates and digitally sign the transactions. A critical decision that every blockchain network has to make is to figure out that which node will append the next block to the blockchain. This decision is made using a consensus mechanism.

Now we will look at the how a blockchain validates transactions and creates and adds blocks to grow the blockchain. Now, let us discuss a general scheme for creating blocks to get to know how blocks are generated and what the relationship is between transactions and blocks:

1. A node starts a transaction by first creating and then digitally signing it with its private key. A transaction can represent various actions in a blockchain. Most commonly this is a data structure that represents transfer of value between users on the blockchain network. Transaction data structure usually consists of some logic of transfer of value, relevant rules, source and destination addresses, and other validation information.
2. A transaction is propagated (flooded) by using a flooding protocol, called Gossip protocol, to peers that validate the transaction based on preset criteria. Usually, more than one node are required to verify the transaction.
3. Once the transaction is validated, it is included in a block, which is then propagated onto the network. At this point, the transaction is considered confirmed.
4. The newly-created block now becomes part of the ledger, and the next block links itself cryptographically back to this block. This link is a hash pointer. At this stage, the transaction gets its second confirmation and the block gets its first confirmation.
5. Transactions are then reconfirmed every time a new block is created. Usually, six confirmations in the Bitcoin network are required to consider the transaction final.
Note that steps 4 and 5 are considered non-compulsory, as the transaction itself is finalized in step 3; however, block confirmation and further transaction reconfirmations, if required, are then carried out in step 4 and step 5.
With this, introduction to blockchain is completed, and next let us learn various types of blockchain.

1.3 Types of Blockchain

Blockchains are broadly categorized into public, private, consortium, and hybrid blockchains, each type offers unique characteristics, benefits, and use cases. Public blockchains enable open access and decentralization, while private blockchains prioritize security and control. Consortium blockchains serve collaborative networks, and hybrid blockchains combine features of both public and private models. Let us learn one by one in the following sub-sections. Before that permissioned and permission less blockchains.

1.3.1 Permissioned & Permission less Blockchains

A permissioned blockchain is a type of blockchain network that restricts access and participation to a select group of authorized users. Unlike permissionless blockchains, where anyone can join and validate transactions, permissioned blockchains require participants to obtain permission before they can access the network or perform certain actions.

Access Control: Only authorized participants can join the network, ensuring that all nodes are known and vetted. This allows for greater control over who can validate transactions and access data.

Centralized Governance: Typically governed by a consortium of organizations or a central authority, which makes decisions about network rules and policies.

Enhanced Privacy: Transactions and data are often more private, as sensitive information can be kept off-chain or shared only among authorized parties.

Customizable Protocols: Organizations can customize consensus mechanisms and other protocols to meet their specific needs and requirements.

1.3.2 Public blockchains

As the name suggests, public blockchains are not owned by anyone. They are open to the public, and anyone can participate as a node in the decision-making process. Users may or may not be rewarded for their participation. All users of these permissionless or unpermissioned ledgers maintain a copy of the ledger on their local nodes and use a distributed consensus mechanism to decide the eventual state of the ledger. Public Blockchain is secured with proof of work or proof of stake they can be used to displace traditional financial systems. The more advanced side of this blockchain is the smart contract that enabled this blockchain to support decentralization. Bitcoin and Ethereum are both considered public blockchains.

1.3.3. Private Blockchain

These blockchains are not as decentralized as the public blockchain only selected nodes can participate in the process, making it more secure than the others. They are open to some authorized users only. These blockchains are operated in a closed network. In this few people are allowed to participate in a network within a company/organization. With proper security and maintenance, this blockchain is a great asset to secure information without exposing it to the public eye. Therefore, companies use them for internal auditing, voting, and asset management. An example of private blockchains is Hyperledger, Corda.

1.3.4 Semiprivate or Hybrid Blockchain

With semiprivate blockchains, part of the blockchain is private and part of it is public. Note that this is still just a concept today, and no real world POCs have yet been developed. With a semi-private blockchain, the private part is controlled by a group of individuals, while the public part is open for participation by anyone.

This hybrid model can be used in scenarios where the private part of the blockchain remains internal and shared among known participants, while the public part of the blockchain can still be used by anyone, optionally allowing mining to secure the blockchain. This way, the blockchain as a whole can be secured using PoW, thus providing consistency and validity for both the private and public parts. This type of blockchain can also be called a semi-decentralized model, where it is controlled by a single entity but still allows for multiple users to join the network by following appropriate procedures.

1.3.5 Consortium or Federated Blockchain

It is a creative approach that solves the needs of the organization. This blockchain validates the transaction and also initiates or receives transactions.

This is an innovative method to solve the organization's needs. Some part is public and some part is private. In this type, more than one organization manages the blockchain.

It has high potential in businesses, banks, and other payment processors. Food tracking of the organizations frequently collaborates with their sectors making it a federated solution ideal for their use. Examples of consortium Blockchain are Tendermint and Multichain.

1.3.6. Tokenized & Tokenless Blockchains

These blockchains are standard blockchains that generate cryptocurrency as a result of a consensus process via mining or initial distribution. Bitcoin and Ethereum are prime examples of this type of blockchain.

Tokenless blockchains are designed in such a way that they do not have the basic unit for the transfer of value. However, they are still valuable in situations where there is no need to transfer value between nodes and only the sharing of data among various trusted parties is required. This is similar to full private blockchains, the only difference being that use of tokens is not required. This can also be thought of as a shared distributed ledger used for storing data. It does have its benefits when it comes to immutability, security, and consensus, driven updates but are not used for common blockchain application of value transfer or cryptocurrency.

1.4 Consensus

Consensus is the backbone of a blockchain and, as a result, it provides decentralization of control through an optional process known as mining. The choice of the consensus algorithm is also governed by the type of blockchain in use; that is, not all consensus mechanisms are suitable for all types of blockchains. For example, in public permissionless blockchains, it would make sense to use PoW instead of a simple agreement mechanism that is perhaps based

on proof of authority. Therefore, it is essential to choose an appropriate consensus algorithm for a particular blockchain project.

Consensus is a process of agreement between distrusting nodes on the final state of data. To achieve consensus, different algorithms are used. It is easy to reach an agreement between two nodes, but when multiple nodes are participating in a distributed system and they need to agree on a single value, it becomes quite a challenge to achieve consensus. This process of attaining agreement common state or value among multiple nodes despite the failure of some nodes is known as distributed consensus.

1.4.1 Consensus Mechanism

A consensus mechanism is a set of steps that are taken by most or all nodes in a blockchain to agree on a proposed state or value. For more than three decades, this concept has been researched by computer scientists in industry and academia. Consensus mechanisms have most recently come into the limelight and gained considerable popularity with the advent of blockchain and Bitcoin.

There are various requirements that must be met to provide the desired results in a consensus mechanism. The following describes these requirements:

- Agreement: All honest nodes decide on the same value
- Termination: All honest nodes terminate execution of the consensus process and eventually reach a decision
- Validity: The value agreed upon by all honest nodes must be the same as the initial value proposed by at least one honest node
- Fault tolerant: The consensus algorithm should be able to run in the presence of faulty or malicious nodes (Byzantine nodes)
- Integrity: This is a requirement that no node can make the decision more than once in a single consensus cycle

There are two general categories of consensus mechanisms. These categories deal with all types of faults (fail stop type or arbitrary). These common types of consensus mechanisms are as follows:

Traditional Byzantine Fault Tolerance (BFT)-based: With no compute-intensive operations, such as partial hash inversion (as in Bitcoin PoW), this method relies on a simple scheme of

nodes that are publisher-signed messages. Eventually, when a certain number of messages are received, then an agreement is reached.

Leader election-based consensus mechanisms: This arrangement requires nodes to compete in a leader election lottery, and the node that wins proposes a final value. For example, the PoW used in Bitcoin falls into this category.

1.4.2 Consensus Algorithms Available

The consensus algorithms available today, or that are being researched in the context of blockchain, are presented here. The following is not an exhaustive list, but it includes all notable algorithms.

- Proof of Work (PoW): This type of consensus mechanism relies on proof that adequate computational resources have been spent before proposing a value for acceptance by the network. This scheme is used in Bitcoin, Litecoin, and other cryptocurrency blockchains. Currently, it is the only algorithm that has proven to be astonishingly successful against any collusion attacks on a blockchain network, such as the Sybil attack.
- Proof of Stake (PoS): This algorithm works on the idea that a node or user has an adequate stake in the system; that is, the user has invested enough in the system so that any malicious attempt by that user would outweigh the benefits of performing such an attack on the network. This idea was first introduced by Peercoin, and it is going to be used in the Ethereum blockchain version called Serenity. Another important concept in PoS is coin age, which is a criterion derived from the amount of time and number of coins that have not been spent.
- Delegated Proof of Stake (DPoS): This is an innovation over standard PoS, whereby each node that has a stake in the system can delegate the validation of a transaction to other nodes by voting. It is used in the BitShares blockchain.
- Proof of Elapsed Time (PoET): Introduced by Intel in 2016, PoET uses a Trusted Execution
- Environment (TEE) to provide randomness and safety in the leader election process via a guaranteed wait time. It requires the Intel Software Guard Extensions (SGX) processor to provide the security guarantee for it to be secure.
- Proof of Deposit (PoD): In this case, nodes that wish to participate in the network have to make a security deposit before they can mine and propose blocks. This mechanism is used in the Tendermint blockchain.

- Proof of Importance (PoI): This idea is significant and different from PoS. PoI not only relies on how large a stake a user has in the system, but it also monitors the usage and movement of tokens by the user in order to establish a level of trust and importance. It is used in the NEM coin blockchain.
- Federated consensus or federated Byzantine consensus: This mechanism is used in the stellar consensus protocol. Nodes in this protocol retain a group of publicly-trusted peers and propagate only those transactions that have been validated by the majority of trusted nodes.
- Reputation-based mechanisms: As the name suggests, a leader is elected by the reputation it has built over time on the network. It is based on the votes of other members.
- PBFT: This mechanism achieves state machine replication, which provides tolerance against Byzantine nodes. Various other protocols including PBFT, PAXOS, RAFT, and Federated Byzantine Agreement (FBA) are also being used or have been proposed for use in many different implementations of distributed systems and blockchains.
- Proof of Activity (PoA): In this scheme, PoW and PoS are combined together to achieve consensus and good level of security. This scheme is more energy efficient as PoW is used only in the first stage of the mechanism, after the first stage it switches to PoS which consumes negligible energy.
- Proof of Capacity (PoC): This scheme uses hard disk space as a resource to mine the blocks. This is different from PoW, where CPU resources are used. In in PoC, hard disk space is utilized for mining and as such is also known as hard drive mining. This concept was first introduced in the Burstcoin cryptocurrency.
- Proof of Storage (PoS): This scheme allows for the outsourcing of storage capacity. This scheme is based on the concept that a particular piece of data is probably stored by a node which serves as a means to participate in the consensus mechanism. Several variations of this scheme have been proposed, such as Proof of Replication, Proof of Data Possession, Proof of Space, and Proof of Space-Time.

1.5 Decentralization using Blockchain

In this chapter, we will discuss the concept of decentralization in the context of blockchain. The fundamental basis of blockchain is that no single central authority is in control, and, in this chapter, we will present examples of various methods of decentralization and routes

to achieve this. We will discuss the decentralization of the blockchain ecosystem, decentralized applications, and platforms for achieving decentralization, in the subsequent sections.

Decentralization is a core benefit and service provided by blockchain technology. By design, blockchain is a perfect vehicle for providing a platform that does not need any intermediaries and that can function with many different leaders chosen via consensus mechanisms. This model allows anyone to compete to become the decision-making authority. This competition is governed by a consensus mechanism, and the most commonly used method is known as Proof of Work (PoW).

Decentralization is applied in varying degrees from a semi-decentralized model to a fully decentralized one depending on the requirements and circumstances. Decentralization can be viewed from a blockchain perspective as a mechanism that provides a way to remodel existing applications and paradigms, or to build new applications, in order to give full control to users. Information and Communication Technology (ICT) has conventionally been based on a centralized paradigm whereby database or application servers are under the control of a central authority, such as a system administrator. With Bitcoin and the advent of blockchain technology, this model has changed and now the technology exists, which allows anyone to start a decentralized system and operate it with no single point of failure or single trusted authority. It can either be run autonomously or by requiring some human intervention, depending on the type and model of governance used in the decentralized application running on blockchain.

The following diagram shows the different types of systems that currently exist: central, decentralized, and distributed.

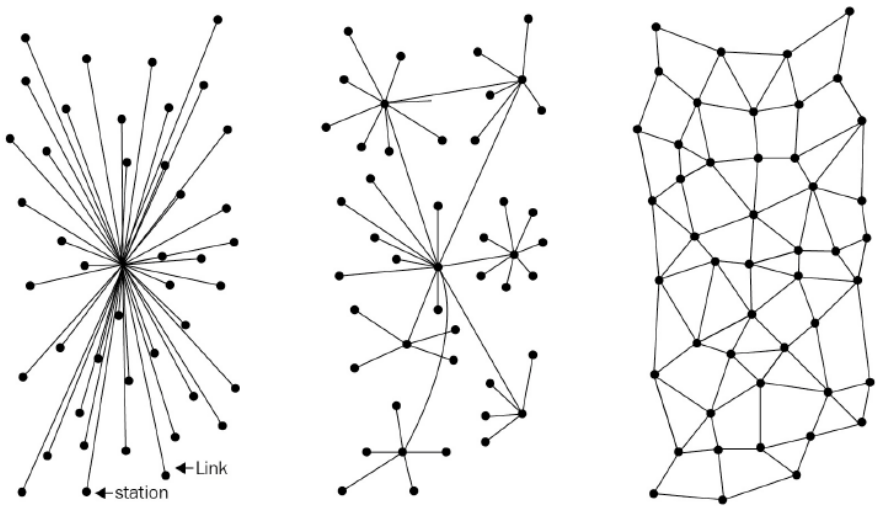

Fig 1.5 Different types of networks

Centralized systems are conventional (client-server) IT systems in which there is a single authority that controls the system, and who is solely in charge of all operations on the system. All users of a centralized system are dependent on a single source of service. The majority of online service providers including Google, Amazon, eBay, Apple's App Store, and others use this conventional model for delivering services.

A distributed system, data and computation are spread across multiple nodes in the network. Sometimes, this term is confused with parallel computing. While there is some overlap in the definition, the main difference between these systems is that in a parallel computing system, computation is performed by all nodes simultaneously in order to achieve the result; for example, parallel computing platforms are used in weather research and forecasting, simulation and financial modeling. On the other hand, in a distributed system, computation may not happen in parallel and data is replicated across multiple nodes that users view as a single, coherent system.

Variations of both of these models are used with to achieve fault tolerance and speed. In the parallel system model, there is still a central authority that has control over all nodes, which governs processing. This means that the system is still centralized in nature. The critical difference between a decentralized system and distributed system is that in a distributed system, there still exists a central authority that governs the entire system; whereas, in a decentralized system, no such authority exists.

A decentralized system is a type of network where nodes are not dependent on a single master node; instead, control is distributed among many nodes. This is analogous to a model where each department in an organization is in charge of its own database server, thus taking away the power from the central server and distributing it to the sub departments who manage their own databases.

A significant innovation in the decentralized paradigm that has given rise to this new era of decentralization of applications is decentralized consensus. This mechanism came into play with Bitcoin, and it enables a user to agree on something via a consensus algorithm without the need for a central, trusted third party, intermediary, or service provider.

Two methods can be used to achieve decentralization: disintermediation and competition (Contest-driven decentralization). These methods will be discussed in detail in the sections that follow.

Disintermediation

The concept of disintermediation can be explained with the aid of an example. Imagine that you want to send money to a friend in another country. You go to a bank who, for a fee, will transfer your money to the bank in that country. In this case, the bank maintains a central database that is updated, confirming that you have sent the money. With blockchain technology, it is possible to send this money directly to your friend without the need for a bank. All you need is the address of your friend on the blockchain. This way, the intermediary; that is, the bank, is no longer required, and decentralization is achieved by disintermediation. It is debatable, however, how practical decentralization through disintermediation is in the financial sector due to massive regulatory and compliance requirements. Nevertheless, this model can be used not only in finance but in many different
industries as well.

Contest-driven decentralization

In the method involving competition, different service providers compete with each other in order to be selected for the provision of services by the system. This paradigm does not achieve complete decentralization. However, to a certain degree, it ensures that an intermediary or service provider is not monopolizing the service. In the context of blockchain technology, a system can be envisioned in which smart contracts can choose an external data provider from a large number of providers based on their reputation, previous score, reviews, and quality of service. This method will not result in full decentralization, but it allows smart contracts to make a free choice based on the criteria just mentioned. This way, an environment of competition is cultivated among service providers where they compete with each other to become the data provider of choice.

1.5.1 The decentralization framework example

Let's evaluate a money transfer system as an example of an application selected to be decentralized. The four questions discussed previously are used to evaluate the decentralization requirements of this application. The answers to these questions are as follows:

1. Money transfer system
2. Disintermediation
3. Bitcoin
4. Atomicity

The responses indicate that the money transfer system can be decentralized by removing the intermediary, implemented on the Bitcoin blockchain, and that a security guarantee will be provided via atomicity. Atomicity will ensure that transactions execute successfully in full or not execute at all. We have chosen Bitcoin blockchain because it is the longest established blockchain which has stood the test of time.

Similarly, this framework can be used for any other system that needs to be evaluated in terms of decentralization. The answers to these four simple questions help clarify what approach to take to decentralize the system.

1.6 Blockchain and Full Ecosystem of Decentralization

To achieve complete decentralization, it is necessary that the environment around the blockchain also be decentralized. The blockchain is a distributed ledger that runs on top of conventional systems. These elements include storage, communication, and computation. There are other factors, such as identity and wealth, which are traditionally based on centralized paradigms, and there's a need to decentralize these aspects as well in order to achieve a sufficiently decentralized ecosystem.

1.6.1 Storage

Data can be stored directly in a blockchain, and with this fact it achieves decentralization. However, a significant disadvantage of this approach is that a blockchain is not suitable for storing large amounts of data by design. It can store simple transactions and some arbitrary data, but it is certainly not suitable for storing images or large blobs of data, as is the case with traditional database systems.

A better alternative for storing data is to use Distributed Hash Tables (DHTs). DHTs were used initially in peer-to-peer file sharing software, such as BitTorrent, Napster, Kazaa, and Gnutella. DHT research was made popular by the CAN, Chord, Pastry, and Tapestry projects. BitTorrent is the most scalable and fastest network, but the issue with BitTorrent and the others is that there is no incentive for users to keep the files indefinitely.

Users generally don't keep files permanently, and if nodes that have data still required by someone leave the network, there is no way to retrieve it except by having the required nodes rejoin the network so that the files once again become available.

Two primary requirements here are high availability and link stability, which means that data should be available when required and network links also should always be accessible. InterPlanetary File System (IPFS) by Juan Benet possesses both of these properties, and its vision is to provide a decentralized World Wide Web by replacing the HTTP protocol. IPFS

uses Kademlia DHT and Merkle Directed Acyclic Graph (DAG) to provide storage and searching functionality, respectively.

The incentive mechanism for storing data is based on a protocol known as Filecoin, which pays incentives to nodes that store data using the Bitswap mechanism. The Bitswap mechanism lets nodes keep a simple ledger of bytes sent or bytes received in a one-to-one relationship. Also, a Git-based version control mechanism is used in IPFS to provide structure and control over the versioning of data.

There are other alternatives for data storage, such as Ethereum Swarm, Storj, and MaidSafe. Ethereum has its own decentralized and distributed ecosystem that uses Swarm for storage and the Whisper protocol for communication. MaidSafe aims to provide a decentralized World Wide Web.

BigchainDB is another storage layer decentralization project aimed at providing a scalable, fast, and linearly scalable decentralized database as opposed to a traditional file system. BigchainDB complements decentralized processing platforms and file systems such as Ethereum and IPFS.

1.6.2 Communication

The internet (the communication layer in blockchain) is considered to be decentralized. This belief is correct to some extent, as the original vision of the internet was to develop a decentralized communications system.

Services such as email and online storage are now all based on a paradigm where the service provider is in control, and users trust such providers to grant them access to the service as requested. This model is based on unconditional trust of a central authority (the service provider) where users are not in control of their data. Even user passwords are stored on trusted third-party systems.

Thus, there is a need to provide control to individual users in such a way that access to their data is guaranteed and is not dependent on a single third party. Access to the internet (the communication layer) is based on Internet Service Providers (ISPs) who act as a central hub for internet users. If the ISP is shut down for any reason, then no communication is possible with this model.

An alternative is to use mesh networks. Even though they are limited in functionality when compared to the internet, they still provide a decentralized alternative where nodes can talk directly to each other without a central hub such as an ISP.

An example of a Meshnet is FireChat (http://www.opengarden.com/firechat.html), which allows iPhone users to communicate with each other directly in a peer-to-peer fashion without an internet connection.

Now imagine a network that allows users to be in control of their communication; no one can shut it down for any reason. This could be the next step toward decentralizing communication networks in the blockchain ecosystem. It must be noted that this model may only be vital in a jurisdiction where the internet is censored and controlled by the government.

As mentioned earlier, the original vision of the internet was to build a decentralized network; however, over the years, with the advent of large-scale service providers such as Google, Amazon, and eBay, control is shifting towards these big players. For example, email is a decentralized system at its core; that is, anyone can run an email server with minimal effort and can start sending and receiving emails. There are better alternatives available, for example, Gmail and Outlook.com, which already provide managed services for end users, so there is a natural inclination toward selecting from these large centralized services as they are more convenient and free. This is one example that shows how the internet has moved toward centralization.

Free services, however, are offered at the cost of exposing valuable personal data, and many users are unaware of this fact. Blockchain has once again given this vision of decentralization to the world, and now concerted efforts are being made to harness this technology and take advantage of the benefits that it can provide.

1.6.3 Computing power and decentralization

Decentralization of computing or processing power is achieved by a blockchain technology such as Ethereum, where smart contracts with embedded business logic can run on the blockchain network. Other blockchain technologies also provide similar processing-layer platforms, where business logic can run over the network in a decentralized manner.

The following diagram shows a decentralized ecosystem overview. At the bottom layer, the internet or Meshnets provide a decentralized communication layer. On the next layer up, a storage layer uses technologies such as IPFS and BigchainDB to enable decentralization. Finally, at the next level up, you can see that blockchain serves as a decentralized processing (computation) layer. Blockchain can, in a limited way, provide a storage layer too, but that severely hampers the speed and capacity of the system. Therefore, other solutions such as IPFS and BigchainDB are more suitable to store large amounts of data in a decentralized way.

The Identity, Wealth layers are shown at the top level. Identity on the internet is a vast topic, and systems such as BitAuth and OpenID provide authentication and identification services with varying degrees of decentralization and security assumptions:

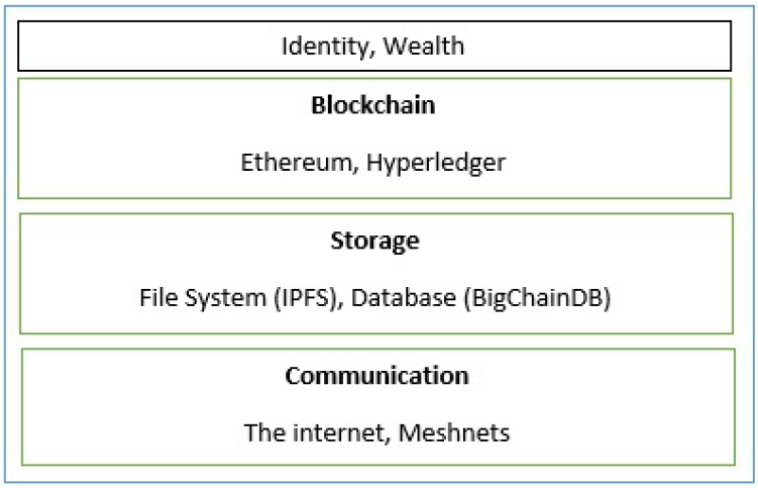

Fig 1.6 Decentralization Ecosystem

The blockchain is capable of providing solutions to various issues relating to decentralization. A concept relevant to identity known as Zooko's Triangle requires that the naming system in a network protocol be secure, decentralized, and is able to provide human-meaningful and memorable names to the users. Conjecture has it that a system can have only two of these properties simultaneously. Nevertheless, with the advent of blockchain in the form of Namecoin, this problem was resolved. It is now possible to achieve security, decentralization, and human-meaningful names with the Namecoin blockchain. Let us learn the following concepts related to decentralization.

1.6.4 Smart Contract

A smart contract is a decentralized program. Smart contracts do not necessarily need a blockchain to run; however, due to the security benefits that blockchain technology provides, blockchain has become a standard decentralized execution platform for smart contracts. A smart contract usually contains some business logic and a limited amount of data. The business logic is executed if specific criteria are met. Actors or participants in the blockchain use these smart contracts, or they run autonomously on behalf of the network participants.

1.6.5 Decentralized Organizations (DO)

Decentralized Organizations are software programs that run on a blockchain and are based on the idea of actual organizations with people and protocols. Once a DO is added to the

blockchain in the form of a smart contract or a set of smart contracts, it becomes decentralized and parties interact with each other based on the code defined within the DO software.

1.6.6 Decentralized Autonomous Organizations

Just like DOs, a Decentralized Autonomous Organization (DAO) is also a computer program that runs atop a blockchain and embedded within it are governance and business logic rules. DAO and DO are fundamentally the same thing. The main difference, however, is that DAOs are autonomous, which means that they are fully automated and contain artificially-intelligent logic. DOs, on the other hand, lack this feature and rely on human input to execute business logic.

The Ethereum blockchain led the way with the initial introduction of DAOs. In a DAO, the code is considered the governing entity rather than people or paper contracts. However, a human curator maintains this code and acts as a proposal evaluator for the community. DAOs are capable of hiring external contractors if enough input is received from the token holders (participants).

Currently, DAOs do not have any legal status, even though they may contain some intelligent code that enforces certain protocols and conditions. However, these rules have no value in the real-world legal system at present.

One day, perhaps an Autonomous Agent (AA); that is, a piece of code that runs without human intervention, commissioned by a law enforcement agency or regulator will contain rules and regulations that could be embedded in a DAO for the purpose of ensuring its integrity from a legalistic and compliance perspective. The fact that DAOs are purely-decentralized entities enables them to run in any jurisdiction.

1.6.7 Requirements of Decentralization

For an application to be considered decentralized, it must meet the following criteria.

- The DApp should be fully open source and autonomous, and no single entity should be in control of a majority of its tokens. All changes to the application must be consensus-driven based on the feedback given by the community.
- Data and records of operations of the application must be cryptographically secured and stored on a public, decentralized blockchain to avoid any central points of failure.
- A cryptographic token must be used by the application to provide access and rewards to those who contribute value to the applications, for example, miners in Bitcoin.

- The tokens must be generated by the DApp according to a standard cryptographic algorithm. This generation of tokens acts as a proof of the value to contributors (for example, miners).

1.7 Platforms of Decentralization

There are many platforms available for decentralization. Many organizations around the world have introduced platforms that promise to make distributed application development easy, accessible, and secure. Some of these platforms are presented in this section.

1.7.1 Ethereum

Ethereum is the first blockchain to introduce a Turing-complete language and the concept of a virtual machine. This is in stark contrast to the limited scripting language in Bitcoin and many other cryptocurrencies. With the availability of its Turing-complete language called Solidity, endless possibilities have opened for the development of decentralized applications. This blockchain provides a public blockchain to develop smart contracts and decentralized applications. Currency tokens on Ethereum are called Ethers.

1.7.2 Maidsafe

MaidSafe provides a Secure Access For Everyone (SAFE) network that is made up of unused computing resources, such as storage, processing power, and the data connections of its users. The files on the network are divided into small chunks of data, which are encrypted and distributed randomly throughout the network. This data can only be retrieved by its respective owner. One key innovation of MaidSafe is that duplicate files are automatically rejected on the network, which helps reduce the need for additional computing resources needed to manage the load. It uses Safecoin as a token to incentivize its contributors.

1.7.3 Lisk

Lisk is a blockchain application development and cryptocurrency platform. It allows developers to use JavaScript to build decentralized applications and host them in their respective sidechains. Lisk uses the Delegated Proof of Stake (DPOS) mechanism for consensus whereby 101 nodes can be elected to secure the network and propose blocks. It uses the Node.js and JavaScript backend, while the frontend allows the use of standard technologies, such as CSS3, HTML5, and JavaScript.

Lisk uses LSK coin as a currency on the blockchain. Another derivative of Lisk is Rise, which is a Lisk-based decentralized application and digital currency platform. It offers a greater focus on the security of the system

1.8 Summary

In this Unit, blockchain technology has been introduced. First, we discussed some basic concepts of distributed systems, and then we reviewed the history of blockchain. We also discussed concepts such as electronic cash.

Furthermore, we presented various definitions of blockchain from different point of views. We also introduced you to few applications of blockchain technology. Next, we explored different types of blockchain. Finally, we examined the benefits and limitations of this new technology. Few topics such as blockchain scalability and adaptability are also presented. The concept of decentralization, which is central to the idea behind blockchains were presented with structure. The Unit concluded with the exposure of various platforms of decentralization.

Short Questions:

1. Define blockchain in your own understanding.
2. What are the various types of blockchain?
3. What is permissioned and permissionless blockchain?
4. What is autonomous agent in the context of decentralization?
5. Write any two platforms of decentralization.

Long questions:

1. Describe the generic elements of blockchain.
2. Explain decentralization in blockchain
3. What are the requirements of decentralization? Explain
4. Describe decentralization ecosystem in blockchain with neat sketch
5. What is smart contract? Explain with illustration.
6. Explain various platforms of decentralization

UNIT 2 INTRODUCTION TO CRYPTOCURRENCY

In this Unit, you are going to learn a Bitcoin, a cryptocurrency an application of blockchain. Let us discuss the main components which composed Bitcoin like, Digital keys, Addresses, Transactions, Blockchain Miners and the Bitcoin network Wallets.

Keywords: Bitcoin, digital keys, addresses, transactions, miners and wallets

2.1 Introduction to Cryptocurrency

Since the 1980s, electronic cash (e-cash) or digital currency protocols have existed that are based on a model proposed by David Chaum, while proposing blockchain. Two fundamental e-cash system issues need to be addressed: accountability and anonymity:

Accountability is required to ensure that cash is spendable only once (double-spend problem) and that it can only be spent by its rightful owner. Double spend problem arises when same money can be spent twice. As it is quite easy to make copies of digital data, this becomes a big issue in digital currencies as you can make many copies of same digital cash.

Anonymity is required to protect users' privacy. As with physical cash, it is almost impossible to trace back spending to the individual who actually paid the money.

Both of these problems can be solved using two cryptographic operations, namely blind signatures and secret sharing. In 2009, the first practical implementation of an electronic cash (e-cash) system named Bitcoin appeared. For the very first time, it solved the problem of distributed consensus in a trustless network. It used public key cryptography with a Proof of Work (PoW) mechanism to provide a secure, controlled, and decentralized method of minting digital currency. The key innovation was the idea of an ordered list of blocks composed of transactions and cryptographically secured by the PoW mechanism. Let us learn how concepts from electronic cash schemes and distributed systems were combined to create Bitcoin.

Before proceeding to learn a bitcoin, a cryptocurrency, you are presented various units and denominations.

To understand cryptocurrency units and denominations, it is first important to understand what a cryptocurrency unit is. Simply put, a cryptocurrency unit is a standardized measure of value for a particular digital asset. This is similar to how the dollar is the standard unit of currency in the United States or how the euro is the standard unit of currency in Europe.

There are two main types of cryptocurrency units: base units and derived units.

Base units are the primary or base measure of value for a particular digital asset. For example, Bitcoin (BTC) is typically measured in satoshis (or sats), the smallest unit of Bitcoin that can be traded on the blockchain.

Unit	Symbol	Bitcoin value
bitcoin	BTC or □	1
millibit	mBTC	0.001
bit	μBTC	0.000 001
satoshi	sat	0.00 000 001

Fig 2.1 Cryptocurrency Units

Now, we will see that how a user will use the Bitcoin network from an end user's perspective. One of the most common transactions is sending money to someone else, therefore, we will see that how a payment can be sent from one user to another on the Bitcoin network.

A payment transaction in the Bitcoin network can be divided into the following steps:

1. Transaction starts with a sender signing the transaction with their private key
2. Transaction is serialized so that it can be transmitted over the network
3. Transaction is broadcasted to the network
4. Miners listening for the transactions picks up the transaction
5. Transaction are verified for their validity by the miners
6. Transaction are added to the candidate/proposed block for mining
7. Once mined, the result is broadcasted to all nodes on the Bitcoin network

Mining, transaction and other relevant concepts will become clearer in the following sections in this unit.

2.2 Digital keys & Addresses

On the Bitcoin network, possession of bitcoins and transfer of value via transactions is rely upon cryptographic keys, private keys, public keys, and addresses. Here, we will see that how private and public keys are used in the Bitcoin network. Elliptic Curve Cryptography (ECC), a public key cryptography, is used to generate public and private key pairs in the Bitcoin network.

2.2.1 Private keys in Bitcoin

Private keys are required to be kept safe and normally resides only on the owner's side. Private keys are used to digitally sign the transactions proving the ownership of the bitcoins. Private keys are fundamentally 256-bit numbers randomly chosen in the range specified by the secp256k1 ECDSA curve recommendation. Any randomly chosen 256-bit number from 0x1 to 0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140 is a valid private key. Private keys are usually encoded using Wallet Import Format (WIF) in order to make them easier to copy and use. It is a way to represent the full size private key in a different format. WIF can be converted into a private key and vice versa. The steps are described here.

The following is an example of a private key:
A3ED7EC8A03667180D01FB4251A546C2B9F2FE33507C68B7D9D4E1FA5714195201

When it is converted into WIF format it looks like this:
L2iN7umV7kbr6LuCmgM27rBnptGbDVc8g4ZBm6EbgTPQXnj1RCZP

Also, mini private key format is sometimes used to create the private key with a maximum of up to 30 characters in order to allow storage where physical space is limited, for example, etching on physical coins or encoding in damage-resistant QR codes. The QR code becomes more damage resistant because more dots can be used for error correction and less for encoding the private key. The private key encoded using mini private key format is also sometimes called minikey. The first character of mini private key is always uppercase letter S. A mini private key can be converted into a normal size private key but an existing normal size private key cannot be converted into a mini private key. This format was used in Casascius physical bitcoins. The Bitcoin core client also allows the encryption of the wallet that contains the private keys.

2.2.2 Public keys in Bitcoin

Public keys exist on the blockchain and all network participants can see it. Public keys are derived from private keys due to their special mathematical relationship with the private keys. Once a transaction signed with the private key is broadcasted on the Bitcoin network, public keys are used by the nodes to verify that the transaction has indeed been signed with the corresponding private key. This process of verification proves the ownership of the bitcoin.

Bitcoin uses ECC based on the secp256k1 standard. More specifically it makes use of ECDSA to ensure that funds remain secure and can only be spent by the legitimate owner. A

public key is 256-bits in length. Public keys can be represented in an uncompressed or compressed format. Public keys are fundamentally x and y coordinates on an elliptic curve. In an uncompressed format public keys are presented with a prefix of 0x4 in a hexadecimal format. The x and y coordinates are both 32-bit in length. In total, the compressed public key is 33-bytes long as compared to 65-bytes in the uncompressed format. The compressed version of public keys includes only the x part, since the y part can be derived from it. This means that instead of storing both x and y as the public key only x can be stored with the information that if y is even or odd.

Initially, Bitcoin client used uncompressed keys, but starting from Bitcoin core client 0.6, compressed keys are used as standard. This resulted in almost 50% reduction of space used to store public keys in the blockchain.

Keys are identified by various prefixes, described as follows:

- Uncompressed public keys use 0x04 as the prefix
- Compressed public key starts with 0x03 if the y 32-bit part of the public key is odd
- Compressed public key starts with 0x02 if the y 32-bit part of the public key is even

2.2.2 Addresses in Bitcoin

A bitcoin address is created by taking the corresponding public key of a private key and hashing it twice, first with the SHA-256 algorithm and then with RIPEMD-160. The resultant 160-bit hash is then prefixed with a version number and finally encoded with a Base58Check encoding scheme. The bitcoin addresses are 26-35 characters long and begin with digit 1 or 3. A typical bitcoin address looks like a string shown here:

1ANAguGG8bikEv2fYsTBnRUmx7QUcK58wt

Currently, there are two types of addresses, the commonly used P2PKH and another P2SH type, starting with number 1 and 3, respectively. In the early days, Bitcoin used direct Pay to Pubkey, which is now superseded by P2PKH. These types will be explained later in the chapter. However, direct Pay to Pubkey is still used in Bitcoin for coinbase addresses. Addresses should not be used more than once; otherwise, privacy and security issues can arise. Avoiding address reuse circumvents anonymity issues to an extent, Bitcoin has some other security issues as well, such as transaction malleability, Sybil attacks, race attacks and selfish mining which require different approaches to resolve. Transaction malleability has been resolved with so-called Segregated Witness soft fork upgrade of the Bitcoin protocol. This concept will be explained later.

Base58Check encoding

Bitcoin addresses are encoded using the Base58Check encoding. This encoding is used to limit the confusion between various characters, such as 0OIl as they can look the same in different fonts. The encoding basically takes the binary byte arrays and converts them into human-readable strings. This string is composed by utilizing a set of 58 alphanumeric symbols.

Vanity addresses

As bitcoin addresses are based on base-58 encoding, it is possible to generate addresses that contain human readable messages, called as vanity address. Vanity addresses are generated using a purely brute-force method. An example of a paper wallet with vanity address is shown in the following screenshot.

Fig2.2 Sample vanity address

on the right-hand bottom corner the public vanity address with QR code is displayed. The paper wallets can be stored physically as an alternative to electronic storage of private keys.

Multisignature addresses

As the name implies, these addresses require multiple private keys. In practical terms, it means that in order to release the coins a certain set of signatures is required. This is also known as M-of-N MultiSig. Here, M represents threshold or the minimum number of signatures required from N number of keys to release the bitcoins.

2.3 Transactions

Transactions are at the core of the bitcoin ecosystem. Transactions can be as simple as just sending some bitcoins to a bitcoin address, or it can be quite complex depending on the requirements. Each transaction is composed of at least one input and output. Inputs can be thought of as coins being spent that have been created in a previous transaction and outputs as coins being created. If a transaction is minting new coins, then there is no input and therefore no signature is needed. If a transaction is to send coins to some other user (a bitcoin address), then it needs to be signed by the sender with their private key and a reference is also required to the previous transaction in order to show the origin of the coins. Coins are, in fact, unspent transaction outputs represented in Satoshis. Transactions are not encrypted and are publicly visible in the blockchain. Blocks are made up of transactions and these can be viewed using any online blockchain explorer.

2.3.1 The transaction life cycle

The following steps describe the transaction life cycle:

1. A user/sender sends a transaction using wallet software or some other interface.
2. The wallet software signs the transaction using the sender's private key.
3. The transaction is broadcasted to the Bitcoin network using a flooding algorithm.
4. Mining nodes (miners) who are listening for the transactions verify and include this transaction in the next block to be mined. Just before the transaction are placed in the block they are placed in a special memory buffer called transaction pool. The purpose of the transaction pool is explained in the next section.
5. Mining starts, which is a process by which the blockchain is secured and new coins are generated as a reward for the miners who spend appropriate computational resources.
6. Once a miner solves the PoW problem it broadcasts the newly mined block to the network. PoW is explained in detail later in this chapter.
7. The nodes verify the block and propagate the block further, and confirmations start to generate.
8. Finally, the confirmations start to appear in the receiver's wallet and after approximately three confirmations, the transaction is considered finalized and confirmed. However, three to six is just a recommended number; the transaction can be considered final even after the first confirmation. The key idea behind waiting for six confirmations is that the probability of double spending is virtually eliminated after three confirmations.

2.3.2 Transaction fee

Transaction fees are charged by the miners. The fee charged is dependent upon the size and weight of the transaction. Transaction fees are calculated by subtracting the sum of the inputs and the sum of the outputs.

A simple formula can be used: fee = sum(inputs) - sum(outputs)

The fees are used as an incentive for miners to encourage them to include a user transaction in the block the miners are creating. All transactions end up in the memory pool, from where miners pick up transactions based on their priority to include them in the proposed block. The calculation of priority is introduced later in this chapter; however, from a transaction fee point of view, a transaction with a higher fee will be picked up sooner by the miners.

There are different rules based on which fee is calculated for various types of actions, such as sending transactions, inclusion in blocks, and relaying by nodes. Fees are not fixed by the Bitcoin protocol and are not mandatory; even a transaction with no fee will be processed in due course but may take a very long time. This is however no longer practical due to the high volume of transactions and competing investors on the Bitcoin network, therefore it is advisable to provide a fee always. The time for transaction confirmation usually ranges from 10 minutes to over 12 hours in some cases. Transaction time is dependent on transaction fees and network activity. If the network is very busy then naturally transactions will take longer to process and if you pay a higher fee then your transaction is more likely to be picked by miners first due to additional incentive of the higher fee.

Transaction pools, also known as memory pools, are basically created in local memory (computer RAM) by nodes in order to maintain a temporary list of transactions that are not yet confirmed in a block. Transactions are included in a block after passing verification and based on their priority.

2.3.3 The transaction data structure

A transaction at a high level contains metadata, inputs, and outputs. Transactions are combined to create a block.

The transaction data structure is shown in the following table:

Field	Size	Description
Version number	4 bytes	Used to specify rules to be used by the miners and nodes for transaction processing.
Input counter	1-9 bytes	The number (positive integer) of inputs included in the transaction.

List of inputs	Variable	Each input is composed of several fields, including Previous Tx hash, Previous Txout-index, Txin-script length, Txin-script, and optional sequence number. The first transaction in a block is also called a coinbase transaction. It specifies one or more transaction inputs.
Output counter	1-9 bytes	A positive integer representing the number of outputs.
List of outputs	Variable	Outputs included in the transaction.
Lock time	4 bytes	This field defines the earliest time when a transaction becomes valid. It is either a Unix timestamp or block height.

Table 2.3.3 Transaction Data Structure

2.3.4 Script Language

Bitcoin uses a simple stack-based language called script to describe how bitcoins can be spent and transferred. This scripting language is based on a Forth programming language like syntax and uses a reverse polish notation in which every operand is followed by its operators. It is evaluated from the left to the right using a Last In, First Out (LIFO) stack.

Scripts use various opcodes or instructions to define their operation. Opcodes are also known as words, commands, or functions. Earlier versions of the Bitcoin node had a few opcodes that are no longer used due to bugs discovered in their design.

The various categories of the scripting opcodes are constants, flow control, stack, bitwise logic, splice, arithmetic, cryptography, and lock time.

A transaction script is evaluated by combining *ScriptSig* and *ScriptPubKey*. *ScriptSig* is the unlocking script, whereas, *ScriptPubKey* is the locking script.

This is how a transaction to be spent is evaluated:

1. First, it is unlocked and then it is spent
2. *ScriptSig* is provided by the user who wishes to unlock the transaction
3. *ScriptPubkey* is part of the transaction output and specifies the conditions that need to be fulfilled in order to spend the output
4. In other words, outputs are locked by *ScriptPubKey* that contains the conditions, when met will unlock the output, and coins can then be redeemed

2.3.5 Contracts

As defined in the Bitcoin core developer guide, contracts are basically transactions that use the Bitcoin system to enforce a financial agreement. This is a simple definition but has far-reaching consequences as it allows users to design complex contracts that can be used in many

real-world scenarios. Contracts allow the development of a completely decentralized, independent, and reduced risk platform.

Various contracts, such as escrow, arbitration, and micropayment channels, can be built using the Bitcoin scripting language. The current implementation of a script is very limited, but various types of contracts are still possible to develop. For example, the release of funds only if multiple parties sign the transaction or perhaps the release of funds only after a certain time has elapsed. Both of these scenarios can be realized using *multisig andtransaction* lock time options.

2.3.6 Transaction verification

This verification process is performed by Bitcoin nodes as follows:

1. Check the syntax and ensure that the syntax and data structure of the transaction conforms to the rules provided by the protocol.
2. Verify that no transaction inputs and outputs are empty.
3. Check whether the size in bytes is less than the maximum block size.
4. The output value must be in the allowed money range (0 to 21 million BTC).
5. All inputs must have a specified previous output, except for coinbase transactions, which should not be relayed.
6. Verify that nLockTime must not exceed 31-bits. (nLockTime specifies the time before which transaction will not be included in the block.)
7. For a transaction to be valid, it should not be less than 100 bytes.
8. The number of signature operations in a standard transaction should be less than or not more than two.
9. Reject nonstandard transactions; for example, ScriptSig is allowed to only push numbers on the stack. ScriptPubkey not passing the isStandard() checks. The isStandard() checks specify that only standard transactions are allowed.
10. A transaction is rejected if there is already a matching transaction in the pool or in a block in the main branch.
11. The transaction will be rejected if the referenced output for each input exists in any other transaction in the pool.
12. For each input, there must exist a referenced output unspent transaction.
13. For each input, if the referenced output transaction is the coinbase, it must have at least 100 confirmations; otherwise, the transaction will be rejected.

14. For each input, if the referenced output does not exist or has been spent already, the transaction will be rejected.
15. Using the referenced output transactions to get input values, verify that each input value, as well as the sum, is in the allowed range of 0-21 million BTC. Reject the transaction if the sum of input values is less than the sum of output values.
16. Reject the transaction if the transaction fee would be too low to get into an empty block.
17. Each input unlocking script must have corresponding valid output scripts.

2.3.7 Transaction malleability

Transaction malleability in Bitcoin was introduced due to a bug in the bitcoin implementation. Due to this bug, it became possible for an adversary to change the transaction ID of a transaction, thus resulting in a scenario where it would appear that a certain transaction has not been executed. This can allow scenarios where double deposits or withdrawals can occur. In other words, this bug allows the changing of the unique ID of a Bitcoin transaction before it is confirmed. If the ID is changed before confirmation, it would seem that the transaction did not occur at all which can then allow these attacks.

2.4 Blockchain Structure

Now, let us discuss detailed view of the blockchain structure. Blockchain is a public ledger of a timestamped, ordered, and immutable list of all transactions on the Bitcoin network. Each block is identified by a hash in the chain and is linked to its previous block by referencing the previous block's hash. Blockchain is a chain of blocks where each block is linked to its previous block by referencing the previous block header's hash. This linking makes sure that no transaction can be modified unless the block that records it and all blocks that follow it are also modified. The first block is not linked to any previous block and is known as the genesis block.

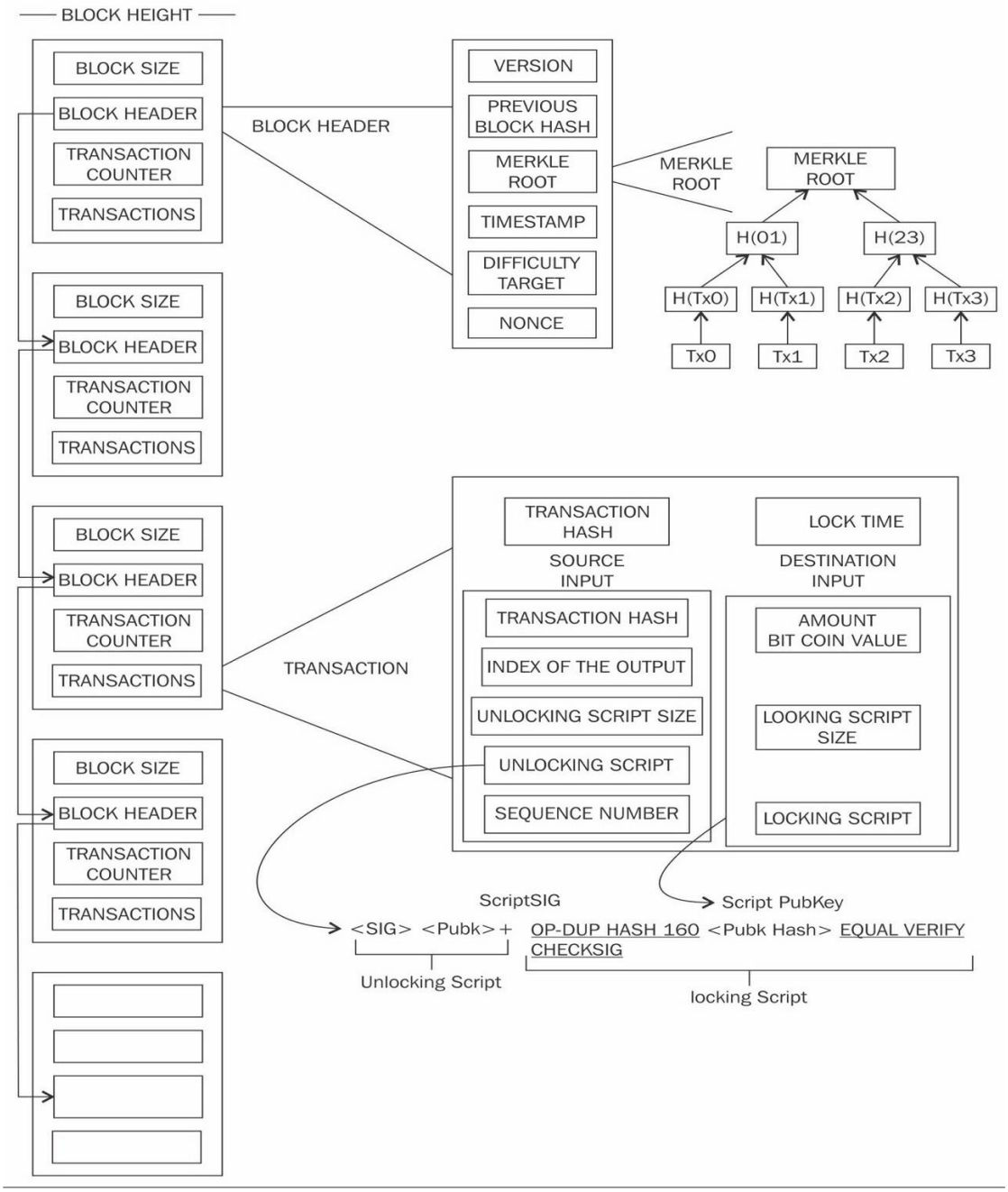

Fig 2.4 Overall structure of Block, Header

The preceding diagram shows a high-level overview of the Bitcoin blockchain. On the left-hand side blocks are starting from top to bottom. Each block contains transactions and block headers which are further magnified on the right-hand side. On the top, first, block header is expanded to show various elements within the block header. Then on the right-hand side the Merkle root element of the block header is shown in magnified view which shows that how Merkle root is calculated. Further down transactions are also magnified to show the structure of a transaction and the elements that it contains. Also, note that transactions are then further elaborated by showing that what locking and unlocking scripts look like.

2.5 Mining

Mining is a process by which new blocks are added to the blockchain. Blocks contain transactions that are validated via the mining process by mining nodes on the Bitcoin network. Blocks, once mined and verified are added to the blockchain which keeps the blockchain growing. This process is resource-intensive due to the requirements of PoW where miners compete in order to find a number which is less than the difficulty target of the network. This difficulty in finding the correct value (also called sometimes the mathematical puzzle) is there to ensure that the required resources have been spent by miners before a new proposed block can be accepted.

New coins are minted by the miners by solving the PoW problem, also known as partial hash inversion problem. This process consumes a high amount of resources including computing power and electricity. This process also secures the system against frauds and double spending attacks while adding more virtual currency to the Bitcoin ecosystem. Roughly one new block is created (mined) every 10 minutes to control the frequency of generation of bitcoins.

This frequency needs to be maintained by the Bitcoin network and is encoded in the bitcoin core clients in order to control the money supply. Miners are rewarded with new coins if and when they discover new blocks by solving PoW. Miners are paid transaction fees in return for including transactions in their proposed blocks. New blocks are created at an approximate fixed rate of every 10 minutes. The rate of creation of new bitcoins decreases by 50%, every 210,000 blocks, roughly every 4 years. When bitcoin was initially introduced, the block reward was 50 bitcoins; then in 2012, this was reduced to 25 bitcoins. In July 2016, this was further reduced to 12.5 coins (12 coins) and the next reduction is estimated to be on July 4, 2020. This will reduce the coin reward further down to approximately six coins.

Approximately 144 blocks, that is, 1,728 bitcoins are generated per day. The number of actual coins can vary per day; however, the number of blocks remains at 144 per day. Bitcoin supply is also limited and in 2140, almost 21 million bitcoins will be finally created and no new bitcoins can be created after that. Bitcoin miners, however, will still be able to profit from the ecosystem by charging transaction fees.

2.5.1 Tasks of the miners

Once a node connects to the bitcoin network, there are several tasks that a bitcoin miner performs:

1. Synching up with the network: Once a new node joins the bitcoin network, it downloads the blockchain by requesting historical blocks from other nodes. This is mentioned here in the context of the bitcoin miner; however, this not necessarily a task only for a miner.

2. Transaction validation: Transactions broadcasted on the network are validated by full nodes by verifying and validating signatures and outputs.

3. Block validation: Miners and full nodes can start validating blocks received by them by evaluating them against certain rules. This includes the verification of each transaction in the block along with verification of the nonce value.

4. Create a new block: Miners propose a new block by combining transactions broadcasted on the network after validating them.

5. Perform Proof of Work: This task is the core of the mining process and this is where miners find a valid block by solving a computational puzzle. The block header contains a 32-bit nonce field and miners are required to repeatedly vary the nonce until the resultant hash is less than a predetermined target.

6. Fetch reward: Once a node solves the hash puzzle (PoW), it immediately broadcasts the results, and other nodes verify it and accept the block. There is a slight chance that the newly minted block will not be accepted by other miners on the network due to a clash with another block found at roughly the same time, but once accepted, the miner is rewarded with 12.5 bitcoins and any associated transaction fees.

2.5.2 The Mining Algorithm

The mining algorithm consists of the following steps.

1. The previous block's header is retrieved from the bitcoin network.
2. Assemble a set of transactions broadcasted on the network into a block to be proposed.
3. Compute the double hash of the previous block's header combined with a nonce and the newly proposed block using the SHA-256 algorithm.
4. Check if the resultant hash is lower than the current difficulty level (target) then PoW is solved. As a result of successful PoW the discovered block is broadcasted to the network and miners fetch the reward.
5. If the resultant hash is not less than the current difficulty level (target), then repeat the process after incrementing the nonce. As the hash rate of the bitcoin network increased, the total amount of 32-bit nonce was exhausted too quickly. In order to address this issue, the extra nonce solution was implemented, whereby the coin base transaction is used as a source of extra nonce to provide a larger range of nonce to be searched by the miners.

This process can be visualized by using the following flowchart:

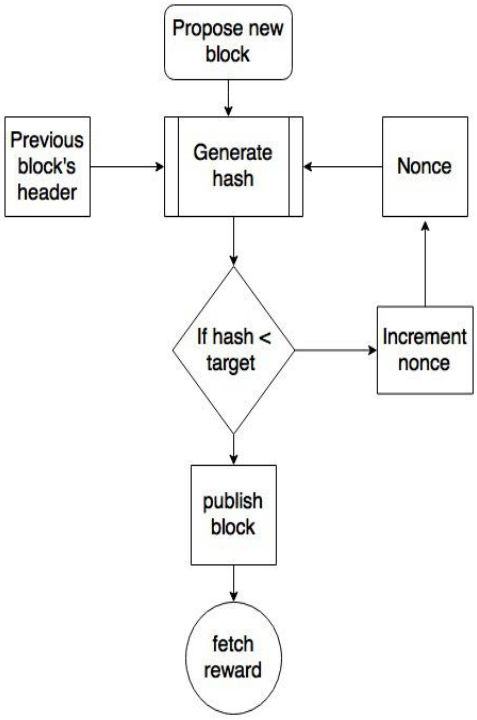

Fig.2.5.2 Mining Process

When Bitcoin started in 2009 the mining reward used to be 50 bitcoins. After every 210,000 blocks, the block reward halves. In November 2012 it halved down to 25 bitcoins. Currently, it is 12.5 BTC per block since July 2016. Next halving is on Friday, 12 June 2020 after which the block reward will be reduced down to 6.25 BTC per block. This mechanism is hardcoded in Bitcoin to regulate, control inflation and limit the supply of bitcoins

Mining difficulty increased over time and bitcoins that could be mined by single CPU laptop computers now require dedicated mining centers to solve the hash puzzle. The current difficulty level can be queried using the Bitcoin command-line interface using the following command:

```
$ bitcoin-cli getdifficulty
1452839779145
```

This number represents the difficulty level of the Bitcoin network. Recall from previous sections that miners compete to find a solution to a problem. This number, in fact shows, that how difficult it is to find a hash which is lower than the network difficulty target. All successfully mined blocks must contain a hash that is less than this target number. This number is updated every 2 weeks or 2016 blocks to ensure that on average 10-minute block generation time is maintained.

2.6 Bitcoin Network and Payments

In this section, we will present the Bitcoin network, relevant network protocols, and wallets. We will explore different types of wallets that are available for bitcoin. Moreover, we will examine how the Bitcoin protocol works and the types of messages exchanged on the network between nodes, during various node and network operations.

2.6.1 The Bitcoin Network

The Bitcoin network is a peer-to-peer network where nodes exchange transactions and blocks. There are different types of nodes on the network. There are two main types of nodes, full nodes and SPV nodes. Full nodes, as the name implies, are implementations of Bitcoin core clients performing the wallet, miner, full blockchain storage, and network routing functions. However, it is not necessary to perform all these functions.

Simple Payment Verification (SPV) nodes or lightweight clients perform only wallet and network routing functionality. The latest version of Bitcoin protocol is 70015 and was introduced with Bitcoin core client 0.13.2.

Some nodes prefer to be full blockchain nodes with complete blockchain as they are more secure and play a vital role in block propagation while some nodes perform network routing functions only but do not perform mining or store private keys (the wallet function). Another type is solo miner nodes that can perform mining, store full blockchain, and act as a Bitcoin network routing node.

There are a few nonstandard but heavily used nodes that are called pool protocol servers. These nodes make use of alternative protocols, such as the stratum protocol. These nodes are used in mining pools. Nodes that only compute hashes use the stratum protocol to submit their solutions to the mining pool. Some nodes perform only mining functions and are called mining nodes. It is possible to run an SPV client which runs a wallet and network routing function without a blockchain. SPV clients only download the headers of the blocks whiles yncing with the network and when required they can request transactions from full nodes. The verification of transactions is possible by using Merkle root in the block header with Merkle branch to prove that the transaction is present in a block in the blockchain.

Most protocols on the internet are line-based, which means that each line is delimited by a carriage return (\r) and newline (\n) character. Stratum is also a line-based protocol that makes use of plain TCP sockets and human-readable JSON-RPC to operate and communicate between nodes. Stratum is commonly used to connect to mining pools.

The Bitcoin network is identified by its different magic values. A list is shown as follows:

Bitcoin network.

Network	Magic value	Hex
main	0xD9B4BEF9	F9 BE B4 D9
testnet3	0x0709110B	0B 11 09 07

Magic values are used to indicate the message origin network.

A full node performs four functions: wallet, miner, blockchain, and the network routing node. Before we examine that how Bitcoin discovery protocol and block synchronization works, we need to understand that what are the different types of messages that Bitcoin protocol uses. The list of message types is provided here.

There are 27 types of protocol messages in total, but they're likely to increase over time as the protocol grows. The most commonly used protocol messages and their explanation are listed as follows:

version: This is the first message that a node sends out to the network, advertising its version and block count. The remote node then replies with the same information and the connection is then established.

verack: This is the response of the version message accepting the connection request.

inv: This is used by nodes to advertise their knowledge of blocks and transactions.

getdata: This is a response to inv, requesting a single block or transaction identified by its hash.

getblocks: This returns an inv packet containing the list of all blocks starting after the last known hash or 500 blocks.

getheaders: This is used to request block headers in a specified range.

tx: This is used to send a transaction as a response to the getdata protocol message.

2.7 The Wallets

The wallet software is used to store private or public keys and Bitcoin address. It performs various functions, such as receiving and sending bitcoins. Nowadays, software usually offers both functionalities: Bitcoin client and wallet. On the disk, the Bitcoin core client wallets are stored as the Berkeley DB file:

```
$ file wallet.dat
wallet.dat: Berkeley DB (B-tree, version 9, native byte-order)
```

Private keys are generated by randomly choosing a 256-bit number by wallet software. The rules of generation are predefined and were discussed in Chapter 4, Public Key Cryptography.

Private keys are used by wallets to sign the outgoing transactions. Wallets do not store any coins, and there is no concept of wallets storing balance or coins for a user. In fact, in the Bitcoin network, coins do not exist; instead, only transaction information is stored on the blockchain (more precisely, UTXO, unspent outputs), which are then used to calculate the number of bitcoins.

In Bitcoin, there are different types of wallets that can be used to store private keys. As a software program, they also provide some functions to the users to manage and carry out transactions on the Bitcoin network.

In Bitcoin, there are different types of wallets that can be used to store private keys. As a software program, they also provide some functions to the users to manage and carry out transactions on the Bitcoin network.

The various types of wallets are non-deterministic, deterministic, hierarchical deterministic, paper wallets, online and mobile wallets.

Non-deterministic wallets

These wallets contain randomly generated private keys and are also called just a bunch of key wallets. The Bitcoin core client generates some keys when first started and generates keys as and when required. Managing a large number of keys is very difficult and an error-prone process can lead to theft and loss of coins. Moreover, there is a need to create regular backups of the keys and protect them appropriately, for example, by encrypting them in order to prevent theft or loss.

Deterministic wallets

In this type of wallet, keys are derived out of a seed value via hash functions. This seed number is generated randomly and is commonly represented by human-readable mnemonic code words. Mnemonic code words are defined in BIP 39, a Bitcoin improvement proposal for mnemonic code for generating deterministic keys. This BIP is available at https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki. This phrase can be used to recover all keys and makes private key management comparatively easier.

Hierarchical Deterministic wallets

Defined in BIP32 and BIP44, Hierarchical Deterministic (HD) wallets store keys in a tree structure derived from a seed. The seed generates the parent key (master key), which is used to generate child keys and, subsequently, grandchild keys. Key generation in HD wallets does not generate keys directly; instead, it produces some information (private key generation information) that can be used to generate a sequence of private keys. The complete hierarchy of private keys in an HD wallet is easily recoverable if the master private key is known. It is

because of this property that HD wallets are very easy to maintain and are highly portable. There are many free and commercially available HD wallets available. For example, Trezor (https://trezor.io), Jaxx (https://jaxx.io/) and Electrum (https://electrum.org/).

Brain wallets

The master private key can also be derived from the hash of passwords that are memorized. The key idea is that this passphrase is used to derive the private key and if used in HD wallets, this can result in a full HD wallet that is derived from a single memorized password. This is known as a brain wallet. This method is prone to password guessing and brute force attacks but techniques such as key stretching can be used to slow down the progress made by the attacker.

Paper wallets

As the name implies, this is a paper-based wallet with the required key material printed on it. It requires physical security to be stored. Paper wallets can be generated online from various service providers, such .0 as https://bitcoinpaperwallet.com/ or https://www.bitaddress.org/.

Hardware wallets

Another method is to use a tamper-resistant device to store keys. This tamper-resistant device can be custom built or with the advent of NFC-enabled phones, this can also be a Secure Element (SE) in NFC phones. Trezor and Ledger wallets (various types) are the most commonly used Bitcoin hardware wallets.

Online wallets

Online wallets, as the name implies, are stored entirely online and are provided as a service usually through the cloud. They provide a web interface to the users to manage their wallets and perform various functions such as making and receiving payments. They are easy to use but imply that the user trusts the online wallet service provider. An example of online wallet is GreenAddress, which is available at https://greenaddress.it/en/.

Mobile wallets

Mobile wallets, as the name suggests, are installed on mobile devices. They can provide various methods to make payments, most notably the ability to use smartphone cameras to scan QR codes quickly and make payments. Blockchain, breadwallet, Copay, and Jaxx are some of the widely used mobile wallets.

2.8 Choice of Wallet:

The choice of Bitcoin wallet depends on several factors such as security, ease of use, and available features. Out of all these attributes, security of course comes first and when making a decision about which wallet to use, security should be of paramount importance. Hardware wallets tend to be more secure as compared to web wallets because of their tamper resistant design. Web wallets by nature are hosted on websites, which may not be as secure as a tamper resistant hardware device. Generally, mobile wallets for smartphone devices are quite popular due to a balanced combination of features and security. There are many companies offering these wallets on the iOS App Store and Android Play. It is however quite difficult to suggest that which type of wallet should be used, it also depends on personal preferences and features available in a wallet. It is advisable that security should be kept in mind while making decision on which wallet to choose.

2.9 Bitcoin payments

Bitcoins can be accepted as payments using various techniques. Bitcoin is not recognized as a legal currency in many jurisdictions, but it is increasingly being accepted as a payment method by many online merchants and ecommerce websites. There are a number of ways in which buyers can pay the business that accepts bitcoins. For example, in an online shop, Bitcoin merchant solutions can be used, whereas in traditional, physical shops, point of sale terminals and other specialized hardware can be used. Customers can simply scan the QR code with the seller's payment URI in it and pay using their mobile devices. Bitcoin URIs allow users to make payments by simply clicking on links or scanning QR codes. Uniform Resource Identifier (URI) is basically a string that represents the transaction information. It is defined in BIP 21. The QR code can be displayed near the point of the sale terminal. Nearly all Bitcoin wallets support this feature.

Various payment solutions, such as XBTerminal and 34 Bytes bitcoin Point of Sale (POS) terminal are available commercially.

Generally, these solutions work by following these steps:

1. The sales person enters the amount of money to be charged in Fiat currency, for example, US Dollars
2. Once the value is entered in the system the terminal prints a receipt with QR code on it and other relevant information such as amount
3. The customer can then scan this QR code using their mobile Bitcoin wallet to send the payment to the Bitcoin address of the seller embedded within the QR code

4. Once the payment is received on the designated Bitcoin address, a receipt is printed out as a physical evidence of sale.

The bitcoin payment processor, offered by many online service providers, allows integration with e-commerce websites. There are many options available. These payment processors can be used to accept bitcoins as payments. Some service providers also allow secure storage of bitcoins, such as bitpay, https://bitpay.com.

2.10 Alternate Coins

The success of Ethereum and Bitcoin has resulted in many alternative currency projects and thereby alternate coins. Alternative approaches to bitcoin can be divided broadly into two categories, based on the primary purpose of their development. If the primary goal is to build a decentralized blockchain platform, they are called alternative chains; if the sole purpose of the alternative project is to introduce a new virtual currency, it is called an altcoin.

Altcoins must be able to attract new users, trades, and miners otherwise the currency will have no value. Currency gains its value, especially in the virtual currency space, due to the network effect and its acceptability by the community. If a coin fails to attract enough users then soon it will be forgotten. Users can be attracted by providing an initial amount of coins and can be achieved by using various methods.

Methods of providing an initial number of altcoins are given as follows:

Create a new blockchain: Altcoins can create a new blockchain and allocate coins to initial miners, but this approach is now unpopular due to many scam schemes or pump and dump schemes where initial miners made a profit with the launch of a new currency and then disappeared.

Proof of Burn (PoB): Another approach to allocating initial funds to a new altcoin is PoB, also called a ***one-way peg or price ceiling***. In this method users permanently destroy a certain quantity of bitcoins in proportion to the quantity of altcoins to be claimed. For example, if ten bitcoins were destroyed then altcoins can have a value no greater than some bitcoins destroyed. This means that bitcoins are being converted into altcoins by burning them.

Proof of ownership: Instead of permanently destroying bitcoins, an alternative method is to prove that users own a certain number of bitcoins. This proof of ownership can be used to claim altcoins by tethering altcoin blocks to Bitcoin blocks.

Pegged sidechains: Sidechains are blockchains separate from the bitcoin network but bitcoins can be transferred to them. Altcoins can also be transferred back to the bitcoin network. This concept is called a two-way peg.

2.10.1 Theoretical Foundations of Altcoins

In this section, various theoretical concepts are introduced that has been developed with the introduction of different altcoins in the past few years.

Alternatives to Proof of Work (PoW)

The PoW scheme in the context of cryptocurrency was first used in Bitcoin and served as a mechanism to provide assurance that a miner had completed the required amount of work to find a block. This process in return provided decentralization, security, and stability for the blockchain. This is the primary vehicle in Bitcoin for providing decentralized distributed consensus. PoW schemes are required to have a much-desired property
called progress freeness, which means that the reward for consuming computational resources should be random and proportional to the contribution made by the miners. In this case, some chance of winning the block reward is given to even those miners who have comparatively less computational power.

Alternatives to Proof of Storage

Also known as proof of retrievability, this is another type of proof of useful work that requires storage of a large amount of data. Introduced by Microsoft Research, this scheme provides a useful benefit of distributed storage of archival data. Miners are required to store a pseudo, randomly-selected subset of large data to perform mining.

Proof of Stake (PoS)

This proof is also called virtual mining. This is another type of mining puzzle that has been proposed as an alternative to traditional PoW schemes. In this scheme, the idea is that users are required to demonstrate possession of a certain amount of currency (coins) thus proving that they have a stake in the coin. The simplest form of the stake is where mining is made comparatively easier for those users who demonstrably own larger amounts of digital currency. The benefits of this scheme are twofold; first acquiring large amounts of digital currency is relatively difficult as compared to buying high-end ASIC devices and second it results in saving computational resources.

2.10.1.1 Various stake types

Different type of stakes, such as Proof of coinage, Proof of Deposit, Proof of Burn and Proof of Activity, will now be introduced in the following subsections.

Proof of coinage

The age of a coin is the time since the coins were last used or held. This is a different approach from the usual form of PoS where mining is made easier for users who have the highest stake in the altcoin. In the coin-age based approach, the age of the coin (coinage) is reset every time a block is mined. The miner is rewarded for holding and not spending coins for a period of time. This mechanism has been implemented in Peercoin combined with PoW in a creative way.

The difficulty of mining puzzles (PoW) is inversely proportional to the coinage, meaning that if miners consume some coinage using coin-stake transactions, then the PoW requirements are relieved.

Proof of Deposit (PoD)

The core idea behind this scheme is that newly minted blocks by miners are made unspendable for a certain period. More precisely the coins get locked for a set number of blocks during the mining operation. The scheme works by allowing miners to perform mining at the cost of freezing a certain number of coins for some time.

Proof of Burn

As an alternate expenditure to computing power, PoB, in fact, destroys a certain number of bitcoins to get equivalent altcoins. This is commonly used when starting up a new coin projects as a means to provide a fair initial distribution. This can be considered an alternative mining scheme where the value of the new coins comes from the fact that previously a certain number of coins have been destroyed.

Proof of Activity (PoA)

This scheme is a hybrid of PoW and PoS. In this scheme, blocks are initially produced using PoW, but then each block randomly assigns three stakeholders that are required to digitally sign it. The validity of subsequent blocks is dependent on the successful signing of previously randomly chosen blocks.

2.10.1.2 Difficulty adjustment and retargeting algorithms

Another concept that has been introduced with the advent of bitcoin and altcoins is difficulty in retargeting algorithms. In bitcoin, a difficulty target is calculated simply by the following equation; however other coins

have either developed their algorithms or implemented modified versions of the bitcoin difficulty algorithm:

T = Time previous * time actual / 2016 * 10 min

The idea behind difficulty regulation in bitcoin is that a generation of 2016 blocks should take roughly around two weeks (inter-block time should be around 10 minutes). If it takes longer than two weeks to mine 2016 blocks, then the difficulty is decreased, and if it takes less than two weeks to mine 2016 blocks, then the difficulty is increased. When ASICs were introduced due to a high block generation rate, the difficulty increased exponentially, and that is one drawback of PoW algorithms that are not ASIC resistant. This leads to mining power centralization.

2.10.2 Bitcoin Limitations

Altcoins were developed specifically to address limitations in Bitcoin. The most prominent and widely discussed limitation is the lack of anonymity in Bitcoin. We will now discuss some of the limitations of Bitcoin.

2.10.2.1 Privacy & Anonymity

Various proposals have been made to address the privacy issue in Bitcoin. These proposals fall into three categories: mixing protocols, third-party mixing networks, and inherent anonymity.

A brief discussion of each category is presented as follows:

Mixing Protocols

These schemes are used to provide anonymity to bitcoin transactions. In this model, a mixing service provider, an intermediary or a shared wallet, is used. Users send coins to this shared wallet as a deposit, and then, the shared wallet can send some other coins to the destination. Users can also receive coins that were sent by others through this intermediary. This way the link between outputs and inputs is no longer there and transaction graph analysis will not be able to reveal the actual relationship between senders and receivers.

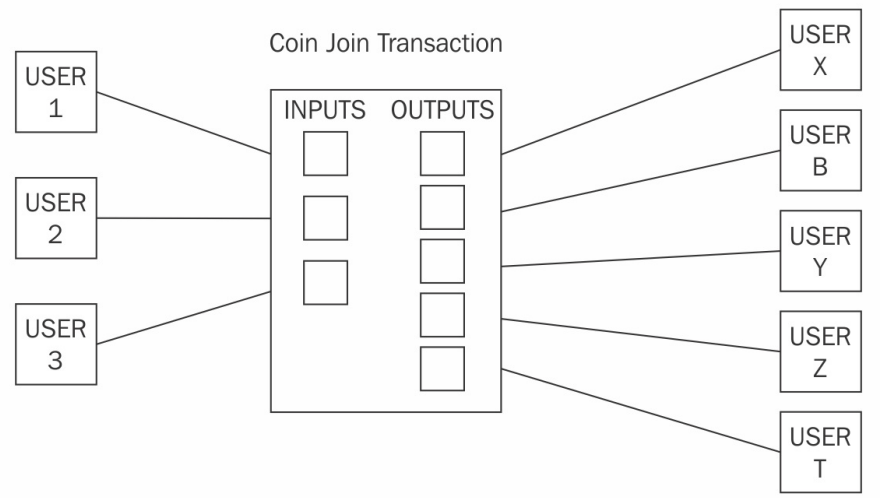

Fig 2.10.2.1 Three users joining their transaction into a single larger CoinJoin transaction

CoinJoin is one example of mixing protocols, where two transactions are joined together to form a single transaction while keeping the inputs and outputs unchanged. The core idea behind CoinJoin is to build a shared transaction that is signed by all participants. This technique improves privacy for all participants involved in the transactions.

Third-party mixing services

Various third-party mixing services are available, but if the service is centralized, then it poses the threat of tracing the mapping between senders and receivers because the mixing service knows about all inputs and outputs. Moreover, fully centralized miners even pose the risk of the administrators of the service stealing the coins.

Various services, with varying degrees of complexity, such as CoinShuffle, Coinmux, and Darksend in Dash coin are available that are based on the idea of CoinJoin (mixing) transactions. CoinShuffle is a decentralized alternative to traditional mixing services as it does not require a trusted third party. CoinJoin-based schemes, however, have some weaknesses, most prominently the possibility of launching a denial of service attack by users who committed to signing the transactions initially but now are not providing their signature, thus delaying or stopping joint transaction altogether.

Inherent anonymity

This category includes coins that support privacy inherently and is built into the design of the currency. The most popular is Zcash, which uses Zero-Knowledge Proofs (ZKP) to achieve anonymity. Other examples include Monero, which makes use of ring signatures to provide anonymous services.

2.10.3 Development of altcoins

Altcoin projects can be developed from a coding point of view by simply forking the bitcoin or another coin's source code, but this probably is not enough. Writing code or forking the code for an existing coin is the trivial part, the challenging issue is how to start a new currency so that investors and users can be attracted to it. Generally, the following steps are taken in order to start a new coin project. From a technical point of view, in the case of forking the code of another coin, for example, bitcoin, there are various parameters that can be changed to effectively create a new coin, which are given below:

Consensus algorithms

There is a choice of consensus algorithms available, for example PoW used in Bitcoin or PoS, used in Peercoin. There are also other algorithms available such as Proof of Capacity (PoC) and few others, but PoW and PoS are the most common choices.

Hashing algorithms

This is either SHA-256, Scrypt, X11, X13, X15, or any other hashing algorithm that is adequate for use as a consensus algorithm.

Difficulty adjustment algorithms

Various options are available in this category to provide difficulty retargeting mechanisms. The most prominent examples are KGW, DGW, Nite's Gravity Wave, and DigiShield. Also, all these algorithms can be tweaked based on requirements to produce different results; therefore, many variants are possible.

Inter-block time

This is the time elapsed between the generation of each block. For bitcoin the blocks are generated every 10 minutes, for litecoin it's 2.5 minutes. Any value can be used but an appropriate value is usually between a few minutes; if the generation time is too fast it might destabilize the blockchain, if it's too slow it may not attract many users.

Block rewards

A block reward is for the miner who solves the mining puzzle and is allowed to have a coinbase transaction that contains the reward. This used to be 50 coins in bitcoin initially and now many altcoins set this parameter to a very high number; for example, in Dogecoin it is 10,000, currently.

Reward halving rate

This is another important factor; in bitcoin, it is halved every 4 years and now is set to 12.5 bitcoins. It's a variable number that can be set to any time period or none at all depending on the requirements.

Block size and transaction size

This is another important factor that determines how high or low the transaction rate can be on the network. Block sizes in bitcoin are limited to 1 MB but in altcoins, it can vary depending on the requirements.

Interest rate

This property applies only to PoS systems where the owner of the coins can earn interest at a rate defined by the network in return for some coins that are held on the network as a stake to protect the network.

Coinage

This parameter defines how long the coin has to remain unspent in order for it to become eligible to be considered stake worthy.

Total supply of coins

This number sets the total limit of the coins that can ever be generated. For example, in Bitcoin the limit is 21 million, whereas in Dogecoin it's unlimited. This limit is fixed by the block reward.

2.11 Namecoin

Namecoin is the first fork of the Bitcoin source code. The key idea behind Namecoin is to provide improved decentralization, censorship resistance, privacy, security, and faster decentralized naming.

Namecoin provides the following three services:

Secure storage and transfer of names (keys)

Attachment of some value to the names by attaching up to 520 bytes of data

Production of a digital currency (Namecoin)

Namecoin also for the first time introduced merged mining, which allows a miner to mine on more than one chain simultaneously. The idea is simple but very effective: miners create a Namecoin block and produce a hash of that block. Then the hash is added to a Bitcoin block and miners solve that block at equal to or greater than the Namecoin block difficulty to prove that enough work has been contributed towards solving the Namecoin block.

The coinbase transaction is used to include the hash of that block. Then the hash is added to a Bitcoin block and miners solve that block at equal to or greater than the Namecoin block difficulty to prove that enough work has been contributed towards solving the Namecoin block.

The coinbase transaction is used to include the hash of the transactions from Namecoin (or any other altcoin if merged mining with that coin). The mining task is to solve Bitcoin blocks whose coinbase scriptSig contains a hash pointer to Namecoin (or any other altcoin) block. This is shown in the following diagram:

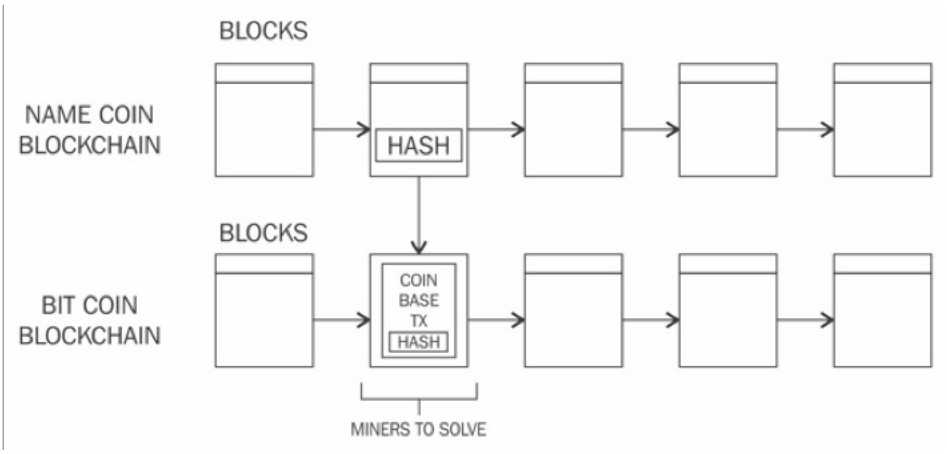

Fig 2.11 Merged mining Visulaization

If a miner manages to solve a hash at the bitcoin blockchain difficulty level, the bitcoin block is built and becomes part of the Bitcoin network. In this case, the Namecoin hash is ignored by the bitcoin blockchain. On the other hand, if a miner solves a block at Namecoin blockchain difficulty level a new block is created in the Namecoin blockchain. The core benefit of this scheme is that all the computational power spent by the miners contributes towards securing both Namecoin and Bitcoin.

2.12 Prime Coin

Primecoin is the first digital currency on the market that introduced a useful PoW, as opposed to Bitcoin's SHA256-based PoW. Primecoin uses searching prime numbers as a PoW. Not all types of prime number meet the requirements to be selected as PoW. Three types of prime numbers (known as Cunningham chain of the first kind, Cunningham chain of the second kind, and bi-twin chains) meet the requirements of a PoW algorithm to be used in cryptocurrencies.

The difficulty is dynamically adjusted via a continuous difficulty evaluation scheme in Primecoin blockchain. The efficient verification of PoW based on prime numbers is also of

high importance, because if verification is slow, then PoW is not suitable. Therefore, prime chains are selected as a PoW because finding prime chains gets difficult as the chain increases in length whereas verification remains quick enough to warrant being used as an efficient PoW algorithm. It is also important that once a PoW has been verified on a block, it must not be reusable on another block. This is accomplished in Primecoin by a combination of PoW certificates and hashing it with the header of the parent block in the child block. The PoW certificate is produced by linking the prime chain to the block header hash. It also requires that the block header's origin be divisible by the block header hash. If it is, it is divided and after division, the quotient is used as a PoW certificate. Another property of the adjustable difficulty of PoW algorithms is met by introducing difficulty adjustment every block instead of every 2,016, as is the case with bitcoin. This is a smoother approach as compared to bitcoin and allows readjustment in the case of sudden increases in hash power. Also, the total number of coins generated is community-driven, and there is no definite limit on the number of coins Primecoin can generate. Primecoins can be traded on major virtual currency trading exchanges. The current market cap of Primecoin is $17,482,507 at the time of writing (March, 2018). It is not very large but, because Primecoin is based on a novel idea and there is a dedicated community behind it, this continues to hold some market share.

2.13 Zcash

Zcash is the first currency that uses a specific type of ZKPs known as Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (ZK-SNARKs) to provide complete privacy to the user. These proofs are concise and easy to verify; however, setting up the initial public parameters is a complicated process. The latter include two keys: the proving key and verifying key. The process requires sampling some random numbers to construct the public parameters. The issue is that these random numbers, also called toxic waste, must be destroyed after the parameter generation in order to prevent counterfeiting of Zcash. For this purpose, the Zcash team came up with a multi-party computation protocol to generate the required public parameters collaboratively from independent locations to ensure that toxic waste is not created. Because these public parameters are required to be created by the Zcash team, it means that the participants in the ceremony are trusted. This is the reason why the ceremony was very open and conducted by making use of a multi-party computation mechanism.

This mechanism has a property whereby all of the participants in the ceremony will have to be compromised to compromise the final parameters. When the ceremony is completed all

participants physically destroyed the equipment used for private key generation. This action eliminates any trace of the participants' part of the private key on the equipment.

ZK-SNARKs must satisfy the properties for completeness, soundness, succinctness, and non-interactivity. Completeness means that there is a definite strategy for a prover to satisfy a verifier that an assertion is true. On the other hand, soundness means that no prover can convince the verifier that a false statement is true. Succinctness means that messages passed between the prover and verifier are tiny in size. Finally, the property non-interactive means that the verification of correctness of an assertion can be carried out without any interaction or very little interaction. Also, being a ZKP, the property of zero-knowledge needs to be met too.

Zcash developers have introduced the concept of a Decentralized Anonymous Payment scheme (DAP scheme) that is used in the Zcash network to enable direct and private payments. The transactions reveal no information about the origin, destination, and amount of the payments. There are two types of addresses available in Zcash, Z address and T address. Z addresses are based on ZKPs and provide privacy protection whereas T addresses are similar to those of bitcoin. A snapshot of various attributes of Zcash (after an initial slow start) is shown as follows:

Attribute	**Value**
Name	Zcash
Launch date	28/10/16
Main purpose	Currency
Currency Code	ZEC
Maximum coins	21 million
Block time	10 minutes
Consensus facilitation algorithm	Proof of Work (equihash)
Difficulty adjustment algorithm	DigiShield V3 (modified)
Mining hardware	CPU, GPU
Difficulty adjustment period	1 block

Zcash uses an efficient PoW scheme named asymmetric PoW (Equihash), which is based on the Generalized Birthday Problem. It allows very efficient verification. It is a memory-hard and ASIC-resistant function. A novel idea, “initial slow mining”, has been introduced with Zcash, which means that the block reward increases gradually over a period until it reaches the 20,000th block. This allows for initial scaling of the network and experimentation by early miners, and adjustment by Zcash developers if required. The slow

start did have an impact on price due to scarcity as the price of ZEC on its first day of launch reached roughly 25,000 USD.

Zcash can be bought on major digital currency sellers and exchanges such as CryptoGo (https://cryptogo.com). Another exchange where Zcash can be bought or sold is Crypto Robot 365 (https://cryptorobot365.com). There are multiple methods to mine Zcash. Currently, CPU and GPU mining are possible. Various commercial cloud mining pools also offer contracts for mining Zcash.

2.14 Smart Contracts

A smart contract is an electronic transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries.

A smart contract is a secure and unstoppable computer program representing an agreement that is automatically executable and enforceable.

Smart contracts usually operate by managing their internal state using a state machine model. This allows development of an effective framework for programming smart contracts, where the state of a contract is advanced further based on some predefined criteria and conditions.

A smart contract has the following four properties:

- Automatically executable
- Enforceable
- Semantically sound
- Secure and unstoppable

The first two properties are required as a minimum, whereas the latter two may not be required or implementable in some scenarios and can be relaxed. For example, a financial derivatives contract does not perhaps need to be semantically sound and unstoppable but should at least be automatically executable and enforceable at a fundamental level. On the other hand, a title deed needs to be semantically sound and complete, therefore, for it to be implemented as a smart contract, the language must be understood by both computers and people.

2.15 Ricardian Contracts

These contracts were used initially in a bond trading and payment system called Ricardo. The fundamental idea is to write a document that is understandable and acceptable by both a court of law and computer software. Ricardian contracts address the challenge of

issuance of value over the internet. It identifies the issuer and captures all the terms and clauses of the contract in a document to make it acceptable as a legally binding contract.

A Ricardian contract is a document that has several of the following properties:

- A contract offered by an issuer to holders
- A valuable right held by holders and managed by the issuer
- Easily readable by people (like a contract on paper)
- Readable by programs (parsable, like a database)
- Digitally signed
- Carries the keys and server information
- Allied with a unique and secure identifier

In practice, the contracts are implemented by producing a single document that contains the terms of the contract in legal prose and the required machine-readable tags. This document is digitally signed by the issuer using their private key. This document is then hashed using a message digest function to produce a hash by which the document can be identified. This hash is then further used and signed by parties during the performance of the contract to link each transaction, with the identifier hash thus serving as an evidence of intent. This is depicted in the next diagram, usually called a bowtie model.

2.15.1 The diagram shows number of elements:

The World of Law on the left-hand side from where the document originates. This document is a writtencontract in legal prose with some machine-readable tags. This document is then hashed. The resultant message digest is used as an identifier throughout the World of Accountancy, shown on the right-hand side of the diagram. The World of Accountancy element represents any accounting, trading, and information systems that are being used in the business to perform various business operations. The idea behind this flow is that the message digest generated by hashing the document is first used in a so-called genesis transaction, or first transaction, and then used in every transaction as an identifier throughout the operational execution of the contract. This way, a secure link is created between the original written contract and every transaction in the World of Accounting:

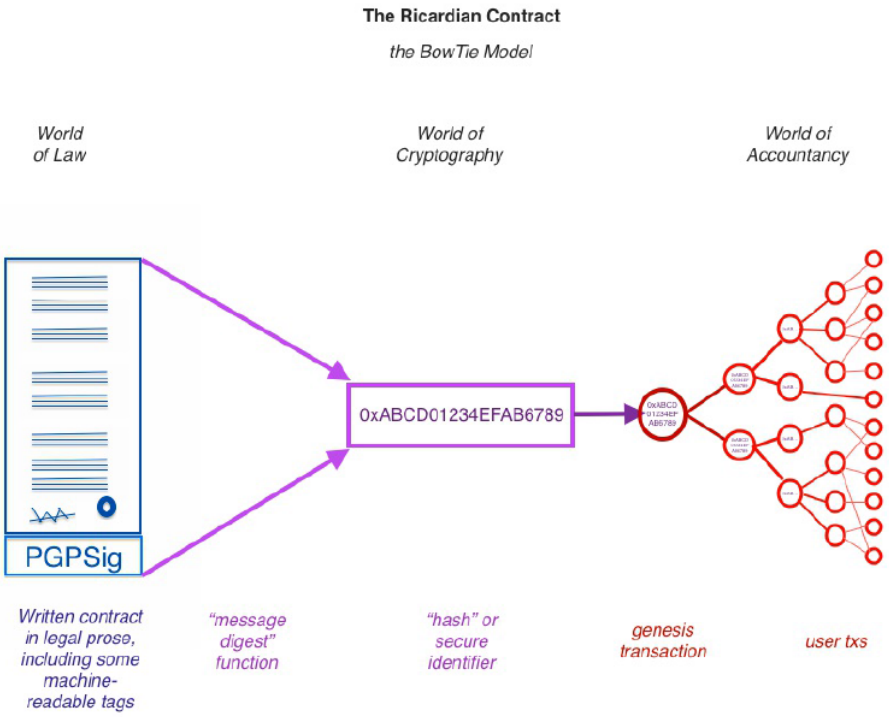

Fig. 2.15 Ricardian Contract bowtie diagram

A Ricardian contract is different from a smart contract in the sense that a smart contract does not include any contractual document and is focused purely on the execution of the contract. A Ricardian contract, on the other hand, is more concerned with the semantic richness and production of a document that contains contractual legal prose. The semantics of a contract can be divided into two types: operational semantics and denotational semantics.

The first type defines the actual execution, correctness, and safety of the contract, and the latter is concerned with the real-world meaning of the full contract. Some researchers have differentiated between smart contract code and smart legal contracts where a smart contract is only concerned with the execution of the contract. The second type encompasses both the denotational and operational semantics of a legal agreement. It perhaps makes sense to categorize smart contracts based on the difference between semantics, but it is better to consider smart contracts as a standalone entity that is capable of encoding legal prose and code (business logic) in it.

In Bitcoin, a straightforward implementation of basic smart contracts (conditional logic) can be observed, which is entirely oriented towards the execution and performance of the contract, whereas a Ricardian contract is more geared towards producing a document that is understandable by humans with some parts that a computer program can understand. This can be viewed as legal semantics versus operational performance (semantics versus performance) as shown in the following diagram. The diagram shows that Ricardian contracts are more

semantically-rich, whereas smart contracts are more performance-rich. A smart contract is made up to have both of these elements (performance and semantics) embedded together, which completes an ideal model of a smart contract. A Ricardian contract can be represented as a tuple of three objects, namely Prose, parameters, and code. Prose represents the legal contract in natural language; code represents the program that is a computer-understandable representation of legal prose; and parameters join the appropriate parts of the legal contract to the equivalent code. Ricardian contracts have been implemented in many systems, such as CommonAccord, OpenBazaar, OpenAssets, and Askemos.

2.16 Summary

This Unit started with the introduction to Bitcoin network, following it with a discussion on Bitcoin node discovery and block synchronization protocols. Moreover, we presented different types of network messages. Then we examined different types of Bitcoin wallets and discussed various attributes and features of each type. Following this, we looked at Bitcoin payments and payment processors. In the last section, we discussed Bitcoin innovations, which included topics such as Bitcoin Improvement Proposals, and advanced Bitcoin protocols.

Short Questions:

1. What is the smallest unit of Bitcoin that can be traded on the blockchain?
2. What are the two types of addresses?
3. What is multisignature address?
4. How a transaction to be spent is evaluated?
5. What is the significance of SPV (Single Payment Verification) node in Blockchain network?
6. What is the difference between deterministic and non-deterministic wallets?

Long Questions:

1. Describe, how a payment can be sent from one user to another on the Bitcoin network?
2. Explain the transaction life cycle.
3. Describe the process of transaction verification.
4. Explain the overall Blockchain Structure with neat diagram.
5. What is mining? Explain the step by step mining algorithm.

UNIT 3 ETHEREUM

Overview

An Introduction to the Ethereum is presented in this Unit. In this Unit, fundamentals and various theoretical concepts behind Etherium is presented. A discussion on various components, protocols, and algorithms relevant to the Ethereum blockchain will be given in detail so that you can understand the theory behind this blockchain paradigm. An introduction to wallet software, mining, and setting up Ethereum nodes is also presented.

Learning Objectives:

Study the components of Etherium and programming with Solidarity

3.1 Introduction

Vitalik Buterin (https://vitalik.ca) conceptualized Ethereum in November, 2013. It allows the development of arbitrary programs (smart contracts) for blockchain and decentralized applications. In Bitcoin, the scripting language is limited in nature and allows necessary operations only.

The first version of Ethereum, called Olympic, was released in May, 2015. Two months later, a version of Ethereum called Frontier was released in July, 2015. After about a year of this release, another version named Homestead with various improvements was released in March, 2016. Latest Ethereum release is called Byzantium which is the first part of the development phase of Ethereum called Metropolis. This release implemented a planned hard fork at block number 4,370,000 on October 16, 2017. The second part of this release called Constantinople is expected in 2018 but there is no exact time frame available yet. The final planned release of Ethereum is called Serenity and is envisaged to introduce the final version of PoS based blockchain instead of PoW. This chapter covers Byzantium, which at the time of writing is the latest version of Ethereum.

Formal specification of Ethereum has been described in the yellow paper which can be used to develop Ethereum implementations.

3.2 Ethereum Blockchain

Ethereum, like any other blockchain, can be visualized as a transaction-based state machine. The core idea is that in Ethereum blockchain, a genesis state is transformed into a final state by executing transactions incrementally. The final transformation is then accepted as the absolute undisputed version of the state.

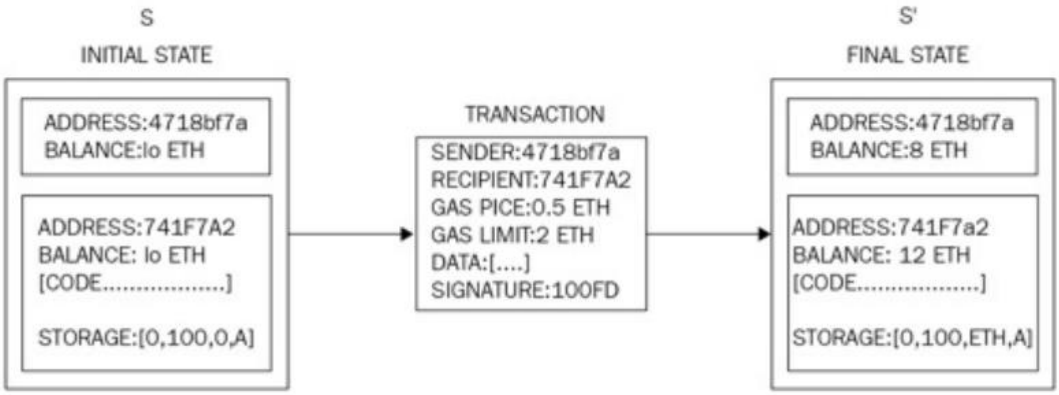

Fig. 3.2 Ethereum Blockchain Transition diagram

The Ethereum state transition function as shown above, where a transaction execution has resulted in a state transition. In this example, a transfer of two Ether from address 4718bf7a to address 741f7a2 is initiated. The initial state represents the state before the transaction execution, and the final state is what the morphed state looks like. Mining plays a central role in state transition, and we will elaborate the mining process in detail in the later sections. The state is stored on the Ethereum network as the world state. This is the global state of the Ethereum blockchain.

3.2.1 Functioning of Etherium

In this section, we will see how Ethereum works from a user's point of view. For this purpose, I will present the most common use case of transferring funds. In our use case, from one user (Bashir) to another (Irshad). We will use two Ethereum clients, one for sending funds and the other for receiving. There are several steps involved in this process which are described here:

1. First either a user requests money by sending the request to the sender, or the sender decides to send money to the receiver. The request can be sent by sending the receivers Ethereum address to the sender. For example, there are two users, Bashir and Irshad. If Irshad requests money from Bashir, then she can send a request to Bashir by using QR code. Once Bashir receives this request he will either scan the QR code or manually type in Irshad's Ethereum address and send Ether to Irshad's address. This request is encoded as a QR code shown in the following screenshot which can be shared via email, text or any other communication methods. Jaxx wallet is used in this example, but you can use any wallet software to achieve the same functionality. There are various wallet software available online, in the iOS App Store and the Android Play Store.
2. Once Bashir receives this request he will either scan this QR code or copy the Ethereum address in the Ethereum wallet software and initiate a transaction. This process is shown

in the following screenshot where the Jaxx Ethereum wallet software on iOS is used to send money to Irshad. The following screenshot shows that the sender has entered both the amount and destination address for sending Ether. Just before sending the Ether the final step is to confirm the transaction which is also shown here:

3. Once the request (transaction) of sending money is constructed in the wallet software, it is then broadcasted to the Ethereum network. The transaction is digitally signed by the sender as proof that he is the owner of the Ether.
4. This transaction is then picked up by nodes called miners on the Ethereum network for verification and inclusion in the block. At this stage, the transaction is still unconfirmed.
5. Once it is verified and included in the block, the PoW process starts.
6. Once a miner finds the answer to the PoW problem, by repeatedly hashing the block with a new nonce, this block is immediately broadcasted to the rest of the nodes which then verifies the block and PoW.
7. If all the checks pass then this block is added to the blockchain, and miners are paid rewards accordingly.
8. Finally, Irshad gets the Ether, and it is shown in her wallet software.

The most common usage of Ethereum network of transferring Ether from a user to another is presented above. This case was just a quick overview of the transaction process in order to introduce the concept. More in-depth technical details will be explained in the upcoming sections of this chapter where we discuss various components of the Ethereum ecosystem.

3.3 The Ethereum Network

The Ethereum network is a peer-to-peer network where nodes participate in order to maintain the blockchain and contribute to the consensus mechanism. Networks can be divided into three types, Mainnet, Testnet and private net, based on requirements and usage.

Mainnet is the current live network of Ethereum. The current version of mainnet is Byzantium (Metropolis) and its chain ID is 1. Chain ID is used to identify the network. A block explorer which shows detailed information about blocks and other relevant metrics is available at https://etherscan.io. This can be used to explore the Ethereum blockchain.

Testnet is also called Ropsten and is the widely used test network for the Ethereum blockchain. This test blockchain is used to test smart contracts and DApps before being deployed to the production live blockchain. Moreover, being a test network, it allows experimentation and research. The main testnet is called Ropsten which contains all features of other smaller and special purpose testnets that were created for specific releases.

As the name suggests, Privatenet is the private network that can be created by generating a new genesis block. This is usually the case in private blockchain distributed ledger networks, where a private group of entities start their blockchain and use it as a permissioned blockchain.

3.3.1 Components of the Ethereum ecosystem

The Ethereum blockchain stack consists of various components. At the core, there is the Ethereum blockchain running on the peer-to-peer Ethereum network. Secondly, there's an Ethereum client (usually Geth) that runs on the nodes and connects to the peer-to-peer Ethereum network from where blockchain is downloaded and stored locally. It provides various functions, such as mining and account management. The local copy of the blockchain is synchronized regularly with the network. Another component is the web3.js library that allows interaction with the geth client via the Remote Procedure Call (RPC) interface. This architecture can be visualized in the following diagram:

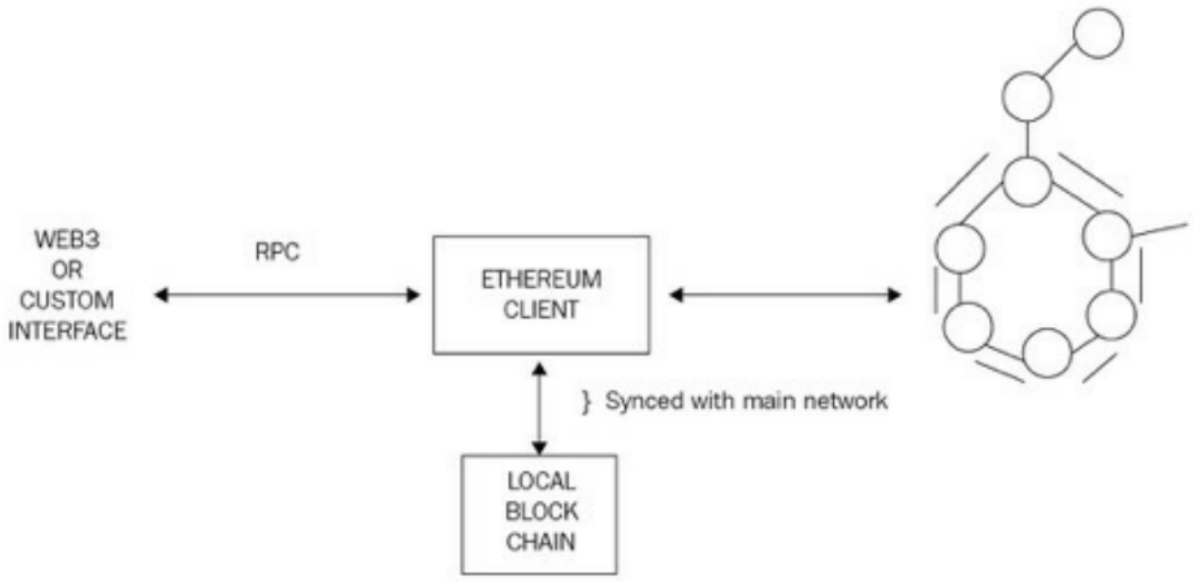

Fig.3.3.1 The Ethereum Components

A list of elements present in the Ethereum blockchain are:

- Keys and addresses
- Accounts
- Transactions and messages
- Ether cryptocurrency/tokens
- The EVM
- Smart contracts

3.3.2 Keys and addresses

Keys and addresses are used in Ethereum blockchain mainly to represent ownership and transfer of Ether. Keys are used in pairs of private and public type. The private key is generated randomly and is kept secret whereas a public key is derived from the private key. Addresses are derived from the public keys which are a 20-bytes code used to identify accounts.

The process of key generation and address derivation is described here:

1. First, a private key is randomly chosen (256 bits positive integer) under the rules defined by elliptic curve secp256k1 specification (in the range [1, secp256k1n − 1]).

2. The public key is then derived from this private key using ECDSA recovery function. We will discuss this later in the next section, Accounts in the context of digital signatures.

3. An address is derived from the public key which is the right most 160 bits of the Keccak hash of the public key.

An example of how keys and addresses look like in Ethereum is shown here:

1. Private key: *b51928c22782e97cca95c490eb958b06fab7a70b9512c38c36974f47b954ffc4*

2.Public key: *3aa5b8eefd12bdc2d26f1ae348e5f383480877bda6f9e1a47f6a4afb35cf998ab847*

f1e3948b11 73622dafc6b4ac198c97b18fe1d79f90c9093ab2ff9ad99260

3. Address: *0x77b4b5699827c5c49f73bd16fd5ce3d828c36f32*

3.3.3 Accounts

Accounts are one of the main building blocks of the Ethereum blockchain. Ethereum, being a transaction driven state machine, the state is created or updated as a result of the interaction between accounts and transaction execution. Operations performed between and on the accounts, represent state transitions. The state transition is achieved using what's called the Ethereum state transition function, which works as follows:

1. Confirm the transaction validity by checking the syntax, signature validity, and nonce.
2. The transaction fee is calculated, and the sending address is resolved using the signature. Furthermore, sender's account balance is checked and subtracted accordingly, and nonce is incremented. An error is returned if the account balance is not enough.
3. Provide enough Ether (gas price) to cover the cost of the transaction. We will cover gas and relevant concept shortly in this chapter. This is charged per byte incrementally

proportional to the size of the transaction. In this step, the actual transfer of value occurs. The flow is from the sender's account to receiver's account. The account is created automatically if the destination account specified in the transaction does not exist yet. Moreover, if the destination account is a contract, then the contract code is executed. This also depends on the amount of gas available. If enough gas is available, then the contract code will be executed fully; otherwise, it will run up to the point where it runs out of gas.

4. In cases of transaction failure due to insufficient account balance or gas, all state changes are rolled back except for fee payment, which is paid to the miners.
5. Finally, the remainder (if any) of the fee is sent back to the sender as change and fee are paid to the miners accordingly. At this point, the function returns the resulting state which is also stored on the blockchain.

There are two kinds of accounts exist in Ethereum, Externally Owned Accounts (EOAs) and Contract Accounts (CAs) . EOAs are similar to accounts that are controlled by a private key in Bitcoin. CAs are the accounts that have code associated with them along with the private key.

Various properties of each type of accounts are described here:

Externally Owned Accounts (EOA0:

- EOAs has ether balance They are capable of sending transactions
- They have no associated code
- They are controlled by private keys
- Accounts contain a key-value store
- They are associated with a human user

Contract Accounts (CA):

- CAs have Ether balance.
- They have associated code that is kept in memory/storage on the blockchain.
- They can get triggered and execute code in response to a transaction or a message from other contracts. It is worth noting that due to the Turing-completeness property of the Ethereum blockchain, the code within contract accounts can be of any level of complexity. The code is executed by Ethereum Virtual Machine (EVM) by each mining node on the Ethereum network. EVM is discussed later in the chapter in the The EVM section.

- CA Can maintain their permanent state and can call other contracts. It is envisaged that in the serenity release, the distinction between externally owned accounts and contract accounts may be eliminated.
- They are not intrinsically associated with any user or actor on the blockchain.
- CAs contain a key-value store.

3.3.4 Transactions and Messages

A transaction in Ethereum is a digitally signed data packet using a private key that contains the instructions that, when completed, either result in a message call or contract creation. Transactions can be divided into two types based on the output they produce:

Message call transactions: This transaction simply produces a message call that is used to pass messages from one contract account to another.

Contract creation transactions: As the name suggests, these transactions result in the creation of a new contract account.

This means that when this transaction is executed successfully, it creates an account with the associated code. Both of these transactions are composed of some standard fields, which are described here.

- ***Nonce***: Nonce is a number that is incremented by one every time a transaction is sent by the sender. It must be equal to the number of transactions sent and is used as a unique identifier for the transaction. A nonce value can only be used once. This is used for replay protection on the network.
- ***Gas price***: The gas price field represents the amount of Wei required to execute the transaction. In other words, this is the amount of Wei you are willing to pay for this transaction. This is charged per unit of gas for all computation costs incurred as a result of the execution of this transaction Wei is the smallest denomination of ether; therefore, it is used to count ether.
- ***Gas limit***: The gas limit field contains the value that represents the maximum amount of gas that can be consumed to execute the transaction. The concept of gas and gas limit will be covered later in the chapter in more detail. For now, it is sufficient to say that this is the amount of fee in ether that a user (for example, the sender of the transaction) is willing to pay for computation.
- ***To***: As the name suggests, the to field is a value that represents the address of the recipient of the transaction. This is a 20-byte value.

- ***Value***: Value represents the total number of Wei to be transferred to the recipient; in the case of a contract account, this represents the balance that the contract will hold.
- ***Signature***: Signature is composed of three fields, namely v, r, and s. These values represent the digital signature (R, S) and some information that can be used to recover the public key (V). Also, the sender of the transaction can also be determined from these values. The signature is based on ECDSA (Elliptic Curve Digital Signature Algorithm) scheme and makes use of the secp256k1 curve.

  V is a single byte value that depicts the size and sign of the elliptic curve point and can be either 27 or 28. V is used in the ECDSA recovery contract as a recovery ID. This value is used to recover (derive) the public key from the private key. In secp256k1, the recovery ID is expected to be either 0 or 1. In Ethereum, this is offset by 27. More details on the ECDSARECOVER function will be provided later in this chapter.

  R is derived from a calculated point on the curve. First, a random number is picked up, which is multiplied by the generator of the curve to calculate a point on the curve. The x coordinate part of this point is R. R is encoded as a 32-byte sequence. R must be greater than 0 and less than the secp256k1n limit (115792089237316195423570985008687907852837564279074904382605163141518161494337).

  S is calculated by multiplying R with the private key and adding it into the hash of the message to be signed and by finally dividing it by the random number chosen to calculate R. S is also a 32-byte sequence. R and S together represent the signature. To sign a transaction, the ECDSASIGN function is used, which takes the message to be signed and the private key as an input and produces V, a single byte value; R, a 32-byte value, and S, another 32-byte value.

  The equation is ECDSASIGN (Message, Private Key) = (V, R, S)
- ***Init***: The Init field is used only in transactions that are intended to create contracts, that is, contract creation transactions. This represents a byte array of unlimited length that specifies the EVM code to be used in the account initialization process. The code contained in this field is executed only once when the account is created for the first time, it (init) gets destroyed immediately after that. Init also returns another code section called body, which persists and runs in response to message calls that the contract account may receive. These message calls may be sent via a transaction or an internal code execution. Data: If the transaction is a message call, then the data field is used

instead of init, which represents the input data of the message call. It is also unlimited in size and is organized as a byte array.

This structure can be visualized in the following diagram, where a transaction is a tuple of the fields mentioned earlier, which is then included in a transaction ***trie*** (a modified Merkle-Patricia tree) composed of the transactions to be included. Finally, the root node of transaction trie is hashed using a Keccak 256-bit algorithm and is included in the block header along with a list of transactions in the block.

Transactions can be found in either transaction pools or blocks. In transaction pools, they wait for verification by a node, and in blocks, they are added after successful verification. When a mining node starts its operation of verifying blocks, it starts with the highest paying transactions in the transaction pool and executes them one by one. When the gas limit is reached, or no more transactions are left to be processed in the transaction pool, the mining starts.

In this process, the block is repeatedly hashed until a valid nonce is found such that, once hashed with the block, it results in a value less than the difficulty target. Once the block is successfully mined, it will be broadcasted immediately to the network, claiming success, and will be verified and accepted by the network.

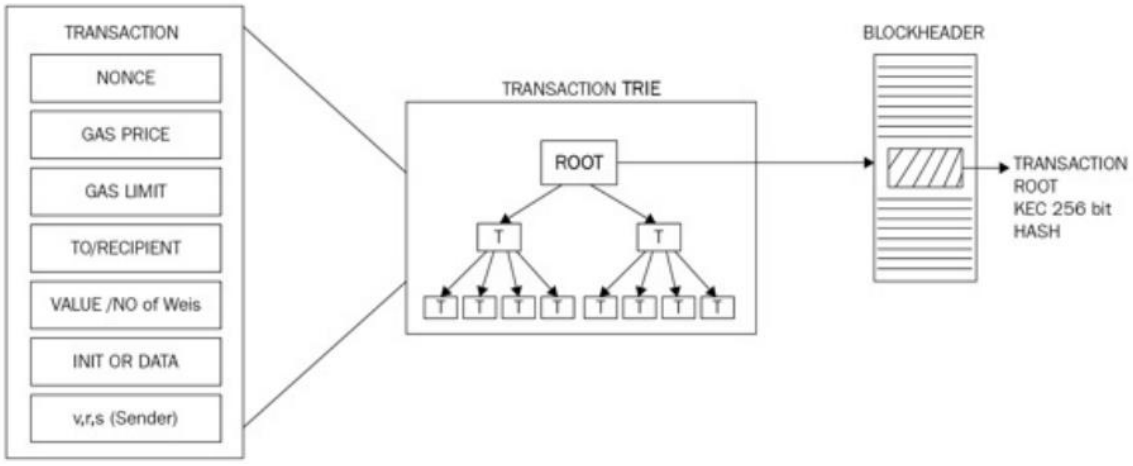

Fig 3.3.4 The relationship between Transaction, Transaction Trie & block header

3.3.5 State storage in the Ethereum blockchain

Ethereum blockchain is a transaction and consensus-driven state machine. The state needs to be stored permanently in the blockchain. For this purpose, world state, transactions, and transaction receipts are stored on the blockchain in blocks. We discuss these components next.

3.3.5.1 The world state

It is a mapping between Ethereum addresses and account states. The addresses are 20 bytes (160 bits) long. This mapping is a data structure that is serialized using Recursive Length Prefix (RLP). RLP is a specially developed encoding scheme that is used in Ethereum to serialize binary data for storage or transmission over the network and also to save the state in a Patricia tree on storage media. The RLP function takes an item as an input, which can be a string or a list of items and produces raw bytes that are suitable for storage and transmission over the network. RLP does not encode data; instead, its primary purpose is to encode structures.

3.3.5.2 The account state

The account state consists of four fields: nonce, balance, storage root and code hash and is described in detail here:

- **Nonce**: This is a value that is incremented every time a transaction is sent from the address. In case of contract accounts, it represents the number of contracts created by the account. Contract accounts are one of the two types of accounts that exist in Ethereum; they will be explained later on in the chapter in more detail.
- **Balance**: This value represents the number of Weis which is the smallest unit of the currency (Ether) in Ethereum held by the address.
- **Storage root**: This field represents the root node of a Merkle Patricia tree that encodes the storage contents of the account.
- **Code hash**: This is an immutable field that contains the hash of the smart contract code that is associated with the account. In the case of normal accounts, this field contains the Keccak 256-bit hash of an empty string. This code is invoked via a message call.

The world state and its relationship with accounts trie, accounts, and block header can be visualized in the following diagram. It shows the account data structure in the middle of the diagram, which contains a storage root hash derived from the root node of the account storage trie shown on the left. The account data structure is then used in the world state trie, which is a mapping between addresses and account states.

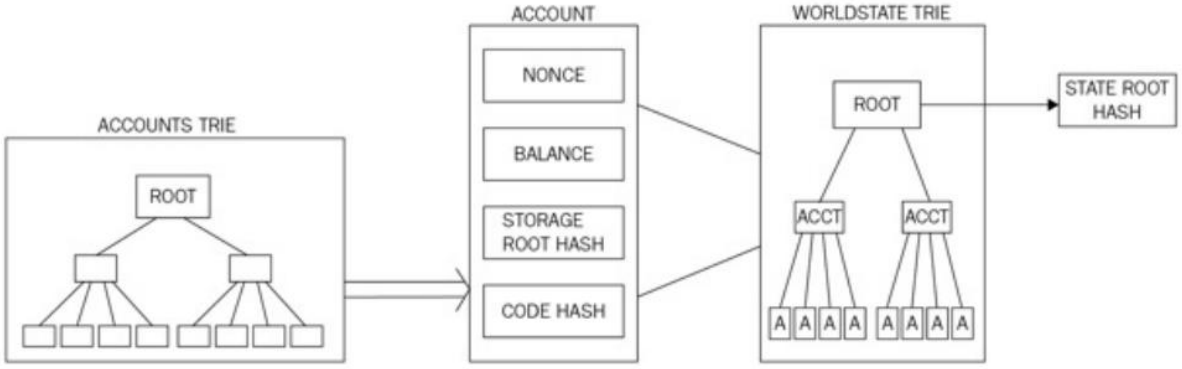

Fig 3.3.5.2 The relationship between account trie, accounts and block header

3.3.6 Ether Cryptocurrency

As an incentive to the miners, Ethereum also rewards its own native currency called Ether, abbreviated as ETH. There are now two Ethereum blockchains: one is called Ethereum Classic, and its currency is represented by ETC, whereas the hard-forked version is ETH, which continues to grow and on which active development is being carried out. ETC, however, has its following with a dedicated community that is further developing ETC, which is the unforked original version of Ethereum.

Ether is minted by miners as a currency reward for the computational effort they spend to secure the network by verifying and with validation transactions and blocks. Ether is used within the Ethereum blockchain to pay for the execution of contracts on the Ethereum Virtual Machine(EVM). Ether is used to purchase gas as crypto fuel, which is required to perform computation on the Ethereum blockchain.

The denomination table is shown as follows:

Unit	Alternative name	Wei value	Number of Weis
Wei	Wei	1 Wei	1
KWei	Babbage	1^3 Wei	1,000
MWei	Lovelace	1^6 Wei	1,000,000
GWei	Shannon	1^9 Wei	1,000,000,000
Micro Ether	Szabo	1^12 Wei	1,000,000,000,000
Milly Ether	Finney	1^15 Wei	1,000,000,000,000,000
Ether	Ether	1^18 Wei	1,000,000,000,000,000,000

Ether Ether 1^18 Wei 1,000,000,000,000,000,000

3.4. Programming Languages

In this section, let us discuss on programming languages that can be used to program smart contracts on Ethereum.

Code for smart contracts in Ethereum is written in a high-level language such as Serpent, LLL, Solidity, or Viper and is converted into the bytecode that EVM understands for it to be executed.

Solidity is one of the high-level languages that has been developed for Ethereum with JavaScript like syntax to write code for smart contracts. Once the code is written, it is compiled into bytecode that's understandable by the EVM using the Solidity compiler called solc. Official Solidity documentation is available at http://solidity.readthedocs.io/en/latest/.

Low-level Lisp-like Language (LLL) is another language that is used to write smart contract code. Serpent is a Python-like high-level language that can be used to write smart contracts for Ethereum.

Vyper is a newer language which has been developed from scratch to achieve a secure, simple, and auditable language. LLL and Serpent are no longer supported by the community and their usage has almost diminished.

Most commonly used language is Solidity, which we will discuss in this section. For example, a simple program in Solidity is shown as follows:

```
pragma solidity ^0.4.0;
contract Test1 {
 uint x=2;
function addition1(uint x) returns (uint y) {
 y=x+2;
}
 }
```

This program is converted into bytecode, as shown in the following subsection.

Runtime bytecode Raw hex codes: 606060405260e060020a6000350463989e17318114601c575b6000565b34600057602….

Opcodes: PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x2 PUSH1 0x0 SSTORE CALLVALUE PUSH1 0x0 JUMPI JUMPDEST PUSH1 0x45 DUP1 PUSH1

3.4.1 Opcodes and their meaning

There are different opcodes that have been introduced in the EVM. Opcodes are divided into multiple categories based on the operation they perform. The list of opcodes with their meaning and usage is presented here. These tables show the mnemonic, hex value of the mnemonic, the number of items that will be removed from the stack when this mnemonic executes (POP), the number of items that are added to the stack (PUSH) when this mnemonic executes, associated gas cost with the mnemonics and purpose of the mnemonic.

3.4.2 Arithmetic operations

All arithmetic in EVM is modulo 2^256. This group of opcodes is used to perform basic arithmetic operations. The value of these operations starts from 0x00 up to 0x0b.

Table 3.4.2 Arithmetic Operations

Mnemonic	Value	POP	PUSH	Gas	Description
STOP	0x00	0	0	0	Halts execution
ADD	0x01	2	1	3	Adds two values
MUL	0x02	2	1	5	Multiplies two values
SUB	0x03	2	1	3	Subtraction operation
DIV	0x04	2	1	5	Integer division operation
SDIV	0x05	2	1	5	Signed integer division
MOD	0x06	2	1	5	Modulo remainder operation
SMOD	0x07	2	1	5	Signed modulo remainder
ADDMOD	0x08	3	1	8	Modulo addition operation
MULMOD	0x09	3	1	8	Module multiplication operation
EXP	0x0a	2	1	10	Exponential operation (repeated multiplication of the base)
SIGNEXTEND	0x0b	2	1	5	Extends the length of two's complement signed integer

Note that STOP is not an arithmetic operation but is categorized in this list of arithmetic operations due to the range of values (0s) it falls in.

3.4.3 Logical Operations

Logical operations include operations that are used to perform comparisons and bitwise logic operations. The value of these operations is in the range of 0x10 to 0x1a.

Table 3.4.3 Logic Operations

Mnemonic	Value	POP	PUSH	Gas	Description
LT	0x10	2	1	3	Less than
GT	0x11	2	1	3	Greater than
SLT	0x12	2	1	3	Signed less than comparison
SGT	0x13	2	1	3	Signed greater than comparison
EQ	0x14	2	1	3	Equal comparison
ISZERO	0x15	1	1	3	Not operator
AND	0x16	2	1	3	Bitwise AND operation
OR	0x17	2	1	3	Bitwise OR operation
XOR	0x18	2	1	3	Bitwise exclusive OR (XOR) operation
NOT	0x19	1	1	3	Bitwise NOT operation
BYTE	0x1a	2	1	3	Retrieve single byte from word

3.4.4 Cryptographic Operations

There is only one operation in this category named SHA3. It is worth noting that this is not the standard SHA-3 standardized by NIST but the original Keccak implementation.

Mnemonic	Value	POP	PUSH	Gas	Description
SHA3	0x20	2	1	30	Used to calculate Keccak 256-bit hash.

Note that 30 is the cost of the operation. Then 6 gas is paid for each word. Therefore, the formula for SHA3 gas cost becomes 30 + 6 * (size of input in words). Apart from these, the set of instructions contains all mnemonics that are necessary to store items on stack and memory are also required for controlling program flow.

The other operations include, push, duplicate, system, exchange and logging operations.

3.5 Blocks and Blockchain

Blocks are the main building blocks of a blockchain. Ethereum blocks consist of various elements, which are described as follows:

- The block header
- The transactions list
- The list of headers of ommers or uncles

The most important and complex part of a block in Ethereum is the block header. The header contains valuable information, which is described in detail here:

Parent hash: This is the Keccak 256-bit hash of the parent (previous) block's header. Ommers hash: This is the Keccak 256-bit hash of the list of ommers (uncles) blocks included in the block.

The beneficiary: Beneficiary field contains the 160-bit address of the recipient that will receive the mining reward once the block is successfully mined.

State root: The state root field contains the Keccak 256-bit hash of the root node of the state trie. It is calculated after all transactions have been processed and finalized.

Transactions root: The transaction root is the Keccak 256-bit hash of the root node of the transaction trie. Transaction trie represents the list of transactions included in the block.

Receipts root: The receipts root is the Keccak 256-bit hash of the root node of the transaction receipt trie. This trie is composed of receipts of all transactions included in the block. Transaction receipts are generated after each transaction is processed and contain useful post-transaction information.

Logs bloom: The logs bloom is a bloom filter that is composed of the logger address and log topics from the log entry of each transaction receipt of the included transaction list in the block.

Difficulty: The difficulty level of the current block.

Number: The total number of all previous blocks; the genesis block is block zero. Gas limit: The field contains the value that represents the limit set on the gas consumption per block. Gas used: The field contains the total gas consumed by the transactions included in the block.

Timestamp: Timestamp is the epoch Unix time of the time of block initialization.

Extra data: Extra data field can be used to store arbitrary data related to the block. Only up to 32 bytes are allowed in this field.

Mixhash: Mixhash field contains a 256-bit hash that once combined with the nonce is used to prove that adequate computational effort (PoW) has been spent in order to create this block.

Nonce: Nonce is a 64-bit hash (a number) that is used to prove, in combination with the mixhash field, that adequate computational effort (PoW) has been spent in order to create this block.

The following figure shows the detailed structure of the block and block header:

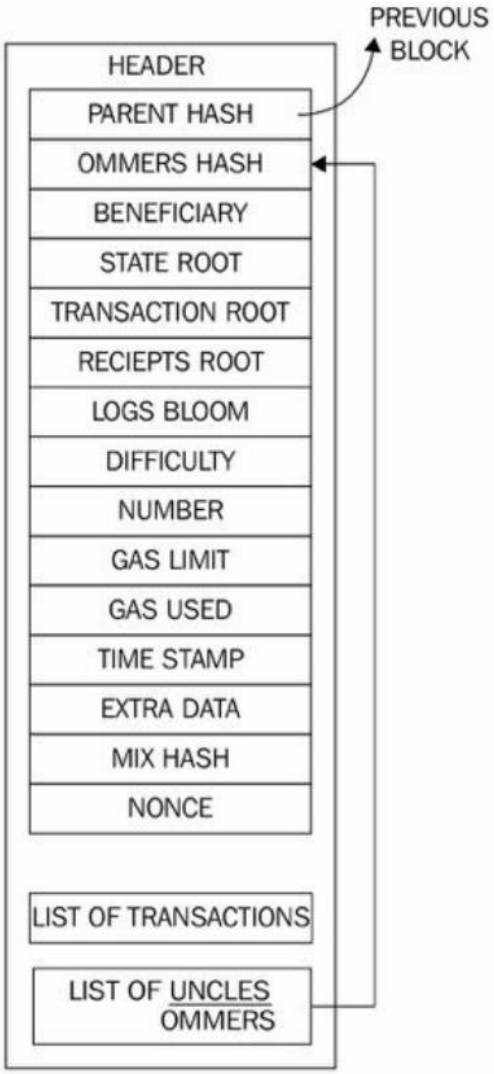

Fig.3.5 Block Structure with block header

3.5.1 Ethereum block Validation

An Ethereum block is considered valid if it passes the following checks:

- Consistent with uncles and transactions, this means that all ommers (uncles) satisfy the property that they are indeed uncles and also if the PoW for uncles is valid.
- If the previous block (parent) exists and is valid.
- If the timestamp of the block is valid. This means that the current block's timestamp must be higher than the parent block's timestamp. Also, it should be less than 15 minutes into the future. All block times are calculated in epoch time (Unix time).
- If any of these checks fails, the block will be rejected.

3.5.2 Block finalization

Block finalization is a process that is run by miners to validate the contents of the block and apply rewards. It results in four steps being executed. These steps are described here in detail:

1. Ommers validation: Validate ommers (stale blocks also called uncles). In the case of mining, determine ommers. The validation process of the headers of stale blocks checks whether the header is valid and the relationship of the Uncle with the current block satisfies the maximum depth of six blocks. A block can contain a maximum of two uncles.

2. Transaction validation: Validate transactions. In the case of mining, determine transactions. The process involves checking whether the total gas used in the block is equal to the final gas consumption after the final transaction i.e. cumulative gas used by the transactions included in the block.

3. Reward application: Apply rewards, which means updating the beneficiary's account with a reward
balance. In Ethereum, a reward is also given to miners for stale blocks, which is 1/32 of the block reward. Uncles that are included in the blocks also receive 7/8 of the total block reward. The current block reward is 3 Ether. It was reduced from 5 with Byzantium release of Ethereum. A block can have a maximum of two uncles.

4. State and nonce validation: Verify the state and block nonce. In the case of mining, compute a valid state and block nonce.

3.5.3 Block difficulty

Block difficulty is increased if the time between two blocks decreases, whereas it increases if the block time between two blocks decreases. This is required to maintain a roughly consistent block generation time. The difficulty adjustment algorithm in Ethereum's Homestead release is shown as follows:

```
block_dif = parent_dif + parent_dif // 2048 *
max(1 - (block_timestamp - parent_timestamp) // 10, -99) +
int(2**((block.number // 100000) - 2))
```

The preceding algorithm means that, if the time difference between the generation of the parent block and the current block is less than 10 seconds, the difficulty goes up. If the time

difference is between 10 to 19 seconds, the difficulty level remains the same. Finally, if the time difference is 20 seconds or more, the difficulty level decreases. This decrease is proportional to the time difference.

3.5.4 Gas

Gas is required to be paid for every operation performed on the Ethereum blockchain. This is a mechanism that ensures that infinite loops cannot cause the whole blockchain to stall due to the Turing-complete nature of the EVM. A transaction fee is charged as some amount of Ether and is taken from the account balance of the transaction originator.

A fee is paid for transactions to be included by miners for mining. If this fee is too low, the transaction may never be picked up; the more the fee, the higher are the chances that the transactions will be picked up by the miners for inclusion in the block. Conversely, if the transaction that has an appropriate fee paid is included in the block by miners but has too many complex operations to perform, it can result in an out-of-gas exception if the gas cost is not enough. In this case, the transaction will fail but will still be made part of the block, and the transaction originator will not get any refund.

Transaction cost can be estimated using the formula: *Total cost = gasUsed * gasPrice*

Here, gasUsed is the total gas that is supposed to be used by the transaction during the execution and gasPrice is specified by the transaction originator as an incentive to the miners to include the transaction in the next block. This is specified in Ether. Each EVM opcode has a fee assigned to it. It is an estimate because the gas used can be more or less than the value specified by the transaction originator originally. For example, if computation takes too long or the behavior of the smart contract changes in response to some other factors, then the transaction execution may perform more or fewer operations than intended initially and can result in consuming more or fewer gas. If the execution runs out of gas, everything is immediately rolled back; otherwise, if the execution is successful and there is some remaining gas, then it is returned to the transaction originator.

3.5.5 Fee Schedule

Each operation costs some gas; a high-level fee schedule of a few operations is shown as an example here:

Operation Name	Gas Cost
step	1
stop	0
suicide	0
sha3	30
sload	20
txdata	5
transaction	500
contract creation	53000

Based on the preceding fee schedule and the formula discussed earlier, an example calculation of the SHA-3 operation can be calculated as follows:

- SHA-3 costs 30 gas.
- Assume that current gas price is 25 GWei, convert it into ether, which is 0.000000025 Ether. After multiplying both: 0.000000025 * 30, we get 0.00000075 Ether.
- In total, 0.00000075 Ether is the total gas that will be charged.

Gas is charged in three scenarios as a prerequisite to the execution of an operation:

- The computation of an operation
- For contract creation or message call
- Increase in the usage of memory

3.5.6 Supporting Protocols

Various supporting protocols are available to assist the complete decentralized ecosystem. This includes Whisper and Swarm protocols. In addition to the contracts layer, which is the core blockchain layer, there are additional layers that need to be decentralized in order to achieve a complete decentralized ecosystem. This includes decentralized storage and decentralized messaging. Whisper, which is being developed for Ethereum, is a decentralized messaging protocol, whereas Swarm is a decentralized storage protocol. Both of these technologies provide the basis for a fully decentralized web. Both the technologies are described in the following sections.

3.5.6.1 Whisper

Whisper provides decentralized peer-to-peer messaging capabilities to the Ethereum network. In essence, Whisper is a communication protocol that DApps use to communicate with each other. The data and routing of messages are encrypted within Whisper communications. Whisper makes use of DEVp2p wire protocol for exchanging messages between nodes on the network. Moreover, it is designed to be used for smaller data transfers and in scenarios where real-time communication is not required. Whisper is also designed to provide a communication layer that cannot be traced and provides dark communication between parties. Blockchain can be used for communication, but that is expensive, and a consensus is not really required for messages exchanged between nodes. Therefore, Whisper can be used as a protocol that allows censor resistant communication. Whisper messages are ephemeral and have an associated time to live TTL. Whisper is already available with Geth and can be enabled using the --shh option while running the Geth Ethereum client.

3.5.6.2 Swarm

Swarm has been developed as a distributed file storage platform. It is a decentralized, distributed, and peer-topeer storage network. Files in this network are addressed by the hash of their content. This is in contrast to the traditional centralized services, where storage is available at a central location only. This is developed as a native base layer service for the Ethereum Web 3 stack. Swarm is integrated with DEVp2p, which is the multiprotocol network layer of Ethereum. Swarm is envisaged to provide a Distributed Denial of Service (DDOS)-resistant and fault-tolerant distributed storage layer for Ethereum Web 3.0. Similar to shh in Whisper, Swarm has a protocol called bzz which is used by each Swarm node to perform various Swarm protocol operations.

The following diagram gives a high-level overview of how Swarm and Whisper fit together and work with the Ethereum blockchain:

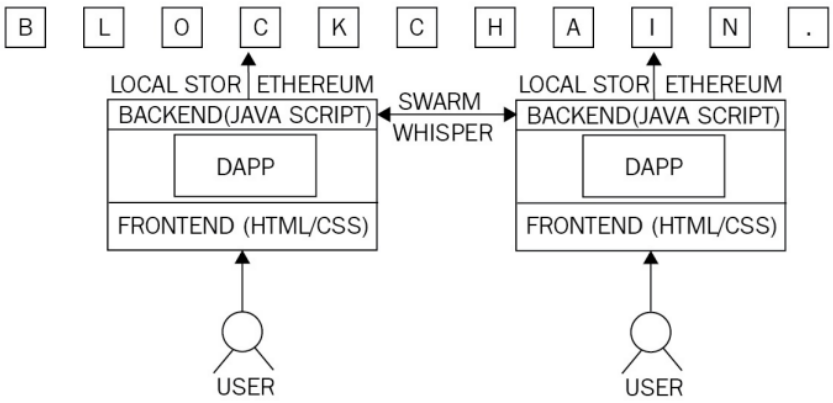

Fig.3.5.6.2 Blockchain, Swarm & Whisper

With the development of Whisper and Swarm a complete decentralized ecosystem emerges, where Ethereum is considered a decentralized computer (state), Whisper as decentralized communication, and Swarm as decentralized storage. If you recall, in Chapter 2, Decentralization we mentioned that decentralization of the whole ecosystem is highly desirable as opposed to only decentralization of the core computation element. Development of Whisper and Swarm is a step towards decentralization of the complete blockchain ecosystem.

3.6 Programming Language – Solidity Language

Code for smart contracts in Ethereum is written in a high-level language such as Serpent, LLL, Solidity, or Viper and is converted into the bytecode that EVM understands for it to be executed.

Smart contracts can be programmed in a variety of languages for Ethereum blockchain. There are five languages that can be used in order to write contracts:

- **Mutan:** This is a Go-style language, which was deprecated in early 2015 and is no longer used.
- **LLL:** This is a Low-level Lisp-like Language, hence the name LLL. This is also not used anymore.
- **Serpent:** This is a simple and clean Python-like language. It is not used for contract development anymore and not supported by the community anymore.
- **Solidity**: This language has now become almost a standard for contract writing for Ethereum.
- **Vyper:** This language is a Python-like experimental language that is being developed to bring security, simplicity, and auditability to smart contract development.

Solidity is one of the high-level languages that has been developed for Ethereum with JavaScript like syntax to write code for smart contracts. Once the code is written, it is compiled into bytecode that's understandable by the EVM using the Solidity compiler called solc.

For example, a simple program in Solidity is shown as follows:

```
pragma solidity ^0.4.0;
contract Test1
{
uint x=2;
function addition1(uint x) returns (uint y)
{
```

```
y=x+2;
}
}
```

This program is converted into bytecode using compiler. Let us learn how to compile Solidity code

We will discuss various constructs of Solidity language in detail, which is currently the most popular development language for smart contract development on Ethereum.
There are a number of tools available for Ethereum development. The following diagram shows the taxonomy of various development tools, clients, IDEs, and development frameworks for Ethereum:, as shown in the following subsection.

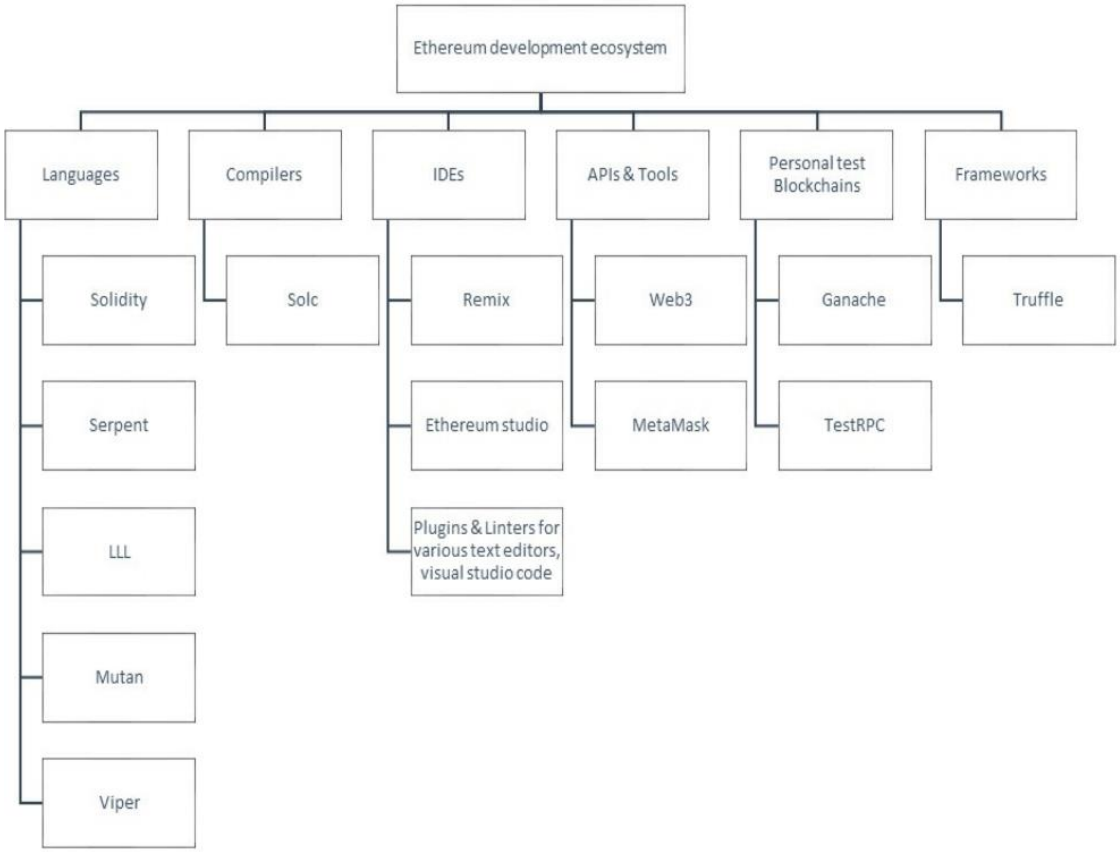

Fig. 3.6 Taxonomy of Ethereum development ecosystem components

The preceding taxonomy does not include all frameworks and tools that are out there for development on Ethereum. It shows most commonly used tools and frameworks and also the ones that we will use in our examples.

3.6.1 Solidity Language

Solidity is a domain-specific language of choice for programming contracts in Ethereum. There are, however, other languages that can be used, such as Serpent, Mutan, and LLL but Solidity is the most popular at the time of writing this. Its syntax is closer to both JavaScript and C. Solidity has evolved into a mature language over the last few years and is quite easy to use, but it still has a long way to go before it can become advanced, standardized, and feature-rich like other well-established languages such as Java, C or C Sharp. Nevertheless, this is the most widely used language available for programming contracts currently. It is a statically typed language, which means that variable type checking in Solidity is carried out at compile time. Each variable, either state or local, must be specified with a type at compile time. This is beneficial in the sense that any validation and checking is completed at compile time and certain types of bugs, such as interpretation of data types, can be caught earlier in the development cycle instead of at runtime, which could be costly, especially in the case of the blockchain / smart contracts paradigm. Other features of the language include inheritance, libraries, and the ability to define composite data types. Solidity is also called a contract-oriented language. In Solidity, contracts are equivalent to the concept of classes in other object-oriented programming languages.

Solidity compiler, ***solc,*** converts a high-level solidity language into Ethereum Virtual Machine (EVM) bytecode so that it can be executed on the blockchain by EVM.

Solidity has two categories of data types: value types and reference types.

The value types are variables that are always passed by a valued. This means that value of types hold their value or data directly, allowing a variable's value held in memory to be directly accessible by accessing the variable.

The Reference types store the address of the memory location where the value is stored. This is in contrast with value types, which store the actual value of a variable directly with the variable itself. When using reference types, it is essential to explicityly specify the storage area where the type is stored, for example, *memory*, *storage* or *calldata*.

3.6.2 Value Types

Value types mainly include Boolean, Integer, Addresses, and literals.

Boolean: This data type has two possible values, true or false, for example

```
bool v = true; or
bool v = false;
```

Integers: This data type represents integers. The following table shows various keywords used to declare integer data types:

Keyword	Types	Details
Int	Signed integer	int8 to int256, which means that keywords are available from int8 up to int256 in increments of 8, for example, int8, int16, int24.
Uint	Unsigned integer	uint8, uint16, ... to uint256, unsigned integer from 8 bits to 256 bits. The usage is dependent on the requirements that how many bits are required to be stored in the variable.

For example, in this code, note that uint is an alias for uint256:

uint256 x;

uint y;

uint256 z;

These types can also be declared with the constant keyword, which means that no storage slot will be reserved by the compiler for these variables. In this case, each occurrence will be replaced with the actual value:

uint constant z=10+10;

State variables are declared outside the body of a function, and they remain available throughout the contract depending on the accessibility assigned to them and as long as the contract persists.

Address: This data type holds a 160-bit long (20 byte) value. This type has several members that can be used to interact with and query the contracts. These members are described here:

Balance: The balance member returns the balance of the address in Wei.

Send: This member is used to send an amount of ether to an address (Ethereum's 160-bit address) and

returns true or false depending on the result of the transaction, for example, the following:

address to = 0x6414cc08d148dce9ebf5a2d0b7c220ed2d3203da; address from = this;

if (to.balance < 10 && from.balance > 50) to.send(20);

Call functions: The call, callcode, and delegatecall calls are provided in order to interact with functions that do not have ABI. These functions should be used with caution as they are not safe to use due to the impact on type safety and security of the contracts.

Array value types (fixed size and dynamically sized byte arrays): Solidity has fixed size and dynamically sized byte arrays. Fixed size keywords range from bytes1 to bytes32, whereas dynamically sized

keywords include bytes and string. The bytes keyword is used for raw byte data and string is used for strings encoded in UTF-8. As these arrays are returned by the value, calling them will incur gas cost. length is a member of array value types and returns the length of the byte array.

An example of a static (fixed size) array is as follows:

```
bytes32[10] bankAccounts;
```

An example of a dynamically sized array is as follows:

```
bytes32[] trades;
```

Get length of trades by using the following code:

```
trades.length;
```

Literals: These are used to represent a fixed value. There are different types of literals that are described as follows:

- *Integer Literals*: These are a sequence of decimal numbers in the range of 0-9.
    - Example, uint8 x = 2;
- *String Literals*: This type specifies a set of characters written with double or single quotes.
    - Example, 'packt', "packt"
- *Hexadecimal Literals*: These are prefixed with the keyword hex and specified within double or single quotation marks.
    - Example, hex 'BDAAC'
- *Enums* : This allows the creation of user-defined types.
    - Example :

      ```
      enum Order {Filled, Placed, Expired };
      Order private ord;
      ord=Order.Filled;
      ```

      Explicit conversion to and from all integer types is allowed with enums.

Reference Types: are passed by reference and are discussed below. These are also known as Complex types. Reference types include arrays, structs and mappings.

- **Arrays** represent a contiguous set of elements of the same size and type laid out at a memory location. The concept is the same as any other programming language. Arrays have two members named length and push: uint[] OrderIds;

- **Structures** are the constructs that can be used to group a set of dissimilar data types under a logical group. These can be used to define new types, as shown in the following example:

```
pragma solidity ^0.4.0;
contract TestStruct {
struct Trade
{
uint tradeid;
uint quantity;
uint price;
string trader;
}
//This struct can be initialized and used as below
Trade tStruct = Trade({tradeid:123, quantity:1, price:1, trader:"equinox"});
}
```

Data location specifies where a particular complex data type will be stored. Depending on the default or annotation specified, the location can be storage or memory. This is applicable to arrays and structs and can be specified using the storage or memory keywords. As copying between memory and storage can be quite expensive, specifying a location can be helpful to control the gas expenditure at times. Calldata is another memory location that is used to store function arguments. Parameters of external functions use calldata memory. By default, parameters of functions are stored in memory, whereas all other local variables make use of storage. State variables, on the other hand, are required to use storage.

- **Mappings** are used for a key to value mapping. This is a way to associate a value with a key. All values in thismap are already initialized with all zeroes, for example, the following:

mapping (address => uint) offers; This example shows that offers is declared as a mapping.

Another example makes this clearer:

```
mapping (string => uint) bids;
bids["packt"] = 10;
```

This is basically a dictionary or a hash table where string values are mapped to integer values. The mapping named bids has string packt mapped to value 10.

3.7 Control Structures

Control structures available in solidity language are if...else, do, while, for, break, continue, and return. They work exactly the same as other languages such as C-language or JavaScript.

Some examples are shown here:

```
if: If x is equal to 0 then assign value 0 to y else assign 1 to z:
if (x == 0)
y = 0;
else
z = 1;
do: Increment x while z is greater than 1:
do{
x++;
} (while z>1);
while: Increment z while x is greater than 0:
while(x > 0){
z++;
}
for, break, and continue: Perform some work until x is less than or equal to 10.
This for loop will run 10
times, if z is 5 then break the for loop:
for(uint8 x=0; x<=10; x++)
{
//perform some work
z++
if(z == 5) break;
}
```

It will continue the work similarly, but when the condition is met, the loop will start again.

return: Return is used to stop the execution of a function and returns an optional value.

For example: return 0;

It will stop the execution and return value of 0.

3.7.1 Events:

Events in Solidity can be used to log certain events in EVM logs. These are quite useful when external interfaces are required to be notified of any change or event in the contract.

These logs are stored on the blockchain in transaction logs. Logs cannot be accessed from the contracts but are used as a mechanism to notify change of state or the occurrence of an event (meeting a condition) in the contract.

In a simple example here, the valueEvent event will return true if the x parameter passed to function Matcher is equal to or greater than 10:

```
pragma solidity ^0.4.0;
contract valueChecker
{
uint8 price=10;
event valueEvent(bool returnValue);
function Matcher(uint8 x) public returns (bool)
{
if (x>=price)
{
valueEvent(true);
return true;
}
}
}
```

3.7.2 Inheritance:

Inheritance is supported in Solidity. The is keyword is used to derive a contract from another contract. In the following example, valueChecker2 is derived from the valueChecker contract. The derived contract has access to all non-private members of the parent contract:

```
pragma solidity ^0.4.0;
contract valueChecker
{
uint8 price = 20;
event valueEvent(bool returnValue);
function Matcher(uint8 x) public returns (bool)
{
if (x>=price)
{
valueEvent(true);
return true;
```

```
}
}
}
contract valueChecker2 is valueChecker
{
function Matcher2() public view returns (uint)
{
return price+10;}}
```

In the preceding example, if the uint8 price = 20 is changed to uint8 private price = 20, then it will not be accessible by the valueChecker2 contract. This is because now the member is declared as private, it is not allowed to be accessed by any other contract.

3.8 Functions

Functions in Solidity are modules of code that are associated with a contract. Functions are declared with a name, optional parameters, access modifier, optional constant keyword, and optional return type. This is shown in the following example:

function orderMatcher (uint x)

private constant returns(bool return value)

In the preceding example, function is the keyword used to declare the function. orderMatcher is the function name, uint x is an optional parameter, private is the access modifier or specifier that controls access to the function

from external contracts, constant is an optional keyword used to specify that this function does not change anything in the contract but is used only to retrieve values from the contract and returns (bool return value) is the

optional return type of the function.

How to define a function: The syntax of defining a function is shown as follows:

function <name of the function>(<parameters>) <visibility specifier> returns

(<return data type> <name of the variable>)

{

<function body>

}

Function signature: Functions in Solidity are identified by its signature, which is the first four bytes of the Keccak-256 hash of its full signature string. This is also visible in Remix IDE, as

shown in the following screenshot. f9d55e21 is the first four bytes of 32-byte Keccak-256 hash of the function named Matcher.

Function hash as shown in Remix IDE. In this example function, Matcher has the signature hash of d99c89cb. This information is useful in order to

build interfaces.

Input parameters of a function: Input parameters of a function are declared in the form of <data type>

<parameter name>. This example clarifies the concept where uint x and uint y are input parameters of the checkValues function:

```
contract myContract
{
function checkValues(uint x, uint y)
{
}
}
```

Output parameters of a function: Output parameters of a function are declared in the form of <data type>

<parameter name>. This example shows a simple function returning a uint value:

```
contract myContract
{
function getValue() returns (uint z)
{
z=x+y;
}
}
```

A function can return multiple values. In the preceding example function, getValue only returns one value, but a function can return up to 14 values of different data types. The names of the unused return parameters can be omitted optionally.

Internal function calls: Functions within the context of the current contract can be called internally in a direct manner. These calls are made to call the functions that exist within the same contract. These calls result in simple JUMP calls at the EVM bytecode level.

External function calls: External function calls are made via message calls from a contract to another contract. In this case, all function parameters are copied to the memory. If a call to an

internal function is made using the this keyword, it is also considered an external call. The this variable is a pointer that refers to the current contract. It is explicitly convertible to an address and all members for a contract are inherited from the address.

3.8.1 Fallback functions:

This is an unnamed function in a contract with no arguments and return data. This function executes every time Ether is received. It is required to be implemented within a contract if the contract is intended to receive Ether; otherwise, an exception will be thrown and Ether will be returned. This function also executes if no other function signatures match in the contract. If the contract is expected to receive Ether, then the fallback function should be declared with the payable modifier. The payable is required; otherwise, this function will not be able to receive any Ether. This function can be called using the address.call() method as, for example, in the following:

```
function ()
{
throw;
}
```

In this case, if the fallback function is called according to the conditions described earlier; it will call throw, which will roll back the state to what it was before making the call. It can also be some other construct than throw; for example, it can log an event that can be used as an alert to feed back the outcome of the call to the calling application.

- **Modifier functions:** These functions are used to change the behavior of a function and can be called before other functions. Usually, they are used to check some conditions or verification before executing the function. _ (underscore) is used in the modifier functions that will be replaced with the actual body of the function when the modifier is called. Basically, it symbolizes the function that needs to be guarded. This concept is similar to guard functions in other languages.
- **Constructor function**: This is an optional function that has the same name as the contract and is executed once a contract is created. Constructor functions cannot be called later on by users, and there is only one constructor allowed in a contract. This implies that no overloading functionality is available.
- **Function visibility specifiers** (access modifiers): Functions can be defined with four access specifiers as follows: External: These functions are accessible from other contracts and transactions. They cannot be called internally unless the this keyword is used.

- **Public:** By default, functions are public. They can be called either internally or using messages.
- **Internal**: Internal functions are visible to other derived contracts from the parent contract.
- **Private:** Private functions are only visible to the same contract they are declared in.
- **Function Modifiers:**
    - pure: This modifier prohibits access or modification to state
    - view: This modifier disables any modification to state
    - payable: This modifier allows payment of ether with a call
    - constant: This modifier disallows access or modification to state
- **Other important keywords/functions throw**: throw is used to stop execution. As a result, all state changes are reverted. In this case, no gas is returned to the transaction originator because all the remaining gas is consumed.

3.8.2 Layout of a Solidity source code file

- **Version Pragma :** In order to address compatibility issues that may arise from future versions of the solc version, pragma can be used to specify the version of the compatible compiler as, for example, in the following: pragma solidity ^0.5.0 This will ensure that the source file does not compile with versions smaller than 0.5.0 and versions starting from 0.6.0.
- **Import:** in Solidity allows the importing of symbols from the existing Solidity files into the current global scope. This is similar to import statements available in JavaScript, as for example, in the following: import "module-name";
- Comments can be added in the Solidity source code file in a manner similar to C-language. Multiple line comments are enclosed in /* and */, whereas single line comments start with //.
    - An example Solidity program is as follows, showing the use of pragma, import, and comments:

```
1  pragma solidity ^0.4.0; //specify the compiler version
2  /*
3  This is a simple value checker contract that checks the value
4  provided and returns boolean value based on the condition
5  expression evaluation.
6  */
7  import "dev.oraclize.it/api.sol";
8  contract valuechecker {
9      uint price=10;
10     //This is price variable declare and initialized with value 10
11     event valueEvent(bool returnValue);
12     function Matcher (uint8 x) returns (bool)
13     {
14         if ( x >= price)
15         {
16             valueEvent(true);
17             return true;
18         }
19     }
20 }
```

Fig.3.8.2 Sample Solidity program as shown in Remix IDE

3.9 Summary

In this Unit, we have explored Ethereum test networks and how-to setup private Ethereum networks. We have also seen that how the command-line tool can be used to perform various functions and how we can interact with the Ethereum blockchain and tools, programming languages, such as solidity, and frameworks are available for development of smart contracts on Ethereum.

Short Questions :

1. List the elements of ethereum blockchain
2. What are value types ?
3. How the level of difficulty is computed ?
4. Assume that current gas price is 25 GWei, SHA-3 costs 30 gas, find the total cost of gas in Ethereum.
5. What is the cryptographic operation used in Ethereum

Long Questions :

1. Explain Block & Block header
2. Describe the relationship between Transaction, Transaction Trie & block header with diagram
3. What are the components of ethereum ecosystem? Explain
4. Explain the Keys & addresses in ehereum
5. Explain how the inheritance is supported in Solidity?

UNIT 4 - WEB3 AND HYPERLEDGER

Overview

In this Unit, you will learn Web3 API and Hyperledger. Let us explore web3 with examples on how smart contracts are written, tested, and deployed to the Ethereum blockchain. We will also learn that how HTML and JavaScript frontends can be developed to interact with smart contract. Hyperledger is a project, used 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.

Learning Objective

To study and explore web3 API and Hyperledger to write contracts and to deploy in Ehereum blockchain.

4.1 Introduction to Web3

Web3 is a JavaScript library that can be used to communicate with an Ethereum node via RPC communication.Web3 works by exposing methods that have been enabled over RPC. This allows the development of user interfaces that make use of the Web3 library in order to interact with the contracts deployed over the blockchain. This is a powerful library and can be explored further by attaching a geth instance. Later in this section, you will be introduced to the concepts and techniques of making use of Web3 via JavaScript/HTML frontends. The ***geth*** instance can be attached using the following command:

$ geth attach ipc: ~/etherprivate/geth.ipc

Once the ***geth*** JavaScript console is running, Web3 can be queried, for example:

- web3.version {
  api : ”0.15.3”,
  ethereum : “0x3f”,
  network = “786”,
  node : “Geth/v1.5.2-stable-c865/linunx,
  whisper : unidentified,
  getEthereum : function {callback},
  getNetwork : function {callback},
  getNorde : function {callback},
  getWhisper : function {callback}
  }

4.2 Contract Deployment

A simple contract can be deployed using geth and interacted with using Web3 via the command-line interface that geth provides (console or attach). The following are the steps to achieve that. As an example, the following source code will be used:

```
pragma solidity ^0.4.0;
contract valueChecker {
uint price=10;
event valueEvent(bool returnValue);
function Matcher (uint8 x) public returns (bool) {
        if (x>=price) {
          valueEvent(true);
            return true; }
}   }
```

1. Run geth client using the following command:

$ ./geth --datadir ~/etherprivate/ --networkid 786 --rpc -rpcapi 'web3,eth,debug,personal' --rpccorsdomain '*'

2. You will also want to open another terminal and run the following command. The geth console should

already be running by using the following command; if not run it using the following command:

$ ./geth attach ipc:/Users/drequinox/etherprivate/geth.ipc

3. This step will need the Web3 deployment of the contract. This can be obtained from the Remix browser. First, paste the following source code in the Remix IDE, to get the required Web3 deployment object for deployment:

```
pragma solidity ^0.4.0;
contract valueChecker {
uint price=10;
event valueEvent(bool returnValue);
function Matcher (uint8 x) public returns (bool) {
        if (x>=price) {
          valueEvent(true);
         return true; }
  }
}
```

Once the code is pasted in the Remix IDE, it will look like this:

4. Now copy the Web3 deployment script by clicking on the Details button and copy the function into the clipboard. This can be achieved by clicking on the icon next to WEB3DEPLOY caption in the IDE.

5. The Web3 deployment script is shown in the following code snippet:

```
Var valuecheckerContract = web3.eth.contract ([{"constant":false,"inputs"
:[{"name":"x","type":"uint8"}],"name":"Matcher","outputs":[{"var valuechecker =
valuecheckerContract.new(
{
from: web3.eth.accounts[0],
data:
'0x6060604052600a600055341561001457600080fd5b61010380610023600039600f
300606060405260043610603f576000357c0100000000000000000000000gas: '4700000'
}, function (e, contract){
console.log(e, contract);
if (typeof contract.address !== 'undefined') {
console.log('Contract mined! address: ' + contract.address + ' transactionHash: ' +
contract.transactionHash);
}
})
```

6. Ensure that the accounts are unlocked and coinbase is set correctly.

First list the accounts, by using the following command, which outputs the account 0, as shown:

```
> personal.listAccounts[0]
"0xcf61d213faa9acadbf0d110e1397caf20445c58f"
```

Now unlock the account using the following command. It will need the passphrase (password) that you

used originally when creating this account. Enter the password to unlock the account:

7. Now open the geth console that has been opened previously, and deploy the contract: However, before deploying the contract, make sure that mining is running on the geth node. The command below can be used to start mining under the geth console. > Miner.start(1)

8. Now paste this Web3 deployment script in the ***geth*** console

9. After the contract is deployed successfully you can query various attributes related to this contract which we will also use later in this example. For example, contract address, ABI definition and so on, Remember all these commands are issued via geth console, that we have

already opened and used for contract deployment. After the contract is deployed successfully you can query various attributes related to this contract which we will also use later in this example. Remember all these commands are issued via geth console, that we have already opened and used for contract deployment.

4.3 POST Requests

The world wide Web and HTTP are based on a number of request methods or 'verbs', including POST and GET. POST is a request method supported by HTTP used by the World Wide Web. The POST request method requests that a web server accepts the data enclosed in the body of the request message, most likely for storing it.It is mainly used when uploading a file or when submitting a completed web form.

Curl is an internet transfer engine, an open source software, being used in cars, television sets, routers, printers, audio equipment, mobile phones, tablets, medical devices, settop boxes, computer games, media players and is the Internet transfer engine for many software applications.

4.3.1 HTML & Web3 javascript

It is desirable to interact with the contracts in a user-friendly manner via a web page. It is possible to interact with the contracts using the web3.js library from HTML/JS/CSS-based web pages. The HTML content can be served using any HTTP web server, whereas web3.js can connect via local RPC to the running Ethereum client (geth) and provide an interface to the contracts on the blockchain. This architecture can be visualized in the following diagram:

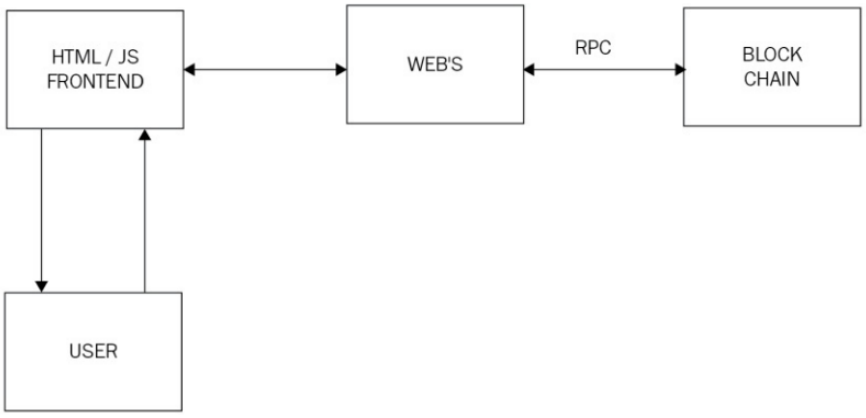

Fig.4.3.1 Web3.js and blockchain interaction architecture

4.3.2 Example of web3.js to allow interaction with the contracts through a web page

In the following section, an example will be presented that will make use of web3.js to allow interaction with the contracts via a web page served over a simple HTTP web server. This can be achieved by following these steps:

1. First, create a directory named */simplecontract/app*, the home directory. This is the directory under your user on Linux. This can be any directory, but in this example home directory is used.
2. Then, create a file named *app.js*, as shown here:

```
var Web3 = require('web3');
if (typeof web3 !== 'undefined') {
web3 = new Web3(web3.currentProvider);
} else {
web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
}
web3.eth.defaultAccount = web3.eth.accounts[0];
var SimpleContract = web3.eth.contract([ {
"constant": false,
"inputs": [ {
"name": "x",
"type": "uint8"
} ],
"name": "Matcher",
"outputs": [ {
"name": "",
"type": "bool"
} ],
"payable": false,
"stateMutability": "nonpayable",
"type": "function"
},
{ "anonymous": false,
"inputs": [ {
"indexed": false,
"name": "returnValue",
```

```
"type": "bool"
} ],
"name": "valueEvent",
"type": "event"
} ]);
var simplecontract =
SimpleContract.at("0xd9d02a4974cbeb10406639ec9378a782bf7f4dd2");
console.log(simplecontract);
function callMatchertrue() {
  var txn = simplecontract.Matcher.call(12);
{ };
console.log("return value: " + txn);
}
function callMatcherfalse() {
var txn = simplecontract.Matcher.call(1);{
};
console.log("return value: " + txn);
}
```

This file contains various elements. The most important is **ABI** (**Application Binary Interface**), which can be queried using geth, generated using solidity compiler or copied directly from remix IDE contract details.

3. Create a file named index.html, as shown here:

```
<html>
<head>
<title>SimpleContract Interactor</title>
<script src="./web3.js"></script>
<script src="./app.js"></script>
</head>
<body>
<button onclick="callMatchertrue()">callTrue</button>
<button onclick="callMatcherfalse()">callFalse</button>
</body>
</html>
```

4.3.3 Creating a web3 object

```
if (typeof web3 !== 'undefined') {
web3 = new Web3(web3.currentProvider);
}
else {
web3 = new Web3(new Web3.providers.HttpProvider("http://localhost: 8545"));
}
```

This code checks whether there is already an available provider; if yes, then it will set the provider to the current provider. Otherwise, it sets the web3 provider to localhost: 8001; this is where the local instance of geth is running. In order to check the availability by calling any web3 method, following piece of code is used

```
var simplecontract =
SimpleContract.at("0xd9d02a4974cbeb10406639ec9378a782bf7f4dd2");
console.log(simplecontract);
```

This line of code simply uses console.log to print the simple contract attributes. Once this call is successful, it means that the web3 object has been created correctly and HttpProvider is available. Any other call can be used to verify the availability, but as a simple example, printing simple contract attributes been used in the preceding example.

4.3.4 Contract functions

Once the web3 object is correctly created and simplecontractinstance is created, calls to the contract functions can be made easily as shown in the following example:

```
function callMatchertrue()
{
var txn = simplecontractinstance.Matcher.call(12);
{
};
console.log("return value: " + txn);
}
function callMatcherfalse()
{
var txn = simplecontractinstance.Matcher.call(1);{
};
console.log("return value: " + txn);
}
```

Calls can be made using simplecontractinstance.Matcher.call and then by passing the value for the argument. Recall the Matcher function in solidity code:

function Matcher (uint8 x) returns (bool)

It takes one argument x of type uint8 and returns a Boolean value, either true or false. Accordingly, the call is made to the contract, as shown here:

```
var txn = simplecontractinstance.Matcher.call(12);
```

In the preceding example, console.log is used to print the value returned by the function call. Once the result of the call is available in the txn variable, it can be used anywhere throughout the program, for example, as a parameter for another JavaScript function.

Finally, the HTML file named index.html is created with the following code:

```
<html>
<head>
<title>SimpleContract Interactor</title>
<script src="./web3.js"></script>
<script src="./app.js"></script>
</head>
<body>
<button onclick="callMatchertrue()">callTrue</button>
<button onclick="callMatcherfalse()">callFalse</button>
</body>
</html>
```

It is recommended that a web server be running in order to serve the HTML content (index.html as an example).

This example demonstrates how the Web3 library can be used to interact with the contracts on the Ethereum blockchain.

4.4 Development Frameworks

There are various development frameworks now available for Ethereum Truffle and Embark are some of the frameworks that can be used to make the process simpler and quicker, Truffle is the most widely used framework for Ethereum development. Truffle can be used to develop a full decentralized application. We will see all the steps involved in this process such as initialization, testing, migration, and deployment. First, we will see the initialization process.

4.4.1 Truffle Intialization

Truffle can be initialized by running the following command. First, create a directory for the project, for example:

```
$ mkdir testdapp
```

Then, change directory to testdapp and run the following command:

```
$ truffle init
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!
Commands:
Compile: truffle compile
Migrate: truffle migrate
Test contracts: truffle test
```

Once the command is successful, it will create the directory structure shown here. This can be viewed using the

tree command in Linux:

```
$ tree
. ├
─
─
contracts
│ └── Migrations.sol
├── migrations
│ └── 1_initial_migration.js
├── test
├── truffle-config.js
└── truffle.js
```

3 directories, 4 files

This command creates three main directories, named contracts, migrations, and test. As seen in the preceding example, a total of 3 directories and 4 files have been created.

- ***contracts:*** This directory contains solidity contract source code files. This is where Truffle will look for solidity contract files during migration.
- ***migration***: This directory has all the deployment scripts.

- ***test***: As the name suggests, this directory contains relevant test files for applications and contracts.

Finally, Truffle configuration is stored in the *truffle.js* file, which is created in the root folder of the project from where truffle init was run. When truffle init is run, it will create a skeleton tree with files and directories. In previous versions of Truffle, this used to produce a project named MetaCoin. This project is now available as a Truffle box.

As an example, first, you will be introduced to how to use various commands in Truffle in order to test and deploy webpack box, which contains the MetaCoin project. Interaction with the contract Truffle also provides a console (a command-line interface) that allows interaction with the contracts. All deployed contracts are already instantiated and ready to use in the console. This is an REPL-based interface that means Read, Evaluate, and Print Loop. Similarly, in the geth client (via attach or console), REPL is used by exposing JSRE (JavaScript runtime environment). The console can be accessed by issuing the following command: as "$ truffle console" Once the console is available, various methods can be run in order to query the contract.

4.5 Hyperledger as a protocol

Hyperledger 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.

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.

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. In this section, let us learn the design architecture, and implementation of Fabric framework.

4.5.1 Reference Architecture

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 highlevel components are shown in the reference architecture diagram shown here:

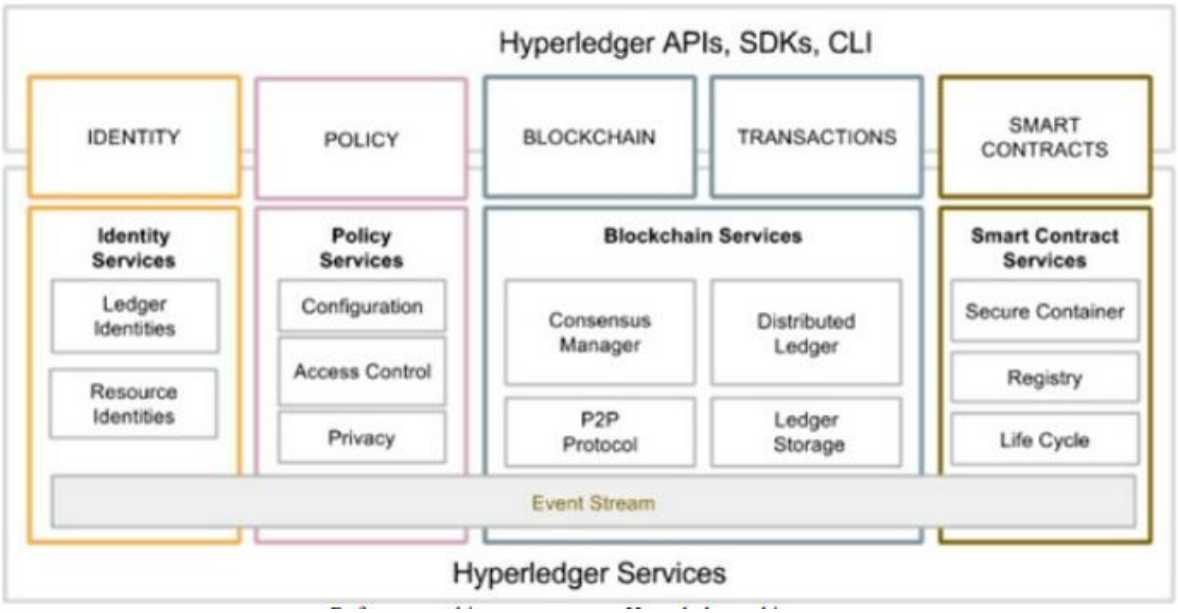

Fig 4.5.1 Hyperledger Architecture

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.

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.

4.5.2 Requirements off 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.

4.5.2.1 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.

4.5.2.2 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. 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.

4.5.2.3 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.

4.5.2.4 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.

4.5.2.5 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.

4.5.2.6 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.

4.5.2.7 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.

4.5.2.8 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.

4.5.2.9 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.

4.6 Hyperledger Fabric

The fabric is a blockchain framework is intended to provide a foundation for the development of blockchain solutions with a modular architecture. It enable a modular, open, and flexible approach towards building blockchain networks. 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 (Docker) which is used to run smart contracts in an isolated contained environment.

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). 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*.

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.

4.6.1 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 TCerts, which are used for one-time transactions. All peers and applications are idetified using certificate authority. Authentication service is provided by the certificate authority. MSPs can also interface with existing identity services like LDAP.

4.6.2 Blockchain services

Blockchain services are at the core of the Hyperledger Fabric. 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.

4.7 Distributed Ledger

Blockchain and world state are two main elements of the distributed ledger. Blockchain is simply a cryptographically linked list of blocks 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.

- **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).

4.7.1 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.

4.7.2 Applications on Fabric

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.

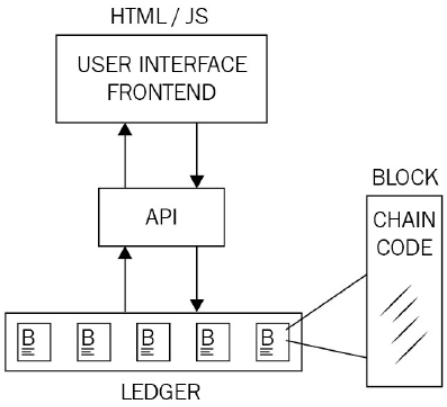

Fig 4.7.2 Typical fabric application

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 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

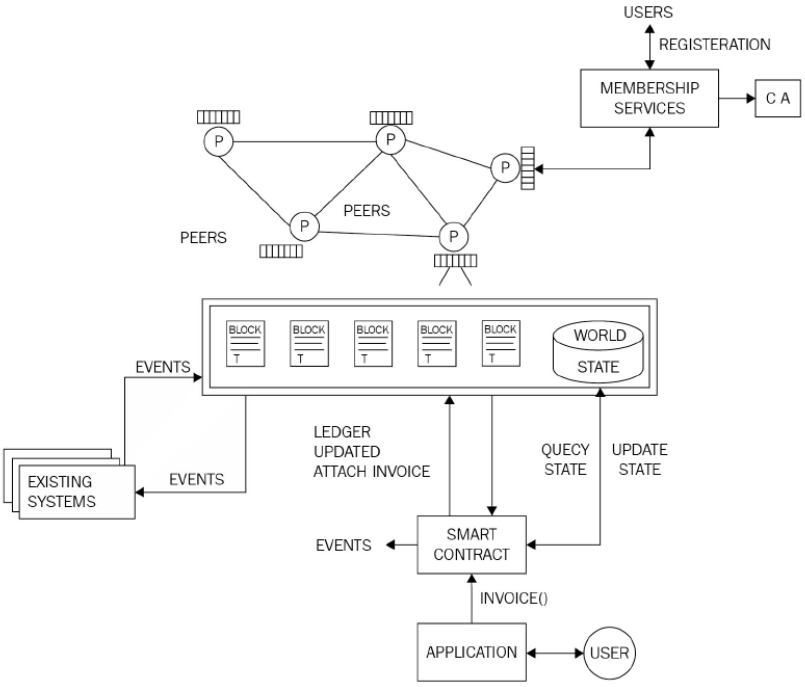

Fig.4.6 A high-level overview of Hyperledger Fabricc

This 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 through the invoice() method, and smart contracts can query or update the state of the blockchain.

4.7.3 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.

4.7.4 The transaction life cycle in Hyperledger Fabric

There are several steps that are involved in a transaction flow in Hyperledger Fabric, which are 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

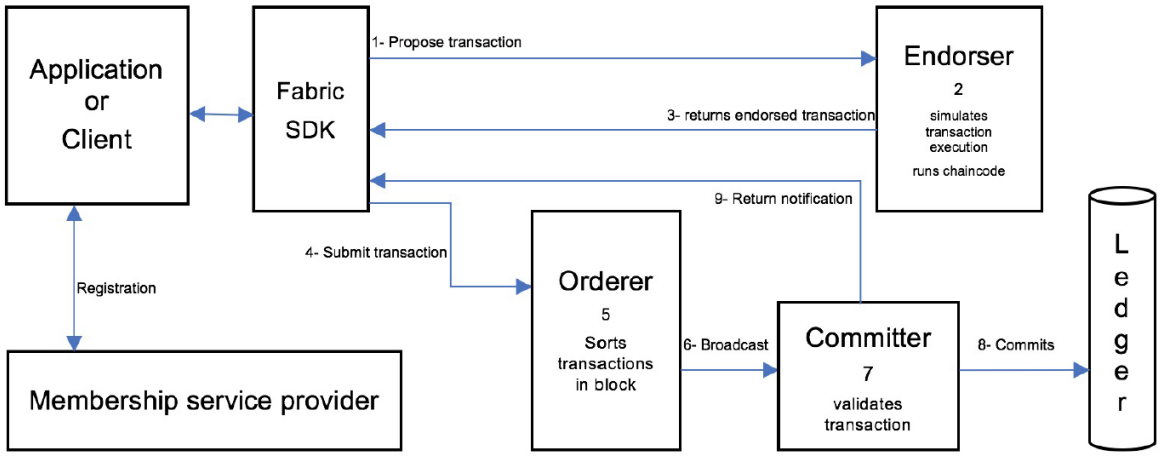

Fig 4.7.4 Transaction Flow Architecture

In this 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 Corda.

4.8 Corda – a Distributed Ledger

Corda is a distributed ledger technology (DLT) platform that was specifically designed for financial services. 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.

4.8.1 Corda 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.

4.8.1.1 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. 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.

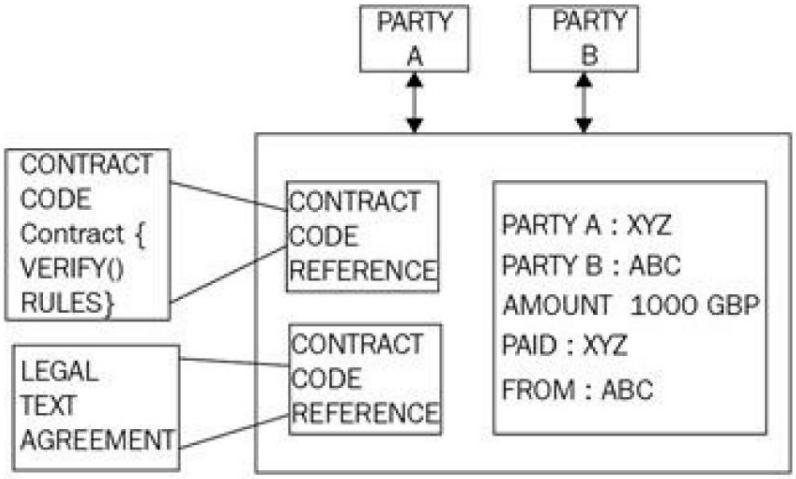

Fig. 4.8.1.1 State Object Example

4.8.1.2 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.

4.8.1.3 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.

4.8.1.4 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.

4.8.2 Components

There are many components, nodes,

4.8.2.1 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:

Fig 4.8.2.1 Two nodes communicating in a Corda network

In this 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.

4.8.2.2 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.

4.8.2.3 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. In other words, this service resolves the identities of the participants to physical nodes.

4.8.2.4 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 time stamping 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.

4.8.2.5 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.

4.8.3 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.

4.8.4 CorDapp

The core model of Corda consists of state objects, transactions, and transaction protocols, which when combinedwith 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 byte code. 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.

4.8.5 Corda Development Environment

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 with an example template available at https://github.com/corda/cordapp-template.

4.9 Summary

In this Unit, we have gone through an introduction to the Hyperledger. Firstly, the core ideas behind the Hyperledger project were discussed, and then main Hyperledger projects were discussed in detail, namely Hyperledger Fabric and Corda. In the next Unit, 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 which will be discussed in the next Unit.

UNIT 5 ALTERNATIVE BLOCK CHAINS AND NEXT EMERGING TRENDS

Learning Outcome: To know the various alternate blockchains from different perspective, adopting to the emerging trends.

Keywords: alternate blockchains, Kadena, Ripple, Stellar, Interledger, scalability

5.1 Introduction

In this Unit, you 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.

An introduction to new blockchain solutions will be given first, 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 unit will cover all these technologies and related concepts in detail. We will explore alternative block chains, Kadena, Ripple, Stellar, Quorum and various other blockchains in this Unit.

5.2 Kadena

Kadena is a private blockchain that has successfully addressed scalability and privacy issues in blockchain systems, 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.

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.

5.2.1 Consensus Mechanism in Kadena

Salable 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

5.2.2 Contract Execution 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.

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. 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.

5.3 Ripple

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.

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 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.

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.

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.

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.

5.4 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 *Transaction Type* 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 sub section.

5.4.1 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.

5.4.2 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.

5.4.3 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 isidentified 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.

5.5 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, which are presented below:

5.5.1 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.

5.5.2 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.

5.5.3 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.

5.5.4 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 the 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.

5.6 Rootstock (RSK)

Before discussing Rootstock (RSK) in detail, let us learn some concepts that are fundamental to the design of Rootstock. These concepts include sidechains, drivechains, and two-way pegging.

- **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.
- **Sidechain** is a blockchain that runs in parallel with a main blockchain and allows transfer of value between them. Thismeans 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** 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.

5.7 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.

5.7.1 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. 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.

5.7.1.2 Transactions in Quorum

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

5.8 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, Tenderrmint core & Tendermint Socket Protocol, which are described in the following section.

Tenderrmint core is a consensus engine that enables secure replication of transactions on each node in the network. Tendermint Socket Protocol (TMSP) 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.

5.9 Scalability and other Challenges

This section aims to provide an introduction to various challenges that need to be addressed before blockchains can become a mainstream technology. Even though various use cases and proof of concept systems have been developed and the technology works well for many of the scenarios, there still is a need to address some fundamental limitations that are present in blockchains in order to make this technology more adaptable. At the top of the list of these issues comes scalability and then privacy. Both of these are important limitations to address, especially as blockchains are envisioned to be used in privacy-demanding industries too. These two issues are explained in detail in the following subsections.

5.9.1 Scalability

This problem has been a focus of intense debate, rigorous research, and media attention for the last few years. This is the single most important problem that could mean the difference between wider adaptability of blockchains or limited private use only by consortiums. As a result of substantial research in this area, many solutions have been proposed, which are discussed in the following section. From a theoretical perspective, the general approach toward tackling the scalability issue generally revolves around protocol-level enhancements. For example, a commonly mentioned solution to Bitcoin scalability is to increase its block size. Other proposals include off-chain solutions that offload certain processing to off-chainnetworks, for example, off-chain state networks.

Based on the aforementioned solutions, generally, the proposals can be divided into two categories:

on-chain solutions that are based on the idea of changing fundamental protocols on which the blockchain operates, and

off-chain solutions that make use of network andprocessing resources off-chain in order to enhance the blockchain.

In another approach, a blockchain can be divided into various abstract layers called **planes**. Each plane is responsible for performing specific functions. These include the network plane, consensus plane, storage plane,view plane, and side plane. This abstraction allows bottlenecks and limitations to be addressed at each plane individually and in a structured manner. A brief

overview of each layer is given in the following subsections with some references to the Bitcoin system.

- **Network Plane**

  A key function of the network plane is transaction propagation. In Bitcoin, this plane underutilizes the network bandwidth due to the way transaction validation is performed by a node before propagation and duplication of transaction propagation, first in the transaction broadcast phase, and then after mining in a block.

- **Consensus plane**

  The second layer is called the consensus plane. This layer is responsible for mining and achieving consensus. Bottlenecks in this layer revolve around limitations in PoW algorithms whereby increasing consensus speed and bandwidth results in compromising the security of the network due to an increase in the number of forks.

- **Storage plane**

  The storage plane is the third layer, which stores the ledger. Issues in this layer revolve around the need for each node to keep a copy of the entire ledger, which leads to certain inefficiencies, such as increased bandwidth and storage requirements. Bitcoin has a method available called **pruning**, which allows a node to operate without the need to keep the full blockchain in its storage. Pruning means that when a Bitcoin node has downloaded the blockchain and validated it, it deletes the old data that it has already validated. This saves storage space. This functionality has resulted in major improvements from a storage point of view.

- **View plane**

  Next on the list is the view plane, which proposes an optimization which is based on the proposal that bitcoin miners do not need the full blockchain to operate, and a view can be constructed out of the complete ledger as a representation of the entire state of the system, which is sufficient for miners to function. Implementation of views will eliminate the need for mining nodes to store the full blockchain.

- **Side Plane**

  This plane represents the idea of off-chain transactions whereby the concept of payment or transaction channels is used to offload the processing of transactions between participants but is still backed by the main Bitcoin blockchain.

The aforementioned model can be used to describe limitations and improvements in current blockchain designs in a structured manner. Also, there are several general strategies

that have been proposed over the last few years which can address the limitations in current blockchain designs such as Ethereum and Bitcoin. These approaches are also characterized and discussed individually in the following section.

5.9.1.1 Block size increase

This is the most debated proposal for increasing blockchain performance (transaction processing throughput). Currently, Bitcoin can process only about three to seven transactions per second, which is a major inhibiting factor in adapting the Bitcoin blockchain for processing microtransactions. Block size in Bitcoin is hardcoded to be 1 MB, but if the block size is increased, it can hold more transactions and can result in faster confirmation time. There are several **Bitcoin Improvement Proposals** (**BIPs**) made in favor of block size increase. These include BIP 100, BIP 101, BIP 102, BIP 103, and BIP 109.

5.9.1.2 Block Interval Reduction

Another proposal is to reduce the time between each block generation. The time between blocks can bedecreased to achieve faster finalization of blocks but may result in less security due to the increased number of forks. Ethereum has achieved a block time of approximately 14 seconds. This is a significant improvement from the Bitcoin blockchain, which takes 10 minutes to generate a new block. In Ethereum, the issue of high orphaned blocks resulting from smaller times between blocks is mitigated by using the **Greedy Heaviest Observed Subtree** (**GHOST**) protocol whereby orphaned blocks (uncles) are also included in determining the valid chain. Once Ethereum moves to **Proof of Stake** (**PoS**), this will become irrelevant as no mining will be required and almost immediate finality of transactions can be achieved.

5.9.1.3 Invertible Bloom Lookup Tables

This is another approach that has been proposed to reduce the amount of data required to be transferred between the Bitcoin nodes. In this approach, it does not result in a hard fork of Bitcoin if implemented. The key idea is based on the fact that there is no need to transfer all transactions between nodes; instead, only those that are not already available in the transaction pool of the syncing node are transferred. This allows quicker transaction pool synchronization between nodes, thus increasing the overall scalability and speed of the Bitcoin network.

5.9.1.4 State Channels

This is another approach proposed for speeding up the transaction on a blockchain network. The basic idea is to use side channels for state updating and processing transactions

off the main chain; once the state is finalized, it is written back to the main chain, thus offloading the time-consuming operations from the main blockchain.

State channels work by performing the following three steps:

1. First, a part of the blockchain state is locked under a smart contract, ensuring the agreement and business logic between participants.
2. Now off-chain transaction processing and interaction is started between the participants that update the state only between themselves for now. In this step, almost any number of transactions can be performed without requiring the blockchain and this is what makes the process fast and a best candidate for solving blockchain scalability issues. However, it could be argued that this is not a real on-blockchain solution such as, for example, sharding, but the end result is a faster, lighter, and robust network which can prove very useful in micropayment networks, IoT networks, and many other applications.
3. Once the final state is achieved, the state channel is closed and the final state is written back to the main blockchain. At this stage, the locked part of the blockchain is also unlocked.

This process is shown in the following diagram:

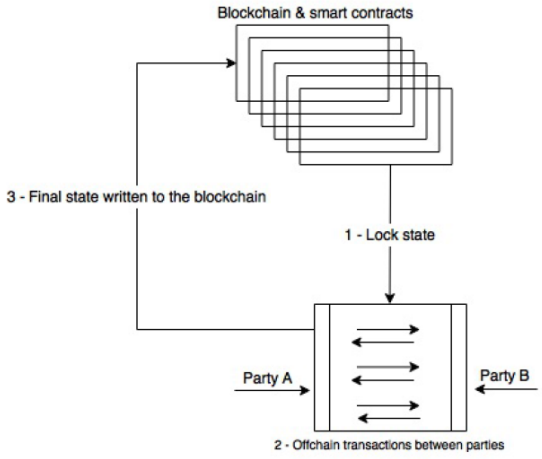

Fig 5.9.1.4 State Channels

This technique has been used in the Bitcoin lightning network and Ethereum's Raiden.

5.9.1.5 Private blockchain

Private blockchains are inherently fast because no real decentralization is required and participants on the network do not need to mine; instead, they can only validate transactions. This can be considered as a workaround to the scalability issue in public blockchains; however, this is not the solution to the scalability problem. Also, it should be noted that private

blockchains are only suitable in specific areas and setups such as enterprise environments where all participants are known.

5.9.1.6 Sidechains & Subchains

Sidechains can improve scalability indirectly by allowing many sidechains to run along with the main blockchain while allowing usage of perhaps comparatively less secure and faster sidechains to perform transactions but still pegged with the main blockchain. The core idea of sidechains is called a two-way peg, which allows transfer of coins from a parent chain to a sidechain and vice versa. Subchain is a relatively new technique, based on the idea of weak blocks that are created in layers until a strong block is found. Weak blocks can be defined as those blocks that have not been able to be mined by meeting the standard network difficulty criteria but have done enough work to meet another weaker difficulty target. Miners can build subchains by layering weak blocks on top of each other unless a block is found that meets the standard difficulty target. At this point, the subchain is closed and becomes the strong block. Advantages of this approach include reduced waiting time for the first verification of a transaction. This technique also results in a reduced chance of orphaning blocks and speeds up transaction processing. This is also an indirect way of addressing the scalability issue. Subchains do not require any soft fork or hard fork to implement but need acceptance by the community.

5.9.1.7 Plasma

It is based on the idea of running smart contracts on root blockchain (Ethereum MainNet) and have child blockchains that perform high number of transactions to feedback small amounts of commitments to the parent chain. In this scheme, blockchains are arranged in a tree hierarchy with mining performed only on the root (main) blockchain which feeds the proofs of security down to child chains. This is also called a Layer-2 system, like state channels also operate on Layer 2, and not on the main chain.

5.9.2 Privacy

Privacy of transactions is a much-desired property of blockchains. However, due to its very nature, especially in public blockchains, everything is transparent, thus inhibiting its usage in various industries where privacy is of paramount importance, such as finance, health, and many others. There are different proposals made to address the privacy issue and some progress has already been made. Several techniques, such as **Indistinguishability Obfuscation** (**IO**), usage of homomorphic encryption, ZKPs, and ring signatures, which are discussed in the following sections.

5.9.2.1 Indistinguishability Obfuscation

This cryptographic technique may serve as a silver bullet to all privacy and confidentiality issues in blockchains but the technology is not yet ready for production deployments. IO allows for code obfuscation, which is a very ripe research topic in cryptography and, if applied to blockchains, can serve as an unbreakable obfuscation mechanism that will turn smart contracts into a black box. The key idea behind IO is what's called by researchers a multilinear jigsaw puzzle, which basically obfuscates program code by mixing it with random elements, and if the program is run as intended, it will produce expected output but any other way of executing would render the program look random and garbage.

5.9.2.2 Homomorphic encryption

This type of encryption allows operations to be performed on encrypted data. Imagine a scenario where the data is sent to a cloud server for processing. The server processes it and returns the output without knowing anything about the data that it has processed. This is also an area ripe for research and fully homomorphic encryption that allows all operations on encrypted data is still not fully deployable in production; however, major progress in this field has already been made. Once implemented on blockchains, it can allow processing on ciphertext which will allow privacy and confidentiality of transactions inherently. For example, the data stored on the blockchain can be encrypted using homomorphic encryption and computations can be performed on that data without the need for decryption, thus providing privacy service on blockchains. This concept has also been implemented in a project named Enigma which is available online at (https://www.media.mit.edu/projects/enigma/overview/) by MIT's Media Lab. Enigma is a peer-to-peer network which allows multiple parties to perform computations on encrypted data without revealing anything about the data.

5.9.2.3 Zero-Knowledge Proofs

ZKPs have recently been implemented in Zcash successfully, as seen in Chapter 8, Alternative Coins. More specifically, SNARK (short for Succinct Non-Interactive Argument of Knowledge) have been implemented in order to ensure privacy on the blockchain. The same idea can be implemented in Ethereum and other blockchains also. Integrating Zcash on Ethereum is already a very active research project being run by the Ethereum R&D team and the Zcash Company.

5.9.3 Security

Even though blockchains are generally secure and make use of asymmetric and symmetric cryptography as required throughout the blockchain network, there still are few

caveats that can result in compromising the security of the blockchain. There are a few examples of transaction malleability, eclipse attacks, and the possibility of double spending in bitcoin that, in certain scenarios, have been shown to work by various researchers. Transaction malleability opens up the possibility of double withdrawal or deposit by allowing a hacker to change a transaction's unique ID before the Bitcoin network can confirm it, resulting in a scenario where it would seem that transactions did not occur. BIP 62 is one of the proposals along with SegWit that have suggested solutions to solve this issue. It should be noted that this is a problem only in the case of unconfirmed transactions, that is, scenarios where operational processes rely on unconfirmed transactions. In the case of normal applications that only rely on confirmed transactions, this is not an issue. Information eclipse attacks in Bitcoin can result in double spending. The idea behind eclipse attacks is that the Bitcoin node is tricked into connecting only with the attacker node IPs. This opens up the possibility of a 51 % attack by the attacker. This has been addressed to some extent in Bitcoin client v0.10.1.

5.9.3.1 Smart contract security

Recently, a lot of work has been started in smart contract security and, especially, formal verification of smart contracts is being discussed and researched. This was all triggered especially due to the infamous DAO hack. Formal verification is a process of verifying a computer program to ensure that it satisfies certain formal statements. This is now a new concept and there are a number of tools available for other languages that achieve this; for example, Frama-C (https://frama-c.com) is available for analyzing C programs. The key idea behind formal verification is to convert the source program into a set of statements that is understandable by the automated provers. For this purpose, Why3 (http://why3.lri.fr) is commonly used, and a formal verifier for Solidity also makes use of that. An experimental but operational verifier is available in browser Solidity already. Smart contract security is of paramount importance now, and many other initiatives have also been taken in order to devise methods that can analyze Solidity programs and find bugs.

This weakness is a type of race condition. It is also called frontloading and is possible due to the fact that the order of transactions within a block can be manipulated. As all transactions first appear in the memory pool, the transactions there can be monitored before they are included in the block. This allows a transaction to be submitted before another transaction, thus leading to controlling the behavior of a smart contract.

Timestamp dependency bugs are possible in scenarios where the timestamp of the block is used as a source of some decision-making within the contract, but timestamps can be manipulated by the miners. Call stack depth limit is another bug that can be exploited due to the fact that

the maximum call stack depth of EVM is 1,024 frames. If the stack depth is reached while the contract is executing then, in certain scenarios, the send or call instruction can fail, resulting in non-payment of funds. The call stack depth bug was addressed in the EIP 50 hard fork https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md.

The reentrancy bug was exploited in the DAO attack to siphon out millions of dollars into a child DAO. The reentrancy bug basically means that a function can be called repeatedly before the previous (first) invocation of the functions has completed. This is particularly unsafe in Ether withdrawal functions in Solidity smartcontracts.

In addition to the aforementioned bugs, there are several other problems that should be kept in mind while writing contracts. These bugs include that fact that if sending funds to another contract, handle it carefully because send can fail and even if throw is used as a catch-all mechanism, it will not work.

Other standard software bugs such as integer overflow and underflow are also quite significant and any use of integer variables should be carefully implemented in Solidity. For example, a simple program where uint8 is used to parse through elements of an array with more than 255 elements can result in an endless loop. This occurs because uint8 is limited to 256 numbers.

5.10 Summary

In this Unit, readers have been introduced to the security, confidentiality, and privacy aspects of blockchain technology. Privacy was discussed, which is another major inhibiting factor in adapting public blockchains for various industries.

Short Questions:

1. Write any two alternate blockchains.
2. What is interledger?
3. What is state channel?
4. What are the factors to be considered in smart contract security?
5. What is sidechain?
6. What is race condition in smart contract

Long Questions:

1. Describe consensus mechanism in Kadena
2. Explain the process of contract execution in Kadena
3. What are the components of Ripple? Explain
4. What are the limitations of blockchain? How it can be overcome?
5. Explain the functioning of state channels with a neat sketch
6. What are the various layers in scalability? Explain