KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

EDUCATION IS DEDICATION

CST428
BLOCKCHAIN TECHNOLOGIES

Assignment II

Name :
Sr No :
Roll No :

Blockchain

In 2008, a groundbreaking paper, entitled *Bitcoin: A Peer-to-Peer Electronic Cash System*, was written on the topic of peer-to-peer e-cash under the pseudonym of *Satoshi Nakamoto*.

This paper is available at `https://bitcoin.org/bitcoin.pdf`.

It introduced the term **chain of blocks**. No one knows the actual identity of Satoshi Nakamoto. After introducing Bitcoin in 2009, he remained active in the Bitcoin developer community until 2011. He then handed over Bitcoin development to its core developers and simply disappeared. Since then, there has been no communication from him whatsoever, and his existence and identity are shrouded in mystery. The term "chain of blocks" evolved over the years into the word "blockchain."

As stated previously, blockchain technology incorporates a multitude of applications that can be implemented in various economic sectors. Particularly in the finance sector, significant improvement in the performance of financial transactions and settlements manifests as highly desirable time-and-cost reductions. Additional light will be shed on these aspects of blockchain in *Chapter 19*, *Blockchain – Outside of Currencies*, where practical use cases will be discussed in detail for various industries. For now, it is sufficient to say that parts of nearly all economic sectors have already realized the potential and promise of blockchain, and have embarked, or will do so soon, on the journey to capitalize on the benefits of blockchain technology.

Blockchain defined

A good place to start learning what blockchain is would be to see its definition. There are some different ways that blockchain may be defined; following are two of the most widely accepted definitions:

Layman's definition: Blockchain is an ever-growing, secure, shared recordkeeping 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.

Technical definition: Blockchain is 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.

Now, let's examine things in some more detail. We will look at the keywords from the technical definition one by one.

Peer-to-peer

The first keyword in the technical definition is **peer-to-peer**, or **P2P** This means that there is no central controller in the network, and all participants (nodes) talk to each other directly. This property allows for transactions to be conducted directly among the peers without third-party involvement, such as by a bank.

Distributed ledger

Dissecting the technical definition further reveals that blockchain is a "distributed ledger," which 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.

Cryptographically secure

Next, we see that this ledger is "cryptographically secure," which means that cryptography has been used to provide security services that make this ledger secure against tampering and misuse. These services include non-repudiation, data integrity, and data origin authentication. You will see how this is achieved later in *Chapter 4*, *Public Key Cryptography*, which introduces the fascinating world of cryptography.

Append-only

Another property that we encounter is that blockchain is "append-only," which means that data can only be added to the blockchain in *time-sequential order*. This property implies that once data is added to the blockchain, it is almost impossible to change that data and it can be considered practically immutable. In other words, blocks added to the blockchain cannot be changed, which allows blockchain to become an immutable and tamper-proof ledger of transactions.

However, remember that it can be changed in rare scenarios wherein collusion against the blockchain network by bad actors succeeds in gaining more than 51 percent of the power. Otherwise, the blockchain is practically immutable.

There may be some legitimate reasons to change data in the blockchain once it has been added, such as the "right to be forgotten" or "right to erasure" (also defined in the GDPR ruling: `https://gdpr-info.eu/art-17-gdpr/`).

However, those are individual cases that need to be handled separately and that require an elegant technical solution. For all practical purposes, blockchain is indeed immutable and cannot be changed.

Updatable via consensus

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 that ensure all parties agree on the final state of the data on the blockchain network and resolutely agree upon it to be true. Consensus algorithms are introduced later in this chapter, and then in more detail in *Chapter 5*, *Consensus Algorithms*.

Blockchain architecture

Having detailed the primary features of blockchain, we are now in a position to begin to look at its actual architecture. We'll begin by looking at how blockchain acts as a layer within a distributed peer-to-peer network.

Blockchain by layers

Blockchain can be thought of as a layer of a distributed peer-to-peer network running on top of the internet, as can be seen in the following diagram. It is analogous to SMTP, HTTP, or FTP running on top of TCP/IP:

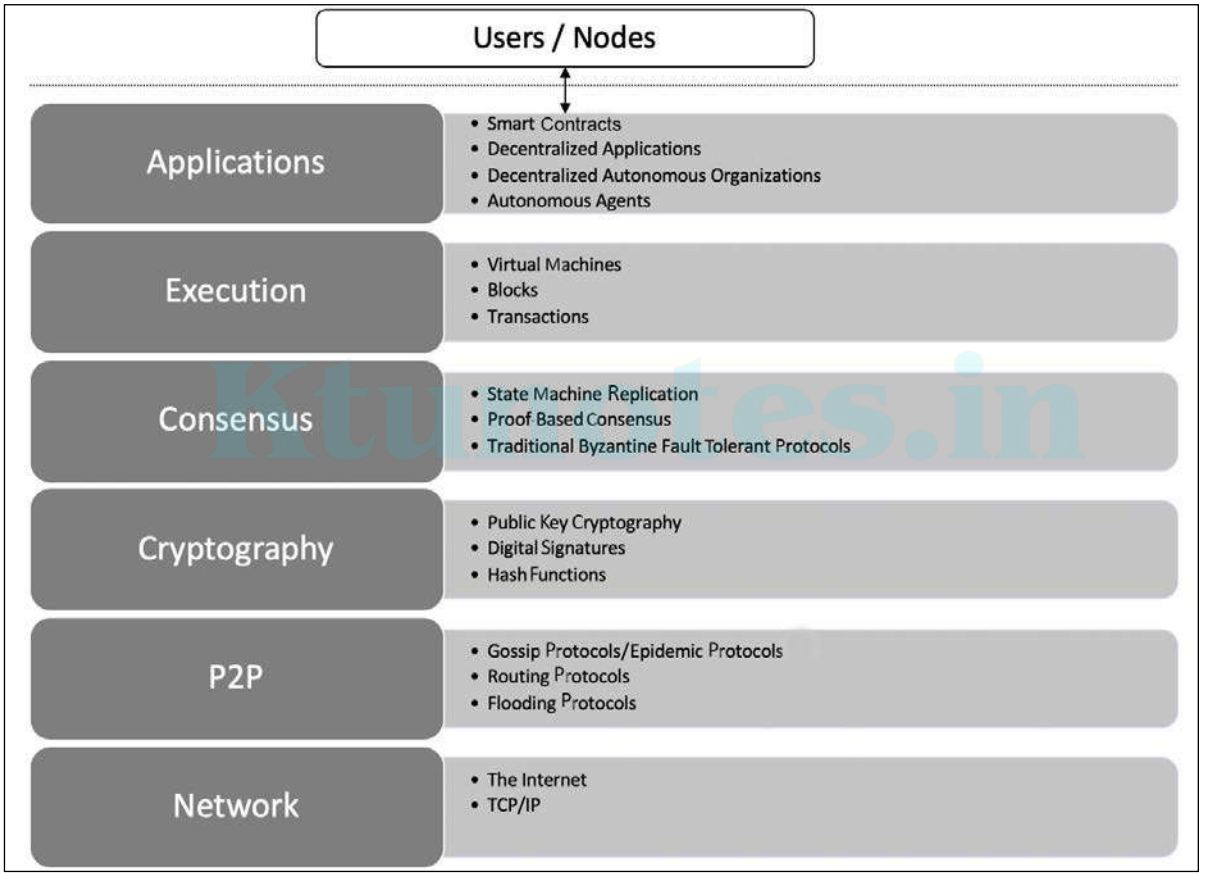


Figure 1.5: The architectural view of a generic blockchain

Now we'll discuss all these elements one by one:

- In the preceding diagram, the lowest layer is the **Network**, which is usually the internet and provides a base communication layer for any blockchain.
- A peer-to-peer network runs on top of the **Network** layer, which consists of information propagation protocols such as gossip or flooding protocols.

- After this comes the **Cryptography** layer, which contains crucial cryptographic protocols that ensure the security of the blockchain. These cryptographic protocols play a vital role in the integrity of blockchain processes, secure information dissemination, and blockchain consensus mechanisms. This layer consists of public key cryptography and relevant components such as digital signatures and cryptographic hash functions. Sometimes, this layer is abstracted away, but it has been included in the diagram because it plays a fundamental role in blockchain operations.
- Next comes the **Consensus** layer, which is concerned with the usage of various consensus mechanisms to ensure agreement among different participants of the blockchain. This is another crucial part of the blockchain architecture, which consists of various techniques such as SMR, proof-based consensus mechanisms, or traditional (from traditional distributed systems research) Byzantine fault-tolerant consensus protocols.
- Further to this, we have the **Execution** layer, which can consist of virtual machines, blocks, transaction, and smart contracts. This layer, as the name suggests, provides executions services on the blockchain and performs operations such as value transfer, smart contract execution, and block generation. Virtual machines such as **Ethereum Virtual Machine** (**EVM**) provide an execution environment for smart contracts to execute.
- Finally, we have the **Applications** layer, which is composed of smart contracts, decentralized applications, DAOs, and autonomous agents. This layer can effectively contain all sorts of various user level agents and programs that operate on the blockchain. Users interact with the blockchain via decentralized applications. We will discuss more about decentralized applications in *Chapter 2*, *Decentralization*.

All these concepts will be discussed in detail later in this book in various chapters. Next, we'll look at blockchain from more of a business-oriented perspective.

Blockchain in business

From a business standpoint, a blockchain can be defined as a platform where peers can exchange value/e-cash using transactions without the need for a centrally trusted arbitrator. For example, for cash transfers, banks act as a trusted third party. In financial trading, a central clearing house acts as a trusted third party between two or more trading parties. This concept is compelling, and, once you absorb it, you will realize the enormous potential of blockchain technology. This disintermediation allows blockchain to be a decentralized consensus mechanism where no single authority is in charge of the database. Immediately, you'll see a significant benefit of decentralization here, because if no banks or central clearing houses are required, then it immediately leads to cost savings, faster transaction speeds, and more trust.

We've now looked at what blockchain is at a fundamental level. Next, we'll go a little deeper and look at some of the elements that comprise a blockchain.

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. The structure of a generic blockchain can be visualized with the help of the following diagram:

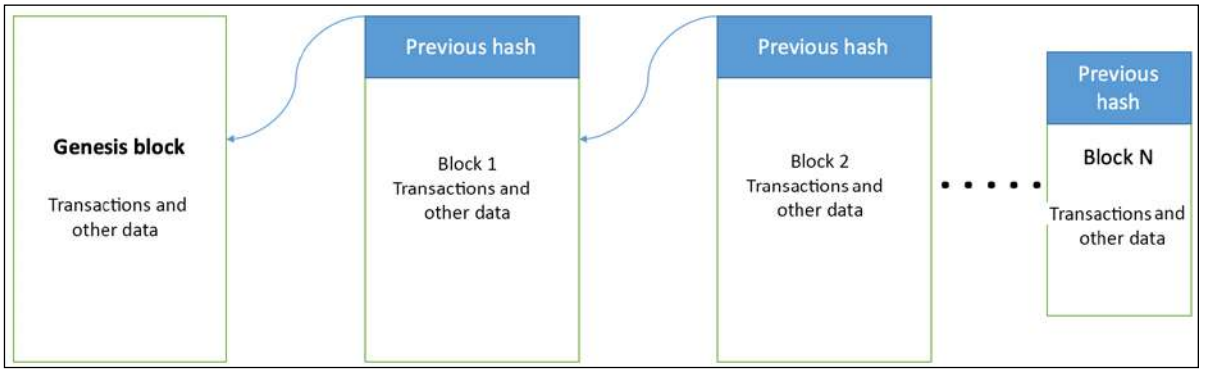


Figure 1.6: Generic structure of a blockchain

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.
- **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. A block is composed of a block header and a selection of transactions bundled together and organized logically. A block contains several elements, which we introduce as follows:
    - A reference to a previous block is also included in the block unless it is a genesis block. This reference is the hash of the header of the previous 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.
    - 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. A block also includes the nonce value.
    - A **timestamp** is the creation time of the block.

- **Merkle root** is a hash of all of the nodes of a Merkle tree. In a blockchain block, it is the combined hash of the transactions in the block. Merkle trees are widely used to validate 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. We will elaborate further on these concepts in *Chapter 4*, *Public Key Cryptography*.
- In addition to the block header, the block contains transactions that make up the block body. 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 contains transactions and its size varies depending on the type and design of the blockchain.

The following structure is a simple block diagram that depicts a block. Specific block structures relative to their blockchain technologies will be discussed later in the book with greater in-depth technical detail:

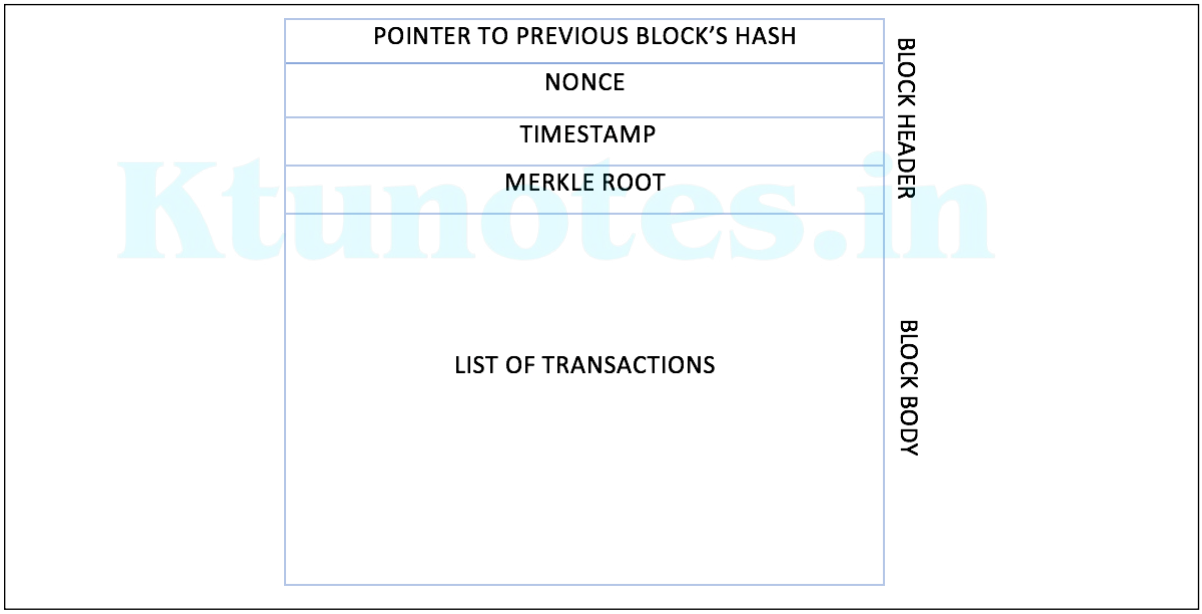


Figure 1.7: The generic structure of a block

Generally, however, there are just a few attributes that are essential to the functionality of a block: the block header, which is composed of the hash of the previous block's header, the timestamp, nonce, Merkle root, and the block body that contains the transactions. There are also other attributes in a block, but generally, the components introduced in this section are usually available in a block.

- **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.
- **The 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 consists of sets of commands that allow nodes to transfer bitcoins from one address to another. Script is a limited language, in the sense that it only allows essential operations that are necessary for executing transactions, but it does not allow for arbitrary program development.

Think of the scripting language as a calculator that only supports standard preprogrammed arithmetic operations. As such, the Bitcoin Script language cannot be called "Turing complete." In simple words, a Turing complete language means that it can perform any computation. It is named after Alan Turing, who developed the idea of a Turing machine that can run any algorithm however complex. Turing complete languages need loops and branching capability to perform complex computations. Therefore, Bitcoin's scripting language is not Turing complete, whereas Ethereum's Solidity language is.

To facilitate arbitrary program development on a blockchain, a Turing complete programming language is needed, and it is now a very desirable feature to have for blockchains. Think of this as a computer that allows the development of any program using programming languages. Nevertheless, the security of such languages is a crucial question and an essential and ongoing research area. We will discuss this in greater detail in *Chapter 6*, *Introducing Bitcoin*, *Chapter 10*, *Smart Contracts*, and the chapters on *Ethereum Development*, later in this book.

- **Virtual machine**: This is an extension of the transaction script introduced previously. 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. However, virtual machines are not available on all blockchains. Various blockchains use virtual machines to run programs such as **Ethereum Virtual Machine** (**EVM**) and **Chain Virtual Machine** (**CVM**). EVM is used in the 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 by nodes on the blockchain network as a result of transaction execution.
- **Smart contracts**: 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. The *smart contract* feature is not available on all blockchain platforms, but it is now becoming a very desirable feature due to the flexibility and power that it provides to blockchain applications. Smart contracts have many use cases, including but not limited to identity management, capital markets, trade finance, record management, insurance, and e-governance. Smart contracts will be discussed in more detail in *Chapter 10*, *Smart Contracts*.

- **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). Nodes can also perform other functions such as simple payment verification (lightweight nodes), validation, and many other functions depending on the type of the blockchain used and the role assigned to the node. Nodes also perform a transaction signing function. Transactions are first created by nodes and then also digitally signed by nodes using private keys as proof that they are the legitimate owner of the asset that they wish to transfer to someone else on the blockchain network. This asset is usually a token or virtual currency, such as Bitcoin, but it can also be any real-world asset represented on the blockchain by using tokens. There are also now standards related to tokens; for example, on Ethereum, there are ERC20, ERC721, and a few others that define the interfaces and semantics of tokenization. We will cover these in *Chapter 12*, *Further Ethereum*.

A high-level diagram of blockchain architecture highlighting the key elements mentioned previously is shown as follows:

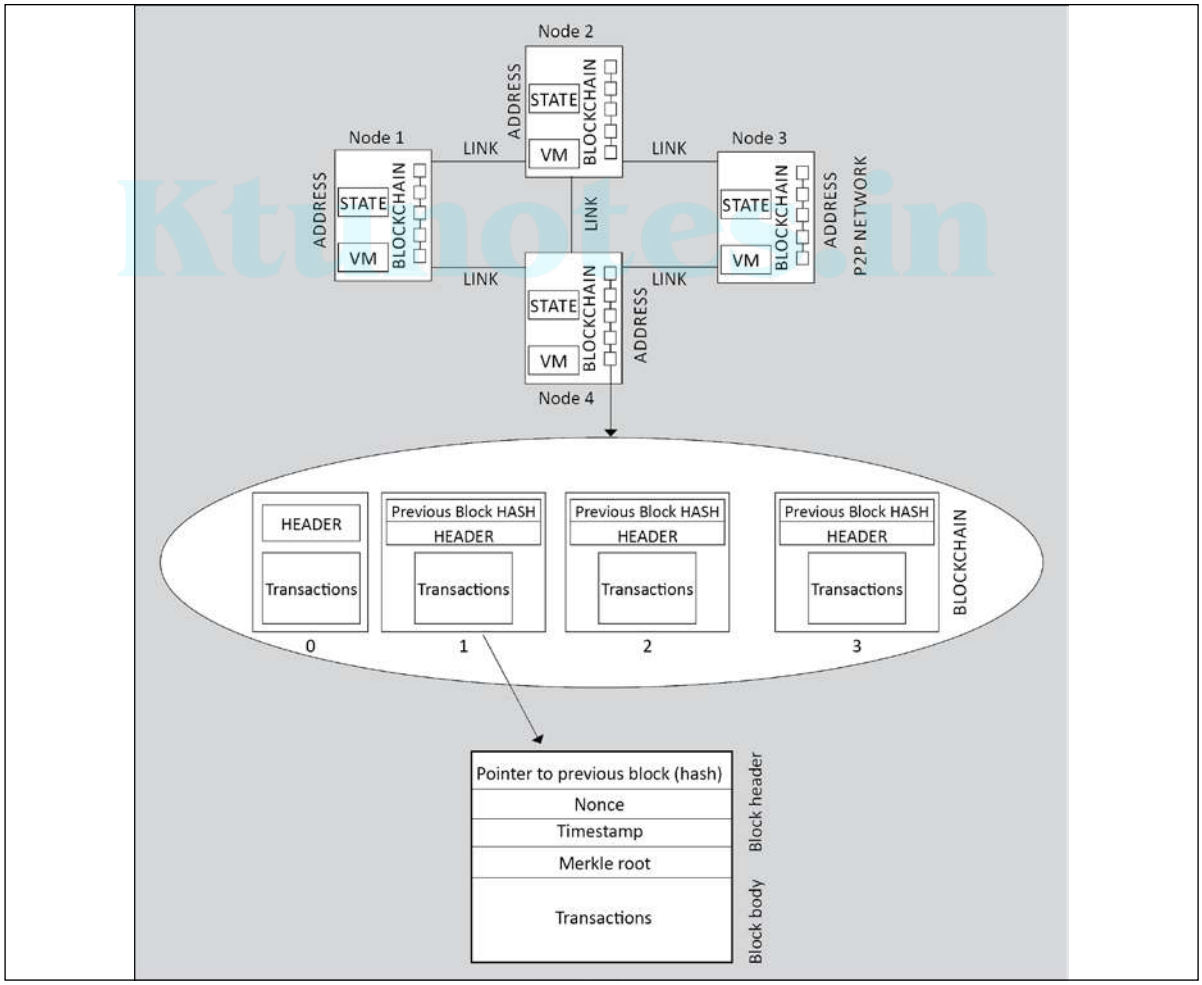


Figure 1.8: Generic structure of a blockchain network

The preceding diagram shows a four-node blockchain network (top), each maintaining a chain of blocks, virtual machine, state machine, and address. The blockchain is then further magnified (middle) to show the structure of the chain of blocks, which is again magnified (bottom) to show the structure of a transaction. Note that this is a generic structure of a blockchain; we will see specific blockchains structures in detail in the context of Ethereum and Bitcoin blockchains later in this book.

How blockchain works

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 validate and digitally sign the transactions. A critical decision that every blockchain network has to make is to figure out which node will append the next block to the blockchain. This decision is made using a *consensus mechanism*. The consensus mechanism will be described later in this chapter. For now, we will look at how a blockchain validates transactions and creates and adds blocks to grow the blockchain.

We will look at a general scheme for creating blocks. This scheme is presented here to give you a general idea of how blocks are generated and what the relationship is between transactions and blocks:

1. **Transaction is initiated**: A node starts a transaction by first creating it 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 the transfer of value between users on the blockchain network. The transaction data structure usually consists of some logic of transfer of value, relevant rules, source and destination addresses, and other validation information. Transactions are usually either a cryptocurrency transfer (transfer of value) or smart contract invocation that can perform any desired operation. A transaction occurs between two or more parties. This will be covered in more detail in specific chapters on Bitcoin and Ethereum later in the book.
2. **Transaction is validated and broadcast**: A transaction is propagated (broadcast) usually by using data-dissemination protocols, such as *Gossip protocol*, to other peers that validate the transaction based on preset validity criteria. Before a transaction is propagated, it is also verified to ensure that it is valid.
3. **Find new block**: When the transaction is received and validated by special participants called miners on the blockchain network, it is included in a block, and the process of mining starts. This process is also sometimes referred to as "finding a new block." Here, nodes called miners race to finalize the block they've created by a process known as mining.
4. **New block found**: Once a miner solves a mathematical puzzle (or fulfills the requirements of the consensus mechanism implemented in a blockchain), the block is considered "found" and finalized. At this point, the transaction is considered confirmed. Usually, in cryptocurrency blockchains such as Bitcoin, the miner who solves the mathematical puzzle is also rewarded with a certain number of coins as an incentive for their effort and the resources they spent in the mining process.

5. **Add new block to the blockchain**: The newly created block is validated, transactions or smart contracts within it are executed, and it is propagated to other peers. Peers also validate and execute the block. It now becomes part of the blockchain (ledger), and the next block links itself cryptographically back to this block. This link is called a hash pointer.

This process can be visualized in the diagram as follows:

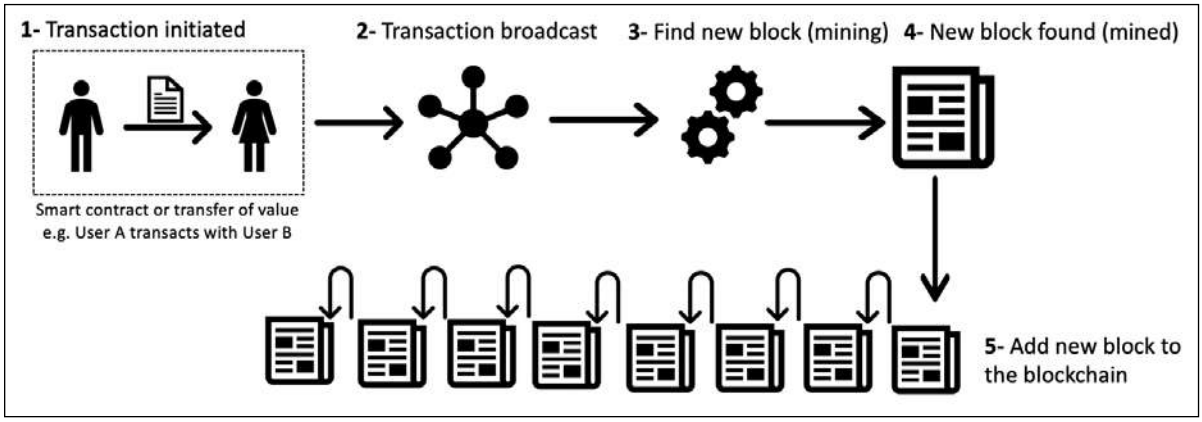


Figure 1.9: How a block is generated

This completes the basic introduction to blockchain. In the next section, you will learn about the benefits and limitations of this technology.

Benefits, features, and limitations of blockchain

Numerous advantages of blockchain technology have been discussed in many industries and proposed by thought leaders around the world who are participating in the blockchain space. The notable benefits of blockchain technology are as follows:

1. **Decentralization**: This is a core concept and benefit of blockchain. There is no need for a trusted third party or intermediary to validate transactions; instead, a consensus mechanism is used to agree on the validity of transactions.
2. **Transparency and trust**: As blockchains are shared and everyone can see what is on the blockchain, this allows the system to be transparent. As a result, trust is established. This is more relevant in scenarios such as the disbursement of funds or benefits where personal discretion in relation to selecting beneficiaries needs to be restricted.
3. **Immutability**: Once the data has been written to the blockchain, it is extremely difficult to change it back. It is not genuinely immutable, but because changing data is so challenging and nearly impossible, this is seen as a benefit to maintaining an immutable ledger of transactions.

4. **High availability**: As the system is based on thousands of nodes in a peer-to-peer network, and the data is replicated and updated on every node, the system becomes highly available. Even if some nodes leave the network or become inaccessible, the network as a whole continues to work, thus making it highly available. This redundancy results in high availability.
5. **Highly secure**: All transactions on a blockchain are cryptographically secured and thus provide network integrity. Any transactions posted from the nodes on the blockchain are verified based on a predetermined set of rules. Only valid transactions are selected for inclusion in a block. The blockchain is based on proven cryptographic technology that ensures the integrity and availability of data. Generally, confidentiality is not provided due to the requirements of transparency. This limitation is the leading barrier to its adoption by financial institutions and other industries that require the privacy and confidentiality of transactions. As such, the privacy and confidentiality of transactions on the blockchain are being researched very actively, and advancements are already being made. It could be argued that, in many situations, confidentiality is not needed and transparency is preferred. For example, with Bitcoin, confidentiality is not an absolute requirement; however, it is desirable in some scenarios. A more recent example is Zcash (`https://z.cash`), which provides a platform for conducting anonymous transactions. Other security services, such as non-repudiation and authentication, are also provided by blockchain, as all actions are secured using private keys and digital signatures.
6. **Simplification of current paradigms**: The current blockchain model in many industries, such as finance or health, is somewhat disorganized. In this model, multiple entities maintain their own databases and data sharing can become very difficult due to the disparate nature of the systems. However, as a blockchain can serve as a single shared ledger among many interested parties, this can result in simplifying the model by reducing the complexity of managing the separate systems maintained by each entity.
7. **Faster dealings**: In the financial industry, especially in post-trade settlement functions, blockchain can play a vital role by enabling the quick settlement of trades. Blockchain does not require a lengthy process of verification, reconciliation, and clearance because a single version of agreed-upon data is already available on a shared ledger between financial organizations.
8. **Cost-saving**: As no trusted third party or clearing house is required in the blockchain model, this can massively eliminate overhead costs in the form of the fees, which are paid to such parties.
9. **Platform for smart contracts**: A blockchain is a platform on which programs can run that execute business logic on behalf of the users. This is a very useful feature but not all blockchains have a mechanism to execute *smart contracts*; however, this is a very desirable feature. It is available on newer blockchain platforms such as Ethereum and MultiChain, but not on Bitcoin.

Smart contracts

Blockchain technology provides a platform for running smart contracts. These are automated, autonomous programs that reside on the blockchain network and encapsulate the business logic and code needed to execute a required function when certain conditions are met. For example, think about an insurance contract where a claim is paid to the traveler if the flight is canceled. In the real world, this process normally takes a significant amount of time to make the claim, verify it, and pay the insurance amount to the claimant (traveler). What if this whole process were automated with cryptographically-enforced trust, transparency, and execution so that as soon as the smart contract received a feed that the flight in question has been canceled, it automatically triggers the insurance payment to the claimant? If the flight is on time, the smart contract pays itself.

This is indeed a revolutionary feature of blockchain, as it provides flexibility, speed, security, and automation for real-world scenarios that can lead to a completely trustworthy system with significant cost reductions. Smart contracts can be programmed to perform any actions that blockchain users need and according to their specific business requirements.

10. **Smart property**: It is possible to link a digital or physical asset to the blockchain in such a secure and precise manner that it cannot be claimed by anyone else. You are in full control of your asset, and it cannot be double-spent or double-owned. Compare this with a digital music file, for example, which can be copied many times without any controls. While it is true that many **Digital Rights Management** (**DRM**)schemes are being used currently along with copyright laws, none of them are enforceable in the way a blockchain-based DRM can be. Blockchain can provide digital rights management functionality in such a way that it can be enforced fully. There are famously broken DRM schemes that looked great in theory but were hacked due to one limitation or another. One example is the Oculus hack: `http://www.wired.co.uk/article/oculus-rift-drm-hacked`. Another example is the PS3 hack; also, copyrighted digital music, films, and e-books are routinely shared on the internet without any limitations. We have had copyright protection in place for many years, but digital piracy refutes all attempts to fully enforce the law. On a blockchain, however, if you own an asset, no one else can claim it unless you decide to transfer it. This feature has far-reaching implications, especially in DRM and e-cash systems where double-spend detection is a crucial requirement. The double-spend problem was first solved without the requirement of a trusted third party in Bitcoin.

As with any technology, some challenges need to be addressed in order to make a system more robust, useful, and accessible. Blockchain technology is no exception. In fact, much effort is being made in both academia and industry to overcome the challenges posed by blockchain technology. The most sensitive blockchain problems are as follows:

- **Scalability**: Currently, blockchain networks are not as scalable as, for example, current financial networks. This is a known area of concern and a very ripe area for research.
- **Adoption**: Often, blockchain is seen as a nascent technology. Even though this perspective is rapidly changing, there is still a long way to go before the mass adoption of this technology. The challenge here is to allow blockchain networks to be easier to use so that adoption can increase. In addition, several other challenges such as scalability (introduced previously) exist, which must be solved in order to increase adoption.
- **Regulation**: Due to its decentralized nature, regulation is almost impossible on blockchain. This is sometimes seen as a barrier toward adoption because, traditionally, due to the existence of regulatory authorities, consumers have a certain level of confidence that if something goes wrong they can hold someone accountable. However, in blockchain networks, no such regulatory authority and control exists, which is an inhibiting factor for many consumers.
- **Relatively immature technology**: As compared to traditional IT systems that have benefited from decades of research, blockchain is still a new technology and requires a lot of research to achieve maturity.
- **Privacy and confidentiality**: Privacy is a concern on public blockchains such as Bitcoin where everyone can see every single transaction. This transparency is not desirable in many industries such as the financial, law, or medical sectors. This is also a known concern and a lot of valuable research with some impeccable solutions has already been developed. However, further research is still required to drive the mass adoption of blockchain.

All of these issues and possible solutions will be discussed in detail in *Chapter 21*, *Scalability and Other Challenges*.

You now know the basics of blockchain and its benefits and limitations. Now, let's take a look at the various types of blockchain that exist.

Types of blockchain

Based on the way that blockchain has evolved over the last few years, it can be divided into multiple categories with distinct, though sometimes partially overlapping attributes. You should note that the tiers described earlier in the chapter are a different concept, whereby the logical categorization of blockchain, based upon its evolution and usage, is presented.

In this section, we will examine the different types of blockchains from a technical and business use perspective. These blockchain types can occur on any blockchain tier, as there is no direct relationship between those tiers mentioned earlier and the various types of blockchain.

In this section, we'll examine:

- Distributed ledgers
- Distributed Ledger Technology (DLT)
- Blockchains
- Ledgers

Distributed ledgers

First, I need to clarify an ambiguity. It should be noted that a *distributed ledger* is a broad term describing shared databases; hence, all blockchains technically fall under the umbrella of shared databases or distributed ledgers. Although all blockchains are fundamentally distributed ledgers, all distributed ledgers are not necessarily blockchains.

A critical difference between a distributed ledger and a blockchain is that a distributed ledger does not necessarily consist of blocks of transactions to keep the ledger growing. Rather, a blockchain is a special type of shared database that is comprised of blocks of transactions. An example of a distributed ledger that does not use blocks of transactions is R3's Corda (`https://www.corda.net`). Corda is a distributed ledger that is developed to record and manage agreements and is especially focused on the financial services industry. On the other hand, more widely known blockchains like Bitcoin and Ethereum make use of blocks to update the shared database.

As the name suggests, a distributed ledger is distributed among its participants and spread across multiple sites or organizations. This type of ledger can be either private or public. The fundamental idea here is that, unlike many other blockchains, the records are stored contiguously instead of being sorted into blocks. This concept is used in Ripple, which is a blockchain- and cryptocurrency-based global payment network.

Distributed Ledger Technology

It should be noted that over the last few years, the terms distributed ledger or DLT have grown to be commonly used to describe blockchain in the finance industry. Sometimes, blockchain and DLT are used interchangeably. Though this is not entirely accurate, it is how the term has evolved recently, especially in the finance sector. In fact, DLT is now a very active and thriving area of research in the financial sector. From a financial sector point of view, DLTs are permissioned blockchains that are used by consortiums. DLTs usually serve as a shared database, with all participants known and verified. They do not have a cryptocurrency and do not require mining to secure the ledger.

At a broader level, DLT is an umbrella term that represents Distributed Ledger Technology as a whole, comprising of blockchains and distributed ledgers of different types.

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 "un-permissioned" 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. Bitcoin and Ethereum are both considered public blockchains.

Private blockchains

As the name implies, private blockchains are just that—private. That is, they are open only to a consortium or group of individuals or organizations who have decided to share the ledger among themselves. There are various blockchains now available in this category, such as Kadena and Quorum. Optionally, both of these blockchains can also run in public mode if required, but their primary purpose is to provide a private blockchain.

Semi-private blockchains

With semi-private 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 proofs of concept 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.

Sidechains

More precisely known as "pegged sidechains," this is a concept whereby coins can be moved from one blockchain to another and then back again. Typical uses include the creation of new *altcoins* (alternative cryptocurrencies) whereby coins are burnt as a proof of an adequate stake. "Burnt" or "burning the coins" in this context means that the coins are sent to an address that is un-spendable, and this process makes the "burnt" coins irrecoverable. This mechanism is used to bootstrap a new currency or introduce scarcity, which results in the increased value of the coin.

This mechanism is also called "Proof of Burn" and is used as an alternative method for distributed consensus to PoW and **Proof of Stake** (**PoS**). The example provided previously for burning coins applies to a **one-way pegged sidechain**. The second type is called a **two-way pegged sidechain**, which allows the movement of coins from the main chain to the sidechain and back to the main chain when required.

This process enables the building of smart contracts for the Bitcoin network. Rootstock is one of the leading examples of a sidechain, which enables smart contract development for Bitcoin using this paradigm. It works by allowing a two-way peg for the Bitcoin blockchain, and this results in much faster throughput.

Permissioned ledger

A *permissioned ledger* is a blockchain where participants of the network are already known and trusted. Permissioned ledgers do not need to use a distributed consensus mechanism; instead, an agreement protocol is used to maintain a shared version of the truth about the state of the records on the blockchain. In this case, for verification of transactions on the chain, all verifiers are already preselected by a central authority and, typically, there is no need for a mining mechanism.

By definition, there is also no requirement for a permissioned blockchain to be private, as it can be a public blockchain but with regulated access control. For example, Bitcoin can become a permissioned ledger if an access control layer is introduced on top of it that verifies the identity of a user and then allows access to the blockchain.

Shared ledger

This is a generic term that is used to describe any application or database that is shared by the public or a consortium. Generally, all blockchains fall into the category of a shared ledger.

Fully private and proprietary blockchains

There is no mainstream application of these types of blockchains, as they deviate from the core concept of decentralization in blockchain technology. Nonetheless, in specific private settings within an organization, there could be a need to share data and provide some level of guarantee of the authenticity of the data.

An example of this type of blockchain might be to allow for collaboration and the sharing of data between various government departments. In that case, no complex consensus mechanism is required, apart from simple SMR and an agreement protocol with known central validators. Even in private blockchains, tokens are not really required, but they can be used as a means of transferring value or representing some real-world assets.

Tokenized 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

These 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 fully private blockchains, the only difference being that the use of tokens is not required. This can also be thought of as a shared distributed ledger used for storing and sharing data between the participants. It does have its benefits when it comes to immutability, tamper proofing, security, and consensus-driven updates but is not used for a common blockchain application of value transfer or cryptocurrency. Most of the permissioned blockchains can be seen as an example of tokenless blockchains, for example, Hyperledger Fabric or Quorum. Tokens can be built on these chains as an application, but intrinsically these blockchains do not have a token associated with them.

All the aforementioned terminologies are used in literature, but fundamentally all these blockchains are distributed ledgers and fall under the top-level category of DLTs. We can view these different types in the simple chart as follows:

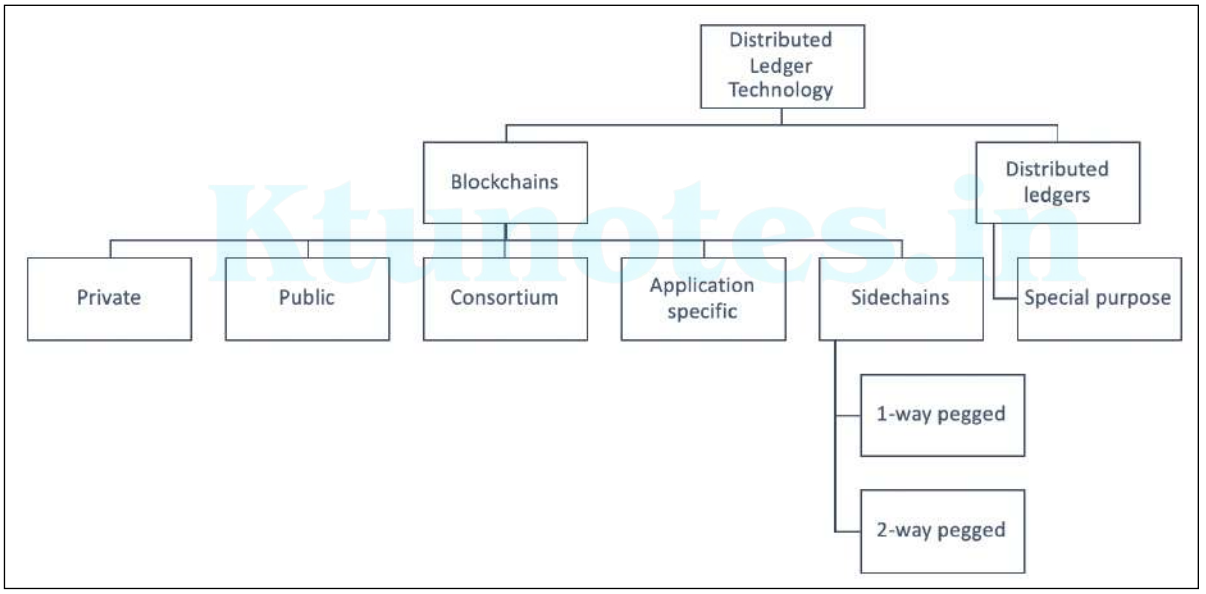


Figure 1.10: DLT hierarchy

This ends our examination of the various types of blockchain. We'll now move on to the next section to discuss the concept of consensus.

Consensus

Consensus is the backbone of a blockchain, as it provides the decentralization of control through an optional process known as **mining**. The choice of the **consensus algorithm** to utilize is 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 mechanisms that are more suitable for permissioned blockchains, such as **Proof of Authority** (**PoA**) or traditional Byzantine fault-tolerant consensus mechanisms. Therefore, it is essential to choose an appropriate consensus algorithm for a particular blockchain project.

Consensus is a process of achieving 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 (in client-server systems, for example), 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 on a common state or value among multiple nodes despite the failure of some nodes is known as **distributed consensus**.

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. With the advent of blockchain and Bitcoin, consensus mechanisms have come into the limelight again and gained considerable popularity.

There are various requirements for a consensus mechanism. The following describes these requirements:

- **Agreement**: All honest nodes decide on the same value.
- **Integrity**: This is a requirement that no node can make the decision more than once in a single consensus cycle.
- **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 correctly in the presence of faulty or malicious nodes (Byzantine nodes).
- **Termination**: All honest nodes terminate the execution of the consensus process and eventually reach a decision.

Having seen these general requirements, we'll now look at the different types of consensus mechanisms.

Types of consensus mechanisms

All consensus mechanisms are developed to deal with faults in a distributed system and to allow distributed systems to reach a final state of agreement. There are two general categories of consensus mechanisms. These categories deal with all types of faults (fail-stop types or arbitrary). These common types of consensus mechanisms are as follows:

- **Proof-based consensus mechanisms**: This arrangement requires nodes to compete in a leader-election lottery, and the node that wins proposes the final value. The algorithm works on the principle of providing proof of some work and the possession of some authority or tokens to win the right of proposing the next block. For example, the PoW mechanism used in Bitcoin falls into this category, where a miner who solves the computational puzzle as proof of computational effort expended wins the right to add the next block to the blockchain.
- **Traditional fault tolerance-based**: With no compute-intensive operations, such as partial hash inversion (as in Bitcoin PoW), this type of consensus mechanism relies on a simple scheme of nodes that publish and verify signed messages in a number of phases. Eventually, when a certain number of messages are received over a period of rounds (phases), then an agreement is reached.

To achieve fault tolerance, replication is used. This is a standard and widely used method to achieve fault tolerance. In general, there are two types of faults that a node can experience:

- **Fail-stop faults**: This type of fault occurs when a node merely has crashed. Fail-stop faults are the easier ones to deal with of the two fault types. Paxos or the RAFT protocol, introduced earlier in this chapter, are normally used to deal with this type of fault. These faults are simpler to deal with.
- **Byzantine faults**: The second type of fault is one where the faulty node exhibits malicious or inconsistent behavior arbitrarily. This type is difficult to handle since it can create confusion due to misleading information. This can be a result of an attack by adversaries, a software bug, or data corruption. SMR protocols such as **Practical Byzantine Fault Tolerance** (**PBFT**) was developed to address this second type of faults.

Many other implementations of consensus protocols have been proposed in traditional distributed systems. **Paxos** is the most famous of these protocols. It was introduced by **Leslie Lamport** in 1989. With Paxos, nodes are assigned various roles such as Proposer, Acceptor, and Learner. Nodes or processes are named replicas, and consensus is achieved in the presence of faulty nodes by an agreement among a majority of nodes.

An alternative to Paxos is RAFT, which works by assigning any of three states; that is, *Follower*, *Candidate*, or *Leader* to the nodes. A Leader is elected after a Candidate node receives enough votes, and all changes then have to go through the Leader. The Leader commits the proposed changes once replication on the majority of the follower nodes is completed.

We will briefly touch on some aspects of consensus in blockchain now, but more detail on the theory of consensus mechanisms from a distributed system point of view and also from the blockchain perspective will be presented in *Chapter 5*, *Consensus Algorithms*.

Consensus in blockchain

Consensus is a distributed computing concept that has been used in blockchain in order to provide a means of agreeing to a single version of the truth by all peers on the blockchain network. This concept was previously discussed in the *distributed systems* section of this chapter. In this section, we will address consensus in the context of blockchain technology. Some concepts presented following are still relevant to the distributed systems theory, but they are explained from a blockchain perspective.

Roughly, the following describes the two main categories of consensus mechanisms:

1. **Proof-based, leader-election lottery-based, or the Nakamoto consensus** whereby a leader is elected at random (using an algorithm) and proposes a final value. This category is also referred to as the fully decentralized or permissionless type of consensus mechanism. This type is well used in the Bitcoin and Ethereum blockchain in the form of a PoW mechanism.
2. **Byzantine fault tolerance (BFT)-based** is a more traditional approach based on rounds of votes. This class of consensus is also known as the consortium or permissioned type of consensus mechanism.

BFT-based consensus mechanisms perform well when there are a limited number of nodes, but they do not scale well. On the other hand, leader-election lottery-based (PoW) consensus mechanisms scale very well but perform very slowly. As there is significant research being conducted in this area, new types of consensus mechanisms are also emerging, such as the semi-decentralized type, which is used in the Ripple network. The Ripple network will be discussed in detail in this book's online content pages, here: `https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf`. There are also various other proposals out there, which are trying to find the right balance between scalability and performance. Some notable projects include PBFT, Hybrid BFT, BlockDAG, Tezos, Stellar, and GHOST.

The consensus algorithms available today, or that are being researched in the context of blockchain, are presented as follows. 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. The Sybil attack will be discussed in *Chapter 6*, *Introducing Bitcoin*.
- **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. In this model, the chances of proposing and signing the next block increase with the coin age.

- **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 **SGX** (**Software Guard Extensions**) processor to provide the security guarantee for it to be secure. This concept is discussed in more detail in *Chapter 17*, *Hyperledger*, in the context of the Intel Sawtooth Lake blockchain project.
- **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. More information about this coin is available from NEM's website (`https://nem.io`).
- **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.
- **Practical Byzantine Fault Tolerance** (**PBFT**): This mechanism achieves SMR, 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**): This scheme is a combination of PoS and PoW, which ensures that a stakeholder is selected in a pseudorandom but uniform fashion. This is a comparatively more energy-efficient mechanism as compared to PoW. It utilizes a new concept called "Follow the Satoshi." In this scheme, PoW and PoS are combined together to achieve consensus and a 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. We will discuss these ideas further in *Chapter 7*, *Bitcoin Network and Payments*, where protocols are reviewed in the context of advanced Bitcoin protocols.
- **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 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**: 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.
- **Proof of Authority** (**PoA**): This scheme utilizes the identity of the participants called validators as a stake on the network. Validators are known and have the authority to propose new blocks. Validators propose the new blocks and validate them as per blockchain rules. Commonly used PoA algorithms are Clique and Aura.

Some prominent protocols in blockchain will be discussed in detail in *Chapter 5*, *Consensus Algorithms*. In this chapter, a light introduction is presented only.

Remember, earlier in this chapter we said that distributed systems are difficult to build and a distributed system cannot have consistency, availability, and partition tolerance at the same time. This is a proven result; however, in blockchain, it seems that this theorem is somehow violated. In the next section, we will introduce the CAP theorem formally and discuss why blockchain appears to achieve all three properties simultaneously.

2

Decentralization

Decentralization is not a new concept. It has been in use in strategy, management, and government for a long time. The basic idea of decentralization is to distribute control and authority to the peripheries of an organization instead of one central body being in full control of the organization. This configuration produces several benefits for organizations, such as increased efficiency, expedited decision making, better motivation, and a reduced burden on top management.

In this chapter, the concept of decentralization will be discussed in the context of blockchain. The fundamental basis of blockchain is that no single central authority is in control of the network. This chapter will present examples of various methods of decentralization and ways to achieve it. Furthermore, the decentralization of the blockchain ecosystem, decentralized applications, and platforms for achieving decentralization will be discussed in detail. Many exciting applications and ideas emerge from the decentralized blockchain technology, such as decentralized finance and decentralized identity, which will be introduced in this chapter.

Decentralization using blockchain

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. A consensus mechanism governs this competition, and the most famous 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, 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 to allow 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 the blockchain.

The following diagram shows the different types of systems that currently exist: central, distributed, and decentralized. This concept was first published by Paul Baran in *On Distributed Communications: I. Introduction to Distributed Communications Networks* (Rand Corporation, 1964):

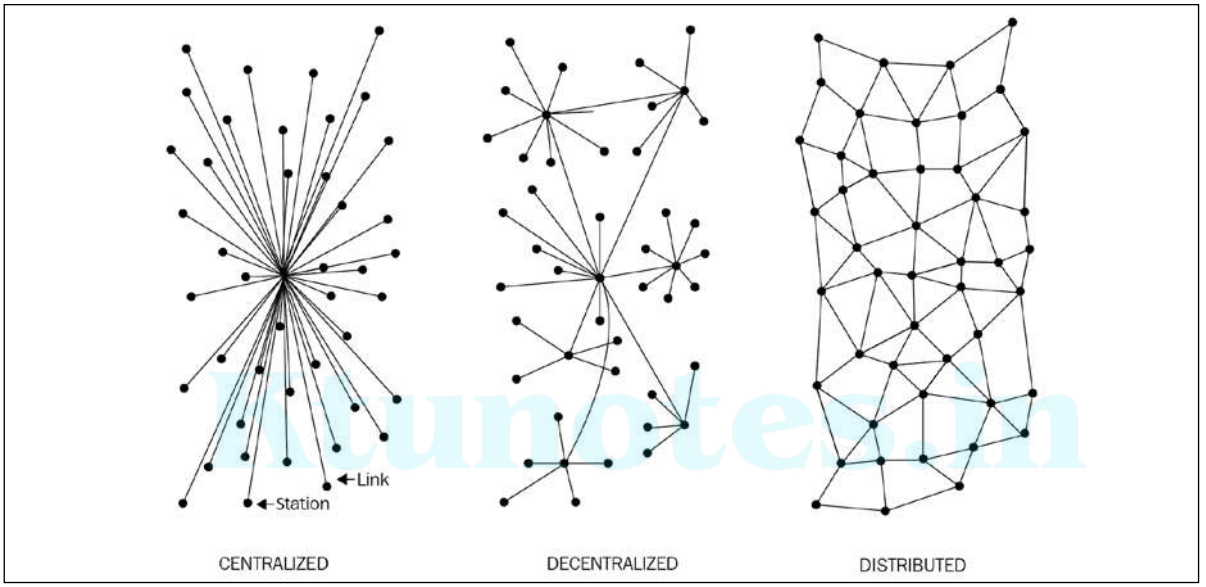


Figure 2.1: Different types of networks/systems

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, and Apple's App Store, use this conventional model to deliver services.

In 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 to achieve fault tolerance and speed. In the parallel system model, there is still a central authority that has control over all nodes and 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 is still 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.

We can also now view the different types of networks shown earlier from a different perspective, where we highlight the controlling authority of these networks as a symbolic hand, as shown in the following diagram. This model provides a clearer understanding of the differences between these networks from a decentralization point of view:

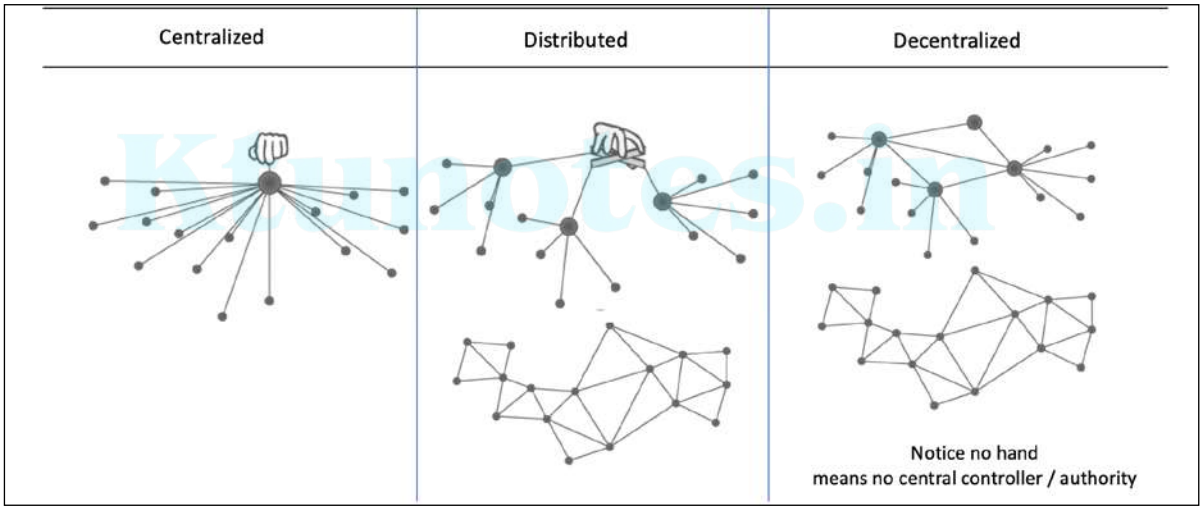


Figure 2.2: Different types of networks/systems depicting decentralization from a modern perspective

The preceding diagram shows that the centralized model is the traditional one in which a central controller exists, and it can be viewed as a depiction of the usual client/server model. In the middle we have distributed systems, where we still have a central controller but the system comprises many dispersed nodes. On the right-hand side, notice that there is no hand/controller controlling the networks.

This is the key difference between decentralized and distributed networks. A decentralized system may look like a distributed system from a topological point of view, but it doesn't have a central authority that controls the network.

The differences between distributed and decentralized systems can also be viewed at a practical level in the following diagrams:

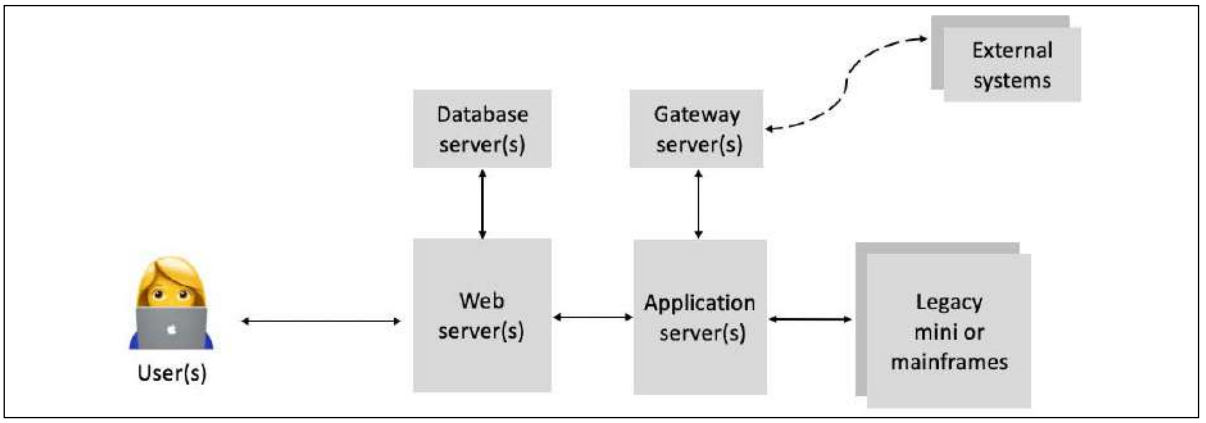


Figure 2.3: A traditional distributed system comprises many servers performing different roles

The following diagram shows a decentralized system (based on blockchain) where an exact replica of the applications and data is maintained across the entire network on each participating node:

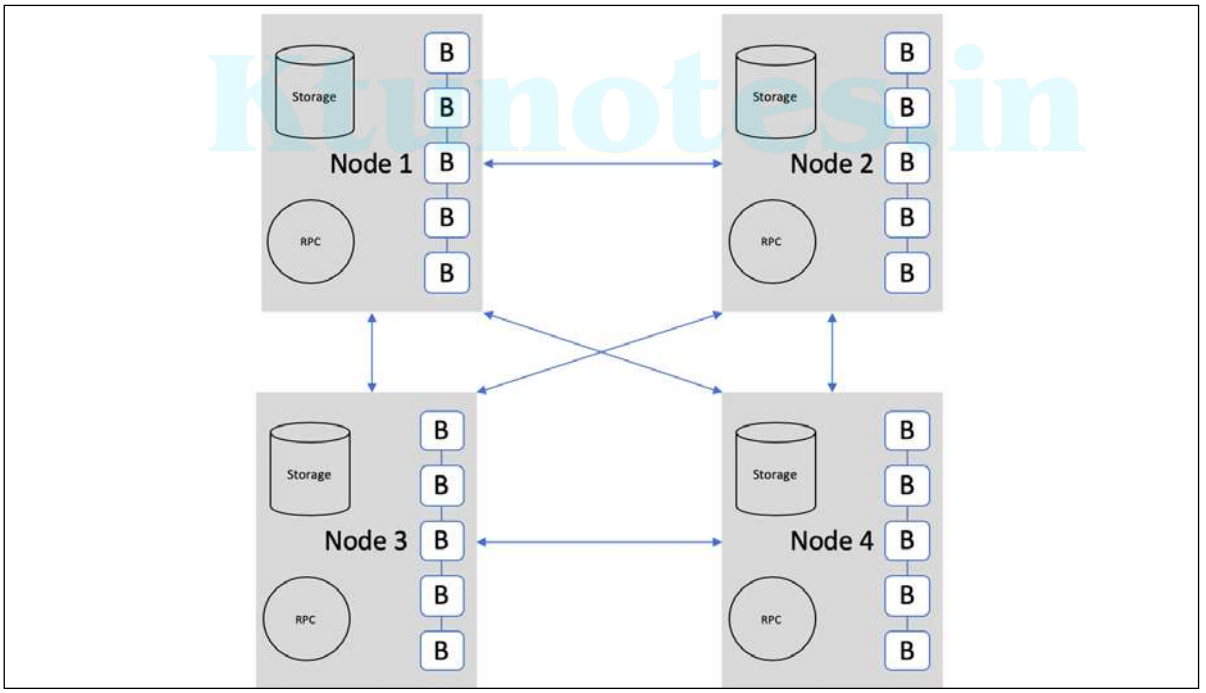


Figure 2.4: A blockchain-based decentralized system (notice the direct P2P connections and the exact replicas of blocks)

A comparison between centralized and decentralized systems (networks/applications) is shown in the following table:

Feature	Centralized	Decentralized
Ownership	Service provider	All users
Architecture	Client/server	Distributed, different topologies
Security	Basic	More secure
High availability	No	Yes
Fault tolerance	Basic, single point of failure	Highly tolerant, as service is replicated
Collusion resistance	Basic, because it's under the control of a group or even single individual	Highly resistant, as consensus algorithms ensure defense against adversaries
Application architecture	Single application	Application replicated across all nodes on the network
Trust	Consumers have to trust the service provider	No mutual trust required
Cost for consumer	Higher	Lower

The comparison in the table only covers some main features and is not an exhaustive list of all features. There may be other features of interest that can be compared too, but this list should provide a good level of comparison.

Now we will discuss what methods can be used to achieve decentralization.

Methods of decentralization

Two methods can be used to achieve decentralization: disintermediation and competition. 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, which, 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 the massive regulatory and compliance requirements. Nevertheless, this model can be used not only in finance but in many other industries as well, such as health, law, and the public sector. In the health industry, where patients, instead of relying on a trusted third party (such as the hospital record system) can be in full control of their own identity and their data that they can share directly with only those entities that they trust. As a general solution, blockchain can serve as a decentralized health record management system where health records can be exchanged securely and directly between different entities (hospitals, pharmaceutical companies, patients) globally without any central authority.

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.

In the following diagram, varying levels of decentralization are shown. On the left side, the conventional approach is shown where a central system is in control; on the right side, complete disintermediation is achieved, as intermediaries are entirely removed. Competing intermediaries or service providers are shown in the center. At that level, intermediaries or service providers are selected based on reputation or voting, thus achieving partial decentralization:

Figure 2.5: Scale of decentralization

There are many benefits of decentralization, including transparency, efficiency, cost saving, development of trusted ecosystems, and in some cases privacy and anonymity. Some challenges, such as security requirements, software bugs, and human error, need to be examined thoroughly.

For example, in a decentralized system such as Bitcoin or Ethereum where security is normally provided by private keys, how can we ensure that an asset or a token associated with these private keys cannot be rendered useless due to negligence or bugs in the code? What if the private keys are lost due to user negligence? What if due to a bug in the smart contract code the decentralized application becomes vulnerable to attack?

Before embarking on a journey to decentralize everything using blockchain and decentralized applications, it is essential that we understand that not everything can or needs to be decentralized.

This view raises some fundamental questions. Is a blockchain really needed? When is a blockchain required? In what circumstances is blockchain preferable to traditional databases? To answer these questions, go through the simple set of questions presented below:

Question	Yes/No	Recommended solution
Is high data throughput required?	Yes	Use a traditional database.
	No	A central database might still be useful if other requirements are met. For example, if users trust each other, then perhaps there is no need for a blockchain. However, if they don't or trust cannot be established for any reason, blockchain can be helpful.
Are updates centrally controlled?	Yes	Use a traditional database.
	No	You may investigate how a public/private blockchain can help.
Do users trust each other?	Yes	Use a traditional database.
	No	Use a public blockchain.
Are users anonymous?	Yes	Use a public blockchain.
	No	Use a private blockchain.
Is consensus required to be maintained within a consortium?	Yes	Use a private blockchain.
	No	Use a public blockchain.
Is strict data immutability required?	Yes	Use a blockchain.
	No	Use a central/traditional database.

Answering all of these questions can help you decide whether or not a blockchain is required or suitable for solving the problem. Beyond the questions posed in this model, there are many other issues to consider, such as latency, choice of consensus mechanisms, whether consensus is required or not, and where consensus is going to be achieved. If consensus is maintained internally by a consortium, then a private blockchain should be used; otherwise, if consensus is required publicly among multiple entities, then a public blockchain solution should be considered. Other aspects, such as immutability, should also be considered when deciding whether to use a blockchain or a traditional database. If strict data immutability is required, then a public blockchain should be used; otherwise, a central database may be an option.

As blockchain technology matures, there will be more questions raised regarding this selection model. For now, however, this set of questions is sufficient for deciding whether a blockchain-based solution is suitable or not.

Now we understand different methods of decentralization and have looked at how to decide whether a blockchain is required or not in a particular scenario. Let's now look at the process of decentralization, that is, how we can take an existing system and decentralize it. First, we'll briefly look at the different ways to achieve decentralization.

Routes to decentralization

There are systems that pre-date blockchain and Bitcoin, including BitTorrent and the Gnutella file-sharing system, which to a certain degree could be classified as decentralized, but due to a lack of any incentivization mechanism, participation from the community gradually decreased. There wasn't any incentive to keep the users interested in participating in the growth of the network. With the advent of blockchain technology, many initiatives are being taken to leverage this new technology to achieve decentralization. The Bitcoin blockchain is typically the first choice for many, as it has proven to be the most resilient and secure blockchain and has a market cap of nearly $166 billion at the time of writing. Alternatively, other blockchains, such as Ethereum, serve as the tool of choice for many developers for building decentralized applications. Compared to Bitcoin, Ethereum has become a more prominent choice because of the flexibility it allows for programming any business logic into the blockchain by using **smart contracts**.

How to decentralize

Arvind Narayanan and others have proposed a framework in their book *Bitcoin and Cryptocurrency Technologies* that can be used to evaluate the decentralization requirements of a variety of issues in the context of blockchain technology. The framework raises four questions whose answers provide a clear understanding of how a system can be decentralized:

1. What is being decentralized?
2. What level of decentralization is required?
3. What blockchain is used?
4. What security mechanism is used?

The first question simply asks you to identify what system is being decentralized. This can be any system, such as an identity system or a trading system.

The second question asks you to specify the level of decentralization required by examining the scale of decentralization, as discussed earlier. It can be full disintermediation or partial disintermediation.

The third question asks developers to determine which blockchain is suitable for a particular application. It can be Bitcoin blockchain, Ethereum blockchain, or any other blockchain that is deemed fit for the specific application.

Finally, a fundamental question that needs to be addressed is how the security of a decentralized system will be guaranteed. For example, the security mechanism can be atomicity-based, where either the transaction executes in full or does not execute at all. This deterministic approach ensures the integrity of the system. Other mechanisms may include one based on reputation, which allows for varying degrees of trust in a system.

In the following section, let's evaluate a money transfer system as an example of an application selected to be decentralized.

Decentralization framework example

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 do not execute at all. We have chosen the Bitcoin blockchain because it is the longest established blockchain and 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.

To achieve complete decentralization, it is necessary that the environment around the blockchain also be decentralized. We'll look at the full ecosystem of decentralization next.

Blockchain and full ecosystem decentralization

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.

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. **Inter-Planetary 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 Graphs** (**DAGs**) to provide storage and searching functionality, respectively. The concept of DHTs and DAGs will be introduced in detail in *Chapter 4*, *Public Key Cryptography*.

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. All of these projects are discussed later in this book in greater detail.

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

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 the 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 mesh network is Firechat, which allows iPhone users to communicate with each other directly in a peer-to-peer fashion without an Internet connection. More information is available at `https://www.opengarden.com/firechat/`.

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 toward 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 already provide managed services for end users, so there is a natural inclination toward selecting one of these large centralized services as they are more convenient and free to use. 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 revived the vision of decentralization across the world, and now concerted efforts are being made to harness this technology and take advantage of the benefits that it can provide.

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 an overview of a decentralized ecosystem. In the bottom layer, the Internet or mesh networks provide a decentralized communication layer. In the next layer up, a storage layer uses technologies such as IPFS and BigChainDB to enable decentralization. Finally, in the next level up, you can see that the 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 for storing large amounts of data in a decentralized way. The Identity and 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:

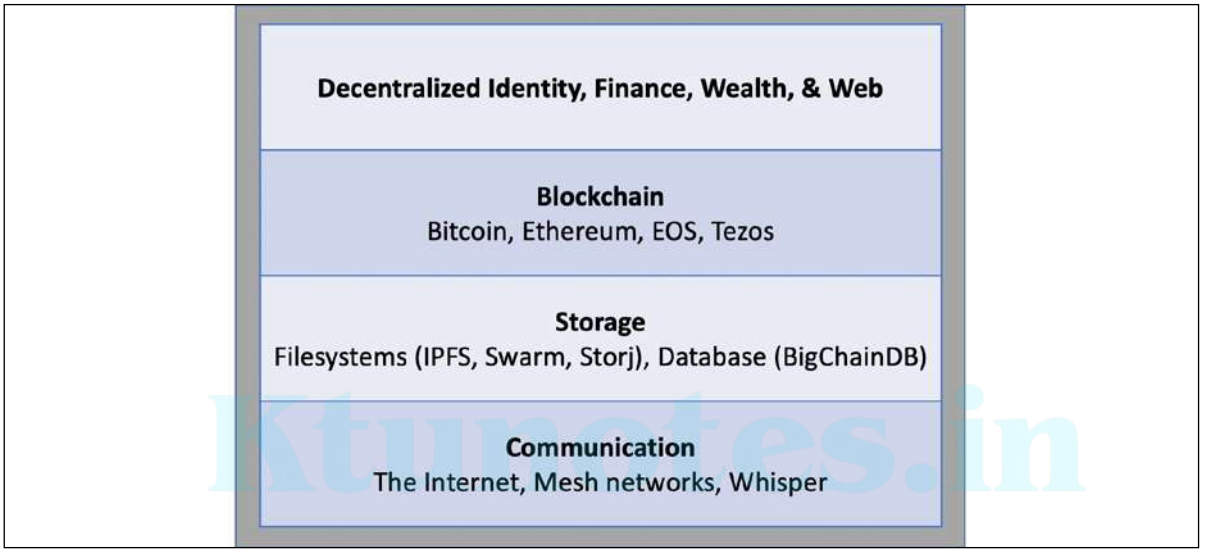


Figure 2.6: Decentralized ecosystem

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 is secure, decentralized, and 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. However, this is not a panacea, and it comes with many challenges, such as reliance on users to store and maintain private keys securely. This opens up other general questions about the suitability of decentralization to a particular problem.

Decentralization may not be appropriate for every scenario. Centralized systems with well-established reputations tend to work better in many cases. For example, email platforms from reputable companies such as Google or Microsoft would provide a better service than a scenario where individual email servers are hosted by users on the Internet.

There are many projects underway that are developing solutions for a more comprehensive distributed blockchain system. For example, Swarm and Whisper are developed to provide decentralized storage and communication for Ethereum. We will discuss Swarm and Ethereum in more detail in *Chapter 13*, *Ethereum Development Environment*.

With the advent of blockchain technology, it is now possible to build software versions of traditional physical organizations in the form of **Decentralized Organizations** (**DOs**) and other similar constructs, which we will examine in detail shortly.

Moreover, with the emergence of the decentralization paradigm, different terminology and buzzwords are now appearing in the media and academic literature, which we will explore in the next section.

Pertinent terminology

The following concepts are worth citing in the context of decentralization. The terminology introduced here is often used in the literature concerning decentralization and its applications.

Smart contracts

A **smart contract** is a software program that usually runs on a blockchain. 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.

More information on smart contracts will be provided in *Chapter 10*, *Smart Contracts*.

Autonomous agents

An **Autonomous Agent** (**AA**) is an artificially intelligent software entity that acts on the behalf of its owner to achieve some desirable goals without requiring any or minimal intervention from its owner.

Decentralized organizations

DOs 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.

Decentralized autonomous organizations

Just like DOs, a **decentralized autonomous organization** (**DAO**) is also a computer program that runs on top of a blockchain, and embedded within it are governance and business logic rules. DAOs and DOs 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.

Ethereum blockchain led the way with the 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).

The most famous DAO project is **The DAO**, which raised $168 million in its crowdfunding phase. The DAO project was designed to be a venture capital fund aimed at providing a decentralized business model with no single entity as owner. Unfortunately, this project was hacked due to a bug in the DAO code, and millions of dollars' worth of **ether** currency (**ETH**) was siphoned out of the project and into a child DAO created by hackers. A major network change (hard fork) was required on the Ethereum blockchain to reverse the impact of the hack and initiate the recovery of the funds. This incident opened up the debate on the security, quality, and need for thorough testing of the code in smart contracts in order to ensure their integrity and adequate control. There are other projects underway, especially in academia, that are seeking to formalize smart contract coding and testing.

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 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. Thus, they raise a big question as to how the current legal system could be applied to such a varied mix of jurisdictions and geographies.

Decentralized autonomous corporations

Decentralized autonomous corporations (**DACs**) are similar to DAOs in concept, though considered to be a subset of them. The definitions of DACs and DAOs may sometimes overlap, but the general distinction is that DAOs are usually considered to be nonprofit, whereas DACs can earn a profit via shares offered to the participants and to whom they can pay dividends. DACs can run a business automatically without human intervention based on the logic programmed into them.

Decentralized autonomous societies

Decentralized autonomous societies (**DASes**) are a concept whereby an entire society can function on a blockchain with the help of multiple, complex smart contracts and a combination of DAOs and **decentralized applications** (**DApps**) running autonomously. This model does not necessarily translate to a free-for-all approach, nor is it based on an entirely libertarian ideology; instead, many services that a government commonly offers can be delivered via blockchains, such as government identity card systems, passports, and records of deeds, marriages, and births. Another theory is that, if a government is corrupt and central systems do not provide the levels of trust that a society needs, then that society can start its own virtual one on a blockchain that is driven by decentralized consensus and transparency. This concept might look like a libertarian's or cypherpunk's dream, but it is entirely possible on a blockchain.

Decentralized applications

All the ideas mentioned up to this point come under the broader umbrella of decentralized applications, abbreviated to DApps. DAOs, DACs, and DOs are DApps that run on top of a blockchain in a peer-to-peer network. They represent the latest advancement in decentralization technology.

DApps at a fundamental level are software programs that execute using either of the following methods. They are categorized as Type 1, Type 2, or Type 3 DApps:

1. **Type 1:** Run on their own dedicated blockchain, for example, standard smart contract based DApps running on Ethereum. If required, they make use of a native token, for example, ETH on Ethereum blockchain.

For example, Ethlance is a DApp that makes use of ETH to provide a job market. More information about Ethlance can be found at `https://ethlance.com`.

2. **Type 2**: Use an existing established blockchain. that is, make use of Type 1 blockchain and bear custom protocols and tokens, for example, smart contract based tokenization DApps running Ethereum blockchain.
   An example is DAI, which is built on top of Ethereum blockchain, but contains its own stable coins and mechanism of distribution and control. Another example is Golem, which has its own token GNT and a transaction framework built on top of Ethereum blockchain to provide a **decentralized marketplace** for computing power where users share their computing power with each other in a peer-to-peer network.

A prime example of Type 2 DApps is the OMNI network, which is a software layer built on top of Bitcoin to support trading of custom digital assets and digital currencies. More information on the OMNI network can be found at `https://www.omnilayer.org`.

More information on the Golem network is available at `https://golem.network`.

More information on DAI is available at `https://makerdao.com/en/`.

3. **Type 3**: Use the protocols of Type 2 DApps; for example, the SAFE Network uses the OMNI network protocol.

More information on the SAFE Network can be found at `https://safenetwork.tech`.

Another example to understand the difference between different types of DApps is the USDT token (Tethers). The original USDT uses the OMNI layer (a Type 2 DApp) on top of the Bitcoin network. USDT is also available on Ethereum using ERC20 tokens. This example shows that a USDT can be considered a Type 3 DApp, where the OMNI layer protocol (a Type 2 DApp) is used, which is itself built on Bitcoin (a Type 1 DApp). Also, from an Ethereum point of view USDT can also be considered a Type 3 DApp in that it makes use of the Type 1 DApp Ethereum blockchain using the ERC 20 standard, which was built to operate on Ethereum.

More information can be found about Tether at `https://tether.to`.

In the last few years, the expression DApp has been increasingly used to refer to any end-to-end decentralized blockchain application, including a user interface (usually a web interface), smart contract(s), and the host blockchain. The clear distinction between different types of DApps is now not commonly referred to, but it does exist. Often, DApps are now considered just as apps (blockchain apps) running on a blockchain such as Ethereum, Tezos, NEO, or EOS without any particular reference to their type.

There are thousands of different DApps running on various platforms (blockchains) now. There are various categories of these DApps covering media, social, finance, games, insurance, and health. There are various decentralized platforms (or blockchains) running, such as Ethereum, EOS, NEO, Loom, and Steem. The highest number of DApps currently is on Ethereum.

Requirements of a DApp

For an application to be considered decentralized, it must meet the following criteria. This definition was provided in a whitepaper by Johnston et al. in 2015, *The General Theory of Decentralized Applications, DApps*:

1. 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.
2. 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.
3. A cryptographic token must be used by the application to provide access for and incentivize those who contribute value to the applications, for example, miners in Bitcoin.
4. The tokens (if applicable) must be generated by the decentralized application using consensus and an applicable cryptographic algorithm. This generation of tokens acts as a proof of the value to contributors (for example, miners).

Generally, DApps now provide all sorts of different services, including but not limited to financial applications, gaming, social media, and health.

Operations of a DApp

Establishment of consensus by a DApp can be achieved using consensus algorithms such as PoW and **Proof of Stake** (**PoS**). So far, only PoW has been found to be incredibly resistant to attacks, as is evident from the success of and trust people have put in the Bitcoin network. Furthermore, a DApp can distribute tokens (coins) via **mining**, **fundraising**, and **development**.

Design of a DApp

A DApp—pronounced Dee-App, or now more commonly rhyming with app—is a software application that runs on a decentralized network such as a distributed ledger. They have recently become very popular due to the development of various decentralized platforms such as Ethereum, EOS, and Tezos.

Traditional apps commonly consist of a user interface and usually a web server or an application server and a backend database. This is a common client/server architecture. This is visualized in the following diagram:

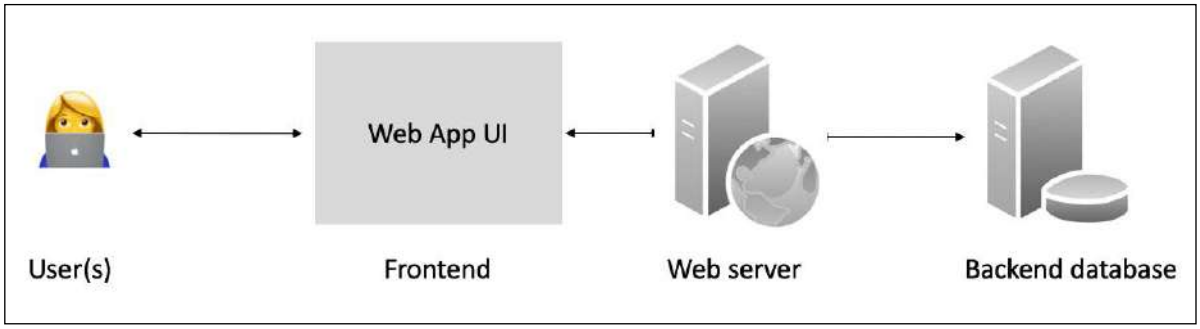


Figure 2.7: Traditional application architecture (generic client/server)

A DApp on the other hand has a blockchain as a backend and can be visualized as depicted in the following diagram. The key element that plays a vital role in the creation of a DApp is a smart contract that runs on the blockchain and has business logic embedded within it:

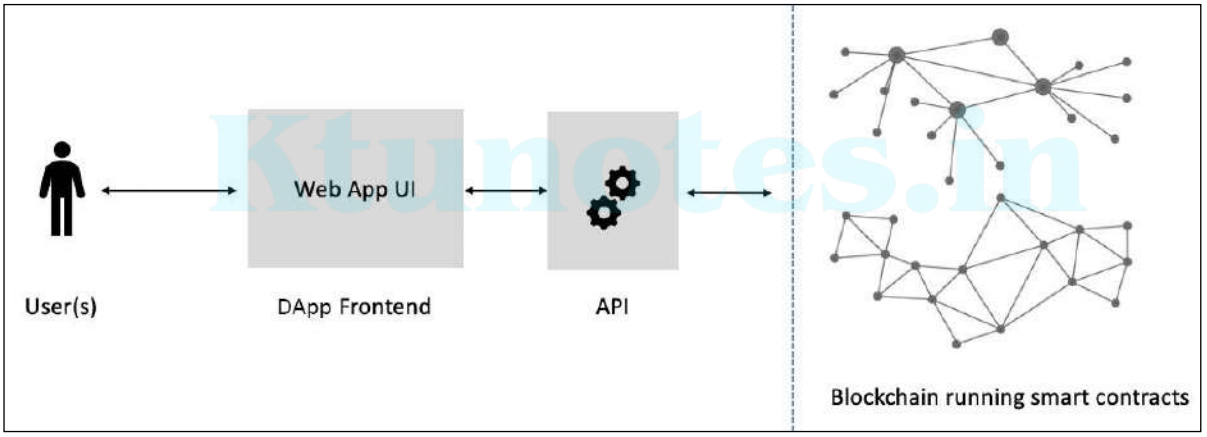


Figure 2.8: Generic DApp architecture

Note that the frontend in either a DApp or app architecture can either be a thick client, a mobile app, or a web frontend (a web user interface). However, it is usually a web frontend commonly written using a JavaScript framework such as React or Angular.

The following comparison table highlights the key properties of and differences between these different types of decentralized entities:

Entity	Autonomous?	Software?	Owned?	Capital?	Legal status	Cost
DO	No	No	Yes	Yes	Yes	High
DAO	Yes	Yes	No	Yes	Unsettled	Low
DAC	Yes	Yes	Yes	Yes	Unsettled	Low
DAS	Yes	Yes	No	Possible	Unsettled	Low
DApp	Yes	Yes	Yes	Optional tokens	Unsettled	Use case dependent

Having covered the main concepts of DApps, it will be useful to explore some specific examples.

DApp examples

Examples of some DApps are provided here.

KYC-Chain

This application provides the facility to manage **Know Your Customer** (**KYC**) data securely and conveniently based on smart contracts.

OpenBazaar

This is a decentralized peer-to-peer network that enables commercial activities directly between sellers and buyers instead of relying on a central party, such as eBay or Amazon. It should be noted that this system is not built on top of a blockchain; instead, distributed hash tables are used in a peer-to-peer network to enable direct communication and data sharing among peers. It makes use of Bitcoin and various other cryptocurrencies as a payment method.

More information regarding Open Bazaar is available at `https://openbazaar.org`.

Lazooz

This is the decentralized equivalent of Uber. It allows peer-to-peer ride sharing and users to be incentivized by proof of movement, and they can earn Zooz coins.

More information on Lazooz is available at `http://lazooz.org`.

Many other DApps have been built on the Ethereum blockchain and are showcased at `http://dapps.ethercasts.com/`.

Now that we have covered the pertinent terminology, DApps, and relevant examples, let's now look at what platforms can be used to build and host DApps.

5
Consensus Algorithms

Consensus is a fundamental problem in distributed systems. Since the 1970s this problem has been researched in the context of distributed systems, but recently, with the advent of blockchain technology, a renewed interest has arisen in developing distributed consensus algorithms that are suitable for blockchain networks. In this chapter, we will explore the underlying techniques behind distributed consensus algorithms, their inner workings, and new algorithms that have been specifically developed for blockchain networks.

In addition, we will introduce various well-known algorithms in a traditional distributed systems arena that can also be implemented in blockchain networks with some modifications, such as Paxos, Raft, and PBFT. We will also explore other mechanisms that have been introduced specifically for blockchain networks such as **Proof of Work** (**PoW**), **Proof of Stake** (**PoS**), and modified versions of traditional consensus such as **Istanbul Byzantine Fault Tolerant** (**IBFT**), which is a modified, **blockchained** version of the **Practical Byzantine Fault Tolerant** (**PBFT**) algorithm, suitable for a blockchain network. Along the way, we'll cover the following topics:

- Introducing the consensus problem
- Analysis and design
- Classification
- Algorithms
- Choosing an algorithm

Before we delve into specific algorithms, we first need to understand some fundamental concepts and an overview of the consensus problem.

Introducing the consensus problem

The distributed consensus problem has been studied extensively in distributed systems research since the late 1970s. Distributed systems are classified into two main categories, namely **message passing** and **shared memory**. In the context of blockchain, we are concerned with the message passing type of distributed systems, where participants on the network communicate with each other via passing messages to each other.

Blockchain is a distributed system that relies upon a consensus mechanism, which ensures the safety and liveness of the blockchain network.

In the past decade, the rapid evolution of blockchain technology has been observed. Also, with this tremendous growth, research regarding distributed consensus has grown significantly. Researchers from industry and academia are especially interested in researching novel methods of consensus. A common research area is to convert traditional (classical) distributed consensus mechanisms into their blockchain variants that are suitable for blockchain networks. Another area of interest is to analyze existing and new consensus protocols.

As we saw in *Chapter 1*, *Blockchain 101*, there are different types of blockchain networks. In particular, two types, permissioned and public (non-permissioned) were discussed. The consensus is also classified based on these two paradigms. For example, Bitcoin is a public blockchain. It runs PoW, sometimes called **Nakamoto consensus**.

In contrast, many permissioned blockchains tend to run variants of traditional or classical distributed consensus. A prime example is IBFT, which is a **blockchained** version of PBFT. Other examples include Tendermint, Casper FFG, and many variants of PBFT. We will discuss more on that later in this chapter.

First, we will look at traditional consensus mechanisms, which are also known as fault-tolerant distributed consensus, classical distributed consensus, or pre-Bitcoin distributed consensus. Distributed consensus is a highly researched problem and many fundamental ideas to describe and elaborate on the problem have been developed. One of them, and arguably the most famous, is the Byzantine generals problem, which we'll describe next. In addition to the Byzantine generals problem, we will look at relevant and important impossibility results, which will help to build a general understanding of the consensus and relevant limitations. An understanding of these concepts is vital to understand what problem exactly is being solved and how.

The Byzantine generals problem

The problem of reaching agreement in the presence of faults or Byzantine consensus was first formulated by M. Pease, R. Shostak, and L. Lamport. In distributed systems, a common goal is to achieve consensus (agreement) among nodes on the network even in the presence of faults. In order to explain the problem, Lamport came up with an allegorical representation of the problem and named it the **Byzantine generals problem**.

The Byzantine generals problem metaphorically depicts a situation where a Byzantine army, divided into different units, is spread around a city. A general commands each unit, and they can only communicate with each other using a messenger. To be successful, the generals must coordinate their plan and decide whether to attack or retreat. The problem, however, is that any generals could potentially be disloyal and act maliciously to obstruct agreement upon a united plan. The requirement now becomes that every honest general must somehow agree on the same decision even in the presence of treacherous generals.

In order to address this issue, honest (loyal) generals must reach a majority agreement on their plan.

Leslie Lamport introduced numerous fundamental ideas and techniques for distributed consensus.

The famous Byzantine generals problem was formulated by Lamport et al. in their paper: Lamport, L., Shostak, R. and Pease, M., 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), pp.382-401.

The paper is available here: `https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/`.

In the digital world, generals are represented by computers (nodes) and communication links are messengers carrying messages. Disloyal generals are faulty nodes. Later in this chapter, we'll see how agreement can be achieved using consensus algorithms even in the presence of faults.

Fault tolerance

A fundamental requirement in a consensus mechanism is that it must be fault-tolerant. In other words, it must be able to tolerate a number of failures in a network and should continue to work even in the presence of faults. This naturally means that there has to be some limit to the number of faults a network can handle, since no network can operate correctly if a large majority of its nodes are failing. Based on the requirement of fault tolerance, consensus algorithms are also called fault-tolerant algorithms, and there are two types of fault-tolerant algorithms.

Types of fault-tolerant consensus

Fault-tolerant algorithms can be divided into two types of fault-tolerance. The first is **Crash fault-tolerance** (**CFT**) and the other is **Byzantine fault-tolerance** (**BFT**). CFT covers only crash faults or, in other words, benign faults. In contrast, BFT deals with the type of faults that are arbitrary and can even be malicious.

Replication is a standard approach to make a system fault-tolerant. Replication results in a synchronized copy of data across all nodes in a network. This technique improves the fault tolerance and availability of the network. This means that even if some of the nodes become faulty, the overall system/network remains available due to the data being available on multiple nodes.

There are two main types of replication techniques:

- **Active replication**, which is a type where each replica becomes a copy of the original state machine replica.
- **Passive replication**, which is a type where there is only a single copy of the state machine in the system kept by the primary node, and the rest of the nodes/replicas only maintain the state.

In this section, we've briefly looked at what replication is and its types. In the context of fault-tolerant consensus mechanisms, replication plays a vital role by introducing resiliency into the system. We'll now introduce another relevant concept, known as **state machine replication**, which is a standard technique used to achieve fault tolerance in distributed systems.

State machine replication

State machine replication (**SMR**) is a de facto technique that is used to provide deterministic replication services in order to achieve fault tolerance in a distributed system. State machine replication was first proposed by Lamport in 1978 in his paper:

Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), pp.558-565.

The paper is available here:

`https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/Time-Clocks-and-the-Ordering-of-Events-in-a-Distributed-System.pdf`

Later, in 1990, Schneider formalized the state machine replication approach and published the results in a paper titled:

Schneider, F.B., 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4), pp.299-319.

The paper is available here:

`https://www.cs.cornell.edu/fbs/publications/SMSurvey.pdf`

Now, just before we discuss SMR further, let's understand what a state machine is. At an abstract level, it is a mathematical model that is used to describe a machine that can be in different states. It is important to understand that a state machine can only have one state at a time. A state machine stores a state of the system and transitions it to the next state as a result of input received. As a result of state transition, an output is produced along with an updated state.

The fundamental idea behind SMR can be summarized as follows:

1. All servers always start with the same initial state.
2. All servers receive requests in a **totally ordered** fashion (sequenced as generated from clients).
3. All servers produce the same deterministic output for the same input.

State machine replication is implemented under a primary/backup paradigm, where a primary node is responsible for receiving and broadcasting client requests. This broadcast mechanism is called **total order broadcast** or **atomic broadcast**, which ensures that backup or replica nodes receive and execute the same requests in the same sequence as the primary.

Consequently, this means that all replicas will eventually have the same state as the primary, thus resulting in achieving consensus. In other words, this means that total order broadcast and distributed consensus are equivalent problems; if you solve one, the other is solved too.

Now that we understand the basics of replication and fault tolerance, it is important to understand that fault tolerance works up to a certain threshold. For example, if a network has a vast majority of constantly failing nodes and communication links, it is not hard to understand that this type of network may not be as fault-tolerant as we might like it to be. In other words, even in the presence of fault-tolerant measures, if there is a lack of resources on a network, the network may still not be able to provide the required level of fault tolerance. In some scenarios, it might be impossible to provide the required services due to a lack of resources in a system. In distributed computing, such impossible scenarios are researched and reported as impossibility results.

In distributed computing, impossibility results provide an understanding of whether a problem is solvable and the minimum resources required to do so. If the problem is unsolvable, then these results give a clear understanding that a specific task cannot be accomplished and no further research is necessary. From another angle, we can say that impossibility results (sometimes called unsolvability results) show that certain problems are not computable under insufficient resources. Impossibility results unfold deep aspects of distributed computing and enable us to understand why certain problems are difficult to solve and under what conditions a previously unsolved problem might be solved.

The requirement of minimum available resources is known as **lower bound results**. The problems that are not solvable under any conditions are known as **unsolvability results**. For example, it has been proven that asynchronous deterministic consensus is impossible. This result is known as the FLP impossibility result, which we'll introduce next.

FLP impossibility

FLP impossibility is a fundamental unsolvability result in distributed computing theory that states that in an asynchronous environment, the deterministic consensus is impossible, even if only one process is faulty.

FLP is named after the authors' names, Fischer, Lynch, and Patterson, who in 1985 introduced this result. This result was presented in their paper:

Fischer, M.J., Lynch, N.A. and Paterson, M.S., 1982. Impossibility of distributed consensus with one faulty process (No. MIT/LCS/TR-282). *Massachusetts Inst of Tech Cambridge lab for Computer Science*.

The paper is available at:

`https://apps.dtic.mil/dtic/tr/fulltext/u2/a132503.pdf`

To circumvent **FLP impossibility**, several techniques have been introduced in the literature. These techniques include:

- **Failure detectors**, which can be seen as **oracles** associated with processors to detect failures.
- **Randomized algorithms** have been introduced to provide a probabilistic termination guarantee. The core idea behind the randomized protocols is that the processors in such protocols can make a random choice of decision value if the processor does not receive the required quorum of trusted messages.
- **Synchrony assumptions**, where additional synchrony and timing assumptions are made to ensure that the consensus algorithm terminates and makes progress.

Now that we understand a fundamental impossibility result, let's look at another relevant result that highlights the unsolvability of consensus due to a lack of resources: that is, a lower bound result. We can think of lower bound as a minimum amount of resources, for example, the number of processors or communication links required to solve a problem. In other words, if a minimum required number of resources is not available in a system, then the problem cannot be solved. In the context of a consensus problem, a fundamental proven result is *lower bounds on the number of processors*, which we describe next.

Lower bounds on the number of processors to solve consensus

As we described previously, there are proven results in distributed computing that state several lower bounds, for example, the minimum number of processors required for consensus or the minimum number of rounds required to achieve consensus. The most common and fundamental of these results is the minimum number of processors required for consensus. These results are listed below:

- In the case of CFT, at least $2F + 1$ number of nodes is required to achieve consensus.
- In the case of BFT, at least $3F + 1$ number of nodes is required to achieve consensus.

F represents the number of failures.

These lower bounds are discussed in several papers with relevant proofs. The most fundamental one is by Lamport et. al:

Pease, M., Shostak, R. and Lamport, L., 1980. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2), pp.228-234.

The paper is available here:

`https://lamport.azurewebsites.net/pubs/reaching.pdf`

Another paper that provides a number of impossibility proofs is:

Fischer, M.J., Lynch, N.A. and Merritt, M., 1986. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1), pp.26-39.

The paper is available here:

`https://apps.dtic.mil/dtic/tr/fulltext/u2/a157411.pdf`

Liveness: This requirement generally means that something good eventually happens.

- **Termination**. This liveness property states that each honest node must eventually decide on a value.

With this, we have covered the classification and requirements of consensus algorithms. In the next section, we'll introduce various consensus algorithms, which we can evaluate using the consensus models covered in this section.

Algorithms

In this section, we will discuss the key algorithms in detail. We'll be looking at the two main types of fault-tolerant algorithms, CFT and BFT.

CFT algorithms

We'll begin by looking at some algorithms that solve the consensus problem with crash fault tolerance. One of the most fundamental algorithms in this space is Paxos.

Paxos

Leslie Lamport developed Paxos. It is the most fundamental distributed consensus algorithm, allowing consensus over a value under unreliable communications. In other words, Paxos is used to build a reliable system that works correctly, even in the presence of faults.

Paxos was proposed first in 1989 and then later, more formally, in 1998 in the following paper:

Lamport, L., 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), pp.133-169.

The paper is available here:

`https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf.`

Note that Paxos works under an asynchronous network model and supports the handling of only benign failures. This is not a Byzantine fault-tolerant protocol. However, later, a variant of Paxos was developed that provides Byzantine fault tolerance. The original paper in which a Byzantine fault-tolerant version of Paxos is described is available here:

Lamport, L., 2011, September. Byzantizing Paxos by refinement. *International Symposium on Distributed Computing* (pp. 211-224). Springer, Berlin, Heidelberg.

A link to the paper is available here:

`http://lamport.azurewebsites.net/pubs/web-byzpaxos.pdf`

Paxos makes use of *2F + 1* processes to ensure fault tolerance in a network where processes can crash fault, that is, experience benign failures. Benign failure means either the loss of a message or a process stops. In other words, Paxos can tolerate one crash failure in a three-node network.

Paxos is a two-phase protocol. The first phase is called the *prepare* phase, and the next phase is called the *accept* phase. Paxos has proposer and acceptors as participants, where the proposer is the replicas or nodes that propose the values and acceptors are the nodes that accept the value.

How Paxos works

The Paxos protocol assumes an asynchronous message-passing network with less than 50% of crash faults. As usual, the critical properties of the Paxos consensus algorithm are safety and liveness. Under safety, we have:

- **Agreement**, which specifies that no two different values are agreed on. In other words, no two different learners learn different values.
- **Validity**, which means that only the proposed values are decided. In other words, the values chosen or learned must have been proposed by a processor.

Under liveness, we have:

- **Termination**, which means that, eventually, the protocol is able to decide and terminate. In other words, if a value has been chosen, then eventually learners will learn it.

Processes can assume different roles, which are listed as follows:

- **Proposers**, elected leader(s) that can propose a new value to be decided.
- **Acceptors**, which participate in the protocol as a means to provide a majority decision.
- **Learners**, which are nodes that just observe the decision process and value.

A single process in a Paxos network can assume all three roles.

The key idea behind Paxos is that the proposer node proposes a value, which is considered final only if a majority of the acceptor nodes accept it. The learner nodes also learn this final decision.

Paxos can be seen as a protocol that is quite similar to the two-phase commit protocol. **Two-phase commit** (**2PC**) is a standard atomic commitment protocol to ensure that transactions are committed in distributed databases only if all participants agree to commit. Even if a single node cannot agree to commit the transaction, it is fully rolled back.

Similarly, in Paxos, in the first phase, the proposer sends a proposal to the acceptors, if and when they accept the proposal, the proposer broadcasts a request to commit to the acceptors. Once the acceptors commit and report back to the proposer, the proposal is considered final, and the protocol concludes. In contrast with the two-phase commit, Paxos introduced ordering (sequencing to achieve total order) of the proposals and majority-based acceptance of the proposals instead of expecting all nodes to agree (to allow progress even if some nodes fail). Both of these improvements contribute toward ensuring the safety and liveness of the Paxos algorithm.

An excellent explanation of the two-phase commit is available here:

`https://en.wikipedia.org/wiki/Two-phase_commit_protocol`

We'll now describe how the Paxos protocol works step by step:

1. The proposer proposes a value by broadcasting a message, <*prepare(n)*>, to all acceptors.
2. Acceptors respond with an acknowledgment message if proposal *n* is the highest that the acceptor has responded to so far. The acknowledgment message <*ack(n, v, s)*> consists of three variables where *n* is the proposal number, *v* is the proposal value of the highest numbered proposal the acceptor has accepted so far, and *s* is the sequence number of the highest proposal accepted by the acceptor so far. This is where acceptors agree to commit the proposed value. The proposer now waits to receive acknowledgment messages from the majority of the acceptors indicating the **chosen** value.
3. If the majority is received, the proposer sends out the "accept" message <*accept(n, v)*> to the acceptors.
4. If the majority of the acceptors accept the proposed value (now the "accept" message), then it is decided: that is, agreement is achieved.
5. Finally, in the learning phase, acceptors broadcast the "accepted" message <*accepted(n, v)*> to the proposer. This phase is necessary to disseminate which proposal has been finally accepted. The proposer then informs all other learners of the decided value. Alternatively, learners can learn the decided value via a message that contains the accepted value (decision value) multicast by acceptors.

We can visualize this process in the following diagram:

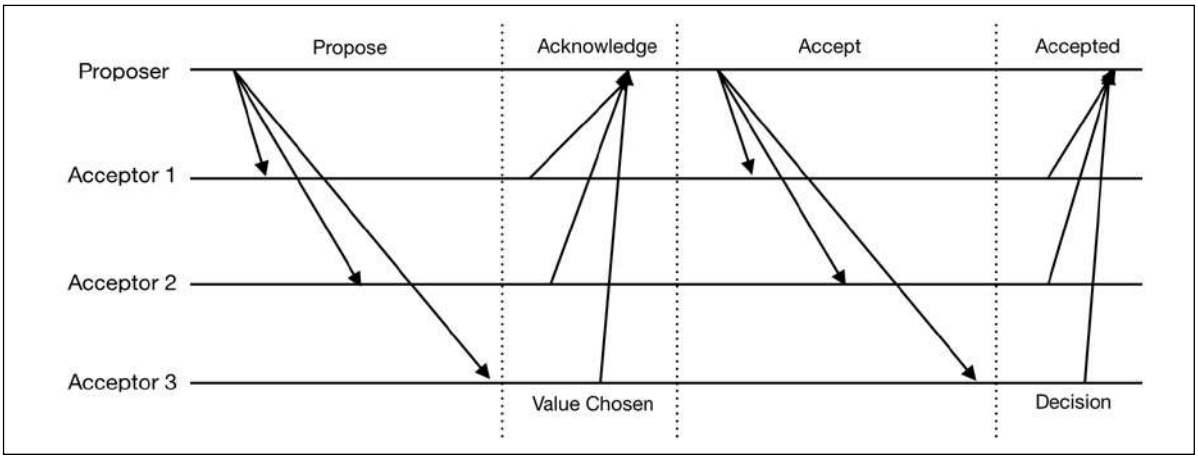


Figure 5.1: How Paxos works

Let's move on to see how Paxos achieves the much-desired properties of safety and liveness.

How Paxos achieves safety and liveness

A natural question arises about how Paxos ensures its safety and liveness guarantees. Paxos, at its core, is quite simple, yet it achieves all these properties efficiently. The actual proofs for the correctness of Paxos are quite in-depth and are not the subject of this chapter. However, the intuition behind each property is presented as follows:

- **Agreement** is ensured by enforcing that only one proposal can win votes from a majority of the acceptors.
- **Validity** is ensured by enforcing that only the genuine proposals are decided. In other words, no value is committed unless it is proposed in the proposal message first.
- **Liveness** or termination is guaranteed by ensuring that at some point during the protocol execution, eventually there is a period during which there is only one fault-free proposer.

For detailed proofs and correctness analyses, refer to the following papers:

Lamport, L., 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), pp.133-169.

`https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf`

Lamport, L., 2006. Fast Paxos. *Distributed Computing*, 19(2), pp.79-103.

`https://www.microsoft.com/en-us/research/publication/fast-paxos/`

In summary, the key points to remember about Paxos are that:

- First, a proposer suggests a value with the aim that acceptors achieve agreement on it.
- The decided value is a result of majority consensus among the acceptors and is finally learned by the learners.

There are many other protocols that have emerged from basic Paxos, such as Multi-Paxos, Fast Paxos, and Cheap Paxos.

Even though the Paxos algorithm is quite simple at its core, it is seen as challenging to understand, and many academic papers have been written to explain it. This slight problem, however, has not prevented it being implemented in many production networks, such as Google's Spanner, as it has proven to be the most efficient protocol to solve the consensus problem.

Nevertheless, there have been attempts to create alternative easy-to-understand algorithms. Raft is such an attempt to create an easy-to-understand CFT algorithm.

Raft

The Raft protocol is a CFT consensus mechanism developed by Diego Ongaro and John Ousterhout at Stanford University. In Raft, the leader is always assumed to be honest.

At a conceptual level, it is a replicated log for a **replicated state machine** (**RSM**) where a unique leader is elected every "term" (time division) whose log is replicated to all follower nodes.

Raft is composed of three sub-problems:

- **Leader election** (a new leader election in case the existing one fails)
- **Log replication** (leader to follower log synch)
- **Safety** (no conflicting log entries (index) between servers)

The Raft protocol ensures election safety, leader append only, log matching, leader completeness, and state machine safety.

Each server in Raft can have either a **follower**, **leader**, or **candidate** state.

The protocol ensures election **safety** (that is, only one winner each election term) and **liveness** (that is, some candidate must eventually win).

How Raft works

The following steps will describe how the Raft protocol functions. At a fundamental level, the protocol is quite simple and can be described simply by the following sequence:

Node starts up –> Leader election –> Log replication

1. First, the node starts up.
2. After this, the leader election process starts. Once a node is elected as leader, all changes go through that leader.
3. Each change is entered into the node's log.
4. Log entry remains uncommitted until the entry is replicated to follower nodes and the leader receives write confirmation votes from a majority of the nodes, then it is committed locally.
5. The leader notifies the followers regarding the committed entry.
6. Once this process ends, agreement is achieved.

The state transition of the Raft algorithm can be visualized in the following diagram:

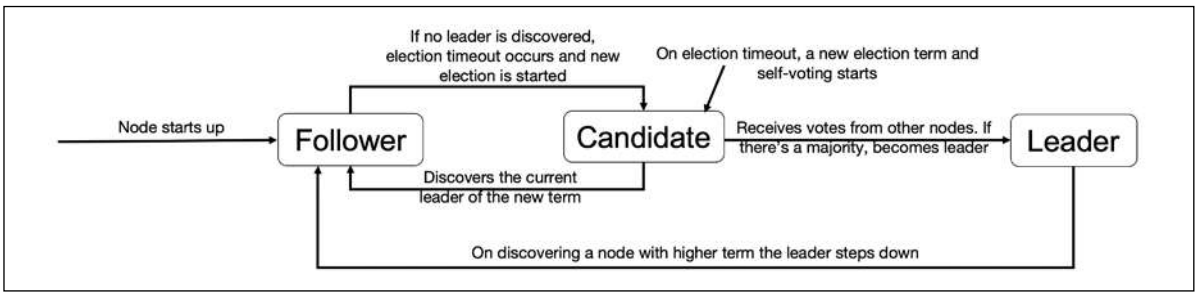


Figure 5.2: Raft state transition

We saw earlier that data is eventually replicated across all nodes in a consensus mechanism. In Raft, the log (data) is eventually replicated across all nodes. We describe this process of log replication next.

Log replication

Log replication logic can be visualized in the following diagram. The aim of log replication is to synchronize nodes with each other.

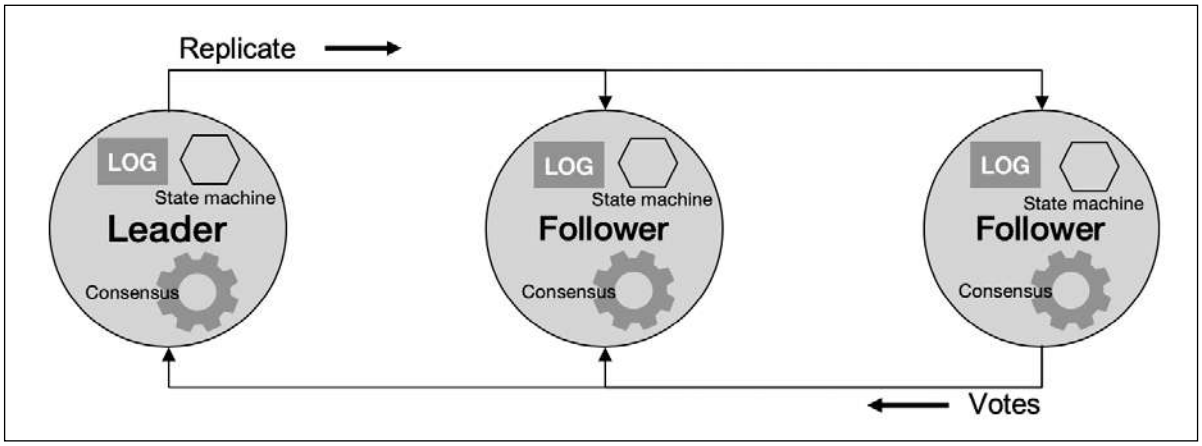


Figure 5.3: Log replication mechanism

Log replication is a simple mechanism. As shown in the preceding diagram, the leader is responsible for log replication. Once the leader has a new entry in its log, it sends out the requests to replicate to the follower nodes. When the leader receives enough confirmation votes back from the follower nodes indicating that the replicate request has been accepted and processed by the followers, the leader commits that entry to its local state machine. At this stage, the entry is considered committed.

With this, our discussion on CFT algorithms is complete. Now we'll introduce Byzantine fault-tolerant algorithms, which have been a research area for many years in distributed computing.

BFT algorithms

We described the formulation of the Byzantine generals problem at the start of this chapter. In this section, we'll introduce the mechanisms that were developed to solve the Byzantine generals (consensus in the presence of faults) problem.

Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (**PBFT**) was developed in 1999 by Miguel Castro and Barbara Liskov. PBFT, as the name suggests, is a protocol developed to provide consensus in the presence of Byzantine faults. Before PBFT, Byzantine fault tolerance was considered impractical. With PBFT, it was demonstrated for the first time that practical Byzantine fault tolerance is possible.

PBFT comprises three sub-protocols called **normal operation**, **view change**, and **checkpointing**.

Normal operation sub-protocol refers to a scheme that is executed when everything is running normally and no errors are in the system. View change is a sub-protocol that runs when a faulty leader node is detected in the system. Checkpointing is another sub-protocol, which is used to discard the old data from the system.

The PBFT protocol comprises three phases or steps. These phases run in a sequence to achieve consensus. These phases are pre-prepare, prepare, and commit, which we will cover in detail shortly.

The protocol runs in rounds where, in each round, an elected leader node, called the primary node, handles the communication with the client. In each round, the protocol progresses through the three previously mentioned phases. The participants in the PBFT protocol are called replicas, where one of the replicas becomes primary as a leader in each round, and the rest of the nodes acts as backups. PBFT is based on the SMR protocol introduced earlier. Here, each node maintains a local log, and the logs are kept in sync with each other via the consensus protocol: that is, PBFT.

As we saw earlier, in order to tolerate Byzantine faults, the minimum number of nodes required is $N = 3F + 1$, where N is the number of nodes and F is the number of faulty nodes. PBFT ensures Byzantine fault tolerance as long as the number of nodes in a system stays $N >= 3F + 1$.

We will now look at how the PBFT protocol works.

In summary, when a client sends a request to a primary, the protocol initiates a sequence of operations between replicas, which eventually leads to consensus and a reply back to the client. These sequences of operations are divided into different phases:

- Pre-prepare
- Prepare
- Commit

In addition, each replica maintains a local state comprising three main elements:

- Service state
- A message log
- A number representing that replica's current view

Now we'll discuss the phases mentioned above one by one, starting with pre-prepare.

Pre-prepare:

This is the first phase in the protocol, where the primary node, or **primary**, receives a request from the client. The primary node assigns a sequence number to the request. It then sends the pre-prepare message with the request to all backup replicas.

When the pre-prepare message is received by the backup replicas, it checks a number of things to ensure the validity of the message:

- First, whether the digital signature is valid.
- After this, whether the current view number is valid.
- Then, that the sequence number of the operation's request message is valid.

- Finally, if the digest/hash of the operation's request message is valid.

If all of these elements are valid, then the backup replica accepts the message. After accepting the message, it updates its local state and progresses toward the prepare phase.

Prepare:

A prepare message is sent by each backup to all other replicas in the system. Each backup waits for at least *2F + 1* prepare messages to be received from other replicas. They also check whether the prepare message contains the same view number, sequence number, and message digest values. If all these checks pass, then the replica updates its local state and progresses toward the commit phase.

Commit:

In the commit phase, each replica sends a commit message to all other replicas in the network. The same as the prepare phase, replicas wait for *2F + 1* commit messages to arrive from other replicas. The replicas also check the view number, sequence number, and message digest values. If they are valid for *2F+ 1* commit messages received from other replicas, then the replica executes the request, produces a result, and finally, updates its state to reflect a commit. If there are already some messages queued up, the replica will execute those requests first before processing the latest sequence numbers. Finally, the replica sends the result to the client in a reply message.

The client accepts the result only after receiving *2F+ 1* reply messages containing the same result.

Now, let's move on and look at how PBFT works in some more detail.

How PBFT works

At a high level, the protocol works as follows:

1. A client sends a request to invoke a service operation in the primary.
2. The primary multicasts the request to the backups.
3. Replicas execute the request and send a reply to the client.
4. The client waits for replies from different replicas with the same result; this is the result of the operation.

Now we'll describe each phase of the protocol (**pre-prepare**, **prepare**, and **commit**) in more formal terms.

The pre-prepare sub-protocol algorithm:

1. Accepts a request from the client.
2. Assigns the next sequence number.
3. Sends the pre-prepare message to all backup replicas.

The prepare sub-protocol algorithm:

1. Accepts the pre-prepare message. If the backup has not accepted any pre-prepare messages for the same view or sequence number, then it accepts the message.
2. Sends the prepare message to all replicas.

The commit sub-protocol algorithm:

1. The replica waits for *2F* prepare messages with the same view, sequence, and request.
2. Sends a commit message to all replicas.
3. Waits until a *2F + 1* valid commit message arrives and is accepted.
4. Executes the received request.
5. Sends a reply containing the execution result to the client.

In summary, the primary purpose of these phases is to achieve consensus, where each phase is responsible for a critical part of the consensus mechanism, which after passing through all phases, eventually ends up achieving consensus.

One key point to remember about each phase is listed as follows:

Pre-prepare: This phase assigns a unique sequence number to the request. We can think of it as an orderer.

Prepare: This phase ensures that honest replicas/nodes in the network agree on the total order of requests within a view. Note that the pre-prepare and prepare phases together provide the total order to the messages.

Commit: This phase ensures that honest replicas/nodes in the network agree on the total order of requests across views.

The normal view of the PBFT protocol can be visualized as shown as follows:

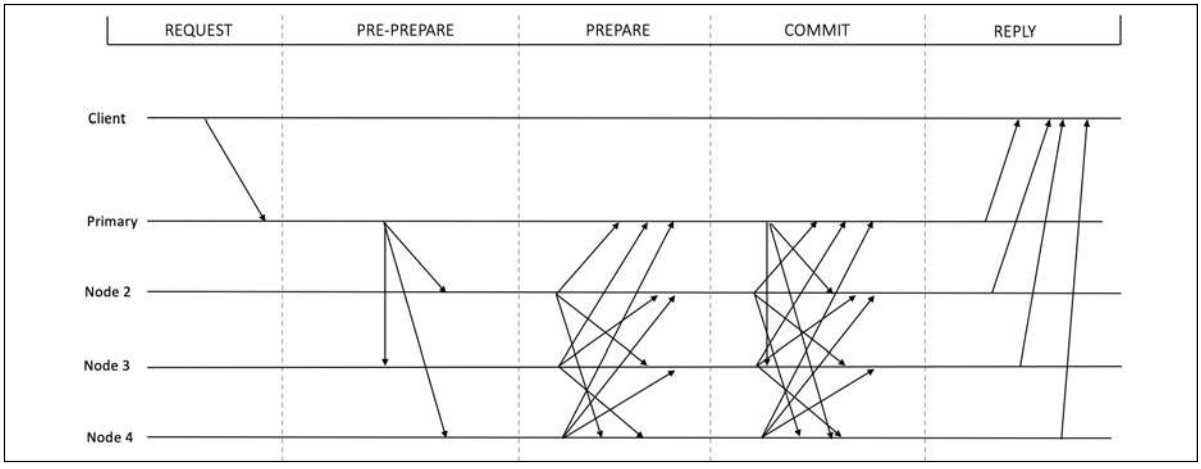


Figure 5.4: PBFT protocol

During the execution of the protocol, the integrity of the messages and protocol operations must be maintained to provide an adequate level of security and assurance. This is maintained by the use of digital signatures. In addition, certificates are used to ensure the adequate majority of participants (nodes).

Note that these certificates are not usual digital certificates commonly used in PKI and IT infrastructures to secure assets such as servers.

We'll describe the concept of certificates next.

Certificates in PBFT:

Certificates in PBFT protocols are used to demonstrate that at least *2F + 1* nodes have stored the required information. In other words, the collection of *2F + 1* messages of a particular type is considered a certificate. For example, if a node has collected *2F + 1* messages of type prepare, then combining it with the corresponding pre-prepare message with the same view, sequence, and request represents a certificate, called a prepared certificate. Similarly, a collection of *2F +1* commit messages is called a commit certificate.

There are also a number of variables that the PBFT protocol maintains in order to execute the algorithm. These variables and the meanings of these are listed as follows:

State variable	Explanation
v	View number
m	Latest request message
n	Sequence number of the message
h	Hash of the message
i	Index number
C	Set of all checkpoints
P	Set of all pre-prepare and corresponding prepare messages
O	Set of pre-prepare messages without corresponding request messages

We can now look at the types of messages and their formats, which becomes quite easy to understand if we refer to the preceding variables table shown.

Types of messages:

The PBFT protocol works by exchanging several messages. A list of these messages is presented as follows with their format and direction.

The following table contains message types and relevant details:

Message	From	To	Format	Signed by
Request	Client	Primary	`<REQUEST, m>`	Client
Pre-Prepare	Primary	Backups	`<PRE-PREPARE, v, n, h>`	Client
Prepare	Replica	Replicas	`<PREPARE, v, n, h, i>`	Replica
Commit	Replica	Replicas	`<COMMIT, v, n, h, i>`	Replica
Reply	Replicas	Client	`<REPLY, r, i>`	Replica
View change	Replica	Replicas	`<VIEWCHANGE, v+1, n, C, P, i>`	Replica
New view	Primary replica	Replicas	`<NEWVIEW, v + 1, v, O>`	Replica
Checkpoint	Replica	Replicas	`<CHECKPOINT, n, h, i>`	Replica

Let's look at some specific message types that are exchanged during the PBFT protocol.

View-change:

View-change occurs when a primary is suspected faulty. This phase is required to ensure protocol progress. With the view change sub-protocol, a new primary is selected, which then starts normal mode operation again. The new primary is selected in a round-robin fashion.

When a backup replica receives a request, it tries to execute it after validating the message, but for some reason, if it does not execute it for a while, the replica times out and initiates the view change sub-protocol.

In the view change protocol, the replica stops accepting messages related to the current view and updates its state to view-change. The only messages it can receive in this state are checkpoint messages, view-change messages, and new-view messages. After that, it sends a view-change message with the next view number to all replicas.

When this message arrives at the new primary, the primary waits for at least *2F* view-change messages for the next view. If at least *2F* view-change messages are received it broadcasts a new view message to all replicas and progresses toward running normal operation mode once again.

When other replicas receive a new-view message, they update their local state accordingly and start normal operation mode.

The algorithm for the view-change protocol is shown as follows:

1. Stop accepting **pre-prepare**, **prepare**, and **commit** messages for the current view.
2. Create a set of all the certificates prepared so far.
3. Broadcast a view-change message with the next view number and a set of all the prepared certificates to all replicas.

The view change protocol can be visualized in the following diagram:

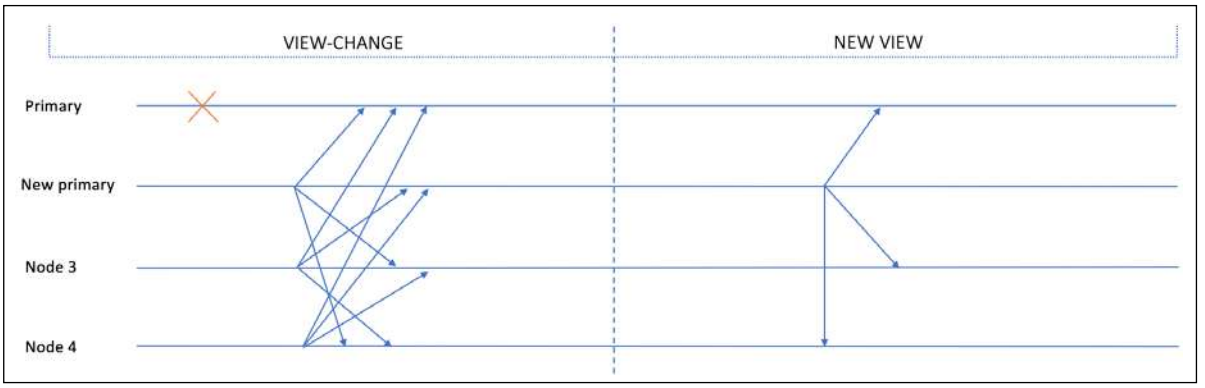


Figure 5.5: View-change sub-protocol

The view-change sub-protocol is a mechanism to achieve liveness. Three smart techniques are used in this sub-protocol to ensure that, eventually, there is a time when the requested operation executes:

1. A replica that has broadcast the view-change message waits for *2F+1* view-change messages and then starts its timer. If the timer expires before the node receives a new-view message for the next view, the node will start the view change for the next sequence but will increase its timeout value. This will also occur if the replica times out before executing the new unique request in the new view.
2. As soon as the replica receives *F+1* view-change messages for a view number greater than its current view, the replica will send the view-change message for the smallest view it knows of in the set so that the next view change does not occur too late. This is also the case even if the timer has not expired; it will still send the view change for the smallest view.
3. As the view change will only occur if at least *F+1* replicas have sent the view-change message, this mechanism ensures that a faulty primary cannot indefinitely stop progress by successively requesting view changes.

Next, let's look in more detail at the checkpoint sub-protocol, another important PBFT process.

The checkpoint sub-protocol:

Checkpointing is a crucial sub-protocol. It is used to discard old messages in the log of all replicas. With this, the replicas agree on a stable checkpoint that provides a snapshot of the global state at a certain point in time. This is a periodic process carried out by each replica after executing the request and marking that as a checkpoint in its log. A variable called **low watermark** (in PBFT terminology) is used to record the sequence number of the last stable checkpoint. This checkpoint is then broadcast to other nodes. As soon as a replica has at least *2F+1* checkpoint messages, it saves these messages as proof of a stable checkpoint. It discards all previous **pre-prepare, prepare**, and **commit** messages from its logs.

PBFT advantages and disadvantages:

PBFT is indeed a revolutionary protocol that has opened up a new research field of practical Byzantine fault-tolerant protocols. The original PBFT does have some strengths, but it does have some limitations. We'll discuss most of the commonly cited strengths and limitations in the following section.

Strengths:

PBFT provides immediate and deterministic transaction finality. This is in contrast with the PoW protocol, where a number of confirmations are required to finalize a transaction with high probability.

PBFT is also energy efficient as compared to PoW, which consumes a tremendous amount of electricity.

Weaknesses:

PBFT is not very scalable. This is the reason it is more suitable for consortium networks, instead of public blockchains. It is, however, much faster than PoW protocols.

Sybil attacks can be carried out on a PBFT network, where a single entity can control many identities to influence the voting and subsequently the decision. However, the fix is trivial and, in fact, this is not very practical in consortium networks where all identities are known on the network. This problem can be addressed simply by increasing the number of nodes in the network.

PBFT in blockchain:

In the traditional client-server model, PBFT works well; however, in the case of blockchain, directly implementing PBFT in its original state may not work correctly. This is because PBFT's original design was not developed for blockchain. In the following section, we present a few changes that are required to create a blockchain version of PBFT. This is not an exhaustive list; however, it does provide a baseline of requirements.

This research resulted in IBFT and PBFT implementation in Sawtooth and other blockchains. In all these scenarios, some changes have been made in the core protocol to ensure that they're compatible with the blockchain environment.

PBFT has been implemented in Hyperledger Sawtooth. More details on this implementation can be found here: `https://github.com/hyperledger/sawtooth-rfcs/blob/master/text/0019-pbft-consensus.md`

We will now introduce an algorithm called IBFT, which was inspired by PBFT. Another algorithm that has been inspired by PBFT and DLS is Tendermint, which we will also present shortly.

Nakamoto consensus

Nakamoto consensus, or **PoW**, was first introduced with Bitcoin in 2009. Since then, it has stood the test of time and is the longest-running blockchain network. This test of time is a testament to the efficacy of the PoW consensus mechanism. Now, we will explore how PoW works.

At a fundamental level, the PoW mechanism is designed to mitigate Sybil attacks, which facilitates consensus and the security of the network.

A Sybil attack is a type of attack that aims to gain a majority influence on the network to control the network. Once a network is under the control of an adversary, any malicious activity could occur. A Sybil attack is usually conducted by a node generating and using multiple identities on the network. If there are enough multiple identities held by an entity, then that entity can influence the network by skewing majority-based network decisions. The majority in this case is held by the adversary.

It is quite easy to obtain multiple identities and try to influence the network. However, in Bitcoin, due to the hashing power requirements, this attack is mitigated.

How PoW works

In a nutshell, PoW works as follows:

- PoW makes use of hash puzzles.

- A node proposes a block has to find a nonce such that *H (nonce || previous hash || Tx || Tx|| . . . ||Tx) < Threshold value*.

The process can be summarized as follows:

- New transactions are broadcast to all nodes on the network.
- Each node collects the transactions into a candidate block.
- Miners propose new blocks.
- Miners concatenate and hash with the header of the previous block.
- The resultant hash is checked against the target value, that is, the network difficulty target value.
- If the resultant hash is less than the threshold value, then PoW is solved, otherwise, the nonce is incremented and the node tries again. This process continues until a resultant hash is found that is less than the threshold value.

We can visualize how PoW works with the diagram shown here:

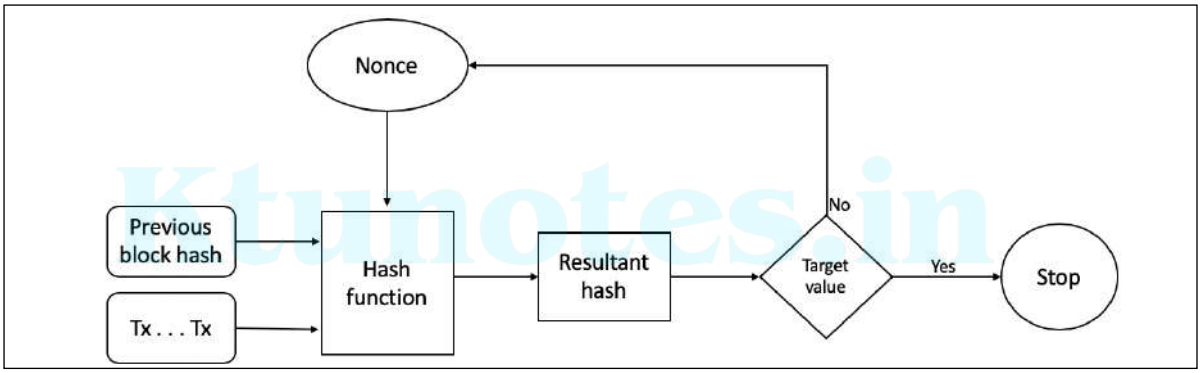


Figure 5.10: PoW diagram

PoW as a solution to the Byzantine generals problem

The original posting on this by Satoshi Nakamoto can be found here:

`https://satoshi.nakamotoinstitute.org/emails/cryptography/11/`

The key idea behind PoW as a solution to the Byzantine generals problem is that all honest generals (miners in the Bitcoin world) achieve agreement on the same state (decision value). As long as honest participants control the majority of the network, PoW solves the Byzantine generals problem. Note that this is a probabilistic solution and not deterministic.

Nakamoto versus traditional consensus

Nakamoto consensus was the first of its kind. It solved the consensus problem, which had been traditionally solved using pre-Bitcoin protocols like PBFT. A natural question that arises is, are there any similarities between the Nakamoto world and traditional distributed consensus? The short answer is yes, because we can map the properties of PoW consensus to traditional Byzantine consensus. This mapping is useful to understand what properties of traditional consensus can be applied to the Nakamoto world and vice versa.

In traditional consensus algorithms, we have **agreement**, **validity**, and **liveness** properties, which can be mapped to Nakamoto-specific properties of the **common prefix**, **chain quality**, and **chain growth** properties respectively.

The **common prefix** property means that the blockchain hosted by honest nodes will share the same large common prefix. If that is not the case, then the **agreement** property of the protocol cannot be guaranteed, meaning that the processors will not be able to decide and agree on the same value.

The **chain quality** property means that the blockchain contains a certain required level of correct blocks created by honest nodes (miners). If chain quality is compromised, then the **validity** property of the protocol cannot be guaranteed. This means that there is a possibility that a value will be decided that is not proposed by a correct process, resulting in safety violation.

The **chain growth** property simply means that new correct blocks are continuously added to the blockchain. If chain growth is impacted, then the **liveness** property of the protocol cannot be guaranteed. This means that the system can deadlock or fail to decide on a value.

PoW chain quality and common prefix properties are introduced and discussed in the following paper:

Garay, J., Kiayias, A. and Leonardos, N., 2015, April. The bitcoin backbone protocol: Analysis and applications. *In Annual International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 281-310). Springer, Berlin, Heidelberg.

`https://eprint.iacr.org/2014/765.pdf`

Variants of PoW

There are two main variants of PoW algorithms, based on the type of hardware used for their processing.

CPU-bound PoW

CPU-bound PoW refers to a type of PoW where the processing required to find the solution to the cryptographic hash puzzle is directly proportional to the calculation speed of the CPU or hardware such as ASICs. Because ASICs have dominated the Bitcoin PoW and provide somewhat undue advantage to the miners who can afford to use ASICs, this CPU-bound PoW is seen as shifting toward centralization. Moreover, mining pools with extraordinary hashing power can shift the balance of power towards them. Therefore, memory-bound PoW algorithms have been introduced, which are ASIC-resistant and are based on memory-oriented design instead of CPU.

Memory-bound PoW

Memory-bound PoW algorithms rely on system RAM to provide PoW. Here, the performance is bound by the access speed of the memory or the size of the memory. This reliance on memory also makes these PoW algorithms ASIC-resistant. Equihash is one of the most prominent memory-bound PoW algorithms.

PoW consumes tremendous amounts of energy; as such, there are several alternatives suggested by researchers. One of the first alternatives proposed is PoS.

Proof of stake (PoS)

PoS is an energy-efficient alternative to the PoW algorithm, which consumes an enormous amount of energy. PoS was first used in Peercoin, and now, prominent blockchains such as EOS, NxT, Steem, and Tezos are using PoS algorithms. Ethereum, with its Serenity release, will soon transition to a PoS-based consensus mechanism.

The stake represents the number of coins (money) in the consensus protocol staked by a blockchain participant. The key idea is that if someone has a stake in the system, then they will not try to sabotage the system. Generally speaking, the chance of proposing the next block is directly proportional to the value staked by the participant. However, there are a few intricate variations, which we will discuss shortly. There are different variations of PoS, such as chain-based PoS, committee-based PoS, BFT-based PoS, delegated PoS, leased PoS, and master node PoS.

In PoS systems, there is no concept of mining in the traditional Nakamoto consensus sense. However, the process related to earning revenue is sometimes called virtual mining. A PoS miner is called either a validator, minter, or stakeholder.

The right to win the next proposer role is usually assigned randomly. Proposers are rewarded either with transaction fees or block rewards. Similar to PoW, control over the majority of the network in the form of the control of a large portion of the stake is required to attack and control the network.

PoS mechanisms generally select a stakeholder and grant appropriate rights to it based on their staked assets. The stake calculation is application-specific, but generally, is based on balance, deposit value, or voting among the validators. Once the stake is calculated, and a stakeholder is selected to propose a block, the block proposed by the proposer is readily accepted. The probability of selection increases with a higher stake. In other words, the higher the stake, the better the chances of winning the right to propose the next block.

How PoS works

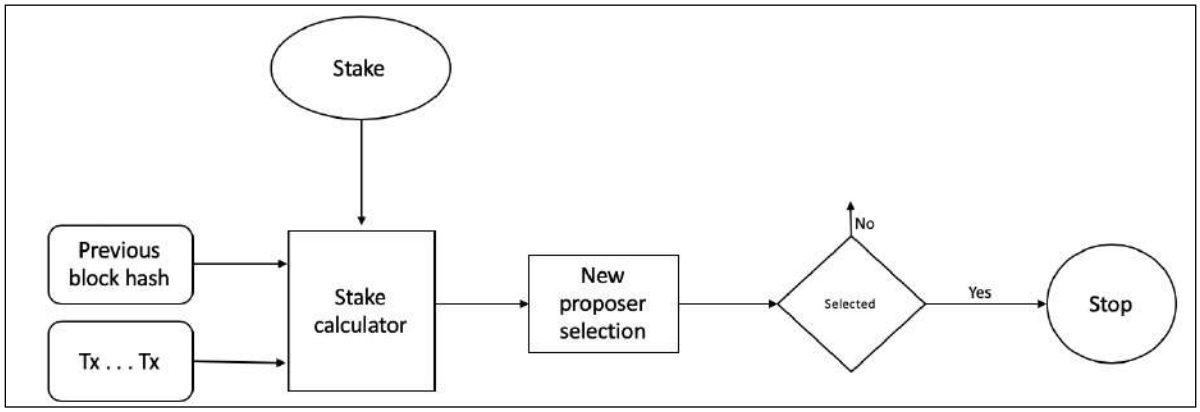


Figure 5.11: PoS diagram

In the preceding diagram, a generic PoS mechanism is shown where a stake calculator function is used to calculate the amount of staked funds and, based on that, select a new proposer.

Types of PoS

There are several types of PoS algorithm.

Chain-based PoS

This mechanism is very similar to PoW. The only change from the PoW mechanism is the block generation method. A block is generated in two steps by following a simple protocol:

- Transactions are picked up from the memory pool and a candidate block is created.
- A clock is set up with a constant tick interval, and at each clock tick, whether the hash of the block header concatenated with the clock time is less than the product of the target value and stake value is checked. This process can be shown in a simple formula:

$$Hash\ (B\ ||\ clock\ time) < target\ x\ stake\ value$$

The stake value is dependent on the way the algorithm is designed. In some systems, it is directly proportional to the amount of stake, and in others, it is based on the amount of time the stake has been held by the participant (also called coinage). The target is the mining difficulty per unit of the value of stake.

This mechanism still uses hashing puzzles, as in PoW. But, instead of competing to solve the hashing puzzle by consuming a high amount of electricity and specialized hardware, the hashing puzzle in PoS is solved at regular intervals based on the clock tick. A hashing puzzle becomes easier to solve if the stake value of the minter is high.

Peercoin was the first blockchain to implement PoS.

Nxt (`https://www.jelurida.com/nxt`) and Peercoin (`https://www.peercoin.net`) are two examples of blockchains where chain-based PoS is implemented.

Committee-based PoS

In this scheme, a group of stakeholders is chosen randomly, usually by using a **verifiable random function** (**VRF**). This VRF, once invoked, produces a random set of stakeholders based on their stake and the current state of the blockchain. The chosen group of stakeholders becomes responsible for proposing blocks in sequential order.

This mechanism is used in the Ouroboros PoS consensus mechanism, which is used in Cardano. More details on this are available here:

`https://www.cardano.org/en/ouroboros/`

VRFs were introduced in *Chapter 4*, *Public Key Cryptography*. More information on VRF can be found here:

`https://tools.ietf.org/id/draft-goldbe-vrf-01.html`.

Delegated PoS

DPoS is very similar to committee-based PoS, but with one crucial difference. Instead of using a random function to derive the group of stakeholders, the group is chosen by stake delegation. The group selected is a fixed number of minters that create blocks in a round-robin fashion. Delegates are chosen via voting by network users. Votes are proportional to the amount of the stake that participants have on the network. This technique is used in Lisk, Cosmos, and EOS. DPoS is not decentralized as a small number of known users are made responsible for proposing and generating blocks.

Bitcoin definition

Bitcoin can be defined in various ways; it's a protocol, a digital currency, and a platform. It is a combination of a peer-to-peer network, protocols, and software that facilitates the creation and usage of the digital currency. Nodes in this peer-to-peer network talk to each other using the Bitcoin protocol.

Decentralization of currency was made possible for the first time with the invention of Bitcoin. Moreover, the double-spending problem was solved in an elegant and ingenious way in Bitcoin. The double-spending problem arises when, for example, a user sends coins to two different users at the same time and they are verified independently as valid transactions. The double-spending problem is resolved in Bitcoin by using a distributed ledger (the blockchain) where every transaction is recorded permanently, and by implementing a transaction validation and confirmation mechanism. This process will be explained later in the chapter, in the *Mining* section.

ɔes

Cryptographic keys

On the Bitcoin network, possession of Bitcoins and the transfer of value via transactions are reliant upon private keys, public keys, and addresses. **Elliptic Curve Cryptography** (**ECC**) is used to generate public and private key pairs in the Bitcoin network. We have already covered these concepts in *Chapter 4, Public Key Cryptography*, and here we will see how private and public keys are used in the Bitcoin network.

Private keys in Bitcoin

Private keys are required to be kept safe and normally reside only on the owner's side. Private keys are used to digitally sign the transactions, proving 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.

For example, the consider the following private key:

```
A3ED7EC8A03667180D01FB4251A546C2B9F2FE33507C68B7D9D4E1FA5714195201
```

When converted into WIF format, it looks as shown here:

```
L2iN7umV7kbr6LuCmgM27rBnptGbDVc8g4ZBm6EbgTPQXnj1RCZP
```

Interested readers can do some experimentation using the tool available at `http://gobittest.appspot.com/PrivateKey`.

Also, **mini private key format** is sometimes used to create the private key with a maximum of 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 is more damage resistant because more dots can be used for error correction and less for encoding the private key.

QR codes use Reed-Solomon error correction. Discussion on the error correction mechanism and its underlying details is out of the scope of this book, but interested readers can refer to `https://en.wikipedia.org/wiki/QR_code#Error_correction` for further information.

A private key encoded using mini private key format is also sometimes called a **minikey**. The first character of the mini private key is always the uppercase letter `S`. A mini private key can be converted into a normal-sized private key, but an existing normal-sized private key cannot be converted into a mini private key. This format was used in **Casascius** physical bitcoins.

Interested readers can find more information at `https://en.Bitcoin.t/wiki/Casascius_physical_bitcoins`.

Figure 6.9: A Casascius physical Bitcoin's security hologram paper with mini key and QR code

The Bitcoin core client also allows the encryption of the wallet that contains the private keys.

As we learned in *Chapter 4*, *Public Key Cryptography*, private keys have their own corresponding public keys. Public keys on their own are useless and private keys on their own are equally of no use. Pairs of public and private keys are required for the normal functioning of any public key cryptography-based systems such as the Bitcoin blockchain. In the next section, we introduce the usage of public keys in Bitcoin.

Public keys in Bitcoin

Public keys exist on the blockchain and all network participants can see them. Public keys are derived from private keys due to their special mathematical relationship with those private keys. Once a transaction signed with the private key is broadcast 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 an **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) to ensure that funds remain secure and can only be spent by the legitimate owner. If you need to refresh the relevant cryptography concepts, you can refer to *Chapter 4*, *Public Key Cryptography*, where ECC was explained. Public keys can be represented in uncompressed or compressed format, and are fundamentally x and y coordinates on an elliptic curve. In uncompressed format, public keys are presented with a prefix of `0x4` in hexadecimal format. x and y coordinates are both 32 bytes in length. In total, a compressed public key is 33 bytes long, compared to the 65-byte uncompressed format. The compressed version of public keys include only the x part, since the y part can be derived from it.

The reason why the compressed version of public keys works is that if the ECC graph is visualized, it reveals that the y coordinate can be either below the x axis or above the x axis, and as the curve is symmetric, only the location in the prime field is required to be stored. If y is even then its value lies above the x axis, and if it is odd then it is below the x axis. This means that instead of storing both x and y as the public key, only x needs to be stored with the information about whether y is even or odd.

Initially, the Bitcoin client used uncompressed keys, but starting from Bitcoin Core client 0.6, compressed keys are used as standard. This resulted in an 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. Uncompressed public keys are 65 bytes long. They are encoded as 256-bit unsigned big-endian integers (32 bytes), which are concatenated together and finally prefixed with a byte `0x04`. This means 1 byte for the `0x04` prefix, 32 bytes for the x integer, and 32 bytes for y integer, which makes it 65 bytes in total.
- Compressed public keys start with `0x03` if the y 32-byte (256-bit) part of the public key is odd. It is 33 bytes in length as 1 byte is used by the `0x03` prefix (depicting an odd y) and 32 bytes for storing the x coordinate.

- Compressed public keys start with `0x02` if the y 32-byte (256-bit) part of the public key is even. It is 33 bytes in length as 1 byte is used by the `0x02` prefix (depicting an even y) and 32 bytes for storing the x coordinate.

Having talked about private and public keys, let's now move on to another important aspect of Bitcoin: addresses derived from public keys.

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 SHA256 algorithm and then with RIPEMD160. 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 digits 1 or 3.

A typical Bitcoin address looks like the string shown here:

```
1ANAguGG8bikEv2fYsTBnRUmx7QUcK58wt
```

Addresses are also commonly encoded in a QR code for easy distribution. The QR code of the preceding Bitcoin address is shown in the following image:

Figure 6.10: QR code of the Bitcoin address 1ANAguGG8bikEv2fYsTBnRUmx7QUcK58wt

Currently, there are two types of addresses, the commonly used P2PKH and another P2SH type, starting with numbers 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, but Bitcoin has some other security issues as well, such as transaction malleability, Sybil attacks, race attacks, and selfish mining, all of which require different approaches to resolve.

Transaction malleability has been resolved with the so-called SegWit soft-fork upgrade of the Bitcoin protocol. This concept will be explained later in the chapter.

Figure 6.11: From bitaddress.org, a private key and Bitcoin address in a paper wallet

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. More explanation and logic can be found in the `base58.h` source file in the Bitcoin source code:

```
/**
* Why base-58 instead of standard base-64 encoding?
* - Don't want 0OIl characters that look the same in some fonts and
*      could be used to create visually identical looking data.
* - A string with non-alphanumeric characters is not as easily accepted as input.
* - E-mail usually won't line-break if there's no punctuation to break at.
* - Double-clicking selects the whole string as one word if it's all
alphanumeric.
*/
```

This file is present in the Bitcoin source code and can be viewed at `https://github.com/bitcoin/bitcoin/blob/c8971547d9c9460fcbec6f54888df83f002c3dfd/src/base58.h`.

Now that we have covered private keys, public keys, Base58 encoding, and addresses, with that knowledge we can now examine how a Bitcoin address is generated by using all these elements. We briefly touched on it when we introduced addresses in Bitcoin earlier, but now we will see this in more detail with a diagram.

The following diagram shows how an address is generated, from generating the private key to the final output of the Bitcoin address:

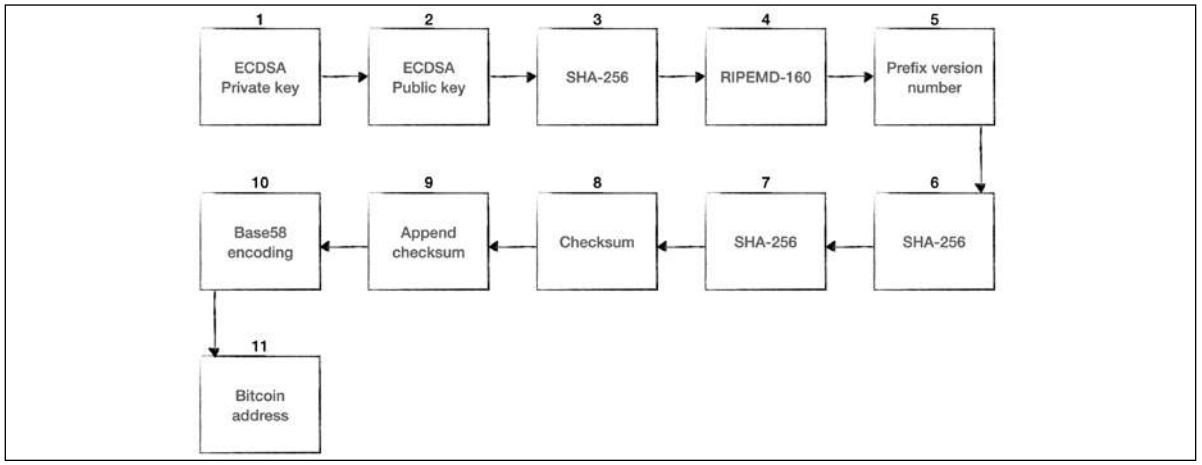


Figure 6.12: Address generation in Bitcoin

In the preceding diagram, there are several steps that we will now explain:

1. In the first step, we have a randomly generated ECDSA private key.
2. The public key is derived from the ECDSA private key.
3. The public key is hashed using the SHA-256 cryptographic hash function.
4. The hash generated in *step 3* is hashed using the RIPEMD-160 hash function.
5. The version number is prefixed to the RIPEMD-160 hash generated in *step 4*.
6. The result produced in *step 5* is hashed using the SHA-256 cryptographic hash function.
7. SHA-256 is applied again.
8. The first 4 bytes of the result produced from *step 7* is the address `checksum`.
9. This `checksum` is appended to the RIPEMD-160 hash generated in *step 4*.
10. The resultant byte string is encoded into a Base58-encoded string by applying the Base58 encoding function.
11. Finally, the result is a typical Bitcoin address.

In addition to common types of addresses in Bitcoin, there are some advanced types of addresses available in Bitcoin too. We discuss these next.

Vanity addresses

As Bitcoin addresses are based on Base58 encoding, it is possible to generate addresses that contain human-readable messages. An example is shown as follows—note that in the first line, the name `BasHir` appears:

Figure 6.13: Vanity public address encoded in QR

Vanity addresses are generated using a purely brute-force method. An example of a paper wallet with a vanity address is shown in the following screenshot:

Figure 6.14: Vanity address generated from https://bitcoinvanitygen.com/

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

Multi-signature addresses

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

Remember that we discussed this concept in *Chapter 4*, *Public Key Cryptography*.

With this section, we've finished our introduction to addresses in Bitcoin. In the next section, we will introduce Bitcoin transactions, which are the most fundamental and important aspect of Bitcoin.

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 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 should send coins to some other user (a Bitcoin address), then it needs to be signed by the sender with their private key. In this case, a reference is also required to the previous transaction to show the origin of the coins. Coins are unspent transaction outputs represented in Satoshis.

Transactions are not encrypted and are publicly visible on the blockchain. Blocks are made up of transactions, and these can be viewed using any online blockchain explorer.

The transaction lifecycle

Now, let's look at the lifecycle of a Bitcoin **transaction**. The steps of the process are as follows:

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 transactions are placed in the block, they are placed in a special memory buffer called the transaction pool. The purpose of the transaction pool is explained in the next section.
5. Next, the mining starts, which is the process through which the blockchain is secured and new coins are generated as a reward for the miners who spend appropriate computational resources. Once a miner solves the PoW problem, it broadcasts the newly mined block to the network. PoW is explained in detail in the *Mining* section. The nodes verify the block and propagate the block further, and confirmations start to generate.
6. 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 six confirmations.

When a transaction is created by a user and sent to the network, it ends up in a special area on each Bitcoin software client. This special area is called the transaction pool or memory pool.

Coinbase transactions

A coinbase transaction or generation transaction is always created by a miner and is the first transaction in a block. It is used to create new coins. It includes a special field, also called the coinbase, which acts as an input to the coinbase transaction. This transaction also allows up to 100 bytes of arbitrary data storage.

A coinbase transaction input has the same number of fields as a usual transaction input, but the structure contains the coinbase data size and fields instead of the unlocking script size and fields. Also, it does not have a reference pointer to the previous transaction. This structure is shown in the following table:

Field	Size	Description
Transaction hash	32 bytes	Set to all zeroes as no hash reference is used
Output index	4 bytes	Set to `0xFFFFFFFF`
Coinbase data length	1-9 bytes	2-100 bytes
Data	Variable	Any data
Sequence number	4 bytes	Set to `0xFFFFFFFF`

All transactions in the Bitcoin system go under a validation mechanism. We introduce how Bitcoin transactions are validated in the next section.

Transaction validation

This verification process is performed by Bitcoin nodes. There are three main things that nodes check when verifying a transaction:

1. That transaction inputs are previously unspent. This validation step prevents double spending by verifying that the transaction inputs have not already been spent by someone else.
2. That the sum of the transaction outputs is not more than the total sum of the transaction inputs. However, both input and output sums can be the same, or the sum of the input (total value) could be more than the total value of the outputs. This check ensures that no new bitcoins are created out of thin air.
3. That the digital signatures are valid, which ensures that the script is valid.

Even though transaction construction and validation are generally a secure and sound process, some vulnerabilities exist in Bitcoin. We will now introduce some Bitcoin's infamous shortcomings.

Blockchain

As we discussed in *Chapter 1*, *Blockchain 101*, a blockchain is a distributed ledger of transactions. Specifically, from Bitcoin's perspective, the blockchain can be defined as a public, distributed ledger holding a timestamped, ordered, and immutable record of all transactions on the Bitcoin network. Transactions are picked up by miners and bundled into blocks for mining. Each block is identified by a hash and is linked to its previous block by referencing the previous block's hash in its header.

The data structure of a Bitcoin block is shown in the following table:

Field	Size	Description
Block size	4 bytes	The size of the block.
Block header	80 bytes	This includes fields from the block header described in the next section.
Transaction counter	Variable	The field contains the total number of transactions in the block, including the coinbase transaction. Size ranges from 1-9 bytes.
Transactions	Variable	All transactions in the block.

The block header mentioned in the previous table is a data structure that contains several fields. This is shown in the following table:

Field	Size	Description
Version	4 bytes	The block version number that dictates the block validation rules to follow.
Previous block's header hash	32 bytes	This is a double SHA256 hash of the previous block's header.
Merkle root hash	32 bytes	This is a double SHA256 hash of the Merkle tree of all transactions included in the block.
Timestamp	4 bytes	This field contains the approximate creation time of the block in Unix-epoch time format. More precisely, this is the time when the miner started hashing the header (the time from the miner's location).
Difficulty target	4 bytes	This is the current difficulty target of the network/block.
Nonce	4 bytes	This is an arbitrary number that miners change repeatedly to produce a hash that is lower than the difficulty target.

The following diagram provides a detailed view of the blockchain structure. As shown in the following diagram, 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:

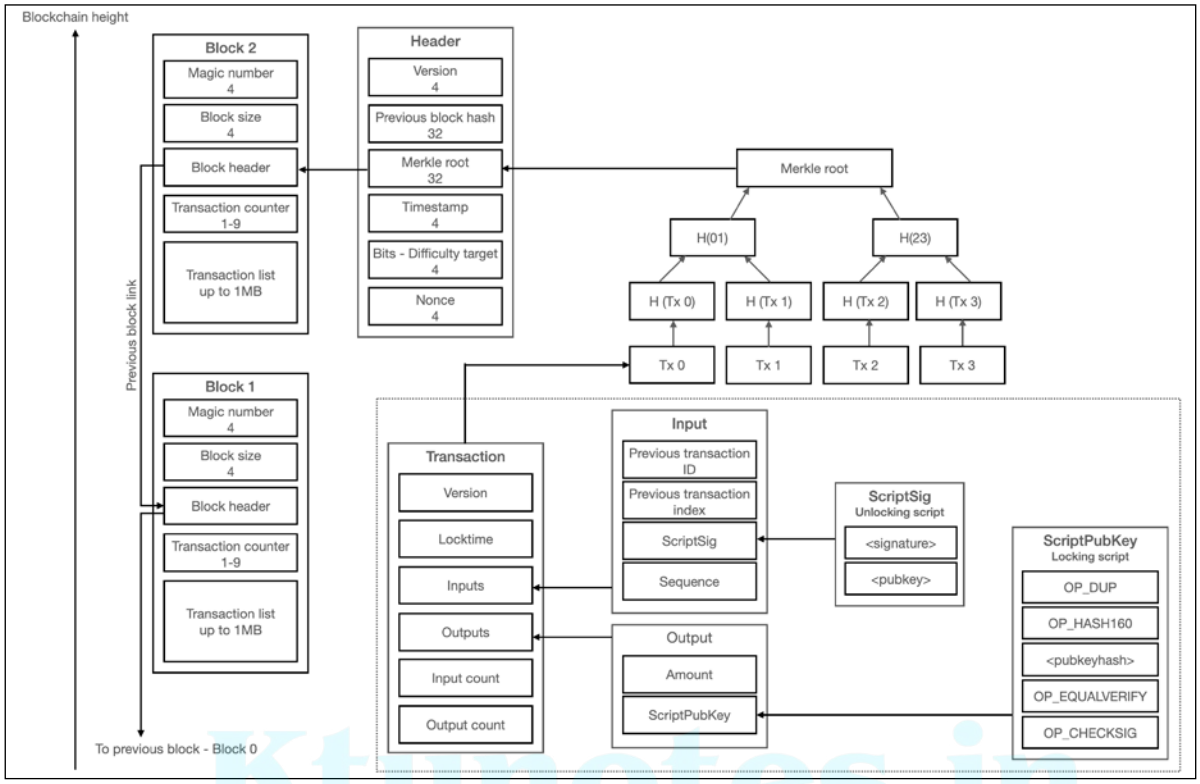


Figure 6.16: A visualization of blockchain, block, block header, transactions, and scripts

The preceding diagram shows a high-level overview of the Bitcoin blockchain. On the left-hand side, blocks are shown starting from bottom to top. Each block contains transactions and block headers, which are further magnified on the right-hand side. At the top, first, the block header is enlarged 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 how Merkle root is constructed.

We have discussed Merkle trees in detail previously. You can refer to *Chapter 4*, *Public Key Cryptography*, if you need to revise the concept.

Further down the diagram, 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 to show what locking and unlocking scripts look like. The size (in bytes) of each field of block, header and transaction is also shown as a number under the name of the field.

Let's move on to discuss the first block in the Bitcoin blockchain: the genesis block.

The genesis block

This is the first block in the Bitcoin blockchain. The genesis block was hardcoded in the Bitcoin core software. In the genesis block, the coinbase transaction included a comment taken from The Times newspaper:

"*The Times 03/Jan/2009 Chancellor on brink of second bailout for banks*"

This message is a proof that the first Bitcoin block (genesis block) was not mined earlier than January 3rd, 2009. This is because the genesis block was created on January 3rd, 2009 and this news excerpt was taken from that day's newspaper.

The following representation of the genesis block code can be found in the `chainparams.cpp` file available at `https://github.com/Bitcoin/Bitcoin/blob/master/src/chainparams.cpp`:

```
static CBlock CreateGenesisBlock(uint32_t nTime, uint32_t nNonce, uint32_t nBits,
int32_t nVersion, const CAmount& genesisReward)
{
    const char* pszTimestamp = "The Times 03/Jan/2009 Chancellor on brink of
second bailout for banks";
    const CScript genesisOutputScript = CScript() << ParseHex("04678afdb0fe554827
1967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5
c384df7ba0b8d578a4c702b6bf11d5f") << OP_CHECKSIG;
    return CreateGenesisBlock(pszTimestamp, genesisOutputScript, nTime, nNonce,
nBits, nVersion, genesisReward);
}
```

Bitcoin provides protection against double-spending by enforcing strict rules on transaction verification and via mining. Transactions and blocks are added to the blockchain only after the strict rule-checking explained in the *Transaction validation* section on successful PoW solutions.

Block height is the number of blocks before a particular block in the blockchain. PoW is used to secure the blockchain. Each block contains one or more transactions, out of which the first transaction is the coinbase transaction. There is a special condition for coinbase transactions that prevent them from being spent until at least 100 blocks have passed in order to avoid a situation where the block may be declared stale later on.

Stale and orphan blocks

Stale blocks are old blocks that have already been mined. Miners who keep working on these blocks due to a fork, where the longest chain (main chain) has already progressed beyond those blocks, are said to be working on a stale block. In other words, these blocks exist on a shorter chain, and will not provide any reward to their miners.

Orphan blocks are a slightly different concept. Their parent blocks are unknown. As their parents are unknown, they cannot be validated. This problem occurs when two or more miners discover a block at almost the same time. These are valid blocks and were correctly discovered at some point in the past but now they are no longer part of the main chain. The reason why this occurs is that if there are two blocks discovered at almost the same time, the one with a larger amount of PoW will be accepted and the one with the lower amount of work will be rejected. Similar to stale blocks, they do not provide any reward to their miners.

We can see this concept visually in the following diagram:

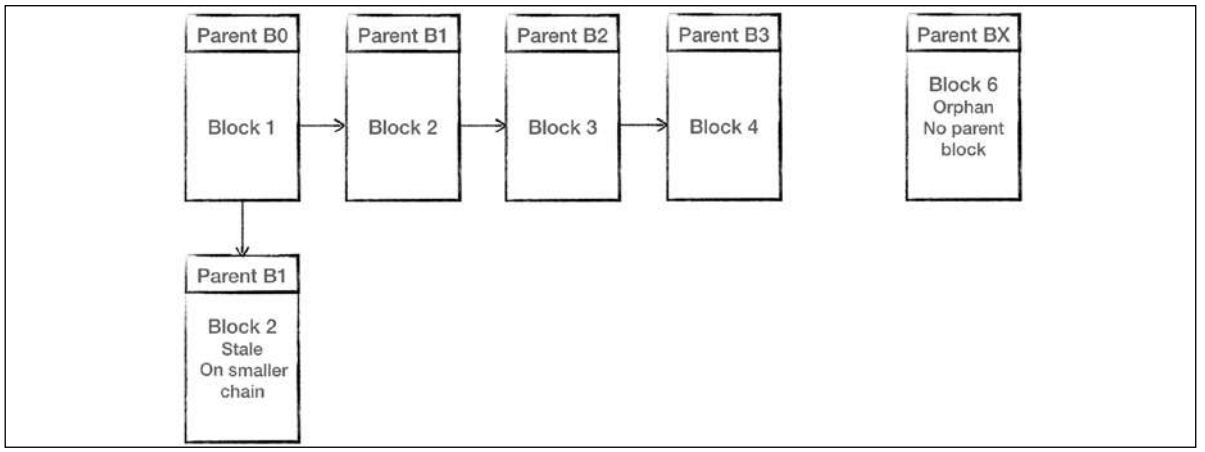


Figure 6.17: Orphan and stale blocks

In the preceding introduction to stale blocks, we introduced a new term, a **fork**. A fork is a condition that occurs when two different versions of the blockchain exist. It is acceptable in some conditions, and detrimental in a few others. There are different types of forks that can occur in a blockchain:

- Temporary forks
- Soft forks
- Hard forks

Because of the distributed nature of Bitcoin, network forks can occur inherently. In cases where two nodes simultaneously announce a valid block, it can result in a situation where there are two blockchains with different transactions. This is an undesirable situation but can be addressed by the Bitcoin network only by accepting the longest chain. In this case, the smaller chain will be considered orphaned. If an adversary manages to gain control of 51% of the network hash rate (computational power), then they can impose their own version of the transaction history.

Forks in a blockchain can also occur with the introduction of changes to the Bitcoin protocol. In the case of a **soft fork**, a client that chooses not to upgrade to the latest version supporting the updated protocol will still be able to work and operate normally. In this case, new and previous blocks are both acceptable, thus making a soft fork backward compatible. Miners are only required to upgrade to the new soft fork client software in order to make use of the new protocol rules. Planned upgrades do not necessarily create forks because all users should have updated already.

A **hard fork**, on the other hand, invalidates previously valid blocks and requires all users to upgrade. New transaction types are sometimes added as a soft fork, and any changes such as block structure changes or major protocol changes result in a hard fork.

As Bitcoin evolves and new upgrades and innovations are introduced in it, the version associated with blocks also changes. These versions introduce various security parameters and new features. The latest Bitcoin block version is 4, which was proposed with BIP65 and has been used since Bitcoin Core client 0.11.2. Since the implementation of BIP9, bits in the **nVersion** field are used to indicate soft-fork changes.

Details of BIP0065 are available at `https://github.com/Bitcoin/bips/blob/master/bip-0065.mediawiki`.

Size of the blockchain

Bitcoin is an ever-growing chain of blocks and is increasing in size. The current size of the Bitcoin blockchain stands at approximately 269 GB. The following figure shows the increase in size of the blockchain over time:

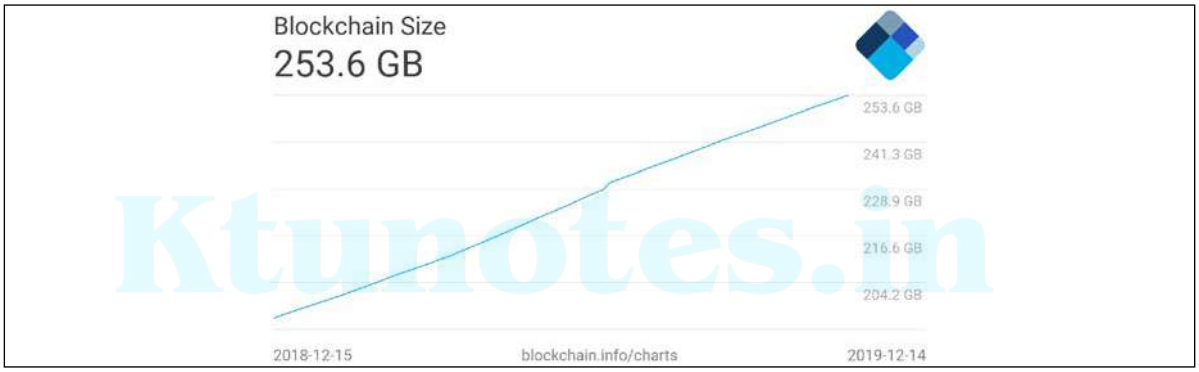


Figure 6.18: Size of Bitcoin blockchain over time

As the chain grows and more miners are added to the network, the network difficulty also increases.

Network difficulty

Network difficulty refers to a measure of how difficult it is to find a new block, or in other words, how difficult it is to find a hash below the given target.

New blocks are added to the blockchain approximately every 10 minutes, and the network difficulty is adjusted dynamically every 2,016 blocks in order to maintain a steady addition of new blocks to the network.

Network difficulty is calculated using the following equation:

$$Target = Previous\ target * Time/2016 * 10\ minutes$$

Difficulty and target are interchangeable and represent the same thing. The *Previous target* represents the old target value, and *Time* is the time spent to generate the previous 2,016 blocks. Network difficulty essentially means how hard it is for miners to find a new block; that is, how difficult the hashing puzzle is now.

In the next section, mining is discussed, which will explain how the hashing puzzle is solved.

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 to find a number 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 miners have spent the required resources before a new proposed block can be accepted. The miners mint new coins by solving the PoW problem, also known as the partial hash inversion problem. This process consumes a high amount of resources, including computing power and electricity. This process also secures the system against fraud 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. It is encoded in the Bitcoin Core client to control the "money supply."

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 an average of 144 per day. Bitcoin supply is also limited. In 2140, all 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.

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 that only concerns miners.
2. **Transaction validation**: Transactions broadcast 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 broadcast on the network after validating them.

5. **Perform PoW**: 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.

We've discussed the tasks—let's now consider the rewards for performing them.

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 broadcast 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 SHA256 algorithm.
4. Check if the resulting hash is lower than the current difficulty level (the 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 the 32-bit nonce was exhausted too quickly. In order to address this issue, the extra nonce solution was implemented, whereby the coinbase transaction is used to provide a larger range of nonces to be searched by the miners.

This process is visualized in the following flowchart:

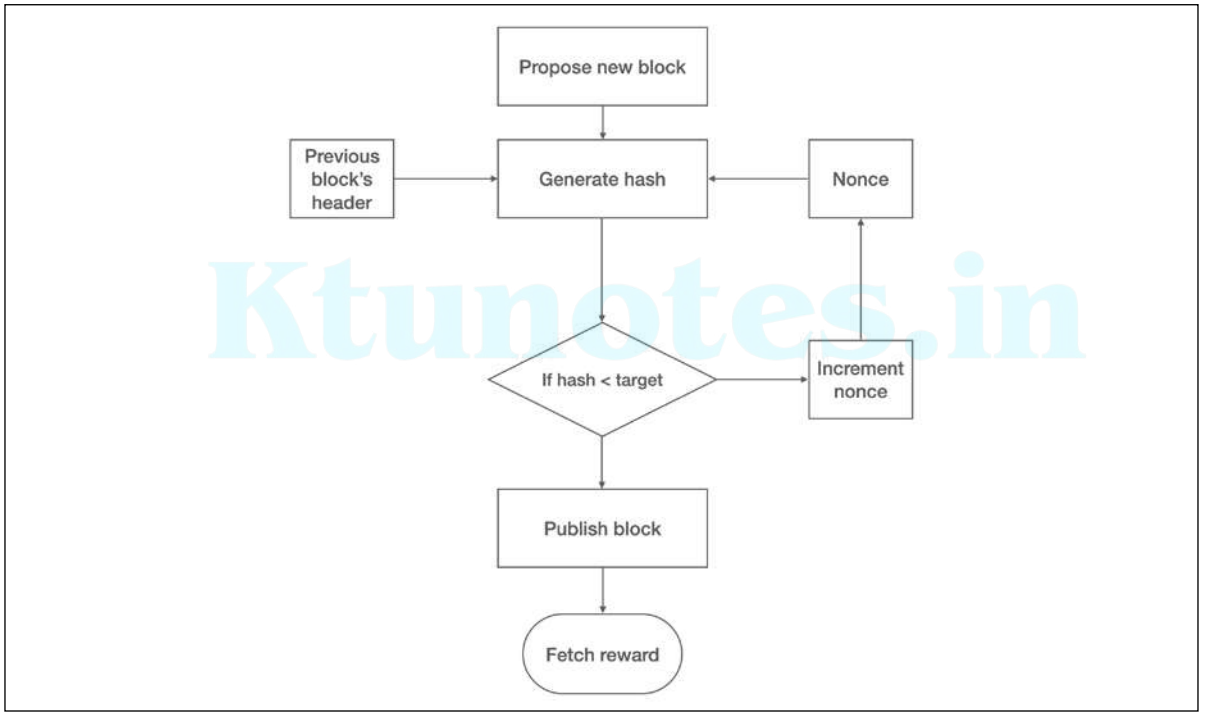


Figure 6.19: Mining process

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

```
$ bitcoin-cli getdifficulty
```

This generates something like the following:

```
12876842089683.48
```

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 how difficult it is to find a hash 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 2,016 blocks to ensure that on average, the 10-minute block generation time is maintained. Bitcoin network difficulty has increased in a roughly exponential fashion. The following graph shows this difficulty level over a period of one year:

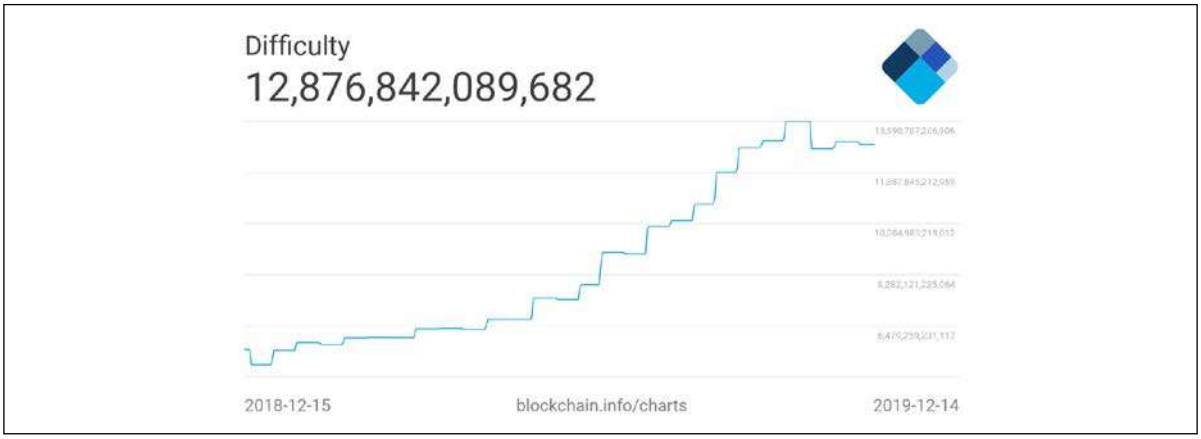


Figure 6.20: Mining difficulty since late 2018

The preceding graph shows the difficulty of the Bitcoin network over last year, and it has increased quite significantly. The reason why mining difficulty increases is because, in Bitcoin, the block generation time always has to be around 10 minutes. This means that if blocks are being mined too quickly because of faster hardware, the difficulty increases accordingly so that the block generation time can remain at roughly 10 minutes per block. This phenomenon is also true in reverse. If blocks take longer than 10 minutes on average to mine, then the difficulty is decreased. The difficulty is calculated every 2,016 blocks (around two weeks) and adjusted accordingly. If the previous set of 2,016 blocks were mined in a period of less than two weeks, then the difficulty increases. Similarly, if 2,016 blocks were found in more than two weeks (bearing in mind that if blocks are mined every 10 minutes, then 2,016 blocks take 2 weeks to be mined), then the difficulty is decreased.

The Bitcoin miners have to calculate hashes to solve the PoW algorithm. If the difficulty goes up then a higher hash rate is required to find the blocks. The difficulty increases accordingly if more hashing power is added due to more miners joining the network. Now let's explain the hash rate in a bit more detail.

The hash rate

The hash rate basically represents the rate of hash calculation per second. In other words, this is the speed at which miners in the Bitcoin network are calculating hashes to find a block. In the early days of Bitcoin, it used to be quite small, as CPUs were used, which are relatively weak in mining terms. However, with dedicated mining pools and **Application Specific Integrated Circuits** (**ASICs**) now, this has gone up exponentially in the last few years. This has resulted in increased difficulty in the Bitcoin network.

The following hash rate graph shows the hash rate increases over time and is currently measured in exa-hashes. This means that in 1 second, Bitcoin network miners are computing more than 24,000,000,000,000,000,000 hashes per second:

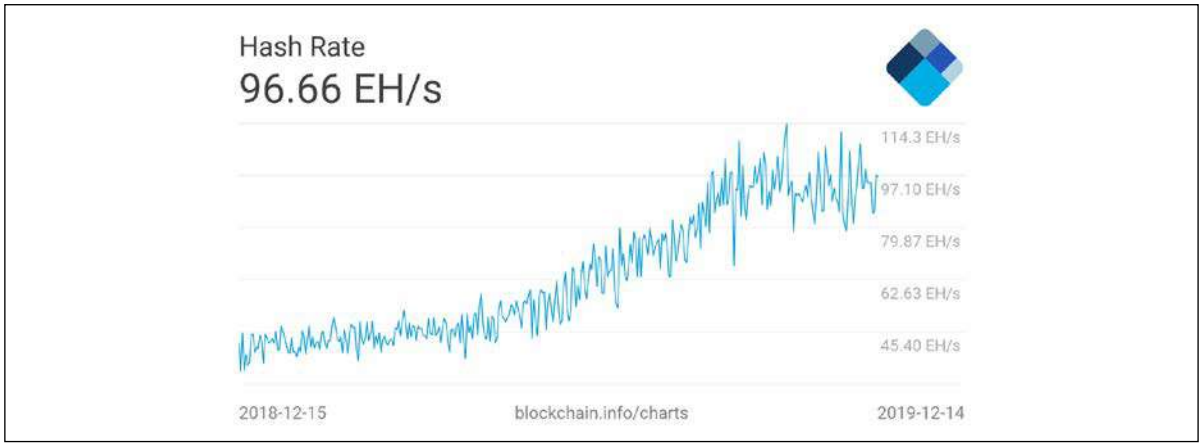


Figure 6.21: Hashing rate over time (measured in exa-hashes), shown over a period of 1 year

Wallets

The wallet software is used to generate and store cryptographic keys. It performs various useful functions, such as receiving and sending Bitcoin, backing up keys, and keeping track of the balance available. Bitcoin client software usually offers both functionalities: Bitcoin client and wallet. On disk, the Bitcoin Core client wallets are stored as a `Berkeley DB` file:

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

Private keys are generated by randomly choosing a 256-bit number provided by the 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 software, they also provide some functions to the users to manage and carry out transactions on the Bitcoin network. Let's take a look at the common types of 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 also generates keys as and when required. Managing a large number of keys is very difficult and an error-prone process that can lead to the 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 from 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 BIP39, 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`. These phrases can be used to recover all keys and make private key management comparatively easier.

Hierarchical deterministic wallets

Defined in BIP32 and BIP44, 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 hashes 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 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. 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:

Figure 7.5: A Trezor wallet

Online wallets

Online wallets, as the name implies, are stored entirely online and are provided as a service usually via 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 an 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 us with various methods to make payments, most notably the ability to use smartphone cameras to scan QR codes quickly and make payments.

Mobile wallets are available for Android and iOS and include Blockchain Wallet, Breadwallet, Copay, and Jaxx:

Figure 7.6: Jaxx mobile 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 deciding about which wallet to use, security should be of paramount importance. Hardware wallets tend to be more secure compared to web wallets because of their tamper-resistant design. Web wallets, by their very nature, are hosted on websites, which may not be as secure as a tamper-resistant hardware device.

Generally, mobile wallets for smart phone 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 Google Play. It is, however, quite difficult to suggest which type of wallet should be used as it also depends on personal preferences and the features available in the wallet. Security should be kept in mind while making a decision on which wallet to choose.

SPV client, which we introduced earlier, is also a type of Bitcoin wallet. Other types of Bitcoin wallets, especially mobile wallets, are mostly API wallets that rely on a mechanism where private and public keys are stored locally on the device where the wallet software is installed. Here, trusted backend severs are used for providing blockchain data via APIs.

Now that we've discussed the topic of Bitcoin wallets, let's see how Bitcoin payments can be adopted for business.

10

Smart Contracts

This chapter introduces smart contracts. This concept is not new; however, with the advent of blockchain technology, interest in this idea has revived. Smart contracts are now an ongoing and intense area of research in the blockchain space. Many blockchains have emerged that support smart contracts.

Due to benefits such as the increased security, cost-saving, and transparency that smart contracts can bring to many industries (especially the finance industry), rigorous research is in progress at various commercial and academic institutions to make the implementation of smart contracts easier, more practical, business-friendly, and more secure as soon as possible.

In this chapter, we will explore the ideas that led to the development of smart contracts, including its relevant history and key definitions. We will also cover concepts like the deployment of smart contracts, notable security aspects of smart contracts, and, finally, oracles. First, we will look at the history of smart contracts.

History

Nick Szabo first theorized smart contracts in the 1990s, in an article called *Formalizing and Securing Relationships on Public Networks*. This theory was presented almost 20 years before the real potential and benefits of smart contracts were appreciated, that is, before the invention of Bitcoin and the subsequent development of other more advanced blockchain platforms, such as Ethereum.

Smart contracts are described by Szabo as follows:

> *"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. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs."*

The original article that was written by Szabo is available at `http://firstmonday.org/ojs/index.php/fm/article/view/548`.

Smart contract functionality was implemented in a limited fashion in Bitcoin in 2009. Bitcoin supports a restricted scripting language called **script**, which allows the transfer of bitcoins between users. However, this is not a Turing complete language and does not support arbitrary program development. It can be regarded as a limited function calculator with only simple arithmetic operations, whereas smart contracts can be considered general-purpose computers that support writing any program.

Definition

There are many definitions of smart contracts. An online search reveals many definitions. However, while the definitions provided online at various sources are correct and useful, in my opinion, they are not complete. It is crucial to properly and completely define what a smart contract is. The following is my attempt to provide a comprehensive generalized definition of a smart contract:

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

Dissecting this definition reveals that a smart contract is, fundamentally, a computer program that is written in a language that a computer or target machine can understand. Also, it encompasses agreements between parties in the form of business logic. Another fundamental idea is that smart contracts are automatically executed according to the instruction that is coded in, for example, when certain conditions satisfy. They are enforceable, which means that all contractual terms perform as specified and expected, even in the presence of adversaries.

Enforcement is a broader term that encompasses traditional enforcement in the form of a law, along with the implementation of specific measures and controls that make it possible to execute contract terms without requiring any intervention.

Preferably, smart contracts should not rely on any traditional methods of enforcement. Instead, they should work on the principle that *code is the law*, which means that there is no need for an arbitrator or a third party to enforce, control, or influence the execution of a smart contract. Smart contracts are self-enforcing as opposed to legally enforceable. This idea may sound like a libertarian's dream, but it is entirely possible and is in line with the true spirit of smart contracts.

Moreover, they are secure and unstoppable, which means that these computer programs are fault-tolerant and executable in a reasonable (finite) amount of time. These programs should be able to execute and maintain a healthy internal state, even if external factors are unfavorable. For example, imagine a typical computer program that is encoded with some logic and executes according to the instruction coded within it. However, if the environment it is running in or the external factors it relies on deviate from the usual or expected state, the program may react arbitrarily or abort. Smart contracts must be immune to this type of issue.

Blockchain platforms play a vital role in providing the necessary underlying network with security guarantees required to run the smart contracts.

In some scenarios, security and unstoppability may well be considered optional features. Still, it will provide more significant benefits in the long run if security and unstoppable properties are included in the smart contract definition. This inclusion will allow researchers to focus on these aspects from the start and will help to build strong foundations on which further research can then be based. There is also a suggestion by some researchers that smart contracts do not need to be automatically executable; instead, they can be what's called automatable, due to the manual human input required in some scenarios. For example, the manual verification of a medical record might be needed by a qualified medical professional. In such cases, fully automated approaches may not work best. While it is true that, in some instances, human input and control are helpful, they are not necessary. For a contract to be truly smart, in my opinion, it has to be fully automated. Certain inputs that need to be provided by people can and should also be automated. Oracles can be used for this purpose. We will discuss oracles in more detail later on in this chapter.

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

There is also an ongoing debate on whether computer code is acceptable as a conventional contract in a court of law. A smart contract is different in presentation from traditional legal prose, albeit they do represent and enforce all required contractual clauses. Still, a court of law does not understand computer code. This dilemma raises several questions about how a smart contract can be legally binding: can it be developed in such a way that it is readily acceptable and understandable in a court of law? Is it possible for dispute resolution be implemented within the code? Moreover, regulatory and compliance requirements are other topics that require addressing before smart contracts can become as efficient as traditional legal documents.

The legality of smart contracts is uncertain in many jurisdictions. Still, a recent exciting development in this space made crypto assets and smart contracts valid in English law by recognizing crypto assets as tradeable property and smart contracts as enforceable agreements. This announcement was made by the **UK Jurisdiction Taskforce** (**UKJT**) of the Lawtech Delivery Panel. More information about this, including the full legal statement, is available here: `https://technation.io/news/uk-takes-significant-step-in-legal-certainty-for-smart-contracts-and-cryptocurrencies/`.

Even though smart contracts are named smart, they only do what they have been programmed to do. This very property of smart contracts ensures that smart contracts produce the same output every time they are executed. The deterministic nature of smart contracts is highly desirable in blockchain platforms due to consistency requirements.

Now, this gives rise to a problem whereby a large gap between the real world and the blockchain world emerges. In this situation, natural language is not understood by the smart contract, and, similarly, computer code is not acceptable for the natural world. So, a few questions arise: how can real-life contracts be deployed on a blockchain? How can this bridge between the real world and the smart contract world be built?

These questions open up various possibilities, such as making smart contract code readable not only by machines but also by people. If humans and machines can both understand the code written in a smart contract, it might become acceptable in legal situations, as opposed to just a piece of code that no one understands except for programmers. This desirable property is an area that is ripe for research, and a significant research effort has been expended in this area to answer questions around the semantics, meaning, and interpretation of smart contracts.

Some work has already been done to describe natural language contracts formally, by combining both smart contract code and natural language contracts through linking contract terms with machine-understandable elements. This is achieved using a markup language called the **Legal Knowledge Interchange Format** (**LKIF**), which is an XML schema for representing theories and proofs. It was developed under the Estrella project in 2008.

More information is available in the research paper at `https://doi.org/10.1007/978-3-642-15402-7_30`.

Further details can be found here: `http://www.estrellaproject.org/?page_id=5`.

Another requirement regarding the properties of smart contracts is that they must be deterministic. This property will allow a smart contract to be run by any node on a network and achieve the same result. If the result differs even slightly between nodes, then a consensus cannot be reached, and a whole paradigm of distributed consensuses on the blockchain can fail. Moreover, it is also desirable that the contract language itself is deterministic, thus ensuring the integrity and stability of the smart contracts. A deterministic property means that for the same input, there is always the same output. In other words, the system must not manifest different behavior for the same input in different runs. In a blockchain network, this would mean that smart contracts developed using the smart contract programming language do not produce different results on different nodes.

Let's take, for example, various floating-point operations calculated by various functions in a variety of programming languages that can produce different results in different runtime environments. Another example is some math functions in JavaScript, which can produce different results for the same input on different browsers and can, in turn, lead to various bugs. This scenario is unacceptable in smart contracts because if the results are inconsistent between the nodes, then a consensus will never be achieved.

The deterministic feature ensures that smart contracts always produce the same output for a specific input. In other words, programs, when executed, produce a reliable and accurate business logic that is entirely in line with the requirements programmed in the high-level code.

In summary, a smart contract has the following properties:

- **Automatically executable**: It is self-executable on a blockchain without requiring any intervention.
- **Enforceable**: This means that all contract conditions are enforced automatically.
- **Secure**: This means that smart contracts are tamper-proof (or tamper-resistant) and run with security guarantees. The underlying blockchain usually provides these security guarantees; however, the smart contract programing language and the smart contract code themselves must be correct, valid, and verified.
- **Deterministic**: The deterministic feature ensures that smart contracts always produce the same output for a specific input. Even though it can be considered to be part of the secure property, defining it here separately ensures that the deterministic property is considered one of the important properties.
- **Semantically sound**: This means that they are complete and meaningful to both people and computers.
- **Unstoppable**: This means that adversaries or unfavorable conditions cannot negatively affect the execution of a smart contract. When the smart contracts execute, they complete their performance deterministically in a finite amount of time.

It could be argued that the first four properties are required as a minimum, whereas the latter two may not be necessary or applicable 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, enforceable, deterministic, and secure. 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 that it is written in must be understood by both computers and people.

Ian Grigg addressed this issue of interpretation with his invention of Ricardian contracts, which we will introduce in the next section.

Smart contract templates

Smart contracts can be implemented in any industry where they are required, but the most popular use cases relate to the financial sector. This is because blockchain first found many use cases in the finance industry and, therefore, sparked enormous research interest in the financial industry long before other areas. Recent work in the smart contract space specific to the financial sector has proposed the idea of smart contract templates. The idea is to build standard templates that provide a framework to support legal agreements for financial instruments.

Christopher D. Clack et al. proposed this idea in their paper published in 2016, named *Smart Contract Templates: Foundations, design landscape and research directions*.

The paper is available at `https://arxiv.org/pdf/1608.00771.pdf`.

The paper also suggested that **domain-specific languages** (**DSLs**) should be built to support the design and implementation of smart contract templates. A language named **common language for augmented contract knowledge** (**CLACK**) has been proposed, and research has started to develop this language. This language is intended to be very rich and is expected to provide a large variety of functions ranging from supporting legal prose to the ability to be executed on multiple platforms and cryptographic functions.

Clack et al. also carried out work to develop smart contract templates that support legally-enforceable smart contracts. This proposal has been discussed in their research paper, *Smart Contract Templates: essential requirements and design options*.

This paper is available at `https://arxiv.org/pdf/1612.04496.pdf`.

The main aim of this paper is to investigate how legal prose could be linked with code using a markup language. It also covers how smart legal agreements can be created, formatted, executed, and serialized for storage and transmission. This work is ongoing and remains an open area for further research and development.

Contracts in the finance industry are not a new concept, and various DSLs are already in use in the financial services industry to provide a specific language for a particular domain. For example, there are DSLs available that support the development of insurance products, represent energy derivatives, or are being used to build trading strategies.

A comprehensive list of financial DSLs can be found at `http://www.dslfin.org/resources.html`.

It is also essential to understand the concept of DSLs, as this type of programming language can be developed to program smart contracts. DSLs are different from **general-purpose programming languages** (**GPLs**). DSLs have limited expressiveness for a particular application or area of interest. These languages possess a small set of features that are sufficient and optimized for a specific domain only. Unlike GPLs, they are not suitable for building large general-purpose application programs.

Based on the design philosophy of DSLs, it can be envisaged that such languages will be developed specifically to write smart contracts. Some work has already been done, and **Solidity** is one such language that has been introduced with the Ethereum blockchain to write smart contracts. Vyper is another language that has been recently introduced for Ethereum smart contract development.

This idea of DSLs for smart contract programming can be further extended to a **GPL**. A smart contract modeling platform can be developed where a domain expert (not a programmer but a front desk dealer, for example) can use a graphical user interface and a canvas (drawing area) to define and illustrate the definition and execution of a financial contract. Once the flow is drawn and completed, it can be emulated first to test it and then be deployed from the same system to the target platform, which can be a smart contract on a blockchain or even a complete **decentralized application** (**DApp**). This concept is also not new, and a similar approach is already used in a non-blockchain domain, in the Tibco StreamBase product, which is a Java-based system used for building event-driven, high-frequency trading systems.

It has been proposed that research should also be conducted in the area of developing high-level DSLs that can be used to program a smart contract in a user-friendly graphical user interface, thus allowing a non-programmer domain expert (for example, a lawyer) to design smart contracts.

Apart from DSLs, there is also a growing interest in using general-purpose, already established programming languages like Java, Go, and C++ to be used for smart contract programming. This idea is appealing, especially from a usability point of view, where a programmer who is already familiar with, for example, Java, can use their skills to write Java code instead of learning a new language. The high-level language code can then be compiled into a low-level bytecode for execution on the target platform. There are already some examples of such systems, such as in EOSIO blockchains, where C++ can be used to write smart contracts, which are compiled down to the web assembly for execution.

An inherent limitation with smart contracts is that they are unable to access any external data. The concept of oracles was introduced to address this issue. An oracle is an off-chain source of information that provides the required information to the smart contracts on the blockchain.

Some earlier discussions and thoughts about oracles can be found here:

`https://bitcointalk.org/index.php?topic=8821.msg133523#msg133523`

`https://github.com/orisi/wiki/wiki/Orisi-White-Paper`

`http://gavintech.blogspot.com/2014/06/bit-thereum.html`

Now we will discuss oracles in more detail.

Oracles

Oracles are an essential component of the smart contract and blockchain ecosystem. The limitation with smart contracts is that they cannot access external data because blockchains are closed systems without any direct access to the real world. This external data might be required to control the execution of some business logic in the smart contract; for example, the stock price of a security product that is required by the contract to release dividend payments. In such situations, oracles can be used to provide external data to smart contracts. An oracle can be defined as an interface that delivers data from an external source to smart contracts. Oracles are trusted entities that use a secure channel to transfer off-chain data to a smart contract.

The following diagram shows a generic model of an oracle and smart contract ecosystem:

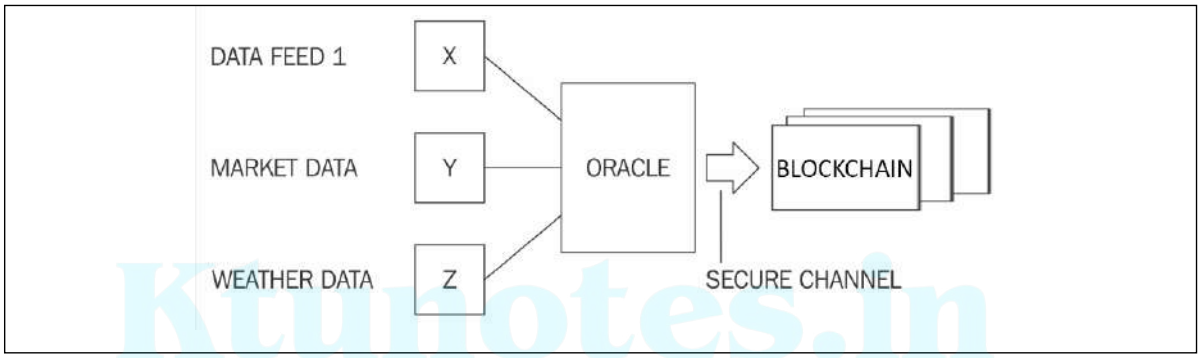


Figure 10.3: A generic model of an oracle and smart contract ecosystem

Depending on the industry and use case requirements, oracles can deliver different types of data ranging from weather reports, real-world news, and corporate actions to data coming from an **Internet of Things** (**IoT**) device.

A list of some of the common use cases of oracles is shown here:

Type of data	Examples	Use case
Market data	Live price feeds of financial instruments. Exchange rates, performance, pricing, and historic data of commodities, indices, equities, bonds, and currencies.	DApps related to financial services, for example, decentralized exchanges and **decentralized finance** (**DeFi**)
Political events	Election results	Prediction markets
Travel information	Flight schedules and delays	Insurance DApps
Weather information	Flooding, temperature, and rain data	Insurance DApps
Sports	Results of football, cricket, and rugby matches	Prediction markets
Telemetry	Hardware IoT devices, sensor data, vehicle location, and vehicle tracker data	Insurance DApps Vehicle fleet management DApps

here are different methods used by oracles to write data into a blockchain, depending on the type of blockchain used. For example, in a Bitcoin blockchain, an oracle can write data to a specific transaction, and a smart contract can monitor that transaction in the blockchain and read the data. Other methods include storing the fetched data in a smart contract's storage, which can then be accessed by other smart contracts on the blockchain via requests between smart contracts depending on the platform. For example, in Ethereum, this can be achieved by using message calls.

The standard mechanics of how oracles work is presented here:

1. A smart contract sends a request for data to an oracle.
2. The request is executed and the required data is requested from the source. There are various methods of requesting data from the source. These methods usually involve invoking APIs provided by the data provider, calling a web service, reading from a database (for example, in enterprise integration use cases where the required data may exist on a local enterprise legacy system), or requesting data from another blockchain. Sources can be any external off-chain data provider on the internet or in an internal enterprise network.
3. The data is sent to a notary to generate cryptographic proof (usually a digital signature) of the requested data to prove its validity (authenticity). Usually, TLSNotary is used for this purpose (`https://tlsnotary.org`). Other techniques include **Android proofs**, **Ledger proofs**, and **trusted hardware-assisted proofs**, which we will explain shortly.
4. The data with the proof of validity is sent to the oracle.
5. The requested data with its proof of authenticity can be optionally saved on a decentralized storage system such as Swarm or IPFS and can be used by the smart contract/blockchain for verification. This is especially useful when the proofs of authenticity are of a large size and sending them to the requesting smart contracts (storing them on the chain) is not feasible.
6. Finally, the data, with the proof of validity, is sent to the smart contract.

This process can be visualized in the following diagram:

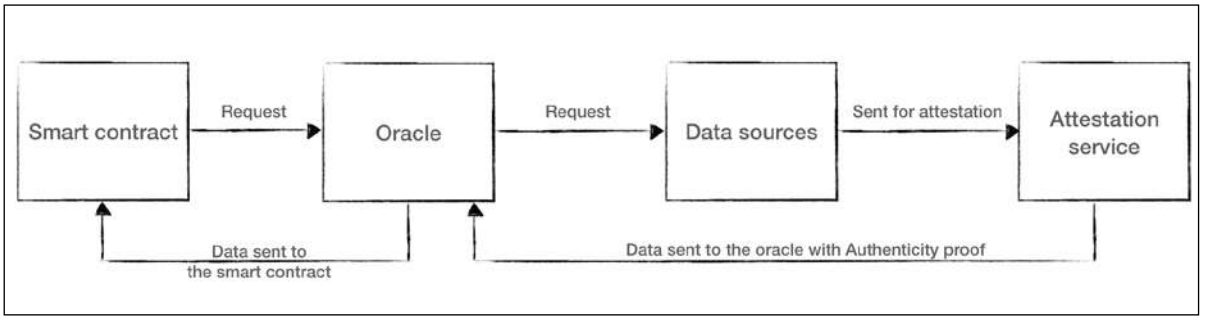


Figure 10.4: A generic oracle data flow

The preceding diagram shows the generic data flow of a data request from a smart contract to the oracle. The oracle then requests the data from the data source, which is then sent to the attestation service for notarization. The data is sent to the oracle with proof of authenticity. Finally, the data is sent to the smart contract with cryptographic proof (authenticity proof) that the data is valid.

Due to security requirements, oracles should also be capable of digitally signing or digitally attesting the data to prove that the data is authentic. This proof is called **proof of validity** or **proof of authenticity**.

Smart contracts subscribe to oracles. Smart contracts can either pull data from oracles, or oracles can push data to smart contracts. It is also necessary that oracles should not be able to manipulate the data they provide and must be able to provide factual data. Even though oracles are trusted (due to the associated proof of authenticity of data), it may still be possible that, in some cases, the data is incorrect due to manipulation or a fault in the system. Therefore, oracles must not be able to modify the data. This validation can be provided by using various cryptographic proofing schemes. We will now introduce different mechanisms to produce cryptographic proof of data authenticity.

Software and network-assisted proofs

As the name suggests, these types of proofs make use of software, network protocols, or a combination of both to provide validity proofs. One of the prime examples of such proofs is TLSNotary, which is a technology developed to be primarily used in the PageSigner project (`https://tlsnotary.org/pagesigner.html`) to provide web page notarization. This mechanism can also be used to provide the required security services to oracles.

TLSNotary

This protocol provides a piece of irrefutable evidence to an auditor that specific web traffic has occurred between a client and a server. It is based on **Transport Layer Security** (**TLS**), which is a standard security mechanism that enables secure, bidirectional communication between hosts. It is extensively used on the internet to secure websites and allow HTTPS traffic.

A discussion of the internals of this protocol is beyond the scope of this book. Interested readers can read the standards document at `http://www.ietf.org/rfc/rfc2246.txt`. The link provided is only for TLS version 1.0 as TLSNotary only supports TLS version 1.0 or 1.1.

The key idea behind using TLSNotary is to utilize the TLS handshake protocol's feature, which allows the splitting of the TLS master key into three parts. Each part is allocated to the server, the auditee, and the auditor. The oracle service provider (`https://provable.xyz`) becomes the auditee, whereas an **Amazon Web Services** (**AWS**) instance, which is secure and locked down, serves as the auditor.

In summary, to prove the authenticity of the data retrieved by oracles from external sources, attestation mechanisms such as TLSNotary are used, which produce verifiable and auditable proofs of communication between the data source and the oracle. This proof of authenticity ensures that the data fed back to the smart contract is indeed retrieved from the source.

TLS-N based mechanism

This mechanism is one of the latest developments in this space. TLS-N is a TLS extension that provides secure non-repudiation guarantees. This protocol allows you to create privacy-preserving and non-interactive proofs of the content of a TLS session. TLS-N based oracles do not need to trust any third-party hardware such as Intel SGX or TLSNotary type service to provide authenticity proofs of data web content (data) to the blockchain. In contrast to TLSNotary, this scheme works on the latest TLS 1.3 standard, which allows for improved security. More information regarding this protocol is available at `https://eprint.iacr.org/2017/578.pdf`.

Hardware device-assisted proofs

As the name suggests, these proofs rely on some hardware elements to provide proof of authenticity. In other words, they require specific hardware to work. Different mechanisms come under this category, and we will briefly introduce those next.

Android proof

This proof relies on Android's SafetyNet software attestation and hardware attestation to create a provably secure and auditable device. SafetyNet validates that a genuine Android application is being executed on a secure, safe, and untampered hardware device. Hardware attestation validates that the device has the latest version of the OS, which helps to prevent any exploits that existed due to vulnerabilities in the previous versions of the OS. This secure device is then used to fetch data from third-party sources, ensuring tamper-proof HTTPS connections. The very use of a provably secure device provides the guarantee and confidence (that is, a proof of authenticity) that the data is authentic.

More information regarding SafetyNet and hardware attestation is available here:

`https://developer.android.com/training/safetynet`

`https://developer.android.com/training/articles/security-key-attestation.html`

Ledger proof

Ledger proof relies on the hardware cryptocurrency wallets built by the ledger company (`https://www.ledger.com`). Two hardware wallets, **Ledger Nano S** and **Ledger Blue**, can be used for these proofs. The primary purpose of these devices is as secure hardware cryptocurrency wallets. However, due to the security and flexibility provided by these devices, they also allow developers to build other applications for this hardware. These devices run a particular OS called **Blockchain Open Ledger Operating System** (**BOLOS**), which, via several kernel-level APIs, allows device and code attestation to provide a provably secure environment.

The secure environment provided by the device can also prove that the applications that may have been developed by oracle service providers and are running on the device are valid, authentic, and are indeed executing on the **Trusted Execution Environment** (**TEE**) of the ledger device. This environment, supported by both code and device attestation, provides an environment that allows you to run a third-party application in a secure and verifiable ledger environment to provide proof of data authenticity. Currently, this service is used by Provable, an oracle service, to provide untampered random numbers to smart contracts.

Ledger proofs, Android proofs, and TLSNotary proofs are used in provable oracles. The official documentation for these methods can be found here: `http://docs.provable.xyz`.

Currently, as these devices do not connect to the internet directly, the ledger devices cannot be used to fetch data from the internet.

Trusted hardware-assisted proofs

This type of proof makes use of trusted hardware, such as TEEs. A prime example of such a hardware device is Intel SGX. A general approach that is used in this scenario is to rely on the security guarantees of a secure and trusted execution provided by the secure element or enclave of the TEE device.

A prime example of a trusted hardware-assisted proof is Town Crier (`https://www.town-crier.org`), which provides an authenticated data feed for smart contracts. It uses Intel SGX to provide a security guarantee that the requested data has come from an existing trustworthy resource.

Intel SGX technology is developed by Intel, which provides a hardware TEE. More information about this is available here: `https://software.intel.com/en-us/sgx`. It is used by many different oracle service providers such as Town Crier and iExec (`https://iex.ec/decentralized-oracles/`).

Town Crier also provides a confidentiality service, which allows you to run confidential queries. The query for the data request is processed inside SGX Enclave, which provides a trusted execution guarantee, and the requested data is transmitted using a TLS-secured network connection, which provides additional data integrity guarantees.

It should be noted that in all of the proof techniques mentioned earlier, there are many types of resources used to provide security guarantees, including hardware, network, and software. The categorization provided here is based on the principal element, either hardware or software, which plays a critical role in the overall security mechanism to provide security.

An issue can already be seen here, and that is the issue of trust. With oracles, we are effectively trusting a third party to provide us with the correct data. What if these data sources turn malicious, or simply due to a fault start provide incorrect data to the oracles? What if the oracle itself fails or the data source stops sending data? This issue can then damage the whole blockchain trust model. This phenomenon is called the **Blockchain oracle problem**. How do you trust a third party about the quality and authenticity of the data they provide? This question is especially real in the financial world, where market data must be accurate and reliable.

There are several proposed ways to overcome this issue. These solutions range from merely trusting a reputable third party to decentralized oracles. We have discussed some of the attestation techniques earlier; however, a third party due to a genuine fault or a malicious intent may still provide data that is incorrect. Even if it is attested later on, the actual data itself is not guaranteed to be accurate. It might be acceptable for a smart contract designer in a use case to accept data for an oracle that is provided by a large, reputable, and trusted third party. For example, the source of the data can be from a reputable weather reporting agency or airport information system directly relaying the flight delays, which can give some level of confidence. However, the issue of centralization remains.

The blockchain oracle problem can be defined formally as the conflict of trust between presumably trusted Oracles (trusted third-party data sources) and completely trustless blockchain.

Based on the evolution of blockchain over the last few years, several types of blockchain oracles have emerged. We informally provided some background on these earlier, but now we will define these formally.

Types of blockchain oracles

There are various types of blockchain oracles, ranging from simple software oracles to complex hardware assisted and decentralized oracles. Broadly speaking, we can categorize oracles into two categories: inbound oracles and outbound oracles. The following section will examine some of these in more detail.

Inbound oracles

This class represents oracles that receive incoming data from external services, and feed it into the smart contract. We will shortly discuss software, hardware, and several other types of inbound oracle.

Software oracles

These oracles are responsible for acquiring information from online services on the Internet. This type of oracle is usually used to source data such as weather information, financial data (stock prices, for example), travel information and other types of data from third-party providers. The data source can also be an internal enterprise system, which may provide some enterprise-specific data. These types of oracle can also be called standard or simple oracles.

Hardware oracles

This type of oracle is used to source data from hardware sources such as IoT devices or sensors. This is useful in use cases such as insurance-related smart contracts where telemetry sensors provide certain information, for example, vehicle speed and location. This information can be fed into the smart contract dealing with insurance claims and payouts to decide whether to accept a claim or not. Based on the information received from the source hardware sensors, the smart contract can decide whether to accept or reject the claim.

These oracles are useful in any situation where real-world data from physical devices is required. However, this approach requires a mechanism in which hardware devices are tamper-proof or tamper-resistant. This level of security can be achieved by providing cryptographic evidence (non-repudiation and integrity) of IoT device's data and an anti-tampering mechanism on the IoT device, which renders the device useless in case of tampering attempts.

Computation oracles

These oracles allow computing-intensive calculations to be performed off-chain. As blockchain is not suitable for performing compute-intensive operations, a blockchain (that is, a smart contract on a blockchain) can request computations to be performed on off-chain high-performance computing infrastructure and get the verified results back via an oracle. The use of oracle, in this case, provides data integrity and authenticity guarantees.

An example of such an oracle is Truebit (`https://truebit.io`). It allows a smart contract to submit computation tasks to oracles, which are eventually completed by miners in return for an incentive.

Aggregation based oracles

In this scenario, a single value is sourced from many different feeds. As an example, this single value can be the price of a financial instrument, and it can be risky to rely upon only one feed. To mitigate this problem, multiple data providers can be used where all of these feeds are inspected, and finally, the price value that is reported by most of the feeds can be picked up. The assumption here is that if the majority of the sources reports the same price value, then it is likely to be correct. The collation mechanism depends on the use case: sometimes it's merely an average of multiple values, sometimes a median is taken of all the values, and sometimes it is the maximum value. Regardless of the aggregation mechanism, the essential requirement here is to get the value that is valid and authentic, which eventually feeds into the system.

An excellent example of price feed oracles is MakerDAO (`https://makerdao.com/en/`) price feed oracle (`https://developer.makerdao.com/feeds/`), which collates price data from multiple external price feed sources and provides a median ETHUSD price to MakerDAO.

Crowd wisdom driven oracles

This is another way that the blockchain oracle problem can be addressed where a single source is not trusted. Instead, multiple public sources are used to deduce the most appropriate data eventually. In other words, it solves the problem where a single source of data may not be trustworthy or accurate as expected. If there is only one source of data, it can be unreliable and risky to rely on entirely. It may turn malicious or become genuinely faulty.

In this case, to ensure the credibility of data provided by third-party sources for oracles, the data is sourced from multiple sources. These sources can be users of the system or even members of the general public who have access to and have knowledge of some data, for example, a political event or a sporting event where members of the public know the results and can provide the required data. Similarly, this data can be sourced from multiple different news websites. This data can then be aggregated, and if a sufficiently high number of the same information is received from multiple sources, then there is an increased likelihood that the data is correct and can be trusted.

Decentralized oracles

Another type of oracles, which primarily emerged due to the decentralization requirements, is called **decentralized** oracles. Remember that in all types of oracles discussed so far, there are some trust requirements to be placed in a trusted third party. As blockchain platforms such as Bitcoin and Ethereum are fully decentralized, it is expected that oracle services should also be decentralized. This way, we can address the *Blockchain Oracle Problem*.

This type of oracle can be built based on a distributed mechanism. It can also be envisaged that the oracles can find themselves source data from another blockchain, which is driven by distributed consensus, thus ensuring the authenticity of data. For example, one institution running their private blockchain can publish their data feed via an oracle that can then be consumed by other blockchains.

A decentralized oracle essentially allows off-chain information to be transferred to a blockchain without relying on a trusted third party.

Augur (visit `https://www.augur.net/whitepaper.pdf` for Jack Peterson et al.'s essay, *A Decentralized Oracle and Prediction Market Platform*) is a prime example of such type of oracles. The Augur white paper is also available here: `https://arxiv.org/abs/1501.01042`.

The core idea behind Augur's oracle is that of crowd wisdom-supported oracles, in which the information about an event is acquired from multiple sources and aggregated into the most likely outcome. The sources in case of Augur are financially motivated reporters who are rewarded for correct reporting and penalized for incorrect reporting.

Decentralized, crowd wisdom based and aggregation supported oracles can be categorized into a broader category of oracles called "consensus driven oracles". Augur is based on Crowd Wisdom based oracle.

Smart oracles

An idea of smart oracle has also been proposed by **Ripple labs** (**codius**). Its original whitepaper is available at `https://github.com/codius/codius-wiki/wiki/White-Paper#from-oracles-to-smart-oracles`. Smart oracles are entities just like oracles, but with the added capability of executing contract code. Smart oracles proposed by Codius run using Google Native Client, which is a sandboxed environment for running untrusted x86 native code.

Outbound oracles

This type, also called **reverse oracles**, are used to send data out from the blockchain smart contracts to the outside world. There are two possible scenarios here; one is where the source blockchain is a producer of some data such as blockchain metrics, which are needed for some other blockchain. The actual data somehow needs to be sent out to another blockchain smart contract. The other scenario is that an external hardware device needs to perform some physical activity in response to a transaction on-chain. However, note that this type of scenario does not necessarily need an oracle, because the external hardware device can be sent a signal as a result of the smart contract event.

On the other hand, it can be argued that if the hardware device is running on an external blockchain, then to get data from the source chain to the target chain, undoubtedly, will need some security guarantees that oracle infrastructure can provide. Another situation is where we need to integrate legacy enterprise systems with the blockchain. In that case, the outbound oracle would be able to provide blockchain data to the existing legacy systems. An example scenario is the settlement of a trade done on a blockchain that needs to be reported to the legacy settlement and backend reporting systems.

Now that we have discussed different types of oracles, we now introduce different service providers that provide these services. Several service providers provide oracle services for blockchain, some of these we introduce following.

Deploying smart contracts

Smart contracts may or may not be deployed on a blockchain, but it makes sense to do so on a blockchain due to the security and decentralized consensus mechanism provided by the blockchain. Ethereum is an example of a blockchain platform that natively supports the development and deployment of smart contracts. We will cover Ethereum in more detail later in this book, in *Chapter 11*, *Ethereum 101*. Smart contracts on an Ethereum blockchain are typically part of a broader DApp.

In comparison, in a Bitcoin blockchain, the transaction timelocks, such as the `nLocktime` field, the `CHECKLOCKTIMEVERIFY` (**CLTV**), and the `CHECKSEQUENCEVERIFY` script operator in the Bitcoin transaction, can be seen as an enabler of a simple version of a smart contract. These timelocks enable a transaction to be locked until a specified time or until a number of blocks, thus enforcing a basic contract that a certain transaction can only be unlocked if certain conditions (elapsed time or number of blocks) are met. For example, you can implement conditions such as *Pay party X, N number of bitcoins after 3 months*. However, this is very limited and should only be viewed as an example of a basic smart contract. In addition to the example mentioned earlier, Bitcoin scripting language, though limited, can be used to construct basic smart contracts. One example of a basic smart contract is to fund a Bitcoin address that can be spent by anyone who demonstrates a **hash collision attack**.

This was a contest that was announced on the Bitcointalk forum where bitcoins were set as a reward for whoever manages to find hash collisions (we discussed this concept in *Chapter 3*, *Symmetric Cryptography*) for hash functions. This conditional unlocking of Bitcoin solely on the demonstration of a successful attack is a basic type of smart contract.

This idea was presented on the Bitcointalk forum, and more information can be found at `https://bitcointalk.org/index.php?topic=293382.0`. This can be considered a basic form of a smart contract.

Various other blockchain platforms support smart contracts such as Monax, Lisk, Counterparty, Stellar, Hyperledger Fabric, Axoni core, Neo, EOSIO, and Tezos. Smart contracts can be developed in various languages, either DSLs or general-purpose languages. The critical requirement, however, is determinism, which is very important because it is vital that regardless of where the smart contract code executes, it produces the same result every time and everywhere. This requirement of the deterministic nature of smart contracts also implies that smart contract code is absolutely bug-free. Validation and verification of smart contracts is an active area of research and a detailed discussion of this topic will be presented in *Chapter 21*, *Scalability and Other Challenges*.

Various languages have been developed to build smart contracts such as Solidity, which runs on **Ethereum Virtual Machine** (**EVM**). It's worth noting that there are platforms that already support mainstream languages for smart contract development, such as Lisk, which supports JavaScript. Another prominent example is Hyperledger Fabric, which supports Golang, Java, and JavaScript for smart contract development. A more recent example is EOSIO, which supports writing smart contracts in C++.

Security is of paramount importance for smart contracts. However, there are many vulnerabilities discovered in prevalent blockchain platforms and relevant smart contract development languages. These vulnerabilities result in some high-profile incidents, such as the DAO attack.

The DAO

The **Decentralized Autonomous Organization** (**DAO**), started in April 2016, was a smart contract written to provide a platform for investment. Due to a bug, called the **reentrancy bug**, in the code, it was hacked in June 2016. An equivalent of approximately 3.6 million ether (roughly 50 million US dollars) was siphoned out of the DAO into another account.

Even though the term hacked is used here, it was not really hacked. The smart contract did what it was asked to do but due to the vulnerabilities in the smart contracts, the attacker was able to exploit it. It can be seen as an unintentional behavior (a bug) that programmers of the DAO did not foresee. This incident resulted in a hard fork on the Ethereum blockchain, which was introduced to recover from the attack.

The DAO attack exploited a vulnerability (reentrancy bug) in the DAO code where it was possible to withdraw tokens from the DAO smart contract repeatedly before giving the DAO contract a chance to update its internal state to indicate that how many DAO tokens have been withdrawn. The attacker was able to withdraw DAOs. However, before the smart contract could update its state, the attacker withdrew the tokens again. This process was repeated many times, but eventually, only a single withdrawal was logged by the smart contract, and the contract also lost record of any repeated withdrawals.

The notion of *code is the law or unstoppable smart contracts* should be viewed with some skepticism as the implementation of these concepts is still not mature enough to deserve complete and unquestionable trust. This is evident from the events after the DAO incident, where the Ethereum foundation was able to stop and change the execution of the DAO by introducing a hard fork on the Ethereum blockchain. Though this hard fork was introduced for genuine reasons, it goes against the true spirit of decentralization, immutability, and the notion that *code is the law*. Subsequently, resistance against this hard fork resulted in the creation of Ethereum Classic, where a large number of users decided to keep mining on the old chain. This chain is the original, non-forked Ethereum blockchain that still contains the DAO. It can be said that on this chain, *the code is still the law*.

There are some interesting message threads and announcements related to this event, which readers may find informative and entertaining:

- An open letter from *The Attacker* of the DAO: `https://pastebin.com/CcGUBgDG`
- Announcement from Ethereum core dev: `https://twitter.com/avsa/status/745313647514226688`
- Reddit discussion on the hard fork, where points made in favor and against can be seen: `https://www.reddit.com/r/ethereum/comments/4sgihm/ethcore_blog_post_in_support_of_a_hard_fork/`
- Hard fork specification: `https://blog.slock.it/hard-fork-specification-24b889e70703`

The DAO attack highlights the dangers of not formally and thoroughly testing smart contracts. It also highlights the absolute need to develop a formal language for the development and verification of smart contracts. The attack also highlighted the importance of thorough testing to avoid the issues that the DAO experienced. There have been various vulnerabilities discovered in Ethereum over the last few years regarding the smart contract development language. Therefore, it is of utmost importance that a standard framework is developed to address all these issues.

Some work has already begun, for example, an online service at `https://securify.ch`, which provides tools to formally verify smart contract code.

Another example is Michelson (`https://www.michelson.org`) for writing smart contracts in the Tezos blockchain. It is a functional programming language suitable for formal verification. We will introduce the Tezos blockchain in more detail in this book's bonus content pages, here `https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf`.

Vyper is another language that aims to provide a secure language for developing smart contracts for EVM. It aims for goals such as security, simplicity, and audibility. It is a strongly typed language with support for overflow checking and signed integers. All these features make Vyper a reasonable choice for writing secure smart contracts.

Even though many different initiatives are aiming to explore and address the security of smart contracts, this field still requires further research to address limitations in smart contract programming languages.

The security of smart contracts is an area of deep interest for researchers. A major area of interest is the formal verification of smart contracts. We will discuss these topics further in *Chapter 21*, *Scalability and Other Challenges*.

Government

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

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

Border control

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

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

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

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

Blockchain technology can provide a solution to this problem by maintaining a blacklist in a smart contract that can be updated as required. Any changes will be immediately visible to all agencies and border control points, thus enabling immediate control over the movement of a suspected travel document. It could be argued that traditional mechanisms like **Public Key Infrastructures** (**PKIs**) and P2P networks can also be used for this purpose, but they do not provide the benefits that a blockchain can provide. With blockchain, the whole system can be simplified without the requirement of complex networks and PKI setups, which will also result in cost reduction. Moreover, blockchain-based systems will provide cryptographically guaranteed immutability, which helps with auditing and discourages any fraudulent activity.

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

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

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

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

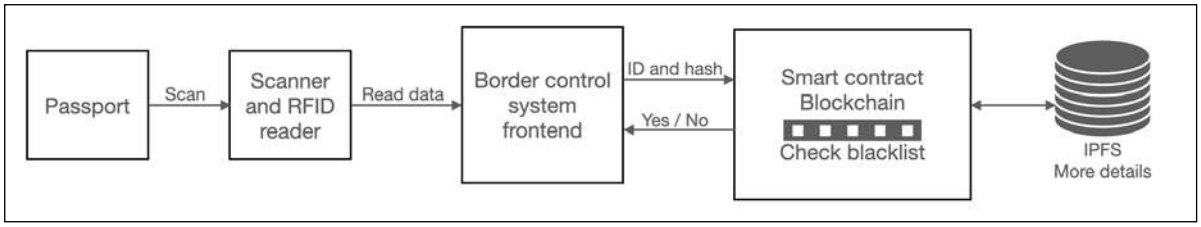


Figure 19.24: Automated border control using blockchain

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

Health

The health industry has also been identified as another major industry that can benefit by adapting blockchain technology. Blockchain can provide an immutable, auditable, and transparent system that traditional P2P networks cannot. Also, blockchain provides a simpler, more cost-effective infrastructure compared to traditional complex PKI networks. In healthcare, major issues such as privacy compromises, data breaches, high costs, and fraud can arise from a lack of interoperability, overly complex processes, transparency, auditability, and control. Another burning issue is counterfeit medicines; especially in developing countries, this is a major cause of concern.

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

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

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

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

In the next section, we'll explore one of the most talked about and anticipated industries that can benefit from blockchain technology: finance.

Finance

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

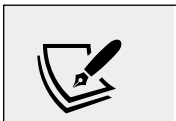
These applications include, but are not limited to, insurance, post-trade settlements, financial crime prevention, and payments.

Insurance

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

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

Several start-ups, such as **Dynamis**, have proposed smart contract-based P2P insurance platforms that run on the Ethereum blockchain. This was initially proposed to be used for unemployment insurance and does not require underwriters in the model.



The Dynamis white paper is available at `http://dynamisapp.com/`.

Post-trade settlement

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

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

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

Notably, the post-trade settlement process usually takes two to three days, and has a dependency on central clearing houses and reconciliation systems. With the shared ledger approach, all participants on the blockchain can immediately see a single version of truth regarding the state of the trade. Moreover, P2P settlement is possible, which results in the reduction of complexity, cost, risk, and the time it takes to settle the trade. Finally, intermediaries can be eliminated by making use of the appropriate smart contracts on the blockchain. Also, regulators can view the blockchain for auditing and regulatory requirements.

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

Financial crime prevention

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

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

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

There are already several solutions available, two of which are listed here:

Crypto-kyc: `https://www.crypto-kyc.com`

KYC-Chain: `https://kyc-chain.com`

There are many other solutions that can be found with a simple search for `Blockchain KYC` on an internet search engine.

Payments

A payment is a transfer of money or its equivalent from one party (the payer) to another (the payee) in exchange for services, goods, or for fulfilling a contract. Payments are usually made in the form of cash, bank transfers, credit cards, and cheques. There are various electronic payment systems in use, such as **Bacs Payment Schemes Limited** (**Bacs**) and the **Clearing House Automated Payment System** (**CHAPS**).

All of these systems are, however, centralized and governed by traditional financial service industry codes and practices. These systems work adequately, but with the advent of blockchain, the potential of technology has arisen to address some of these limitations.

Some of the key advantages that blockchain technology can bring to payments are listed as follows.

Decentralization

Decentralization means that there is no requirement of a trusted third party to process payments. Payments can be made directly between parties without requiring any intermediary. This can result in reduced cost and faster (direct) payments between parties.

Faster settlement

Settlement can be much quicker compared to the traditional network due to the active presence of all parties on the network. Payment data can be shared and seen by all parties at the same time, and due to this settlement becomes quicker and more comfortable. Moreover, there is no requirement of running lengthy reconciliation processes because the data is all there on the blockchain, shared between all parties and readily available, which removes the requirement of the lengthy reconciliation process.

Better resilience

With a payment system running on a blockchain with potentially thousands of nodes around the world, the network becomes naturally resilient. It could also be argued that, with blockchain payments, there is no downtime because blockchain does not rely on traditional **disaster recovery** (**DR**) practices and is also better protected against malicious and **denial of service** attacks.

With all these advantages, it is easy to see how the payments industry can benefit from blockchain technology.

There is also another branch of payments that deals with international or cross-border payments, and comes with its own challenges. We'll discuss this next.

Cross-border payments

In traditional finance, cross-border payment is a complex process that can take days to process and involves multiple intermediaries. Current mechanisms suffer from delays incurred by multiple intermediaries, enforcement of regulations, differences in terms of regulations between different jurisdictions… the list goes on. All of these issues can be addressed by utilizing blockchain technology. The most significant advantage is decentralization, where, due to the lack of requirement of intermediaries, payments can be made directly between businesses or individuals. Also, due to P2P connectivity, the whole process becomes a lot faster—almost immediate, in fact—which results in more productivity and business agility.

Peer-to-peer loans

Blockchain also enables P2P loans, where lenders and borrowers can deal with each other directly instead of relying on a third party. Currently, in the **decentralized finance** (**DeFi**) landscape, there is a large percentage of lending **DApps** that deal with the lending and borrowing of crypto tokens.

Migration planning

This phase creates and finalizes the comprehensive implementation plan for migrating to the target architecture from the current architecture.

Implementation governance

This phase deals with the implementation of the target architecture. A strategy to govern the overall deployment and migration is developed here. We can also perform blockchain solution testing and devise a deployment strategy during this phase.

Architecture change management

This phase is responsible for creating change management guidelines and procedures for the newly implemented target architecture. From a blockchain perspective, this phase can provide change management procedures for the newly implemented enterprise blockchain solut

With this, we've completed our introduction to TOGAF and explored the idea that enterprise blockchain solutions should be viewed through the lens of the enterprise architecture.

The fundamental idea to understand here is that enterprise blockchain solutions are not merely a matter of quickly spinning up a network, creating a few smart contracts, creating a web frontend, and hoping that it will solve business problems. This setup may be useful as a PoC or for an incredibly simple use case but is certainly not an enterprise solution. We suggest that an enterprise blockchain solution must be looked through the lens of the enterprise architecture and regarded as a full grade enterprise solution. This is so that we can effectively achieve the business goals intended to be solved by enterprise blockchain solutions.

Next, let's explore how we could implement a blockchain business solution in an organization that has moved its operations to the cloud.

Blockchain in the cloud

Cloud computing provides excellent benefits to enterprises, including efficiency, cost reduction, scalability, high availability, and security. Cloud computing delivers computing services such as infrastructure, servers, databases, and software over the internet. There are different types of cloud services available; a standard comparison is made between **Infrastructure as a Service** (**IaaS**), **Platform as a Service** (**PaaS**), and **Software as a Service** (**SaaS**). A question arises here: where does blockchain fit in?

Blockchain as a Service, or **BaaS**, is an extension of SaaS, whereby a blockchain platform is implemented in the cloud for an organization. The organization manages its applications on the blockchain, and the rest of the software management, infrastructure management, and other aspects such as security and operating systems are managed by the cloud provider. This means that the blockchain's software and infrastructure are provided and maintained by the cloud provider. The customer or enterprise can focus on their business applications without worrying about other aspects of the infrastructure.

The following is a comparison of different approaches:

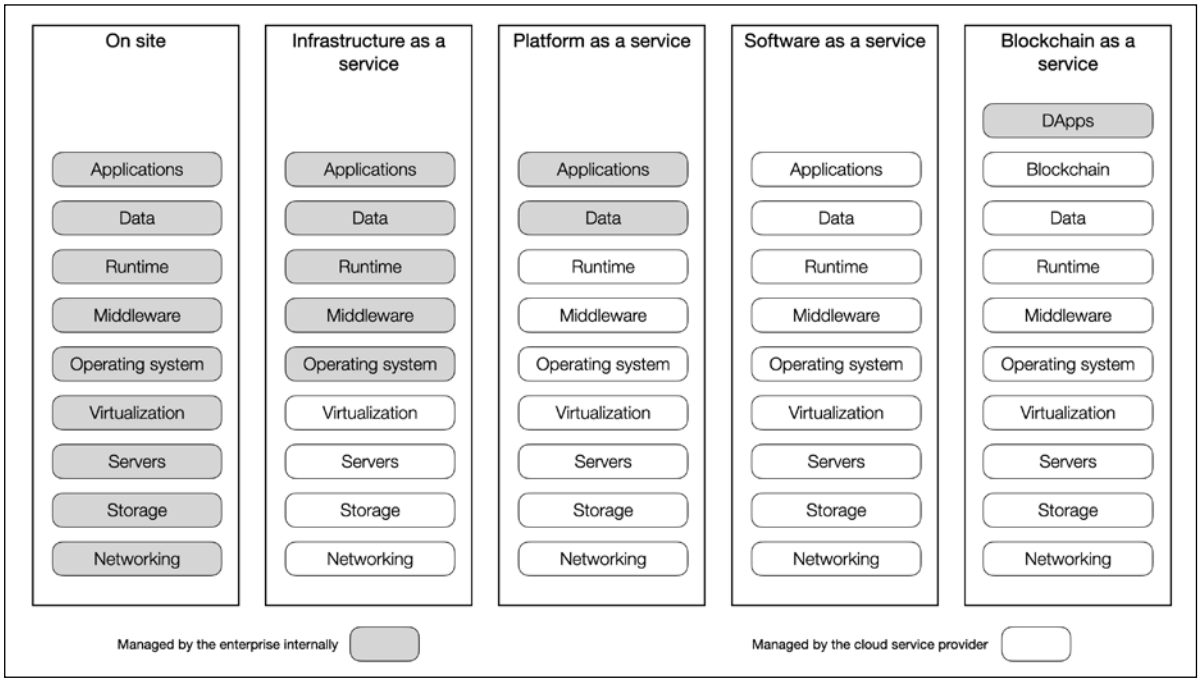


Figure 20.4: Cloud solutions

BaaS can be thought of as a SaaS, where the software is a blockchain. In this case, just like in SaaS, all services are externally managed. In other words, customers get a fully managed blockchain network on which they can build and manage their own **DApps**. Note that in the preceding diagram, under the **Blockchain as a Service** column, **Applications** have been replaced with **Blockchain**, as a differentiator between other cloud services and BaaS. Here, blockchain is the software (application) provided and managed by the cloud service provider. Also, note that **DApps** have been added on top, which are managed by the enterprise internally.

There are many BaaS providers. A few of them are listed as follows:

- **AWS**: `https://aws.amazon.com/blockchain/`
- **Azure**: `https://azure.microsoft.com/en-gb/solutions/blockchain/`
- **Oracle**: `https://www.oracle.com/uk/application-development/cloud-services/blockchain-platform/`
- **IBM**: `https://www.ibm.com/uk-en/cloud/blockchain-platform`

In the next section, we'll introduce some enterprise blockchain platforms.

Blockchain and AI

It is envisaged that other technologies, such as IoT and AI, will converge for the mutual benefit and wider adoption of both blockchain and the other given technology.

The convergence of blockchain with IoT has been discussed at length in *Chapter 21*, *Scalability and Other Challenges*. Briefly, it can be said that due to blockchain's authenticity, integrity, privacy, and shared nature, IoT networks would benefit greatly from making use of blockchain technology. This can be realized in the form of an IoT network that runs on a blockchain, and makes use of a decentralized **mesh network** for communication in order to facilitate **Machine-to-Machine** (**M2M**) communication in real time.

A mesh network is a network topology that allows all nodes on a network to connect with one another in a cooperative and dynamic fashion, to facilitate the efficient routing of data.

All of the data that is generated as a result of M2M communication can be used in **machine learning** processes to augment the functionality of artificially intelligent DAOs or simple AAs. These AAs can act as agents in a blockchain-provided **Distributed Artificial Intelligence** (**DAI**) environment, which can learn over time using machine learning processes. This would enable them to make better decisions for the good of the blockchain.

AI is a field of computer science that endeavors to build intelligent agents that can make rational decisions based on the scenarios and environment that they observe around them. Machine learning plays a vital role in AI technology, by making use of raw data as a learning resource. A key requirement in AI-based systems is the availability of authentic data that can be used for machine learning and model building. Therefore, the explosion of data coming out of IoT devices, smartphones, and other means of data acquisition means that AI and machine learning is becoming more and more powerful. There is, however, a requirement for data authenticity, which is where the convergence with blockchain comes in. Once consumers, producers, and other entities are on a blockchain, the data that is generated as a result of interaction between these entities can be readily used as an input to machine learning engines with a guarantee of authenticity.

It could also be argued that if an IoT device is hacked, it could send malformed data to the blockchain. This issue would be mitigated using blockchain technology, because an IoT device would be part of the blockchain (as a node) and would have the same security properties applied to it as a standard node in the blockchain network. These properties include the incentivization of good behavior, rejection of malformed transactions, strict verification of transactions, and various other checks that are part of blockchain protocol. Therefore, even if an IoT device is hacked, it would be treated as a **Byzantine node** by the blockchain network and would not cause any adverse impact on the network.

The possibility of combining intelligent oracles, intelligent smart contracts, and AAs will give rise to **Artificially Intelligent Decentralized Autonomous Organizations** (**AIDAOs**) that can act on behalf of humans to run entire organizations on their own. This is another side of AI that could potentially become normal in the future. However, more research is required to realize this vision.

The convergence of blockchain technology with various other fields, such as 3D printing, virtual reality, augmented reality, spatial computing, and the gaming industry, is also envisaged. For example, in a multiplayer online game, blockchain's decentralized approach allows more transparency, and can ensure that no central authority is gaining an unfair advantage by manipulating game rules. Each of these topics are currently active areas of research, and more interest and development is expected.

Now, let's discuss the future of blockchain technology, and make some predictions about its development.

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, based on the requirements and usage. These types are described in the following subsections.

The mainnet

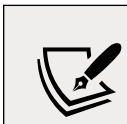
The **mainnet** is the current live network of Ethereum. Its network ID is 1 and its chain ID is also 1. The network and chain IDs are used to identify the network. A block explorer that shows detailed information about blocks and other relevant metrics is available at `https://etherscan.io`. This can be used to explore the Ethereum blockchain.

Testnets

There is a number of testnets available for Ethereum testing. The aim of these test blockchains is to provide a testing environment for smart contracts and DApps before being deployed to the production live blockchain. Moreover, being test networks, they also allow experimentation and research. The main testnet is called *Ropsten*, which contains all the features of other smaller and special-purpose testnets that were created for specific releases. For example, other testnets include *Kovan* and *Rinkeby*, which were developed for testing Byzantium releases. The changes that were implemented on these smaller testnets have also been implemented in Ropsten. Now the Ropsten test network contains all properties of Kovan and Rinkeby.

Private nets

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



A table detailing some major Ethereum networks, including their network IDs and other relevant details, can be found in this book's online resource page here: `https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf`

Network IDs and chain IDs are used by Ethereum clients to identify the network. Chain ID was introduced in EIP155 as part of replay protection mechanism. EIP155 is detailed at `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md`.

A comprehensive list of Ethereum networks is maintained and available at `https://chainid.network`.

More discussion on how to connect to a testnet and how to set up private nets will be had in *Chapter 13*, *Ethereum Development Environment*.

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.

The overall Ethereum ecosystem architecture is visualized in the following diagram:

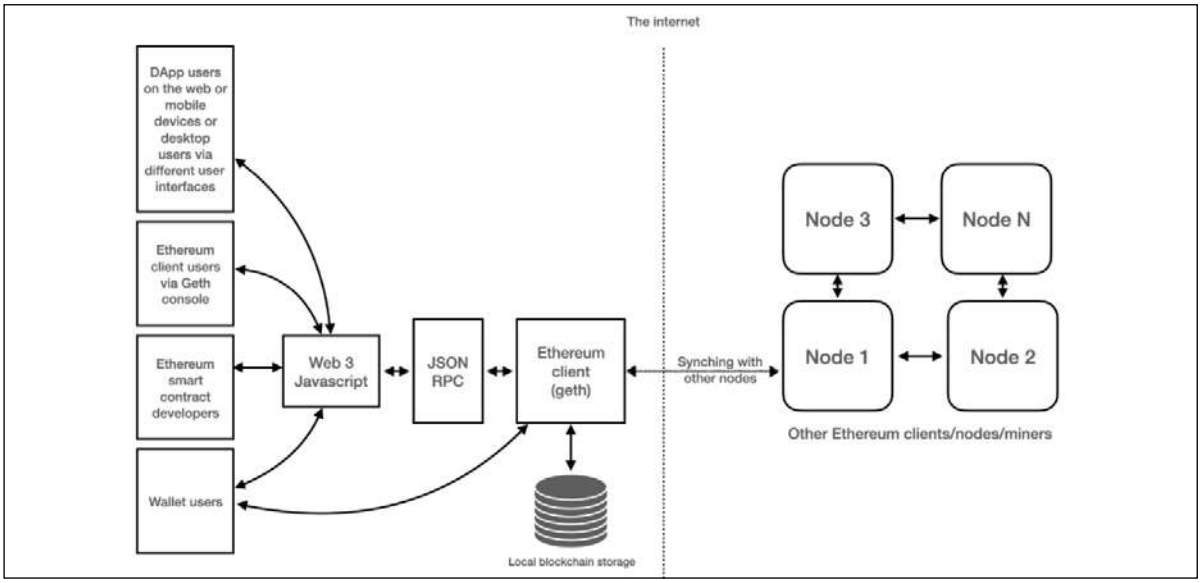


Figure 11.6: Ethereum high-level ecosystem

A list of elements present in the Ethereum blockchain is presented here:

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

In the following sections, we will discuss each of these one by one. Other components such wallets, clients, development tools, and environments will be discussed in the upcoming chapters.

We will also discuss relevant technical concepts related to a high-level element within that section. First, let's have a look at keys and addresses.

Keys and addresses

Keys and addresses are used in the Ethereum blockchain to represent ownership and transfer ether. The keys used are made up of pairs of private and public parts. The private key is generated randomly and is kept secret, whereas a public key is derived from the private key. Addresses are derived from public keys and are 20-byte codes used to identify accounts.

The process of key generation and address derivation is as follows:

1. First, a private key is randomly chosen (a 256-bit positive integer) under the rules defined by the elliptic curve `secp256k1` specification (in the range [`1`, `secp256k1n - 1`]).
2. The public key is then derived from this private key using the **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) recovery function. We will discuss this in the following *Transactions and messages* section, in the context of digital signatures.
3. An address is derived from the public key, specifically, from the rightmost 160 bits of the Keccak hash of the public key.

An example of how keys and addresses look in Ethereum is shown as follows:

- A private key:

  `b51928c22782e97cca95c490eb958b06fab7a70b9512c38c36974f47b954ffc4`

- A public key:

  ```
  3aa5b8eefd12bdc2d26f1ae348e5f383480877bda6f9e1a47f6a4afb35cf998ab847
  f1e3948b1173622dafc6b4ac198c97b18fe1d79f90c9093ab2ff9ad99260
  ```

- An address:

  `0x77b4b5699827c5c49f73bd16fd5ce3d828c36f32`

The preceding private key is shown here only as an example, and should not be reused.

Another key element in Ethereum is an account, which is required in order to interact with the blockchain. It either represents a user or a smart contract.

Accounts

Accounts are one of the main building blocks of the Ethereum blockchain. They are defined by pairs of private and public keys. Accounts are used by users to interact with the blockchain via transactions. A transaction is digitally signed by an account before submitting it to the network via a node. Ethereum, being a *transaction-driven state machine*, the state is created or updated as a result of the interaction between accounts and transaction executions. All accounts have a state that, when combined together, represents the state of the Ethereum network. With every new block, the state of the Ethereum network is updated. 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, the sender's account balance is checked and subtracted accordingly, and the nonce is incremented. An error is returned if the account balance is insufficient.
3. Provide enough ETH (the gas price) to cover the cost of the transaction. We will cover gas and relevant concepts shortly in this chapter. This is charged per byte and is 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 the 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 the 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 the fee is paid to the miners accordingly. At this point, the function returns the resulting state, which is also stored on the blockchain.

Now as we understand accounts in Ethereum generally, let's examine the different types of accounts in Ethereum.

Types of accounts

Two kinds of accounts exist in Ethereum:

- **Externally Owned Accounts** (**EOAs**)
- **Contract Accounts** (**CAs**)

The first type is EOAs, and the other is 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.

The various properties of each type of account are as follows:

EOAs

- They have a state.
- They are associated with a human user, hence are also called user accounts.
- EOAs have an ether balance.
- They are capable of sending transactions.
- They have no associated code.
- They are controlled by private keys.
- EOAs cannot initiate a call message.
- Accounts contain a key-value store.
- EOAs can initiate transaction messages.

CAs

- They have a state.
- They are not intrinsically associated with any user or actor on the blockchain.
- CAs have an ether balance.
- They have associated code that is kept in memory/storage on the blockchain. They have access to storage.
- 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 CAs can be of any level of complexity. The code is executed by the EVM by each mining node on the Ethereum network. The EVM is discussed later in the chapter in the *The Ethereum Virtual Machine (EVM)* section.
- Also, CAs can maintain their permanent states and can call other contracts. It is envisaged that in the Serenity (Ethereum 2.0) release, the distinction between EOAs and CAs may be eliminated.
- CAs cannot start transaction messages.
- CAs can initiate a call message.
- CAs contain a key-value store.
- CAs' addresses are generated when they are deployed. This address of the contract is used to identify its location on the blockchain.

Accounts allow interaction with the blockchain via transactions. We will now explain what an Ethereum transaction is and consider its different types.

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 CA to another.
- **Contract creation transactions**: As the name suggests, these transactions result in the creation of a new CA. 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 as follows:

- **Nonce**: The 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 limits will be covered later in the chapter in more detail. For now, it is sufficient to say that this is the fee amount, 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 CA, this represents the balance that the contract will hold.
- **Signature**: The 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 the ECDSA scheme and makes use of the `secp256k1` curve. The theory of **Elliptic Curve Cryptography** (**ECC**) was discussed in *Chapter 4*, *Public Key Cryptography*. In this section, ECDSA will be presented in the context of its usage in Ethereum.

  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, 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 then 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 as follows:

$$ECDSASIGN\ (Message,\ Private\ Key) = (V,\ R,\ S)$$

- **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 the *body*, which persists and runs in response to message calls that the CA 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**, and represents the input data of the message call. It is also unlimited in size and is organized as a byte array.

This structure is 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** (**MPT**)) composed of the transactions to be included. Finally, the root node of the 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:

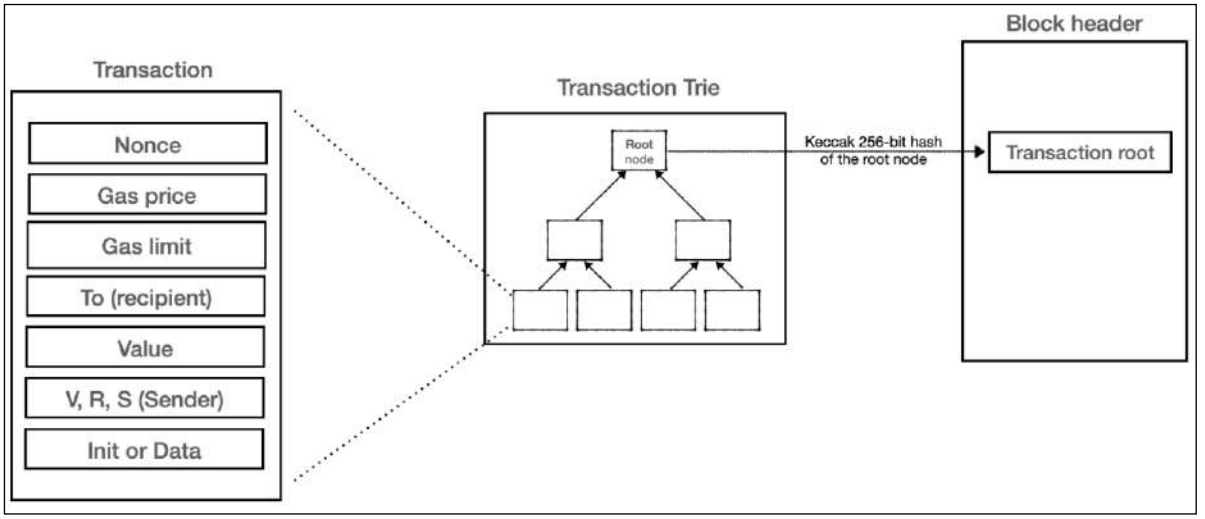


Figure 11.7: The relationship between the transaction, transaction trie, and block header

A block is a data structure that contains batches of transactions. 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. This process is similar to Bitcoin's mining process discussed in the previous chapters, *Chapter 6*, *Introducing Bitcoin* and *Chapter 7*, *Bitcoin Network and Payments*. Ethereum's PoW algorithm is called Ethash, and it was originally intended to be ASIC-resistant where finding a nonce requires large amounts of memory. However, now some ASICs are available for *Ethash* too. Consequently, there is also an agreement among Ethereum core developers to implement *ProgPow*, a new, more ASIC-resistant PoW consensus mechanism. We will explain ASICs and relevant concepts in *Chapter 12*, *Further Ethereum*, where we discuss mining in detail. We will also discuss block data structure in greater detail in this chapter.

The question that arises here is how all these accounts, transactions, and related messages flow through the Ethereum network and how are they stored. We saw earlier, in the *Ethereum – a user's perspective* section, a transaction flow example of how funds can be sent over the network from one user to another.

We need to know how this data looks on the network. So, before we move on to the different types of transactions and messages in Ethereum, let's explore how Ethereum data is encoded for storage and transmission. For this purpose, a new encoding scheme called **Recursive Length Prefix** (**RLP**) was developed, which we will cover here in detail.

RLP

To define RLP, we first need to understand the concept of serialization. Serialization is simply a mechanism commonly used in computing to encode data structures into a format (a sequence of bytes) that is suitable for storage and/or transmission over the communication links in a network. Once a receiver receives the serialized data, it de-serializes it to obtain the original data structure. Serialization and deserialization are also referred as marshaling and un-marshaling, respectively. Some commonly used serialization formats include XML, JSON, YAML, protocol buffers, and XDR. There are two types of serialization formats, namely *text* and *binary*. In a blockchain, there is a need to serialize and deserialize different types of data such as blocks, accounts, and transactions to support transmission over the network and storage on clients.

Why do we need a new encoding scheme when there are so many different serialization formats already available? The answer to this question is that RLP is a deterministic scheme, whereas other schemes may produce different results for the same input, which is absolutely unacceptable on a blockchain. Even a small change will lead to a totally different hash and will result in data integrity problems that will render the entire blockchain useless.

RLP is an encoding scheme developed by Ethereum developers. It 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. It is a deterministic and consistent binary encoding scheme used to serialize objects on the Ethereum blockchain, such as account state, transactions, messages, and blocks. It operates on strings and lists to produce raw bytes that are suitable for storage and transmission. RLP is a minimalistic and simple-to-implement serialization format that does not define any data types and simply stores structures as nested arrays. In other words, RLP does not encode specific data types; instead, its primary purpose is to encode structures.

More information on RLP is available on the Ethereum wiki, at `https://eth.wiki/en/fundamentals/rlp`.

Now, having defined RLP, we can delve deeper into transactions and other relevant elements of the Ethereum blockchain. We will start by exploring different types of transactions on the Ethereum blockchain.

Contract creation transactions

A contract creation transaction is used to create smart contracts on the blockchain. There are a few essential parameters required for a contract creation transaction. These parameters are listed as follows:

- The sender
- The transaction originator
- Available gas
- Gas price
- Endowment, which is the amount of ether allocated
- A byte array of an arbitrary length
- Initialization EVM code
- The current depth of the message call/contract-creation stack (current depth means the number of items that are already present in the stack)

Addresses generated as a result of a contract creation transaction are 160 bits in length. Precisely as defined in the yellow paper, they are the rightmost 160 bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the nonce. Initially, the nonce in the account is set to zero. The balance of the account is set to the value passed to the contract. The storage is also set to empty. The code hash is a Keccak 256-bit hash of the empty string.

The new account is initialized when the EVM code (the initialization EVM code, mentioned earlier) is executed. In the case of any exception during code execution, such as not having enough gas (running **Out of Gas**, or **OOG**), the state does not change. If the execution is successful, then the account is created after the payment of appropriate gas costs.

Since **Ethereum Homestead**, the result of a contract creation transaction is either a new contract with its balance, or no new contract is created with no transfer of value. This is in contrast to versions prior to Homestead, where the contract would be created regardless of the contract code deployment being successful or not due to an OOG exception.

Message call transactions

A message call requires several parameters for execution, which are listed as follows:

- The sender
- The transaction originator
- The recipient
- The account whose code is to be executed (usually the same as the recipient)
- Available gas
- The value
- The gas price
- An arbitrary-length byte array
- The input data of the call
- The current depth of the message call/contract creation stack

Message calls result in a state transition. Message calls also produce output data, which is not used if transactions are executed. In cases where message calls are triggered by EVM code, the output produced by the transaction execution is used. As defined in the yellow paper, a message call is the act of passing a message from one account to another. If the destination account has an associated EVM code, then the EVM will start upon the receipt of the message to perform the required operations. If the message sender is an autonomous object (external actor), then the call passes back any data returned from the EVM operation.

The state is altered by transactions. These transactions are created by external factors (users) and are signed and then broadcasted to the Ethereum network.

Messages are passed between accounts using message calls. A description of messages and message calls is presented next.

Messages

Messages, as defined in the yellow paper, are the data and values that are passed between two accounts. A **message** is a data packet passed between two accounts. This data packet contains data and a value (the amount of ether). It can either be sent via a smart contract (autonomous object) or from an external actor (an **Externally Owned Account**, or **EOA**) in the form of a transaction that has been digitally signed by the sender.

Contracts can send messages to other contracts. Messages only exist in the execution environment and are never stored. Messages are similar to transactions; however, the main difference is that they are produced by the contracts, whereas transactions are produced by entities external to the Ethereum environment (EOAs).

A message consists of the following components:

- The sender of the message
- The recipient of the message
- Amount of Wei to transfer and the message to be sent to the contract address
- An optional data field (input data for the contract)
- The maximum amount of gas (`startgas`) that can be consumed

Messages are generated when the `CALL` or `DELEGATECALL` opcodes are executed by the contract running in the EVM.

In the following diagram, the segregation between the two types of transactions (**contract creation** and **message calls**) is shown:

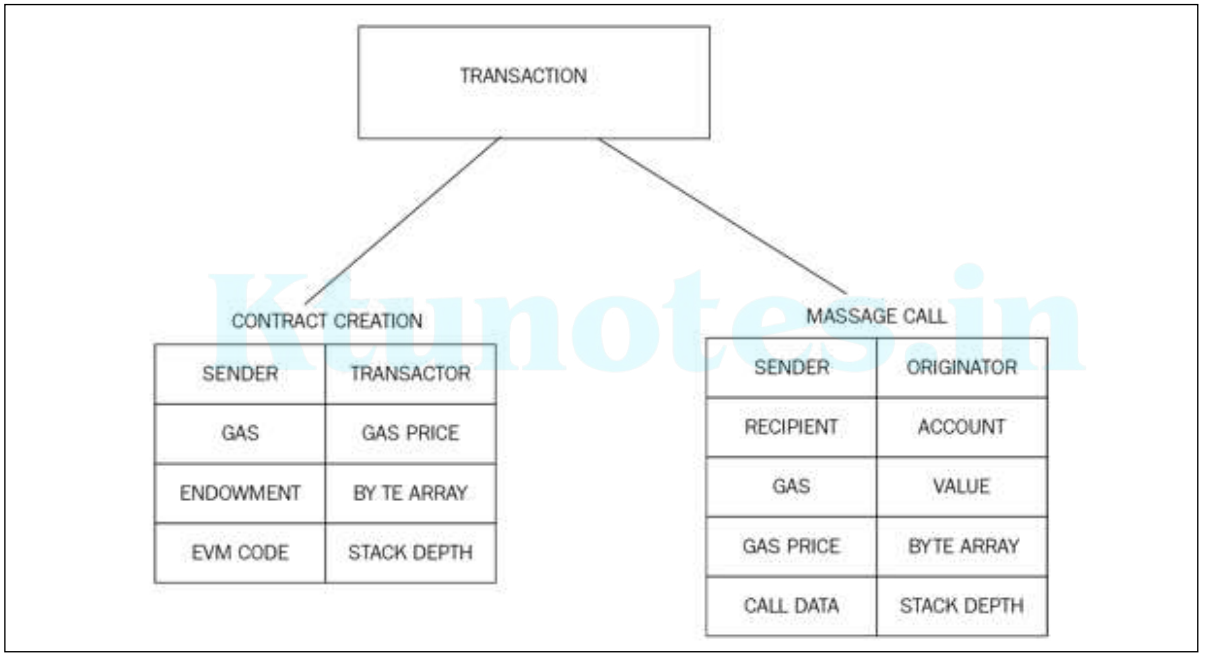


Figure 11.8: Types of transactions and the required parameters for execution

Each of these transactions has fields, which are shown with each type.

Calls

A call does not broadcast anything to the blockchain; instead, it is a local call and executes locally on the Ethereum node. It is almost like a local function call. It does not consume any gas as it is a read-only operation. It is akin to a dry run or a simulated run. Calls also do not allow ether transfer to CAs. Calls are executed locally on a node EVM and do not result in any state change because they are never mined. Calls are processed synchronously and they usually return the result immediately.

Do not confuse a *call* with a *message call transaction*, which in fact results in a state change. A call basically runs message call transactions locally on the client and never costs gas nor results in a state change. It is available in the `web3.js` JavaScript API and can be seen as almost a simulated mode of the message call transaction. On the other hand, a *message call transaction* is a `write` operation and is used for invoking functions in a CA (Contract Account, or smart contract), which does cost gas and results in a state change.

Transaction validation and execution

Transactions are executed after their validity has been verified. The initial checks are listed as follows:

- A transaction must be well formed and RLP-encoded without any additional trailing bytes
- The digital signature used to sign the transaction must be valid
- The transaction nonce must be equal to the sender's account's current nonce
- The gas limit must not be less than the gas used by the transaction
- The sender's account must contain sufficient balance to cover the execution cost

The transaction substate

A transaction substate is created during the execution of the transaction and is processed immediately after the execution completes. This transaction substate is a tuple that is composed of four items. These items are as follows:

- **Suicide set or self-destruct set**: This element contains the list of accounts (if any) that are disposed of after the transaction executes.
- **Log series**: This is an indexed series of checkpoints that allows the monitoring and notification of contract calls to the entities external to the Ethereum environment, such as application frontends. It works like a trigger mechanism that is executed every time a specific function is invoked, or a specific event occurs. Logs are created in response to events occurring in the smart contract. It can also be used as a cheaper form of storage. Events will be covered with practical examples in *Chapter 15*, *Introducing Web3*.
- **Refund balance**: This is the total price of gas for the transaction that initiated the execution. Refunds are not immediately executed; instead, they are used to offset the total execution cost partially.
- **Touched accounts**: Touched accounts can be defined as those accounts which are involved any potential state changing operation. Empty accounts from this set are deleted at the end of the transaction. The final state is reached after deleting accounts in the self-destruct set and emptying accounts from the touched accounts set.

State storage in the Ethereum blockchain

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

The world state

This 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 RLP.

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 the case of CAs, it represents the number of contracts created by the account.
- **Balance**: This value represents the number of weis, which is the smallest unit of the currency (ether) in Ethereum, held by the given address.
- **Storage root**: This field represents the root node of an MPT 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 the accounts trie, accounts, and block header are visualized in the following diagram. It shows the **account state**, or data structure, 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.

The accounts trie is an MPT used to encode the storage contents of an account. The contents are stored as a mapping between Keccak 256-bit hashes of 256-bit integer keys to the RLP-encoded 256-bit integer values.

Finally, the **root node** of the world state trie is hashed using the Keccak 256-bit algorithm and made part of the **block header** data structure, which is shown on the right-hand side of the diagram as the **state root** hash:

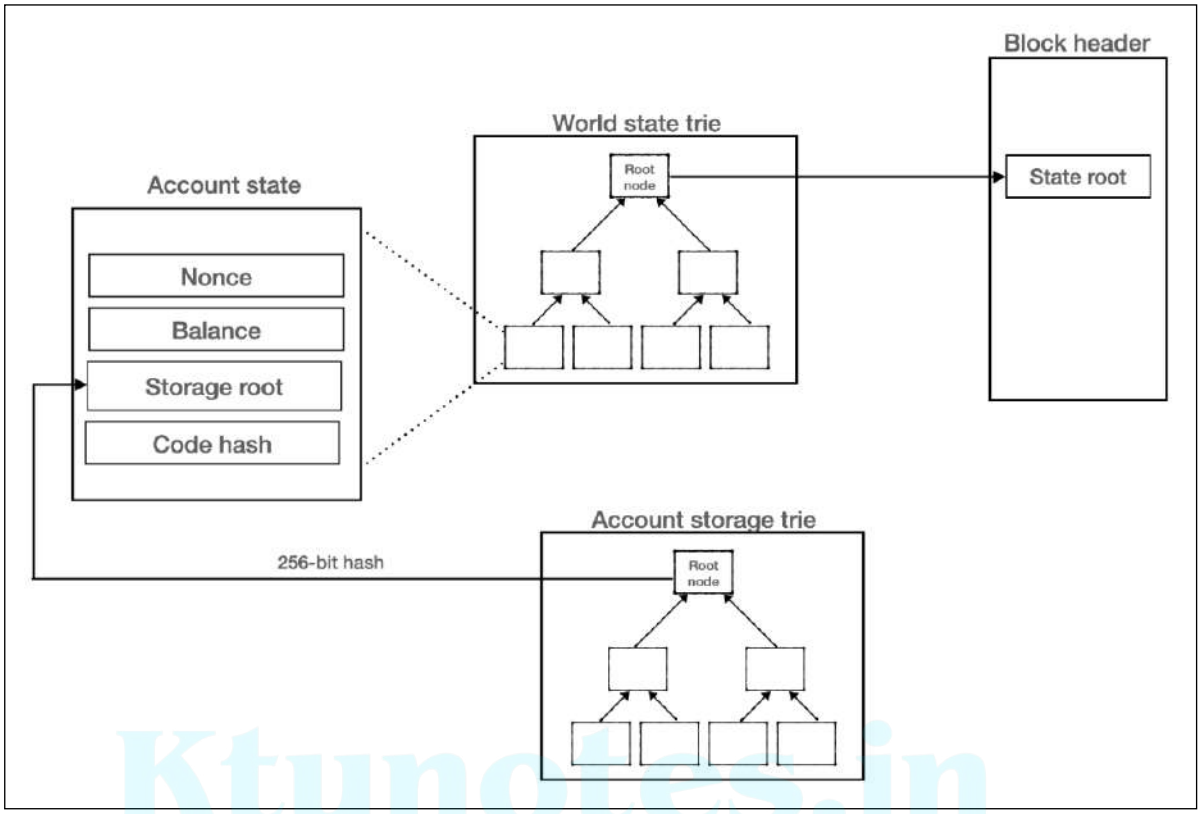


Figure 11.9: The accounts trie (the storage contents of account), account tuple, world state trie, and state root hash and their relationship

Transaction receipts

Transaction receipts are used as a mechanism to store the state after a transaction has been executed. In other words, these structures are used to record the outcome of the transaction execution. It is produced after the execution of each transaction. All receipts are stored in an index-keyed trie. The hash (a 256-bit Keccak hash) of the root of this trie is placed in the block header as the receipt's root. It is composed of four elements as follows:

- **The post-transaction state**: This item is a trie structure that holds the state after the transaction has been executed. It is encoded as a byte array.
- **Gas used**: This item represents the total amount of gas used in the block that contains the transaction receipt. The value is taken immediately after the transaction execution is completed. The total gas used is expected to be a non-negative integer.
- **Set of logs**: This field shows the set of log entries created as a result of the transaction execution. Log entries contain the logger's address, a series of log topics, and the log data.

- **The bloom filter**: A bloom filter is created from the information contained in the set of logs discussed earlier. A log entry is reduced to a hash of 256 bytes, which is then embedded in the header of the block as the logs bloom. A log entry is composed of the logger's address, log topics, and log data. Log topics are encoded as a series of 32-byte data structures. The log data is made up of a few bytes of data.

Since the Byzantium release, an additional field returning the success (`1`) or failure (`0`) of the transaction is also available.

More information about this change is available at `https://github.com/ethereum/EIPs/pull/658`.

This process of transaction receipt generation is visualized in the following diagram:

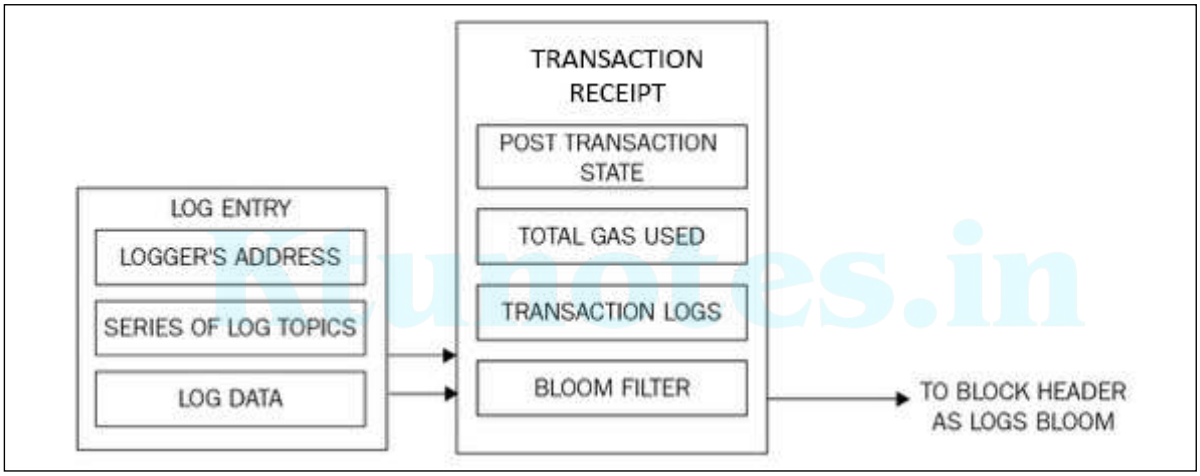


Figure 11.10: Transaction receipts and logs bloom

As a result of the transaction execution process, the state morphs from an initial state to a target state. This concept was discussed briefly at the beginning of the chapter. This state needs to be stored and made available globally in the blockchain. We will see how this process works in the next section.

Ether cryptocurrency/tokens (ETC and ETH)

As an incentive to the miners, Ethereum rewards its own native currency called **ether** (abbreviated as **ETH**). After the **Decentralized Autonomous Organization** (**DAO**) hack described in *Chapter 10*, *Smart Contracts*, a hard fork was proposed in order to mitigate the issue; therefore, 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 own following with a dedicated community that is further developing ETC, which is the unforked original version of Ethereum.

This chapter is focused on ETH, which is currently the most active and official Ethereum blockchain.

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

The denomination table is as follows:

Unit	Alternative name	Wei value	Number of weis
Wei	Wei	1 Wei	1
KWei	Babbage	1e3 Wei	1,000
MWei	Lovelace	1e6 Wei	1,000,000
GWei	Shannon	1e9 Wei	1,000,000,000
microether	Szabo	1e12 Wei	1,000,000,000,000
milliether	Finney	1e15 Wei	1,000,000,000,000,000
ether	ether	1e18 Wei	1,000,000,000,000,000,000

Fees are charged for each computation performed by the EVM on the blockchain. A detailed fee schedule is described in *Chapter 12*, *Further Ethereum*.

The Ethereum Virtual Machine (EVM)

The EVM is a simple stack-based execution machine that runs bytecode instructions to transform the system state from one state to another. The word size of the EVM is set to 256-bit. The stack size is limited to 1,024 elements and is based on the **Last In, First Out** (**LIFO**) queue. The EVM is a Turing-complete machine but is limited by the amount of gas that is required to run any instruction. This means that infinite loops that can result in denial-of-service attacks are not possible due to gas requirements. The EVM also supports exception handling should exceptions occur, such as not having enough gas or providing invalid instructions, in which case the machine would immediately halt and return the error to the executing agent.

The EVM is an entirely isolated and sandboxed runtime environment. The code that runs on the EVM does not have access to any external resources such as a network or filesystem. This results in increased security, deterministic execution, and allows untrusted code (code that can be run by anyone) to be executed on Ethereum blockchain.

As discussed earlier, the EVM is a stack-based architecture. The EVM is big-endian by design, and it uses 256-bit-wide words. This word size allows for Keccak 256-bit hash and ECC computations.

There are three main types of storage available for contracts and the EVM:

- **Memory**: The first type is called memory or volatile memory, which is a word-addressed byte array. When a contract finishes its code execution, the memory is cleared. It is akin to the concept of RAM. `write` operations to the memory can be of 8 or 256 bits, whereas `read` operations are limited to 256-bit words. Memory is unlimited but constrained by gas fee requirements.
- **Storage**: The other type is called storage, which is a key-value store and is permanently persisted on the blockchain. Keys and values are each 256 bits wide. It is allocated to all accounts on the blockchain. As a security measure, storage is only accessible by its own respective CAs. It can be thought of as hard disk storage.
- **Stack**: EVM is a stack-based machine, and thus performs all computations in a data area called the stack. All in-memory values are also stored in the stack. It has a maximum depth of 1024 elements and supports the word size of 256 bits.

The storage associated with the EVM is a word-addressable word array that is non-volatile and is maintained as part of the system state. Keys and values are 32 bytes in size and storage. The program code is stored in **virtual read-only memory** (**virtual ROM**) that is accessible using the `CODECOPY` instruction. The `CODECOPY` instruction copies the program code into the main memory. Initially, all storage and memory are set to zero in the EVM.

The following diagram shows the design of the EVM where the virtual ROM stores the program code that is copied into the main memory using the `CODECOPY` instruction. The main memory is then read by the EVM by referring to the program counter and executes instructions step by step. The program counter and EVM stack are updated accordingly with each instruction execution:

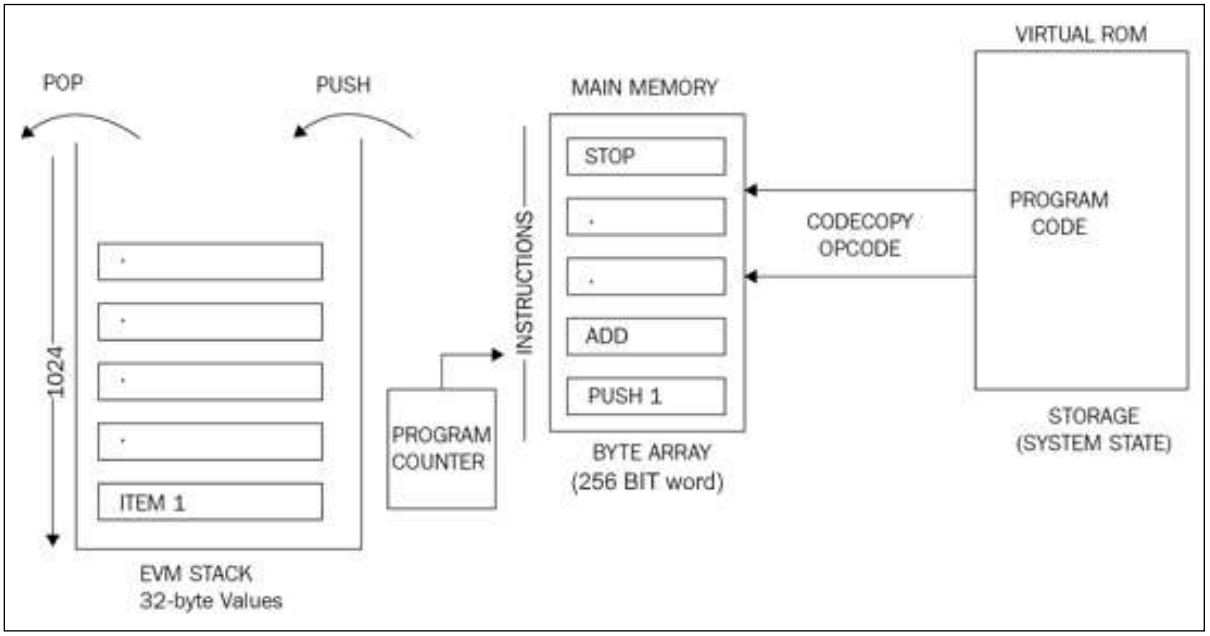


Figure 11.11: EVM operation

The preceding diagram shows an EVM stack on the left side showing that elements are pushed and popped from the stack. It also shows that a program counter is maintained, and is incremented with instructions being read from the main memory. The main memory gets the program code from the virtual ROM/storage via the `CODECOPY` instruction.

EVM optimization is an active area of research, and recent research has suggested that the EVM can be optimized and tuned to a very fine degree to achieve high performance. Research and development on **Ethereum WebAssembly** (**ewasm**)—an Ethereum-flavored iteration of WebAssembly—is already underway. **WebAssembly** (**Wasm**) was developed by Google, Mozilla, and Microsoft, and is now being designed as an open standard by the W3C community group. Wasm aims to be able to run machine code in the browser that will result in execution at native speed. More information and GitHub repository of Ethereum-flavored Wasm is available at `https://github.com/ewasm`.

Another intermediate language called YUL, which can compile to various backends such as the EVM and ewasm, is under development. More information on this language can be found at `https://solidity.readthedocs.io/en/latest/yul.html`.

Execution environment

There are some key elements that are required by the execution environment to execute the code. The key parameters are provided by the execution agent; for example, a transaction. These are listed as follows:

- The system state.
- The remaining gas for execution.
- The address of the account that owns the executing code.
- The address of the sender of the transaction. This is the originating address of this execution (it can be different from the sender).
- The gas price of the transaction that initiated the execution.
- Input data or transaction data depending on the type of executing agent. This is a byte array; in the case of a message call, if the execution agent is a transaction, then the transaction data is included as input data.
- The address of the account that initiated the code execution or transaction sender. This is the address of the sender in case the code execution is initiated by a transaction; otherwise, it is the address of the account.
- The value or transaction value. This is the amount in Wei. If the execution agent is a transaction, then it is the transaction value.
- The code to be executed presented as a byte array that the iterator function picks up in each execution cycle.
- The block header of the current block.
- The number of message calls or contract creation transactions (CALLs, CREATEs or CREATE2s) currently in execution.
- Permission to make modifications to the state.

The execution environment can be visualized as a tuple of ten elements, as follows:

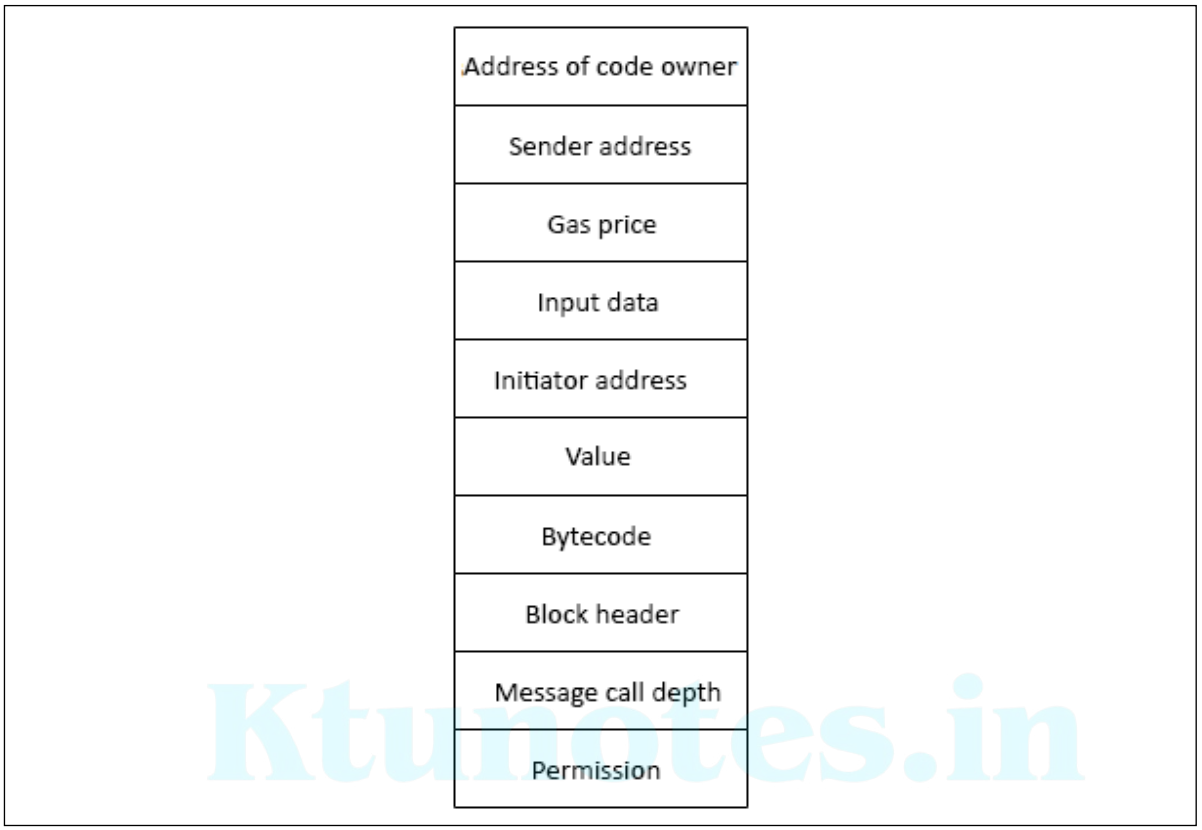


Figure 11.12: Execution environment tuple

The execution results in producing the resulting state, the gas remaining after the execution, the self-destruct or suicide set, log series, and any gas refunds.

The machine state

The machine state is also maintained internally, and updated after each execution cycle of the EVM. An iterator function (detailed in the next section) runs in the EVM, which outputs the results of a single cycle of the state machine.

The machine state is a tuple that consists of the following elements:

- Available gas
- The program counter, which is a positive integer of up to 256
- The contents of the memory (a series of zeroes of size 2^{256})
- The active number of words in memory (counting continuously from position 0)
- The contents of the stack

The EVM is designed to handle exceptions and will halt (stop execution) if any of the following exceptions should occur:

- Not having enough gas required for execution
- Invalid instructions
- Insufficient stack items
- Invalid destination of jump opcodes
- Invalid stack size (greater than 1,024)

Machine state can be viewed as a tuple, as shown in the following diagram:

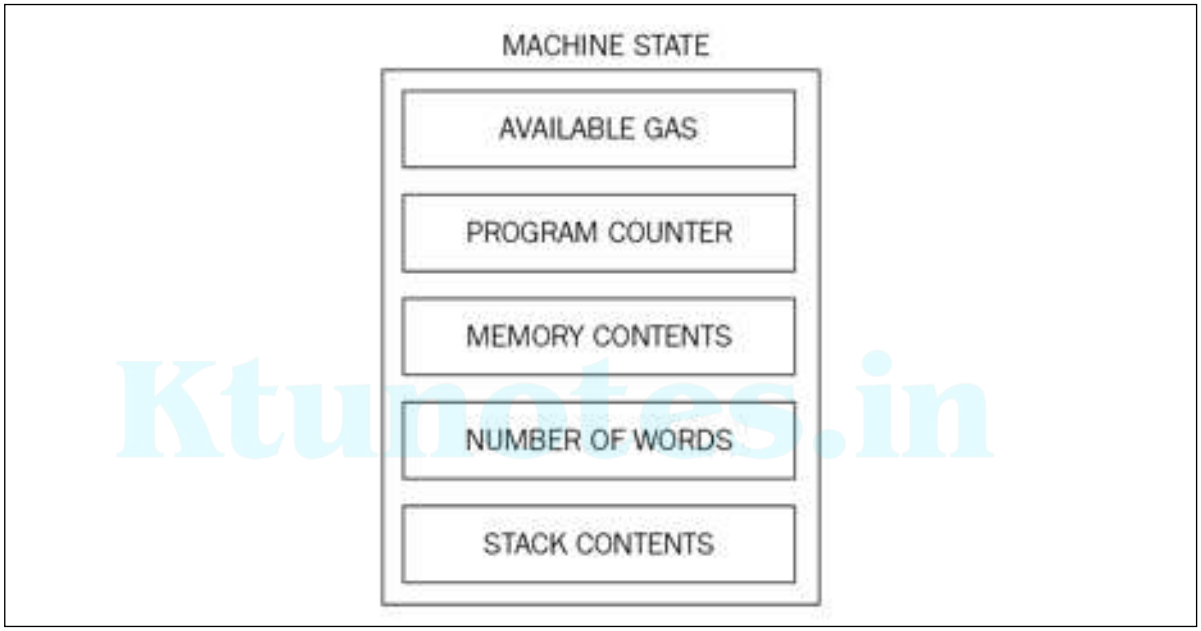


Figure 11.13: Machine state tuple

The iterator function

The iterator function mentioned earlier performs various vital functions that are used to set the next state of the machine and eventually the world state. These functions include the following:

- It fetches the next instruction from a byte array where the machine code is stored in the execution environment.
- It adds/removes (`PUSH/POP`) items from the stack accordingly.
- Gas is reduced according to the gas cost of the instructions/opcodes. It increments the **Program Counter** (**PC**).

The EVM is also able to halt in normal conditions if `STOP`, `SUICIDE`, or `RETURN` opcodes are encountered during the execution cycle.

12

Further Ethereum

This chapter is a continuation of the previous chapter, in which we will examine more Ethereum-based concepts in more detail.

We will cover both a practical and theoretical in-depth introduction to wallet software, mining, and setting up Ethereum nodes. Material relating to various challenges, such as security and scalability faced by Ethereum, will also be introduced. Moreover, prominent advanced supporting protocols, such as Swarm and Whisper, will also be introduced later in the chapter. Finally, Ethereum has several programming languages built in to support smart contract development. We will conclude with an overview of these programming languages.

This chapter continues the discussion of the Ethereum blockchain network elements that we started in the previous chapter, *Chapter 11*, *Ethereum 101*. These elements include:

- Blocks and blockchain
- Wallets and client software
- Nodes and miners
- APIs and tools
- Supporting protocols
- Programming languages

First, we will introduce blocks and blockchain, in continuation of the discussion from the previous chapter.

Blocks and blockchain

Blocks are the main building structure 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

An uncle block is a block that is the child of a parent but does not have any child block. Ommers or uncles are valid, but stale, blocks that are not part of the main chain but contribute to security of the chain. They also earn a reward for their participation but do not become part of the canonical truth.

The transaction list is simply a list of all transactions included in the block. Also, the list of headers of uncles is also included in the block.

Block header: Block headers are the most critical and detailed components of an Ethereum block. The header contains various elements, which are 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 (or uncles) blocks included in the block.
- **The beneficiary**: The 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 once 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. The 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. More details on transaction receipts are provided in the next section.
- **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. Logging is explained in detail in the next section.
- **Difficulty**: The difficulty level of the current block.
- **Number**: The total number of all previous blocks; the genesis block is block zero.
- **Gas limit**: This field contains the value that represents the limit set on the gas consumption per block.
- **Gas used**: This field contains the total gas consumed by the transactions included in the block.
- **Timestamp**: The timestamp is the epoch Unix time of the time of block initialization.

- **Extra data**: The 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**: The mixhash field contains a 256-bit hash that, once combined with the nonce, is used to prove that adequate computational effort (**Proof of Work**, or **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 diagram shows the detailed structure of the block and block header:

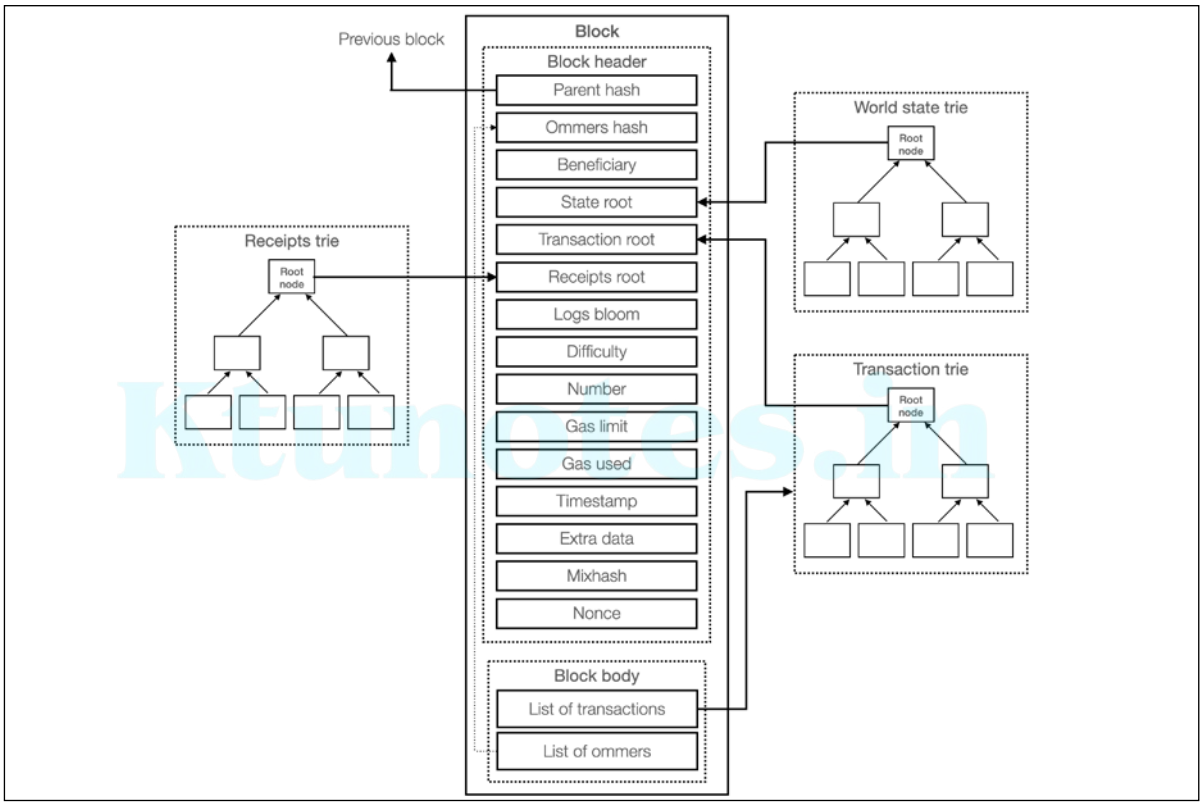


Figure 12.1: A detailed diagram of the block structure with a block header and relationship with tries

The genesis block

The genesis block is the first block in a blockchain network. It varies slightly from normal blocks due to the data it contains and the way it has been created. It contains 15 items that are described here.

From `https://etherscan.io/`, the actual version is shown as follows:

Element	Description
Timestamp	(Jul-30-2015 03:26:13 PM +UTC)
Transactions	8893 transactions and 0 contract internal transactions in this block
Hash	0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3
Parent hash	0x0000000000000000000000000000000000000000000000000000000000000000
SHA-3 uncles	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
Mined by	0x0000000000000000000000000000000000000000 in 15 seconds
Difficulty	17,179,869,184
Total difficulty	17,179,869,184
Size	540 bytes
Gas used	0
Nonce	0x0000000000000042
Block reward	5 ETH
Uncles reward	0
Extra data	In hex (0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbdb7a38e1e50b1b82fa)
Gas limit	5,000

The block validation mechanism

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

- If it is consistent with uncles and transactions. This means that all ommers 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. A list of errors for which the block can be rejected is presented here:
    - The timestamp is older than the parent
    - There are too many uncles
    - There is a duplicate uncle
    - The uncle is an ancestor
    - The uncle's parent is not an ancestor
    - There is non-positive difficulty
    - There is an invalid mix digest
    - There is an invalid PoW

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**. In the case of mining, determine ommers. The validation process of the headers of stale blocks checks whether the header is valid and whether the relationship between the uncle and the current block satisfies the maximum depth of six blocks. A block can contain a maximum of two uncles.
2. **Transaction validation**. In the case of mining, determine transactions. This process involves checking whether the total gas used in the block is equal to the final gas consumption after the final transaction, in other words, the 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 2 ether. It was reduced first from 5 ether to 3 with the Byzantium release of Ethereum. Later, in the Constantinople release (`https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/`), it was reduced further to 2 ether. 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.

After the high-level view of the block validation mechanism, we now look into how a block is received and processed by a node. We will also see how it is updated in the local blockchain:

1. When an Ethereum full node receives a newly mined block, the header and the body of the block are detached from each other. Now remember, in the last chapter, when we introduced the fact that there are three **Merkle Patricia tries** (**MPTs**) in an Ethereum blockchain. The roots of those MPTs or tries are present in each block header as a state trie root node, a transaction trie root node, and a receipt trie root node. We will now learn how these tries are used to validate the blocks.
2. A new MPT is constructed that comprises all transactions from the block.
3. All transactions from this new MPT are executed one by one in a sequence. This execution occurs locally on the node within the **Ethereum Virtual Machine** (**EVM**). As a result of this execution, new transaction receipts are generated that are organized in a new receipts MPT. Also, the global state is modified accordingly, which updates the state MPT (trie).
4. The root nodes of each respective trie, in other words, the state root, transaction root, and receipts root are compared with the header of the block that was split in the first step. If both the roots of the newly constructed tries and the trie roots that already exist in the header are equal, then the block is verified and valid.
5. Once the block is validated, new transaction, receipt, and state tries are written into the local blockchain database.

Block difficulty mechanism

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 as follows:

$$block\_diff = parent\_diff + parent\_diff // 2048 *$$

$$max(1 - (block\_timestamp - parent\_timestamp) // 10, -99) +$$

$$int(2**((block.number // 100000) - 2))$$

Note that // is the integer division operator.

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 by *parent_diff // 2048 * 1*. If the time difference is between 10 and 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 from *parent_diff // 2048 * -1* to a maximum decrease of *parent_diff // 2048 * -99*.

In addition to timestamp difference-based difficulty adjustment, there is also another part (shown in the last line of the preceding algorithm) that increases the difficulty exponentially after every 100,000 blocks. This is the so-called **difficulty time bomb**, or **ice age**, introduced in the Ethereum network, which will make it very hard to mine on the Ethereum blockchain at some point in the future. This will encourage users to switch to **Proof of Stake** (**PoS**), since mining on the PoW chain will eventually become prohibitively difficult.

The difficulty time bomb was delayed via EIP-649 (`https://github.com/ethereum/EIPs/pull/669`) for around 18 months and no clear time frame has yet been suggested.

According to the original estimates based on the algorithm, the block generation time would have become significantly higher during the second half of 2017, and in 2021, it would become so high that it would be virtually impossible to mine on the PoW chain, even for dedicated mining centers. This way, miners will have no choice but to switch to the PoS scheme proposed by Ethereum, called **Casper**.

More information about Casper is available here: `https://github.com/ethereum/research/blob/master/papers/casper-basics/casper_basics.pdf`.

This ice age proposal has been postponed with the release of **Byzantium**. Instead, the mining reward has been reduced from 5 ETH to 3 ETH, in preparation for PoS implementation in Serenity.

In the Byzantium release, the difficulty adjustment formula has been changed to take uncles into account for difficulty calculation. This new formula is shown here:

$$adj\_factor = max((2\ if\ len(parent.uncles)\ else\ 1) - ((timestamp - parent.timestamp)\ //\ 9), -99)$$

Soon after the **Istanbul** upgrade, the difficulty bomb was delayed once again, under the **Muir Glacier** network upgrade with a hard fork, `https://eips.ethereum.org/EIPS/eip-2384`, for roughly another 611 days. The change was activated at block number 9,200,000 on January 2, 2020.

We know that blocks are composed of transactions and that there are various operations associated with transaction creation, validation, and finalization. Each of these operations on the Ethereum network costs some amount of ETH and is charged using a fee mechanism. This fee is also called gas. We will now introduce this mechanism in detail.

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 an 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 costs can be estimated using the following 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 ETH. 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 less gas. If the execution runs out of gas, everything is immediately rolled back; otherwise, if the execution is successful and some gas remains, then it is returned to the transaction originator.

A website that keeps track of the latest gas price and provides other valuable statistics and calculators is available at `https://ethgasstation.info/index.php`.

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

Operation name	Gas cost
Stop	0
SHA3	30
SLOAD	800
Transaction	21000
Contract creation	32000

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 the current gas price is 25 GWei, and convert it into ETH, which is 0.000000025 ETH. After multiplying both, 0.000000025 * 30, we get 0.00000075 ETH.
- In total, 0.00000075 ETH is the total gas that will be charged.

Fee schedule

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 calls
- An increase in the use of memory

A list of instructions and various operations with the gas values has been provided in the previous section.

Now let's move onto another topic, clients and wallets, which are responsible for mining, payments, and management functions, such as account creation on an Ethereum network.

The layout of a Solidity source code file

In the following subsections, we will look at the components of a Solidity source code file, which is important to cover before we move on to writing smart contracts in the next section.

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 in the following example:


```
pragma solidity ^0.5.0
```

This will ensure that the source file does not compile with versions lower than `0.5.0` and versions starting from `0.6.0`.

Import

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 in the following:, for example:

```
import "module-name";
```

Comments

Comments can be added to the Solidity source code file in a manner similar to the 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:

```
pragma solidity ^0.5.0; //specify the solidity compiler version
/*
this is a simple value checker contract that checks the value provided
and returns boolean value (true or false) based on the condition expression
evaluation
*/
import "./mapping.sol"; //import a file
contract valuechecker {
    uint price = 10;
    //price variable declared and initialized with a value of 10
    event valueEvent(bool returnValue);
    function Matcher (uint8 x) public returns (bool) {
        if (x >= price )
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

Figure 14.13: Sample Solidity program as shown in the Remix IDE

In this section, we examined what Solidity code of a smart contract looks like. Now it's time to learn about the Solidity language.

The Solidity language

Solidity is a domain-specific language of choice for programming contracts in Ethereum. There are other languages that can be used, such as Serpent, Mutan, and LLL, but Solidity is the most popular. 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#. Nevertheless, this is the most widely used language currently available for programming contracts.

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 the 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 contract 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.

Variables

Just like any programming language, variables in Solidity are the named memory locations that hold values in a program. There are three types of variables in Solidity: **local variables**, **global variables**, and **state variables**.

Local variables

These variables have a scope limited to only within the function they are declared in. In other words, their values are present only during the execution of the function in which they are declared.

Global variables

These variables are available globally as they exist in the global namespace. They are used for performing various functions such as ABI encoding, cryptographic functions, and querying blockchain and transaction information.

Solidity provides a number of global variables that are always available in the global namespace. These variables provide information about blocks and transactions. Additionally, cryptographic functions, ABI encoding/decoding, and address-related variables are available. A subset of available variables is shown as follows.

This function is used to compute the Keccak-256 hash of the argument provided to the function:

```
keccak256(...) returns (bytes32)
```

This function returns the associated address of the public key from the elliptic curve signature:

```
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)
```


This returns the current block number:

```
block.number
```

This returns the gas price of the transaction:

```
tx.gasprice (uint)
```

There are a number of other global variables available. A comprehensive list and details can be found in Solidity's official documentation: `https://solidity.readthedocs.io/en/latest/units-and-global-variables.html`.

State variables

State variables have their values permanently stored in smart contract storage. 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:

```
pragma solidity >=0.5.0;

contract Addition {
    uint x; // State variable
}
```

Here, `x` is a state variable whose value will be stored in contract storage.

There are three types of state variables, based on their visibility scope

- **Private**: These variables are only accessible internally from within the contract that they are originally defined in. They are also not accessible from any derived contract from the original contract.
- **Public**: These variables are part of the contract interface. In simple words, anyone is able to get the value of these variables. They are accessible within the contract internally by using the `this` keyword. They can also be called from other contracts and transactions. A `getter` function is automatically created for all public variables.
- **Internal**: These variables are only accessible internally within the contract that they are defined in. In contrast to private state variables, they are also accessible from any derived contract from the original (parent) contract.

In the next section, we will introduce the data types supported in Solidity.

Data types

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

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

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 explicitly specify the storage area where the type is stored, for example, *memory*, *storage*, or *calldata*.

Remember that EVM can read and write data in different locations.

- Stack: This is a temporary storage area where EVM opcodes pop and push data during execution of the bytecode.

- Calldata: This is a read-only location that holds the data field of a transaction. It holds function execution data and parameters.

- Memory: This is a temporary memory location that is available to functions of the smart contract during their execution. As soon as the execution of the transaction finishes, this memory is cleared.

- Storage: This is the persistent global memory available to all functions in a smart contract. State variables use storage.

- Code: This is where the code that is executing is stored. This can also be used as a static data storage location.

- Logs: This is the area where the output of the events emitting from the smart contracts is stored.

The specific location used for storing values of a variable depends on the data type of the variable and where the variable has been declared. For example, function parameter variables are stored in *memory*, whereas state variables are stored in *storage*.

Now we'll describe value types in detail.

Value types

Value types mainly include **Booleans**, **integers**, **addresses**, and **literals**, which are explained in detail here.

Boolean

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

```
bool v = true;
```

or

```
bool v = false;
```

This statement assigns the value `true` or `false` to `v` depending on the assignment.

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`, and `int24`.
`uint`	Unsigned integer	`uint8`, `uint16`, ... to `uint256`, unsigned integer from 8 bits to 256 bits. Usage is dependent on 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;
```

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` functions are provided in order to interact with functions that do not have an 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 a 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 the 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. An example is shown as follows:

```
uint8 x = 2;
```

- **String literals**: This type specifies a set of characters written with double or single quotes. An example is shown as follows:

```
'packt' "packt"
```

- **Hexadecimal literals**: These are prefixed with the keyword `hex` and specified within double or single quotation marks. An example is shown as follows:

```
(hex'AABBCC');
```

- **Enums**: This allows the creation of user-defined types. An example is shown as follows:

```
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

As the name suggests, these types are passed by reference and are discussed in the following section. These are also known as **complex types**. Reference types include **arrays, structs**, and **mappings**.

Arrays

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`.

Structs

These constructs can be used to group a set of dissimilar data types under a logical group. These can be used to define new custom 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"});

}
```

In the preceding code, we declared a `struct` named `Trade` that has four fields. `tradeid`, `quantity`, and `price` are of the `uint` type, whereas `trader` is of the `string` type. Once the `struct` is declared, we can initialize and use it. We initialize it by using `Trade tStruct` and assigning `123` to `tradeid`, `1` to `quantity`, and `"equinox"` to `trader`.

Data location

Sometimes it is desirable to choose the location of the variable data storage. This choice allows for better gas expense management. We can use the data location name to specify where a particular complex data type will be stored. Depending on the default or annotation specified, the location can be `storage`, `memory`, or `calldata`. This is applicable to arrays and structs and can be specified using the `storage` or `memory` keywords. `calldata` behaves almost like memory. It is an unmodifiable and temporary area that can be used to store function arguments.

For example, in the preceding structs example, if we want to use only memory (temporarily) we can do that by using the `memory` keyword when using the structure and assigning values to fields in the `struct`, as shown here:

```
Trade memory tStruct;
tStruct.tradeid = 123;
```

As copying between memory and storage can be quite expensive, specifying a location can be helpful to control the gas expenditure at times.

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

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

```
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 the string `packt` mapped to value `10`.

Control structures

The control structures available in the Solidity language are `if...else`, `do`, `while`, `for`, `break`, `continue`, and `return`. They work exactly the same as other languages, such as the 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 in a similar vein, 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 a value of `0`.

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 the function `Matcher` is equal to or greater than `10`:

```
pragma solidity >=0.6.0;
contract valueChecker
{
    uint8 price=10;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

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 >=0.6.0;
contract valueChecker
{
    uint8 price = 20;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit 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`, and thus it is not allowed to be accessed by any other contract. The error message that you will see when attempting to compile this contract is as follows:

```
browser/valuechecker.sol:20:8: DeclarationError: Undeclared identifier.
return price+10;
       ^---^
```

Libraries

Libraries are deployed only once at a specific address and their code is called via the `CALLCODE` or `DELEGATECALL` opcode of the EVM. The key idea behind libraries is code reusability. They are similar to contracts and act as base contracts to the calling contracts. A library can be declared as shown in the following example:

```
library Addition
{
    function Add(uint x,uint y) returns (uint z)
    {
        return x + y;
    }
}
```


This library can then be called in the contract, as shown here. First, it needs to be imported and then it can be used anywhere in the code. A simple example is shown as follows:

```
import "Addition.sol"
function Addtwovalues() returns(uint)
{
    return Addition.Add(100,100);
}
```

There are a few limitations with libraries; for example, they cannot have state variables and cannot inherit or be inherited. Moreover, they cannot receive ether either; this is in contrast to contracts, which can receive ether.

Functions

Functions are pieces of code within a smart contract. For example, look at the following code block:

```
pragma solidity >5.0.0;

contract Test1
{
    uint x=2;
```

```
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

In the preceding code example, with `contract Test1`, we have defined a function called `addition1()`, which returns an unsigned integer after adding `2` to the value supplied via the variable `x`, initialized just before the function.

In this case, `2` is supplied via variable `x`, and the function will return `4` by adding `2` to the value of `x`. It is a simple function, but demonstrates how functions work and what their different elements are.

There are two function types: *internal* and *external* functions.

- **Internal functions** can be used only within the context of the current contract.
- **External functions** can be called via external function calls.

A **function** in Solidity can be marked as a constant. Constant functions cannot change anything in the contract; they only return values when they are invoked and do not cost any gas. This is the practical implementation of the concept of *call* as discussed in the previous chapter.

Note that the state mutability modifier `constant` was removed in Solidity version 0.5.0. Instead, use the `view` or `pure` modifier if you are using Solidity version 0.5.0 or greater.

The syntax to declare a function is shown as follows:

```
function <nameofthefunction> (<parameter types> <name of the variable>)
{internal|external} [state mutability modifier] [returns (<return types> <name of
the variable>)]
```

For example:

```
function addition1() public view returns (uint y)
```

Here, `function` is the keyword used to declare the function, `addition` is the name of the function, `public` is the visibility modifier, `view` is the state mutability modifier, `returns` is the keyword to specify what is returned from the function, and `uint` is the return type with the variable name `y`.

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

```
function orderMatcher (uint x)
private view returns (bool return value)
```

In the preceding code, `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, `view` 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. After this introduction to functions in Solidity, let's consider some of their key elements, parameters, and modifiers:

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

```
function <name of the function>(<parameters>) <visibility specifier> <state
mutability modifier> returns
(<return data type> <name of the variable>)
{
    <function body>
}
```

- **Function signature**: Functions in Solidity are identified by their **signature**, which is the first four bytes of the Keccak-256 hash of its full signature string. This is also visible in the Remix IDE, as shown in the following screenshot. `f9d55e21` is the first four bytes of the 32-byte Keccak-256 hash of the function named `Matcher`:

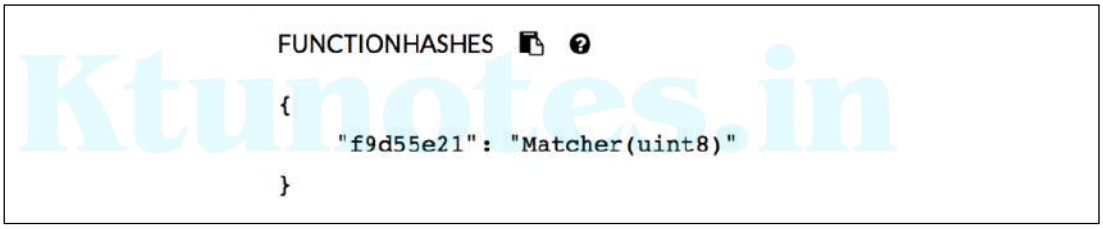


Figure 14.14: Function hash as shown in the Remix IDE

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 as well as take multiple inputs. 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 optionally be omitted. An example of such a function could be:

```
pragma solidity ^0.5.0;

contract Test1
{

    function addition1(uint x, uint y) public pure returns (uint z, uint
a)
    {
        z= x+y ;
        a=x+y;
        return (z,a);

    }
}
```


Here when the code runs, it will take two parameters as input `x` and `y`, add both, and then assign them to `z` and `a`, and finally return `z` and `a`. For example, if we provide 1 and 1 for `x` and `y`, respectively, then when the variables `z` and `a` are returned by the function, both will contain 2 as the result.

- **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 of a contract are inherited from the address.
- **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/access levels): 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 by 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. This is available before version 0.5.0 of Solidity.

Now we've considered some of the defining features of functions, let's consider how Solidity approaches handling errors.

Error handling

Solidity provides various functions for error handling. By default, in Solidity, whenever an error occurs, the state does not change and reverts back to the original state.

Some constructs and convenience functions that are available for error handling in Solidity are introduced as follows:

- **Assert**: This is used to check for conditions and throw an exception if the condition is not met. Assert is intended to be used for internal errors and invariant checking. When called, this method results in an invalid opcode and any changes in the state are reverted back.
- **Require**: Similar to *assert*, this is used for checking conditions and throws an exception if the condition is not met. The difference is that *require* is used for validating inputs, return values, or calls to external contracts. The method also results in reverting back to the original state. It can also take an optional parameter to provide a custom error message.
- **Revert**: This method aborts the execution and reverts the current call. It can also optionally take a parameter to return a custom error message to the caller.
- **Try/Catch**: This construct is used to handle a failure in an external call.
- **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.

This completes a brief introduction to the Solidity language. The language is very rich and under constant improvement. Detailed documentation and coding guidelines are available online at `http://solidity.readthedocs.io/en/latest/`.