Indexes for Blockchain Data

Ευρετήρια για Δεδομένα Blockchain

Kalimeris Markos

Supervisor: Assist. Prof. Katsaros Dimitrios

2^{nd} committee member: Prof. Tousidou Eleni

A Thesis submitted in fulfillment of the requirements
for the degree of Diploma Thesis
in the
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece

February 2019

Dedicated to

my family and friends

Ευρετήρια για Δεδομένα Blockchain

Περίληψη

Το Blockchain είναι μια νέα τεχνολογία, η οποία διαθέτει τεράστιες δυνατότητες εφαρμογής σε ποικίλες βιομηχανίες. Έγινε ευρεύως γνωστή ώντας η βασική τεχνολογία πίσω από τη δημιουργία του ψηφιακού νομίσματος Bitcoin το 2008. Ωστόσο οι τομείς της σημερινής τεχνολογίας που μπορεί να εφαρμοστεί δεν περιορίζονται στον χρηματοοικονομικό τομέα. Το αντικείμενο της παρούσας διπλωματικής είναι η μελέτη αυτής της τεχνολογίας, αλλά και η κατασκευή μιας βάσης δεδομένων για τα δεδομένα της συγκεκριμένης τεχνολογίας ώστε να παρέχεται ένας εύκολος τρόπος αναζήτησης πληροφορίων βάση ευρετηρίου. Πιο συγκεκριμένα αρχικά ερευνά και αναλύει τις έννοιες της τεχνολογίας Blockchain καθώς και τον προσδιορισμό των τεχνικών συνεπειών που παρέχει η τεχνολογία, με βάση τις σημαντικότερες υπάρχουσες πλατφόρμες όπως είναι το Bitcoin, το Ethereum, το Ripple, το Hyperledger κ.α. Δεδομένου του γεγονότος πως το Blockchain , είναι μια συνεχώς αυξανόμενη λίστα εγγραφών, συνδεδεμένη μέσω κρυπτογραφίας, προκύπτει το πρόβλημα της αδυναμίας αναζήτησης πληροφορίων που βρίσκονται κατανεμημένες σε αυτό. Γι' αυτό το λόγο υλοποιείται ένα ευρετήριο βασισμένο στο μοντέλο κλειδιού-τιμής (key-value) χρησιμοποιώντας την κατανεμημένη βάση δεδομένων Redis, και χρησιμοποιώντας τα δεδομένα που εξάγουμε από μια Blockchain πλατφόρμα, το Ethereum στη συγκεκριμένη περίπτωση, μέσω μιας διεπαφής, που προσφέρει το ίδιο το Ethereum σε μορφή βιβλιοθήκης ονομαζόμενης web3, και επιτρέπει την πρόσβαση στην ίδια του τη δομή ανακτώντας δεδομένα από αυτήν.

Indexes for Blockchain Data

Abstract

Blockchain is a brand new technology, which has enormous application capabilities in a variety of industries. It became well-known as the basic technology underlying the creation of the digital cryptocurrency Bitcoin back in 2008. However, the areas of the current technology that can be applied are not limited to the financial sector. The subject of this diploma thesis is to study this technology, but also build database, thus providing and easy and fast way to search for information based on an index. More specifically, it initially explores and analyzes the concepts of Blockchain technology and identifies the technical and financial implications provided by the Blockchain technology based on the most important existing platforms such as Bitcoin, Ethereum, Ripple, Hyperledger and others. Given the fact that the Blockchain is an ever-growing list of records linked via cryptography, it emerges the problem of searching for information distributed into it. For this purpose, and index based on the key-value model is implemented using the distributed Redis database, storing the data we extract from a Blockchain platform, in this case we use Ethereum through an interface that Ethereum itself offers in the form of a library called web3, allowing access to its own structure and retrieving data from it.

Acknowledgements

First of all, I would like to express my deepest appreciation to my supervising professor Dimitrios Katsaros, for his guidance and invaluable support, which has been defining for the implementation of this diploma thesis.

I would also like to thank my friends and fellow students for the help and the support they have provided to me all these years of my studies, and the experiences and memories we have shared which can hardly be forgotten.

Finally, I would like to thank my family for having always been there supporting and helping in every possible and amazing way, and they will always be the most important thing I have in my life.

Contents

Abstract **iv**

Acknowledgements **v**

1 Introduction **1**

1.1 Importance of the problem 1

1.2 Approach 2

1.3 Thesis structure 2

2 Background **4**

2.1 Ledgers 4

2.1.1 Centralized Ledgers 4

2.1.2 Distributed Ledgers 5

2.2 Decentralized vs Distributed 6

2.3 History and Background 7

2.4 Web 3 9

2.4.1 Stack Layers 9

2.4.2 Architecture 16

2.4.3 JSON-RPC 18

2.4.4 web3 Library 19

2.4.5 Web3 Provider 20

2.5 REST API 20

3 Blockchain Infrastructure **23**

3.1 Blockchain Categorization 23

3.1.1 Permissionless Model 23
3.1.2 Permissioned Model 24
3.2 Blockchain Infrastructure 25
3.3 Consensus Models 35
3.3.1 Proof of Work Model 37
3.3.2 Proof of Stake Model 40
3.4 Blockchain Conflicts and Resolutions 42
3.5 Forking 44
3.5.1 Soft Forks 44
3.5.2 Hard Forks 45
3.5.3 Cryptographic Changes and Forks 47
3.6 Smart Contracts 47
3.6.1 Smart contract Applications 49

4 Ethereum Platform 51

4.1 System Overview 51
4.1.1 Blockchain component 51
4.1.2 Peer-to-Peer network component 52
4.1.3 EVM component 53
4.2 Merkle Tree Structure 53
4.3 Scalability 55
4.3.1 Plasma Chains 56
4.3.2 Sharding 56
4.4 Protocol Basics 57
4.4.1 Specification 57
4.4.2 Data Transfer 58
4.5 Transactions 59

5 Implementation 64

5.1 Combining existing parts 64
5.1.1 Illustration of the System Architecture 64
5.2 Redis Storage 66

5.2.1 Redis Functionality 67

5.3 Ethereum Network 69

5.3.1 Blocks 72

5.4 Utilities and Configurations 74

5.4.1 Serialization 76

5.4.2 Address Indexer 77

5.4.3 Getting Addresses and Transactions 78

5.4.4 REST Server 79

5.4.5 Configurations 80

5.4.6 Error Module 81

5.4.7 The Logger 83

5.4.8 Connecting to a node 83

5.4.9 State 84

5.4.10 Loading Components 85

5.5 Running the Indexer 86

6 Conclusion 87

List of Figures

2.1 (a) Centralized Ledger (b) Decentralized Ledger. 5

2.2 (a) Decentralized network. (b) Distributed network. 6

2.3 Distributed DBMS Architecture. 7

2.4 The Web 3 Stack. 10

2.5 Anatomy of a Transaction on the Ethereum Blockchain. 15

2.6 Architecture of a Web 2.0 application vs that of a Web 3.0 application 17

3.1 Anatomy of a single block. 26

3.2 Transaction to transaction payment as used in Bitcoin 29

3.3 Generic Chain of Blocks. 36

3.4 Consensus finality violation resulting in a fork 43

3.5 The chain with block_n(B) adds the next block, the chain with block_n(A) is now orphaned, and longest chain wins 43

3.6 Old nodes, having the white blocks, will accept the black blocks, making a soft fork backwards compatible. The white chain will eventually "die out" as no white blocks are mined any longer when all nodes and mines are upgraded. 45

3.7 The black chain will not accept white blocks, and vice versa, making a hard fork not backwards compatible. Unlike a soft fork, the "old chain", here the chain of black blocks, does not "die out", because it still has enough miners and nodes using it. The chain splits into two separate chains, that share the same transaction history as before the split. 46

4.1 A simple Merkle tree 54

4.2 Sharding in Blockchain 57
4.3 State transition in Ethereum 59
4.4 Ethereum Tries 60
4.5 Relationship between the State Trie and an Ethereum block 61

5.1 Indexer interacting with the Ethereum Blockchain to get data 65
5.2 How Geth communicates with the user running the node and the network 66
5.3 Indexer module requesting data from the REST server, who in turn requests the data from the Redis Database 67

List of Tables

3.1 Examples of Input Values and Corresponding SHA-256 Message Values 31

5.1 Redis Functions used 68

5.2 SyncProgress gives progress indications when the node is synchronising with the Ethereum network. 71

5.3 Every value of the configuration file. 82

Chapter 1

Introduction

In this chapter, a preface is given in order to understand the aim of this thesis. In particular, it explains why indexing data which are scattered across the Blockchain is crucial as what strategy does this thesis follows, in order to index these data and provide an efficient way to query them.

1.1 Importance of the problem

Information relating to an entity in a distributed ledger, specifically the Blockchain, may be scattered throughout it with no index, thus making an extremely difficult task the search of these information. The blockchain technology is becoming more established and it is being used to represent transactions in the real world so the search requirements are growing. Moreover because of the time-ordered structure of the blockchain, the related data exists across multiple blocks and there is no build-in way to identify, group or query it. Thus an index providing the ability to search using specific terms across the blockchain will enhance the scope, power and usability of it. There are different levels of detail at which indexing can be accomplished. It is necessary to index the basic entities of the blockchain, meaning the accounts, blocks and transactions, at a low level. The total data stored across the blockchain requires the ability to locate and retrieve it, so indexing at this basic level is fundamental to high level querying of the blockchain.

1.2 Approach

At first, this thesis approaches the Blockchain technology overall and how it works by explaining its complex nature and structure. It breaks down the components of the Blockchain and analyzes each one of them. Moreover the Ethereum Blockchain platform is discussed in detail because it is used for the purpose of developing the indexing service. This service will index the Ethereum Blockchain and provide a REST API to query all transactions related to a given address and a Socket.IO subscription mechanism to be notified when those are indexed. The Go Ethereum implementation provides web3, which a Javascript programming library that is used to talk to an Ethereum node, providing also a convenient interface the RPC methods. The data extracted from the blocks of the Blockchain will be indexed in Redis, which is a distributed key-value database. For the indexer to work, either an Ethereum client can be used, such as Geth or Parity, syncing a full node locally, or Infura can be used, which is a hosted Ethereum node cluster, meaning that there is no need for a user to set up his own Ethereum node.

1.3 Thesis structure

This thesis is divided into 5 main chapters, each one including smaller sections and possibly subsections. Chapter 2 provides a background and history of the blockchain technology, describing its evolution through time from distributed ledger to the first true Blockchain implementation, the Bitcoin. Chapter 3 breaks down the Blockchain infrastructure analyzing its components individually and explaining how a Blockchain works in general. Chapter 4 focuses on a specific Blockchain implementation, the Ethereum Blockchain, which is used as the main platform for the integration of the indexing service. Explanations on how Ethereum works and information about the REST API are provided, necessary for one to understand the development process and the components used in this project. In Chapter 5 the implementation of the project is presented and how the data from the Ethereum node are being extracted and indexed in Redis and how the query using the REST API is executed. Finally, Chapter 6 concludes the thesis stating our contributions and the future work.

Chapter 2

Background

2.1 Ledgers

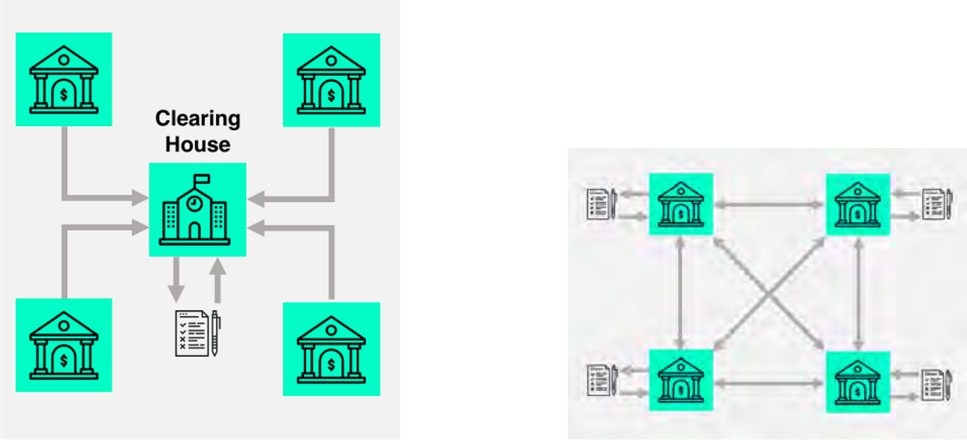
Ledger by definition is a book of record keeping all the financial transactions of one organization. In educational facilities is often called *register*.

2.1.1 Centralized Ledgers

A centralized ledger (Fig. 2.1a) contains all the accounts for recording transactions relating to a company's assets, revenue, liabilities, expenses and owners' equity. Anything that holds financial value needs a ledger. Computerized ledger came into existence, i.e. **Enterprise recourse planning** (**ERP**), the general ledger works as a central repository for accounting data transferred from all sub-ledgers cash management, fixed assets and purchasing. That ledger is the backbone of any accounting system which holds financial and non-financial data for an organization. For example the bank, which is a centralized asset ledger, has total control over which transactions are posted on it. All transactions are controlled by a single entity, that being the bank, and if the entity-in-charge has malicious intent it can do serious harm to its clients.

2.1.2 Distributed Ledgers

A distributed ledger (Fig. 2.1b) is a database that is consensually shared, replicated and synchronized across network spread across multiple sites or institutions. It allows transactions to have public ”witnesses”, thereby making a cyberattack more difficult. The participant at each node of the network can access the recordings shared across that network and can own an identical copy of it. Further, any changes or additions made to the ledger are reflected and copied to all participants in a matter of seconds or minutes. When a ledger update happens, each node constructs the new transaction, and then the nodes vote by consensus algorithm on which copy is correct. Once a consensus has been determined, all the other nodes update themselves with the new, correct copy of the ledger. The operation of distributed ledger technology (DLT) may involve the use of a public or private network potentially containing digitally represented assets, where the participants on the network conduct and verify transactions, and record related data on the network in an encrypted format. Distributed ledgers have many advantages over centralized ledgers. These advantages involve users being in control of their own information and transactions, data being complete, consistent, accurate and widely available and because of decentralization there is no central point of failure. Underlying the distributed ledger technology is the blockchain, which is the technology that underlies bitcoin.

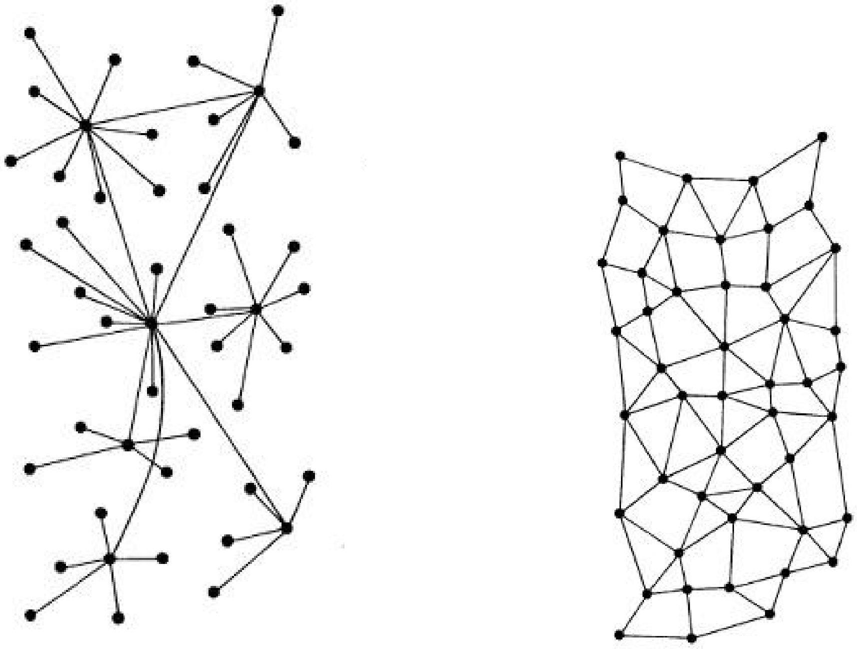


(a) (b)

Figure 2.1: (a) Centralized Ledger (b) Decentralized Ledger.

2.2 Decentralized vs Distributed

A clear distriction between decentralized and distributed systems is made by Baran (1964). In a decentralized system (Fig. 2.2a) there are some central-hierarchical nodes. Decentralized means that there is no single point where the decision is made. Every node makes a decision for its own behavior and the resulting system behavior is the aggregate response. However in a distributed system (Fig. 2.2b) there are no central nodes. Distributed means that the processing is shared across multiple nodes, but the decisions may still be centralized and use complete system knowledge. The difference that essentially separated decentralized from distributed databases is that decentralized databases are a collection of independent databases while a distributed database is a single logical database that is spread physically across computers in multiple locations that are connected by a data communication link.



(a) (b)

Figure 2.2: (a) Decentralized network. (b) Distributed network.

Tanenbaum and Van Steen define distributed computing systems as "A collection of independent computers that appears to its users as a single coherent system" [1]. This definition is further explored by Ozsu and Valduriez as "a number of autonomous processing elements (not necessarily homogeneous) that are interconnected by a computer network and that cooperate in performing their assigned tasks" [2]. Blockchain technology adheres to both definitions, as it appears as single system to its users and cooperate in performing an assigned task in a network. Therefore, Blockchain is a form of a distributed computing system. Ozsu and Valduriez define a distributed database (Fig. 2.3) as "a collection of multiple, logically interrelated databases distributed over a computer network", a distributed database management system as "the software system that permits the management of the distributed database and makes the distribution transparent to the users". The combination of these two terms is called a Distributed Database System. Blockchain is a collection of multiple, logically interrelated databases distributed over a network. Furthermore, it adds a layer of software that manages these databases. Therefore, blockchain is also a form of a Distributed Database System.

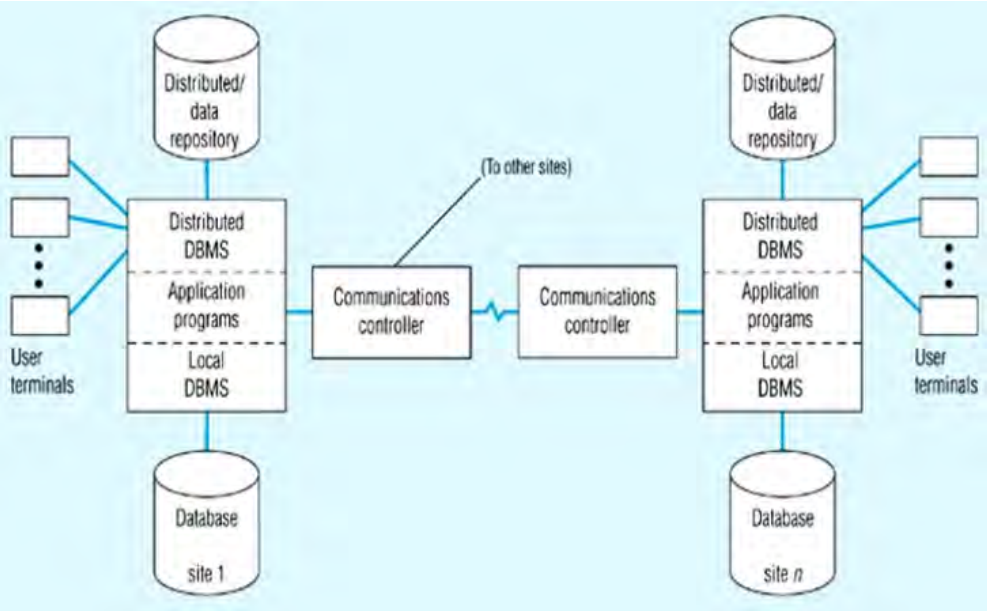


Figure 2.3: Distributed DBMS Architecture.

2.3 History and Background

As mentioned before the technology became widely known in 2008 with the invention of the Bitcoin. However, the used ideas have their roots in the late 1980 and 1990s. In 1991 a research on cryptographically secured chain of blocks and how to electronically sign digital documents was conducted by Haber and Stornetta [3]. In 1998 a paper called The Part-Time Parliament was published by Leslie Lamport. The paper describes a consensus model for reaching agreement on a result in a network of computers where the computers or network itself may be unreliable. Combining the above concepts Satoshi Nakamoto published his paper “Bitcoin: A peer-to-peer electronic cash system” [4], and a year later in 2009 the Bitcoin network was established. This paper contained actually the blueprint that most modern cryptocurrency schemes follow and Bitcoin was the predecessor of many blockchain applications. Before the invention of Bitcoin, many other electronic cash schemes existed, such as eCash conceived by David Chaum in 1982, but it was the use of Blockchain that enabled Bitcoin to be implemented in a distributed way such that no single user controlled the money and there was no single point of failure. Its primary benefit was to enable direct transactions between users without the need for a trusted third party or intermediary such as banks. Bitcoin introduced the concept of miners, users who essentially manage to publish new blocks and maintain copies of the ledger. The automated payment of miners enabled distributed administration of the system. By using a consensus-based maintenance and the Blockchain, it can be ensured that only valid transactions and blocks are added to the Blockchain. Because of the Bitcoin, and the Blockchain in general, anonymity, it is crucial for mechanisms that create trust in an environment where users are strangers to each other. Before the use of the Blockchain, this trust was achieved through intermediaries trusted by both parties. However to reach such a level of trust without intermediaries, four key characteristics are deployed within the Blockchain network, described below:

- **Distributed** – the Blockchain can be distributed. This allows for scaling the number of nodes of a Blockchain network to make it more resilient to attacks by malicious users. A higher number of nodes in the network decrease the ability of the malicious users to tamper with the consensus protocol the Blockchain uses.

- **Ledger** – the technology uses an append only ledger to provide full transactional history. Unlike traditional databases, transactions in a Blockchain cannot be overridden.

- **Secure** – the data integrity is ensured by Blockchain's cryptographic secure nature, meaning that everything contained within the ledger will not be tampered with.
- **Shared** – the ledger is shared amongst multiple participants, thus providing transparency across the node participants in the Blockchain network.

Some Blockchain networks allow anyone who wants to join to anonymously create an account and participate (these blockchain networks are called permissionless), so the above capabilities deliver a lever of trust amongst parties with no prior knowledge of one another. This level of trust allows individuals to transact directly, resulting in transactions being delivered at lower costs and faster.

2.4 Web 3

2.4.1 Stack Layers

Today's internet is a stateless internet, its participants can't hold their own state, nor transfer it from one to another, natively. Blockchains gave a way to hold state in a digitally native way. This new fundamental capability is often referred to as Web 3.

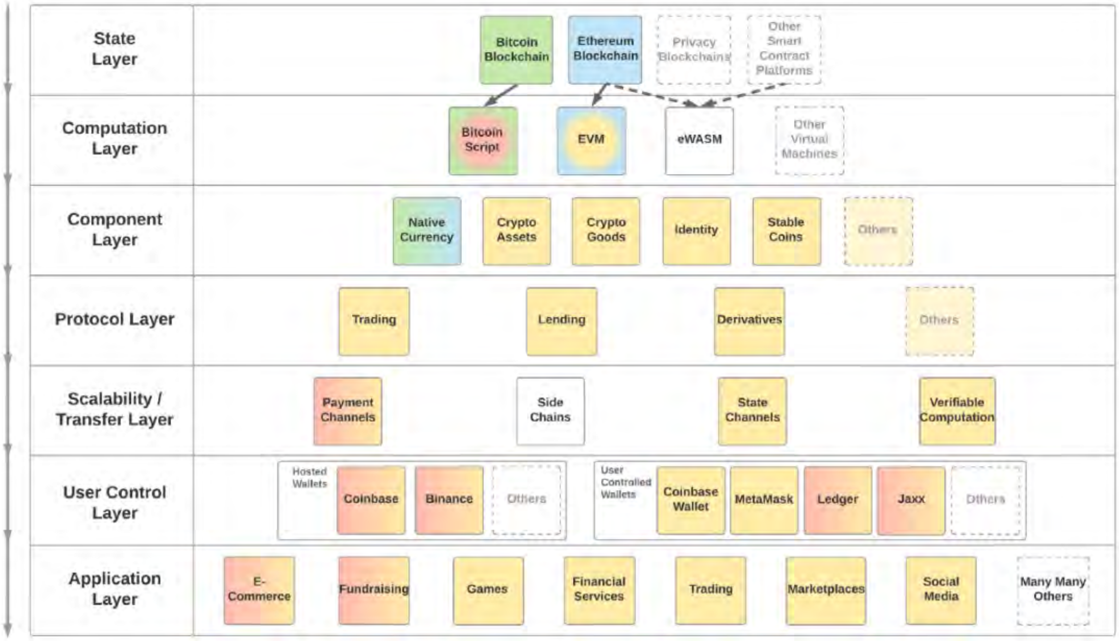


Figure 2.4: The Web 3 Stack.

The layers in the framework above start from the top and get build downwards. Compatibility between modules is represented by colors. For example, Stable Coins (yellow), are compatible with EVM (blue to yellow) but not with the Bitcoin Script (green to red). EVM is compatible with the Ethereum Blockchain (blue), but not with the Bitcoin Blockchain (green). Such modularity is crucial to the robustness of Web 3, because upgrading one of the layers should not require a complete rewrite of everything below it.

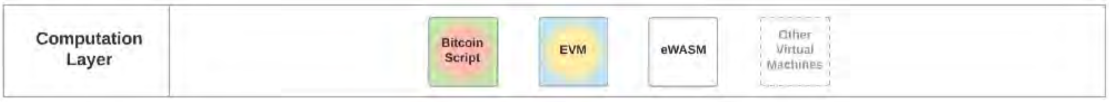
State Layer

The state layer preserves the state of all that happens below it. It is provided by blockchain infrastructure and allows for any participant to take part as long as they follow the rules of the preferred network. This layer can either be a public or private layer. There are technical differentiators between public and private layers, and they will be discussed on a later chapter.

State Layer
Bitcoin Blockchain
Ethereum Blockchain
Privacy Blockchains
Other Smart Contract Platforms

Computation Layer

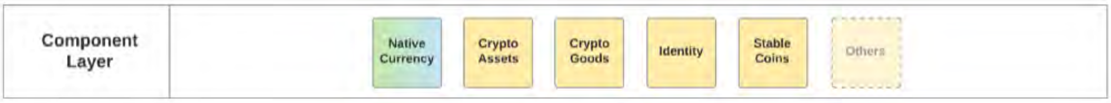
Instructions to computers are given with software. The Web 3 Computation Layer allows users to instruct the State Layer to do what they want. Not every Computation Layer allows for anything to be done. For example the **Ethereum Virtual Machine** (EVM) is a full Turing Complete machine and allows for any arbitrarily complex computation to be executed by a state layer that supports EVM. On the other end **Bitcoin's Script** is very limited and allows very little beyond transaction orders.



The choice of Computation Layer for blockchain developers is a key one as it determines which blockchains a given application can run on. For example an application compiled to EVM can run on the Ethereum Blockchain but not on the Bitcoin Blockchain. The Ethereum Foundation is working to change Ethereum's default Computation Layer to a technology called **eWASM**, meaning that an app compiled to **eWASM** can theoretically run on both Ethereum and another **WASM** compatible blockchain.

Component Layer

Combining the two previous layers, State Layer with the Computation Layer, the design space for new types of digital assets increases 1000 times. There are implementations that have potential to be build entire sub-economies on top of them. Components are build on the Computation Layer are reusing standardized smart contract templates.



For example, *OpenZeppelin* is a well established resource to access such templates. Component's creators are required to issue new smart contracts onto the State Layer. Some examples of these components are presented below.

- **Native Currency:** It is a core part and a required one for any public Blockchain. It gives the right to any participant to pay the Blockchain and receive the desired service, in the form of a transaction. Such examples are *Bitcoin* or *Ether*.

- **Crypto Assets:** Exchangeable assets with a basic set of functionalities and associated metadata. The ICO rise because of them, as it allows anyone to create their own currency. Beyond currency, it allowed many other asset types to be digitized such as stocks or ownership rights. The most common standard is *ERC-20*.

- **Stablecoins:** They are Crypto Assets with a stable value, fixed with a source such as the value of EUR. A stablecoin claims to be an asset that prices itself rather than an asset that is priced by supply and demand. There are different types of practical and theoretical solutions. Such examples are *Dai* or *Reserve*.

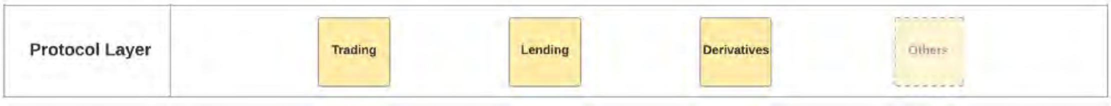
- **Crypto Goods:** They are non-exchangeable assets with a basic set of functionalities and a wider set of metadata associated with it. They are also known as **Non-Fungible Tokens** (**NFT**). It enables unique goods be digitalized such as art, game assets, collectible items, or access rights. The most common standard is *ERC-721*.

- **Identity:** An independent container for identity information. By definition, it provides very little valuable information about what it identifies. However, it allows claims to be associated with the container, which can come from a large pool of sources such as trusted third parties. Common standards are *ERC-725/ERC-735*. **Ethereum Naming Service** (**ENS**) is a relevant different type of identifier resembling the **DNS** protocol, offering a secure and decentralized way to address resources both on and off the Blockchain using simple, human-readable names. Also there exists some protocol proposals by *uPort*.

Protocol Layer

Components created on the State Layer need to come alive. There are certain functions that they have become the standard because they are fundamental and common to the lifecycle of these components. These protocols enable the formation of healthy markets for relevant components, like the physical world, only orders of magnitude cheaper and more efficient. Multiple different protocols have started to gain traction. These take the form of canonical smart contracts that are deployed by the team developing the protocol and called by each application that wants to apply the relevant function onto a component.

- **Trading:** If a component is to have value, it needs to be tradable. Trading protocols allow trading assets in a trustless way. There is a difference between *relayers* and *decentralized exchanges*, which take care of assets on a smart contract. Trades promoted by trading protocols never take care of the traded assets. Such example are *Kyber Network* and *0x*.
- **Lending:** Lending increases the efficiency of any asset as it enables a return on the investment, which may have been zero. With this protocol one can lend money or another digital asset to another, without the restriction of borders. Examples of this protocol are *Dharma* and *ETHLend*.



Transfer Layer

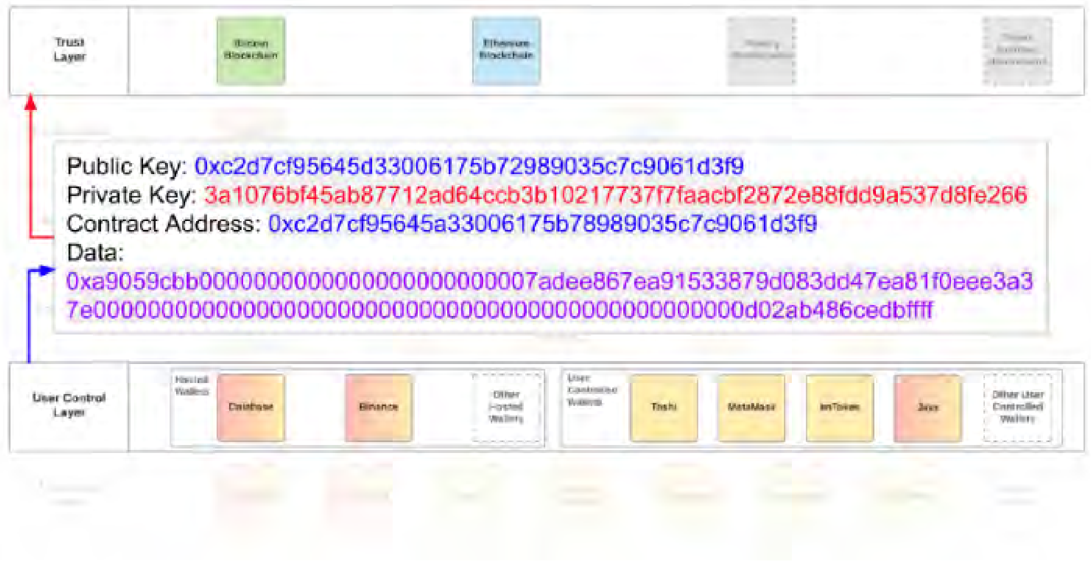
One significant Blockchain problem is its scalability issues. Bitcoin has transaction capacity of 7 transactions per second and Ethereum a capacity of 15 transactions per second. A different layer, known as Layer 2 scalability, for the transfer of state is required to support a robust topology. These scalability solutions need to be compatible with the Computation Layer of the Blockchain. There are proposals for how can this be achieved.

- **Payment Channels:** These channels allow only transfers of a given native currency. The transfer is done via verifiable signatures, attached to the transactions on the State Layer. Examples are Bitcoin's *Lightning Network* and Ethereum's *Raiden*
- **Side Chains:** Allows for transfer of any state. It is done by other Blockchains that are compatible with the main chain. The side chain must be able to interact with the Computation Layer on the main chain. The side chain may be centrally or privately managed infrastructure. Some examples are *PoA Network* and *Loom Network* for **EVM**.
- **State Channels:** These channels allow for the transfer of any state. Signatures are attached to transactions at the State Layer. Some examples are *Counterfactual* and *Celer Network* for **EVM**.

Scalability / Transfer Layer
Payment Channels
Side Chains
State Channels
Verifiable Computation

User Control Layer

Until this layer, it is impossible for a user to utilize any of the functionality created. The main functionality of this layer is to manage a user's private keys and be able to sign transactions on the State Layer. The state of a user's account changes by a transaction at the State Layer, that way being at the core of users' interaction with Web 3 applications.



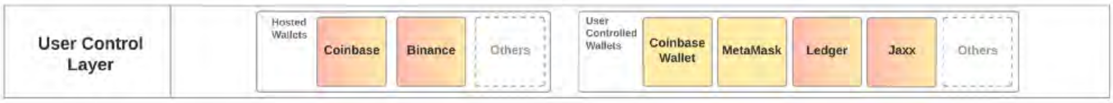
Public Key: 0xc2d7cf95645d33006175b72989035c7c9061d3f9
Private Key: 3a1076bf45ab87712ad64ccb3b10217737f7faacbf2872e88fdd9a537d8fe266
Contract Address: 0xc2d7cf95645a33006175b78989035c7c9061d3f9
Data:
0xa9059cbb0000000000000000000000007adee867ea91533879d083dd47ea81f0eee3a3
7e000000000000000000000000000000000000000000000000d02ab486cedbffff

Figure 2.5: Anatomy of a Transaction on the Ethereum Blockchain.

Wallets are of two types the first called hosted wallets and the second user controlled wallets.

- **Hosted Wallets:** They manage the digital assets on behalf of the user by controlling a limited set of balances on the State Layer. These may group users' digital assets into aggregated accounts and manage individual users' states themselves, outside of the State Layer. This is possible if the assets are of monetary value, however Web 3 applications have brought an increasing number of states making it more complex.

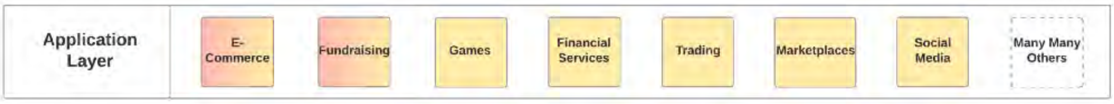
- **User controlled Wallets:** They provide a more flexible and direct way to utilize all the complex operations of Web 3. A User Controlled Wallet takes care of a user's private keys and local signing of each transaction. This means that the wallet software does not replicate the user's private keys allowing a third party to submit transactions on the user's end.



There are some front-end libraries for exposing all available functionality to applications accesses through this layer. In this thesis, such a library is used, called web3.js described in detail later.

Application Layer

The majority of the activity on Web 3 will be through third party applications build on all the layers below. Applications build on Web 3 have different properties and requirements than traditional web applications, and are referred to as **decentralized applications**, or **DApps**.



2.4.2 Architecture

A version of Web 2.0 architecture includes a client software, and a suite of servers providing the content and logic, all controlled by the same single entity. This entity has control over who access its servers' contents, as well as the records of which users own what. There are many examples in the course of years where companies have changed the rules on their users or stopped the service, with users having no power to preserve the value. However, Web 3.0 architecture leverages what's enabled by a universal State Layer. It does this by allowing two things:

1. Applications are allowed to place some or all of content and logic onto a pubic Blockchain. Contrary to Web 2.0 this is public and accessible by anyone
2. Users are allowed to maintain direct control over this content and logic. Contrary to Web 2.0 usually they don't need accounts to interact with blockchain's content

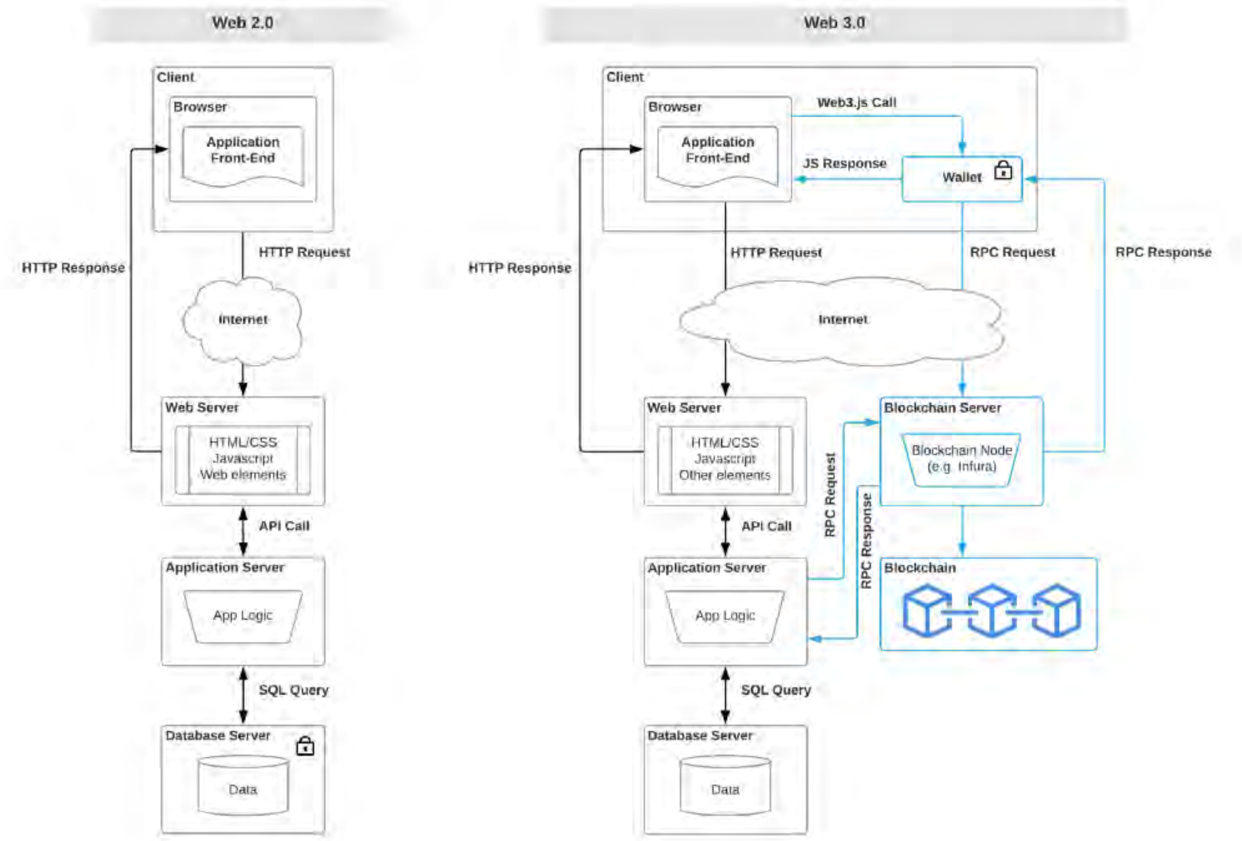


Figure 2.6: Architecture of a Web 2.0 application vs that of a Web 3.0 application

Web 3.0 applications allow this with the help of two key infrastructure pieces.

- **Blockchain Node:** There are two types of users that monitor and interact with the Blockchain – miners and nodes. Miners directly maintain and run the Blockchain. Nodes, on the other hand, monitor and submit transactions to the blockchain. When a wallet wants to submit a transaction to the Blockchain, or query state information from the Blockchain, it makes a call to the node provider, such as *Infura*. Applications' app servers can also interact with the node provider themselves by making similar RPC calls.

- **Wallets:** They interact with the main client front-end to allow a seamless user experience . They do this by allowing applications to send requests to the wallet itself using standard libraries, such as web3.js. A sample web3.js call can be a payment request, asking the user to confirm that the wallet can send a specified amount of digital currency to the application's address. When the user accepts, the wallet first lets the application front-end know with a response, so it can present a "Payment Submitted" message, and then the wallet makes and RPC call to the Blockchain server to submit the approved transaction to the Blockchain. This is where the Blockchain Node comes into play.

```
// Get the contract instance using your contract's abi and address:
const contractInstance = web3.eth.contract(abi).at(contractAddress);

// Get user s web3 address
var sender = web3.eth.accounts[0];

// Call a function of the contract:
contractInstance.someFunction({ from: sender, value: someValue, gas: limit }, (err, res) => { /** do something with results **/ });
```

Code Listing 2.1: Sample web 3 code allowing a DApp to call a smart contract function from a user's wallet

2.4.3 JSON-RPC

JSON is widely known nowadays because is the standard file-format to transmit data objects in the web. It is a lightweight data-interchange format. It can represent numbers, strings, ordered sequences of values, and collections of name/value pairs.

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. Primarily this specification defines several data structures and the rules around their processing. It is transport agnostic in that the concepts can be used within the same process, over sockets, over HTTP, or in many various message passing environments. It uses JSON as data format.

2.4.4 web3 Library

The Ethereum network is formulated out of nodes, which each contain a copy of the Blockchain. When a user wants to call a function on a smart contract, he needs to query one of these nodes and tell it the address of the smart contract, the function he wants to call and the variables to be passed to that function. However Ethereum nodes, like *Geth* or *Parity*, communicating with a language called **JSON-RPC**, that is difficult for human to read.

```
{
"jsonrpc":"2.0",
"method":"eth_sendTransaction",
"params":[{"from":"0xb60e8dd61c5d32be8058bb8eb970870f07233155","to":"0xd46e8dd67c5d32be8058bb8eb970870f07244567","gas":"0x76c0","gasPrice":"0x9184e72a000","value":"0x9184e72a","data":"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"}],
"id":1
}
```

Code Listing 2.2: Example of eth_sendTransaction method

The library used in this project is called *web3.js* [5], and it provides a way to interact with an Ethereum node from inside a JavaScript application. *Web3.js* also provides a convenient interface for the RPC methods, meaning that under the hood it communicates to a local node through RPC calls. The library works with any Ethereum node which exposes an RPC layer. *Web3.js* provides a *web3* object. This object contains two other objects. The first called **eth** (web3.eth) and it is used for Ethereum blockchain interactions. The second is called **shh** (web3.shh) and it is used for Whisper interactions. Essentially whispers are a part of the Ethereum peer-to-peer (P2P) protocol suite that allows messaging between users via the same network the Blockchain runs on. Smart contracts do not have access because the protocol is separate from the Blockchain.

```
#!/usr/bin/env node

var Web3 = require('../index.js');
var web3 = new Web3();

web3.setProvider(new web3.providers.HttpProvider('http://localhost:8545'));

var coinbase = web3.eth.coinbase;
console.log(coinbase);

var balance = web3.eth.getBalance(coinbase);
console.log(balance.toString(10));
```

Code Listing 2.3: Example of setting a provider and getting the balance of an account

2.4.5 Web3 Provider

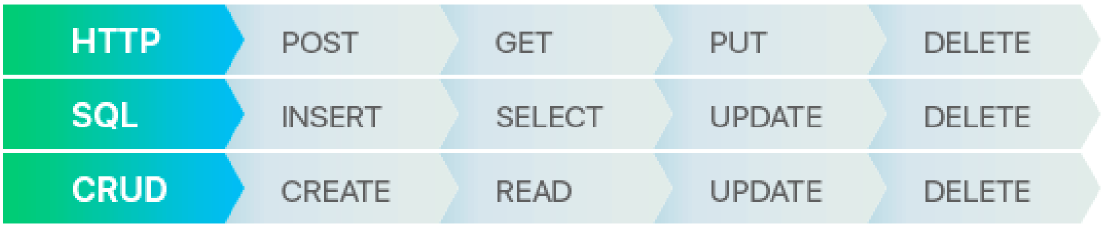
As already stated the Ethereum network is formulated out of nodes that all share a copy of the same data. Setting a Web3 Provider in **web3.js** tells the code which node it should be talking to handle the reads and writes. This process is equal to setting the URL of the remote web server for the API calls in a traditional web application. Usually as a provider the user host its own Ethereum node. However one also has the choice of a third-party service, called *Infura*, which essentially is a service that maintains a set of Ethereum nodes with a caching layer for fast reads, eliminating the need to set up and maintain a Blockchain node locally.

2.5 REST API

According to Wikipedia, "**REST** (**Representational State Transfer**) is a software architectural style that defines a set of constraints to be used for creating web services. Web services that conform to the REST architectural style, termed RESTful web services, provide interoperability between computer systems on the Internet." **RESTful** is the interpretation of this architectural style. That means, if a back-end server has REST API and the user makes client-side requests, from a website or an application, to this API, then the client is RESTful. REST relies heavily on the HTTP protocol. RESTful API comes down to four essential operations, each one of them using its own HTTP method:

- Creating new data (**POST**)
- Receiving data in a convenient format, such as JSON, XML or YAML (**GET**)
- Updating data (**PUT**)
- Deleting data (**DELETE**)

These four methods are also the basic functions in databases so the fact that REST uses a common interface for both request and databases is one great advantage.



HTTP	POST	GET	PUT	DELETE
SQL	INSERT	SELECT	UPDATE	DELETE
CRUD	CREATE	READ	UPDATE	DELETE

Versioning

Versioning for REST APIs is important and should provided at all times. For example, in this project, the API is at the URL http://localhost:3000/thesis, changes are made to it at http://localhost:3000/thesis/v1.

Architecture Design

All resources in REST are entities. They can be independent like **GET /localhost:3000/height** which gives us the processed Blockchain height, or they can be depentant models that rely on their parent model such as **GET /localhost:3000/address/:hash** which retrieves an address transaction by its hash. The colons (:) on a path denotes a variable. They should be replaced by values with actual values of when the request is sent. The above examples are demonstrating **GET** which retrieves the entity the user requested. A successful request returns an entity representation combined with status code 200 (OK), or a status code based on the HTTP response status codes. In this thesis project, we want just to retrieve data from the Blockchain so only the **GET** request can be used.

Chapter 3

Blockchain Infrastructure

This chapter gives an overview about how a Blockchain network works. It analyzes the very different categories of a Blockchain, and it breaks down its core parts, and how the work together and how the Blockchain is formed when everything comes together.

3.1 Blockchain Categorization

Blockchain networks can be categorized based on their permission model, which determines who can maintain them (e.g. publish blocks). If anyone can publish new blocks it is called permissionless. If only a number of users can publish blocks, it is called permissioned. Put it simply, a permissioned blockchain network is like a corporate intranet, while a permissionless blockchain network is like the public internet. Permissioned blockchain networks are often referred to as consortium blockchains.

3.1.1 Permissionless Model

Permissionless blockchain networks are decentralized ledger platforms open to anyone mining blocks, without needing permission from any authority. Since anyone has the right to publish new blocks, this results in the property that anyone can read the blockchain and issue transactions (these transactions are included within the published blocks). Any user within this specific network can read or write to the ledger. Since permissionless blockchain networks are open to everyone, malicious user may as well attempt to publish corrupted blocks or tamper with the validity of the chain. However, for this to be avoided, these networks deploy consensus algorithms that require users to expend or maintain resources when attempting to publish new blocks. This prevents malicious users from corrupting the system. These consensus models include various methods such as proof of work or proof of stake. Using these methods, the consensus systems reward the publishers of the blocks (miners) with the native cryptocurrency, as a way to deal with the malicious behaviors.

3.1.2 Permissioned Model

Permissioned blockchain networks are the ones where mining new blocks must be authorized by some authority (decentralized or centralized). In these types of blockchains there is the need of special permission to read, access and write information on them. They also may allow anyone or restrict access to only authorized participants to submit transactions. These blockchain networks also use consensus models for mining blocks but these methods often do not require the expense of some resource like permissionless blockchains. This is because the identity of each blockchain participant is required and known, maintaining the level of trust with each other. Consensus models in permissioned blockchains are usually faster and computationally cheaper. Such blockchains are also popular among industry-level enterprises and businesses, for which security, identity and role definition are important. Permissioned blockchains may also be used by organizations that wish to work together but not fully trust one another. They can establish such a network and invite business partners to record their transaction on a shared distributed ledger. They can determine the consensus model to be used, based on their mutual trust. Beyond trust permissioned blockchains provide transparency. Moreover some degree of privacy in transactions may be obtained because some permissioned networks provide the ability to selectively reveal transaction information based on participant's identity. For example, it could be that the blockchain records that a transaction between two blockchain network users took place, but the actual contents of transactions is only accessible to the involved parties.

3.2 Blockchain Infrastructure

At a high level Blockchain technology utilizes computer science mechanisms and cryptography, such as cryptographic hash functions, digital signatures, asymmetric-key cryptography, mixed with record keeping concepts, such as append only ledgers. The main components of blockchain are introduced in this section and are: blocks, addresses, transactions, cryptographic hash functions, asymmetric-key cryptography, ledgers, and how blocks are chained together.

Blocks

Blockchain users submit their transactions to the network via software such as digital wallets. The software sends these transactions to a node (or nodes) within the network. The chosen nodes may either be mining nodes or non-mining. The transaction is then propagated to the other nodes in the network, but still is not placed in the Blockchain. Most implementations of the Blockchain, send the pending transaction into a queue once it is distributed to nodes, and then waits until a mining node add it to the Blockchain. Transactions are added to the Blockchain when a block is published (mined) by a miner. A block (Fig. 3.1) contains the block header and the block data. The block header contains metadata for the block. The block data contains a list of validated transactions which have been submitted to the Blockchain. If the providers of the digital assets in each transaction (listed in transaction's "input" values) have each cryptographically signed the transaction and if the transaction is correctly formatted then validity can be ensured. This verifies that the providers had access to the private key which could sign over the available digital assets.

The other full nodes will check the validity of all transactions in a published block and discard the block if it contains invalid information. Note that different Blockchain implementations could define different data fields, however in most Blockchain implementations the data fields of a block are:

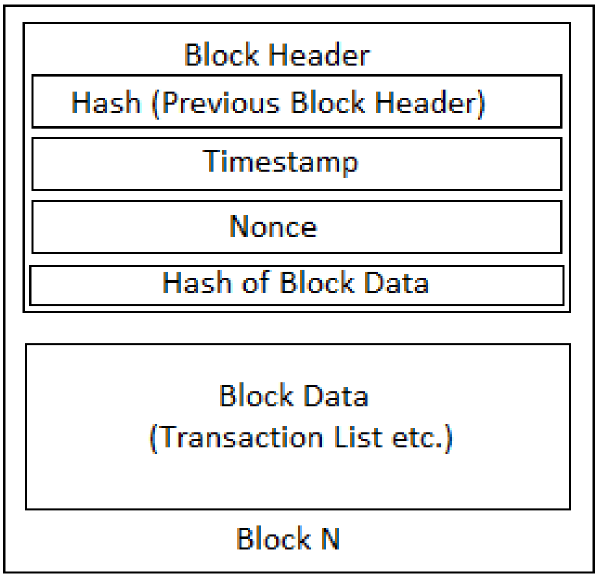


Figure 3.1: Anatomy of a single block.

- Block Header
  - The block number, also known as block height in some Blockchain networks.
  - The previous block header's hash value.
  - A timestamp.
  - The size of the block.
  - The nonce value. A number which is manipulated by the mining node to solve the hash puzzle. Some Blockchain networks use it for mining new blocks, however there are networks that may use it for another purpose other than solving a hash puzzle.
  - A hash representation of the block data (for example generating a Merkle tree, and storing the root hash, or by utilizing a hash of all the combined block data).
- Block Data
  - A list of transactions and ledger events included within the block.
  - Other relevant data.

Addresses

Blockchain networks make use of an address, which is an alphanumeric string of characters derived from the blockchain user's public key using a cryptographic hash function, along with some additional data, such as checksums and version number. Blockchain implementations make use of address as the "from" and "to" endpoints in a transaction. Addresses are not secret and are shorter than the public keys. One of the methods to generate an address is to create a public key, apply a cryptographic hash function to it, and convert the hash to text:

Public key →cryptographic hash function →address

Each blockchain implementation may use a different method to create an address, such as using different cryptographic hash function, Bitcoin for example uses SHA-256 algorithm in contrast to Ethereum that uses the KECCAK-256 or SHA-3 cryptographic function, or derive the public key from a generated private key. For permissionless blockchains, which allow anonymous account creation, a user can generate as many asymmetric-key pairs, and as a result as many addresses as desired, thus allowing a varying level of anonymity. Addresses act as the public-facing identifier in a blockchain network. However users aren't the only source of addresses within a blockchain. With the introduction of smart contracts from Ethereum it become a direct need for them to obtain an address for the purpose of accessing them once they have been deployed within the blockchain. For Ethereum, smart contracts are accessible via a special address called a contract account. This address is created when a smart contract is deployed (the address for a contract account is deterministatically computed from the address of the creator of the smart contract). This contract account allows the execution of the contract whenever it receives a transaction.

Private Key Storage (Wallet)

Users must manage and securely store their own private keys. For this purpose software has been develop to securely store them, instead of having users record them manually. This piece of software is called *wallet* and it can store public keys, private keys and associated addresses. It may also perform other functionalities, such as calculating the total number of digital assets a user may have. If a user loses the private key, then any digital asset associated with that key is lost, because it is computationally impossible to regenerate the same private key. If it is stolen, the attacker will have full access to every digital asset controlled by that key. That makes the security of private keys extremely important so users tend to use special secure hardware to store them. Storage of these private keys is extremely important. That is because data in the Blockchain cannot be modified, and once an attacker steals the private key and publicly transfers the associated funds to another account, the transaction cannot be undone.

Transactions

A transaction represents an interaction between users. For example a transaction represents the transfer of a specific amount of Bitcoins, or other cryptocurrency, between Blockchain users. For business scenarios, it could be a way of recording activities occurring on digital or even physical assets. Each block contains either zero or more transactions. A constant supply of new blocks, even with zero transactions, is critical to maintain security over the Blockchain, that means by having a constant supply of new blocks being published, malicious users are being prevented from "catching up" and producing a longer and altered Blockchain. The data which a transaction hold can be different for different blockchain implementations, but the mechanism for sending transactions is largely the same. A user sends information to the blockchain including some identifier, such as the sender's address, his public key, a digital signature and also transaction inputs and outputs.
Typically a cryptocurrency transaction requires the following information:

- **Inputs** – are usually a list of the digital assets the user wants to transfer. The transaction will reference the source of the digital asset, either the previous transaction where it was given to sender, or the origin event if he is sending new digital assets. The sender digitally signs the transaction for providing proof that they have access to the funds. The digital assets do not change since the input is a reference to past events. Value cannot be added or removed from existing digital assets in case of cryptocurrencies. Instead a single asset can be split into multiple new assets, each with lesser value, or it is possible for multiple assets to be combined to form fewer new assets, with greater value of course. The transaction output is the one who specifies the split or join of these digital assets.

- **Outputs** – are usually accounts that are the receivers of the digital assets along with the quantity they will receive. Each output specifies the number of the assets to be transferred, the identifier of the receiver and usually some set of conditions that must be meet to spend the specific value. Extra funds are explicitly sent back to the sender if the assets provided are more than required.

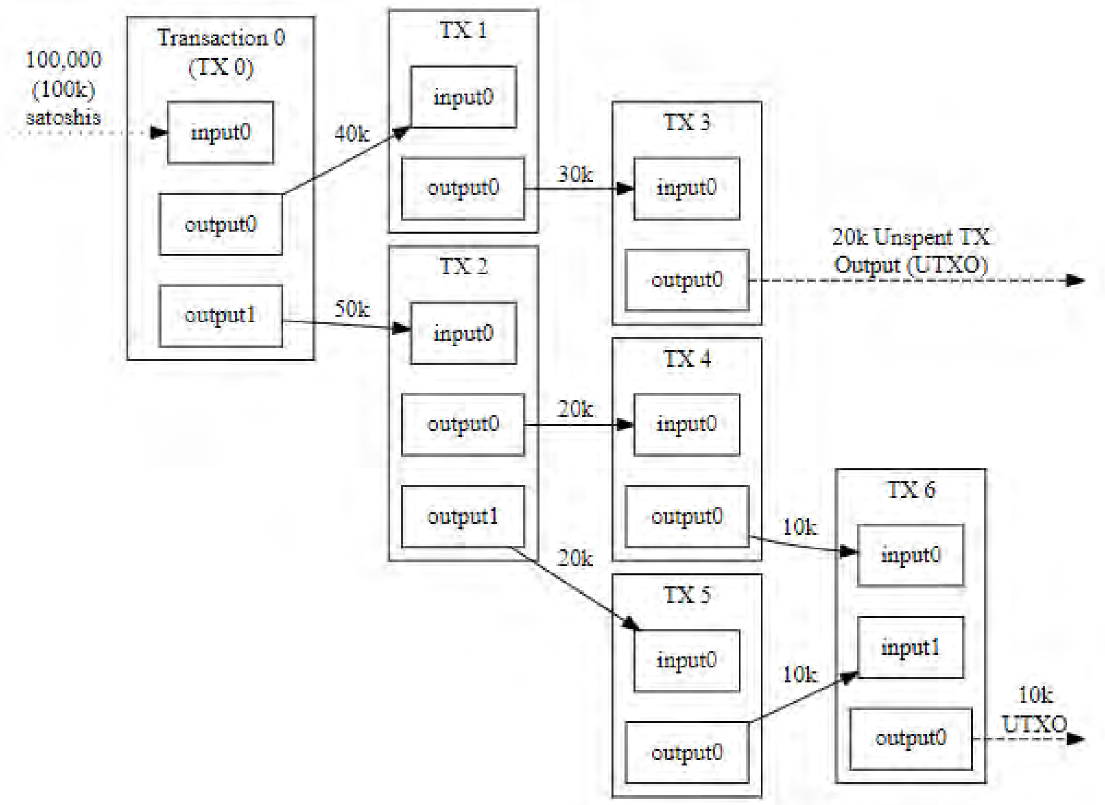


Figure 3.2: Transaction to transaction payment as used in Bitcoin

Generally transactions are used to transfer data not only digital assets. For example someone may simply want to publicly post data on the Blockchain. In the case of smart contract system, transactions can be used to send and process some data and then store the result on the Blockchain. Determining the validity and authenticity of a transaction is of utmost importance. That is because the validity ensures the transaction meets the protocol requirements and any data formats or smart contract requirements. The authenticity on the other hand, determines that the sender had access to the digital assets which have been send. Transactions are digitally signed by the sender's private key and can be verified using his public key.

Cryptographic Hash functions

One of the most important component of the blockchain is the cryptographic hash functions. Hashing is the method of applying a cryptographic hash function to data, which calculates a relatively unique output (called message digest) for an input of nearly any size (e.g. a file, an image or a text). It allows to independently take input data, hash that data, and get the same result, if and only if the was no change in the data. Even the smallest change to the input data will result in a completely different output. An example is shown in the table below.

Generally, according to Wikipedia, the security of cryptographic hash functions can be seen from three different angles: pre-image resistance, second pre-image resistance, and collision resistance.

- **Pre-image resistance**. This means that they are one-way functions, and it is impossible to compute the initial input value given the output value. For example given a hash h find a message m such that $h = hash(m)$.

- **Second pre-image resistance**. This means that given a specific input m_1 it should be impossible to find another input m_2 such that $hash(m_1) = hash(m_2)$. Put in simple words it is impossible given a specific input to find a second input which produces the same result.

- **Collision resistance**. This means that it must be impossible to find two different inputs that hash to the same output. Mathematically put there should be impossible to find m_1 and m_2 such that $hash(m_1) = hash(m_2)$.

A hash function used in many Blockchains, such as Bitcoin, is the **Secure Hash Algorithm** (**SHA**) with an output size of 256 bits (SHA-256). Due to hardware support in many systems this algorithm can be calculated extremely fast. SHA-256 has an output of 32 bytes, generally displayed as a 64-character hexadecimal string. This means that there are $2^{256} = 10^{77}$ possible hash values.

Input Value	SHA-256 Message Value
1	0x6b86b273ff34fce19d6b804eff5a3f5747ada4eaa22f1d49c01e52ddb7875b4b
2	0xd4735e3a265e16eee03f59718b9b5d03019c07d8b6c51f90da3a666eec13ab3
Blockchain Thesis	0xd4735e3a265e16eee03f59718b9b5d03019c07d8b6c51f90da3a666eec13ab3

Table 3.1: Examples of Input Values and Corresponding SHA-256 Message Values

Having said that it is clear that since there are an infinite number of possible input values but a finite number of possible output values, it is possible but highly unlikely to have a collision where $hash(m_1) = hash(m_2)$, meaning two different inputs produce the same output). SHA-256 is said to be collision resistant, since to find a collision in the algorithm one must execute it on average about 2^{128} times, which is 340 undecillions or roughly 3.402 x 10^{38}. To put this into perspective, the hash rate (hash per second) of the entire Bitcoin network in 2015 was 300 quadrillion hashes per second (300.000.000.000.000.000 / s). At that rate, it would take the entire Bitcoin network roughly 35.942.991.748.521 (roughly 3.6 x 10^{13}) years to manufacture a collision (note that the universe is estimated to be 1.37 x 10^{10} years old as estimated by measurements made by the Wilkinson Microwave Anisotropy Probe). Even if any such input m_1 and m_2 that produce the same hash, it would be very unlikely for both inputs to be valid in the context of Blockchain, meaning to be both valid transactions. Within the blockchain, hash functions are used for many tasks, such as:

- Address creation
- Unique identifiers creation
- Securing the block header – a mining node will hash the block header. If the Blockchain is using a proof of work consensus model, the node will need to hash the block header with different nonce values, until it solves the cryptographic

puzzle. The current block header's hash will be included within the next block's header, where it will secure the current block header data.

- Securing the block data – a mining node will hash the block data, creating a hash that will be stored within the block header.

The block header also includes a hash representation of the block data, the block data itself is also secured when the block header hash is stored in the next block.

Cryptographic Nonce

Blocks in the blockchain have a field called *nonce* which stands for *number used only once*. Nonce is an integer number and along with the block number, block data and previous hash serves as an input for the SHA-256 function to calculate the current block's hash. Unlike other components of a block, nonce is designed to be totally under our control. This means that we have a mechanism to vary the hash of the current block while keeping the data inside it intact. Miners compete to find a nonce which will generate a valid hash for the upcoming block. The one that finds it first is allowed to add the block to the Blockchain and gets a reward (e.g. bitcoins, ethers etc).

Asymmetric-key cryptography

Blockchain uses asymmetric-key cryptography (*public-key cryptography*).
Asymmetric-key cryptography uses a pair of keys: a private and a public key that are mathematically related to each other. The public key is made public without reducing the security of the process, however the private key must remain secret for the cryptographic protection of the data. Even though these two keys are related to each other, the private key cannot be determined by the public key. The encryption is done using the private key and the decryption using the public key. The reverse process can also be done, meaning that one can encrypt using the public and decrypt with the private key. This type of cryptography provides a trusting relationship between users who do not know or trust each other, by providing a mechanism to verify the integrity and authenticity of transactions while at the same time it allows

transactions to remain public. The transactions are digitally signed, meaning that the private key is used to encrypt the transaction such that anyone with the public key can decrypt it. Since the public key, as its name states is public, encrypting the transaction with the private key proves that the signer of the transaction has access to private key. Another way is to encrypt data with the user's public key, so only users with access to private key can decrypt it. Asymmetric-key cryptography finds use in many blockchain networks:

- Private keys are used to create addresses.
- Private keys are used to digitally sign transactions.
- Public keys are used to verify signatures generated with private keys.
- Asymmetric-key cryptography provides the ability to verify the possession of the private key of the sender of a transaction since he is digitally signs it.

Ledger

A ledger is essentially a collection of transactions. In the past years people used to keep track of the transactions using pen and paper. However, in modern times, ledgers have been stored digitally, often in large databases owned and operated by a centralized trusted third party. Ledgers like these, with centralized ownership can be implemented in a centralized or in a distributed way (one server for centralized access or coordinating a cluster of servers for distributed fashion). The recent years there is a growing interest in distributed ledgers. Blockchain technology enables such an approach using both distributed architecture as well as distributed ownership. The distributed architecture of blockchain networks involves a much larger set of computers than the typical distributed physical architecture. This growing interest in distributed ledgers is due to possible trust, security and reliability concerns in contrast to ledgers with centralized authority. For such a reason some of the differences between centrally owned ledgers and distributed ledger will be discussed below.

Centrally owned ledgers

- Centrally owned ledgers may be on a homogenous network, where all software and hardware infrastructure may be the same. Because of this, an attack on one part of the network will work on everyone, so the total system security and resiliency are significantly reduced.
- They may be lost or destroyed, so users of ledgers like these must trust that the central authority is backing up the system properly.
- The transactions are not made transparently and sometimes may not be valid, again the users must trust the central authority for validating each transaction.
- The transaction list may not be complete, central authority must include all valid transactions that have been received.
- The transaction data may have been altered, users must trust that past transactions have not been altered by the central authority.
- Such systems may be insecure, they may be breached and personal information be stolen, so it is crucial that once again the central authority implements the best security practices.
- Such ledgers may be located in specific locations. If a network problem were to occur in that location, services of the ledger may become unavailable.

Blockchain as a distributed ledger

- A blockchain network, on the other hand, is heterogeneous network, which means that the software and hardware infrastructure are all different. So an attack on one node is not guaranteed to work on other nodes.
- It is distributed, creating backup copies all updating and syncing to the same ledger data between peers. Every user can maintain their own copy of the blockchain. Whenever new full nodes join the blockchain, they try to discover other full nodes and request the full copy of the blockchain, thus making loss or destruction rather difficult.

- The blockchain must check that all transactions are valid. If a malicious node was transmitting invalid transactions, others would detect and ignore them, preventing them from propagating throughout the entire blockchain.

- The blockchain holds all accepted transactions within its distributed ledger. A new block must be build on top of previous blocks, so if the mining node did not include this reference (the previous hash) to the latest block, it is rejected.

- Utilizing cryptographic mechanisms such as cryptographic hash functions and digital signatures it achieves being tamper resistance.

- Due to its distributed nature, Blockchain has no centralized point of attack. All information a viewable by everyone. An attacker would need to individually target every user on the blockchain. It is almost impossible to target the blockchain itself because of the resistance of the other nodes present in the system. An attack on an individual node would harm only this node, the system overall would remain intact.

- Nodes of the blockchain may be found all around the world, because of the peer-to-peer (P2P) fashion. This reason alone makes it resilient to the loss of any node, or even multiple nodes.

Chaining Blocks

Blocks are chained together through the hash of the previous block's header which every block contains, thus forming the Blockchain. If a previous mined block were changed, it would have a different hash causing all the following blocks to also have different hashes since they include the hash of the previous block. This way altered blocks are easily detected and rejected. Figure 3.3 shows a generic chain of blocks.

3.3 Consensus Models

A key feature of the Blockchain is determining which user publishes the next block. The solution to this problem is the implementation of many possible con-

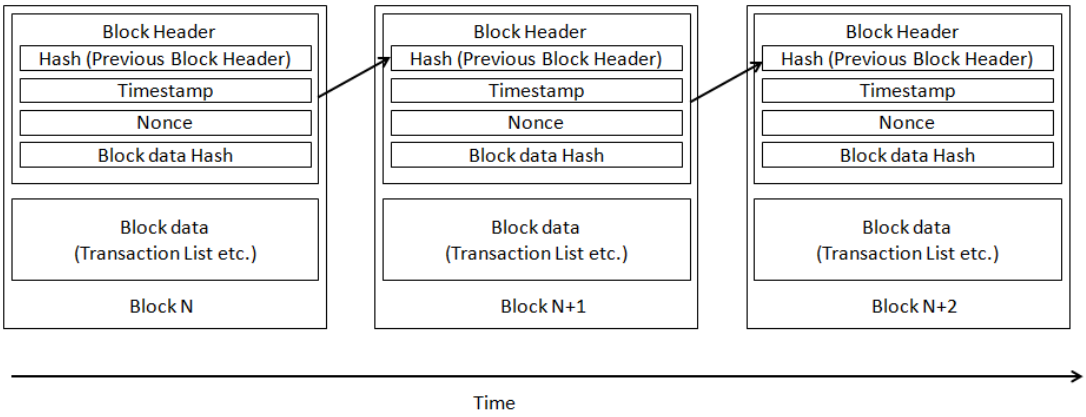


Figure 3.3: Generic Chain of Blocks.

sensus models. Blockchain networks are generally mining nodes competing at the same time to publish the next block. This usually grants them some sort of fee, like bitcoins or ethers. They are usually users who do not trust each other but they know each other by their public addresses. Financial gain is what motivates each node, rather than the well-being of the network. Question such as why would a user propagates a block that another is attempting to publish or how conflicts resolve when multiple nodes publish a block at approximately the same time arises. In such cases the Blockchain uses consensus models to enable distrusting users to work together. There is a pre-configured block called the genesis block, which exists in every Blockchain implementation and every block must be added after it, based on the agreed consensus model. Each must be valid and be able to independently validated by every user in the Blockchain. If there are more than two valid chains presented to a full node, the default mechanism in most Blockchains is that the “longer” chain is the correct one and will be adopted, that’s because it has the most amount of work put into it. This is a very frequent event with some consensus models. The software handles the following properties:

- The initial state of the system is agreed upon (the genesis block)
- The consensus model is agreed upon
- Every block is linked to the previous block by including the previous block header’s hash, except for the genesis block which has no previous block so the

previous block hash is set to all zeros.

- Every block can be independently verified.

A key feature of the Blockchain is that it eliminates the need for a trusted third party to provide the state of the system – every user within the system can verify its integrity. To add a new block all nodes must come to a common agreement over time. For permissionless Blockchains the consensus model must work even when there are malicious users in the network trying to corrupt or alter the Blockchain. In some permissioned Blockchains there may exist a level of trust between nodes so there may not be the need for a resource intensive consensus model to determine the addition of the new block in the chain. Generally if the level of trust increases, the need for intense resource usage for generating trust decreases. In the following sections, the two most used consensus models are presented.

3.3.1 Proof of Work Model

In the proof of work model (PoW), a user publishes the next block by being the first to solve a computationally intensive puzzle. That solution is the "proof" that work has been performed by the user. Checking if the puzzle's solution is valid is easy however solving the puzzle is rather difficult. This enables all other nodes to easily validate any proposed blocks, and reject others who would not satisfy the puzzle. One method is to require that the hash of a block header is less than a target value. Mining nodes make many small changes to their block header, for example they change their nonce, trying to find the correct hash value. The hash of the entire block header must be calculated by the mining node, however because this happens many times it becomes a computationally intensive process. The difficulty is adjusted by the modification of the target value to regulate the mining of blocks. Such an example is Bitcoin, which uses proof of work model, adjusting the mining difficulty every 2016 blocks so the publication rate of a block is around every ten minutes. Essentially by increasing or decreasing the number of zeros leading is adjusting the puzzle's difficulty level. By increasing the number or leading zeros, the difficulty increases, because any solution must be less than the difficulty level –

meaning there are fewer possible solutions. By decreasing the number, the difficulty decreases, because there are more possible solutions. Adjustments like these are made to maintain the computational difficulty of the puzzle, for maintaining the core security mechanism of the network. The puzzle difficulty is increasing as the number of mining nodes increasing. Moreover they try to ensure that no one can take over the block production, but this leads to the necessity of more resource consumption because of the computations. Due to this significant increase, some proof of work Blockchains, move the addition of the mining nodes to areas with remarkable cheaper power supply. An important feature of the proof of work algorithm is that the work of solving one puzzle is independent of the work that needs to be done for another puzzle. This means that when a user receives a completed and valid block from another user, they discard the work they have done so far and start building off the newly received block, because the other mining node will be building off it.

An example using the SHA-256 algorithm, which is the one Bitcoin uses, is demonstrated below, where a computer must find a hash value matching the difficulty level:

SHA-256("markos" + nonce) = hash value starting with "00000" (meaning difficulty = 5).

Above the text string "**markos**" is appended with a nonce value and a hash value is being calculated. The nonce values are numeric values. The solution to the puzzle is as follows:

```
   SHA-256("markos0") =
7989d85d7d71f2afe904bf306175270cf4fc682e17ef63c08f6287a0e0dfea96
(not yet solved)

   SHA-256("markos1") =
a5d41d22b7c0d464142a2916d780ebdfa4ef1c5c2647ffa55c89b6be28371fe3
(not yet solved)

   SHA-256("markos2") =
2bc658bcf21d5e4f4d870e38d593343c209d87a621ac1fbcd147aebe09ad998d
(not yet solved)

   ...
   SHA-256("markos256742") =
000006a7ea83acf484dbc634986ec7b104a8b44acef169b019105420b2e3941c
(SOLVED)
```

To solve the above puzzle took 256741 guesses and it was completed in 4,951 seconds. Each addition leading zero increases difficulty. For example if we increase the zero by one so difficulty equals 6 then it takes 43684682 guesses and it completes in 737,629 seconds (12,29 minutes):

```
   SHA-256("markos43684683") =
00000043d5da7661022f922bed3ce9030b16da88c95c2af95ae8f4dafb45a839
```

There is no way to cut down the completion time of this process; mining nodes cost computation effort, resources and time to find the correct nonce values. The mining nodes attempt to solve the puzzle to claim some sort of reward, usually an amount of cryptocurrency. The user is motivated to mine nodes because of the prospect of reward. Once a mining node has performed this work, the mined block is send with a valid nonce to full nodes in the blockchain. The receiver nodes verify that it fulfills the puzzle requirements, and then add the block to their copy of the blockchain and resend the block to their peer nodes. In this manner, the new block is quickly distributed over the network. Verification of the nonce is extremely easy task since only a single hash needs to be done to check to see if it is the correct solution to the puzzle. For many proof of work blockchains, the mining nodes tend to organize themselves into “pools” where they work together to find the solution to the puzzle and afterwards split the reward. This is done because the work can be distributed between nodes to share the workload and rewards.

3.3.2 Proof of Stake Model

The proof of stake (PoS) model is based on the idea that the more stake a user has invested into the system, the more likely they will want the system to succeed, and the less likely they will want to corrupt it. By stake it is usually meant an amount of cryptocurrency that the user has invested into the blockchain. Once staked, the cryptocurrency is no longer available for spending. Proof of stake blockchains use this amount of stake as a factor for mining new blocks. Thus the possibility of a user mining a new block is tied to the ratio of their stake to the overall blockchain amount of staked cryptocurrency. Ethereum is the most known blockchain network to use the proof of stake consensus model. With this consensus model, the need of intensive and resourceful calculations is eliminated. Some blockchains are giving rewards for block creation in advance, since fewer resources are needed. Systems like this are designed so that all cryptocurrency is already distributed among users rather than being generated at a constant pace. The reward for block publication is usually the earning of user provided transaction fees. How any blockchain uses the stake varies. Below four approaches are discussed below: delegate systems, coin aging systems, multi-round voting and random selection of staked users. Regardless the approach, the more the stake of a user the more likely he is to publish new blocks. When the choice of block publisher is through delegate system, the users vote for nodes to become mining nodes. Users' voting power is directly proportional to their amount of stake, so the larger the stake, the more weight their vote has. Those with the most votes become mining nodes and can validate and publish blocks. Users can also vote against a mining node, trying to remove them from the set of mining nodes. This voting process is continuous and competitive for a node to remain a mining one. Nodes are motivated not to be malicious by the threat of losing their mining status and their rewards. Moreover users vote for delegates, who in turn participate in the governance of the blockchain. Delegates will propose improvements and changes, which will be voted by the other users. With the coin age system, staked cryptocurrency has an age property. After a certain amount of time, such as 10 days or even a month, the staked cryptocurrency can count towards the owning user being selected to publish the next block. Then the age of the staked cryptocurrency is then reset, and becomes unavailable for use until the required amount of time has passed. Users with more stake are allowed to publish more blocks, but they cannot dominate the system, and that's because of a cooldown timer attached to every cryptocurrency coin counted towards creating blocks. Larger group or older coins will increase the probability of being chosen to publish the next block. Usually there is a built-in maximum probability of winning so the stake holders are prevented from hoarding aged cryptocurrencies. With the multi-round voting system, often referred to as Byzantine fault tolerance proof of stake, added complexity is introduced. Several users are selected to create proposed blocks. Then all staked users will cast a vote for a proposed block. After several rounds of voting the new block is decided. This method allows everyone to have a voice in the block selection process. When the choice of block publisher is a random choice, also referred to as chain-based proof of stake, the blockchain will look at all users with stake and choose amongst them based on the ration of stake to the total amount of cryptocurrency staked. So, for example, if a user had 33% of the blockchain stake they would be chosen 33% of the time. There is a known problem when using the proof of stake algorithms called "nothing at stake". If multiple competing blockchains were to exist at some point, due to a temporary ledger conflict, a staked user could act on every competing chain. He may as such for increasing his odds of earning rewards. This can cause the growth of multiple blockchain branches without adjusting into a singular branch. Under proof of stake system, the "rich" users can earn more digital assets due to the fact that they can stake more, however to obtain the majority of digital assets within a system to "control" it is cost restrained.

Other consensus models, less frequently used in the blockchain are the Round Robin Consensus Model, the Proof of Authority Consensus Model and the Proof of Elapsed Time Consensus Model.

3.4 Blockchain Conflicts and Resolutions

There are times when multiple blocks are published at approximately the same time. This can cause differing versions of a blockchain to exist at the same moment, and these must be resolved quickly so the blockchain can achieve consistency. On many distributed networks, it is possible for some systems to remain behind on information or have alternative information. This happens because of the network latency between the nodes. Permissionless blockchains are more likely to have conflicts due to their number or competing nodes trying to publish blocks. Using consensus, the matter of conflicting data can be efficiently resolved. For example:

- node_a creates block_n(a) with transactions #1,2,3. node_a distributes it to some nodes.

- node_b creates block_n(b) with transactions #1,2,4. node_b distributes it to some nodes.

- There is a conflict

    - block_n will not be the same across the network.

        - block_n(a) contains transaction #3, but not #4
        - block_n(b) contains transaction #4, but not #3

These conflicts generate two different versions of the Blockchain, like shown in the Figure below. However, none of these versions are "wrong", they were created with the transaction information each node had at that moment. The competing blocks will likely contain different transactions, so one will see different transactions in block_n(a) not being present in block_n(b). If the blockchain deals with some sort of cryptocurrency, then a situation where some cryptocurrency may be both spend and unspent may occur.

Conflicts must be quickly resolved. Most blockchains will wait until another block is mined and use that chain as the "official" blockchain, meaning that the "longer" chain is the one that wins. As shown in the Figure below the blockchain containing block_n(b) becomes the "official" chain, as it got the next valid block.

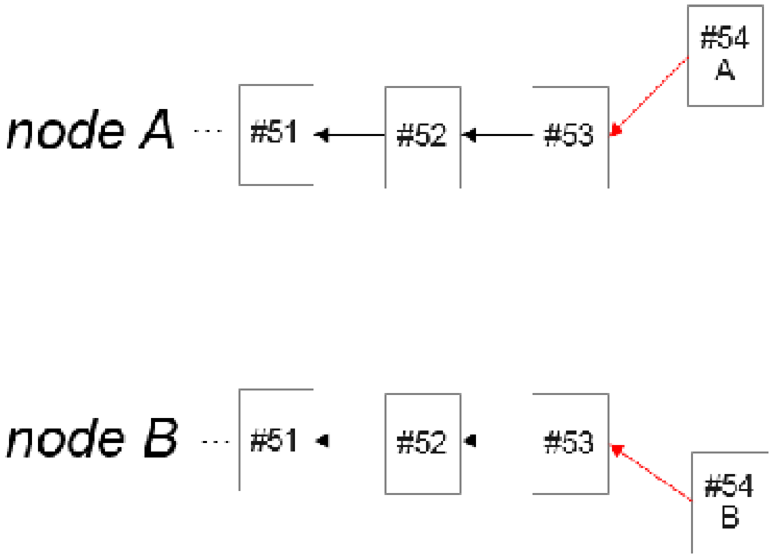


Figure 3.4: Consensus finality violation resulting in a fork

Any transaction present in block_n(a), is returned to the pending transaction pool, where all transactions which have not yet been included within a block reside. This set of pending transactions is maintained locally at each node as there is no central server.

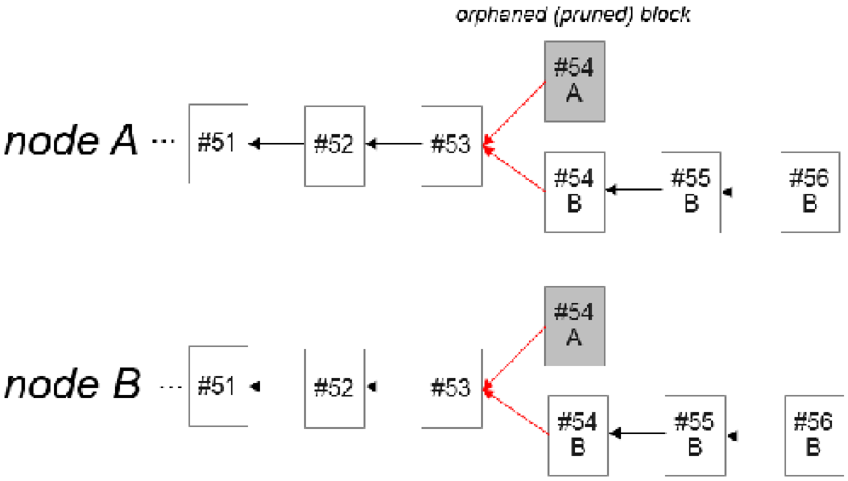


Figure 3.5: The chain with block_n(B) adds the next block, the chain with block_n(A) is now orphaned, and longest chain wins

Because there is a possibility of blocks being overwritten, a transaction is not usually accepted as confirmed until several additional blocks have been created on top of the block containing this transaction. Since blocks can be discarded accepting a block is often probabilistic rather than deterministic, meaning that if more blocks have been built on top of a published block, the more likely for the initial block is to not be overwritten. Some blockchain implementations lock specific older block within the blockchain by creating checkpoints to ensure that a node in a proof of work blockchain gain enormous amount of computing power and create a longer chain that the current, thereby wiping out the entire blockchain.

3.5 Forking

Updating or performing changes the blockchain can be difficult most of the times. Especially for permissionless blockchains is extremely difficult because they are distributed, governed by the consensus of users and they include many users. These changes to the blockchain's data structures and protocol are called forks, and they are divided into two categories: *soft forks* and *hard forks*. For a soft fork the changes are backwards compatible with nodes that are not updated yet. For a hard fork, the changes are not backwards compatible because nodes that have not been updated yet will reject the blocks following the changes. This can potentially lead to the creation of multiple versions of the same blockchain. The term *fork* is also used by some blockchains to describe ledger conflicts, such as two or more blocks having the same block number.

3.5.1 Soft Forks

A soft fork is a change to the blockchain, that as stated before, it is backwards compatible. Updated nodes can continue to transact with non-updated nodes. If no nodes have been upgraded or a very few of them have done it, the updated rules will not be applied. An example of a soft fork occurred on Bitcoin when a new rule was added to support escrow, meaning funds placed into a third party to be disseminated based on conditions such as via multi-signature transactions, and time locked refunds. A proposal was made to repurpose an operation code that performed no operation (OP_NOP2) to CHECKLOCKTIMEVERIFY, which allows a transaction output to be made spendable at a point in the future [14]. Nodes that implemented this change, the software will perform this new operation, but nodes that do not support this change, the transaction is still valid and the execution will continue as if a NOP operation had been executed.

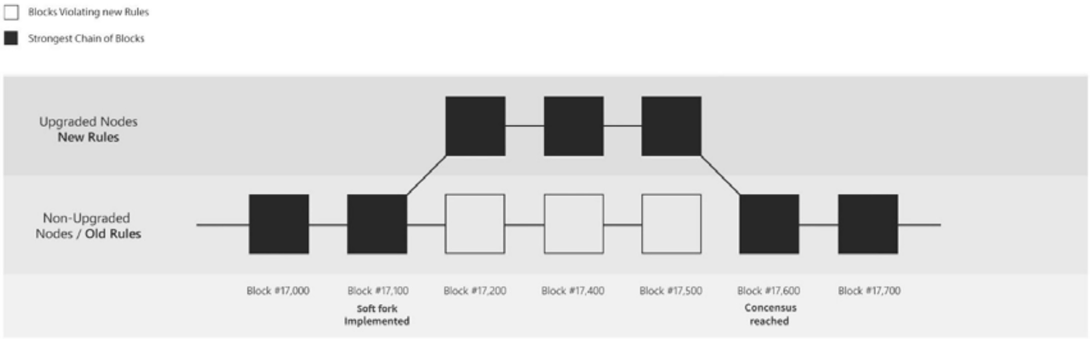


Figure 3.6: Old nodes, having the white blocks, will accept the black blocks, making a soft fork backwards compatible. The white chain will eventually “die out” as no white blocks are mined any longer when all nodes and mines are upgraded.

An example of a soft fork might be that if a blockchain decided to reduce the size of blocks, for example from 2.0 MB to 1.0 MB, updated nodes would adjust the new size of the blocks and continue to transact as normal. Non-updated nodes would still see these blocks as valid – since the change made is not against the rules, meaning that the new block size is lower than their maximum allowed. However if a non-updated node were to create a block with size greater than 1.0 MB, the updated nodes would reject it.

3.5.2 Hard Forks

A hard fork is a change to the blockchain that is not backwards compatible. At some point in time (usually at a specific block number), all nodes will need to switch to using the updated protocol. All nodes need to upgrade to the new protocol so newly formatted blocks are not rejected. Nodes that have not been updated cannot continue to transact because they are programmed to reject blocks that do not follow their version of the block specification. Nodes that do not update will continue to publish blocks using the old format. User nodes that have not updated will reject the new blocks and only accept blocks that match the old format. This leads to the creation of two versions of the same blockchain to exist simultaneously. There is no possible way for two users to interact with each other if they use different hard fork versions. The most known example of hard forking is from Ethereum. In 2016, a smart contract was constructed on Ethereum called the Decentralized Autonomous Organization (DAO). Because of a flow in the smart contract creation, an attacker extracted Ether, resulting in the theft of $50 million. A hard fork proposal was voted on by Ether holders, and the majority of the users agreed to hard fork and create a new version of the blockchain, without the flaw, and that also returned the stolen funds.

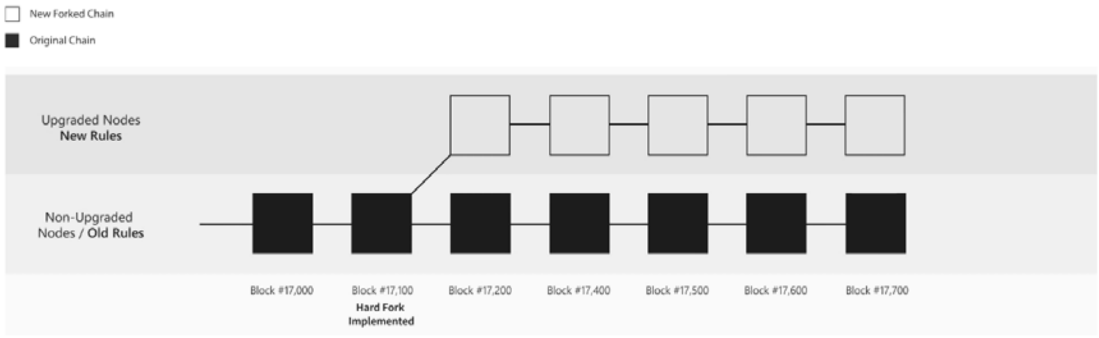


Figure 3.7: The black chain will not accept white blocks, and vice versa, making a hard fork not backwards compatible. Unlike a soft fork, the "old chain", here the chain of black blocks, does not "die out", because it still has enough miners and nodes using it. The chain splits into two separate chains, that share the same transaction history as before the split.

With cryptocurrencies, if there is a hard fork and the blockchain splits then users will have independent currency on both forks, resulting in double the number of the amount of cryptocurrency owned by them. If all activity moves to the new chain, the old one may become useless since the two chains are not compatible. When the Ethereum hard fork took place, the clear users moved to the new fork, and the old fork was renamed and continued its operation.

3.5.3 Cryptographic Changes and Forks

If flaws are found in the cryptographic technologies within the blockchain, creating a hard fork is usually the only possible solution, depending of course on the importance of the error. If, for example, there is an error in the underlying algorithms, a fork would require all future clients to use stronger algorithms. Switching to a new hashing algorithm could pose an important problem because all existing mining hardware would be invalidated. If, for example, a flaw in the SHA-256 algorithm was discovered, blockchains that uses the SHA-256 algorithm would need a hard fork to migrate to a new hashing algorithm. The block that switched over to the new hashing scheme would lock all previous blocks into SHA-256 for verification, and the new hashing algorithm would the one to use from now on. For example, as stated also before the Bitcoin uses the SHA-256 algorithm, while Ethereum uses the Keccak-256 algorithm. However a change of a cryptographic function in a blockchain seems unnecessary right now, and only if sometime in the future quantum computers are developed there will be the need for replacing them.

3.6 Smart Contracts

Nick Szabo was the first to describe the term smart contract back in 1994. He described them as “a computerized transaction protocol that executes the terms of a contract. The general objectives of smart contract design 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.” [6]. Smart contracts extend and take advantage of the blockchain technology. A smart contract is a collection of code and data, also referred as functions or state, that is deployed using cryptographically signed transactions on the blockchain. Such examples are the well-known Ethereum’s smart contracts or Hyperledger Fabric’s chaincode. Nodes are responsible for the execution of smart contracts within the blockchain, and they must obtain the same results from this execution, which in turn are recorded on the blockchain. Essentially, smart contracts are blocks of code that can self-execute certain functions, with certain parameters, when predefined criteria are met. They are activated by someone sending a transaction to their address on the blockchain. Blockchain users are able to create transactions which send data to public functions offered by a smart contract. The smart contract executes the appropriate method using the data provided by the user to perform a service. The code exists on the blockchain and it is tamper evident and resistant and among other purposes can also be used as a trusted third party. A smart contract can store information, perform calculations, and if appropriate, automatically send funds to other accounts. It is not necessary for a smart to perform only financial actions. Also not every blockchain is able to run smart contracts. Smart contracts must be deterministic and are meant to be Turing complete. That means that given a specific input they will always produce the same output, and ideally should be capable of computing everything that can be computed as long as the code has access to unlimited resources and there's no shortage of time. Additionally, after the execution, all nodes executing the smart contract must agree on the new state. To achieve this, smart contracts cannot operate on data outside of what is directly passed into it, everything a user wish the smart contract to obtain must pass it as a parameter to it. For smart contracts that uses data from outside the context of its own system is said to use an "Oracle". Typically in the context of a business process, a smart contract code can represent a multi-party transaction. In such a scenario, the benefit is that this can provide validated data and transparency that can create trust, reduce the time needed for a transaction to complete and reduce costs from reconciliation that exists in traditional business to business applications. The mining nodes execute the smart contract code simultaneously when mining new blocks. There are some blockchain implementations where some mining nodes validate the results of the nodes that execute the smart contract code. For permissionless blockchains, such as Ethereum, the user is issuing a transaction to a smart contract will have to pay for the cost of the code execution. This fee in Ethereum is called Gas and it will be discussed in detail later. There is a limit on the amount of execution time a call to a smart contract can consume, and it is based on the complexity of the code. If this limit is exceeded, the execution stops, and the transaction is discarded. Using this mechanism publisher (miners) are rewarded for executing the smart contract, but also malicious users are prevented from deploying and then accessing smart contracts that will perform Denial of Service (DDos) attacks on the mining nodes by consuming all resources. On the other hand for pemissioned blockchain, such as Hyperledger Fabric's chaincode, there may not a requirement for users to pay for the execution of the code because blockchains like theses are designed around having known participants.

3.6.1 Smart contract Applications

ICOs

Currently the most popular use case for smart contracts is Initial Coin Offerings (ICO). They are means by which startups raise funds for their ventures, avoiding conventional investing regulations and nowadays are produced at an insane rate, roughly one new ICO a week, on the Ethereum blockchain. That's because Ethereum itself was initially an ICO back in 2013, and its founders had acquired nearly $25 million prior to launching a public facing blockchain in 2015. The blockchain evolved from a platform that just runs cryptocurrencies on. It enabled smart contracts which had the following properties:

- Multisignature functionality meaning two or more parties could approve or reject a transaction execution independently
- Automatic execution based on logic programmed into a contract

Mortgages

This process is far from simple. For example, the terms of a mortgage agreement are based on an assessment of the mortgagee's income, outgoings and other circumstances. The process can be made a lot more complicated for both the lender and the mortgagee, by the need to carry out these checks, which third parties take care of. But if the middle men are cut out, the parties could deal directly with each other.

Supply chain management

Supply chain management involves the flow of good from raw material to finished products. Smart contracts can record ownership rights as items move through the supply chain, confirming who is responsible for the product at any given time. Internet of Things sensors made this process a lot easier, by tracking goods from producers to warehouses, from warehouses to manufacturers, and from manufactures to suppliers. The finished product can be verified at each stage of the delivery process until it reaches a customer. Smart contracts can be consulted to find an item which is lost or delayed. Because everything along the way is recorded smart contracts restore the trust in trades.

Protecting copyrighted content

Existing systems do not work well every time multiple parties are involved in the rights of a piece of content that is used for commercial purposes. This has lead to confusion over entitlement, giving some contributors more while others receive nothing. Smart contracts can ensure that royalties go to the intended recipients by recording ownership rights in a decentralized blockchain network.

Smart contracts have many benefits for a wide range of industries, reducing unnecessary costs and time investment while enhancing transparency. Theoretically are more efficient and trustworthy than traditional contract law, and also offer better security as every action is recorded and verified.

Chapter 4

Ethereum Platform

This chapter gives an overview about how the Ethereum Blockchain is working, how nodes in the network communicate with each other and the basic protocol for data transfer.

4.1 System Overview

The Ethereum Blockchain is probably the most evolved yet complex Blockchain system to date. However despite the complexity of the protocol and the security mechanisms that have been designed so far a full Ethereum node is composed of three essential parts: *a)* the Blockchain component, *b)* the peer-to-peer (p2p) network, *c)* the virtual machine

4.1.1 Blockchain component

The Blockchain component of the Ethereum network has been thoroughly explained in the Chapter 3, but to sum up the Blockchain is nothing more than a series of blocks that are chained to each other in specific order. By breaking any one block *b* will also break all its successors *b+1*, *b+2*,... *b+n*. Each block stores together a number of transactions with the hash of the previous block and the *proof of work* of the current block. The proof of work as already explained is just the result of an intensive computation that aims to find the first number, that is called nonce, that, together with the content of the block returns a certain hash. This hash usually starts with a number of 0s that increases according to a parameter called *difficulty*. The higher the difficulty the longer it takes to find a hash. Each block is connected to the previous block with its hash, meaning that the previous block's hash is a part of the content of the current block hash, making the chain hard to break. Breaking a chain means forging a special block at any position in the chain, but with no change to all the following blocks. The network rejects such a change in the chain because it cannot be a legitimate one, which brings the second component: the *peer-to-peer* network.

4.1.2 Peer-to-Peer network component

The Ethereum network is formed by nodes connecting to each other. A full node broadcasts transactions and blocks to the network and receives other transactions and blocks from it. It is also responsible for the synchronization of the current state of the Blockchain with the rest of the network. Except for other experimental or private networks, the official Ethereum networks that are usually used are *Mainnet* and *Testnet*. The Testnet is used to perform experiments of new protocols or features that usually get adopted by the Mainnet when they are ready. The Ethereum Blockchain is way more complex than the Bitcoin due to the fact that on Ethereum it is possible to execute arbitrary code in the form of smart contract. A smart contract in Ethereum is a type of account that contains code, compiled from a high level language like *Solidity*, a contract-oriented programming language and is is designed to target the *Ethereum Virtual Machine* (*EVM*). As for any other compiled language, the smart contract code is hosted in the form of bytecode (or object code) on the Blockchain. Types of data that can be stored on the Ethereum Blockchain are wallets, accounts, transactions and bytecode. Hosting code would be useless if it is not executed. The most well known mechanism to execute bytecode compiled from a high level language is called VM or Virtual Machine.

4.1.3 EVM component

Like many programming languages such as JavaScript, Ethereum has its own virtual machine, a stack based machine to push and pop instructions as in a regular computer with Intel, ARM or AMD CPUs. It is a simple but powerful, Turing complete 256bit Virtual Machine that allows anyone to execute arbitrary EVM Byte Code. The EVM is part of the Ethereum Protocol and plays a crucial role in the consensus engine of the Ethereum system. Anyone is allowed to execute arbitrary code in a trustless environment in which the outcome of an execution can be guaranteed and is fully deterministic. Having said that, the purpose of the virtual machine is to execute smart contract code. This mechanism allows transitions from one state to another, just like a real machine. Thus given a certain block, which stores a number of transactions and given a state s, performing the computation will bring the machine into a new state s'. The state transition mechanism consists of accessing transaction-related accounts, computing operations, and updating or writing the state of the virtual machine. Everything that is executed on the virtual machine will alter its state. After executing all the transactions of a block, the current state will be stored into what will become the next block.

4.2 Merkle Tree Structure

In the process of block verification an important component plays a fundamental role to increase the performance and scalability, the Merkle trees and the Merkle proofs. A Merkle tree is a data structure, a binary tree, in which each node contains the hash of the concatenation of its direct children. Using this structure it is possible to hash large data while still ensuring verification of a branch of the tree independently from the rest. This way if a data chunk has changed, the path, meaning all the hashes of the parent chunks, to the root will change, while a very large part of the tree will stay unchanged. The root of the tree will be changed after modifying any leaf, so checking the root is sufficient to detect such changes. Merkle trees, in Ethereum, are used to save space in each block and allow the light clients to detect and verify changes efficiently. In any other way, the nodes would have to store and verify large data making it extremely inefficient. Storing only the root of the Merkle tree that represents the hash of all the transactions in that block is sufficient for verification. It is not necessary to store all the transactions in the block. In each Ethereum's block, three Merkle roots are actually stored, not just one as in a Bitcoin's block.

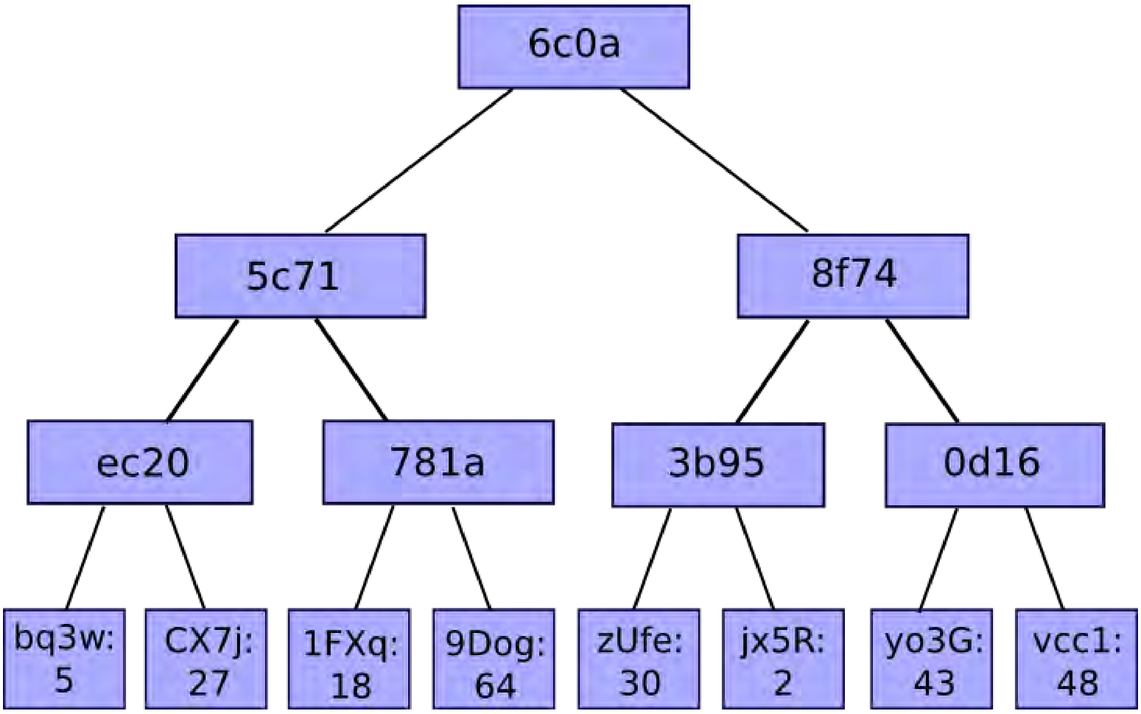


Figure 4.1: A simple Merkle tree

The three Merkle roots in each block header are:

- a Merkle root of the Merkle tree of all transactions
- a Merkle root for receipts (pieces of data that show the effect of executing each transaction)
- a Merkle root for the state of the virtual machine (EVM state)

This way the performance and scalability are increased, because transactions, receipts and states do not need to be stored in each block, only full nodes would do that. For the proof to be computed the node creates a fake block on the local filesystem, sets the state s from the current block, and applies a transaction. The same node also pretends to be a light client, using only the Merkle roots in the block. All queries to fulfill requirements from the transaction are sent to the server. The server responds by sending the requested data as a proof. Then the client executes the same and checks if the local result matches what the server sent as a proof, finally accepting or rejecting it.

4.3 Scalability

One major issue of every Blockchain is scalability. Every full-node executes each transaction and stores the entire state for security reasons and to maintain high degree of decentralisation. However there is an exponential increase in the number of transactions generated by the system. This would be the case as more and more DApps (distributed applications) are created and executed on the same Blockchain. Several forms of parallelization are being investigated trying to solve the scalability issue in Ethereum. As a matter of fact, to be comparable to traditional centralised systems, a Blockchain should be scalable, secure and decentralised. However as of today two of such properties are possible. Currently four different ways are being investigated to solve the scalability issue: *a)* plasma chains, *b)* sharding, *c)* interactive verification for scalable computation and *d)* state channels However only plasma chains and sharding seems to be the most promising ones. Scalability is affected by the Blockchain's architecture. In all blockchain protocols each node stores all states and all transactions. State in Blockchain as already mentioned is account balances, contract code and storage. Every node processes all transactions, by executing them on their local virtual machine and saving receipts and next state back to the Blockchain. While this ensures that every node can check everything, this reduces scalability. Such a node would require a larger and larger filesystem to store blocks and more and more computing resources to verify all transactions. Ethereum can process around 12-30 transactions per second, mainly because it is limited by the computational power of one node. Such a system is definitely very secure but not scalable at all. The key to break such a hard limit is to process many transactions in parallel.

4.3.1 Plasma Chains

Plasma is a proposal that introduces the concept of *sidechain* or child Blockchain. It uses a series of smart contracts to create hierarchical trees of sidechains. Plasma can be compared to having a blockchain into a blockchain relaying information back to the main chain whenever required. In the main chain a smart contract creates the sidechain. Therefore the rules of the main Blockchain also apply to the sidechain. However the sidechain is maintained and controlled by independent nodes, thus making it easier for the nodes on the main chain to process other transactions. Such an approach seems to work very well for micropayments among parties that report their balances to the main chain only after the series of payments have been accepted and validated. From a scalability perspective, Plasma does not impose any constraint about the number of sidechains that can be created, leading to theoretically infinite scaling. The major benefits of plasma are faster and way cheaper transactions off-chain.

4.3.2 Sharding

Sharding is a concept that's widely used in databases, to make them more efficient. A shard is a horizontal portion of a database, with each shard stored in a separate server instance. This spreads the load and makes the database more efficient. In case of the Blockchain, each node will have only a part of the data on the Blockchain, and not the entire information, when sharding is implemented. Nodes that maintain a shard maintain information only on that shard in a shared manner, so within a shard, the decentralization is still maintained. However, each node doesn't load the information on the entire blockchain, thus helping in scalability. Thus the number of transactions in such a blockchain would be the sum of the transactions of every single asset. If each shard is dedicated to each asset, shard-specific nodes can process shard-specific transactions in parallel, still maintaining strong security guarantees.

In order to perform the aforementioned capability, special nodes called **collators** accept transactions on shard k and create **collations**. A collation has a collation

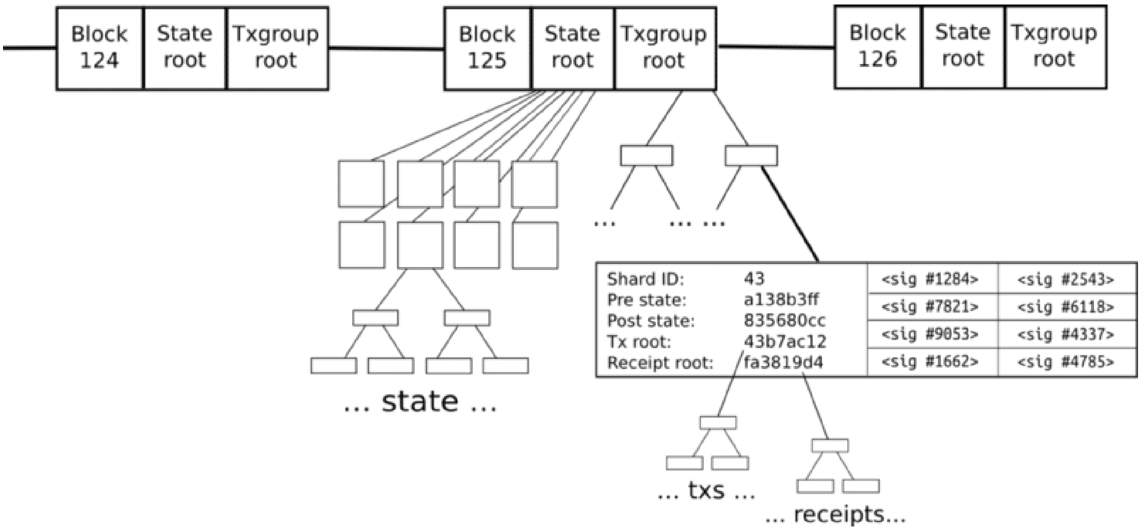


Figure 4.2: Sharding in Blockchain

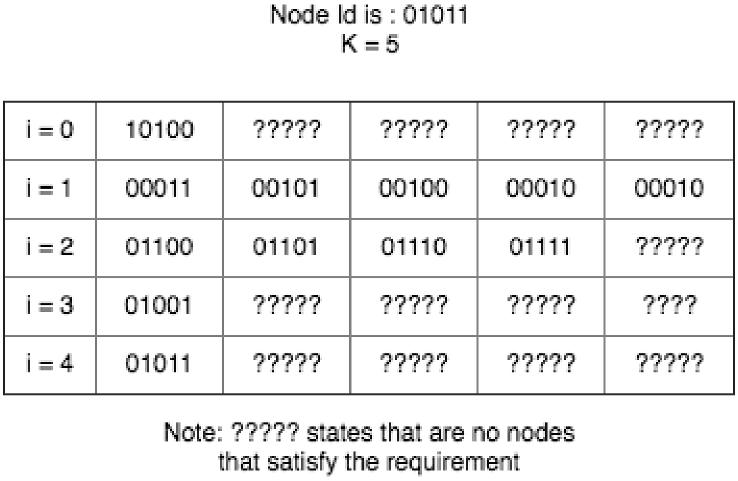
header, that will be stored in a block on the main Blockchain. With such information it is possible to verify that all the transactions of a collation are valid, after they have been processed by nodes other than those on the main Blockchain. This, in turn, increases the throughput of the entire Blockchain and the number of transactions per second.

4.4 Protocol Basics

For a peer-to-peer network to be complete there must be an implementation of node discovery, which allows peers to discover more nodes that support and run an Ethereum client. Ethereum uses a Kademlia-like protocol for node discovery. *Kademlia* is a Distributed Hash Table for peer-to-peer networks that can create a distributed server that all peers can store and read data in a decentralized way.

4.4.1 Specification

Every node has a specific ID that is unique. This said Ethereum uses the SHA3 hash of our public key, node-IDs also consist of global IP and port so a connection can be established. Each node stores a table of known active nodes each has K nodes in it. The node's main goal is to maintain exactly K known nodes in each row. Row i represents all the known nodes that have the same first i bits in their address as the current node (row numbering starts at 0). Each node can request from another node to lookup a node. A lookup request consists of the requested node ID and the source node ID.



Node Id is : 01011
K = 5

i = 0	10100	?????	?????	?????	?????
i = 1	00011	00101	00100	00010	00010
i = 2	01100	01101	01110	01111	?????
i = 3	01001	?????	?????	?????	????
i = 4	01011	?????	?????	?????	?????

Note: ????? states that are no nodes
that satisfy the requirement

This protocol allows Ethereum nodes to discover nodes and connect to them efficiently on order to maintain the network's decentralized factor. When a new node tries to join the network, it needs to establish a list of some known peers that it can communicate with. Ethereum clients are hard-coded with three main peers that are maintained by the Ethereum Foundation and so upon joining the network for the first time, the new peer will ask one of these bootstrap peers for a list of active nodes. These three bootstrap peers are a form of centralization.

4.4.2 Data Transfer

Ethereum uses a protocol named Recursive-Length Prefix, *RLP*, for data transfer. This protocol enables nodes to transfer encrypted, serialized data. When two nodes want to communicate, they send each other some cryptographic data, such as the public keys, to make sure all of the subsequent data transfer is encrypted and cryptographically signed. Then, both nodes send to each other the protocols and the versions of these protocols they support. The three Ethereum protocols are: *a) eth* for the Ethereum protocol, *b) shh* for Ethereum's Whisper protocol and *c) les* for

Light Ethereum Node Subprotocol After all messages are encrypted and a protocol agreed upon, the subsequent messages are dependant on the protocol chosen.

4.5 Transactions

In this thesis the indexer maps addresses to transactions, and in this section a closer look at the Ethereum's transactions structure is provided. As the yellow paper [7] mentions, Ethereum is a transaction-based "state" machine, a technology on which all transaction based state machine concepts may be built.

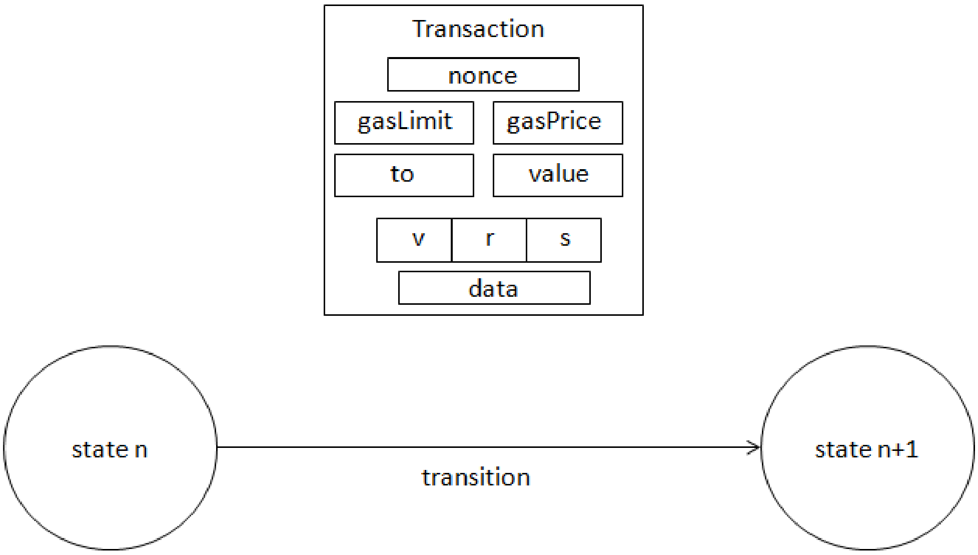


Figure 4.3: State transition in Ethereum

The Ethereum blockchain begins at its own genesis block. From the genesis state, at block 0, onward, actions such as transactions, contracts and mining will continually change the state of Ethereum. Data such as account balances are not stored directly in the block headers of the Ethereum Blockchain. Only the root node hashes of the transaction trie, state trie and receipts trie are stored directly in the block headers as shown in figure 4.4

A transaction that has been fully confirmed it is recorded in the transaction trie and it is never altered thus making it parmanent data in the Blockchain. Ethereum

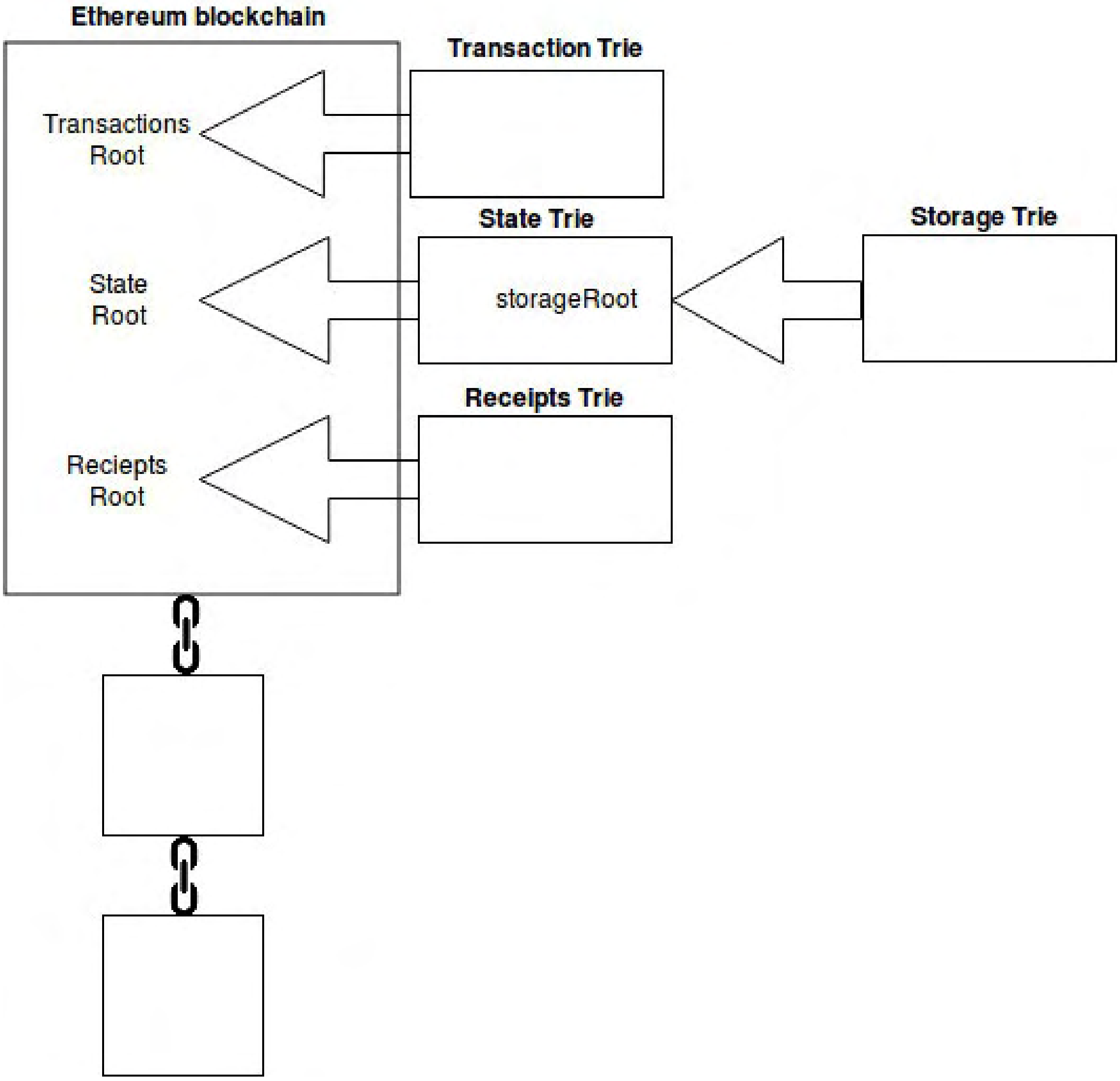


Figure 4.4: Ethereum Tries

uses trie data structures to manage data, all explained below.

State trie

There is one, and one only, global state trie in Ethereum, and it is constantly updated. The state trie contains a *key* and *value* pair for every account which exists on the Ethereum network. This *key* is a single 160 bit identifier and is essentially the address of an Ethereum account. The *value* in the global state trie is created by encoding the following account details of an Ethereum account by using the RLP encoding method:

- **nonce**: the count of the number of outgoing transactions, starting with 0
- **balance**: the amount of ether in the account
- **storageRoot**: the hash associated with the storage of the account
- **codeHash**: the hash of the code governing the account, if this is empty then the account is a normal account that can be accessed with its private key else it is a smart contract whose interactions are determined by its code

The state trie's root node (a hash of the entire state trie at a given point in time) is used as a secure and unique identifier for the state trie; the state trie's root node is cryptographically dependent on all internal state trie data.

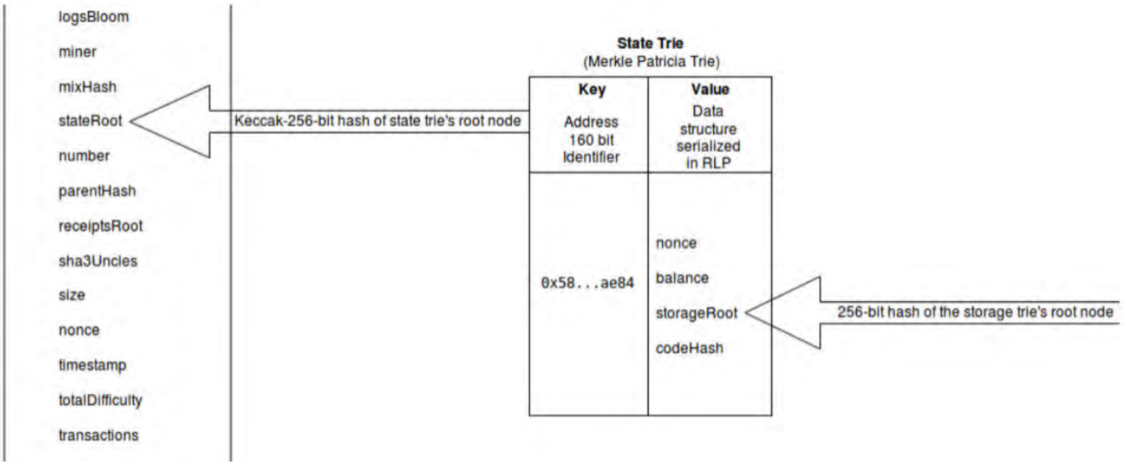


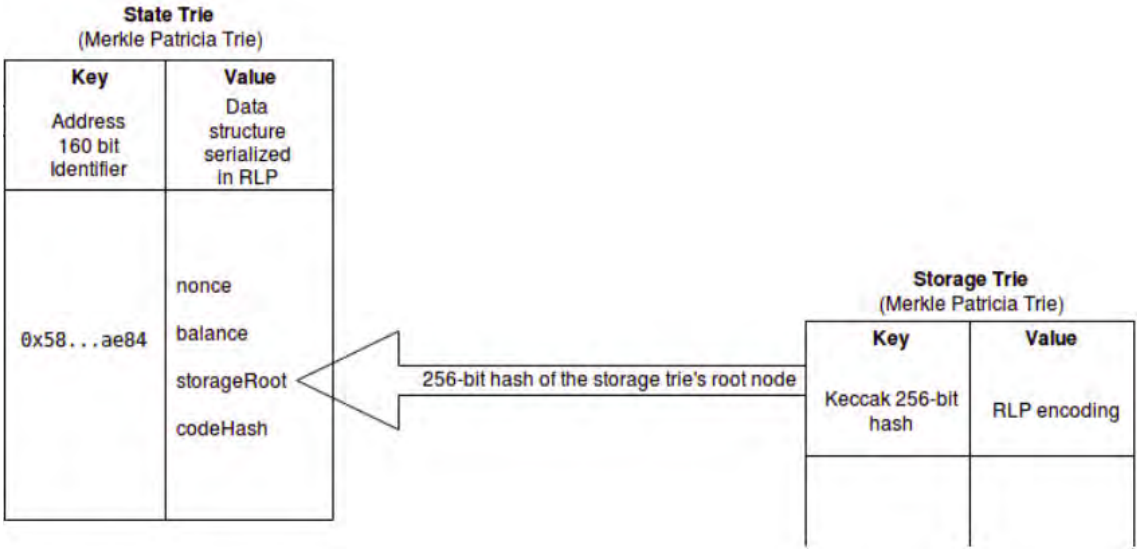
Figure 4.5: Relationship between the State Trie and an Ethereum block

Storage trie

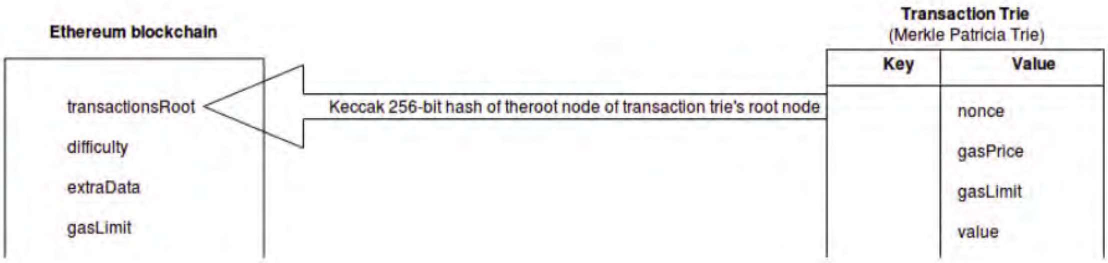
A storage trie is where all of the contract data lives. Each Ethereum account has its own storage trie. A 256-bit hash of the storage trie's root node is stored as the *storageRoot* value in the global state trie.

Transaction trie

Each Ethereum block has its own separate transaction trie and a block contains many transactions. The miner who assembles the block is the one who decides the



order of the transactions in a block. The path to a specific transaction in the transaction trie, is via the index, which is RLP encoded, of where the transaction exists in the block. Mined blocks are never updated and so the position of the transaction in a block is never changed, meaning that once the location of a transaction in a block's transaction trie is located, the same path can be taken to retrieve the same result.



Transactions fields

A transaction in Ethereum is stored as [nonce, gasprice, startgas, to, value, data, v, r, s]. These input fields are explained below:

- **nonce**: the count of the number of outgoing transactions, starting with 0

- **gasPrice**: the price to determine the amount of ether the transaction will cost
- **startgas**: maximum amount of gas allowed for the transaction
- **to**: the account the transaction is sent to, if empty, the transaction will create a contract
- **value**: the amount of ether to send
- **data**: could be an arbitrary message or function call to a contract or code to create a contract
- **v**, **r**, **s**: all three of them make up the ECDSA signature

Chapter 5

Implementation

This chapter discusses the components that have been implemented, so as the Indexer extracts the data from the Ethereum Blockchain, index them using Redis as a transactional database, and then provide a REST API to query all transactions related to a given address. Moreover, a bigger picture from a software engineering perspective is shown, so that the reader understands how the different components and modules are combined. Finally, it explains the steps that are followed in order to implement the data extraction and how they are indexed in the database.

5.1 Combining existing parts

5.1.1 Illustration of the System Architecture

It is of high importance, before going into much details, to show the overall illustration of the whole system architecture. In Figure 5.1, we see that we use the Ethereum network as the Blockchain platform and *Geth* which is an Ethereum client that runs a full Ethereum node implemented in *Go*. Geth offers three interfaces, these being the command line subcommands and options, a JSON-RPC server and an interactive console as shown in Figure 5.2. Put it simply *Geth* is a program which serves as a node for the Ethereum blockchain, and via which a user can mine Ether and create software which runs on the Ethereum Virtual Machine (EVM). Another Ethereum client that can be used is *Parity* [8]. Parity has some advantages over

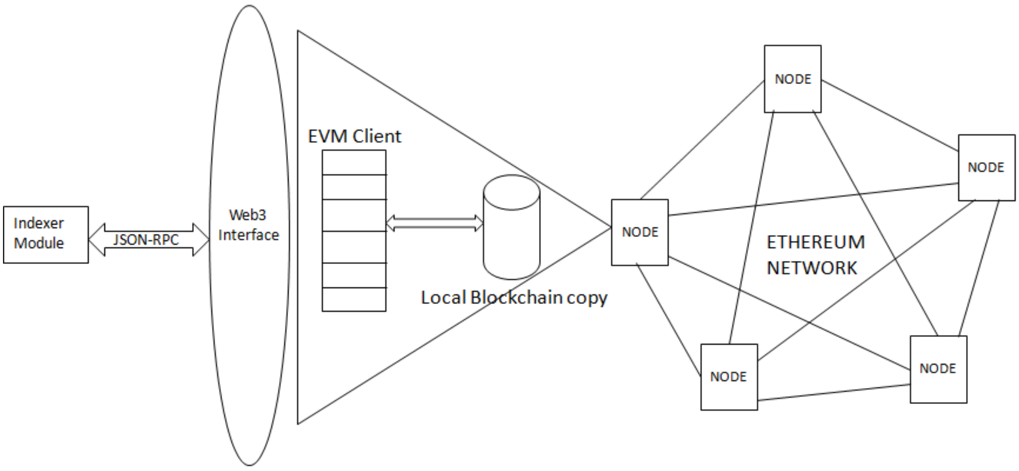


Figure 5.1: Indexer interacting with the Ethereum Blockchain to get data

Geth, mostly because it uses the pruning algorithm and the hard drive usage won't grow exponentially, meaning that the chaindata folder which stores the Blockchain won't add many gigabytes of blocks every week. Moreover Parity has a passive mode to reduce CPU and network load on leaf nodes, and the warp sync allows the user to sync the Blockchain from scratch in hours as opposed to days. In this project Geth is used as a node, and we need to connect to an Ethereum network to download the Blockchain. We connect to the *Rinkeby* testnet to download an synchronize a full node, using the following command

```
geth --rinkeby --datadir=$HOME/.ethereum --cache=2048 --rpc --
    rpcapi=eth,web3,net,personal --syncmode=fast
```

We use a testnet instead of the mainnet, because this way we do not have to use real Ether and there is no way we can cause a conflict to the main chain as described before. Also they are fast and provide more feedback.

After the node is fully synced we can find the *chaindata* folder, where the Blockchain is actually stored in the *./˜ethereum* directory. The file *testnet* is our test Blockchain. We also have RPC and IPC API provided to communicate with our node through console. IPC generally works on your local computer. In the Ethereum space, IPC normally involves geth creating a IPC pipe (which is rep-

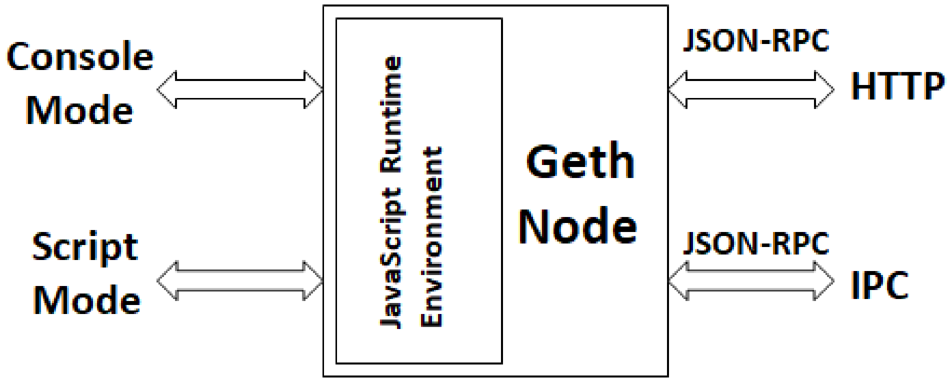


Figure 5.2: How Geth communicates with the user running the node and the network

resented by the file $HOME/.ethereum/geth.ipc) on our computer's local filesystem. Other processes on the same computer can then use the IPC file to create bi-directional communications with geth. In this thesis an IPC communication is disabled. We use the RPC communication protocol. Our endpoint is set to *localhost:8545* (*127.0.0.1:8545*), so only other processes in our system can communicate via this RPC endpoint. For the purpose of communication we use the *web3.js* library, which is provided by Geth and it is a compatible JavaScript API. The web3.js is a collection of libraries which allow you to interact with a local or remote Ethereum node, using a HTTP or IPC connection.

5.2 Redis Storage

In this thesis, we use Redis as our primary storage. Redis is an open source in-memory data structure store, used as a database supports data structures such as strings, hashes, lists and sets. Redis is written in C, thus making it extremely fast, and it is a NoSQL database. Redis provides a command line interface, named **redis-cli**, that enables us to communicate with the Redis Server running in the background. Redis is supported by JavaScript, and it offers many commands to interact with the server. The ones used will be explained in a later section. By default Redis Client uses **127.0.0.1** and port **6379** but it can be easily changed. Redis is a key value database and as shown in Fig 5.3 the values are stored in *Address* to *Transactions* manner. After queried for a specific transaction by its hash, Redis will return an answer of the form ”txs”:[”blockNumber”: number, ”hash”: string] as shown in the figure above. We also used Redis Desktop Manager, which is a tool that offers a GUI to access the Redis DB and also perform some basic operations such as view keys as a tree, CRUD keys, execute commands via shell.

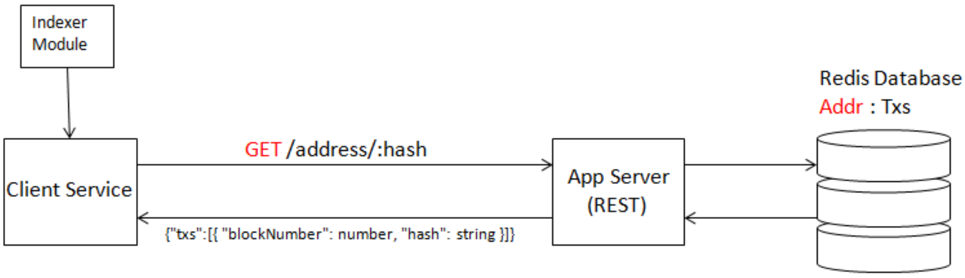


Figure 5.3: Indexer module requesting data from the REST server, who in turn requests the data from the Redis Database

The implementation of the Redis storage used can be found in the ./src/redis_storage folder. We create a client, to perform any action we want after the connection is established, with the command *redis.createClient()* provided by the API for Node.js using the default IP, but we change the default port to *6380* if not specified. The Database uses a set of functions to store and return the data after a query from the REST Server.

5.2.1 Redis Functionality

We use the Bluebird API instead of native Promises, making everything asynchronous. By using the following commands

```
bluebird.promisifyAll(redis.RedisClient.prototype);
bluebird.promisifyAll(redis.Multi.prototype);
```

we add a *Async* to all node_redis functions.

Redis Functions	After Bluebird
multi	-
set	setAsync
get	getAsync
sadd	-
exec	execAsync
smembers	smembersAsync

Table 5.1: Redis Functions used

Starting Block

By defining a *blockNumber* from which we want to start searching, we implemented a function to ignore previous blocks or start from the beginning if there is a wrong index as shown below:

```
const startFromBlock = (blockNumber, txs) => {
  let idx = txs.findIndex( (tx) => tx.blockNumber > blockNumber);
  if(idx === -1) {
    idx = 0;
  }
  return txs.slice(idx);
}
```

Update Address

To asynchronously update an address, we update them as a Redis transaction, meaning that they will update as group and sequentially. This will happen using the *multi* command, the Redis transaction will be serialized, and then executed asynchronously with the exec command as shown:

```
const updateAddr = async (addresses) => {
    const multi = client.multi();
    for(let addr in addresses) {
      let { txs } = addresses[addr];
      txs.forEach(({ hash, blockNumber, inbound }) =>
   multi.sadd(addr, serializeRow(blockNumber, hash, inbound))
     ;
 }
 await multi.execAsync();
}
```

Adding and Returning Values

To get the requested Ethereum transaction given an address we get all transactions related to the specific address, unserialize them and then as described above we use the *startBlock()* function. Adding a value to database is a simple as turning the hexadecimal address to buffer JavaScript object and invoking the *setAsync* Redis method to add it to the set.

5.3 Ethereum Network

Web3 Instantiation

Web3 is the main class of anything related to Ethereum. The property *modules* returns an object with the classes of all major sub modules, and in this thesis we use the **Eth** module, which provides us with the ability to interact with the Ethereum network, and the **Net** module, which is used for interacting with network properties of the Blockchain. We need to give client a way to connect to the Blockchain. Specifically, the *web3.js* library requires a *Provider* object that includes the connection protocol and the address/port of the node we are going to connect to. We create a web3 module to configure the provider setting the host to **127.0.0.1**(the *localhost*). The *web3.js* library supports 3 different providers, but in this thesis we use either the **HTTP** provider or the **WebSocket** one.

Web3 Methods

As previously said, we first used a provider to connect to Ethereum, and we set up the Redis storage where our data will be stored. We have also implemented a module to get the needed data from the Ethereum network. We use the following methods provided by the *web3.js* library:

- web3.eth.getBlock
- web3.eth.getBlockNumber
- web3.eth.isSyncing
- web3.eth.getTransaction
- web3.eth.subcribe

At first we use the **p_limit** package to limit the aforementioned web3 methods queries in the Ethereum network because at times there can be a large amount of requests, thus making sure they are not hitting the network with thousand of concurrent HTTP requests. We bind each method to create a new function of that method and we try to call each web3 method after a 5 second delay if the call has failed. The sample code of binding is featured below:

```
web3methods.forEach((method) => {
    const original = this[method].bind(this);
  this[method] = (options, execOpts) => {
  const wrapped = wrap(original, [options]);
    return execWithLimitedConcur(() => this.tryWeb3Call(wrapped,
       execOpts));
  }
});
```

The **tryWeb3Call(wrapped, execOpts)** that is used above is a function that first tries to call the method that is stated above and if that fails then checks if there is an Ethereum Node running when trying to use that method, and after a delay tries to connect again.

The rest functions of the Ethereum class are utilizing the *web3.js* methods for returning data from the Blockchain such as getting a block containing all transaction as objects using the *web3.eth.getBlock(num, true)* build-in function or getting a transaction by its hash using *web3.eth.getTransaction(prefixHex(hash))*.

Moreover we have implemented a function for getting the block number based to the syncing state of the Blockchain. Based on the API a syncing object looks like

```
{
    startingBlock: 100,
    currentBlock: 312,
    highestBlock: 512,
    knownStates: 234566,
    pulledStates: 123455
}
```

and by examining the **go-ethereum** (**Geth**), which can be found on Github [9], we can understand the meaning of each property of the syncing object, as seen at the table below

Field Name	Type	Description
StartingBlock	uint64	Block number where sync began
CurrentBlock	uint64	Current block number where sync is at
HighestBlock	uint64	Highest alleged block number in the chain
KnownStates	uint64	Total number os state trie entries known about
PulledStates	uint64	Number of state trie entries already downloaded

Table 5.2: SyncProgress gives progress indications when the node is synchronising with the Ethereum network.

Also we use the **web3.eth.isSyncing()** function to check the sync status of the Blockchain. If it is not syncing we just request and get the number of the most recent block, otherwise we get the current block by checking the pulled states with the current block as shown below

```
let syncing = sync.currentBlock || sync.pulledStates
```

The **web3.eth.getTransaction(*hash*)** function is used as stated in the API, using only the hash to retrieve the wanted transaction and after retrieving the transaction we create a function to get the defaults from the transaction we just received. The defaults in a transaction are *a)* the chain ID, *b)* the gas Price, *c)* the nonce, *d)* the value and *e)* the data Every one of them can either be retrieved by extracting them from the web3 object, or by using the build-on function provided by the API, such as **web3.eth.net.getId()** or **web3.eth.getGasPrice()** and if we can a wrong gas value, we utilize the *web3.eth.estimateGas* function to execute the transaction and get the correct gas value.

A function to check for changes in the Blockchain is also implemented which uses the **web3.eth.subscribe()**. We know by the API that we can subscribe for specific events in the Blockchain. There are many types of subscription but we subscribe to *newBlockHeaders*, which as the name states subscribes to incoming block headers and can be used as timer to check for changes on the Blockchain.

```
if(this.canSub) {
    return this.web3.eth.subcribe('newBlockHeaders');
}
```

If the *canSub* variable is true we return a block header.

5.3.1 Blocks

Event Emitter

In this thesis we use an implementation of the native Event Emitter module found in Node.js. This new module is called EventEmitter2 and is extending the interface of EventEmitter with additional features and can simply be installed using the npm package manager. We use its constructor with just two options: *a)* wildcard and *b)* delimiter. Setting the wildcard to true so we are able to use wildcards and the delimiter to ':', so we can segment namespaces, such as 'blocks:number', which will be used.

Lodash Library

In many modules we created we used functional programming to achieve the needed utilities. Lodash [10] is the JavaScript library that provides such utility functions. Here the _.debounce function of this library is used to ensure a given task does not fire too often to destroy the performance.

Block class

Thus we create a class *Block* which extends the above mentioned EventEmitter2 interface. We create a *block cache*, and we use again the **p_limit** library to set the prefetch of the blocks to one at a time, so we can process it and then save it in the cache. There is an event emitted called 'block:number' to get the block, but is filtered through a debounce function waiting 1 millisecond and invoking the event at the leading edge of the timeout.

At the initialization we ensure the correct network connectivity and load the latest Blockchain block as well as our latest saved block. We get these blocks utilizing the functions we created and explained in the previous Ethereum section. Also the subscription is enabled so we get the data from the latest block header as mentioned in the previous section, and we are polling for new blocks in a 6 second interval, before prefetching a new block again. The prefetch function starts at the latest saved block number and ends 20 blocks after as we set this fixed number as the standard block fetch size. First the function checks if the block exists in the cache and if it reaches the end it means that every block is up to date.

```
let prefetchCount = 0;
let index = start;
while(index < end) {
  if(!this.blockCache.has(index)) {
    prefetchCount++;
  }
  index++;
}

if(!prefetchCount) {
  return;
}
```

Then starting from the top again, and for a batch size of 20 blocks it fetches them and updates the index accordingly.

Moreover using the **getBlocks()** function implemented in the previous section, we retrieve a block, and using the **getBlocks()** in the cache module we get all the needed blocks asynchronously to set them into the actual cache. To assist these two functions we implemented a **getNextBlocks()** function, which prepares the next batch of blocks, starting each time from the latest processed block, and adds them into the cache. We get the buffered height by subtracting the latest block number from the Blockchain, from the our latest saved block number.

```
const bufferedHeight = this.latestBlockNum - this.savedBlockNum;
```

Minimum buffered height is also defined as it is equal to the already confirmed height plus one. If the minimum buffered height is lower than the buffered height, we find the target block that is our latest saved block plus the minimum buffered height, and we wait until we get this block. In any other occasion we prepare the next batch of blocks for processing.

5.4 Utilities and Configurations

For the sake of simplicity and reusability we have implemented an utility module that provides many functions useful for processing the transactions of a block. A detailed overview of these functions is described below.

Prefix and Unprefix

The prefix function takes as an argument a hexadecimal address, or a transaction hash and checks if this string starts with '0x'. If that is the case it just returns it. In any other occasion it adds the '0x' to the start of the string.

The unprefix function does the exact opposite of the prefix function, that is cutting the '0x' from the beginning of the hexadecimal string, or returning the string untouched if there is no '0x' in the transaction hash or the address.

This is extremely useful when we are trying to identify if a hash is a transaction hash, and as already stated in previous section it must be equal to 64 bytes without the prefix. In the same manner we are trying to identify an address hash which is equal to 40 bytes without its prefix.

Treating other Hexadecimals

Other functions for treating hexadecimals consists of using the build-in JavaScript functions to convert a hexadecimal number to a string buffer or vice versa, a utility useful for serialization or unserialization of an address hash for example. We have created functions for checking if a given string is a hexadecimal using regular expressions for testing it. Also given a block number we implemented a function to convert that decimal number, to hexadecimal.

Jitter and Retries

In general jitter is the difference in packet delay, meaning that it is the measuring time difference in packet inter-arrival time. We implemented a function to calculate the backoff to assist us in the number of retries in the network.

```
if(jitter <= 0 || jitter >= 1) {
  throw new errors.InvalidInput('Jitter must between 0 and 1 (0 <
     jitter < 1)');
}
let min = 1 - jitter;
let max = 1 + jitter;

let factor = (Math.random() * (max - min)) + min;
backoff = factor * delay;

return backoff;
```

Code Listing 5.1: Code to calculate the backoff for retrying

Using the code snippet from above we created a function to retry when we are trying to get a block, setting always the maximum attempts to infinity, but trying to acquire the block in a 30 second window. When that time passes we are giving up.

Assert Option Type

One last utility function used throughout in the entire project in this thesis is a function called *assertOptionType*. The purpose of this function is to evaluate the type of an object provided and if it does not match the actual type it throws an error stating the exact type of the object provided. Below the code for asserting an option type is demonstrated.

```
const assertOptionType = (object, option, optionType) => {
    const real = typeof object[option];
    if(real !== optionType) {
    throw new errors.InvalidInput(`Expected ${optionType} "${option
       }", but got "${real}"`);
  }
}
```

5.4.1 Serialization

For a transaction to be sent, it must be serialized. Also when we want to add a transaction to our Redis database we must first serialize it and then store it. Checking how a transaction is stored in the Blockchain, we found that it is stored as

```
[nonce, gasprice, startgas, to, value, data, v, r, s]
```

and all these field are stored in the Big Endian format as shown below:

```
fields = [
        ('nonce', big_endian_int),
        ('gasprice', big_endian_int),
        ('startgas', big_endian_int),
        ('to', utils.address),
        ('value', big_endian_int),
        ('data', binary),
        ('v', big_endian_int),
        ('r', big_endian_int),
        ('s', big_endian_int),
    ]
```

Code Listing 5.2: Taken from pyethereum library

The Geth Ethereum Client is storing everything also in the Big Endian format. Having said that we create two function to write a number when we want to serialize and to read it for deserialization. We use the build-in JavaScript function for reading or writing in the Big Endian format, and based on a offset we read or write from the specified position on the buffer. We use the build-in JavaScript functions *a)* writeUInt8(length, offset) *b)* writeUInt16BE(length, offset) *c)* writeUInt32BE(length, offset) for writing the value in the buffer, below there are option for an 8-bit, 16-bit or 32-bit integer. The default is set to 8-bit integers.

The same also applies for the reading procedure when we want to deserialize we use the *a)* readUInt8(offset) *b)* readUInt16BE(offset) *c)* readUInt32BE(offset) functions provided by the JavaScript language.

We use the first *writeInt()* function to create a buffer to store the serialized object, and we take advantage of it in the *redis.js* module when we want to store a serialized row. In the same way the *readInt()* function works for deserialization. To deserialize a row in *redis.js* we just use the buffer created by the *serialize* function, we use the *unserialize* function and we get as a result the **block number**, the **hash** and the **inbound** as single objects, in their original format, meaning from buffer to *int*, as far as block number is concerned, and from buffer to *hex* for the hash and the inbound.

5.4.2 Address Indexer

The transaction_addresses module, is a module that creates an indexer for the address. We are setting an interval of 10 seconds, that is resetting periodically so our average will not be weighed down by the past too much as shown below:

```
setInterval(() => {
    const secPass = (Date.now() - startTime) / 1000;
    logger.log('Blocks per second: ', total / secPass);
    startTime = Date.now();
    total = 0;
}, 10000).unref();
```

While there is no error we are waiting always for the next batch of blocks and when we get it we are adding them to our database and emit an event *'block:number'* for every block stored. After that we count the total number of blocks stored so far.

```
while(!stop) {
    try {
        blocks = await processor.getNextBlocks( { maxBatchSize:
            blksPerBatch, confirmHeight })
    }
    catch(error) {
        eventEmitter.emit('error', error);
        stop = true;
        break;
    }
    await state.addBlocks(blocks.slice())
    blocks.forEach((blk) => {
        eventEmitter.emit('block:number', blk.number);
    });
    total += blocks.length;
}
```

5.4.3 Getting Addresses and Transactions

Moving on we find an address based on its hash and block number, and we use this address and the forementioned **eth.getTransaction** function, to get all transactions related to that address.

```
txs = await bluebird.map(txs, async (tx) => getTransaction(tx.hash,
    timeoutOpts));
```

Moreover based on the inbound of our node, we check and we assign the address to the sender (*from* field) or to the recipient (*to* field).

```
if(tx.inbound) {
    fromAccount = null;
    toAccount = addr;
}
```

```
else {
    fromAccount = addr;
    toAccount = null;
}
```

With the help of the *bluebird* library, we take advantage of its *_.map* function, to create two functions, one for getting the addresses and the other for retrieving the transactions, both of them were needed in the implementation of the REST Server which will be explained in the next section.

5.4.4 REST Server

The REST Server responds to requests about block numbers, addresses, transactions and the health of the server. For this purpose we assigned these routes to a JavaScript object we called *routes*. The properties of the object are displayed below:

```
const routes = {
    blockNumber: '/blockNumber',
    addresses: '/addresses',
    transactions: '/transactions',
    transactionsDefaults: '/transactionsDefaults',
    health: '/health'
}
```

For implementing the server we chose the **Koa** Library over the widely used **Express** Library.

Koa Library

Koa is a web framework for Node.js [11]. It is leveraging async functions in JavaScript, allowing us to discard callback functions, and increases error-handling. The core module of Koa.js does not support routes, however, so we had to also install *koa-router* module to use routes. Koa.js identifies and understands the operations required by HTTP methods and it provides a *request* and *response* object, and it allows to encapsulate these two objects into a single on using Context, which provides additional functionality.

Server Module

We set the timeout options to 10 seconds and we wrapped to it the **getTransactions**, the **getTransactionDefaults**, and the proccessed block number. We used a body parser for the errors and a http proxy using the *koa-better-http-proxy* library for checking if all RPC methods exists else the access is forbidden, and finally the proxy path resolver is set to **'/'**. Our server port is set to run on port 3000 but it can be changed through a config file which will be explained in the next section. Using the object *routes*, which as already stated holds the routes, using the *koa* library the requests are matched to corresponding route, meaning the we can check the health of the server in route **'/health'**, or retrieve the requested block number in the route **'/blockNumber'**. Also the batch size for the addresses and the transaction requests is set to a max 100. Finally there is an error handler on the HTTP status error codes, for not being able to find the route if it does not exists, or for bad requests or timeouts.

5.4.5 Configurations

Configuration Module

In the configuration module, there are functions that parse the *conf.js* file, which holds the critical values for the indexer to function. The module uses the **assertOptionType**, from the utility module, to check the validity of every single field of the *conf.js* file. The fields that the function check their correctness are: *a)* the starting block *b)* the block per batch *c)* the port *d)* the confirmation height *e)* the node *f)* the hostname *g)* the use of a websocket and its port *h)* the HTTP port and *i)* Redis and its port.

After the validity check, the data the indexer needs are parsed from the *conf.js* file. Then a path is resolved from the current working directory, and a file for the database is created at this path, named **transaction_address-index.db**

The Config File

The configuration file holds values crucial for the indexer to operate. The form is as follows

```
module.exports = {
    network: 'mainnet',
    dir: './',
    blocksPerBatch: 100,
    startBlock: 300000,
    confirmHeight: 15,
    port: 3000,
    storage: 'redis',
    redis: {
      port: 6379
    }
    node: {
      host: 'localhost',
      httpPort: 8545,
      wsPort: 8546
    }
}
```

And presented in the Table 5.3 below, every value is explained.

The last field called **node**, with subfields *host*, *httpPort*, *wsPort* is implemented for non-private networks. If a node is up and running the value of the **network** field can be set to ’private’ and there will be no need for the node field.

5.4.6 Error Module

In this thesis, in every module, whenever something went wrong an error value was raised. For this purpose we created an error module to handle every single error that happened throughout the operation of the indexer. The main error classes for this thesis are: *a)* InvalidInput, *b)* NotFound, *c)* Timeout, *d)* Duplicate and *e)* InsufficientFunds. If any of the above errors are caught during execution the indexer throws an error invoking one of the above error classes with the appropriate message. Moreover we set the following key errors for extra safety. *a)* RangeError

Field	Value	Description
network	mainnet	the chosen Ethereum Network
dir	./	The current working directory
blocksPerBatch	100	How many blocks to wait for, before processing, can be any value
startBlock	309562	The block to start indexing, can be any value
confirmHeight	15	Height at which a block is considered confirmed
port	3000	Port to run HTTP server on
storage	'redis'	Our database, if another database is used this field can change
redis	port: 6379	The port which Redis database is running

Table 5.3: Every value of the configuration file.

b) ReferenceError *c)* EvalError *d)* URIError *e)* TypeError *f)* SyntaxError

The Error module, also has functions to check if a node is connected, or ignore an error it not matches to any of those implemented and needed. The function *matches()* is the one that identifies if the thrown error exists in the pool of errors we used and based on that assumption it returns either true or false. This particular function is crucial to the implementation of the REST Server as it is the one that matches the error type produced with the corresponding status code and constructs the returned error value that consists of this status and the type and message of the error as shown:

```
let errorValue = {
    status,
  body: {
    type,
    msg,
  }
}
```

5.4.7 The Logger

To have a complete overview of everything that happens, make sure everything is working as expected and find problems or errors, and for this purposes we have created a logger. The logger is a very simple module that creates a wrapper of five attributes: *a)* the actual log, *b)* the info, *c)* the warnings, *d)* the debug and *e)* the error. They are created five different log files, each one with the corresponding names, meaning that there is a different file for the actual logs, the info, the warnings, the debugging info when needed and the errors. We have also binded the logs to the console, so if used one can see every output in the console instead of a file. For instance, in the *block.js* module an debug log was instantiated, named **BLOCKS**, for diagnosing problems, and giving detailed information and monitoring the Blockchain or when prefetching blocks for the cache. For the storage output, a log file named *storage:redis* is created. As long as the main indexer is concerned a log called *indexer* is created and one for the REST Server is also created, under the name *server*. Finally a log called *web3* is created for the purpose of logging data related to the Node in use and the Blockchain network. A prefix in front of every log file is set so every one begins with **markos_thesis:** followed by the forementioned names.

5.4.8 Connecting to a node

A provider is how **web3** talks to the Blockchain. Providers take JSON-RPC requests and return the response. This is normally done by submitting the request to an HTTP or IPC socket based server. In this thesis two ways to connect to a node are provided:

- an HTTP provider, because most nodes support it and
- a WebSocket provider which works remotely and it is faster than an HTTP.

There also exists the IPC Provider, which uses the local filesystem and it is faster and the most secure but it is not implemented in this thesis.

We use an option variable to get infomation about the hostname, the http port, the websocket port, the logger and a boolean variable indicating if we use the WebSocket provider (useWS).

Based on this we check if the *useWS* is *false*, and if thats the case we connect using the HTTP provider like:

```
web3.setProvider(new Web3.providers.HttpProvider(`http://${host}:${httpPort}`));
```

If the *useWS* is *true*, then we instatiate a variable called *wsOptions* with a timeout of 30 seconds, and an origin header using the hostname, as a JavaScript object.

We also set a *provider* variable using the WebSocket provider, with 10 seconds reconnect time if something goes wrong. If everything runs correctly, and provider connects successfully we log it, and finally set the valid provider using the *web3.setProvider(provider)* function.

5.4.9 State

The module *state.js* creates and handles the state of the Redis storage. First we are getting the *hash*, the *sender* and the *recipient* of a transaction and un-prefixing the hexadecimal using a function we implemented called **getTxFromToHash()**, and then we map this transaction to an address. The core of this module however is to provide a way to store the blocks with every transaction that they hold. First it calculates the valid number of transactions, meaning that every transaction in a block has send an amount of ether and it is not an empty transaction. Then it ensures that every block and every transaction of each block is valid, and associates all transactions that have been executed. If a block does not have any transactions it stores it at the end. Otherwise it ensures the next block has the correct block number stores it and update Redis.

```
const mapTxsToAddr = (transactions) => transactions.reduce((map, tx
  ) => {
    const { blockNumber } = tx;
  const { hash, from, to } = getTxFromToHash(tx);
  if(!map[from]) {
    map[from] = createAddr();
  }
  if(!map[to]) {
    map[to] = createAddr();
  }

  map[from].txs.push({hash, blockNumber, inbound: false});
  map[to].txs.push({hash, blockNumber, inbound: true});

  return map;
}, {});
```

5.4.10 Loading Components

In the *index.js* module everything comes together. When creating the instance to run the Blockchain Indexer the components like the blocks per batch or the starting block, explained in the previous subsection 5.4.5, are set by a config file, and we implemented a module to process these components so we are able to get them individually. We create the default components variable above all, setting the indexer and the server value to *true*. Then it basically uses all of the already created modules, meaning *a)* the blocks module, *b)* the server module, *c)* the eth module, *d)* the api module and *e)* the state module to create and initiate every one of them. We set a JavaScript object variable to hold every value. The *ret_val* as is called, uses the *redis* module, getting from the config file the host name and the port to create our database. It also holds the state of our storage, and the Ethereum API calls. It also instatiates the REST Server using the standard **V1** prefix, creating a logger for the server, and setting the RPC methods for it, and it saves them as its properties. Finally it utilizes the *block.js* module as explained in subsection 5.3.1, to process every incoming block, and the *transaction_addresses.js* module as explained in subsection 5.4.2. The **indexer** then starts and logs every component with its corresponding value.

5.5 Running the Indexer

In order to run the indexer from the command line we use a JavaScript Library called

```
npm start
```

in the console, a message is displayed stating the the process of Blockchain and the REST server is starting. The command can also be chained if we run

```
npm start get <hash>
```

so it will check if this hash corresponds to a certain address or a transaction and it will log it on the console.

Chapter 6

Conclusion

In this thesis we dive into the new technology, the Blockchain, explaining every detail of the way it is structured as well as its functionality of Blockchain's complex nature presenting the overall architecture of the backbone of a system many widely known platforms and cryptocurrencies, like Ethereum and Bitcoin, uses. On the second part we presented a way to index Blockchain data so it will be possible to query and retrieve them from the database stored. The service also provides a REST API to query all transactions related to a given address. In this implementation Geth is used as the Ethereum node, but it is possible to use Infura which is a hosted Ethereum node cluster. The JavaScript Library, web3.js, provided a way to access the Ethereum Network to extract the necessary block data for the indexing purposes. The data then are indexed into Redis, a key-value database, thus becoming available for querying using the REST API. However due to the ordered nature of data in Blockchain, they must be processed in order from genesis to the Blockchain tip for data integrity to be ensured. In this thesis the indexing begins from the Genesis block and then iterate through each transaction in the next blocks, indexing the needed values, and this process is repeated. This works but it scales linearly because as the Blockchain grows the time to index the entire chain will grow at the same pace as the total number of transactions. Nonetheless there is room for future improvements. As this project is implemented it is relatively easy to create another database and use it after changing the config file. Instead of a key-value storage, as Redis is, a NoSQL document orientated database can be used because data is encoded in JSON format which is the format every method from the web.js library returns. Proper choices for future development can be MongoDB and Apache CouchDB. Another approach to the problem of Blockchain indexing is to represent Blockchain concepts using standard OWL ontology or vocabulary. For example EthON is an OWL ontology designed to describe the Ethereum blockchain. So an idea is to create an entirely new solution where the fundamental underlying data model is the collection of linked named RDF graphs, and to build an index for these RDF graphs which can be easily queried using techniques like SPARQL. Finally, as far as performance is concerned, it would be improved by parallelizing the indexing of the Blockchain, a difficult task though given the static and time-ordered nature of the Blockchain. A possible way to deal with this is to load pre-validated Ethereum blocks, thus skipping the work of checking input dependencies, into Hadoop Distributed File System (HDFS) and then process them with a MapReduce cluster. Then the indexing service runs to keep the Blockchain up to date.

Bibliography

[1] Andrew Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Vol. 3. Jan. 2002.

[2] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Jan. 1999. DOI: 10.1007/978-1-4419-8834-8.

[3] Stuart Haber and W Scott Stornetta. "How to Time-Stamp a Digital Document". In: *Journal of Cryptology* 3 (Sept. 1999). DOI: 10.1007/BF00196791.

[4] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: *Cryptography Mailing list at https://metzdowd.com* (Mar. 2009).

[5] *JavaScript web3.js API*. https://github.com/ethereum/wiki/wiki/JavaScript-API.

[6] Nick Szabo. *Smart contracts*. http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html. 1994.

[7] D. Wood. "Ethereum: a Secure Decentralised Generalised Transaction Ledger". In: 2014.

[8] *Parity - The fast, light, and robust EVM and WASM client*. https://github.com/paritytech/parity-ethereum.

[9] *Official Go implementation of the Ethereum protocol*. https://github.com/ethereum/go-ethereum.

[10] *The Lodash API reference. Lodash — A modern JavaScript utility library delivering modularity, performance & extras*. https://lodash.com/docs/4.17.11.

[11] *Koa web framework for Node.js.* `https://github.com/koajs/koa`.

[12] Yannis Manolopoulos, Yannis Theodoridis, and Vassilis Tsotras. "Advanced Database Indexing". In: 17 (Jan. 2000). DOI: `10.1007/978-1-4419-8590-3`.

[13] Allan Third and John Domingue. "Linked Data Indexing of Distributed Ledgers". In: (Apr. 2017), pp. 1431–1436. DOI: `10.1145/3041021.3053895`.

[14] Zibin Zheng et al. "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends". In: June 2017. DOI: `10.1109/BigDataCongress.2017.85`.

[15] Héctor Ugarte. "A more pragmatic Web 3.0: Linked Blockchain Data". In: (Mar. 2017). DOI: `10.13140/RG.2.2.10304.12807/1`.

[16] Shailak Jani. "An Overview of Ethereum & Its Comparison with Bitcoin". In: (Feb. 2018).

[17] Imen Filali et al. "RDF Data Indexing and Retrieval: A survey of Peer-to-Peer based solutions". In: (Nov. 2010).

[18] Wren Chan and Aspen Olmsted. "Ethereum transaction graph analysis". In: Dec. 2017, pp. 498–500. DOI: `10.23919/ICITST.2017.8356459`.

[19] *Ethereum Light Client Protocol.* `https://github.com/ethereum/wiki/wiki/Light-client-protocol`.

[20] Oleksandr Vashchuk and Roman Shuwar. "Pros and cons of consensus algorithm proof of stake. Difference in the network safety in proof of work and proof of stake". In: *Electronics and Information Technologies* 9 (Jan. 2018). DOI: `10.30970/eli.9.106`.

[21] Vitalik Buterin. *Ethereum: A next-generation smart contract and decentralized application platform.* Accessed: 2016-08-22. 2014. URL: `https://github.com/ethereum/wiki/wiki/White-Paper`.

[22] Deepa Kumar and Mustafa Abdul Rahman. "Simplified HDFS architecture with blockchain distribution of metadata". In: *International Journal of Applied Engineering Research* 12 (Jan. 2017), pp. 11374–11382.

[23] Heiner Stuckenschmidt et al. “Index Structures and Algorithms for Querying Distributed RDF Repositories”. In: Jan. 2004, pp. 631–639. DOI: 10.1145/988672.988758.

[24] Sergei Tikhomirov. *Ethereum: State of Knowledge and Research Perspectives.* Feb. 2018, pp. 206–221. ISBN: 978-3-319-75649-3. DOI: 10.1007/978-3-319-75650-9_14.

[25] Tobin Lehman and Michael Carey. “A Study of Index Structures for Main Memory Database Management Systems”. In: Aug. 1986, pp. 294–303.

[26] Dejan Vujicic, Dijana Jagodic, and Randjic Sinisa. “Blockchain technology, bitcoin, and Ethereum: A brief overview”. In: Mar. 2018, pp. 1–6. DOI: 10.1109/INFOTEH.2018.8345547.

[27] Elisa Bertino et al. *Indexing Techniques for Advanced Database Systems.* Jan. 1997. ISBN: 0-7923-9985-4. DOI: 10.1007/978-1-4615-6227-6.