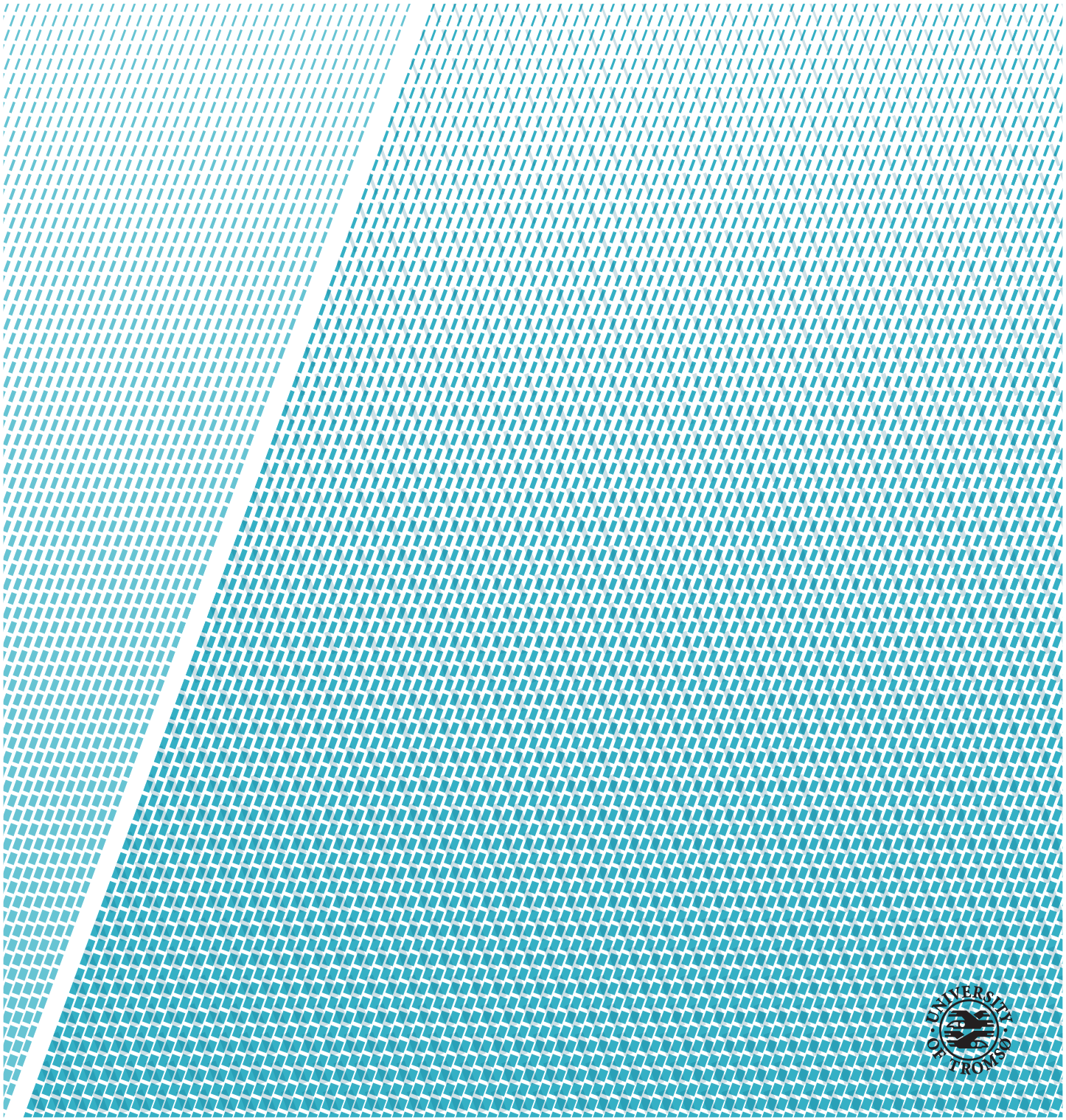
UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Paying for Bandwidth in Blockchain Internet Applications

Enrico Tedeschi

INF-3995 Capstone Project in Computer Science



This thesis document was typeset using the *UiT Thesis LaTeX Template*.
© 2016 – http://github.com/egraff/uit-thesis

Abstract

Understanding the concept behind distributed cryptocurrencies is the first step to improve and take the best out of them. This thesis will give a detailed explanation of cryptocurrency mechanisms such as Bitcoin and Ethereum. It first takes a deep look into centralized and decentralized digital currency, enhancing the focus points of the second one, explaining it and summarizing the most important concepts related to it, such as blocks, Merkle tree, transaction fees, proof of work, and blockchain. Then this thesis provides empirical experiments on the actual Bitcoin blockchain. Plots, results and considerations are showed and described. Furthermore three models are proposed and they make users aware of how they could spend money to get the best advantages from the blockchain bandwidth. These models are generated using statistical regression on the data retrieved and polynomial interpolation to define functions that mathematically describe them. We show that the fee paid to the miner is related to block creation time and also to the time that a transaction needs to be visible in the whole network. This affects average bandwidth available to an application. Plus, using the data collected in 4 months, we define a function that predicts the Bitcoin blockchain growth.

Bitcoin blockchain has been analyzed with a blockchain analytics system, developed using Bitcoin's API. This thesis gives also a measurement about accuracy of data provided from Bitcoin. Graphs about the data retrieved and how this data are manipulated are explained as well. This analysis includes bandwidth and latency for blocks retrieval, the growth of the blockchain, the average time for a transaction to be visible in the ledger of public data and also the relation between the fee paid to mine a block and the amount of time that this block need to be created. Furthermore, a little portion of the blockchain is saved locally in a text file, including the IP of the mining nodes and the IP of all the nodes participating to the ledger.

Contents

List of Figures

List of Tables

My list of definitions

1

Introduction

In 1964, Paul Baran [18] represented a very clear topology describing the differences between a centralized, decentralized and distributed network (Figure 1.1). Since then, the attention in developing systems moved from a centralized scheme to a distributed one, leaving most of the computation to every single user in the network rather than a central coordinator. Such a change might be easy for systems that do not require much of security, where authentication or authorization is minimal. However, the more a system needs to be secure, the more the decentralization process might be tricky as it becomes very important to trust and rely some trusted central coordinator. Systems that more than others need to be secure, are the one related to e-commerce, banking and trades, all systems that have to deal with money.

With the spreading of distributed systems the concept of a decentralized digital currency has become a reality. Until 2008 e-commerce used to rely exclusively on financial institutions serving as trusted third parties. Those are involved in the electronic payments process and they have to guarantee consistency of the transactions and security of data. In 2008, Satoshi Nakatomo has presented Bitcoin [30], the first decentralized digital currency and in 2013 Gravin Wood has presented Ethereum [36]. The second one introduces some new features which were not available in Bitcoin by creating a total new decentralized digital currency.

These decentralized digital currencies are not dependent on any trusted third parties and and they are built over a peer-2-peer network where every compo-

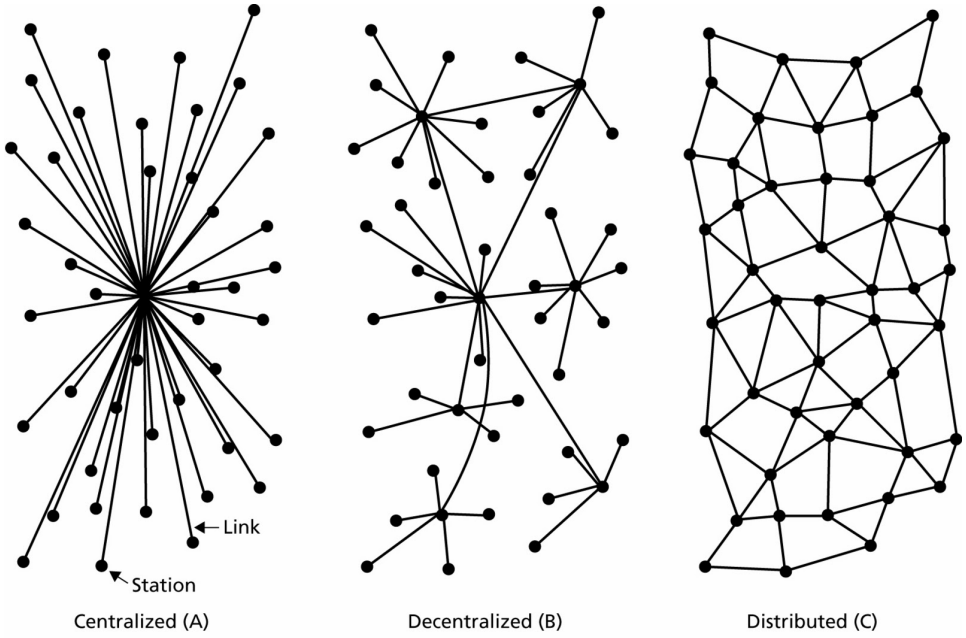


Figure 1.1: Differences between network topologies. Source: On Distributed Communication Networks, Paul Baran, 1964.

nent has the same privileges. These systems allow money exchange without a central authority, which means that the fees are much lower and you can use it in every country. The only fee a user would pay is indeed the one that allows a transaction to be validated in the public ledger of data.

The order of transaction is essential in any cryptocurrency systems. However, establishing correct order can be problematic in decentralized cryptocurrency systems, such as Bitcoin [30] or Ethereum [36], as these systems allow arbitrary nodes to join, including nodes that are malicious. If arbitrary or Byzantine faults are allowed, the system might be left in an inconsistent or invalid state [25]. The ability to mask Byzantine faults has been implemented in various systems such as Byzantium [22], HRDB [35] and MITRA [27]. These protocol guarantees consistency of transactions having f faulty nodes, with a total of N nodes where $N = 2f + 1$ or $N = 3f + 1$. The Fireflies protocol [23] provides secure and scalable membership management and communication substrate in overlay network with Byzantine members.

1.1 Blockchains

The need to tolerate malicious members was the reason for introducing the *blockchain* into cryptocurrency systems. The fundamental principle behind the blockchain is that consensus on transaction ordering is based on contributed computational power rather than number of participants. The blockchain works by appending transactions in blocks. Every block is generated after a relevant computation, and each new block is appended to the public ledger of data (the blockchain), having in that way an ever growing chain of data containing every transaction ever happened. A detailed explanation of the blockchain is viewed in Chapter 2.5.

Besides its use in cryptocurrency, this blockchain technology opens up to several usages in different sectors such as trading, file storage or identity management. Indeed it is already used by NASDAQ in its private socket market. If used in a peer-2-peer file sharing network, the blockchain removes the need of a centralized data base and heavy storage areas. Moreover it allows users to create tamper-proof digital identities for themselves.

1.2 Problem Statement

In order to make efficient use of blockchain technologies in our applications, we first need to understand its performance characteristics, benefits, and limitations. For instance, according to the Ethereum white paper [6], the Bitcoin blockchain grew at a rate of approximately 1 MB per hour from 2008 until 2014. This certainly limits usage of the blockchain.

In blockchain based applications, all information transfer are done in context of a transaction. Before a transaction T written by process A becomes visible to process B, T has to be processed in the blockchain from all the participant of the network, and appended to the ledger of data. This causality relations is needed to, for instance, ensure that the money sent from A will be available to be spent by B. In this context of transactions, we then define the following:

The Definition 1. *write bandwidth* is the time for a transaction to be written from A and untill it is persisted by the blockchain nodes;

and:

The Definition 2. *read bandwidth* is the time for a transaction to be visible in B after it is written in the blockcahin.

The total time taken by a transaction to go from A to B is called total bandwidth, or just *bandwidth*.

A node A that wants to send *tx* money to B, should consider to add to it a *fee*, as showed in Figure 1.2, that will compensate miners after a block is created (mining process explained in Chapter 2.8).

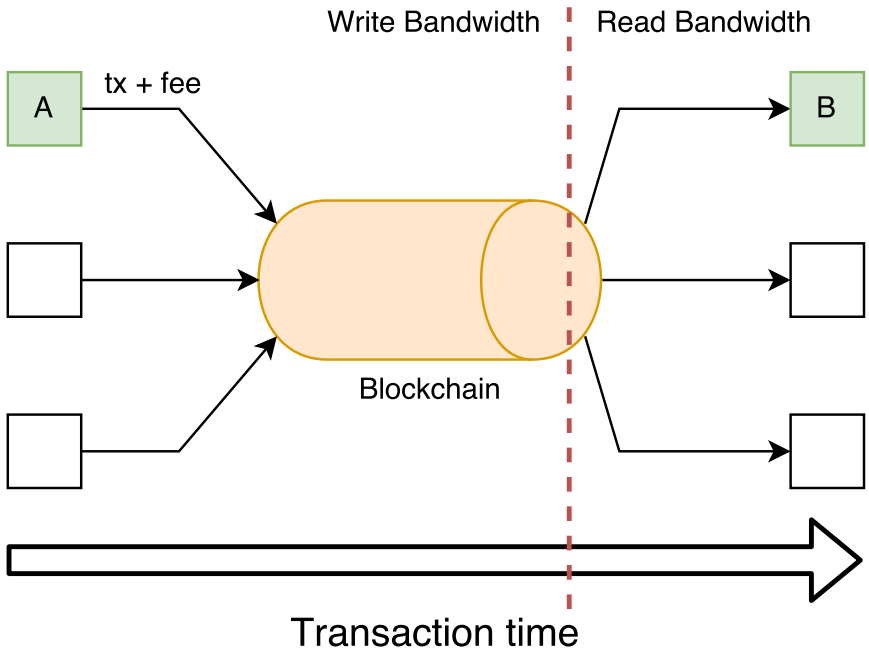


Figure 1.2: Read and write bandwidth during a transaction from A to B through the blockchain.

In this thesis we are particularly interested in studying the fundamental ability of the blockchain to transfer information. Our thesis is that:

applications can control available bandwidth by paying mining fees.

To support our thesis we will build a system that can observe the existing Bitcoin blockchain, and analyze bandwidth correlations. We also define a model that can predict accurately the blockchain growth.

1.3 Method / Context

In this thesis we will analyze part of the blockchain, the most recent one, according to how many nodes the user decide to retrieve, time by time. Because

of the API given it will be hard to retrieve more than 2000 blocks per time, since each block is fetched individually and the read bandwidth is very low.

Our assumption is that we can get sufficient information about the blockchain growth, the block creation time and the time for a transaction to be visible in the public ledger of data, by retrieving and analyzing blocks with a finer granularity than the one represented in the Bitcoin website. Moreover, sampling data from a single node in the blockchain gives statistics representative of the whole system.

For the analysis, data are retrieved block per block and part of the blockchain is saved in a text file. This finer granularity allows us to have a lot of informations that may be hidden in the statistical analysis provided from Bitcoin. It is also possible to use the informations retrieved to make future predictions about how much the Bitcoin blockchain will grow, using polynomial interpolation on the data. According on how many blocks ago are fetched, it is possible to have an accurate prediction on the blockchain growth for the next few years.

We are going to compare more recent data, retrieved real time, with the Bitcoin one and see the differences of the blockchain growth. Moreover, In the Bitcoin website for blockchain analysis, `blockchain.info` [2], the finer granularity shows data for the last 7 days while we are collecting and monitoring data at every block creation (~ 8-10 min). In that way is easier for us to check if there are any abnormalities in the ledger of public data.

1.4 Outline

This chapter introduces the Blockchain, blocks, mining, fee and other key words related to decentralized digital currencies. Chaper 2 gives a detailed explanation on the main aspects of this new technology and how decentralized cryptocurrencies work. It will talk about consensus protocol, data structure, mining process, gas, fees and proof of work. Chapter 3 explains how the blockchain analytics system was designed and implemented, which data are taken into consideration and why and also how data are retrieved and organized into text files. Chapter 4 talks about the evaluation and the results of the data analysis. It shows the plots generated with some consideration and the comparison between the expected results from the Bitcoin blockchain. Plus, prediction models obtained with statistical analysis are showed. Finally, Chapter 5 includes future implementation of the system, possible development of this project and conclusions of the overall work.

2

Technical Background

This chapter will give a detailed explanation about decentralized digital currencies, explaining all the main concepts and key words that you should know when you read about cryptocurrency. The differences between centralized and decentralized cryptocurrencies are discussed, enhancing their main characteristics, pros and cons. Concepts such as blockchain, proof of work, state, block, transaction, and gas fee are described and discussed.

Before talking about decentralized digital currency with its characteristics is better to underline the differences with a centralized one. As the Table 2.1 shows, the biggest diversity is that the centralized digital currency is under the control of a Central Authority (CA), while the decentralized one only require consensus among the participants in order to make changes and execute transactions. Transaction history is unalterable in a decentralized environment while it could be changed from the CA in a centralized one. Table 2.1 shows also that in decentralized systems, as in all peer-2-peer networks, every user has full control over their assets and data, which means that there is no client and server but every node is both. The Figure 2.1 enhance that the CA provides authentication and authorization in order to execute a transaction from Alice to Bob.

Even though using a decentralized topology has advantages in matter of distribution and participant scalability, it could also have some side effects such as inefficiency if compared to the centralized one that can use relational databases that fit the needs of the application. Furthermore there is the need to give more

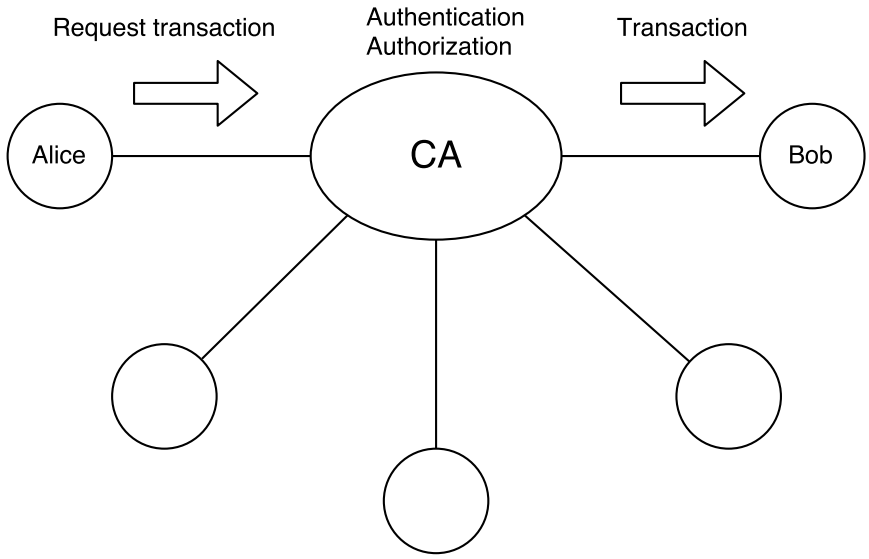


Figure 2.1: Example of centralized digital currency, Alice does a transaction to Bob through the Central Authority.

attention to cryptography and use private-public key technology to authenticate since there is no central authority giving authorization and providing authentication, indeed anyone can join the peer-2-peer network and use the ledger of data. If all the participants have the same privileges then is easier for a malicious node to attack the network.

The most relevant papers about digital currency and blockchain were read, studied and analyzed. They include the Bitcoin [30] paper, which introduce the concept of *proof-of-work* similar to the Hashcash [17] one presented in 2002, the Ethereum white paper [6] which explains the main differences with Bitcoin and it goes into details about the Ethereum structure, logical with *Merkle tree* and physical with *blocks*. Both Bitcoin and Ethereum support the feature to encode rules or script for processing transactions and this feature has developed in Ethereum with the name of *smart contract*. An interesting read about smart contract might be the one proposed from the University of Singapore in 2015, "Making Smart Contracts Smarter" [28].

2.1 Decentralized Digital Currencies

The main decentralized digital currencies at the present are Bitcoin (2008) and Ethereum (2013). They facilitate transactions between consenting individuals who would otherwise have no means to trust each other and deal with geo-

Table 2.1: Centralized vs Decentralized digital currencies

Centralized	Decentralized
Under the control of a central authority	Requires consensous among users to make changes
Users are dependent from the control given from the CA	Users have full control over their assets/data
No need to enforce cryptography or to create keys	User's data is uniquely identified by private key
Only users allowed by CA can participate	Anyone can join and use ledger
Historic transactions can be changed by the CA	Historic transactions are unalterable and permanent
Can be run very efficiently using relational databases that fit the need of the applications	Inefficient for the cost of mining

Table 2.2: Cryptocurrency in Ethereum

Multiplier	Name
10^0	Wei
10^{12}	Szabo
10^{15}	Finney
10^{18}	Ether

graphical separation and interfacing difficulties. According to Nakamoto [30] an electronic coin is defined:

The Definition 3. An *eletronic coin* is a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin.

All the transactions are registered in *blocks* and both systems use the *blockchain* technology to maintain a serial order of them. The blockchain is a consensus protocol that users run to maintain and secure a shared ledger of data and it is formed of all blocks, ordered by time and connected between each other, like shows Figure 2.3. A transaction is accepted only after being approved from all participants in the network (Figure 2.2) and a block is created only after the *block validation* (Section 2.4). Digital currency in Bitcoin is called bitcoin currency (BTC) while for Ethereum is called ether (ETH) (Table 2.2).

Proof-of-work, introduced with Bitcoin and previously with Hashcash, is used to

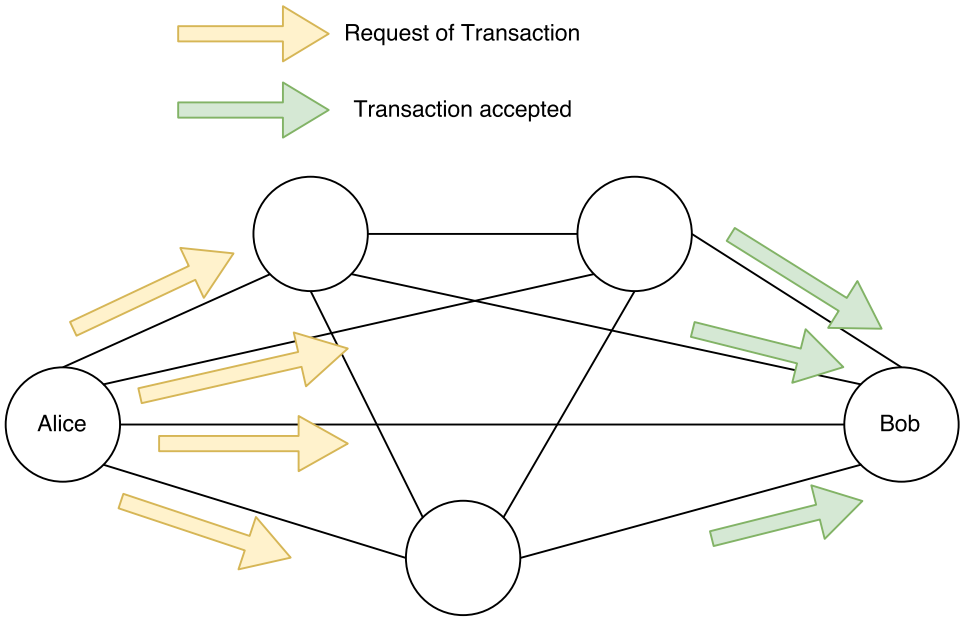


Figure 2.2: Decentralized authorization if Alice wants to transfer money to Bob.

provide authorization of transactions. It makes block creation computationally hard since they have to satisfy certain conditions, in this way a malicious attack would need a computational power higher than all the others participant to the network to change the order of a transaction or to put a "fake" block into the blockchain. *Authentication* is provided with cryptography and public-private key mechanism. In a transaction is present a cryptographic signature produced by the private key associated with the owner's address.

If the physical implementation of a decentralized cryptocurrency system is represented by the blocks, the logical one is characterized by *states* (more about the logical state in Chapter 2.3). All the transactions and blocks are logically stored in a *Merkle tree* which keeps information about the states with all the corresponding transactions. The logic behind this data structure is that every non-leaf node is labelled with the hash of the labels or values of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures [19]. In Ethereum Merkle tree has been replaced with a Patricia tree and it is called ***trie*** (explained in Chapter 2.6). Nonetheless, decentralized digital currencies also have some side effects. The most relevant one is *scalability*, due to the steady growth of the blockchain (Chapter 2.5). It should be also considered that decentralized cryptocurrencies operate in open (or permissionless) networks in which the ledger of data could be manipulated from arbitrary adversaries and according also to the paper from University of Singapore [28] security of smart contracts has not received much

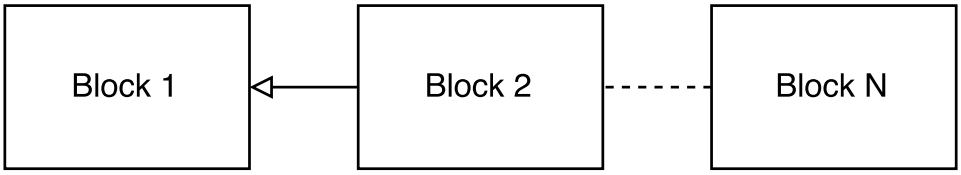


Figure 2.3: Simplified structure of the blockchain.

attention yet. And since the only part not protected from cryptography is the *order of transactions* [6], an attacker would try to convince the network that a transaction occurred earlier than another one to gain money. The security bugs in smart contracts are classified as *Transaction-Ordering Dependence*, *Timestamp Dependence*, *Mishandled Exceptions* and *Reentrancy Vulnerability* [28].

2.2 Transactions And Gas

A transaction is the most relevant piece of information saved in the blockchain. A list of transactions is stored in each block. Transactions must be publicly announced, in that way the earliest transaction is the one that counts, avoiding the *double spending* problem without having a *mint* or a CA. All the nodes then would agree on the order of the transactions thanks to the consensus protocol, described in Chapter 2.5.

2.2.1 Gas And Payment

Before talking about transactions in decentralized cryptocurrencies, is better to introduce the *gas* concept. In order to avoid issues of network abuse, all programmable computation in Ethereum and Bitcoin is subject to fees [36]. These fees are called *gas* and every transaction has a **gasLimit** and a **gasPrice** associated with it. The transaction is considered invalid if the account balance cannot support such a purchase. Any unused gas at the end of the transaction is refunded. Gas does not exist outside of the execution of a transaction.
Transactions are free to specify any **gasPrice** that they wish, however miners (Chapter 2.8) are free to ignore transactions as they please. Chapter 4.4 shows how transactions with insufficient gas can sometimes be ignored by miners.

2.2.2 Transactions

We can define a transaction as:

The Definition 4. A *transaction* (formally, T) is a single cryptographically-signed instruction constructed by an actor with the scope of giving money to someone else in the network.

Each transaction T contains the following attributes:

T_n **(nonce):** A scalar value equal to the number of transactions sent by the sender.

T_g **(gasLimit):** A scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up-front before any computation is done and smay not be increased later.

T_p **(gasPrice):** A scalar value equal to the number of Wei or BTC to be paid per unit of gas for all computation costs incurred as a result of the execution of this transaction.

T_t **(to):** The 160-bit address of the message call's recipient.

T_v **(value):** A scalar value equal to the number of Wei (Table 2.2) or BTC to be transferred.

T_d **(data):** An unlimited size byte array specifying the input data of the message call.

A transaction could also be defined as a valid arc between two states. The "validity" is defined by the *state transaction function* Υ. In Ethereum, Υ together with σ are considerably more powerful then any existing comparable system. Υ allows components to carry out arbitrary computation, while σ allows components to store arbitrary state between transactions [36]. According to the Ethereum white paper [6], the Ethereum state function Υ is defined as follows:

1. Check if the transaction is well-formed, the signature is valid and the nonce matches the nonce in the sender's account. If not, return an error.

2. Calculate the transaction *fee* as $STARTGAS \times GASPRICE$, and determinate the sending address from the signature. Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend return an error.

Table 2.3: Bitcoin transaction vs Ethereum message

Messages (Ethereum)	Transactions (Bitcoin)
Created either externally or from a *contract*	Only created internally
There is an explicit option for Ethereum messages to contain data	–
The recipient of an Ethereum message, if it is a contract account, has the option to return a response	–

3. Initialize $GAS \leftarrow STARTGAS$ and take off a certain quantity of gas per byte to pay for the bytes in the transaction.

4. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution run out of gas.

5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all the state changes except the payment of the fees, and add the fees to the miner's account.

6. Otherwise, refund the fees for all remaining gas to the sender and send the fees paid for gas consumed to the miner.

A transaction in Bitcoin is referred in Ethereum as a *message*. Messages and transactions have the same purpose with only three main differences listed in Table 2.3. In the Figure 2.4 is showed that a transaction changes the state σ to σ' and it needs to have information about the sender and the receiver address with also the value to transfer. The *signature* proves the authenticity of the sender and it is encrypted with the sender's private key according to provide authentication. An example of a contract code's transaction in Ethereum is viewed in Appendix B.5.

2.3 State

The state is the logical representation of a cryptocurrency system. Indeed the state is not encoded in the block (which is the physical representation) in any way; it is purely an abstraction to be remembered by the validating node and can only be computed for any block by starting from the *genesis* state

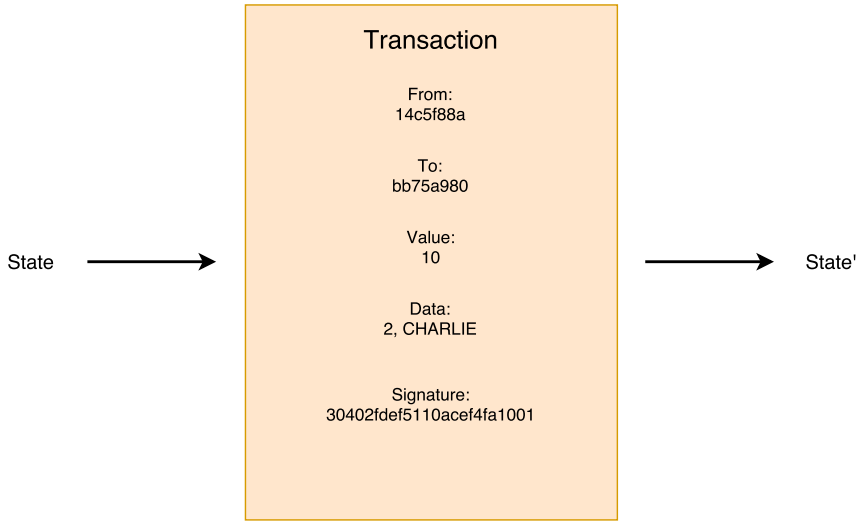


Figure 2.4: Example of transaction between two states, σ and σ'.

and sequentially applying every transaction contained in each block [6]. The definition of a state according to Ethereum [36] is the following:

The Definition 5. The *state* is represented with σ and it is a mapping between addresses (160-bit identifiers) and account states. Though not stored on the blockchain, it is assumed that the implementation will maintain this mapping in a modified Merkle Patricia tree.

Ethereum can be viewed as a *transaction-based* state machine (the concept of state machine is well explained from Jeffrey Ullman and John Hopcroft [34]), where its state is updated after every transaction and a state σ goes to σ' after a certain transaction T, like Figure 2.4 shows.

$$\sigma \xrightarrow{T} \sigma'$$

Having a sort of chain of states, where g is the *genesis* state.

$$g \xrightarrow{T_1} \sigma_1 \xrightarrow{T_2} \sigma_2 \xrightarrow{T_3} \ldots \xrightarrow{T_n} \sigma_n$$

The Ethereum *state transaction function* is represented with Υ and it is defined as follows [36]:

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T) \tag{2.1}$$

The new state σ_{t+1} is the output of the function Υ having as inputs a transaction

T and a state σ_t. The state in Ethereum is mapped in a Merkle Patricia tree (Chapter 2.6) which is called *trie*.

According to Ethereum paper the state has four fields [36]:

nonce: A scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account. For account of address a in state σ, this would be formally denoted $\sigma[a]_n$.

balance: A scalar value equal to the number of Wei owned by this address. Formally denoted $\sigma[a]_b$.

storageRoot: A 256-bit hash of the root node of a Merkle Patricia tree (Chapter 2.6) that encodes the storage contents of the account (a mapping between 256-bit integer values), encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the recursive length prefix (RLP) (Appendix A) encoded 256-bit integer values. The hash is formally denoted $\sigma[a]_s$.

codeHash: The hash of the EVM code of this account. This field is immutable and cannot be changed after contruction. All such code fragments are contained in the state database under their corresponding hashes for later retrieval. This hash is formally denoted $\sigma[a]_c$, and thus the code may be denoted as b, given that $KEC(b) = \sigma[a]_c$.

The concept of state is strongly related with the Merkle tree's one, defined in Chapter 2.6.

2.4 Blocks

If the state is the logical representation of a cryptocurrency system, then the block can be viewed as the physical structure of it. All the transactions ever occurred and approved in the system are stored in *blocks*. We define a Block with B, for *current block* and H for *block header*.

The Definition 6. A *block* is a collection of relevant pieces of information.

In every block one or more transactions are stored. Each block is connected with the predecessor through the information stored in *previousHash*. Below are listed the main informations gathered in the block according to Ethereum

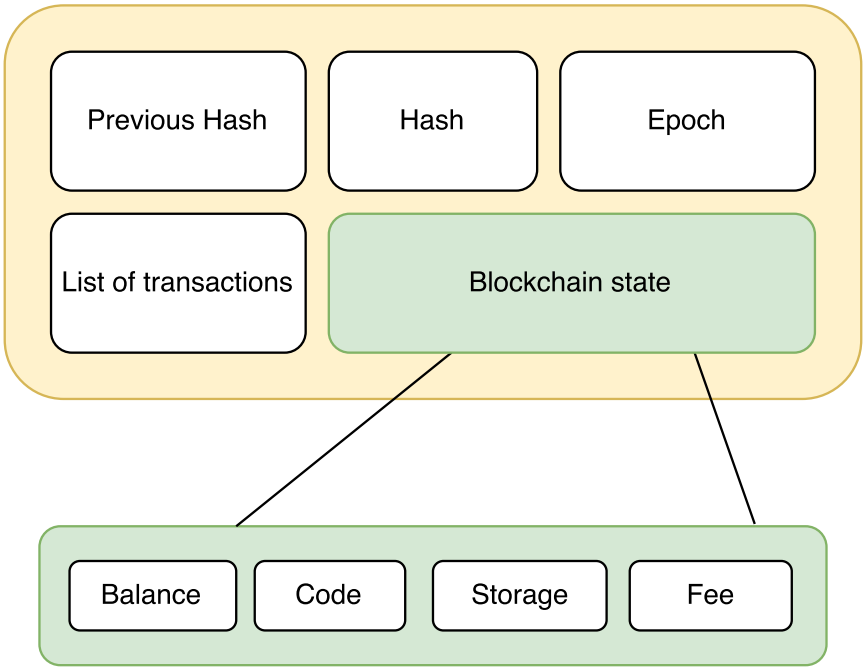


Figure 2.5: Design of a block in popular cryptocurrencies like Bitcoin and Ethereum, enhancing its main attributes.

paper [36].

B_h **(hash):** The keccak 256-bit hash identifier representing the block.

B_p **(previousHash):** The keccak 256-bit hash of the parent block's header.

B_n **(nonce):** A 64-bit hash which proves that a sufficient amount of computation has been carried out on this block.

B_s **(epoch):** or timestamp, is a scalar value equal to the epoch at this block's inception.

B_t **(transactionRoot):** The keccak 256-bit hash of the root node of the *trie* (Chapter 2.6) structure populated with each transaction in the transaction list portion of the block.

B_g **(fee):** is the fee payed to the miner or pool of miners for the computational effort to create the block.

A block needs to be processed and validate before it can be written on the blockchain. This process is called *validation algorithm*, and in Ethereum it is

defined as follows [6]:

1. Check if the previous block referenced exists and its valid.

2. Check that the timestamp of the block is greater than that of the referenced previous block and less than 15 minutes in the future.

3. Check that the block number, difficulty, transaction root, uncle root and gas limit are valid.

4. Check that the *proof-of-work* on the block is valid.

5. Let $S[0]$ be the STATE_ROOT of the previous block.

6. Let TX be the block's transaction list, with n transactions. For i in 0 to $n - 1$, set $S[i + 1] = APPLY(S[i], TX[i])$. If any application returns an error, or if the total gas consumed in the block up until this point exceeds the GASLIMIT, return an error.

7. Let S_FINAL be $S[n]$, but adding the block reward paid to the *miner* (Chapter 2.8).

8. Check if S_FINAL is the same as the STATE_ROOT. if it is, the block is valid; otherwise is not valid.

2.5 Blockchain

We talked a lot about the blockchain so far and this chapter has the goal to explain what the blockchain is.

The Definition 7. The *blockchain* can be defined as a new way of serialization in decentralized systems.

The old paradigm of *centralized consensus*, Figure 1.1 (A), could be replaced by the *decentralized* one, Figure 1.1 (B). Bitcoin and Ethereum's consensus protocols are based on a decentralized scheme, which transfers authority and trust to a decentralized virtual network and enables its nodes to continuously and sequentially record transactions on a public "block", creating an unique "chain": the blockchain [33]. It is very important to enhance that the consensus logic is separate from the application itself. In that way a variety of systems could use the blockchain paradigm whether they are money or not-money related, and applications can be written to be organically decentralized.

The blockchain is like a place where you store any data semi-publicly in a linear container space, the block [33]. The concept of *public-key cryptography* is used in the blockchain to guarantee authorization and authentication. Security then depends only on keeping the private key private, so the public key can be published without compromising security [32]. Anyone can verify the owner of a transaction because of the public signature but only the owner can unlock what's inside because only he has the private key for it. A blockchain then behaves almost like a database, except that part of the information stored (its "header") is public. The problem of security though is not discussed in this thesis. Security in overlay networks is treated in Fireflies paper [24].

We can also define the blockchain as a set of blocks, connected together like shows the Figure 2.3. This connection is possible because of the *previousHash* field, current in all the blocks, Figure 2.5. A blockchain can be viewed then as a *shared single source of truth*. The idea behind the blockchain is to create a simple decentralized consensus protocol, based on nodes combining transactions into a "block" every ten minutes creating an ever-growing blockchain, with proof of work as a mechanism through which nodes gain the right to participate in the system [6].

2.5.1 Security in the Blockchain

In the Bitcoin system the only part not protected by cryptography is the *order of transaction*, so the only way for an attacker to gain from this would be [6]:

1. Send money to a merchant in excange for some product.

2. Produce another transaction sending the same amount of money to himself.

3. Try to convince the network that his transaction to himself was the one who came first.

When the merchant accepts the payment the attacker can't just change the block previously created since that one was accepted by the blockchain and other new blocks were appended to it. The attackers then should fork a totally new blockchain starting from that block, but to do so it would need to re-do the proof of work (Chapter 2.8.1) for each new block and so it will need more computational power than all other miners in the network combined. This system gives more importance to computational power rather than number of participants (like in Byzantine fault tolerance protocol) and a malicious node would need more computational power than all the other nodes in the network put together.

2.6 Merkle Tree

We talked about the state as the logical representation of a digital currency. All the states concerning a block are organized in a logical structure, called Merkle tree. The latter is a multi-level data structure and every block is stored in it (the hash of the block to be precise). This is important for *scalability* since the "hash" of a block is only the hash of a block header, 200-byte piece of data containing the timestamp, nonce, previous block hash and the root hash of this data structure called Merkle tree. The Merkle tree stores all the transactions in the block and is defined as follows [6]:

The Definition 8. a *Merkle tree* is a type of binary tree where all the informations are stored in the leafs, so it has a huge number of end-nodes, each node is the hash of its two children and finally a single root node, also formed from the hash of its two children representing the top of the tree.

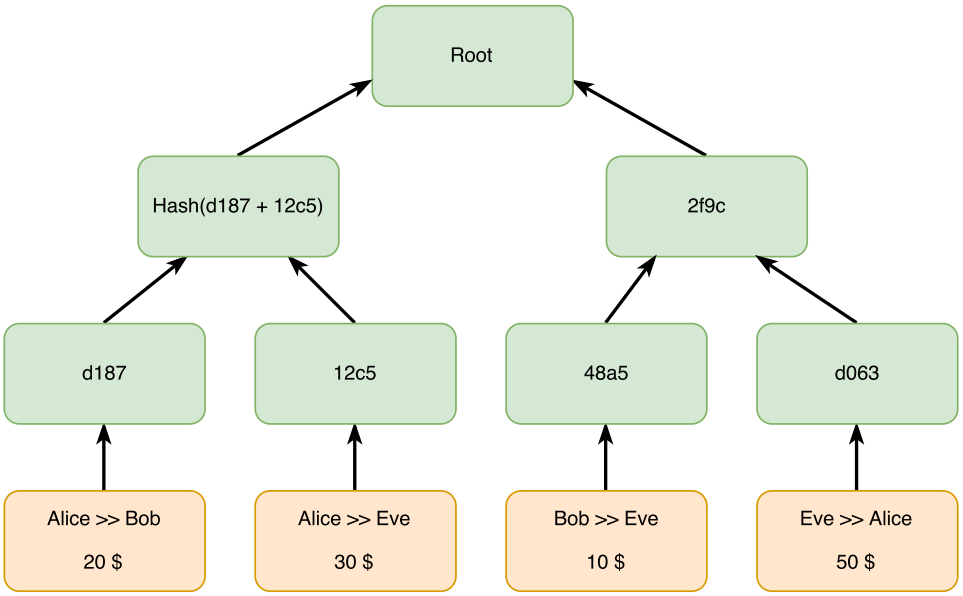


Figure 2.6: Example of a Merkle Tree, the leafs are the transactions in the block and the parent nodes are the hash of their children. Hashes are propagated upward.

The purpose of a Merkle tree is that a node can download only the header of a block from one source, a small part of the tree relevant to them, and still be assured that all of the data is correct. The reason why this works is that hashes propagate upward [6]. The Figure 2.6 shows how these hashes are propagated, not allowing to a malicious attacker to swap in a fake transaction into the bottom of a Merkle tree since this change will cause a change in the node

above, and then a change in the node above that, finally changing the root of the tree and therefore the hash of the entire block causing an invalid proof of work (Chapter 2.8.1) and appearing as another new block that is ready to be saved. In other words, a Merkle tree keeps all the information about a certain block, storing all the transactions in its leafs and it does that in a secure way, avoiding any tamper of the data.

2.6.1 Patricia Tree

In Ethereum, Merkle tree has been modified in a *Merkle Patricia tree* and it is called *trie*. Merkle Patricia trees provide a cryptographically authenticated data structure that can be used to store all (key, value) bindings. Both trees are fully deterministic meaning that given the same (key, value) bindings is guaranteed to be exactly the same down to the last byte [3]. Furthermore, Patricia tree guarantee a $O(log(n))$ time for *lookups*, *inserts* ans *deletes*.

The *storageRoot* of the state mentioned in Chapter 2.3 represents the root of a Merkle tree in the following way [36]:

$$TRIE(L_I^*(\sigma[a]_s)) \equiv \sigma[a]_s \tag{2.2}$$

Where the function L_I^* represents the set of $(key, value)$ pairs stored in the Merkle tree in that way:

$$L_I((k, v)) \equiv (KEC(k), RLP(v)) \tag{2.3}$$

The $(key, value)$ pairs are encrypted and then stored. The key is encrypted with Keccak-256 while the value using RLP.

2.6.2 Scalability Issues

According to Ethereum White Paper [6], a *full node* (Appendix A) in the Bitcoin network, takes up about 15 GB of disk space in April 2014, and it is growing by over a gigabyte per month, allowing in the future only large business to run the full nodes and to participate the network. This is why the Merkle tree protocol is arguably essential to long-term sustainability. Some protocols to prevent that already exist. One is a protocol known as simplified payment verification (SPV) which works by fetching only the data needed. It downloads the block headers, verifies the proof of work on the block headers, and then downloads only the "branches" associated with transactions that are relevant to them. These nodes are called *light nodes*(Appendix A).

2.7 Smart Contracts

Smart contracts are discussed in several papers [20, 28, 10, 29] and they are all related to Ethereum. The state defined in Chapter 2.3 is viewed in Ethereum as an *account*. It contains four fields:

- nonce,
- ether balance,
- contract code,
- the account's storage.

In Ethereum there could be of two types of contracts (Table 2.4): *externally owned accounts*, contarolled by private keys, and *contract accounts*, controlled by their contract code.

The Definition 9. A *smart contract* in Ethereum is an autonomous agent stored in the blockchain, encoded as part of a creation transaction that introduces a contract to the blockchain.

Table 2.4: Contracts in Ethereum

Externally owned accounts	Contract accounts
Controlled by private keys	Controlled by their contract code
It has no code	Every time it receives a message its code activates
Messages are sent by creating and signing a transaction	The code allows to read and write to internal storage and send other messages or create contracts in turn

A smart contract is identified by a *contract address*. Each contract holds some amount of virtual coins (Ether), has its own private storage, and is associated with its predefined executable code. The code of an Ethereum contract is in a low-level stack based bytecode language referred to as Ethereum virtual machine (EVM) code. Users define contracts using high-level programming language which are then compiled into EVM code [28]. In Appendix B.2 there are some example of contract code, taken from the Ethereum's guide website [13]. A smart contract allows to personalize an account, like for example, freezing an account if a certain condition is verified, executing or solving computational puzzle that allow transactions to be successfully registered and more, acting like a proper CA that provides all this options for you.

An example taken from the "Making Smart Contracts Smarter" [28] paper and represented in Appendix B.2, shows a smart contract able to give a reward to users who solve computational puzzle. First an unique address for the new contract is prepared. Then the contract's private storage is allocated and initialized by running the constructor. Finally, the executable EVM code portion is associated with the contract [28].

2.8 Mining

In a decentralized cryptocurrency network transactions need to be approved by all the participant before being validated and confirmed. This validation requires high computational time and it might be expensive in matter of electricity and resources used.

The Definition 10. *Mining* is how transactions are validated and confirmed by the network.

To validate and confirm transactions, the network needs *miners*. According to the Bitcoin miner document [11], Bitcoin miners create new blocks by solving a proof of work problem that is chained through cryptographic proof of the previous block. Bitcoin uses a proof of work (Chapter 2.8.1) system similar to Adam Back's Hashcash [17] while Ethereum is moving to Casper. The work and the effort spent to create new blocks is often referred as mining [11]. The mining process involves identifying a value that when hashed twice with SHA-256, begins with a number of zero bits. If the amount of zeros needed for the creation is raised, the average work required increases exponentially, while a hash can always be verified by executing a single round of double SHA-256 [11, 31].

A miner choose whether include a transaction into a block or not, according to the fee that this transaction has to offer. Meanwhile the miner tries to solve the proof of work. The first miner who solve it get the fee paid from all the transactions included in it. After a block is created all the network is informed, the block is validate and all the transactions in it are accepted and confirmed. At the end, nodes express their acceptance by moving to work on the next block, incorporating the hash of the accepted block [11].

2.8.1 Proof-of-Work

Cynthia Dwork and Moni Naor defined the proof of work as follows [21]:

The Definition 11. *Proof of work* is a piece of data which is difficult (costly,

time-consuming) to produce but easy for others to verify and which satisfies certain requirements.

Mining a block is difficult because the SHA-256 hash of a block's header must be lower or equal to a certain *target value* in order for the block to be accepted by the network. Producing a proof of work can be a random process with low probability so that a lot of trial and error is required on average before a valid proof of work is generated [21].

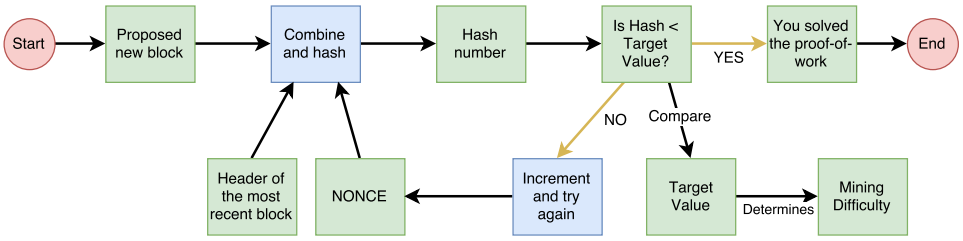


Figure 2.7: Diagram that shows how proof of work works, inspired from `https://www.bitcoinmining.com/` [12].

As said above, proof of work is a part of the mining process and a valid proof of work is determined by incrementing a nonce until a value is found that gives the block's hash the required number of leading zero bits. Once the hashing has produced a valid result, the block cannot be changed without redoing the work. As later blocks are chained after it, the work to change the block would include redoing the work for each subsequent block [11]. The best chain is the longest one, so is the one that requires the greatest amount of effort to be produced, and majority consensus in Bitcoin and Ethereum is represented by this longest chain. In this way if the majority of the computing power is controlled by honest nodes, the honest chain will grow fastest and outpace any competing chains. A very good diagram which represent the proof of work is presented in Figure 2.7.

```
"Hello, world!0" => 1312af178c253f84028d480a6adc1e25e81caa44c749ec81976192e2ec934c64
"Hello, world!1" => e9afc424b79e4f6ab42d99c81156d3a17228d6e1eef4139be78e948a9332a7d8
"Hello, world!2" => ae37343a357a8297591625e7134cbea22f5928be8ca2a32aa475cf05fd4266b7
...
"Hello, world!4248" => 6e110d98b388e77e9c6f042ac6b497cec46660deef75a55ebc7cfdf65cc0b965
"Hello, world!4249" => c004190b822f1669cac8dc37e761cb73652e7832fb814565702245cf26ebb9e6
"Hello, world!4250" => 0000c3af42fc31103f1fdc0151fa747ff87349a4714df7cc52ea464e12dcd4e9
```

Figure 2.8: In this example the difficulty of creation is to have 4 zeros at the beginning of the hash and in this case it will take 4251 attempt before the new block is created. This block satisfy the proof of work. Example taken from bitcoin Wiki [21].

In the Figure 2.8 is listed an example of how proof of work works. The "Hello, world!" message takes 4251 attempts before being created in a way that it satis-

fies the proof of work. The difficulty of the proof of work may change according to the time taken to satisfy it (this is explained in Chapter 2.8.3).

2.8.2 Miners

With paper money a government decides when to print and distribute money. Bitcoin or Ethereum don't have a central government and they use a special software to solve math problems to get a certain amount of coins in exchange. This provides a smart way to issue the currency.

The Definition 12. A *miner* is a computer specifically designed to solve problems according to the proof of work algorithm and they are required to approve transactions.

Since miners are required to approve transactions, more miners means a more secure network. Like the Figure 2.9 shows, a transaction from Alice to Bob needs to be approved from all the miners (M) in the network before it can be saved and stored in the blockchain and then the transaction been accepted.

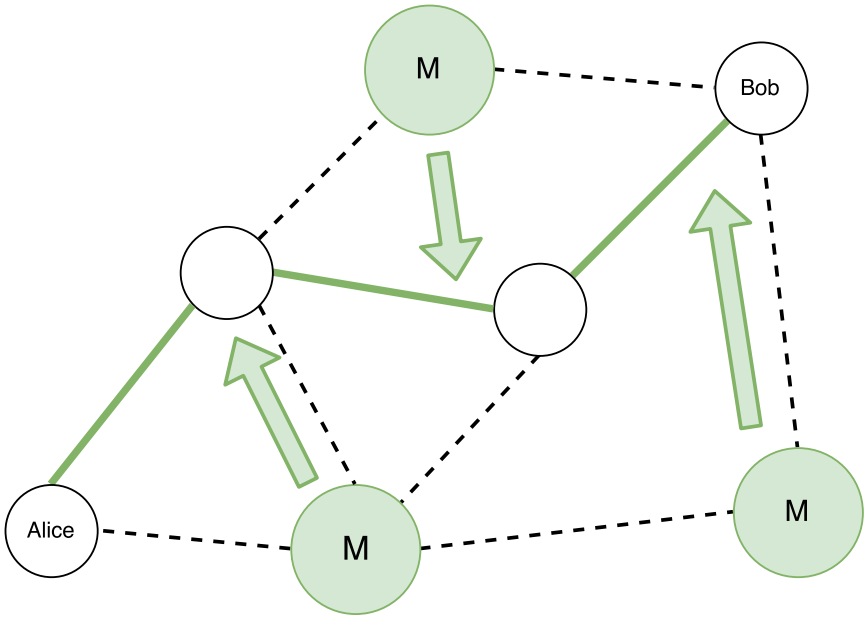


Figure 2.9: Transaction from Alice to Bob needs to be approved from all the miners (M) in the network before it is reported on the ledger.

To become a Bitcoin miner nowadays you need to buy highly specified chips called application specific integrated circuits (ASIC). This hardware can be found at `bitcoinmining.com` [12] and it is showed in Figure 2.10. In early days of Bitcoin it was possible to mine with your computer CPU or high speed video

processor card. Today that's no longer possible since custom Bitcoin ASIC chips offer performance up to 100x the capability of older systems [12] and another reason is the constantly growth of the *mining difficulty* (Chapter 2.8.3).

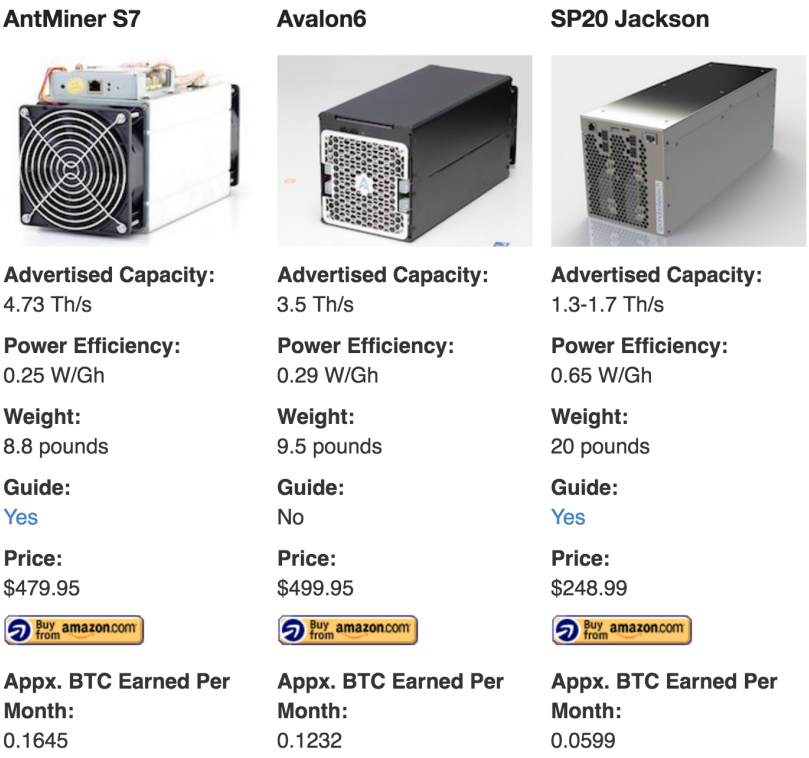


Figure 2.10: Best Bitcoin miner options, according to price per hash and electrical efficiency [12].

2.8.3 Mining Difficulty

The cryptocurrency networks automatically change the difficulty of the math problems depending on how fast they are being solved. The difficulty of this work is adjusted so as to limit the rate at which new blocks can be generated by the network to one every 10 minutes. Due to the very low probability of successful generation, this makes unpredictable which worker in the network will be able to generate the next block [21].

The Definition 13. *Mining difficulty* is a measure of how difficult it is to find a hash below the target value (a 256-bit number) during the proof of work.

More the mining difficulty goes up, more the target value goes down, additionally, more the mining difficulty goes down, more the target value goes up. In

Figure 2.7 it showed how the target value is used in the calculation of difficulty, so that the hash found needs to be lower than this target value. How difficulty works is very good explained in the Figure 2.11. When more miners join the network more blocks are created in the same amount of time, so the average mining time decreases. When that happens, the mining difficulty increases and the block creation rate goes down again, bringing back to normal the average mining time, until more miners will join again.

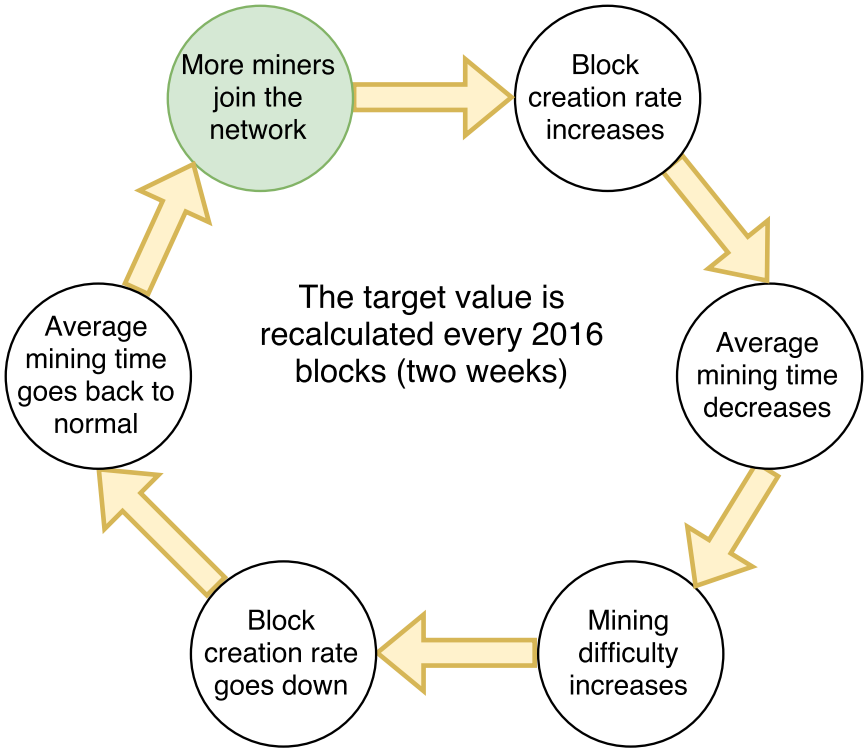


Figure 2.11: How difficulty works in Bitcoin. Picture inspired from `bitcoinmining.com/` [12].

2.8.4 Pools

As the popularity of decentralized cryptocurrencies increase more miners join the network making more difficult for individual to solve the math problems. This is because of the difficulty increment in the network as showed from the website `tradeblock.com` [14] in Figure 2.12. To overcome this miners have developed a way to work together in pools [12].

The Definition 14. A *pooled mining* combines the work of many miners toward a common goal.

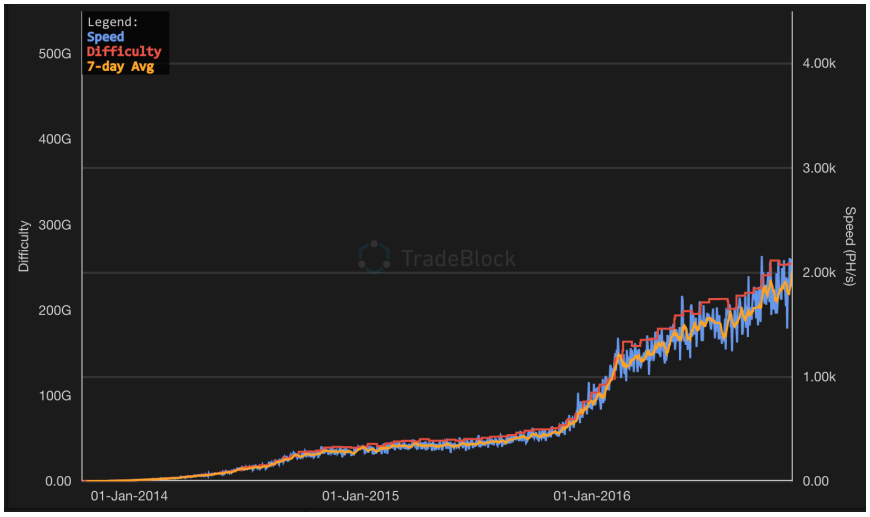


Figure 2.12: Graph of the Bitcoin blockchain difficulty increment since 2014. Taken from `https://tradeblock.com` [14] in November 2016.

Pools of miners find solutions faster than individual member and each miner is rewarded proportionally with the amount of work it provides. Mining is an important and integral part of Bitcoin that ensures *fairness* while keeping the Bitcoin network *stable*, *safe* and *secure* [12].

3

Blockchain Analytics System

In order to understand digital cryptocurrencies systems, to test and make experiments on the blockchain, a complete data analytic system was designed and implemented. This chapter will explain how the blockchain analytics system on Bitcoin was implemented, starting from its architecture until the API used and then how data are processed and manipulated. The system enables real-time observations of the real Bitcoin blockchain.

Our main design considerations for this system were proper handling of:

- Growth of the blockchain,
- average size of a block,
- *r*ead and write *bandwidth*, and
- relation between *fee* paid and *bandwidth*.

Some considerations were done also on the *latency* on blocks retrieval. To speed up data processing, part of the blockchain, miners, and nodes IP addresses are saved locally in text files. Plots are generated in the folder plot/. Some of the result we obtained using this system are presented in Chapter 4. The goal of this

system is to enable analytics on specific portion of the blockchain, retrieving data, manipulating and analyzing them, and after that, making consideration and comparing the result with the statistics provided from Bitcoin. We expected to find a correlation between the fee paid to mine a block and the creation time, plus, we wanted to define a model for the blockchain growth, since the Bitcoin predictions turned out to be wrong.

3.1 Blockchain Data Sources

There are many websites that observe and make analysis on the existing blockchains. The most popular websites for Bitcoin and Ethereum blockchain analysis are `blockchain.info` [2] for Bitcoin, `etherscan.io` [4] and `etherchain.org` [5] for Ethereum. Remote API on the Bitcoin blockchain could be found at `blockchain.info` in the API section. These are API to retrieve data directly from the website and they can be used in a Python application simply as a HTTP request/response. The system we implemented retrieves data from the Bitcoin blockchain, using the API provided from `https://github.com/blockchain/api-v1-client-python`. This API is well suited for Python and provides a nice access to the Bitcoin blockchain, allowing the retrieval of all the information stored in a block and in a transaction.

In the api-v1-client-python mentioned above, the class containing the structure of a block and transaction, used for data retrieval and analysis is *blockexplorer.py*. The object structures of a block and a transaction in Python language are represented in Appendix B.4, B.5. Furthermore, in Appendix B.6 is showed a JSON representation of the block which is returned from the API call.

3.2 System Architecture

The architecture of our blockchain analytic system is showed in Figure 3.1. The system is run mainly locally, importing Bitcoin API and using them for data retrieval to the local system. The website `blockchain.info` provides API to be used remotely with an HTTP request, and they can be easily integrated in the application with a HTTP request/response mechanism. The website `blockchain.info` contains all the information concerning the Bitcoin blockchain and it could be queried both through Python API and web API. This website is monitoring 24-7 the blockchain, producing graphs and statistical analysis on the data observed in the real decentralized network. The local application is monitoring these data as well, producing graphs that are not taken into consideration from the `blockchain.info` website, using a finer granularity that represent the data.

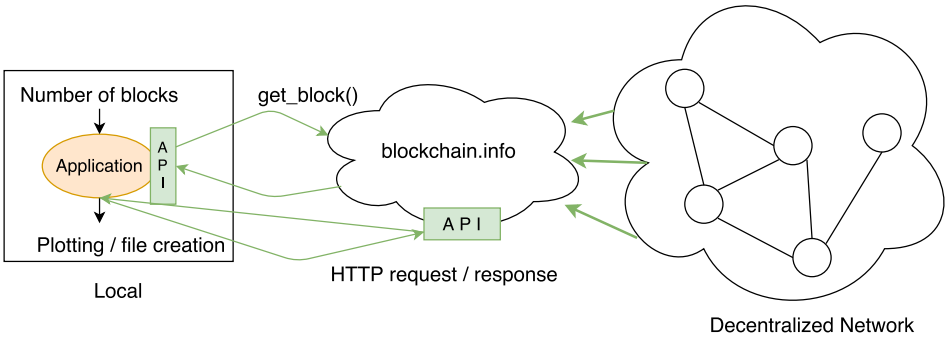


Figure 3.1: Architecture of the developed system for blockchain analysis.

Moreover, the application generates three text files:

blockchain.txt: text file containing information about the portion of the blockchain retrieved, including block hash, height, size, fee and time for each block. These data are important for the later analysis and the file is structured like the following:

```
hash:           block_hash
epoch:          block_epoch
creation_time:  time_mining_block (s)
size:           block_size (byte)
fee:            block_fee (satoshi)
height:         block_height (block_number)
bandwidth:      read_bandwidth (MB/s)
transactions:   number_transactions_in_block
avgttime:       avg_tr_time (s)
```

mining_nodes.txt: text file containing all the IP addresses of mining nodes or pools concerning the blocks in the file "blockchain.txt".

nodes_in_the_network.txt: text file containing all the IP addresses involved in relaying transactions in the network.

The application for blockchain analysis is implemented in the file *observ.py*, it provides to call API functions and to generates all the output files. A small example of how the API calls work in the system is showed in Appendix B.9. Our blockchain analytic system was implemented on a MacBook Pro with MacOS Sierra *v*10.12.1, with a 2.8 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 of RAM, using JetBrains PyCharm (*v*2016.2.3) software as text editor and *Python v*2.7.12 as programming language.

3.2.1 Data Retrieval

Data in the blockchain can be retrieved in different ways: blocks can be added to the top of the file, so the most recent blocks are fetched, or appended to the last element, so the blockchain is analyzed even further in the past. To fetch the most recent block the method showed in Appendix B.9 is used. First, the latest block is fetched, then from the previous is retrieved and so on. The latest block is saved with a structure showed in Appendix B.7. When data are appended to the last block, instead, is necessary first to get the hash from the last element in the "blockchain.txt" file, then the last block is retrieved using the method *get_block(hash)* provided from the Bitcoin API. The implementation of the append is found in Appendix B.8.

Blockchain is meant to be ordered linearly and every block should have its predecessor, for this reason blocks are collected in a way that there will not be any gap in the local blockchain, and if the number of blocks that the user tries to fetch is less than the blocks already added to the global blockchain, then would be impossible to add any new block. In this case a better option would be to *update* the local blockchain (-u command, view the application usage Appendix B.3), which means that the right number of blocks that will complete the local chain will be automatically retrieved. A simplified scheme to add blocks on the local blockchain is showed in Figure 3.2. To guarantee that the order is respected and the blockchain is saved locally in a correct way, a validity check is done every time the application is executed. This check verifies that each block in the local blockchain has the right predecessor by controlling that there is a diminishingly height by one going from one block to another (Appendix B.17).

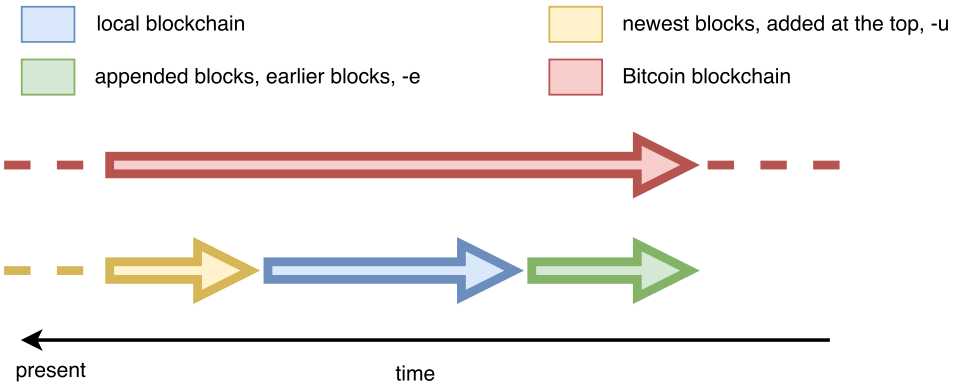


Figure 3.2: Scheme that shows how blocks are written in the local blockchain file.

Once blocks are retrieved a *file.write* is performed. Writing on blockchain.txt can be done by appending blocks at the end or adding them at the beginning of the

file. As Appendix B.12 shows, the adding part is much more complicated since the file needs to be deleted and then written again for each block that needs to be added. As showed in Chapter 3.2 before plotting and getting any kind of information from the data, they are evaluated and manipulated for our purposes. The blockchain.txt file saved, contains information about block hash, height, size, fee, epoch time, creation time, read bandwidth, number of transactions and the average transaction visibility time (average write bandwidth for a block). Of course not all these data are provided from the Bitcoin API, so they are calculated and manipulated runtime, then saved in the text file.

3.2.2 Data Manipulations

Data manipulation is done both before (to save data correctly) and after (to plot data correctly) data are collected in the blockchain.txt file. Once data are retrieved and saved, they are ready to be manipulated in a way that the right and needed information can get out of them.

Each block B, retrieved from the blockchain.txt file, has the following attributes. The *hash*, $B.hash$, is a string containing the 256-bit hash for the block B. The *epoch*, $B.epoch$, which represents the epoch time when the block was created and got visible in the blockchain. This means that the retrieval time is in seconds, starting from the epoch which is 1^{st} January 1970 at 00:00:00. The time is always collected as seconds and if needed, is converted in minutes and hours and saved in other lists. The *size*, $B.size$, is represented in bytes, so every time that MB are needed, the size is divided by 10^6. The growth of the blockchain is represented in GB and the block size and the read/write bandwidth in MB. The *fee*, $B.fee$, represents how much the miner of the block get in satoshi (Appendix A) to mine a certain block B. To be more readable, plots using satoshi are converted in BTC. The *creation time*, $B.creation\_time$, is represented in seconds, and it tells how much time a block needs to be mined. The *number of transactions*, $B.transactions$, tells how many transactions are accepted and stored in the block B. The *write bandwidth*, $B.avgttime$, is the average of the all transactions acceptance time in a certain block and it is measured in seconds. A transaction T is accepted when the block B is created so the $read\_bandwidth_T$ will be:

$$read\_bandwidth_T = B.epoch - T.time \tag{3.1}$$

where $T.time$ is the epoch representing the transaction request.

The read bandwidth is calculated as showed in Appendix B.10. Start time and end time are calculated before and after the block retrieval call, with the function datetime.now(). Time is manipulated to be showed in seconds and

then the ratio to get the bandwidth between MB/s is done and the value is appended in a list that will be written in the file.

The write bandwidth is calculated first for each transaction in the block, as the Appendix B.11 shows, and then an average is done for each block and returned as a data to append in a list containing all the write bandwidths for each block retrieved, this data is written in the blockchain.txt file as *avgttime*.

To represent the growth of the blockchain, a growing time and size list needs to be generated. Assuming that a block is represented with B and its creation time is $B.epoch$, we define the block creation time in the following way:

$$creation\_time_n = \begin{cases} 0, & \text{if } n = 0 \\ B.epoch_n - B.epoch_{n-1}, & \text{if } n > 0 \end{cases} \tag{3.2}$$

where $B.time_{n-1}$ is its predecessor. The growth of the blockchain is calculated by comparing the sum of the creation time and the sum of the block size each time, generating ever growing lists as shows the Appendix B.15. Assuming that $B.size$ is the block size, then:

$$growing\_time_n = \begin{cases} 0, & \text{if } n = 0 \\ creation\_time_n + creation\_time_{n-1}, & \text{if } n > 0 \end{cases} \tag{3.3}$$

and:

$$growing\_size_n = \begin{cases} 0, & \text{if } n = 0 \\ B.size_n + B.size_{n-1}, & \text{if } n > 0 \end{cases} \tag{3.4}$$

where $growing\_time$ is the list containing the sums of the creation time every time that a new block is created and $growing\_size$ is the list containing the size of the blockchain as every block is added.

One of the reason why Python was chosen as a programming language is because data structures are easy to manage and with a Python list you can change the whole data in it using only one line of code (Appendix B.13). It is very easy indeed to swap an entire list from minutes to seconds, from seconds to hours and vice versa. Furthermore, mature API bindings do exist for Python, and it is a very intuitive language with a lot of useful libraries for data manipulation and plotting.

3.2.3 Methods

An UML diagram of the main methods used in the application is represented in Figure 3.3. The application usage is described in Appendix B.3 and the main methods in the systems are:

***get_blockchain(n, [hash])*:** method that retrieve the blocks from the Bitcoin blockchain and saves them in lists that will be the input of the write_blockchain() method. If the attribute hash exists then the retrieval starts from that hash and not from the latest block created in the Bitcoin blockchain. Part of it is represented in Appendix B.8.

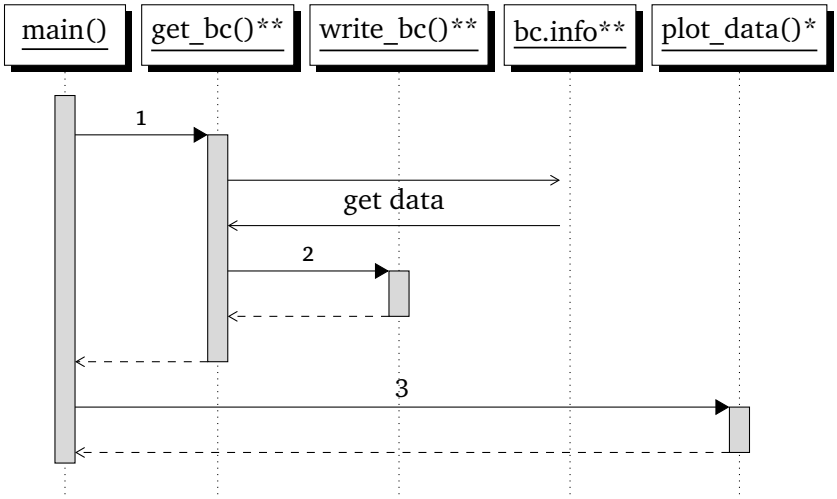
***write_blockchain(<lists to write>, [to_append])*:** method that takes lists in input and write them into a file, according to the parameters that it gets. If to_append is True, then data are appended otherwise are added, like shows Appendix B.12.

***get_list_from_file(attr)*:** this method is probably one of the most important concerning data manipulation. It creates a list, given an attribute as input. Such attribute is a block information stored in the local blockchain and the list generated contains all the values from every block, concerning that particular attribute. Lets say that informations about the block hash want to be analyzed, then it would only be necessary to make a call to this method giving as attribute the string "*hash*", like showed in Appendix B.14. In that way the return list will contain hashes about all the blocks retrieved with the most recent hash In list[0]. Regular expression [15] is used to analyze the file and read from it, in this way is possible to interact with the saved data and get informations from them. This method allows the function plot_data() to easily display statistics.

***plot_data(d, n, [r], [s], [e])*:** this method gets the data from the txt file and then plots all the information retrieved using Matplotlib libraries [8]. It has a description, d, a number of the plot, n, and an optional parameter about the regression, r. It also has the possibility to chose the range of the plot with the start, s, and end, e, parameters. If in the local txt file are present 10000 blocks, is possible to call the method with $start = 7000$ and $end = 9000$, in that way only the blocks in between 7000 and 9000 will be taken into consideration for the plots. In that way is easier to verify if the predictions on the blockchain growth were accurate or not.

The *main()* method provides to call the others and if the general responsibility assignment software patterns (GRASP) principle is followed [26], then this method would be the controller, dispatching calls to other methods for collecting, analyzing and plotting data.

Not all the methods implemented in the system are listed above. Some include data management or check of the blockchain status, such as converting the epoch time in date time format $DDMMYYYY$, get the number of blocks currently present in the local blockchain or create the growing size and time lists. The method *add_mining_nodes(block)* concerns the creation of a file containing



1. call the method *get_blockchain()* for blocks retrieval
2. write data in the txt.file
3. call the plot function for data plotting

* not every function or method is showed in the diagram, the function plot_data() calls some methods for data management.

** the word blockchain has been replaced with the shortcut bc

Figure 3.3: UML sequence diagram of how the application works, where the threads are the methods implemented in *observ.py* file.

mining nodes and a file containing nodes involved in every transaction. The second one are the nodes that rely transactions in all the block retrieved. This means that for each block given in input, every transaction is analyzed, and if the node relying this single transaction is not in the file yet, then the IP for this node is added to it.

3.3 Version Control

For the version control on the source code, a public *git* repository at: `https://github.com/ted92/blockchain.git` was created. Git version *2.6.4* is used in the local environment and every significant update is pushed to the repository to keep an history of all the changes and how the application was developed.

4

Blockchain Observations

In this chapter we present observations of the Bitcoin blockchain captured by the analytics system outlined in Chapter 3. To evaluate our problem definition in Chapter 1.2, we focus on consideration related to read/write bandwidth, the growth of the blockchain, and the relation between the fee paid and the bandwidth achieved. The evaluations are made by analyzing the plots generated in the plot/ folder using Matplotlib [8]. A large number of test has been done, considering always a different number of blocks fetched, at different time. We show that the fee paid is related to the amount of time that a block needs to be mined and be visible in the whole network. This affects average bandwidth available to an application.

4.1 Blockchain Growth

To understand the capacity of the Bitcoin blockchain, we first study its size and how it grows over time. For this, we've been analyzing the blockchain periodically from the 21^{st} September 2016 until the 11^{th} December 2016 using the system outlined in Chapter 3. The plotted growth is shown in Figure 4.1. We observe that the blockchain size grows of 10, 139 GB in 1940 hours. That means a bandwidth usage of $\sim$ 5, 22 MB per hour. Our findings are inconsistent with the observations from 2014 [6] that observed a growth of $\sim$1 MB per hour. Our observations indicate that the growth of the Bitcoin blockchain is $5x$ times bigger than it was expected in 2014 and we believe that a more accurate model

for the blockchain growth can be created.

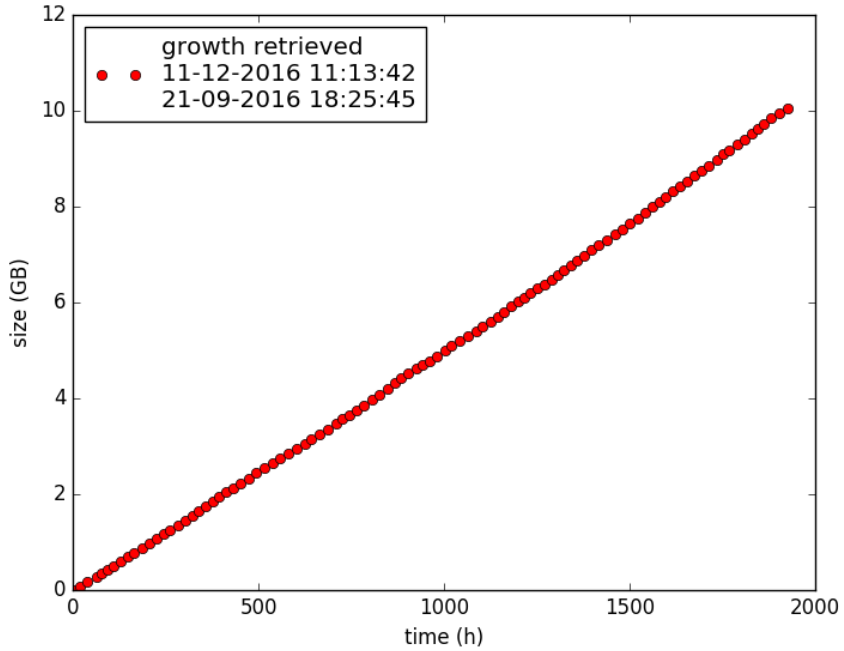
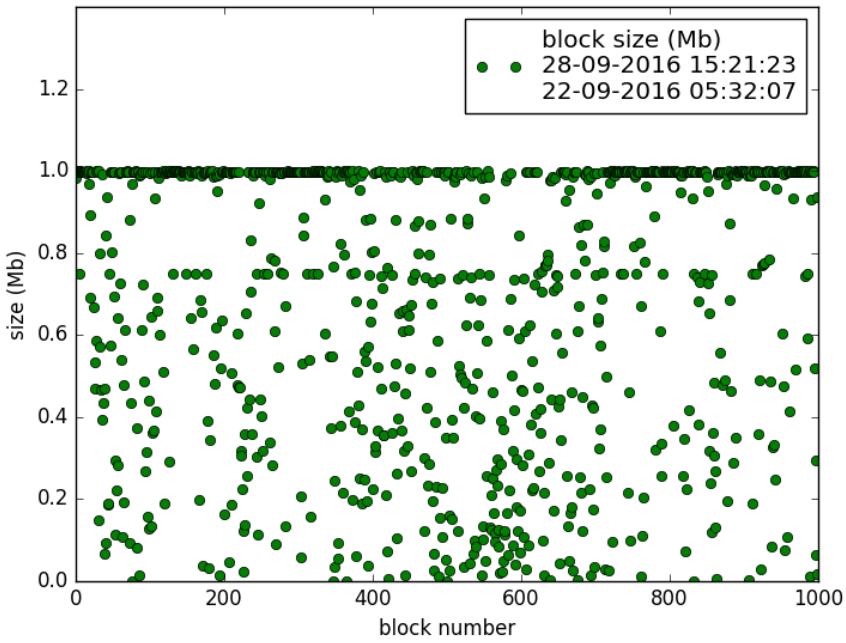
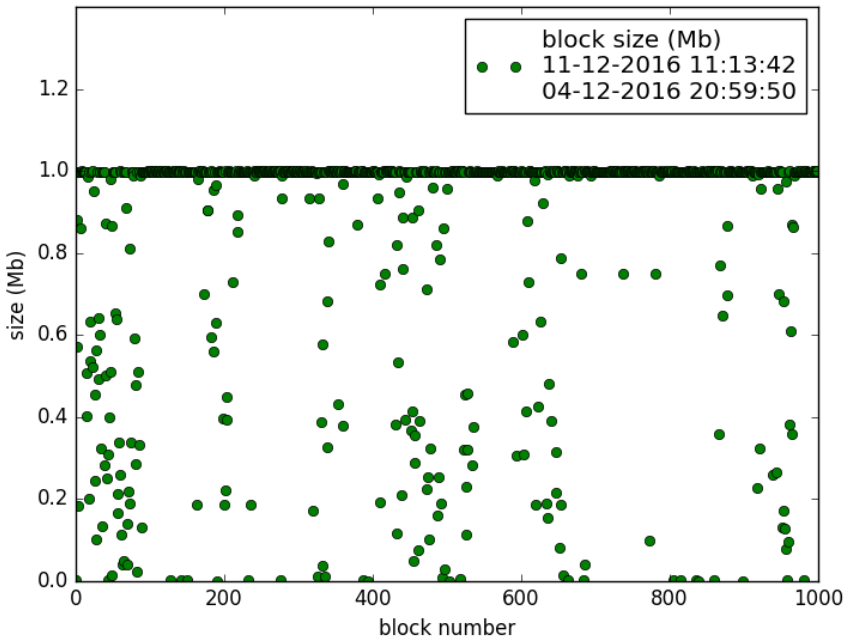


Figure 4.1: The Bitocin blockchain growth according to our analytics system. 12091 blocks are fetched between the 21^{st} September 2016 and the 11^{th} December 2016.

From the plot in Figure 4.1 the growth seems to be linear. However, evaluations made using polynomial interpolation on these data show that the growth has, even if small, a coefficient on the x^2. Our hypothesis is that the non-linear growth is due to a small increment of the block size during the years. To prove this, two different measurements have been done to our local blockchain, showed in Figure 4.2. The first one, Figure 4.2a, considers 1000 blocks fetched from 22^{nd} September 2016 until 28^{th} September 2016, shows that the block size tends to be 1 MB but with quite a lot of blocks having also different sizes. While the latest one, in Figure 4.2b, shows that almost every block out of 1000 are tending to the $\sim$ 1 MB size, having only few blocks under the 1 MB "line". The second measurement was evaluated between 04^{th} December 2016 and 11^{th} December 2016. Furthermore, if the blocks from 2009 in the blockchain are analyzed, their sizes tend to be $\sim$200 kB while in this last months of analysis, December 2016 the block size tend to be of $\sim$ 1 MB. This brings us to believe that the average block size is growing with time, hence the blockchain growth is not linear as previously claimed [6].



(a) Late September 2016.



(b) Mid December 2016.

Figure 4.2: Size comparison of 1000 blocks with more than 2 months of gap.

4.2 Retrieval Block Time

The biggest problem analyzing the blockchain was that the API allows to retrieve only one block per time, making the block fetching incredibly slow. This is the reason why the block retrieval is kept separate from the analytics part, in that way is possible to fetch a fewer number of blocks, adding them time by time and at the end make analysis on them. In Figure 4.4 the read bandwidth for each block retrieval is represented. As said before, the problem of a single-block-fetch affects drastically the latency, and then the read bandwidth as well, taking an average of $\sim$ 1 seconds to retrieve a single block, and in some blocks the read bandwidth is even less than 0, 5 MB/s.

4.3 Block Analysis

The most important attributes of blocks are their size, their creation times, and the number of transaction in each individual block. In this section, we will compare this three attributes to see if there could be any relation between them. The plot in Figure 4.3 shows that the block size tend to be $\sim$ 1 MB. This observation does not depend on how many transactions it contains or how much time it needed to be mined. No relation also was found between the number of transactions in a block and the block creation time. In the Figure 4.3 sometimes the blue line (block creation time) follows the red one (number of transactions in one block), so if one grows then also the other gets higher, but in other cases there is a reverse dependence, so if the number of transactions grows, the block creation time decreases. This unpredictability is good for the security of the blockchain, since malicious node cannot find any dependence on the data which are more representative for a block.

Figure 4.5b shows the creation time for each block, retrieving blocks from the Bitcoin blockchain between the 4^{th} and 12^{th} December 2016. How mining works is described in Chapter 2.8 and according to the difficulty, which is constantly adjusted, a block should be mined every 10 minutes [12]. This claim tend to be verified for most of the blocks, however, there are too many peaks with a creation time of 20 minutes (which is already the double) until 40 minutes, up to even 80 minutes per block. When a block takes 80 minutes to be mined, it will affect the visibility of all the transactions in that block, mining the consistency of the system, since, in an average scenario a sender expects that its transaction is visible after $\sim$ 8-15 minutes.

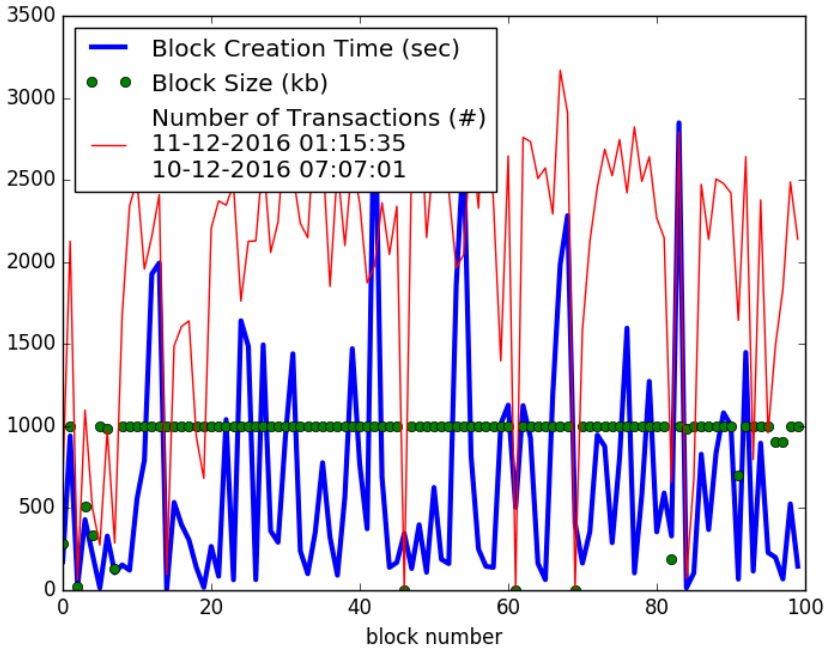


Figure 4.3: Relation between block creation, block size and number of transaction in a block. Measurement done on Bitcoin blockchain from 10^{th} December 2016 until 11^{th} December 2016.

4.4 Bandwidth

As explained in Chapter 1.3, the bandwidth represents the speed, in MB/s, for a transaction to be visible and accepted (write bandwidth), and the speed, always in MB/s, that a node has while fetching this block (read bandwidth). The Figure 4.4 shows that the read bandwidth while monitoring the blockchain for more than 2 months, has an average speed of $\sim$ 1 MB/s and it doesn't go faster than 9 MB/s. This makes the read bandwidth quite slow, not allowing us to fetch as many blocks as we want per time. An average block fetching time is $\sim$ 1 second, having then a total computation time, while fetching 2000 blocks, of $\sim$ 40 minutes. Furthermore, some API errors like "Connection reset by peer" or "Maximum concurrent requests for this endpoint reached" occurred. That leads our system development to allow the block retrieval with small number of blocks, and then append them in a text file, as explained in Chapter 3.

In Figure 4.5 is represented the comparison between the block creation time, Figure 4.5b, the average visibility time for transactions in each block, Figure 4.5a, and the the transaction visibility provided from the Bitcoin website, Figure 4.5c. The comparison was made using exactly the same blocks, retrieved from

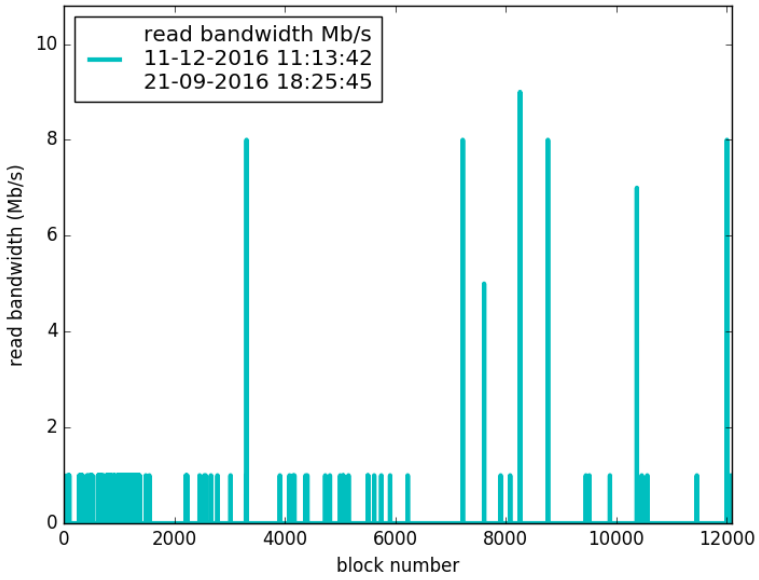
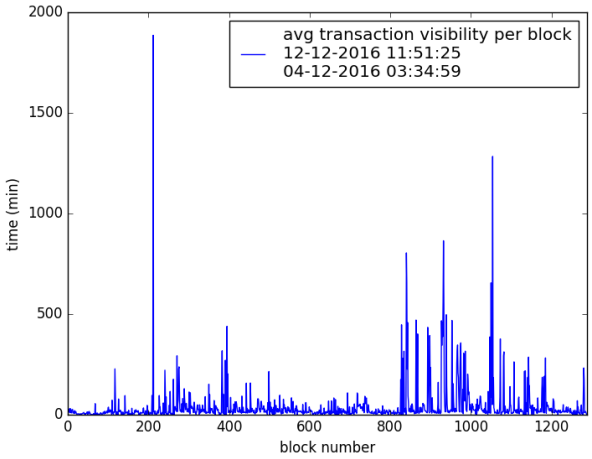


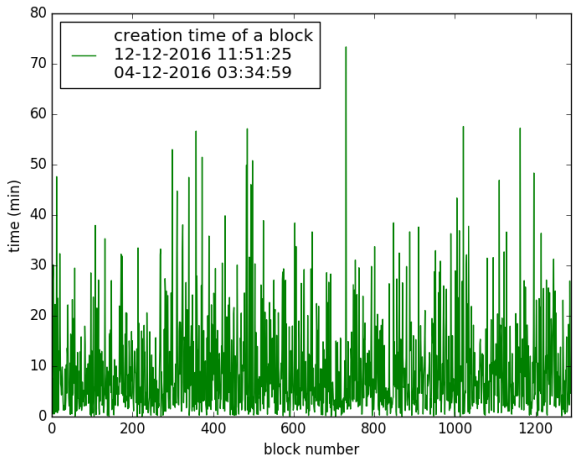
Figure 4.4: Read bandwidth of the Bitcoin blockchain measured according to the size of the block fetched and the time taken to fetch that block. Measurement done Measurement done from 21^{st} September 2016 until 11^{th} December 2016.

4^{th} until 12^{th} December 2016. The first thing to notice is that, despite the block creation time is never higher than 80 minutes, is that there are some transactions that take more than 1 day to be visible. This because miners gives the priority to transactions that are willing to pay an higher fee. A lot of transactions are not paying any fee, so they will not be included immediately to the next block creation. This thing was not very well explained anywhere and it was clear only after this data analysis.

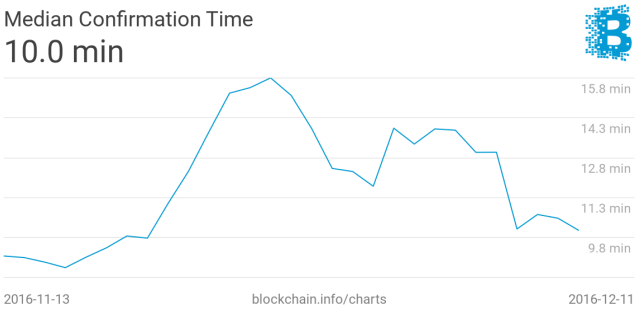
The plot from `blockchain.info` in Figure 4.5c shows that, in one month, the average confirmation time is $\sim$ 10 minutes, with a peak of 16. Considering though a finer granularity, in about a week of transactions, as shows Figure 4.5a, there is yes a coarse median of $\sim$ 10-20 minutes, but there are a lot 40 minutes peaks, some peaks of 300 and 400 minutes and even others of 1000 and 1900 minutes. This data might be relevant for a big company who decides to invest a lot in Bitcoin, then they need to know that a transaction could take even $100x$ time than what is showed in the Bitcoin website.



(a) Average time for a transaction to be visible in the public ledger since its first creation request. Measurement done between 4^{th} and 12^{th} December 2016 on the Bitcoin blockchain.



(b) The block's creation time in the Bitcoin blockchain. Every block is created each m minutes. Data from 4^{th} until 12^{th} December 2016 are considered for this test.



(c) Median confirmation time for a transaction. Data from the official Bitcoin website `blockchain.info` [2].

Figure 4.5: Comparison between block size and the average time for a transaction to be visible in the public ledger.

4.5 Block Fee

In this thesis we aim to find a relation between the fee paid to the miner and the block creation time. Before, we saw that more a client is willing to pay for a transaction fee (T_g) more are the probability that its transaction is included immediately in the next block. Here we want to confirm the relation that exists between the creation time and the fee paid to the miner. In Figure 4.6 an analysis between the 21^{st} September 2016 and the 12^{th} December 2016 is done, retrieving 12100 blocks, and it is pretty clear that there is a certain relation between the series of data analyzed. This confirms that, more computational effort is put into a block creation and more fee is paid to the miner or pool of miners who solve the proof of work. Of course there are occasional exceptions, if a block contains a lot of transaction with T_g equal to zero, the fee paid to the miner will be lower if compared to the one given from a block containing only high priority transactions, which is extremely unfair. This is why current, decentralized digital currencies are trying to avoid transactions with T_g equal to zero by ignoring them. On the other hand the Figure 4.6 shows that there are only couple of blocks that take high computational time, $\sim$ 35-38 minutes, and get a low reward, almost 0 BTC.

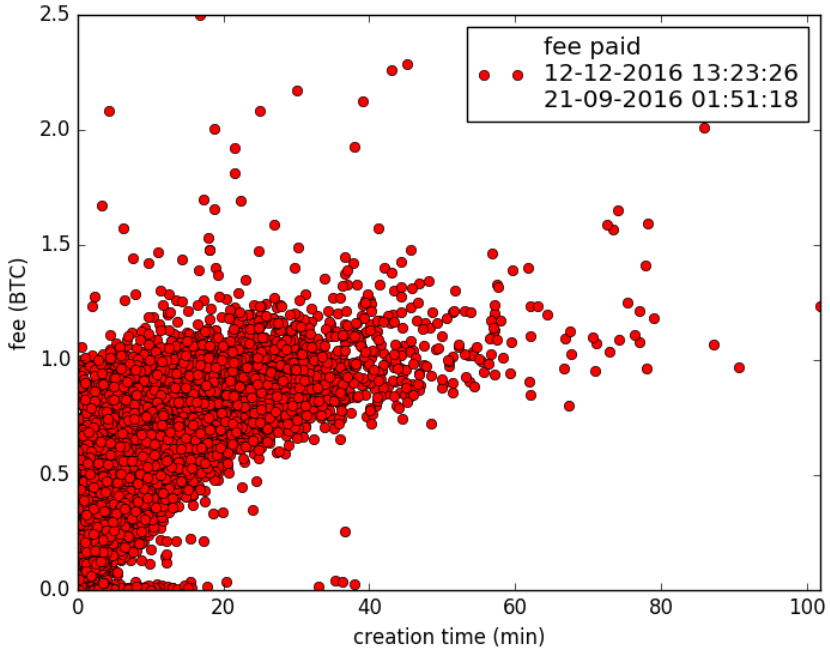


Figure 4.6: Relation between the fee paid to the miner and the block creation time. Measurement done on the Bitcoin blockchain between the 21^{st} September 2016 and the 12^{th} December 2016.

With this observation, it becomes possible to generate a model that, in future implementation, could help blocks to get the expected reward for mining. The model created is showed in Chapter 4.6 and it is generated with the polynomial interpolation of the data in Figure 4.6.

4.6 Models

Next, we generate a model that predict the future growth of the blockchain and help nodes to smartly use their bandwidth. To get the model for the blockchain growth, data are collected and then a statistical regression is made on them. The function that represents the blockchain growth is obtained thanks to the *NumPy* libraries [9]. To find the model that more fits our data, polynomial interpolation was applied [16]. An example of the code regarding it is found in Appendix B.16.

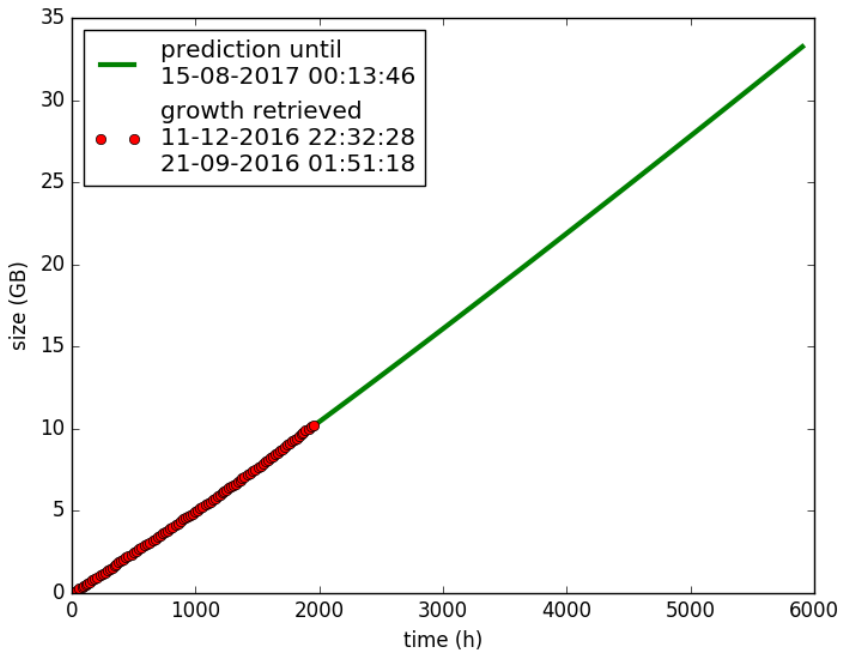


Figure 4.7: Regression of the growth of the blockchain with a prediction model. Measurement done on the Bitcoin blockchain between the 21^{st} September 2016 and the 11^{th} December 2016.

Considering the blockchain growth, a prediction of $3x$ time in the future has been evaluated. If the evaluation is done considering 1000 blocks, with a total

time of 150 hours, then the prediction will be on how much the blockchain will grow in the following 300 hours. The Figure 4.7 shows that the blockchain, as expected, has an almost linear growth. It has though a very little quadratic coefficient due to the slightly increment of the block size among the years. The function evaluated using the polynomial interpolation for the blockchain growth is the following:

$$f_g(x) = \frac{9}{10^8}x^2 + \frac{5}{10^3}x \tag{4.1}$$

According to the prediction in Figure 4.7, analyzing data from 21^{st} September until 11^{th} December 2016, the blockchain will grow of $\sim$ 28 GB by August 2017. To verify the veracity of this prediction, we applied this method from September 2016 to December 2016 so that we have the real growth to compare. Even though the data used were not so many (2000 blocks for 1 month forecast compared to the 12200 blocks used in the Figure 4.7), they were accurate in the prediction. That model has told us that by the 14^{th} of December 2016 the blockchain was supposed to grow of $9,5$ GB, when only data from September 2016 were evaluated. The blockchain has grown with about 10 GB instead, but still, considering the few data analyzed, the model generated is accurate and reliable. More data are used for the prediction, more accurate will be the function generated.

The Figure 4.8 shows that there is a relation between the fee paid and the creation time of a block. This relation seems to be quadratic/logarithmic. The regression is found using NumPy libraries [9] and the function is obtained with the polynomial interpolation. More data are analyzed, more the function generated is accurate. The one showed below in (4.2) is created by collecting data from 21^{st} September 2016 until 12^{th} December 2016.

$$f_{Bg}(x) = \left\{ -\frac{1}{10^4}x^2 + \frac{3}{10^2}x + 0,3 \quad \text{with } x < 100 \right. \tag{4.2}$$

Note that this function is valid if the creation time, x, is lower than 100 minutes. This model could be useful for a miner which expects a certain fee after mining for n minutes, to see if the fee they received is below or above the function model. In that way for the next mining processes a block will be in debit or credit and he will consider only transaction with an higher T_g, or transaction with a lower T_g according of how much is its credit/debit.

The last model generated is showed in Figure 4.9. It represents the average approval time for the transactions in a block, compared to the fee paid (B_g) divided by the number of transactions in that particular block. Then for each block B we calculated the average T_p, $\overline{T_p}$, as:

$$\overline{T_p} = \frac{B_g}{|B_t|} \tag{4.3}$$

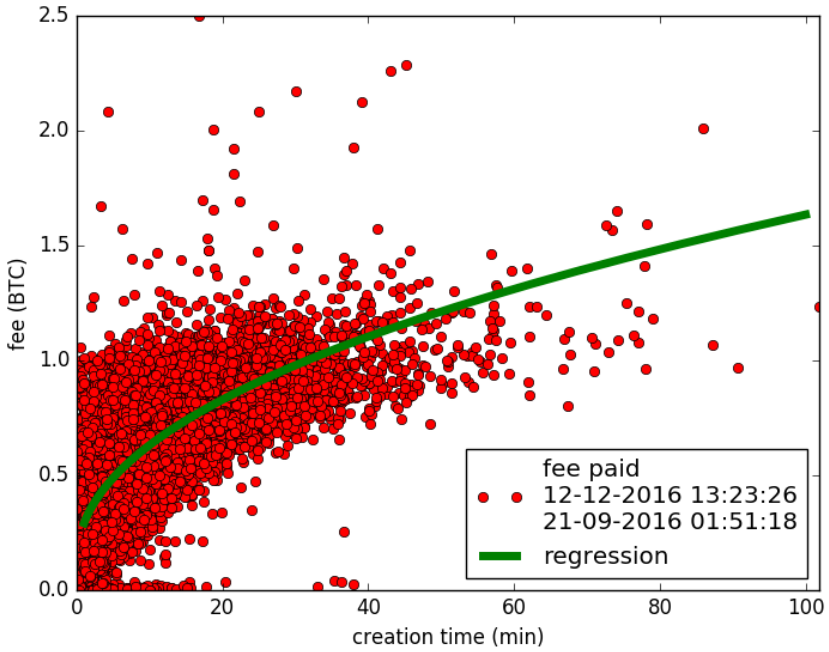


Figure 4.8: Regression of the relation between the fee paid to the miner and the block creation time. Measurement on the Bitcoin blockchain between the 21^{st} September 2016 and the 12^{th} December 2016.

where $|B_t|$ is the number of transactions approved from the block B. The plot in Figure 4.9 shows that if a transaction pays from 0 to 0.001 BTC, then the visibility would be almost random. However if their T_g is increased from 0.002 and 0.006 BTC their visibility will be seldom above 35 minutes and most likely between 5 and 15 minutes. The function generated with the polynomial interpolation is the following:

$$f_t(x) = \left\{ \frac{1}{10^8}x^2 - 5000x + 35 \quad \text{with } x < 0.007 \right. \tag{4.4}$$

The function showed in (4.4) is important for a transaction that needs a certain bandwidth, in a way that it knows, approximately, how much T_g should be willing to pay according to get a positive bandwidth.

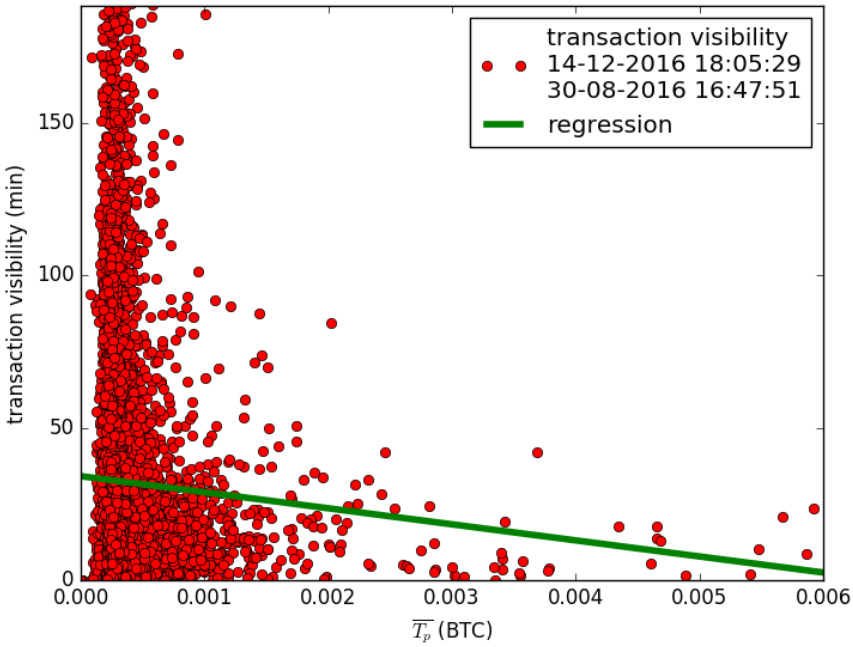


Figure 4.9: Regression of the relation between the fee paid to the miner and the transaction visibility time. Measurement on the Bitcoin blockchain between the 30^{th} August 2016 and the 14^{th} December 2016.

5 Conclusions

This chapter talks about the future implementation of the analytics system, what are the most relevant data to analyze in the future and how they could be implemented and evaluated. These considerations are made after profiling the system. The chapter discusses also about the model generated, showed in Chapter 4.6, their accuracy and their reliability. In Chapter 5.2, we show how the system could change following some design pattern solution.

5.1 Discussion

We are satisfied about the models generated, showed in Chapter 4.6. The blockchain growth prediction turned out to be accurate even with couple of months of analysis, and the more data are collected the more precise it will be. Since when the first block was appended, in 2009, the block size kept increasing. Even if this size changed only from 200 kB to 1 MB, it is still enough to have a quadratic growth from 2009 until 2016, while Bitcoin said it would have been linear.

Despite that we found a relation between the fee and the block creation time, the creation of a block is still a random process, determined from the proof of work, and paying so much fee (T_g) will not guarantee an extremely fast execution. Of course your transaction will be considered as "high priority" and included in the block that is being mined at the moment, but still the bandwidth

is limited from the mining difficulty, the random process of hash generation and from the computational power of the miners. However, thanks to the plot showed in Figure 4.9 we can conclude that if the T_g of a single transaction is higher than 0.003 BTC, then its visibility in the ledger of data will not be more that 50 minutes.

5.2 Future Implementation

Before of thinking about the future implementation, an analysis on the current system has to be done. The blockchain analytics system was evaluated using a line profiler for Python [7]. The results, showed in Figure 5.1, are that the biggest computation in matter of time is the retrieval of the blocks. The API calls with blockexplorer methods take indeed $\sim$ 98% of the time retrieving 500 blocks, while the writing part and the plotting part together take $\sim$ 1.4% of the total time. This is why the fetching and the analysis part are now separated. This separation will allow in the future to analyze a bigger portion of the blockchain, to manipulate more data and to get much more information from it. This also allows to do the analysis in a much faster way, from 17, 5 minutes, fetching 500 blocks, it could take only $\sim$ 2 seconds if the file containing the blockchain is already generated.

In future implementation more data from each block could be considered, such as the correctness of every block created. Correctness for each block is a value obtained putting all the most important informations of a block in a multidimensional vector, in the following way:

$$C = \begin{bmatrix} B_{sz} \\ B_g \\ B_s \\ B_t \end{bmatrix} \tag{5.1}$$

considering for each block the size (B_{sz}), the fee (B_g), the epoch, transformed in creation time (B_s) and the number of transactions in this block (B_t). The correctness of a block then is dependent from the range of these values and this range can change according to the level of correctness desired. For example we could have:

$$C_B = \begin{cases} 300 < B_{sz} \leqslant 1000, & \text{bytes} \\ 0.5 < B_g < 1.5, & \text{BTC} \\ 6 < B_s < 12, & \text{minutes} \\ 500 < B_t < 1500, & \text{transactions} \end{cases} \tag{5.2}$$

Which means that a block is correct if the correctness for this block, C_B, respects these values. In that way is possible to calculate how many blocks among the

```
Line Hits %Time  Line Contents
=================================================

368   # ----- PROGRESS BAR -----
369   500   0.4      sleep(0.01)
370   500   0.0      index_progress_bar += 1
371   500   0.0      printProgress(index_progress_bar,
   number_of_blocks, prefix='Saving␣Blockchain:',
   suffix='Complete', barLength=50)

389   500 0.4        transactions = current_block.
   transactions
# ------------------------------------------------
394   500 98.3       prev_block = blockexplorer.
   get_block(current_block.previous_block)
# ------------------------------------------------
402 # writing all the data retrieved in the file
403   1    0.4       write_blockchain(hash_list,
   epoch_list, creation_time_list, size_list,
   fee_list, height_list, bandwidth_list,
   list_transactions, append_end)

406   1    0.1       print blockchain_info()
```

Figure 5.1: Profiling results while executing the system retrieving 500 blocks

retrieved ones can be considered correct. It will be also possible to add the percentage of each correct/incorrect field, to find any possible relation between those and to get information about whether a block is incorrect, why that happened. It is already implemented, but not tested yet, the relation between the average transaction time and the fee paid to mine a block. Furthermore, following the design pattern principles, refactoring on the code must be done since the method *write_blockchain()* contains some *duplicate code* about writing on the file. The treatment to use might be *extract method* on the *file.write*.

5.3 Comments

A relevant aspect to note is that there are some blocks that take approximately 80 minutes to be created, when the average time is supposed to be ~ 8 min. This affects also the transaction visibility but it is also a side effect of the difficulty and proof of work. Furthermore, the growth of the blockchain is not ~ 1 MB

per hour as claimed from Bitcoin in 2008 but, at the moment, $\sim$ 5 MB each hour, growing of $\sim$ 2 GB in less than 12 days. In conclusion, we demonstrate our thesis and found a relation between the fee paid to a miner and the block creation time. We generate a function which describes this relation and it can be used in future implementations, allowing miners to be fairly rewarded. A block can arbitrary decides whether ignore a transaction that is not willing to pay enough fee (T_g). Plus, we defined a model for growth prediction and we tested its accuracy. Finally, a portion of the blockchain is saved locally and more analysis on it are possible in the future.

References

[1] Bitcoin api's, api-v1-client-python. https://github.com/blockchain/api-v1-client-python.

[2] Bitocoin's blockchain website. https://blockchain.info.

[3] Ethereum wiki/patricia tree. https://github.com/ethereum/wiki/wiki/Patricia-Tree.

[4] Ethereum's blockchain analysis website. https://etherscan.io.

[5] Ethereum's blockchain website. https://etherchain.org/.

[6] Ethereum's white paper. https://github.com/ethereum/wiki/wiki/White-Paper.

[7] Line profiler and kernprof – profile python applications. https://github.com/rkern/line_profiler.

[8] Matplotlib for data plotting. https://matplotlib.org.

[9] Numpy libraries manual – scipy. https://docs.scipy.org/doc/numpy/index.html.

[10] Ethereum foundation - the solidity contract-oriented programming language. Technical report, https://solidity.readthedocs.io/en/develop/, 2014.

[11] Bitcoin mining process. http://bitcoinminer.com/, 2015.

[12] Bitcoin website – mining. https://www.bitcoinmining.com, 2016.

[13] Ethereum project – website. https://www.ethereum.org, 2016.

[14] tradeblock.com – analysis on the blockchain. https://tradeblock.com,

2016.

[15] Alfred Aho and Jeffrey Ullman. *Foundations of Computer Science*, chapter 10: Patterns, Automata, and Regular Expressions. W. H. Freeman, 1992.

[16] Kendall E. Atkinson. *An Introduction to Numerical Analysis (Second edition)*. John Wiley & Sons, 1988.

[17] Adam Back. Hashcash: A denial of service counter-measure. Technical report, 2002.

[18] Paul Baran. *On Distributed Communications: Introduction to Distributed Communication Networks*. The Rand Corporation, 1964.

[19] Georg Becker. *Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis*. 2008.

[20] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. Technical report, Department of Computer Science University of Maryland, College Park, Initiative for Cryptocurrencies and Contracts (IC3), Department of Computer Science Cornell University, 2015.

[21] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. Technical report, IBM Research Division, Almaden Research Center, 1992.

[22] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. Efficient middleware for byzantine fault tolerant database replication. Technical report, Universidade Nova de Lisboa - Portugal, Kaiserslautern and Saarbrücken - Germany, 2011.

[23] Håvard D Johansen, Robbert van Renesse, Ymir Vigfusson, and Dag Johansen. Fireflies: A secure and scalable membership and gossip service. *ACM Transactions on Computer Systems (TOCS)*, 33(2):5:1–5:32, May 2015.

[24] Håvard D. Johansen, Robbert Van Renesse, Ymir Vigfusson, and Dag Johansen. Fireflies: Asecure and scalable membership and gossip service. Technical report, UIT The Arctic University of Norway and Cornell University and Emory University and Reykjavik University, 2015.

[25] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[26] Craig Larman. *Applying UML and patterns*. Prentice Hall, 1998.

[27] Aldelir Fernando Luiz, Lau Cheuk Lung, and Miguel Correia. Mitra: Byzantine fault-tolerant middleware for transaction processing on replicated databases. Technical report, Federal University of Santa Catarina – Brazil, University of Lisboa – IST INESC-ID – Portugal, 2014.

[28] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. Technical report, University of Singapore, 2016.

[29] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. Technical report, University of Singapore, 2015.

[30] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, www.cryptovest.co.uk, 2008.

[31] Chaitya B. Shah and Drashti R. Panchal. Secured hash algorithm-1: Review paper. Technical report, Indus Institute of Technology and Engineering, Gujarat Technological University, 2014.

[32] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1999.

[33] Melanie Swan. *Blockchain: Blueprint for a New Economy*. O'Reilly Media, 2015.

[34] Jeffrey Ullman and John Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[35] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. Technical report, MIT Computer Science and Artificial Intelligence Lab, Cambridge MA USA, 2007.

[36] Dr. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Technical report, 2014.

A

Terminology

RLP: Stands for recursive length prefix. It is a serialization method for encoding arbitrary structured binary data (byte arrays).

KEC-256: Another serialization method generating a 256-bit hash.

full node: A full node in a decentralized digital currency peer-2-peer network, is a node that stores and processes the entirety of every block, storing locally the entire size of the blockchain.

light node: A light node in a decentralized digital currency peer-2-peer network, is a node that only stores the part of the blockchain it needs.

satoshi: Unit of the Bitcoin currency. 100,000,000 satoshi are 1 BTC (Bitcoin).

B

Listing

Listing B.1: Smart contract transaction code in Ethereum.

```
function transfer(address _to, uint256 _value) {
        /* Add and subtract new balances */
        balanceOf[msg.sender] -= _value;
        balanceOf[_to] += _value;
        }
```

Listing B.2: Example of a smart contract that rewards users who solve a computational puzzle [28].

```
contract Puzzle {
  address public owner ;
  bool public locked ;
  uint public reward ;
  bytes32 public diff ;
  bytes public solution ;

  function Puzzle () // constructor {
    owner = msg. sender ;
    reward = msg . value ;
    locked = false ;
    diff = bytes32 (11111); // pre - defined
       difficulty
    }

  function (){ // main code , runs at every
     invocation
    if ( msg. sender == owner ){ // update reward
    if ( locked )
    throw ;
    owner . send ( reward );
    reward = msg . value ;
    }
    else
      if ( msg . data . length > 0){ // submit a
         solution
      if ( locked ) throw ;
      if ( sha256 (msg. data ) < diff ){
          msg. sender . send ( reward ); // send
             reward
          solution = msg. data ;
          locked
          }}}}
```

Listing B.3: Application usage.

```
1  Usage: observ.py -t number
2    -h | --help     : usage
3    -i              : gives info of the blockchain in
       the file .txt
4    -t number       : add on top a number of blocks.
       The blocks retreived will be the most recent
       ones. If the blockchain growth more than the
       block requested do -u (update)
5    -e number       : append blocks at the end of the .
       txt file. Fetch older blocks starting from the
       last retrieved
6    -P              : plot all
7    -p start [end] : plot data in .txt file in a
       certain period of time, from start to end. If
       only start then consider from start to the end
       of the .txt file
8    -R              : plot the regression and the
       models that predict the blockchain
9    -r start [end] : plot the regression and the
       models in a certain period of time, from start
       to end. If only start then consider from start
       to the end of the .txt file
10   -u              : update the local blockchain to
       the last block created
```

Listing B.4: Block object represented in Python according to api-v1-client-python to retrieve data on the blockchain. The function get_block() will return an object of this type [1].

```
class Block:
  def __init__(self, b):
    self.hash = b['hash']
    self.version = b['ver']
    self.previous_block = b['prev_block']
    self.merkle_root = b['mrkl_root']
    self.time = b['time']
    self.bits = b['bits']
    self.fee = b['fee']
    self.nonce = b['nonce']
    self.n_tx = b['n_tx']
    self.size = b['size']
    self.block_index = b['block_index']
    self.main_chain = b['main_chain']
    self.height = b['height']
    self.received_time = b.get('received_time', b['
       time'])
    self.relayed_by = b.get('relayed_by')
    self.transactions = [Transaction(t) for t in b['
       tx']]
    for tx in self.transactions:
        tx.block_height = self.height
```

Listing B.5: Transaction object represented in Python according to api-v1-client-python to retrieve data on the blockchain. The function get_transaction() will return an object of this type [1].

```
class Transaction:
  def __init__(self, t):
    self.double_spend = t.get('double_spend', False)
    self.block_height = t.get('block_height')
    self.time = t['time']
    self.relayed_by = t['relayed_by']
    self.hash = t['hash']
    self.tx_index = t['tx_index']
    self.version = t['ver']
    self.size = t['size']
    self.inputs = [Input(i) for i in t['inputs']]
    self.outputs = [Output(o) for o in t['out']]

    if self.block_height is None:
       self.block_height = -1
```

Listing B.6: Json object returned from the method *get_block()* in the Bitcoin API class blockexplorer.py [1]

```
hash : str
version : int
previous_block : str
merkle_root : str
time : int
bits : int
fee : int
nonce int
n_tx : int
size : int
block_index : int
main_chain : bool
height : int
received_time : int
relayed_by : string
transactions : array of Transaction objects
```

Listing B.7: Structure of the latest block retrieved. The function get_latest_block() will return an object with this structure.

```
1  class LatestBlock:
2  def __init__(self, b):
3  self.hash = b['hash']
4  self.time = b['time']
5  self.block_index = b['block_index']
6  self.height = b['height']
7  self.tx_indexes = [i for i in b['txIndexes']]
```

Listing B.8: Collecting data starting from the last element in the blockchain.txt file.

```
1  earliest_hash = get_earliest_hash()
2  get_blockchain(n, earliest_hash)
3
4  def get_blockchain(n, hash = None):
5    [...]
6    if (hash):  # start the retrieval from the hash
7      append_end = True  # in that way the
          write_blockchain method knows that has to
          append blocks and not write them at the
          beginning
8      last_block = blockexplorer.get_block(hash)
9    [...]
10
11 def get_earliest_hash():
12   hash_list = get_list_from_file("hash") # method to
        collect data from blockchain.txt file having
       as attribute "hash"
13   length = len(hash_list)
14   earliest_hash = hash_list[length - 1]
15   return earliest_hash
```

Listing B.9: Calling `blockchain.info` through python API and retrieving part of the blockchain.

```
1 from blockchain import blockexplorer
2 # get the last block
3 last_block = blockexplorer.get_latest_block()
4 hash_last_block = last_block.hash
5
6 # current block now is the last block
7 current_block = blockexplorer.get_block(
    hash_last_block)
```

Listing B.10: How read bandwidth is calculated, using the function *datetime.now*() before and after the API call.

```
1  start_time = datetime.datetime.now() # ---------
2  current_block = blockexplorer.get_block(
     current_block.previous_block)
3  end_time = datetime.datetime.now() # ---------
4  time_to_fetch = end_time - start_time
5  time_in_seconds = get_time_in_seconds(time_to_fetch)
6
7  #latency
8  fetch_time_list.append(time_in_seconds)
9
10 # calculate Bandwidth with MB/s
11 block_size = float(current_block.size) / 1000000
12 bandwidth = block_size / time_in_seconds
13 bandwidth_list.append(bandwidth)
```

Listing B.11: Function that get the average write bandwidth of the block, calculating the time for each transaction to be visible in the public ledger of data.

```
def get_avg_transaction_time(block):
  # take transactions the block
  transactions = block.transactions

  # get block time -- when it is visible in the
     blockchain, so when it was created
  block_time = block.time

  # list of the creation time for all the
     transaction in the block
  transactions_time_list = []

  # list of the time that each transaction needs
     before being visible in the blockchain
  time_to_be_visible = []

  for t in transactions:
    transactions_time_list.append(float(t.time))

  for t_time in transactions_time_list:
    time_to_be_visible.append(float(block_time -
       t_time))

  average_per_block = sum(time_to_be_visible) / len(
     time_to_be_visible)
  return average_per_block
```

Listing B.12: How the file.write is performed in the blockchain analytics system. Differences with add and append.

```
if(append):
  for i in range(n):
    file.write("block_informations")

else: # add on top
  hash_list_in_file = get_list_from_file("hash")
  first_hash = hash_list_in_file[0]
  elements = len(hash_list_in_file)
  last_hash = hash_list_in_file[elements-1]
  met_first = False
  with io.FileIO(file_name, "a+") as file:
    file.seek(0)  # place at the beginning of the
       file
    existing_lines = file.readlines()  # read the
       already existing lines
    file.seek(0)
    file.truncate()  # delete all the file
    file.seek(0)
    i = 0
    while (i < n):
      if (first_hash == hash[i]):
        met_first = True
      while((met_first == False) and (i < n)):
        # append on top
        file.write("block_informations")
        i = i + 1
        if ((i < n) and (first_hash == hash[i])):
          met_first = True
      # when the block retrieved meets the one
         alread in the blockchain write the old file
      file.writelines(existing_lines)
```

Listing B.13: Changing all the values inside a Python list in one code line.

```
# size_list from byte to MB
size_list[:] = [x / 1000000 for x in size_list]
# time_list from seconds in minutes
time_list[:] = [x / 60 for x in time_list]
```

Listing B.14: Method that allows, given an attribute present in the blockchain.txt file, to create a list containing informations only about this quality using *regular expressions*.

```
1  hash_list = get_list_from_file("hash")
2
3  def get_list_from_file(attribute):
4    list_to_return = []
5    if (os.path.isfile("blockchain.txt")):
6      # open the file and read in it --
          blockchain_file
7      with open("blockchain.txt", "r") as
          blockchain_file:
8        for line in blockchain_file:
9        # regular expression that puts in a list the
            line just read: eg. ['hash', '<block_hash
            >']
10         list = re.findall(r"[\w']+", line)
11         # list[0] --> contains the attribute
12         # list[1] --> contains the value
13         if ((list) and (list[0] == attribute)):
14           list_to_return.append(list[1])
15   return list_to_return
```

Listing B.15: Generation of the growing size and time lists. Calculated following the Equations 3.4, 3.3.

```
def create_growing_time_list(time_list):
  reversed_time_list = time_list[::-1]
  time_to_append = 0
  previous_time = 0
  growing_time_list = []
  growing_time_list.append(previous_time)
  for time_el in reversed_time_list:
    time_to_append = (float(time_el) / (60 * 60)) +
       previous_time  # time in hours
    growing_time_list.append(time_to_append)
    previous_time = time_to_append
  return growing_time_list

def create_growing_size_list(size_list):
  reversed_size_list = size_list[::-1]
  growing_size_list = []
  value_to_append = 0
  size_back = 0
  growing_size_list.append(value_to_append)
  for size_el in reversed_size_list:
    value_to_append = size_el + size_back
    growing_size_list.append(value_to_append)
    size_back = value_to_append
  return growing_size_list
```

Listing B.16: Example of a polynomial interpolation of two lists using NumPy libraries.

```
import numpy as np

model = np.polyfit(list1, list2, deg) # polynomial
   interpolation between list1 and list2 with the
   degree of the return polynomial
# deg = 1 --> linear interpolation
# deg = 2 --> quadratic
# deg = 3 --> cubic
# ...
predicted = np.polyval(model, list1)
plt.plot(list1, predicted, 'b-', label="pol_interp")
```

Listing B.17: Check the status of the local blockhain, True if valid, False if not.

```
def check_blockchain():
  check = True
  list = get_list_from_file("height")
  number = int(list[0])
  length_list = len(list)
  for i in range(length_list):
    if (number != int(list[i])):
      check = False
    number = number - 1
  return check
```