UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

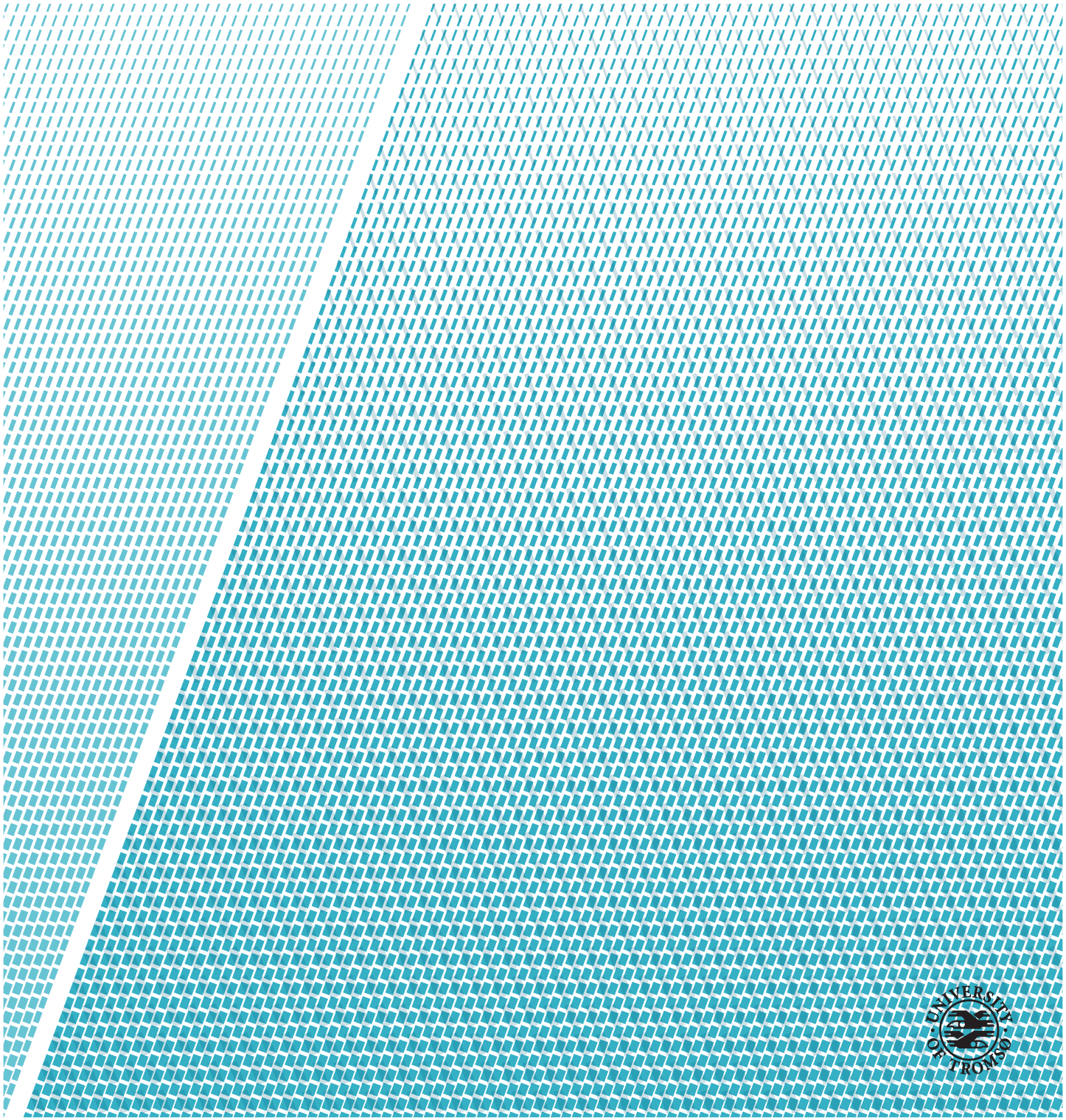
Faculty of Science and Technology
Department of Computer Science

Master Thesis title

subtitle

—

Enrico Tedeschi
Master Thesis in Computer Science



Abstract

Contents

List of Figures

My list of definitions

1 Introduction

1.1 Problem Statement

applications can control available bandwidth by paying mining fees.

1.2 Method / Context

1.3 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.

The Definition 1. 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.

The Definition 2. 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.

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 application programming interface (API) used and then how data are processed and manipulated.

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` [4] for Bitcoin, `etherscan.io` [7] and `etherchain.org` [8] 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

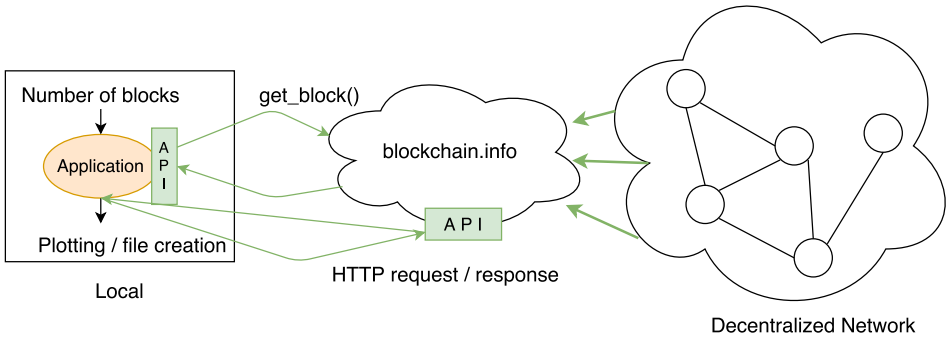


Figure 3.1: Architecture of the developed system for blockchain analysis.

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. 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).

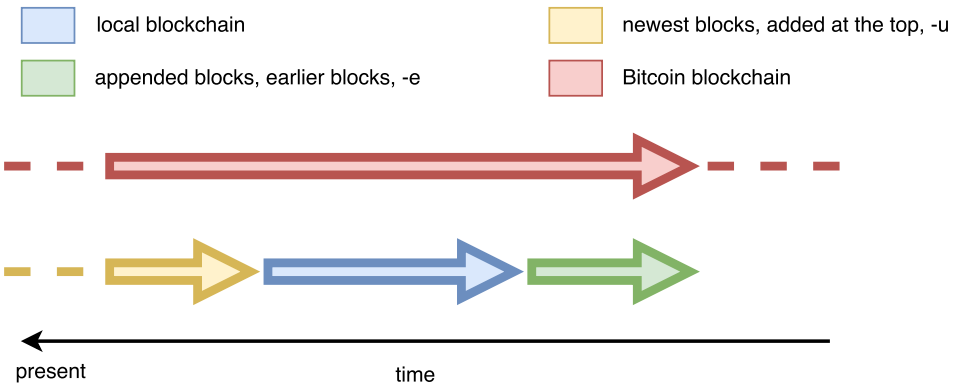


Figure 3.2: Scheme that shows how blocks are written in the local blockchain file.

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:

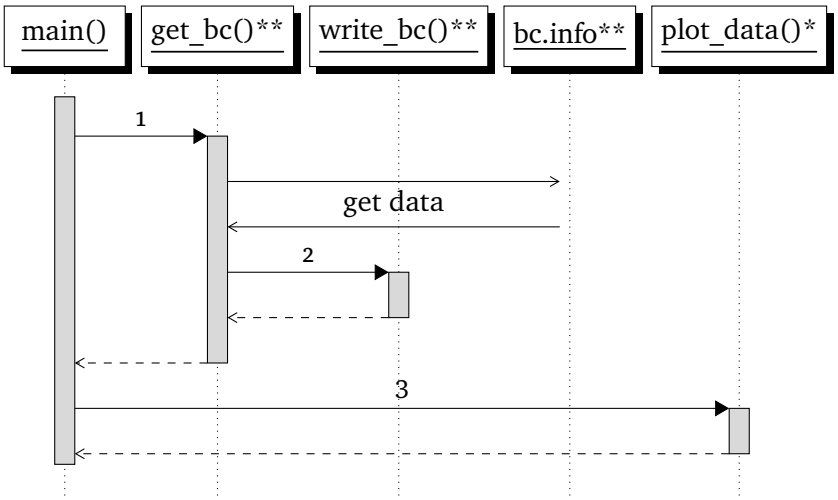
***get_blockchain(n, [hash])*:** method that retrieve the blocks from the Bitcoin blockchain and saves them in lists that will be the input of the write_blockchain() method. If the attribute hash exists then the retrieval starts from that hash and not from the latest block created in the Bitcoin blockchain. 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 [19] 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 [11]. 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.



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.

The *main()* method provides to call the others and if the general responsibility assignment software patterns (GRASP) principle is followed [31], 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 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 1.1, 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 [11]. 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 [9] 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.

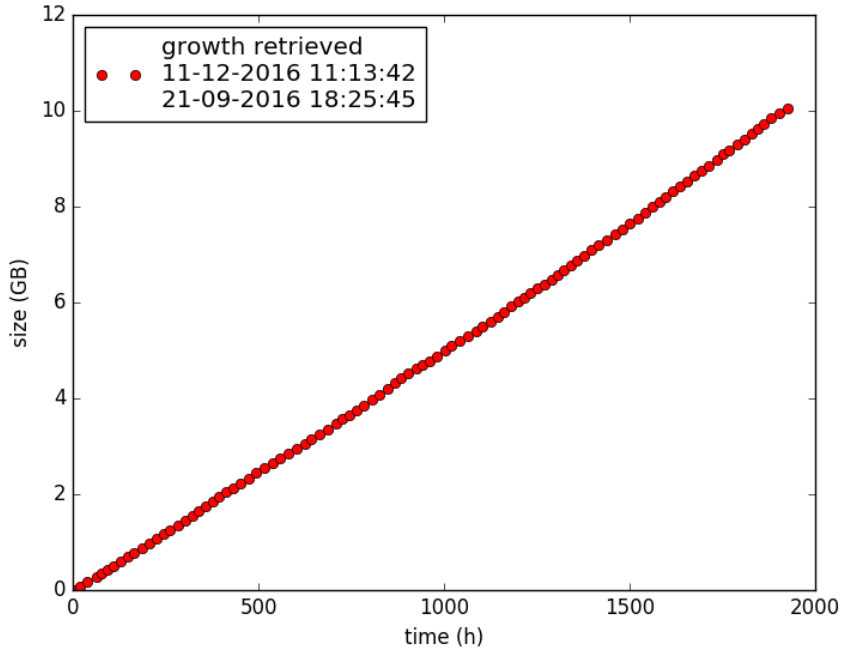
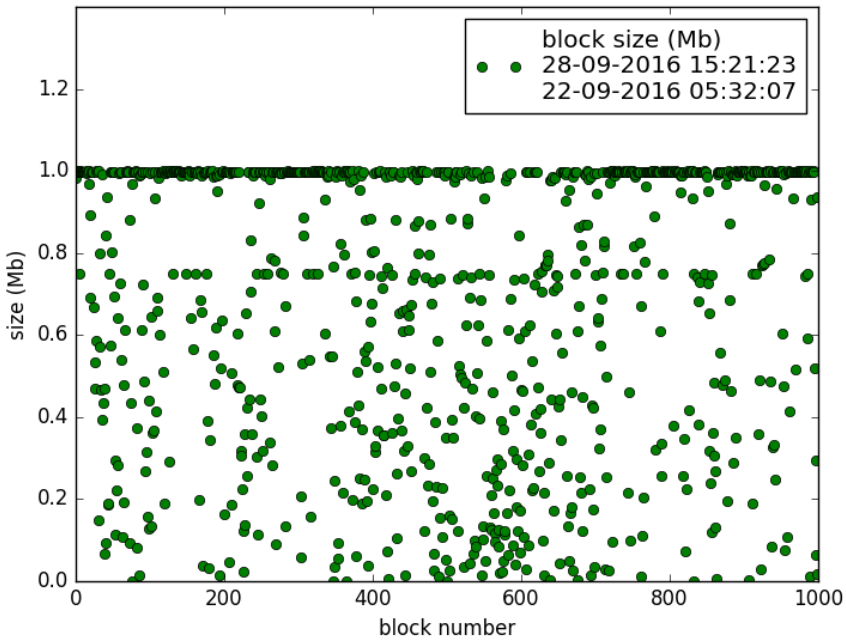
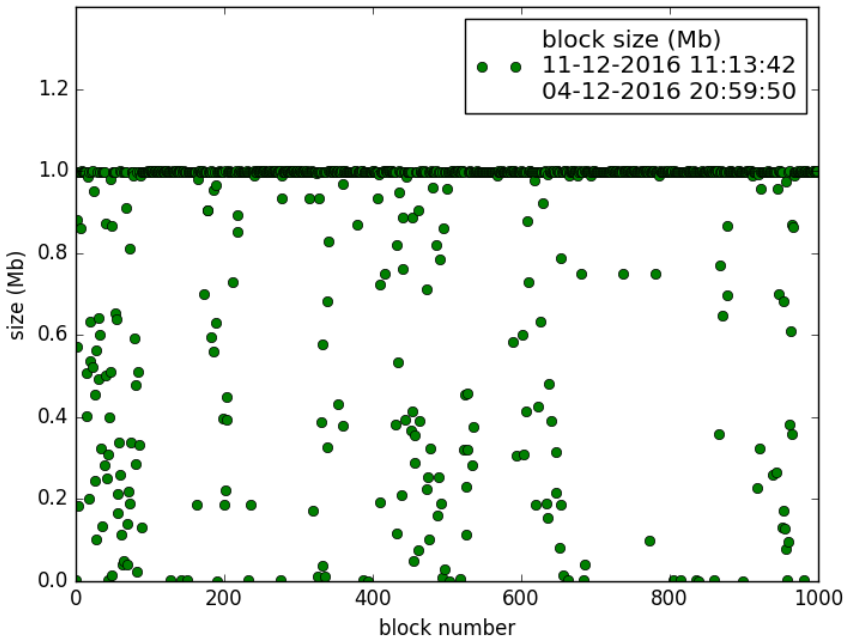


Figure 4.1: The Bitocin blockchain growth according to our analytics system. 12091 blocks are fetched between the 21^{st} September 2016 and the 11^{th} December 2016.

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 [9].



(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 **??** and according to the difficulty, which is constantly adjusted, a block should be mined every 10 minutes [16]. 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.

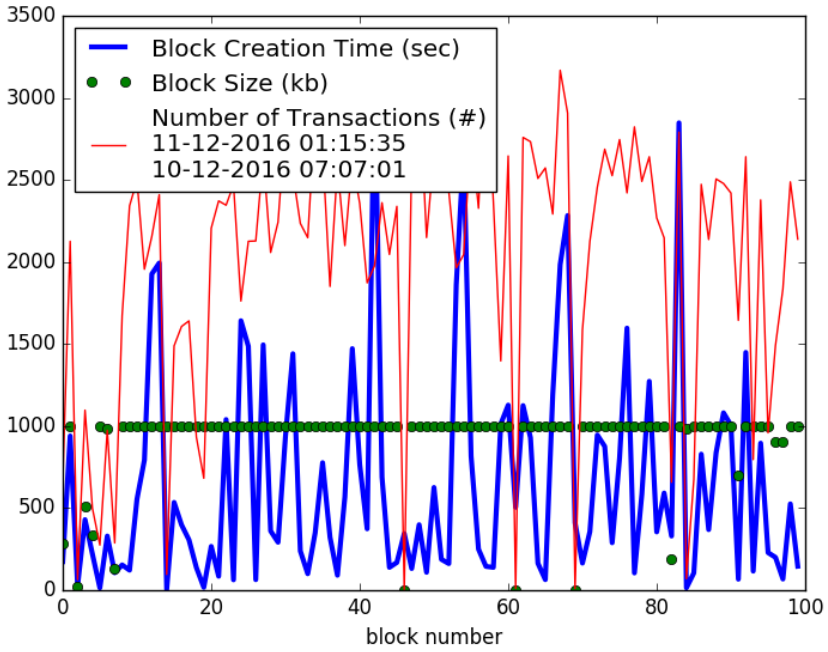


Figure 4.3: Relation between block creation, block size and number of transaction in a block. Measurement done on Bitcoin blockchain from 10^{th} December 2016 until 11^{th} December 2016.

4.4 Bandwidth

As explained in Chapter 1.2, 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

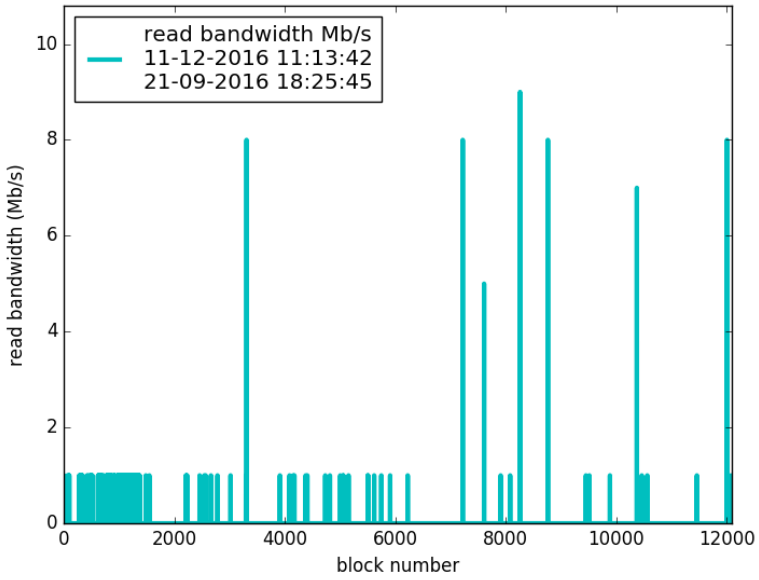
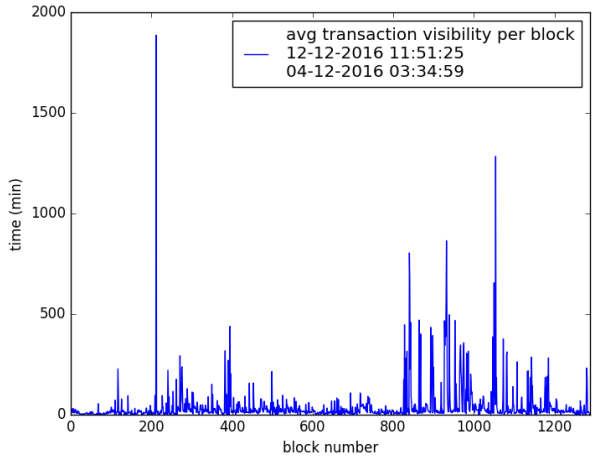


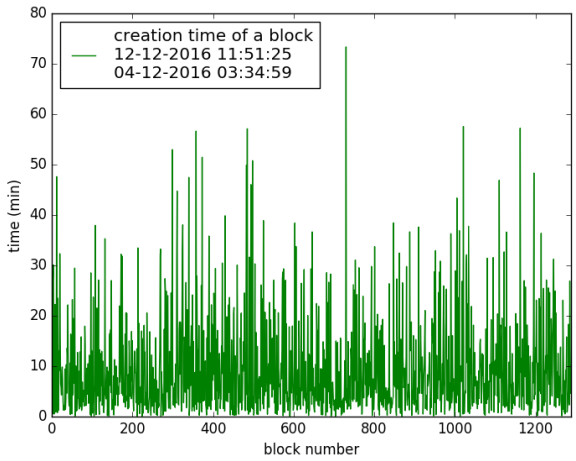
Figure 4.4: Read bandwidth of the Bitcoin blockchain measured according to the size of the block fetched and the time taken to fetch that block. Measurement done Measurement done from 21^{st} September 2016 until 11^{th} December 2016.

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.

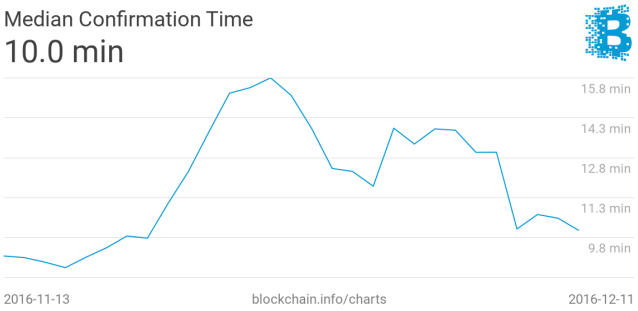
The plot from `blockchain.info` in Figure 4.5c shows that, in one month, the average confirmation time is $\sim$ 10 minutes, with a peak of 16. Considering though a finer granularity, in about a week of transactions, as shows Figure 4.5a, there is yes a coarse median of $\sim$ 10-20 minutes, but there 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` [4].

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 bitcoin currency (BTC).

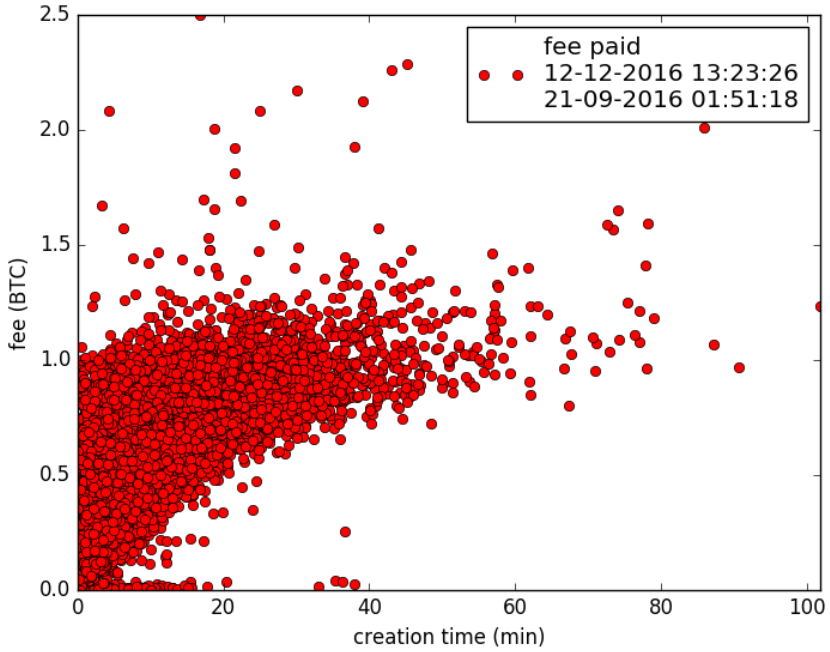


Figure 4.6: Relation between the fee paid to the miner and the block creation time. Measurement done on the Bitcoin blockchain between the 21^{st} September 2016 and the 12^{th} December 2016.

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 [13]. To find the model that more fits our data, polynomial interpolation was applied [28]. An example of the code regarding it is found in Appendix B.16.

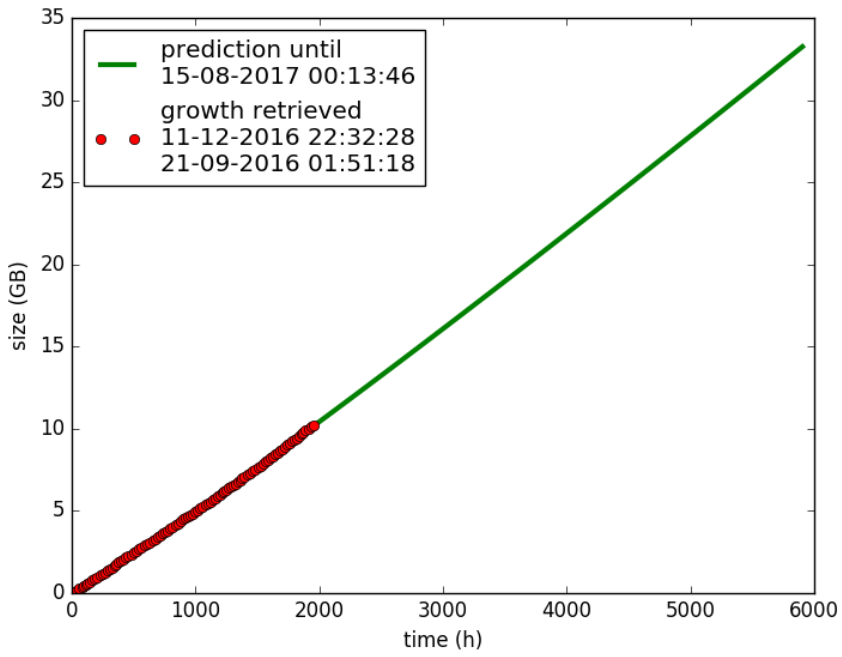


Figure 4.7: Regression of the growth of the blockchain with a prediction model. Measurement done on the Bitcoin blockchain between the 21^{st} September 2016 and the 11^{th} December 2016.

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 [13] 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:

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

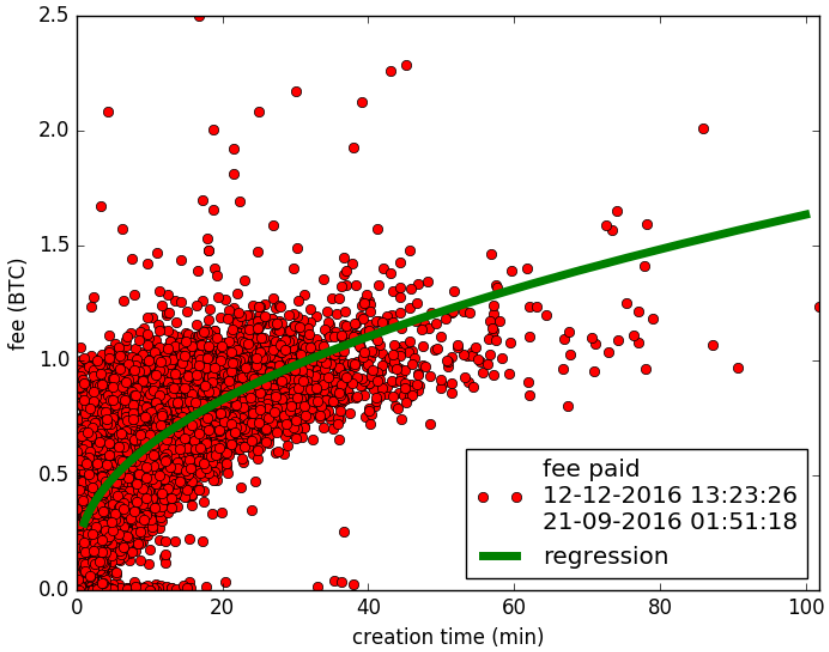


Figure 4.8: Regression of the relation between the fee paid to the miner and the block creation time. Measurement on the Bitcoin blockchain between the 21^{st} September 2016 and the 12^{th} December 2016.

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.

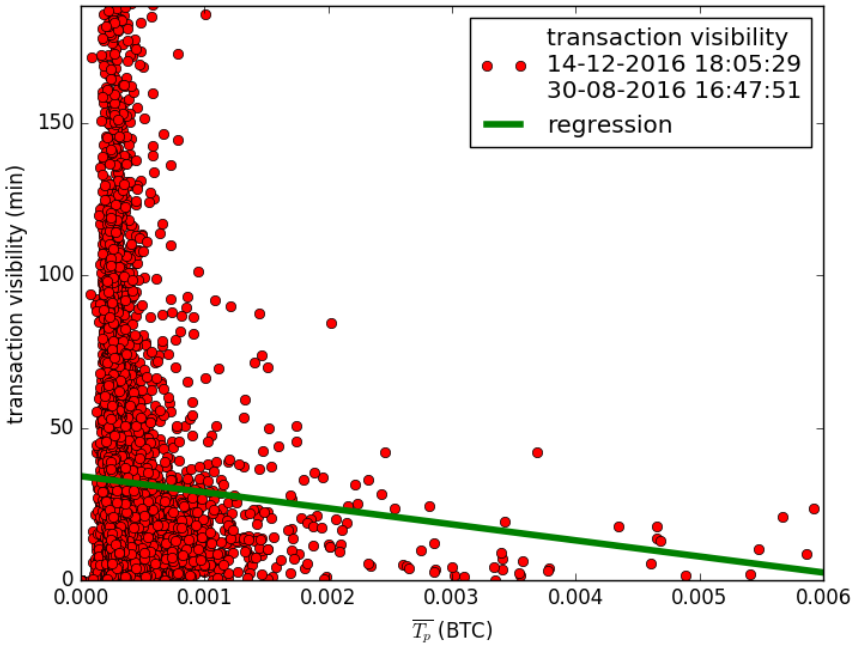


Figure 4.9: Regression of the relation between the fee paid to the miner and the transaction visibility time. Measurement on the Bitcoin blockchain between the 30^{th} August 2016 and the 14^{th} December 2016.

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 [10]. 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.

show bibliography [35], [41], [20], [25], [33], [24], [9], [14], [34], [37], [29], [30], [16], [27], [40], [32], [1], [5], [22], [6], [39], [21], [38], [4], [7], [8], [15], [18], [17], [31], [19], [11], [23], [26], [28], [12], [2], [3], [36].

References

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

[2] Bitcoin mining hashing rate. https://blockchain.info/charts/hash-rate.

[3] Bitcoin nodes. https://bitnodes.21.co/.

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

[5] Ethereum api's, pyethereum. https://github.com/ethereum/pyethereum.

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

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

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

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

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

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

[12] Mining hardware comparison. https://en.bitcoin.it/wiki/Mining_hardware_comparison.

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

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

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

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

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

[18] tradeblock.com – analysis on the blockchain. `https://tradeblock.com`, 2016.

[19] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*. Computer Science Press, Inc., New York, NY, USA, 1992.

[20] Adam Back. Hashcash - a denial of service counter-measure. Technical report, 2002.

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

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

[23] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. *On Scaling Decentralized Blockchains*, pages 106–125. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[24] 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*, pages 79–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[25] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 139–147, London, UK, UK, 1993. Springer-Verlag.

[26] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-ng: A scalable blockchain protocol. *CoRR*, abs/1510.02037, 2015.

[27] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. Efficient middleware

for byzantine fault tolerant database replication. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 107–122, New York, NY, USA, 2011. ACM.

[28] Begnaud Francis Hildebrand. *Introduction to Numerical Analysis: 2Nd Edition*. Dover Publications, Inc., New York, NY, USA, 1987.

[29] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[30] 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.

[31] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[32] Aldelir Fernando Luiz, Lau Cheuk Lung, and Miguel Correia. Mitra: Byzantine fault-tolerant middleware for transaction processing on replicated databases. *SIGMOD Rec.*, 43(1):32–38, May 2014.

[33] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[34] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, 2015. ACM.

[35] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008.

[36] Peter R. Rizun. A transaction fee market exists without a block size limit. Technical report, 2015.

[37] 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.

[38] William Stallings. *Cryptography and Network Security: Principles and*

Practice. Pearson Education, 3rd edition, 2002.

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

[40] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Samuel Madden. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In *ACM SOSP*, Stevenson, WA, October 2007.

[41] 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 [**?**].

```
1  contract Puzzle {
2    address public owner ;
3    bool public locked ;
4    uint public reward ;
5    bytes32 public diff ;
6    bytes public solution ;
7
8    function Puzzle () // constructor {
9      owner = msg. sender ;
10     reward = msg . value ;
11     locked = false ;
12     diff = bytes32 (11111); // pre - defined
          difficulty
13     }
14
15   function (){ // main code , runs at every
        invocation
16     if ( msg. sender == owner ){ // update reward
17     if ( locked )
18     throw ;
19     owner . send ( reward );
20     reward = msg . value ;
21     }
22     else
23       if ( msg . data . length > 0){ // submit a
            solution
24       if ( locked ) throw ;
25       if ( sha256 (msg. data ) < diff ){
26           msg. sender . send ( reward ); // send
               reward
27           solution = msg. data ;
28           locked
29           }}}}
```

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.

```
class LatestBlock:
def __init__(self, b):
self.hash = b['hash']
self.time = b['time']
self.block_index = b['block_index']
self.height = b['height']
self.tx_indexes = [i for i in b['txIndexes']]
```

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

```
earliest_hash = get_earliest_hash()
get_blockchain(n, earliest_hash)

def get_blockchain(n, hash = None):
  [...]
  if (hash):  # start the retrieval from the hash
    append_end = True  # in that way the
       write_blockchain method knows that has to
       append blocks and not write them at the
       beginning
    last_block = blockexplorer.get_block(hash)
  [...]

def get_earliest_hash():
  hash_list = get_list_from_file("hash") # method to
      collect data from blockchain.txt file having
     as attribute "hash"
  length = len(hash_list)
  earliest_hash = hash_list[length - 1]
  return earliest_hash
```

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

```
from blockchain import blockexplorer
# get the last block
last_block = blockexplorer.get_latest_block()
hash_last_block = last_block.hash

# current block now is the last block
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.

```
start_time = datetime.datetime.now() # ---------
current_block = blockexplorer.get_block(
   current_block.previous_block)
end_time = datetime.datetime.now() # ---------
time_to_fetch = end_time - start_time
time_in_seconds = get_time_in_seconds(time_to_fetch)

#latency
fetch_time_list.append(time_in_seconds)

# calculate Bandwidth with MB/s
block_size = float(current_block.size) / 1000000
bandwidth = block_size / time_in_seconds
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
```