BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS
ÚSTAV TELEKOMUNICACÍ

DEVELOPMENT OF A PEER-TO-PEER (P2P) SYSTEM BASED ON BLOCKCHAIN WITH THE AIM OF ENERGY-TRADE NETWORKS

TERM PROJECT
SEMESTRÁLNÍ PROJEKT

AUTHOR — **LUIS VICENTE ARNAO**
AUTOR PRÁCE

SUPERVISOR — **Ing. RADEK FUJDIAK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Abstract

The emergence of Artificial Intelligence (AI), Big Data and new Peer-to-Peer (P2P) networks based on Blockchain systems and technologies are reshaping our world. While the new technological revolution improves the quality of our life, new concerns are triggered. Based on Blockchain systems and their increasingly massive implementation in (mostly) testing systems, this work will try to provide as generic a solution as possible for the test of this kind of networks.

To do this, I have firstly searched lots of papers and a few courses to understand correctly the Blockchain technology (part of the DLTs, the Distributed Ledger Technologies). After this theoretical introduction, I tried a few incomplete projects in different languages to decide which do language I prefer. There are so many codes in C++, Python and Java but it is often mixed a few languages on the same project as it was said, there are incomplete codes.

Finally, the project developed is a Blockchain system realized in different modules so it can be easier to get the parts you want in your project starting from mine.

I really hope this project and the literature I recommend helps you to start in the Blockchain world if you finally decide so. Don't hesitate to ask me by Linked In or any preferred way about doubts.

Keywords

Blockchain, P2P, Peer-to-peer, DLT, Distributed Ledger Technologies, Energy, Electricity, Energy-trade, Electricity-trade, Network, Project code, Python language

Reference

VICENTE ARNAO, Luis. *Development of a peer-to-peer (P2P) system based on Blockchain with the aim of energy-trade networks.* Brno, 2020. Term project. Brno University of Technology, Faculty of Electrical Engineering and Communication. Supervisor Ing. Radek Fujdiak, Ph.D.

Development of a peer-to-peer (P2P) system based on Blockchain with the aim of energy-trade networks

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr.Radek Fujdiak. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Luis Vicente Arnao
June 29, 2020

Acknowledgements

I take this opportunity to express gratitude to all the people that supported me to carry on with my studies and help me to grow up. First, thank you to the unconditional love and support brought by my family (specially my mother, grandparents and uncles) so important in my life. Thanks a lot, to my friends and classmates for helping me to improve and learn every single day. I also place on record, my sense of gratitude to one and all the teachers and researcher who directly or indirectly have helped, taught and motivated me.

Contents

Chapter 1: Introduction

Satoshi Nakamoto published on November 2008 the White Paper [13] who originated the Bitcoin. After this milestone, Blockchain technologies and cryptocurrencies has been relentlessly developed due to its potential power.

First, everyone interested on this technology should understand that a Blockchain is simply a chain of blocks, a string of packets, that builds trust through some special cybersecurity attributes explained below. Due to this, the Blockchain technologies can save time and costs besides improve the security if the nodes can rule with these algorithms to be active part of the network.

One of the most important problems is that this hype around this topic pushed lots of people write codes and papers about it without making a clear difference between distributed ledger technologies (DLT), Blockchain and cryptocurrencies (In example, Bitcoin or Ethereum). Example of this are some really useful courses given by some of the most prestigious entities like Berkeley University [10] or the Linux Foundation [5] whereof I personally recommend the first one.

In this thesis, I am going to focus on the Blockchain to explain the components of mine (on the literature are some papers and two books [2] that explain it more deeply) and create a Blockchain (chain of blocks using DLT using P2P) network step by step. This network will be coded in Python 3 with the idea of trade electricity and to be implemented over a Raspberry Pi network. This means that the nodes on the network (computers or raspis) are clients of an electric system that generate/consume electricity.

Motivation and objectives

There are so many students (and researchers) like me interested on the Blockchain topic. This thesis should be an introduction to this deep world to every person interested on how it works and how to build a Blockchain network "for dummies".

Following this line, it is going to be expound theoretically the main parts of a Blockchain network to be able to explain the development of my project. After these two steps, every reader should be able to develop their own network if desired or at least, understand correctly this trendy technology.

In the introduction, it was told that this Blockchain networks bring some security attributes that can be the optimal option in some scenarios. These attributes are: immutability (once the network verifies or mines a block, this transaction is saved and nobody should be able to destroy it), transparent (every node or participant in the network is visible to the rest of them and can be consulted so every node has access to all the transaction data), secure (the confidence on the process of creation of new blocks comes directly from the distributed algorithm used to it that makes impossible an unauthorized access to the Blockchain), consensus-based (enough relevant network participants must agree a transaction, the propagation of the transactions is paramount), flexible (Smart Contracts can

be written into the platform so the Blockchain network can evolve depending on the necessities) and of course, distributed (this distributed ledger technology propagates every incoming transaction among the nodes of the network).

Content description

This diploma thesis needs a correct structure to reflect correctly the main parts of this extensive work. This is the reason why I consider interesting to add this brief part to the thesis.

After an abstract and this introduction, on the Chapter 1 that tries to motivate the reader explaining the research problem and objectives, my methods to develop the project and my conclusion about it, it starts the first of the three parts: the prior analysis.

The prior analysis explains the theory needed to know about the main topics. These topics are the DLTs, the Blockchain networks and the Energy trading networks. This theory is located on the Chapter 2. At the same time, this chapter explains the math behind the Public key and Elliptic Curve algorithms to end with the current state of the art.

The second part, Chapter 3 is my project's development and implementation. This practical part is compound by the explanation of some important libraries and functions utilized (following the main idea of Python programming), the explanation of my code and the experimental validation with some screenshots I took to make it more clear.

The last but not the least is the conclusions summary and the proposed future projects. Due to the particularity of my project (the idea of a project to help people to develop projects starting from this one) it is important to add some project proposed in the Chapter 4.

Of course, this thesis counts with a Literature.

Chapter 2: Prior analysis

In this chapter, the main idea is to introduce the relative technologies I consider needed to know to understand the Blockchain and this project.

Starting with the DLTs or Distributed Ledger Technologies, it can be understood the idea of saving information without servers but using multiple locations or nodes. After this explanation, it can be explained the Blockchain networks and with both of this concepts, finally, it will be easy to understand the networks that nowadays are developed to trade energy between peers using Blockchain.

In addition, it is interesting to understand two cryptographic algorithms like the Public Key and the Elliptic Curve. After explaining these technologies and algorithms, it can be easily summarized analysis of the current state of the art.

To help the reader understand this ideas importance, it can be shown some examples to see how the Blockchain investigation and success applications are being built and see the importance of the technologies and protocols here explained.

1. Bitcoin: One of the most famous Blockchain applications and cryptocurrencies. Due to it is based on Blockchain network, it can be said that the Bitcoin network uses Peer-to-Peer connections between nodes, and besides, the information is saved distributed in multiple nodes (without servers), so it can be said that it is a Distributed Ledger Technology. Finally, the security system chosen is a version of Elliptic Curve Digital Signature Algorithm (ECDSA), in turn, using the Public Key Cryptography (most of the actual Blockchain Systems uses this Public Key Cryptography application).

2. Ethereum: This one is other really famous cryptocurrency. In this case, it is too used the Blockchain (part of the DLTs and P2P networks) and the Elliptic Curve cryptography.

3. My network: It is a Blockchain network (it counts on P2P and DLT network) too. The difference is that due to it is a educational example, the Public Key signature has been used instead of the Elliptic Curve signature (much more complex) as it will be explained below, due to both of this two systems have a security problem against quantum computers.

4. Alternative Blockchain networks: To avoid the Elliptic Curve and Public Key Cryptography, there are some alternative networks starting to investigate different cryptographic protocols like W-OTS (Winternitz one time signature). More information about this curious method in this link. It explains how it is tried to avoid the quantum computer power to break the cryptography protocol security.

   This networks are being developed and are just investigation networks and ideas so it can not be treated as a commercial network yet.

2.1 Distributed Ledger Technologies

The Distributed Ledger Technologies (DLTs) have conceptually the same origin as the Blockchain Technologies: distributed digital registers. First of all, Blockchains are part of the DLTs but all the DLTs are not Blockchain. The DLT focus their power on improve the transparency by making copies of some information in different parts so it is harder to hack or corrupt the information. The particularity of Blockchain is that this one, adds the packets of information in a chain (often) in a hash to improve the confidentiality and compression of this information. The hash, it just a summary of the information that independently on the amount of data you put as input, it will give you and output with the same number of bits. E.g.: SHA256 (Secure Hash Algorithm of 256 bits) will give an output of 256 bits independently if the input is an image or the book Don Quixote.

This DLTs are at the same time part of the Peer to Peer (P2P) technologies. If traditional networks make a successive amount of request to reach a server and ask for a resource, in example a website domain, in the Peer to Peer networks the users can send any kind of media without intermediary or central servers but a network of peers or users like it was done with the Spotify in their origins or the traditional Torrent system. Due to the huge number of variations on Blockchain (part of the Distributed Ledger Technologies) networks it must be done a differentiation between the three most extended identity management schemes on DLT.

U-port depends purely on the network. The system has the risk of unauthorized access to local authentication methods. The network provides a method to recover ownership of lost/compromised uPortID (it is important to have in mind if a user disconnects will lost his credentials. It is needed). This idea it is implemented on the Ethereum cryptocurrency. A different idea is Sovrin, who implements a complex Seal plus Certification plus Verification system that enable each user to control all his credentials.

In the other hand, ShoCard is a system that depends on a "Steward" or controller system to verify the credentials. This idea it is implemented on the Bitcoin cryptocurrency and its system is the traditional one on the Blockchain.

If interested on understand more deeply this part, it is recommended to take a look on these interesting papers about issues to fight against [12] and identity management schemes [3] in DLTs. It is too recommended as introduction to the topics treated in this chapter a few websites: [7], [8], [16] and [15] and the courses: [15] and [5]

One of this DLTs is the Blockchain networks due to it is said a particular type of DLT.

2.2 Blockchain networks

Blockchain is a safe technology and it is a currently well-established technology operating in example in the Bank of China as a valuation system. Despite saying tirelessly all the life that there is not absolutely safe technology, non-repudiation property and the public scrutiny makes this system really different to previous ones and the whole network would solve the issues that can try to spoil [9] a Blockchain system. Despite all this, there are some inherent risks on Blockchain architecture:

- A weakness in a network infrastructure can jeopardize the security of all the Blockchain-based application. e.g.: there are some "cheaters" in a network trying to destabilize the system creating fake information. This "cheaters" must be discovered, expelled and probably, safe logs about this intrusion like a black/grey list to avoid similar

dangers in a future. If not when these fakes spread more than the real information, our system will threat the correct information as a fake and will be ignored.

- In some systems, the communication must be encrypted to ensure that only the system knows who send some information and what does that information means (not even network operators can access to data). e.g.: if a "cheater" knows the identity of other node of our system, could try to impersonate it or even attack him to eliminate some information from our system (what opens the way to the next necessity).

- The problem of using hash codes when we have on the other side a large-scale quantum computer as potential future "cheater" like it is said on [4] (e.g.: A 1024b RSA, Rivest–Shamir–Adleman protocol, would need 2300 logical qbits what means less than 4 hours and the issue of a hash so utilized on the Blockchain systems, it is realized with 100 symbols that means 10010, approximately 266 that if we bear in mind the Grover algorithm can be solved in some seconds using 33 logical qbits. This can be clearly seen on the article [11]). It is interesting remember that this year 2019, IBM has already launched the first commercial quantic computer with 20qbits. An easy option would be increasing the number of bits that would increase the delay in calculus and would in a similar problem and this is one of the inherited problems that are currently not solved on this technology.

One of the most common techniques on Blockchain is the redundancy of information in different nodes to avoid losing or corrupting our system's information (this shows a trouble of Ethereum smart contracts when nodes are isolated and can't send confirmation). This is because of the potential of the DLTs to propagate the information in different nodes.

In conclusion, there are more than a hundred alternate Blockchain systems with different security guarantees because sometimes it is not needed to ensure privacy, or it is used a simpler hash to save some computing time. And the users must not lose their credentials.

To solve these issues, it can be considered some options when developing a Blockchain network:

- Multi-factor authentication for access to the Blockchain network due to the front-end applications risk. This is because "vulnerabilities are usually found in the front-end applications such as the wallets or in the smart contracts with which malicious codes can be built" as Paul Sin, Leader of Deloitte Asia Pacific Blockchain Lab from Deloitte China Consulting Partner said in a speech.

- Redundancy of the information in different users or even redundancy in different users in addition with storing parts of the information (useless without the rest of parts) instead of the whole packet. In this way, it is avoided that the store can interpret it and reduce the encryption.

- To use black/grey lists (a list with forbidden or potentially dangerous users) and some kind of logs file. Adding to this some kind of system that frequently checks the network looking for threats: corrupted information like inconsistencies between users or simply lack of information, cheaters or any malfunction on our request sit-up (request-response system) system. This system can be the just the users.

- Ensure a wide safety network before opening the access to (potentially dangerous) new users and start the normal operation mode. This helps to avoid the 51 percent

attack where there are more intruders than normal users in our network and take advantage of it.

- Being ready to change the cipher and security algorithm if needed due to possible quantic computer attach. This mean that the system can be updated in this area. One option can be to implement a layer model with the security encapsulation on the top.

As it can be seen, there are so many options to develop Blockchain networks depending on the purpose of our network. One of this purposes and the one interested on this project is to trade energy. This kind of projects are starting to be developed as it can be read below.

2.3 Energy trading networks

The increasing number of DERs (Distributed Energy Resources) introduce the necessity of networks specially developed to communicate nodes and send this kind of information. These DERs are small-scale power (generation and storing) systems mainly used to provide an alternative to or an enhancement of the traditional electric power system e.g.:a solar panel on the roof of a particular house connected with a electricity storage. The final idea is to promote the self-consume and the renewable energy to the detriment of the fossil fuels.

This new scenario (often low-voltage networks) is being focused to the P2P (Peer to peer) networks and the Blockchain networks due to their advantages avoiding servers and centralization of this data. These networks must ensure that the constraints are not violated during the storage and exchange of this transactions or data.

Therefore, the project has been focused on this problematic. An interesting paper to read about this topic is [6].

A good example of this kind of projects is Energy Web. The system named Origin they have developed is a market to connect buyers. Their main aim is to impulse the renewable energy procurement and the relative markets like 100 percent certified renewable powered electric vehicle charger systems. Another example is Power Ledger. This company is completely focused on trading renewable energy and environmental commodities as it is said on their website. Their main idea is to create new markets for energy from renewable sources and connect the buyers and the providers/stores. It is interesting to see some success stories to understand how important this projects can be in some years. Another projects explained on their website are systems to set alerts and start charging electric vehicles when power price are the lowest, control systems to understand better the origin (energy source, date of the request and amount of power) of the energy consumed.

More interesting examples cold be the ones in Stromdao, Hyperledger or Corda. Most of them are plug-and-play projects that try to adjust to the scenario or client needs.

After this explanation, it follows my project. The way it has been tried to unify this type of projects and develop a all-in-one Blockchain network.

2.4 Public key and elliptic curve cryptography

The math behind the Blockchain could be summarized with a simple comparison between two really different cryptography protocols: the Public Key and the Elliptic Curve cryptography (ECC).

In summary, the ECC provides an one-way function (shown on the Figure 2.1) whose signature is relatively easy to be verified, but it is really hard to work back from publicly available data (in a similar way to get the Private Key even if we have the Public Key) to discover the private data.

The ECC works by using an equation such as:

$$y^2 = x^3 + ax + b$$

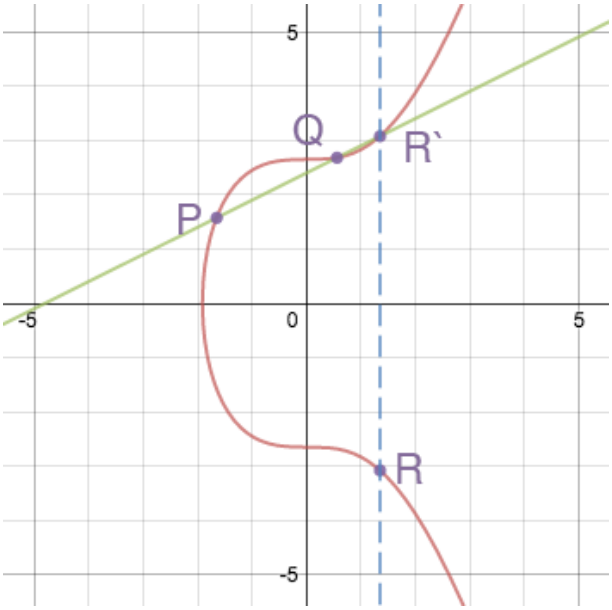

Figure 2.1: Common elliptic curve function

In some tokens like Bitcoin, it is usually taken a=0 and b=7 so the equation looks more simple, it is like:

$$y^2 = x^3 + 7$$

ECC is a type of PK cryptography. It has some curious properties such as the fact that a non-vertical line intersecting two non-tangent points will always intersect a third point on the curve as it is shown on the figure above as shown on the figure above.

The key on this cryptographic algorithm is to discover the third point once we have the two other points.

It can be defined the "addition" operation on the curve, what is basically done in Elliptic Curve Digital Signature Algorithm (ECDSA), the analogous to RSA (named due to its inventors: Rivest–Shamir–Adleman) in PK Cryptosystems. In particular, in ECDSA, addition of two points (p1,p2) and (q1,q2), and the doubling of (p1,p2), are performed as follows (where M as a large prime number):

$$c = \frac{q_2 - p_2}{q_1 - p_1} mod M$$

$$r_1 = (c_2 - p_1 - q_1) mod M$$

$$r_2 = (c(p_1 - r_1)^{-} p_2) mod M$$

And "doubling" of (p1,p2):

$$c = \frac{3p_{12}}{2p_2} mod M$$

$$r_1 = c_2 - 2p_1$$

$$r_2 = c(p_1 - r_1) - p_2$$

As it is said on [1] "Since M is a prime, every non-zero integer from 1 to M-1 has a multiplicative inverse. One other preliminary detail is multiplying in this algebraic structure, in particular to calculate an expression such as m*(p1,p2) for integer m. This can be done by first doubling the input (p1,p2), and then using the addition algorithm repeatedly until m copies of (p1,p2) have been added, but this, of course, is not practical when m and M are very large, as they are in real Blockchain applications. Instead, such operations are typically done using a binary algorithm for multiplication, which are for integers but can be easily adapted to ECDSA.". Then, to compute:

$$r = (n * b) mod M$$

It is needed "r" to be the largest power of two such that "t" is less than or equal to "n", and set r=1. After this, there are 2 conditions:

- If n is greater than or equal to t, set:

$$r = (b + r) mod M$$

$$n = n - t$$

  else set:

$$t = \frac{t}{2}$$

- If t is greater than or equal to 1 set:

$$r = (2r) mod M$$

  and go to the first step (if t<1, then we are done).

Again, as it is said on [1] "Computer scientists will immediately recognize this as almost identical to the binary algorithm for exponent M, except that we are adding and doubling instead of multiplying and squaring.

First, select M, a "base point" (p1,p2), and a private key k1 (integer between 1 and M-1). These are typically selected such that the order of the base point (namely the maximum number of times (p1,p2) can be added to itself before it fails due to a division by zero) is prime and at least as large as M (this is not required but is normally done). This often takes some experimentation, although practical applications can do this very rapidly.

As a concrete example, let us take M=199 (which is prime), and the base point (p1,p2)= (2,24). For this M and (p1,p2), one can calculate that the order n=211. Then let us select as our private key k1=151. We first need to calculate the public key (r1,r2) corresponding to the private key".

This is done by multiplication:

$$(r_1, r_2) = k_1 * (p_1, p_2)$$

Where the multiplication is done either by repeated summation or by the binary algorithm above.

If we do this, we find that the public key (r1,r2)=(64,80). Now select data z1, say z1=104. We shall construct a digital signature of the data.

This is done as follows:

1. Choose an integer k2 between 1 and n-1, where n is the order.

2. Calculate:
$$(s1, s2) = k_2 * (p1, p2)$$
If s1 = 0, return to step 1.

3. Calculate:
$$s2 = \frac{(z_1 + s_1 * k_1)}{k2} modn$$
If s2=0, return to step 1.

Then the digital signature is (s1,s2). In our specific case, if we select k2=115, we calculate (s1,s2)=(99,52).

Now we can test the digital signature, as a third party might to verify that the transaction (which in this example we presume is coded in the data z1=104) is valid.

This is done as follows:

- Calculate:
$$u_1 = (s_2 - 1) modn$$
- Calculate:
$$u_2 = (z_1 * u_1) modn$$
- Calculate:
$$u_3 = (s_1 * u_1) modn$$
- Calculate:
$$(t_1, t_2) = (u_2 * (p_1, p_2) + u_3 * (r_1, r_2)) modn$$
Verify that t1=s1. In our case, we find that the result of step 4 is (t1,t2)=(99,44). Since t1=99=s1 (see above), the validity of the signature is confirmed (it is not necessary for t2 to equal s2).

In conclusion, the ECC system is a PK system evolution to increase the difficulty of the reversal calculus and improve the security to the traditional computers needed due to this computes evolution.

Anyway, as it was said previously, the quantic computers can make both of this two systems obsolete for their completely different way to work and calculate but it is considered it is important to explained this two systems in this section.

The ECC scheme will be shown on the Figure 2.2.

As it has been explained and cited on this article [1], the mathematics used on the ECC is not trivial (neither in the PK system, but it is even more complex) and the scheme needed to implement the scheme are more particular. This is the reason why it is mostly used specific blocks to do this work instead of using general purpose hardware.

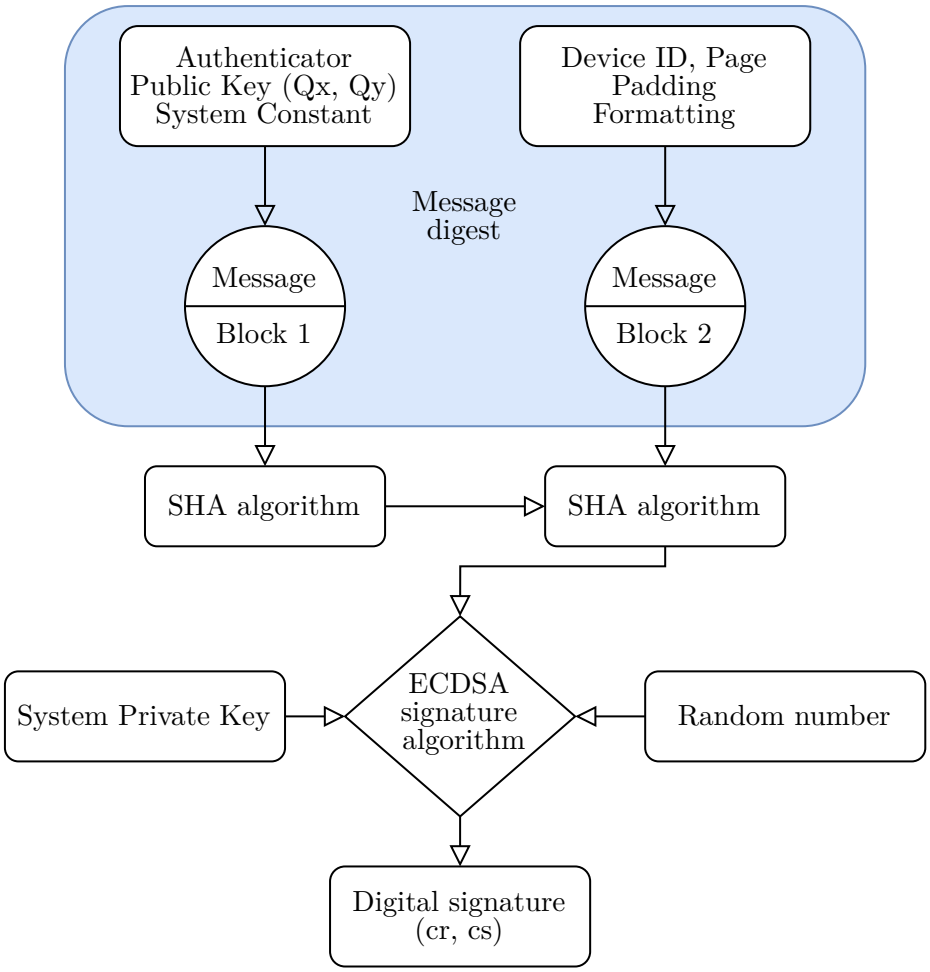

Figure 2.2: Elliptic curve scheme

It is too recommended to read the article [14] to understand more deeply the difference between this two systems due to it is not going to be explained deeper in this thesis. Only remark that due to this is a general purpose Blockchain network, it has been decided to use the Public Key cryptography instead the Elliptic Curve cryptography to simplify (there are already done libraries and functions) the project and be able to finish it. The development of a ECC system could be, by its own, a thesis.

2.5 State of the art analysis

The state of the art is a means to an end. In my current task (to learn about and develop a Blockchain network) it is needed to search the information meticulously due to the huge hype around this topic. This is said because the first problem to fight against is the large number of sites, blogs, article and projects can be found.

First of all, it must be pointed out that the Blockchain technologies and networks are being developed in multiple success stories across the globe. Some of this examples are the Chinese most important monetarian entity: the People's Bank of China (PBC) with some currency systems and insurance systems developed with Blockchain, the most important cryptocurrencies like Bitcoin, Ethereum or Ripple, some healthcare (interesting to save the patient's medical data safe and secure) and logistic systems based on Blockchain or even taxes systems that are starting to be developed with this technology in contrast with the traditional systems (really interesting to track taxes and avoid cheats and frauds).

Some writers does not make clear distinction between Blockchain and another technologies like DLT or P2P. Some others had not correctly structured the idea of how a Blockchain network works (in example, believe that it is mandatory to use an specific technology or algorithm like the PK or ECC just explained on the previous section).

This is why it has been finally decided to begin in the beginning: the White Paper written by Satoshi Nakamoto [13]. After this, it is recommended to search for some trusted papers and books like Mastering Bitcoin [2] to start to make the differentiation between Blockchain and another technologies with some parts in common, some Blockchain systems or success stories like Ethereum, Bitcoin and Power Ledger.

Once the idea of this kind of distributed networks, were the data are (usually) stored as hashes in a chain, is clear it is time to start to decide in which language does it preferred to develop the project. To do this task, it is recommended to find and try some projects (the most of the ones tried did not compiled, used obsolete libraries or were uncompleted projects) so it can be easier to see which of the programming languages already known can be more accurated with this project or if it was interesting to learn a new one (it was considered C++). After trying lots of this uncompleted projects from the most famous code repositories, it was decided to develop the project on Python due to its versatility because there are so many libraries and functions already developed ready to be used by yourself as explained on the "Libraries utilized" chapter.

In this point, it was realized the need of explaining more about the math behind the Public Key Cryptography (and the Elliptic Curve Cryptography in contraposition) you can fin as a section from the Prior analysis chapter.

After developing and testing the differences blocks, join all of them in a project and test the whole project, it is as much important as the project to write the conclusions attached on the "Conclusions summary" chapter.

Chapter 3: Project development

This project has been developed on Python release 3.7 and the Python language is an object oriented programming languages (similar to Java language) due to the already developed Blockchain and cipher libraries. It is important to utilize wisely the libraries already created by the Python community to improve and speed up the development of the code.

This chapter must explain the code and to do it more clear, it has been decided to start explaining the libraries utilized on it explained by its own page. After this, it will be possible to explain in Section 3.2 little chunks of code I consider needed to with pseudo-code to make it the as generic and clear as possible.

Last but not least, I will show some screenshots of the programs running in different sockets and interacting to the other ones in Section 3.3.

In conclusion, this project wants to create a Blockchain network where the MNs (Master Nodes) are connected as shown on the Figure 3.1. These MNs, at the same time, will be connected by clients and eventually, by generators as shown on the Figure 3.2.

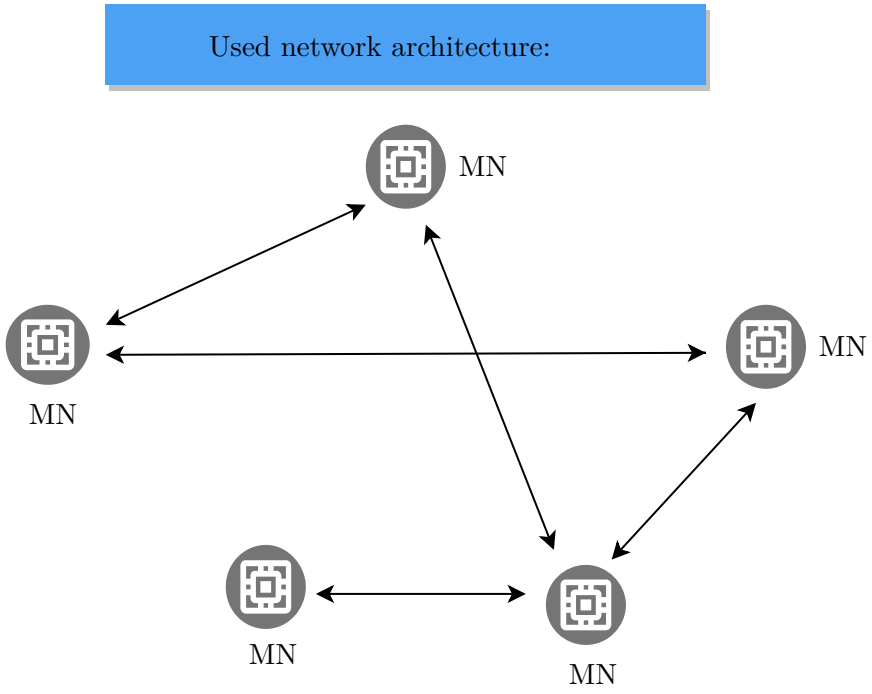

Figure 3.1: Used network architecture

As it can be seen on the Figure 3.2, the connection Generator-Master Node is different to the MN-MN or Client-MN. This is because it is not needed to have a generator connected

in a MN (imagine the case of a client who consumes energy from the network but does not generate electricity).

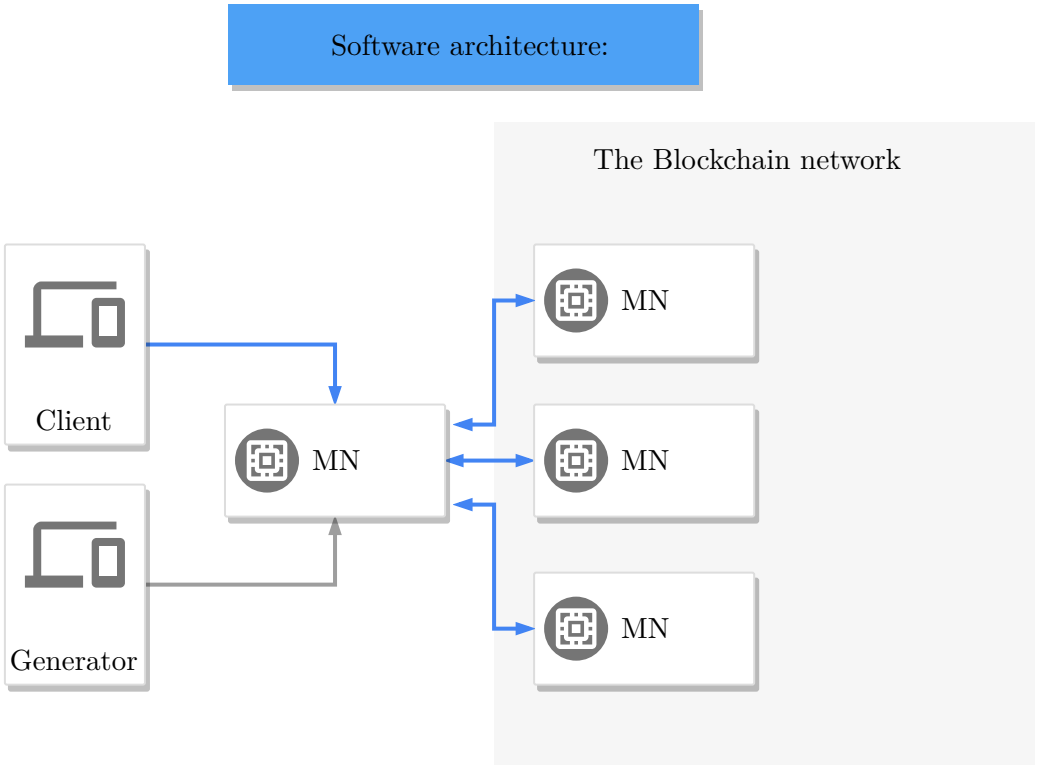

Figure 3.2: Software architecture

3.1 Libraries utilized

As it is said above, here it is going to be explained the libraries utilized in my project and put the links to their proper web-page to make it easier referring to it. At the same time, it will be explained why do I recommend to use this libraries and objects or at least, why were them decided to be used.

- **Argument parser** from argparse

  As it is said on the library: "The argparse module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and argparse will figure out how to parse those out of sys.argv".

  There are three different input argument parsers: SYS, GETOPT and ARGPARSE. Trying to explain it briefly, the main differences between this options are:

  1. SYS comes from a standard Python installation and this is the simplest option. It is quite limited and could not be enough for some purposes.
  2. GETOPT is a more flexible option in comparison with Sys. it would be used in a similar way to *getopt.getopt(args, shortopts, longopts=[])* where "args" is the list of taken arguments, "shortops" is where the option letters are specified and

"longopts" is where the extended versions of the shortopts is selected (e.g.: -b will bean bind, the IP port).

3. ARGPARSE is the most complex (and complete) option. This is due to multiple aditional options like selecting if a variable it is needed (required = True) or not, an explanation it was added named "help"... In conclusion, it is more readable. This is why it was chosen to parse by using ARGPARSE.

It was used for example in the following lines:

- ***parser = ArgumentParser()***
  Here, an parser object is created and named "parser".
- ***parser.add_argument('-b', '–bind', default='127.0.0.1', help='the address to bind to')***
  After creating parser, it must be added some arguments like "bind" in this example.
- ***args = parser.parse_args()***
  When the program is initiated from the command line, the arguments are retrieved and can be stored. In this example, the arguments are stored in "args" to be used in a future.
- ***host_mn = args.bind***
  Finally, it can be easily gotten any item (like bind in the example) from "args" in any time when the program is running.

- **Timestamps** from Datetime

  As it is said on the library: "The datetime module supplies classes for manipulating dates and times. While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation [...] classmethod datetime.now (tz=None) Return the current local date and time.

  If optional argument tz is None or not specified, this is like today(), but, if possible, supplies more precision than can be gotten from going through a time.time() timestamp (for example, this may be possible on platforms supplying the C gettimeofday() function).".

  The additional option to DATETIME would be using timestamps but it is (again) more readable to the final user and it is more clear to see the date and time written like 2020-06-10 15:24:57:12345 than just a random-likely number.

  It was used for example in the following lines:

  - ***now = datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")***
    Here, it is requested the current date and time with a millisecond exactitude (could be easily changed to microsecond) as explained on the library.

- **Ordered lists** from Dictionary

  There are so many ways to order objects in lists or similar higher level objects so studying a few of the most used ones, it was decided to use the Dictionary.

  As it is said on the library: "Fixes dictionary iteration methods. dict.iteritems() is converted to dict.items(), dict.iterkeys() to dict.keys(), and dict.itervalues() to

dict.values() [...]
It also wraps existing usages of dict.items(), dict.keys(), and dict.values() in a call to list.".

It was decided to use DICTIONARY in contrast with arrays or lists due to the type of the argument can change (in example, it could be interesting to save the port of a MN but not the variable port from a client and in the other hand, it could be desired to save the client's PK or connected MN), it is a list where it is not needed to index integers (it can be added directly an object named "key"). In conclusion, a DICTIONARY is the list, array list and ordered buffer evolution to save objects in Python.

It was used for example in the following lines:

- ***payload_dictionary = {'type_of_data': type_of_data, 'sender_address': src_host_mn, 'recipient_address': dst_ipaddress, 'sender_port': src_port_mn}***
  Where a dictionary is created to store and organize correctly as a packet the payload of a transaction with some keys and values (e.g.: 'sender_address': src_host_mn with src_host_mn as a string value).
- ***mn_dict = defaultdict(list)***
  Where a dictionary is defined but not initialized in case it is needed to create it without adding information yet.
- ***tentative_datemined = tentative_transaction.get('date_mined')***
  Where a value is read in the position/key in brackets. In this case, read (from the value with key name "date_mined") and stored in a variable named tentative_datemined.
- ***Balance_dict[tentative_miner] = DEFAULT_BALANCE***
  Where a value is stored in the dictionary. In this case, the dictionary modifies (or adds in case it was not already created) an entry in the dictionary for the key stored in the variable "tentative_miner" and the value "DEFAULT_BALANCE".

- **Object serialization** from Pickle

  As it is said on the library: "The pickle module implements binary protocols for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.".

  It was decided to use the Pickle module because it is the most used serializing and de-serializing binary object in Python. Despite the fact that it must be verified the sender of the pickle to avoid security problems (it can be executed arbitrary code during the unpickling process as explained on the library), it is probably the more complete and easy-to-use object on python. This is one of the reasons why it is so widely used. It was too tried another options like JSON serialization, YAML serialization and read about CSV files to avoid the security issue but finally and after discussing this topic, It was made this decision.

  It was used for example in the following lines:

- ***mn_dict_pickle = pickle.dumps(mn_dict)***
  Where a dictionary named mn_dict (the dictionary were it is stored the list of Master Nodes connected) is pickled on a pickle named mn_dict_pickle.
- ***mn_dict = pickle.loads(mn_dict_pickle)***
  Where a pickle is unpickled to get the dictionary inside and it is saved on the dictionary variable named mn_dict.

- **Pseudo-random integer generator** from Random

  As it is said on the library: "This module implements pseudo-random number generators for various distributions.

  For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement [...] random.randrange(start, stop[, step]) Return a randomly selected element from range(start, stop, step). This is equivalent to choice(range(start, stop, step)), but does not actually build a range object.

  The positional argument pattern matches that of range(). Keyword arguments should not be used because the function may use them in unexpected ways.".

  There is not so much to say about this module.It was chosen this object because is the native pseudo-random integer generator in Python.

  - ***src_port_mn = random.randrange(1024, 65535)***
    Here, this call is used to get a random integer for a dynamically and random selector of port. This is because it is needed to change the port to avoid problems in some Operative Systems (Windows has problems actualizing the free ports, it is done with a selected timeout). This range is because of the list of TCP free port numbers.

- **Socket object and functions** from Socket

  As it is said on the library: "This module provides access to the BSD socket interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms. [...] The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the socket() function returns a socket object whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with read() and write() operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.".

  Like in the previous case of the random library, it was chosen this object because it is the native socket library in Python and it is the most actualized one.

  It was used for example in the following lines:

  - ***c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)***
    Here, an socket type object is created as explained on the library: the family is AF_INET and the type is SOCK_STREAM.

- ***c.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)***
  Where it is added some additional flags/options like telling the kernel to reuse a local socket in TIME_WAIT (modifiable variable) state, without waiting for its natural timeout to expire (needed to receive more than a transaction every 2 minutes).
- ***c.bind((host_c, port_c))***
  Where the code binds the socket to the address (host_c:port_c e.g.: 127.0.0.1:12345). The socket must not already be bound. As you can see, it is working with IPv4 addresses but it could be easily modifiable (It was tried just to make sure).
- ***c.connect((host_mn, port_mn))***
  As it is said in the link above: "Connect to a remote socket at address. [...] If the connection is interrupted by a signal, the method waits until the connection completes, or raise a socket.timeout on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an InterruptedError exception if the connection is interrupted by a signal (or the exception raised by the signal handler).".
- ***c.send(message.encode('ascii'))***
  c.sendall(payload_pickle)
  About these 2 calls, we can read in the link above that: "Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for recv() above. Unlike send(), this method continues to send data from bytes until either all data has been sent or an error occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.".
  In this case, message.encode('ascii') converts the string variable "message" into an string of ascii values able to be sent without pickling. In the other hand, to send a dictionary is always needed to serialize with for example a pickle.
- ***c.shutdown(socket.SHUT_RDWR)***
  As it is said in the link above: "Shut down one or both halves of the connection. If how is SHUT_RD, further receives are disallowed. If how is SHUT_WR, further sends are disallowed. If how is SHUT_RDWR, further sends and receives are disallowed.".
- ***c.close()***
  As it is said in the link above: "Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from makefile() are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected, but it is recommended to close() them explicitly, or to use a with statement around them. Changed in version 3.6: OSError is now raised if an error occurs when the underlying close() call is made.".

- **Public and private keys generator** from RSA
  As it is said on the library: "RSA is the most widespread and used public key algorithm. Its security is based on the difficulty of factoring large integers. The algorithm

has withstood attacks for more than 30 years, and it is therefore considered reasonably secure for new designs. The algorithm can be used for both confidentiality (encryption) and authentication (digital signature). It is worth noting that signing and decryption are significantly slower than verification and encryption. The cryptographic strength is primarily linked to the length of the RSA modulus n. In 2017, a sufficient length is deemed to be 2048 bits. For more information, see the most recent ECRYPT report. Both RSA ciphertexts and RSA signatures are as large as the RSA modulus n (256 bytes if n is 2048 bit long). The module Crypto.PublicKey.RSA provides facilities for generating new RSA keys, reconstructing them from known components, exporting them, and importing them.".

All of this, considering the problems with quantum computers as it was explained previously. In my case, it was randomly generated a 1024 bits private key on the code line displayed.

It was used for example in the following lines:

- ***public_key = private_key.publickey()***
  Where a public key (the pair for the private key) is created from the private one.
- ***public_key_export = public_key.exportKey("PEM")***
  Where the public key is exported in a PEM format. This PEM-encoded data is a series of length plus data pair with the length encoded as 4 octets in big-endian order. It is useful in ssh-rsa key's encoding as our casuistry. This part of the code is needed to send the public key to the notes that receive a signature from the sender to verify if the signature was made with the private key (pair and private) of the public key sent.
- ***h = SHA256.new(payload_pickle)***
  Where a hash is created from a payload (pickle of a dictionary of a payload to be strict). This is part of the signature on public key encryption.

- **Signature verifier** from PKCS1_v1_5

  Basically, the "signer" is an object needed to sign a hash (named "h" in the example below). A few options were taken on consideration for it but this one was the most recommended due to its simplicity and efficiency. As it will be shown below, it is only needed to sign and verify hashes and it is perfect for this purpose.

  It was used for example in the following lines:

  - ***signer = PKCS1_v1_5.new(private_key)***
    Here, a signer is created (with the private key) by the sender of the message as explained on the link above.
  - ***signature = signer.sign(h)***
    After creating a signer, the hash (previously calculated) must be signed.
  - ***if verifier.verify(h, signature):***
    | print("\tMN_T:—> The signature IS authentic.")
    | if host_c == message_dict.get('sender_address'):
    | | print("\tMN_T:—> The packet contains the sender address correctly.")

/ / verification = True / else:
/ print("\tMN_T:—> The signature is NOT authentic.")

If everything goes well, the signature can be verified with the public key (known by the receiver of the message) as explained here. This is a code it has been created to make the code clearer but it is not needed to add it to your project.

In case the signature is correct, it is tested if the packet contains the sender address to know if the sender is correctly saying that "this user" sends the amount of value and avoid frauds.

- **Pipe or queue to communicate threads** from Queue and the RFC8017

  As it is said on the library: "The queue module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The Queue class in this module implements all the required locking semantics. The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the heapq module) and the lowest valued entry is retrieved first.".

  In this case, the FIFO (First In, First Out) queue is the selected one due to the simple priority between transactions received and transactions to be mined. In the code above, a queue variable named "cola_pk" is created to store the Public Key dictionary. This is because it is needed a queue to send objects between threads in case of a future implementation of parallelization. It will be talked later.

  It was used for example in the following lines:

  - ***cola_pk = queue.Queue()***
    Here, a queue is created by default (FIFO queue) and named "cola_pk" due to it is going to be used to save the public keys dictionary.
  - ***mn_dict_pickle = cola_mn.get()***
    Where it is obtained an object from the queue "cola_mn" and stored in the variable named "mn_dict_pickle" (due to the MN list is a dictionary and this dictionary is pickelized to be able to store it on the queue).
  - ***cola_mn.put(mn_dict_pickle)***
    Where the "mn_dict_pickle" is put on the cola_mn queue. The inverse step to the previously explained.

- **Semaphore or lock to control critical sections** from Lock

  As it is said on the library: "This module constructs higher-level threading interfaces on top of the lower level _thread module. See also the queue module.".

  This part of the project is not needed but if it is desired to implement the parallelization and thread system proposed later. Due to the idea of the project, it would be really easy to add the acquire and release conditions before and after the parallel/critical section to make the threads wait for the rest of threads to release the semaphore. A semaphore/lock is a high-level threading interfaces as said, often used to control the access to a critical section (a part of the code where only one thread can access at

the same time e.g.: a variable that changes its value in every access or the access to a printer function) and avoid consistency problems. It is well explained on this site in case of any doubt. It was used for example in the following lines:

- ***semaphore_lock = threading.Lock()***
  Here, a lock object is created and named "semaphore_lock".
- ***semaphore_lock.acquire()***
  Where the semaphore/lock is acquired by a thread. In case of the semaphore/lock counter is 0 after this call, there will be no more threads with the option to access to the sections with a "semaphore_lock" acquire.
- ***semaphore_lock.release()***
  Where the semaphore/lock is released by a thread. Usually, this comes after a acquire and a critical section to let the rest of the threads fight for the critical section and the control of the "semaphore_lock" acquisition.

- **Thread object and functions** from Thread

  As it is said on the library: "A class that represents a thread of control. This class can be safely sub-classed in a limited fashion." and makes it run the target function "threaded" implemented by me.

  The THREADING and the THREAD libraries has been decided to be used for this purpose due to there are the original Python libraries and this means that there are native functions (more efficient due to it was made for this main purpose).

  It was used for example in the following lines:

  - ***t = Thread(target=threaded, args=(c, addr))***
    Here, a thread object is created. The target is the name of the function called when the thread starts to run and the args are a tuple of input arguments. In this case, the arguments needed are the socket which communicates the thread with the sender of the received transaction (remember that the threads are created to deal with the work of the received transactions).
  - ***t.start()***
    Explained as: "Start the thread's activity. It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control. This method will raise a RuntimeError if called more than once on the same thread object".
  - ***t.join()***
    Said that: "Wait until the thread terminates. This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs. When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns , you must call isAlive() after join() to decide whether a timeout happened – if the None thread is still alive, the join() call timed out. When the timeout argument is not present or None, the operation will block until the thread terminates. A thread can be join()ed many times.".

This is the correction in my code to let the option of an easy implementation of threads and parallelization to future versions. Not needed in the concurrent version.

Now all the imported libraries and functions has been explained, it is time to explain my code. It is going to be explained what does the .py document, the functions in each .py and the most important things to understand in each code.

3.2 Explanation of my code

This project contains three .py programs: the master node program, the client and a special or simpler client who focus only on generating value for the network, like a solar panel.

The blockchain.py explained in Subsection 3.2.3 is a MN (Master Node) whose tasks are to connect with other MNs as shown on Figure 3.1, propagate the transactions received (from clients and generators) and mining these transactions (storing and propagating the transaction when it is mined to try earning a reward). Finally, this nodes are responsible for the Blockchain where the mined transactions are stored.

The blockchain_client.py explained in Subsection 3.2.1 is a client whose tasks are create transactions (send an amount of value from its balance to another client's balance) and ask for its current balance when the user desires to.

The blockchain_client_generator.py explained in Subsection 3.2.2 is a client whose only task is generate value and send it to increase a client's balance. The analogy with a solar panel is very accurate due to it.

The first part of the code to be explained are the imports (most of them are common to the three .py codes):

```
1  # ___ SOCKET: ___
2  import socket
3  # ___ THREAD: ___
4  from threading import Thread
5  import threading
6  import random
7  # from _thread import * # the code can be improved by adding parallelization
8  # ___ CREATE PKs: ___
9  import Crypto.Random
10 from Crypto.PublicKey import RSA
11 # ___ STORE: ___
12 import pickle
13 import queue
14 from collections import defaultdict
15 import json
16 from datetime import datetime
17 # ___ VERIFY SIGNATURE: ___
18 from Crypto.Hash import SHA256
19 from Crypto.Signature import PKCS1_v1_5
20 from argparse import ArgumentParser
```

These imports were taken from the "blockchain.py" code but the ones used on the rest of codes comes from here. This is because "blockchain.py" is the most complex node.

This imported libraries are needed to create and control the sockets (line number 2 of this chunk of code), create and control the threads (lines number 4 to 5), create the private key and its public pair (lines number 4 to 5), create, sign and verify hashes (lines number

18 to 20) and some additional tasks like creating pseudo-random integer generator or the encapsulation and storing system. All of them are correctly explained above in Section 3.1.

Another important part of my project is the "type of data" field. Depending on what is the main purpose of a packet, it is said that this packet has a type of data. These types are explained in a comment below but is short, there are transactions (the message sent to inform that a user sent value to other user), current balance request and request of connection:

```
"""
type_of_data:
._____________________________________________
|1:transaction(1/2: without being/already mined) |
|2:currentBalance(1: request) |
|3:connectMN(1: request) |
|_____________________________________________|
"""
```

After these shared parts, it can be explained the first program:

3.2.1 blockchain_client.py

THE MAIN:

After a message printed to advice the start of the program, it is set the input argument named "bind". Depending on if there are input arguments or not, it can be chosen the IP address or let it be chosen by default.

To sign and cipher messages with Public Key Cryptography (PKC), it is needed a public and private key tuple so it is created. The Public Key (PK) is also stored in a special container with the PEM protocol to be sent if required. This is because a PK can not be sent as a normal object in a pickle, it is needed a special container and the PK is needed to verify a signature.

```
        print("\t3-_____ MENU _____\n"
              "\t\t1.: Ask for my Public key\n"
              "\t\t2.: Make a transaction (To send an amount to a client)\n"
              "\t\t3.: Ask for my current balance\n"
              "\t\t0.: End the client.")
        decision = int(input("\tWhat would you (the client) like to do: "))
```

To make the code as clear as possible, it has been decided to implement a menu in a while. This is a infinite loop that will request an action to do something depending on the selected action. To finish this loop and the program, it is needed to select an option, "0" in this case, that changes a boolean variable named "cont" (name derived from continue) into False.

In case the user selects the option "1", the program will show (print a string by console or command line) the PK. It can not be copied from here but it can be good to take a look. As it can be seen, it is needed special functions to deal with the PKs correctly.

In case the user selects the option "2", the user is requesting for a new transaction. To create a new transaction (to sent an amount of value from this client account's balance to other user's balance) it is needed some additional information that must be required to. This additional information are the IP address of the receiver (who will earn the amount of value sent) and the amount of value to be sent, all of them requested by console to the user. It is not needed to request for it but it will essential to get the current date and time when the transaction is created, it will be stored with the variable name "now".

```
1           # 4-Create packet (payload_dictionary)
2           # Payload: dictionary -> string -> pickle ___send___ pickle -> string -> dictionary
3           payload_dictionary = {'type_of_data': type_of_data,
4                                 'sender_address': host_c,
5                                 'recipient_address': destination_ip,
6                                 'value': value,
7                                 'transaction_date': now,
8                                 }
9           payload_str = str(payload_dictionary)
10          payload_pickle = pickle.dumps(payload_str)
11          h = SHA256.new(payload_pickle) # payload_b)
12          signer = PKCS1_v1_5.new(private_key)
13          signature = signer.sign(h)
14          signature_pickle = pickle.dumps(signature)
```

With this information, it can be created the packet that will be sent. This packet or payload (the information in a packet message system that is useful to the user) is stored as a dictionary (lines number 3 to 8). The dictionary structure was already explained but it can be remembered that it is needed to put each value in a key-value tuple (e.g.: 'type_of_data': 11).

This chunk of code is really important because it is possible to appreciate the package envelopment. A dictionary object is converted into string (str) before being introduced in a pickle envelopment (lines number 9 to 10).

At the same time, it can be seen how to create a hash (variable whose name is "h") from a pickle, so it can be signed and how to create a signer to be able to create a signature with this signer and the previously calculated hash. This is shown from the line 11 to 14.

After preparing the transaction (and its signature), it is time to prepare the method to send this message or transaction. For this purpose, it is needed to create a socket and connect it with a receiver socket (already listening when it is tried to connect with it).

```
1           # 5-Get the attributes by CommandLine (additional attributes for Socket connection)
2           host_mn = str(input("\tMaster Node IPv4 ADDRESS (ie.: 127.0.0.1): "))
3           port_mn = int(input("\tMaster Node IPv4 PORT (ie.: 12345): "))
4           while True: # .1-Avoid block because of OSError:WinError10048 or
5               # ConnectionResetError:WinError10054
6               try: # .2-Avoid block [...]
7                   port_c = random.randrange(1024, 65535) # BETWEEN 1024(49152) AND 65535
8                   c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
9                   c.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
10                  c.bind((host_c, port_c))
11                  c.connect((host_mn, port_mn))
```

To connect with the destination socket, it is needed to know the IP address and the port listening for requests, so it is requested to the user. This IP:Port tuple is from the MN who will deal with our just created transaction (verify the correctness of it, propagate, mine and store). It is required the IP and the Port in the lines 2 and 3 respectively.

Due to some issues with some Operative Systems (OS) like Windows, it is needed to be careful with the sockets. This is why some lines are added with the comment "Avoid block because of OSError:WinError10048 or ConnectionResetError:WinError10054". This system will randomly select an integer number to the sender port (in the available range, it must be considered the TCP-IP port list) and try to send to connect with the destination socket. In case it is not successful, it will choose a new random integer number and try it again until it is. The just explained system to avoid blocks is a whileTrue Try-Break-Except loop to catch the exceptions without leaving, it is too frequently called Try-Catch loop.

The last 4 lines create an socket, sets additional options, configure the IP address and port and connects with the destination socket in the MN as explained in the library section.

In case the connection is successful, the client will display (print) a message with the socket-to-socket connection information as shown and will proceed to send the transaction ("payload_pickle").

When the MN receives the transaction successfully, it will send an acknowledge (ACK) and wait for the signature.

The client sends the transaction's signature to verify that it was created by him and avoid frauds. In this point, there are two possible responses: if the MN had already stored this client's PK, it will be seen an ACK but if not, the MN will request for the PK (it was packet in a special package following the PEM protocol).

```
1                # 10-Decide if the Pk-response must be sent (Public Key in a PEM exporter)
2                print('3.2.7-PK request/ACK1 from the MN(_T) :', str(data.decode('ascii')))
3                if not str(data.decode('ascii')) == "ACK1":
4                    message = public_key_export
5                    # 10(*)-Pk sent to server (thread will receive it)
6                    c.send(message)
7                    print("3.2.7(*)-Pk sent to the MN(_T)")
8                    data = c.recv(1024)
9                    print('3.2.7(*)-ACK2 from the MN(_T) :', str(data.decode('ascii')))
```

In case the MN had this client's PK the answer will be ACK1, and will not be needed to send it. Otherwise, in case the MN did not have this client's PK the answer will not be ACK1, so it will be needed to send it. As it is shown on the code displayed, this kind of comparisons are done with a string comparison in an if structure to simplify as much as possible.

When it is needed to sent a PK (already packed following the PEM protocol), the client sends it as can be seen and finally will receive an ACK ("ACK2") from the MN.

After this, everything is done and the thread can be finished.

The socket must be closed and disconnect from the MN socket. It will be printed a message and come back to the start menu (in the client's Main, in the start of this Subsection 3.2.1).

In case the user selects the option "3", it is desired to know the this client's current balance.

To do this task, it is needed to send a balance request to a MN. It has been decided to simplify this option (unnecessary on most of the Blockchain systems) avoiding the signature, so the payload will contain only the type of data correct to choose this option in the MN.

Notice that there are three additional options I recommend for future versions depending on the characteristics demanded on your project:

1. To add a "'sender_address': host" tuple. This can improve the system's privacy as shown on the transaction ToD (Type of data).

2. To add a "'recipient_address': destination_ip" tuple. This could improve the security by adding signature.

3. To add a "'transaction_date': now" tuple. This last idea wants to improve the system by avoiding resend a message. To understand this kind of attack, it can be thought on the double-spend attack, in this case, to discover the balance of another user on the network.

Analogously, it is needed to request for the destination MN's IP and port, and to be careful with the OS issues (when dealing with sockets already used) to create a socket and connect it withe the listening MN's socket. Once the client socket is correctly connected with the listening (MN's) socket, it is time to tell the user it was everything okay and to send the message ("payload_pickle").

The MN will answer with this client's current balance in case it was on this MN's "balance_dict" or with a DEFAULT_BALANCE if not. This default balance is not needed to be set in a non-zero number, but it is recommended to let the clients make at least a transaction in a test Blockchain network.

The last but not least, in case it was not selected a wrong menu option it will be displayed an error message asking for a correct one.

After explaining the client (blockchain_client.py), it is time to explain a special client code: the generator (blockchain_client_generator.py).

3.2.2 blockchain_client_generator.py

This program only contains a main function. The first step in this code is to get the input arguments analogously to the rest of clients as explained above.

It has been added another menu (analogously to the client above explained) but in this case, it is simpler as it can be seen:

```
1    cont = True
2    while cont: # Until the user ask for the option number 0
3        print("\t3-_____ MENU _____\n"
4              "\t\t1.: Generate and send value.\n"
5              "\t\t0.: End the generator.")
6        decision = int(input("\tWhat would you (the generator) like to do: "))
```

There are not so many thing to explain because all of them were already explained on the previous subsection.

Following the rest of the clients line, I have decided to add a menu in the start as can be seen. In this case, this menu contains the main option and the option to exit and finish.

In case the user selects the option "0", the program ends as the rest of clients but, in the other hand, in case the user selects the option "1" it is desired to create value and send it to a client. The first step is to get the additional information needed like the destination IP address, the amount of value to be created and sent (this would be, in example, the amount of electricity generated by a solar panel in a house that must be added to the owner balance) and the date and time.

Once every data needed to create the message are stored, it is time to create a transaction as displayed:

```
1            # Payload: dictionary -> string -> pickle ___send___ pickle -> string -> dictionary
2            payload_dictionary = {'type_of_data': type_of_data,
3                                  'sender_address': "SUN", # inexhaustible energy source
4                                  'recipient_address': destination_ip,
5                                  'value': value,
6                                  'transaction_date': now,
7                                  }
8            payload_str = str(payload_dictionary)
9            payload_pickle = pickle.dumps(payload_str)
```

It follows the same steps: create a dictionary (packet or payload), convert the dictionary object into a string and encapsulate the string in a pickle (envelopment).

In the line number 3, it can be seen that the sender is always "SUN". This is a way to say the system that this transactions is not a trade but a generation of energy so nobody is losing it (the Sun can not get a negative balance).

Then, the packet can be sent to a MN so the transaction can be propagated and mined to make the client earn the value for the electricity generated to the network. To do this, the destination IP address and port are requested to the user.

The transaction in sent as explained on the previous subsection.

Finally, in case the selected option on the menu was neither "0" nor "1", it is requested again because it is not allowed any different option.

After the clients (and the pseudo-clients: generators) were explained, it is time to explain the most important part of the project. The MNs are the nodes who control the network and that is why it was decided to explain firstly the easier codes.

3.2.3 blockchain.py

Before starting to explain the Master Nodes, it must be explained that this code contains three functions and the main. Two of these functions are "connect_with_mn" and "propagate_transaction" whose objective are connect the MN who calls this function with another MN (adding it to the list of MNs, a dictionary to order them) and send a transaction to all the connected MNs that can be retrieved from the already explained list of MNs. In the other hand, "threaded" is a function really similar to a main due to it is the function called by the threads and guide them during their whole time of life.

It must be remembered that after a few initialization, this code runs creating threads to deal with the received transactions and after a few of them, mines the stored received transactions.

After the imports, it is time to define and initialize a few variables in the Master Node.

```
1  semaphore_lock = threading.Lock()
2  cola_pk = queue.Queue() # @ip:pk
3  cola_balance = queue.Queue() # @ip:current_balance
4  cola_mn = queue.Queue() # @ip:port
5  # (tuple with pk? +security)
6  # (tuple with last-activity date? would improve the network efficiency by deleting after
7  # a long time without activity)
8  cola_tobemined = queue.Queue() # date:transaction(without MN miner)
9  cola_tentative_alreadymined = queue.Queue() # date:transaction(with MN miner)
10 cola_alreadymined = queue.Queue() # date:transaction(withMN miner)
11 # (no one can send 2 transactions in our system at the same time and we securitize it by signature)
12 # I am supposing it is not possible to receive 2 different transaction generated at the same
13 # time by different users (if not, it would not be possible to compare the date and we would
14 # need to compare a tuple date+sender_address)
15 # i.e.: dict[(received_date, received_sender_address)] = message_rcv
16 random_gen = Crypto.Random.new().read
17 private_key = RSA.generate(1024, random_gen)
18 MINING_REWARD = 1
19 DEFAULT_BALANCE = 17
20 # MINING_DIFFICULTY = 2 # IN MY SYSTEM I DON'T NEED MINING_DIFICULTY, it would
       be similar to:
21 # 1) increase the number of transaction listened before mining (n) dynamically when the
22 # network grows up
23 # 2) increase the number of cycles before accepting a mined transaction OR increase the
24 # number of transactions to listen before mining (increase the time before accept a transaction
25 # so I let more extreme delay)
```

In the first line, a semaphore or lock is initialized. Remember, this object is really useful on the parallelized programming because it lets multiple threads to use in order (one by one and according the FIFO technique) a critical section.

Between the second and the fifteenth line, the six queues are defined: 3 queues are used to communicate the threads by putting and getting from there the dictionary (correctly enveloped in a pickle) with the connected MNs, current balances and stored PK from the Clients: the other 3 queues are used to save the transactions mined (Blockchain) in "cola_alreadymined", before being mined ("cola_tobemined ") and mined but waiting to verify that it was not mined by another MN before the one who send us the transaction ("cola_tentative_alreadymined"). Some additional ideas for future implementations are commented so it is not needed to add anything else.

The integer pseudo-random generator is initialized as "random_gen" in the same way as the Private Key needed on the PKC (Public Key Cipher).

Finally, there are defined two constant integers. The firs constant is named "MINING_REWARD" and it is the amount of value sent to the miner of a transaction in a similar way as a payment or a tax. The second constant ("DEFAULT_BALANCE") is used in some Blockchain networks to start the system before generating value.

In addition, it is commented some ideas for future versions like the traditional mining difficulty and how can be easily implemented on this system.

After this initializations, it is recommended to understand the three (two and the thread function) functions defined on the code:

```
1  def connect_with_mn(src_host_mn, src_port_mn, dst_ipaddress, dst_portaddress):
2      """
3      This function is called by a Master Node and tries to connect with a Master Node to add it
4      to the mn_dict used to propagate the transactions.
5
6      :param src_host_mn: Source @ip address Master Node
7      :param src_port_mn: Source port address Master Node
8      :param dst_ipaddress: Destination @ip address Master Node
9      :param dst_portaddress: Destination port address Master Node
10     :return: nothing. The mn_dict is actualized inside the function routine
11     """
```

This function "is called by a Master Node and tries to connect with a Master Node to add it to the mn_dict used to propagate the transactions" as it is said on the comment. When a MN calls this function, it requested the connection with other MN by the proper transaction (type of data 31)) sending. After this connection, the connected MN credentials (IP address and listening port) are stored in the MN's dictionary. In case the MN was already connected to the caller MN, this function skips and finishes.

Finally, the socket is closed, the dictionaries are encapsulated and saved on their queues and lock is released.

After this first function and depending on it, the second function deals with the work of propagate (send) a transaction to all the MNs in the list of connected MNs:

```
1  def propagate_transaction(src_host_mn, dst_ipaddress, dst_portaddress, transaction_dict):
2      """
3      This function is called to connect with the destination and send a transaction
4
5      :param src_host_mn: Source @ip address Master Node
6      :param dst_ipaddress: Destination @ip address
7      :param dst_portaddress: Destination port address
8      :param transaction_dict: Transaction to be sent
9      :return: nothing, this function just sends a transaction
```

```
10      """
```

This function can be used to send transactions from a IP to a destination (by introducing IP:Port address as a input argument when called).

It is not needed to explain little chunks of code here because most of the things are repetition of structures already explained.

The last and most important function is named "threaded". Here, it is collected the code for the threads since they are created till they die:

```
1   def threaded(c, addr):
2       # 1-RECEIVE THE MESSAGE
3       host_c = str(addr[0]) # port_c = str(addr[1])
4       semaphore_lock.acquire()
5       message_pickle_rcv = c.recv(1024) # receive the first message from the client
6       message_str = pickle.loads(message_pickle_rcv) # unbox/unpickle the message
7       message_str = message_str.replace("'", "\"")
8       message_dict = json.loads(message_str)
9       received_tod = message_dict.get('type_of_data') # received type_of_data
10      # Depending on the type_of_data received:
11      if received_tod == 11: # type_of_data: 11 (Transaction from Client or MN)
```

The input arguments for this functions are a socket "c" and the address "addr" for it. Once this function is called and before making differentiation on the kind of transaction received, it is needed to prepare the packet received.

It is shown an example of taking the lock with and "acquiere". This call lets the thread enter in a Critical Section only if there are not more thread on it. In my code it is not needed but it is an interesting improve in bigger networks (parallelization of work, in this case, receive transactions).

The transaction just received from the socket it is unpickled, converted into a string and finally and after a few adjustments by replacing characters that can produce an error, it is translated into dictionary (the final object needed to work with). Once the transaction is reconstructed as dictionary, it is possible to take fields and information from it as displayed when it is saved the value in the key "type_of_data".

Depending on the received transaction's type of data field:

- In case the "type_of_data" field is "11", it is a transaction created by a client (either simple or generator). It can too be propagated from a MN.

  In case this transaction comes from a normal client (neither MN nor generator client), it will be needed to ask for the signature (and PK if it was not stored yet) to verify the authority of this transaction. This verification process could be added on the MN/generator case if desired in future implementations.

  To verify a transaction it is needed that the signature matches (as shown below) and that the sender of the transaction field matches with the Client sender (in case the transaction was received from a Client).

```
1               # 4-VERIFY SIGNATURE
2               h = SHA256.new(message_pickle_rcv) # create hash
3               verifier = PKCS1_v1_5.new(pk_cl_imported) # create verifier with the PK from the
                    client
4               if verifier.verify(h, signature):
5                   print("\tMN_T:---------> The signature IS authentic.")
6                   if host_c == message_dict.get('sender_address'):
7                       print("\tMN_T:---------> The packet contains the sender address
                            correctly.")
```

```
8                       verification = True
9               else:
10                  print("\tMN_T:----------> The signature is NOT authentic.")
11                  # verification = False if the signature/sender_address doesn't match
```

If the transaction was successfully verified, it can be added to the list of transactions ready to be mined. To explain the three queues displayed on this bunch of code, I am going to briefly explained the mining process of this Blockchain network:

1. Add the transactions (without being mined) to be mined to the queue named "tobemined". This transactions ready to be mined are propagated to the rest of connected MNs too to let them the opportunity to mine it.
2. Add the transactions (already mined or once mined from "tobemined" to the queue named "tentative_alreadymined". These transactions miner are propagated too to let the rest of nodes know that the transaction was already mined and avoid them an useless effort.
3. Add the transaction from "tobemined" to the queue named "alreadymined". In this moment, the transaction is added to the Blockchain and saved definitely.

In case a mined transaction comes to the MN and shows that the miner MN mined an already saved transaction before, the balance of MNs are corrected. Else, it is ignored.

Following the just explained process, it is searched for the transaction just received on the queue of already mined transactions. In case this was encountered would mean that we had already stored the transaction and it has already been mined, so it can be ignored. In addition, the system could be modified by adding a tuple sender plus date of transactions to make difference between users, this could be useful in big networks.

In case the just received transaction was not already stored on the already mined transactions queue, it is time to search for it in the queue of tentative to be already mined, following the same process. Again, it would be deleted in case the system finds a match.

Finally, in case the transaction was not here, it can be said that it is a new verified transaction. Now, it is time to save this new transaction (on the list of transactions to be mined) and propagate it to the connected MNs so it can be mined as soon as possible. After saving this new transaction, it must be sent to all the connected MNs (propagated). To do this task, it can be used a loop where the already explained function "propagate_transaction" can sent the transaction to each MN saved on the list of connected MNs.

Finally, the lists are put back on their respective queues so the queues are actualized with the new information and the lock is released.

- In case the received transaction had the Type of Data (TOD) 21 instead of the 11 just explained, it would mean that the received transaction was created to request for the current balance of its sender. This is an easy task that it was decided to add so it can be easier to see the evolution of the clients balance and test the system.

  To do this task, it is needed to get the requested balance from the balance list and send it to the client that requested for it.

In some of the biggest Blockchain networks like the Bitcoin or Ethereum network, this information is not requested on the moment. This is because it could be a transaction waiting to be mined (or mined in a different MN) that would change this answer. This would be easily implemented by making the thread sleep as explained on the starting comments with the sleep library function.

After a quick verification to avoid a MN asking for a client's balance (it is not needed but it was added to show how could be done), it is searched on the balance dictionary for the requested balance.

In case this client was not on the balance dictionary yet, it is added with the default balance (this default balance is not needed to be different to zero). But if the balance was on the dictionary, it is directly taken from there.

- In case the received transaction had 31 as TOD field, the received transaction is a request for connect with a MN (the sender).

  First of all, if the MN who requests to connect with this MN already was on our list, this process ends due to the transaction can be skipped. Otherwise, both address and port are added to the MN list as a key and value pair.

  A message is sent as an ACK to tell the sender MN that everything went okay and the MN list ("mn_dict") can be put back on its queue.

- Finally, if the received TOD is equal to 12, the received message is an already mined transaction coming from a MN.

  In case this MN does not have the received transaction in any list, it will be added on the "tentative_already_mined_dict" list waiting to the next mining time to be finally added to this MN's Blockchain ("already_mined_dict").

  In case the received transaction was already saved on the list of already mined transactions, it must be compared which transaction was mined before because that will be the one that choose who was the real miner (the MN who mined the transaction before) and the one who must earn the reward.

After explaining the functions on this .py, it is time to explain the main program.

The MN's main is an infinite loop where after a pre-routine, the system listens for an amount of transactions and after it, mines all the stored transactions.

```
1  def Main():
2      # get keyboard inputs
3      parser = ArgumentParser()
4      parser.add_argument('-b', '--bind', default='127.0.0.1', help='the address to bind to')
5      parser.add_argument('-p', '--port', default=12345, help='the port to bind to')
6      args = parser.parse_args()
7      host_mn = args.bind
8      port_mn = int(args.port)
9      # 0-define variables, define dictionaries and initialize if needed
10     # I create the pk in the MNs this for if I finally add the secure mood between MNs (0,1,2)
11     public_key = private_key.publickey() # 0:pk
12     pk_export = public_key.exportKey("PEM") # 1:pk_exported
13     pk_dict = {host_mn: pk_export, # 2:pk_exported>dict
14                } # I do this for if I finally add the secure mood between MNs
15     balance_dict = {host_mn: DEFAULT_BALANCE,
16                     }
17     mn_dict = defaultdict(list) # defined but not initialized
```

```
18      pk_dict_pickle = pickle.dumps(pk_dict) # 3:pk_exported>dict>pickle
19      mn_dict_pickle = pickle.dumps(mn_dict)
20      balance_dict_pickle = pickle.dumps(balance_dict)
21
22      cola_pk.put(pk_dict_pickle) # 4:pk_exported>dict>pickle>queue
23      cola_balance.put(balance_dict_pickle)
24      cola_mn.put(mn_dict_pickle)
```

It is needed to get some arguments from the command line, firstly as input arguments (that would take default value if not selected) and later some additional ones that will be shown below.

It is too needed to prepare the keys in case of the program used the cipher or the signature (not in this version, but it has been added for if the reader wanted).

The most important part of this chunk of code is the one represented between the lines 13 and 24. This lines show how the lists (dictionaries) are created, enveloped (put in a pickle) and stored in a queue.

After this, the first additional option to be selected is if the MN must connect with another MNs (remember that the P2P architecture needs every node connected to at least another one) as shown here:

```
1       # 1-connect to some MN if needed:
2       # This MNs will be the ones to whom this MN send the transactions when it wants to propagate
3       loop_connectmn = True
4       while loop_connectmn:
5           answer = str(input("> Do you want to connect with some MasterNode? (1/0 or Y/N to choose"
6                              " yes/no): "))
7           if answer == "0" or answer == "no" or answer == "NO" or answer == "N" or answer == "n":
8               print(">> DON'T ADD MN")
9               loop_connectmn = False
10          elif answer == "1" or answer == "yes" or answer == "YES" or answer == "Y" or answer == "
                y":
11              print(">> ADD MN")
12              ipaddress = str(input(">> What this MasterNode's IP address is? (i.e.: 127.0.0.1): "))
13              portaddress = int(input(">> What this MasterNode's port is? (i.e.: 12345): "))
14              connect_with_mn(host_mn, port_mn, ipaddress, portaddress) # str, str, int
```

The second and last input argument to be requested is the number of transactions (named "n") the system needs to receive before start mining all the stored and ready to be mined transactions.

```
1       # 2-to choose n:
2       # How many transactions are there needed in the "ready to be mined" queue before mining
3       # (it's the mining_difficulty)
4       loop_n = True
5       while loop_n:
6           n = int(input("> How many transactions do you want to receive before mining? "
7                         "(Please, choose between 6 and 99): "))
8           if 5 < n < 100:
9               print(">> n=" + str(n))
10              loop_n = False
11      tobemined_dict = defaultdict(list)
12      # dict_tobemined = {new_list: [] for new_list in range(n)} # 0,1,2...n-1
13      # dict_tobemined = dict.fromkeys(range(n), [])
14      tentative_alreadymined_dict = defaultdict(list) # This is a dictionary initialization
15      alreadymined_dict = defaultdict(list)
16      tobemined_dict_pickle = pickle.dumps(tobemined_dict)
17      tentative_alreadymined_dict_pickle = pickle.dumps(tentative_alreadymined_dict)
18      alreadymined_dict_pickle = pickle.dumps(alreadymined_dict)
```

```
19      cola_tobemined.put(tobemined_dict_pickle)
20      cola_tentative_alreadymined.put(tentative_alreadymined_dict_pickle)
21      cola_alreadymined.put(alreadymined_dict_pickle)
```

Finally, the main function enters in a loop of listening for transactions and creating threads to delegate the work of each transaction. Realize that this code is concurrent but it is ready to be changed to parallel programming if desired (it is recommended in large networks to avoid collapsing the listening buffers). The "t.join()" call in the nineteenth line is the way to convert a parallel version of the program in a concurrent one.

```
1       while True: # READY TO LISTEN REQUEST: (IN WHILE true)
2           # 3-listen in a loop until I have n transactions to be mined and mine all of them
3           for i in range(n):
4               # create socket
5               semaphore_lock.acquire() # not needed, just in case
6               s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # create socket
7               s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # set SO_REUSEADDR
                    is important
8               s.bind((host_mn, port_mn)) # finish my socket configuration with my address and port
9               print("\nMN:socket binded to " + host_mn + ":" + str(port_mn))
10              s.listen(5) # the socket can have 5 petitions on queue
11              print("MN:socket is listening")
12              c, addr = s.accept()
13              print('MN:Connected to ' + addr[0] + ':' + str(addr[1]) + ' Starting a new thread...')
14              semaphore_lock.release()
15              # Start a new thread and return its identifier
16              t = Thread(target=threaded, args=(c, addr)) # I can wait for the thread
17              t.start()
18              # it would improve with a concurrent multithread system as explained on the header
19              t.join() # the thread is working and I am going to wait here for it (NO concurrency)
20              c.close()
21              s.close()
```

After the n transactions, it is time to mine. This number could be dynamically increased because it is needed to increase the time before mining in bigger networks, or at least, delegate some threads just to mine.

Before mining the transactions, it is needed to acquire the lock and get the lists (dictionaries) from their respective queues as always.

After this preparations, it is time to start mining the transactions ready to be added to the Blockchain (already mined transactions that were waiting to be confirmed).

First of all, it is important to avoid mining an already mined transaction. To do this, it is needed to compare the transaction ready to be added to the Blockchain (and the ones to be mined) with the transactions already added. In a real system, it would be (often) added a hash not the whole transaction but as it has been said multiple times, this is a educational example and the most important thing is to understand what happens. Without hashes, the efficiency of the stored data is reduced but it can be read anytime it is desired and look at the order of the Blockchain.

At the same time, it is important to avoid mine a transaction to be mined when there is a copy of it already mined on the tentative to be mined queue.

When a transaction is mined, the amount of value sent is subtracted from the sender's balance. The miner MN earns its reward for the mining work and the receiver gets the sent amount less the reward. This analogy between taxes and rewards is only an option, there are other networks where it is sent the full amount of value to the receiver and the sender's balance gets subtracted the amount plus the reward.

Finally, all the lists (dictionaries) are put back in their respective queues and the lock is released to let another thread use them.

3.3 Experimental validation

An easy implementation (from the command line in the directory where the .py are located):

1. python blockchain.py

   The system will choose the address 127.0.0.1:12345 by default for the first MN. The options recommended are 0 and 6 to avoid adding MNs to the connected MN list (it will be done by the second MN on the next step) and set the number of transactions before mining in the minimum (six).

2. python blockchain.py -b 127.0.0.2 -p 23456

   The configuration sets the socket 127.0.0.2:23456. The options recommended are 1, 127.0.0.1, 12345, 0 and 7 to add the first MN to the connected MN list and set the number of transactions before mining in a not divisible by 6 number (this is not needed but it is more graphical in case it is sent transactions only to on of this two MNs).

3. python blockchain_client.py -b 127.0.0.3

   The configuration sets the address 127.0.0.3 due to the port dynamically selected in every connection to speed up the process. The options recommended when it is chosen to create a transaction: 127.0.0.4, 5, 127.0.0.1, 12345. This would send 5 unities of value to the client located in 127.0.0.4 . The options recommended when it is chosen to request for the balance are: 127.0.0.1, 5, 127.0.0.1, 12345

The next implementation would increase the number of MNs connected in a network. It would be needed to reach some MNs to connect with some other MN. This implementation would increase the number of clients too.

Finally, some of this clients could create value (generate electricity) so it can be created some electric generators like this:

4. python blockchain_client_generator.py -b 127.0.0.5

The configuration sets the address 127.0.0.5 with dynamical port. The options recommended when it is chosen to create value (a transaction) are 127.0.0.3, 5, 127.0.0.1, 12345. This would create 5 unities of value and send it to the client 127.0.0.3 by the MN located in 127.0.0.1:12345 who will propagate the transaction and try to mine it firstly.

Following this recommendations, it was prepared an example to create a network similar to the one on the Figure 3.3. As it can be seen, the Client-MN and the MN-MN communication are bidirectionals in contrast with the (Client) Generator-MN communication. The port in the both the client and generator side are dynamically selected.

As it can be seen, and analogously to the Figure 3.2, the generator is an unnecessary or optional node so it is marked with an unidirectional grey line instead of the blue line. The line is unidirectional because it is not needed to send information from the MN to the generator.

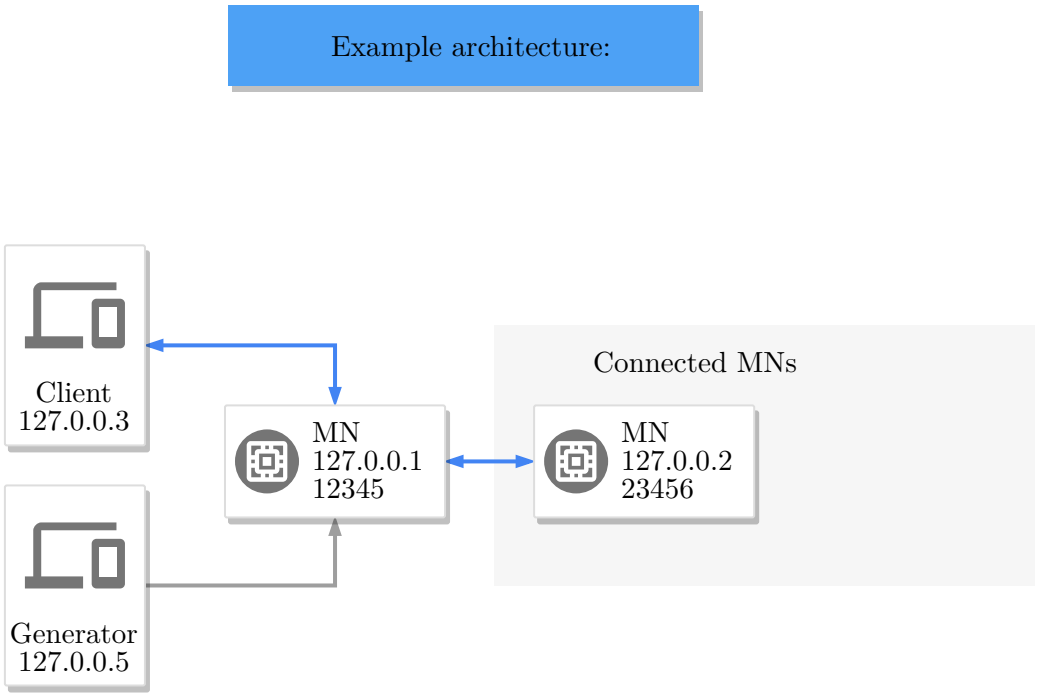

Figure 3.3: Example network architecture

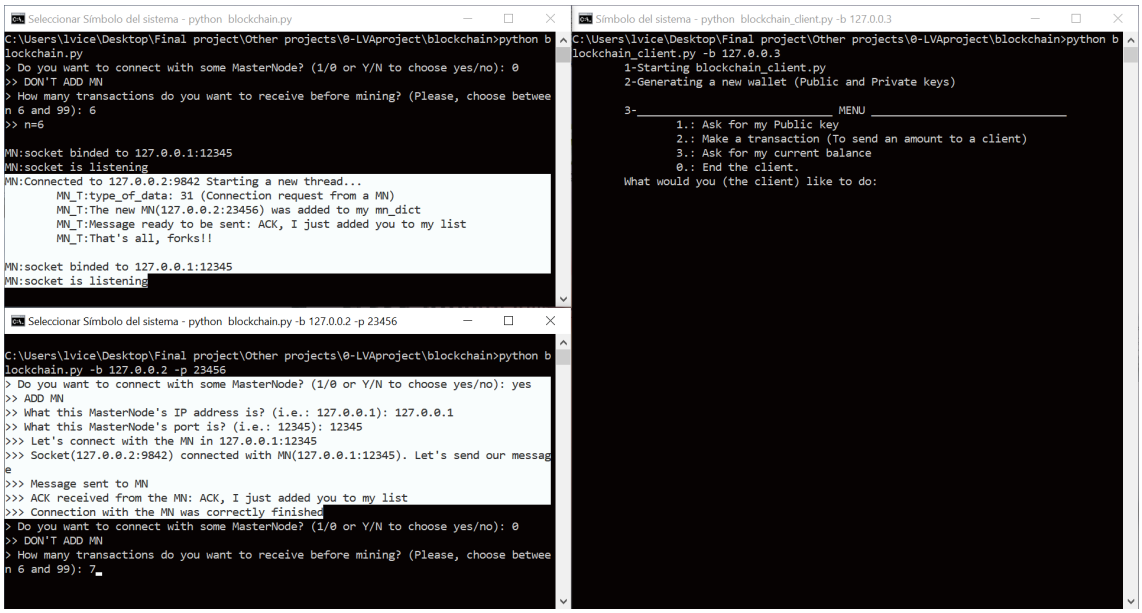

Figure 3.4: Initialization of two MNs and a Client

After creating two MNs (left side of the screenshot displayed as Figure 3.4) and a client (right side) with the options explained above as "easy implementation", the second MN has been connected with the first one (in black over white). In this way, every transaction received or mined by this MNs will be propagated to the other one. This is the first transaction received in 127.0.0.1 (it will mine after the sixth received transaction).

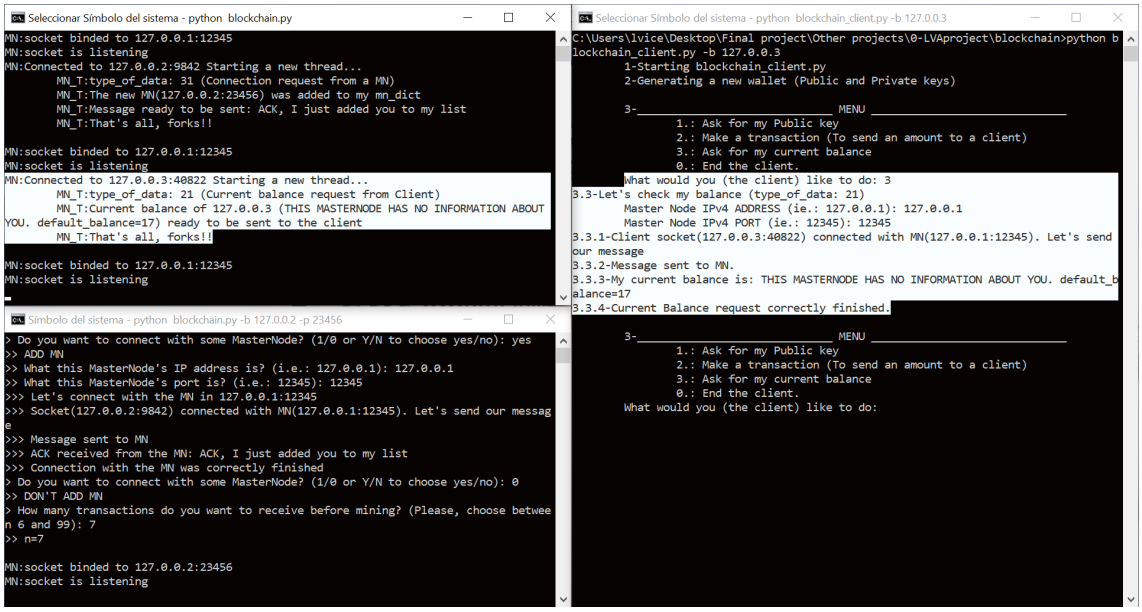

Figure 3.5: First request for current Client's balance

After doing the configuration on the first creenshot, the client can ask for its current balance or to create a transaction (send value). Firstly, it has been chosen the balance request option as shown on pictures from Figure 3.5 to 3.6.

As it can be seen, the client wasn't on this MN's balance list, so it is added with the default balance value: 17.

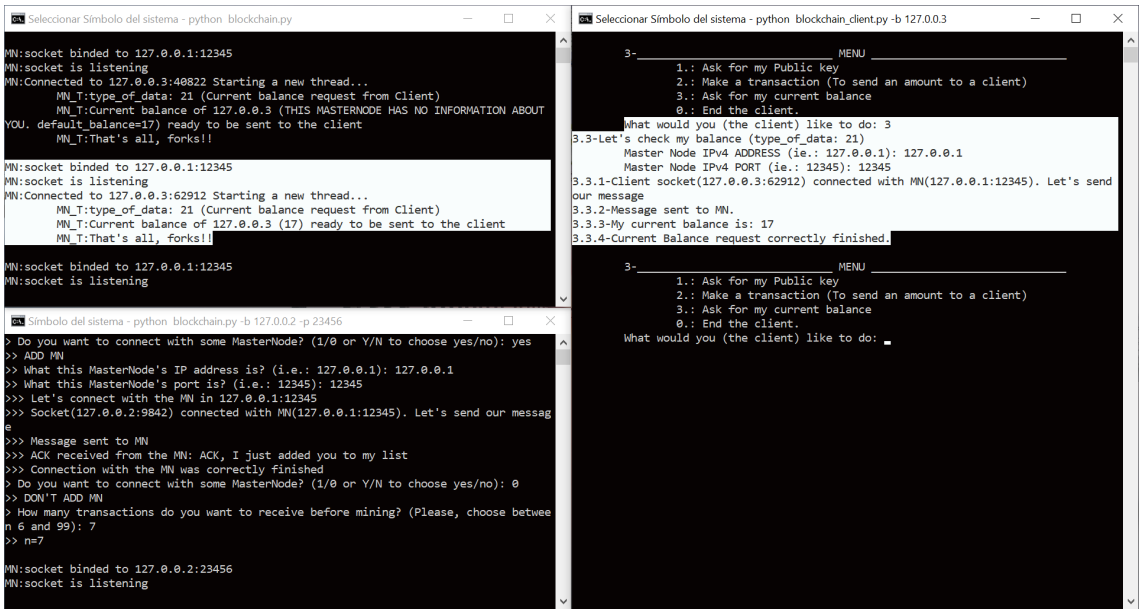

Figure 3.6: Second request for current Client's balance

The second time the client requests for its balance, it will directly receive its current balance as displayed.

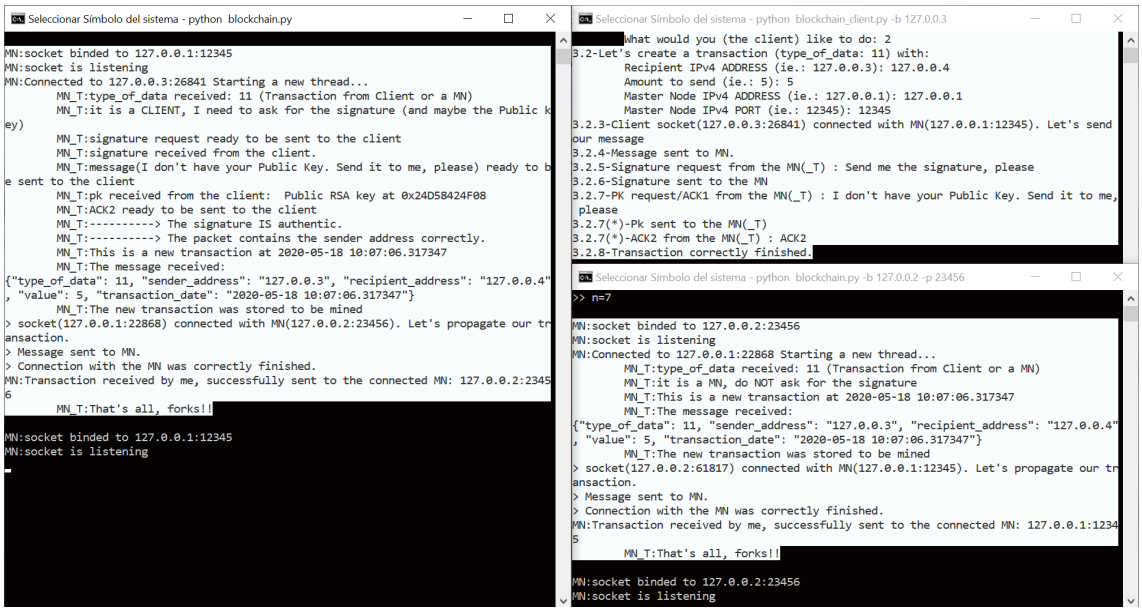

Figure 3.7: Creation of the first transaction

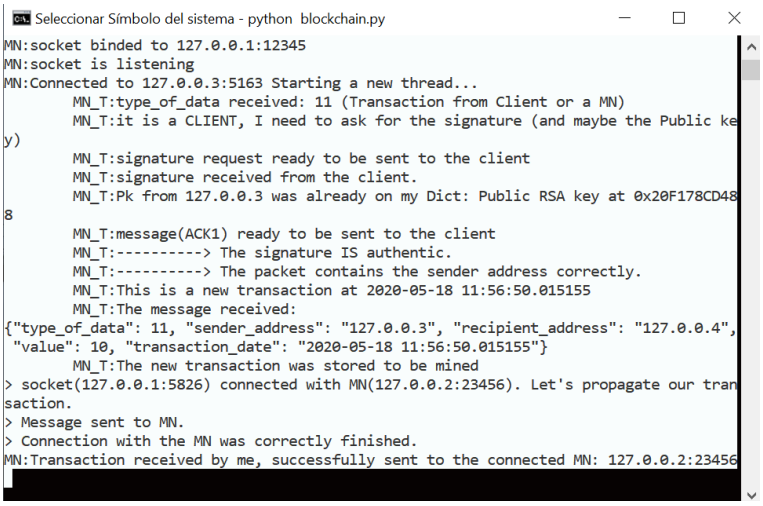

Figure 3.8: Response to the second transaction

The second option on the client side is to create a transaction so it can be sent an amount of value (it has been chosen 5) to another client (127.0.0.4) through a MN (127.0.0.1:12345). The receiver MN verifies the correctness of the transaction asking for the signature, the client's PK to verify the signature if needed (only the first time because it is saved on a dictionary to speed up the transaction system on future transactions). This is shown on the Figure 3.7

It is important to verify that the sender client (the socket that requested the connection) is the one that says to be the sender on the packet (the field sender_address of the transaction). If everything is correct, this MN propagates the transaction to the rest of connected MNs (it will be received by the second MN, displayed on the bottom right corner).

As it can be seen on the second screenshot (Figure 3.8), the PK is already stored and it is skipped to request for it.

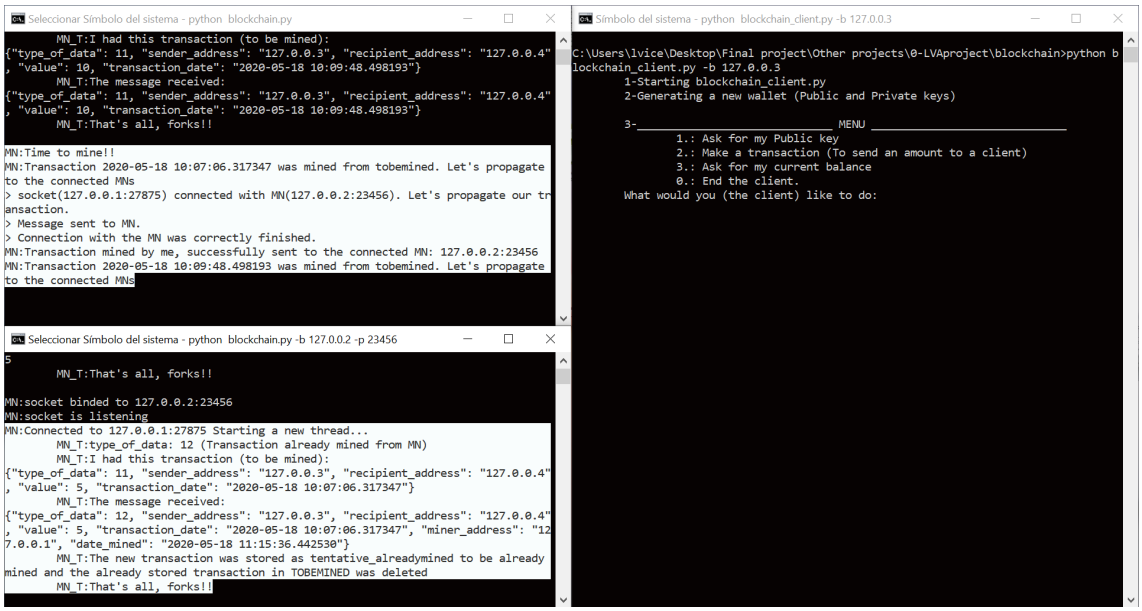

Figure 3.9: Time to the MN to mine the received transactions

After a few transactions (another sent with an amount of 10 and a few requests of balance to make the system run), on the sixth iteration, the first MN will mine all of them as explained and propagate the transactions mined by itself. This is why the second MN receives a transaction mined and overwrites its queues to delete the old transaction (before being mined) and adds the new one (already mined by the first MN).

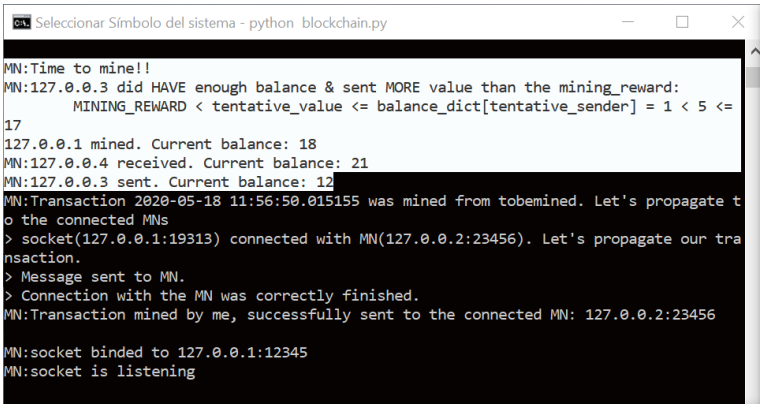

Figure 3.10: Second time the MN is going to mine

On the second time to mine, the transaction stored as mined is accepted (it was not received any other transaction that shows another MN mining the transaction before so the MN adds the amount of value called reward to the miner MN for its work, the amount of value sent (after subtracting the reward) to the receiver and it is subtracted the amount sent to the sender as displayed. The accepted transaction is added to the Blockchain.

In case an incoming transaction shows another MN mining before, it would be changed the balances and the Blockchain.

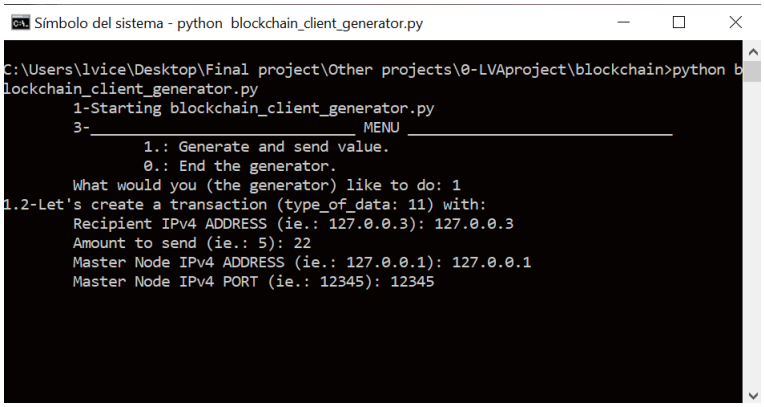

Figure 3.11: The Generator menu

In addition, the generator can be used in the advanced configuration to send amounts of energy generated by the clients to the clients. This is as simple as displayed and of course, nobody lose value due to this energy was generated (not sent) like using a solar panel. In the practice, the sender is named "SUN" and this user does not lose value when sends an amount. This generator is shown on the Figure 3.11 where it can be seen its menu.
As it was said this part of the project should be controlled on a different project to ensure no problems with the electricity meter.

It has been too measured the times in different sections. This performance tests has been done with the time.time function from time library. As it is said on this library: "Return the time in seconds since the epoch as a floating point number". Following this, it is really easy to measure the time between two events by calculating the difference between two measures. Notice that when the elapse of time between two events is less than a minimum of time, it is said that the duration is 0,0.

It has been tried to measure only the computing time so the experiment has been done in loop-back directions on the same computer. To increase the exactitude, it was tried in two different computers (a laptop and a more powerful desktop computer) and the times were really similar due to the simplicity of the calculus (the hardest parts of the code are avoid to be repeated when possible). Of course, the measures had been repeated multiple times (at least, ten times each option or section in each computer).

This study shows that:

- In the Clients (blockchain_client.py code):

  It has been measured the chunk of code before the infinite loop that executes the menu to request for an action. This code prints, parses an argument from command line and generates a private-public key tuple. This sections is computed very fast between 0,160996913 and 0,255102634.

  The other part is the menu. Depending on the option selected, there are four (five if it is counted to introduce and erroneous input argument) different options.

  The first (and fastest) option is to ask the program to print the PK that consumes between 0,0 (in case it is not the first time to be requested) and 0,001996278.

  The second option is to create a new transaction and send it to a MN, this is the opposed case and it is needed a lot of time depending on the user (it is needed the user to write the additional arguments, easily around 5-10 seconds). If it is not counted the time needed by the user to write the additional information, it is needed between 0,723068237 and 2.441816329 (in case it is needed the receiver MN to propagate to lots of MNs). Of course, the more MNs in the list to propagate, the more time needed but it could be reduced by introducing parallel programming on this task (create threads to find MNs on the list and send the transaction, one by one but in a parallel way).

  The third option is to ask for this client's current balance to a MN. This task requires between 0,310797366 and 0,864660891.

  There are two more options, to ask the Client to finish the program and to introduce an erroneous argument. Both of them requires 0,0.

- In the Generators (blockchain_client_generator.py code):

  It has been measured the chunk of code before the infinite loop that executes the menu to request for an action again. In this case the code parses input arguments and prints two times. It is needed 0,0 to resolve this simple tasks.

  On the other hand, the time measured on the menu loop is bigger. Depending on the decided option, if it is required to generate value it is required between 1,247811532 and 1,528101241.

  There are two more options again with "zero" (0,0) time-cost like introduce and erroneous argument and ask the program to finish.

- In the Master Nodes (blockchain.py code):

  There are so many chunks of code analyzed in this code but some of them can be grouped. This similar chunks of code are: the part of code in the main before the loop to request the user how many transactions will be accepted before start mining (a part that depends on the user's reaction time), the one in the main before the loop used to listen for socket connections and the part of the main that fills from the socket creation to the thread start. All of them have really similar and negligible times around 0,0 (the maximum recorded elapse of time was 0,006507158). This is because this codes does not use socket functions. In these group of sections, it is printed messages on the console, gotten the arguments from the parser, initialized the dictionaries, get and put pickles from/on dictionaries, prepared and started the treads and taken and released the locks.

  After this, it mus be talked about the times in the propagate_transaction function. It has been measured times between 0,000997543 and 0,515519142. Remember, this function connects the MN with another MN and sends a transaction.

  The other function is connect_with_mn. This function tries to connect with a MN and in case it is successful, it will be added the IP and port address to a list of connected MNs. The time needed to do this task varies between 0,0 and 0,00769639. It is a really simple task that was encapsulated in a function just to make clearer the code.

  The most important part of this test is the threaded function. To study the elapses of time on this function it is needed to difference the types of data received. In case of a ToD 11 (Transaction), it can be needed to print, get values from lists and compare it, it is possible to need to verify a signature, send messages and propagate. Due to all the just explained tasks, it is needed between 0,015621423 and 0,550095319. In case the ToD is 21 (Client requesting for its balance), 31 (MN connecting with this MN) or 12 (receiving an already mined transaction), the elapse of time needed are really similar and comes from 0,0 to 0,002990484 because the tasks to done are simple.

  The mining time is curiously on of the fastest parts of the code independently of how many transactions are ready to be mined or added to the Blockchain. This is because the code has been carefully done to improve the efficiency and avoid the program block here for so much time. Due to it is a recursive if-else nested section that compares in dictionaries and prints (the part of the code with higher time cost and the one that can be commented), the time varies from 0,0 to 0,0009987537932 (this time was measured with the prints and seventeen transactions ready to be mined).

In conclusion, it can be said that the performance of this Blockchain network depends on the time needed to connect two nodes (sockets).

In adition, the Python IDE used to develop this project (PyCharm from IntelliJ) includes the option to test the performance of the programs. It has been tested the number of calls and the time needed in each section of the program that is going to be explained below.

As it is shown on the Figure 3.4, in this nested architecture, there are four blocks needed by the "blockchain.py" program and an independent block (the threads).

The process is working on the Main around the 99.8% of the execution time. This is due to the main process spends the most of the time waiting for new connections request on the socket.accept call already explained. This is the reason to do not delegate the listening

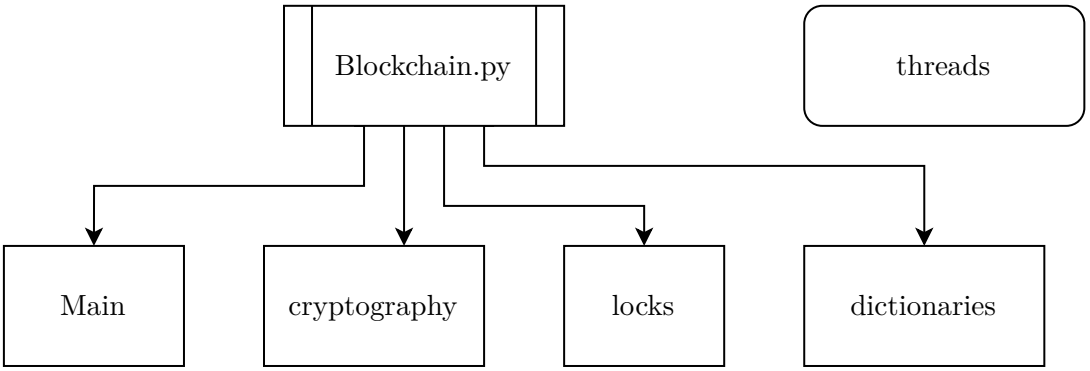

Figure 3.12: Blockchain performance test

action to a thread (in the same way it is done once a transaction comes). In case the number of requests for connection increase a lot (imagine a network with millions of nodes), it could be useful to implement too a thread function for the listening stuff.

The threads block represents the tasks done by the threaded function. This block is not connected with the rest because it is done by different threads, so this work does not requires execution time from the main process. In case this staff was not delegated to other threads, this jobs time could be superior to the Main function if there are so many transaction.

The third block to discuss about is the cryptography where the main process spends around 0,1% of its time. This block represents all the functions working with the SHA256, RSA, hashes and similars. In example, here it is measured the time to create the keys, do and verify the signatures or to create hashes. Curiously, call needed to raise the power of a number (___pow___) is the one that spends the more time in my computers. This is probably due to the multiply and addition function can be done faster than a loop of multiplications on the general purpose CPUs.

In the other hand, there are two blocks dealing with the locks (the control of the critical sections access) and the dictionaries (the system to organize the information). Both of them have a similar time needed in this functions and calls, around the 0,05%.

In case the number of received transactions increased, the time needed on the locks' block could be higher but anyway it would be similar to the cryptography's block. The locks block deals with the creation of this semaphores (locks), and its acquisition and release. The higher measured time is the needed to acquire the lock (the call named "_load_unlock"). This is because it is needed to wait for other threads releasing it and it is used both in the ".acquire" and the ".join" calls.

In the other hand, the dictionaries block create this ordered lists, gets value from them, sets changes and puts new values with the peculiar key-value system. The most required action in this block is to create a new dictionary. This is why, it is avoided to create dictionaries out of the initial pre-protocol.

Similarly, the Client's ("blockchain_client.py") diagram is shown on the Figure 3.5.

In this program, there are not threads neither locks or dictionaries so the scheme is simpler. The measured times variate a lot on the user because it depends on how fast the user choose new actions to do or how much time the client spends waiting for a new task.

Supposing a fast human selection, the measured times indicates that the Clients is in the main block between the 99.6 and the 99.9% of the time. Due to the most of the tries,

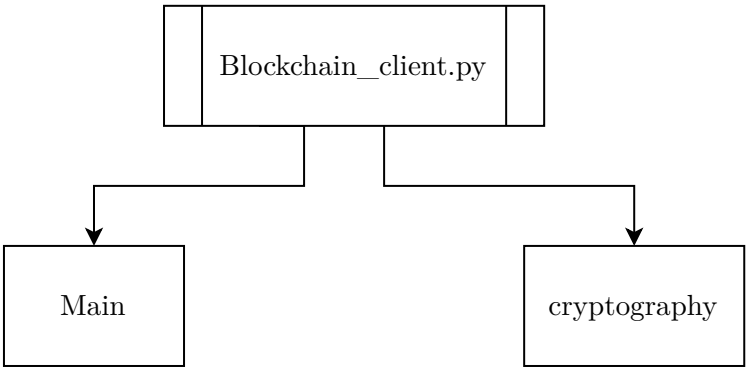

Figure 3.13: Client performance test

it was measured 99.7% it will be chosen this data to compare the rest of blocks times. In similar way to the MN main, the Client main's most expensive (in terms of time) action is to connect a socket with another one (in example, when a client needs to send a transaction to a MN).

In the other hand, the Client needs too so create the private and public keys and does not need to verify signature but to create it (a bit more expensive). Due to this, the Client usually spends more percentage of its process time in the cryptography block. Again, the "___pow___" call is the one that spends the most of the cryptography block time.

Finally, the Generator variant of a client is even simpler as explained is shown on the Figure 3.6.

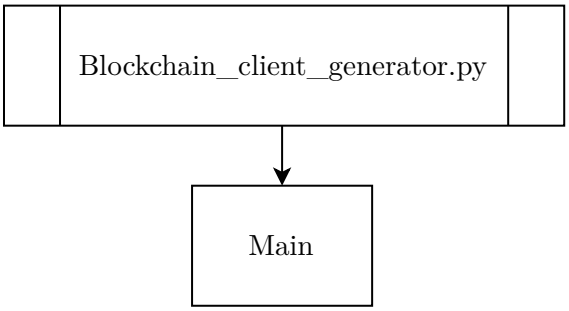

Figure 3.14: Generator performance test

This function focus its main in waiting for the user (or the electricity meter) and when it is active, it connects with a MN socket-to-socket and sends a transaction. In this case, it con be a process in stand by (it is not needed to measure the generation of electricity in every clock cycle). Of course, the most expensive action is to connect with a MN's socket. The percentage of time in this block is not relevant due to its total dependence on the user and how it is measured the generated electricity.

Chapter 4: Conclusions summary

All in all, this thesis lay the foundations for future Blockchain projects. These projects can take all the code and try new technologies and protocols like different cipher protocols, or to take parts of it and create new Blockchain networks to compare them for different purposes. Some of these future projects could be:

- To change the key "transaction_date" which is saved the transactions when received for a tuple, in example it could be utilized the tuple "transaction_date" plus "sender_address".

  This is because of accepting the possibility of receiving more than a transaction generated at the same time if the network is big enough.

- To change dynamically the mining_difficulty (the variable "n" is statically selected on the start of every MN in the code) if it is supposed that the Blockchain network is going to grow up.

  In this way, it could be solved the problem of need to wait for a transaction mined before coming from far. Remember that the further (in number of nodes) a MN is from another one, the more time that can be needed to reach it and propagate a transaction.

- To implement a protocol to improve the efficiency of connections in the network. This would make more efficient the propagation.

  In example, a algorithm to choose the connected MNs in each MN (and even the clients) depending on the delay between them or a load balancing algorithm to avoid a MN with 20 connected MNs and another one with only 1.

- Following the line of the tuples, implement a tuple for the MNs and clients when saved on the list.

  In example, a tuple with the PK would allow to add the signature in every transaction if considered. Or implement a tuple with the date of last activity and clean the list of connected MNs periodically to avoid saving inactive nodes. This secure mood could be too an option to the user.

- Definitely, one of the main ideas of the code is to implement the parallelization in future versions by using threads.

  This is why the code is already prepared for it with locks, queues to communicate the threads and a threaded function to deal with the received transactions and it was explained. In this way, the main thread of blockchain.py (the MN) would focus on listening transactions and every transaction would be dealt by a thread of this father

MN. This concurrent system reduces the time needed between transactions received to a clock cycle of the computer running the process.

Chapter 5: List of lists

List of Abbreviations

- P2P - Peer To Peer
- AI - Artificial Intelligence
- DLT - Distributed Ledger Technologies
- ECDSA - Elliptic Curve Digital Signature Algorithm
- ECC - Elliptic Curve Cryptography
- PKC - Public Key Cryptography or Cipher
- PK - Public Key
- W-OTS - Winternitz One Time Signature
- SHA256 - Secure hash Algorithm of 256 bits
- RSA - Rivest, Shamir and Addleman protocol for the Public Keys
- DER - Distributed Energy Resources
- RFC - Request For Comment
- TOD - Type Of Data
- MN - Master Node
- ACK - Acknowledge type of message frequently used on the TCP systems
- TCP - Transmission Control Protocol
- IP - Internet Protocol address
- IDE - Integrated Development Environment
- CPU - Central Processing Unit

List of Figures

Bibliography

[1] ALI, Z. *Explaining the Math Behind Blockchain Algorithms* [online]. 2019. Available at: https://medium.com/dataseries/explaining-the-math-behind-blockchain-algorithms-98d06e06c2e3#:~:text=Elliptical%20curve%20cryptography%20is%20a,this%20scheme%20are%20not%20trivial.

[2] ANTONOPOULOS, A. *Mastering Bitcoin*. 2nd ed. O'Reilly, 2017. ISBN 978-1-491-954386-6.

[3] DUNPHY, P. and PETITCOLAS, F. *A First Look at Identity Management Schemes on the Blockchain* [online]. 20. Available at: https://ieeexplore.ieee.org/document/8425607.

[4] ELBANSARKHANI, R., GEIHS, M. and BUCHMANN, J. *PQChain: Strategic Design Decisions for Distributed Ledger Technologies against Future Threats* [online]. 2018. Available at: https://ieeexplore.ieee.org/document/8425622.

[5] GROETSEMA, A., GROETSEMA, A., SAHDEV, N., SALAMI, N. et al. *Course Introduction to Hyperledger Blockchain Technologies from The Linux Foundation* [online]. Available at: https://www.edx.org/course/introduction-to-hyperledger-blockchain-technologie.

[6] GUERRERO, J., CHAPMAN, A. and VERBIC, G. *Decentralized P2P Energy Trading under Network Constraints in a Low-Voltage Network* [online]. 2018. Available at: https://www.researchgate.net/publication/327763558_Decentralized_P2P_Energy_Trading_under_Network_Constraints_in_a_Low-Voltage_Network.

[7] GUPTA, A. *Introduction to Blockchain technology | Set 1* [online]. Available at: https://www.geeksforgeeks.org/blockchain-technology-introduction/.

[8] JIMENEZ, P. *Introduccion a Blockchain: ¿Que es y como ha evolucionado?* [online]. Available at: https://www.techedgegroup.com/es/blog/introduccion-blockchain-explicacion-evolucion.

[9] KARAME, G. and CAPKUN, S. *Blockchain Security and Privacy* [online]. 2018. Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8425621.

[10] LIN, R. and AKHTAR, N. *Course Learn the fundamentals of blockchain technology and how it will power the economy of tomorrow from Berkeley University of California* [online]. Available at: https://www.edx.org/course/blockchain-technology.

[11] Maranon, G. *El futuro post-cuantico esta a la vuelta de la esquina y aun no estamos preparados* [online]. 2019. Available at: https://empresas.blogthinkbig.com/futuro-post-cuantico-ciberseguridad/.

[12] Meiklejohn, S. *Top Ten Obstacles along Distributed Ledgers Path to Adoption* [online]. 2018. Available at: https://ieeexplore.ieee.org/document/8425611.

[13] Nakamoto, S. *Bitcoin: A Peer-to-Peer Electronic Cash System* [online]. 2008. Available at: https://bitcoin.org/bitcoin.pdf.

[14] Naziridis, N. *Comparing ECDSA vs RSA* [online]. 2018. Available at: https://www.ssl.com/article/comparing-ecdsa-vs-rsa/.

[15] Pastor, J. *Que es Blockchain: la explicacion definitiva para la tecnologia mas de moda* [online]. Available at: https://www.xataka.com/especiales/que-es-blockchain-la-explicacion-definitiva-para-la-tecnologia-mas-de-moda.

[16] Sanderson, G. *But how does bitcoin actually work?* [online]. Available at: https://www.youtube.com/watch?v=bBC-nXj3Ng4.