Public Ledger for Auctions- An Implementation Guide

Carlos Miguel Ramôa
Science Faculty
University of Porto
Porto, Portugal
up201605785@edu.fc.up.pt

Marco Augusto Primo
Science Faculty
University of Porto
Porto, Portugal
up201800388@edu.fc.up.pt

Paulo Araújo
Science Faculty
University of Porto
Porto, Portugal
up201805060@edu.fc.up.pt

***Abstract*—We describe a public auction ledger that was proposed to us in our Security in Data Systems class. Our public auction ledger uses a blockchain to keep track of auctions, bids and sells, we then use a decentralised peer-to-peer network for bidders and sellers to communicate with each other.**

I. Introduction

Public ledger used in cryptocurrency,for instance BitCoin [6],has become a huge attraction for people looking for secure and anonymous transactions. Thus, it was proposed by the Mr. Dr. Rolando Martis [4] regent teacher of Security and Data Systems chair, the implementation of a public ledger resistant to Sybil and Eclipse attacks. There was some requirement imposed for the implementation. These requirements is listed in the in VIII-B.

The following texts of this report are organized into sections, these sections can be summarized as follows. Section II describe the general proposed architecture for this implementation. Section III briefly describes what a BlockChain is, followed by a detailed explanation of its implementation. Section IV describes what Kademlia DHT is and how it was implemented. Section describes the general communication system used by the nodes. Section VI summarizes what are the avoid attacks in this implementation. Section VII is a guide in the graphical interface for user uses. And last, but not least, Section VIII describes the difficulties encountered when implementing this public ledger and a personal opinion of the members of this group on this implementation.

II. Architecture

The architecture adopted for this implementation can be summarized as follows.

Initially there is a node known as GreetingHost. This node is special as this is the first node in the P2P network. This node is always listening on port 2000 for UDP datagrams.

Other nodes that want to join the network must send a `FIND_NODE` request with their own Key to GreetingHost. So GreetingHost becomes aware of this new node and forwards this message to nodes in the same bucket as the requester. At the end of this process, the node requester is known in the P2P network. For the dynamic configuration of which census should be used, GreetingHost sends an initial configuration describing whether the census will be proof-of-work or proof-of-stake. That way, after this configuration, the requester node can participate in hashing races.

For the prevention of Sybil attacks the amount of energy expended and processing power is high when the census is associated as prof-of-work. A hash with N bits set to 0 is required to hash the new block.

For Eclipse prevention, incoming connections from new nodes are restricted in the routing table. That is, there is a limit to the number of nodes in each node's bucket. This limitation makes it so that when the limit of a bucket is reached and when a new node is added, this new node will only be added if an older one fails to respond to a message silanizing its alive state.

When some action is performed, it becomes a block in the chain. Actions that are transformed into blocks are: creation of a sell-off for items, subscribing for a certain item interest, a node bid for an item being auctioned, and ending the transaction when the auction is closed. In this way, it is possible to have the entire history of actions of the nodes participating in the network.

To avoid non-repudiation of a node. The node that performs an action executes a digital signature on the internal block 1 that will be mined. This procedure prevents attacks from nodes on the network that try, for example, to bid on the names of other nodes.

When there is a block to be mined, the first node to finish will send a `blockFisnihed` message to the entire network. Nodes that receive this message request that a voting process is started to determine which block won the race. This voting occurs because there may be two or more nodes that have finished mining the block at very close time intervals. Thus, each node in the network votes for the block that it considers being the winner of the race. When this voting is finished, the winning block is published on the network and all nodes consider this as the last valid block. Although this election procedure is not exactly the Byzantine fault tolerance algorithm, it can easily be replaced by.

III. BLOCKCHAIN

A. *What is*

Blockchain is a shared and immutable record **??**. The immutability associated with this data structure is due to a hash that guarantees the integrity of the stored data. In fact, without cryptographic primitives like the one-way hash function, in this case, it would not be possible to guarantee integrity. Today, this data structure is used in cryptocurrencies to facilitate the process of recording transactions and tracking assets in a business network.

In this work, the authors make use of this component - Blockchain - to save the integrity and nonrepudiation. To achieve integrity as already mentioned the blockchain has a hash over its content. In this public ledger implementation, it was used an SHA-256 hash function that ensures a 128bits security bit-level. Regarding non-repudiation, each message is signed by the owner before being integrated into the blockchain. A digital signature is also based on public-key cryptographic primitives.

Figure 1 clarifies what a block is in the authors' public ledger implementation. And Figure III describes how it is turned into a blockchain.

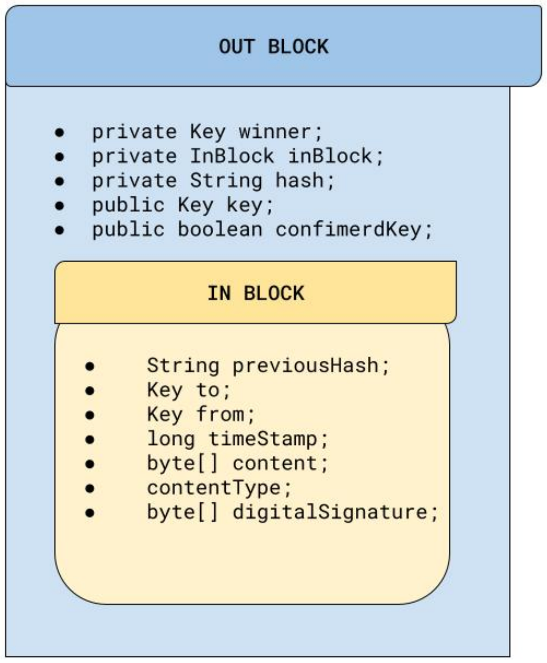


Fig. 1. Block representation with its properties

B. *Implementation*

Block implementation is trivial. The block in this public ledger implementation is a java class that contains the properties as figure 1 exposes. However, what is relevant to mention is how a block is mined.

To mine a block when using proof-of-work is required that the first N bits of the current block hash is 0. For instance, let X be the hash for the last block in the chain, to mine the new block is required that this new block includes hash X and this block is hashed with 0's prefix, this new hash has to be something such: `0006f7cbe59e99a90b5cef6f94f966fd`

In code, this requirement is transformed into a function that uses a nonce variable, initially with a value of 0, which is incremented each cycle and built into the block to find the proper hash.

IV. KADEMLIA OVERVIEW

It is not possible to introduce kademlia without talking about DHT first. Therefore, a brief explanation of DHT is given below

A. *Distributed Hash Table*

A distributed hash table - DHT - is a system that stores data in a decentralized fashion. Mostly DHT provide hash-based, something similar to a hash-table with the big difference that this hash table is spread by N nodes instead of local. A DHT-based system has its search and storage based on keys. In general, each node belonging to the DHT has been assigned a key, and this key determines which other keys belong to values this node is responsible for storing.

B. *Kamdemilia*

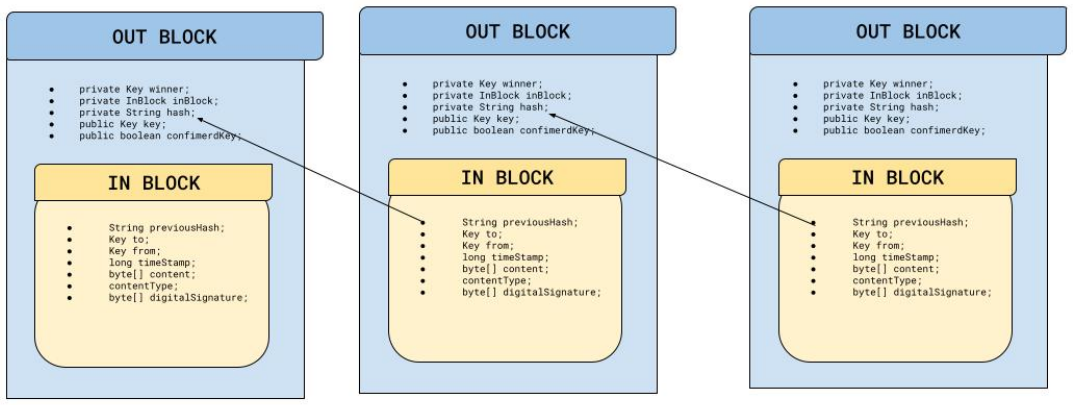
Kademlia is a Distributed Hash Table distributed for peer-to-peer (P2P) networks. Since it is a distributed system, there is no singular point of failure.

Kademlia is a DHT algorithm. This algorithm specifies the structure of a network for storing and retrieving data based on the nodes that constitute it. The nodes in this DHT communicate based on datagrams. In kamdelia, as in other DHTs, each member of this system is identified by a unique key number. It is important to note that both nodes and values in the Kademlia are identified by keys. Keys in kamdelia are 160-bits, thefore the address space is 2^{160}.

Regarding the search for a value, it requires prior knowledge of the associated key and happens as follows. The node that starts the search calculates its distance from the target key. With this information, the node checks in its routing table which is the closest to this key and makes the request for that closest one. In turn, the requested nodes proceeds exactly as the requester node. This behavior is repeated until the request arrives at the node that locally store the data that is identified with the key initially sought. In this way, each iteration of the algorithm will find nodes closest to the key. The Big O notation for searching and storing in kademlia is $O(\log n)$.

To insert values, the procedure is similar, however, together with the key, the value to save is sent as well.

Although the description above looks simple, kademlia is not so simple to implement despite its simple searches and insertions. In fact, for the ecological system to maintain itself, it is necessary to incorporate the notion of a routine table as well as the reallocation of data according to the knowledge of new nodes. Therefore the authors use as main reference this this [2], that contains both a visual and iterative description of the algorithm.



```
•   private Key winner;
•   private InBlock inBlock;
•   private String hash;
•   public Key key;
•   public boolean confimerdKey;

    •   String previousHash;
    •   Key to;
    •   Key from;
    •   long timeStamp;
    •   byte[] content;
    •   contentType;
    •   byte[] digitalSignature;
```

```
•   private Key winner;
•   private InBlock inBlock;
•   private String hash;
•   public Key key;
•   public boolean confimerdKey;

    •   String previousHash;
    •   Key to;
    •   Key from;
    •   long timeStamp;
    •   byte[] content;
    •   contentType;
    •   byte[] digitalSignature;
```

```
•   private Key winner;
•   private InBlock inBlock;
•   private String hash;
•   public Key key;
•   public boolean confimerdKey;

    •   String previousHash;
    •   Key to;
    •   Key from;
    •   long timeStamp;
    •   byte[] content;
    •   contentType;
    •   byte[] digitalSignature;
```

Fig. 2. Chain of Blocks. Where each block has a previous hash to the ancestor block

C. *Kademlia implementation*

In code, the components needed to build a kademlia are broken down into classes. It is in the authors' interest to give a brief explanation of each of these classes, so the sections below comment on each class.

1) Key.java: Key.java classes are the most basic component needed for kademlia to work, both values are identified by it. This class builds a key based on a `Key_length` - which in this case was specified as 160 -. This class, in addition to representing the Key, also has two very important methods, namely:

1) `getFirstSetBitIndex`
    a) This method counts how many digits there are in common between the context key and the parameter key
2) `xor`
    a) The Xor method is an overload function for the existing Xor method in the `BigInteger` class

Therefore a distance for a Key to any other Key can be done using this expression `Key_LENGTH - this.xor(to).getFirstSetBitIndex();`

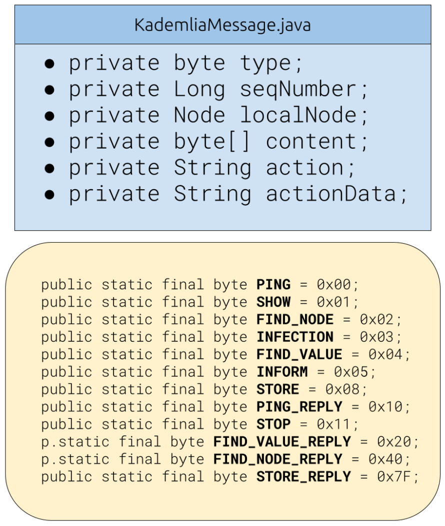
2) Bucket.java: This class as specified in the kademlia workflow is responsible for storing Keys. However, the Keys stored in this component are exclusively for nodes. Also, a bucket has a capacity limit - in this case, 20 was specified -. Important method of this class is:

1) `addNode`
    a) This method is responsible for adding a new node to the bucket. However, this is a simple task to perform until capacity is reached. However, when the bucket is populated, logic must exist to contact the oldest node in the bucket to check for life activity, and if that contacted node does not respond, the new node takes its place. Otherwise, the subsequent node is contacted and the process repeats.

3) RoutingTable.java: This class is the order routing table for its node. Through this table, the node knows which other node to send a request to, be it a value lookup or a request to store an item.

So this class holds N buckets and each bucket holds M nodes. The value of N is equal to the number of bits used for addressing and the value of M is the number specified when creating a bucket. Kademlia suggests an N = 160 and M = 20.

4) KademliaMessage.java: To convey information it is important to establish a well-known language. This is achieved with this specific implementation through the use of the KademliaMessage class as a communication mechanism. When nodes want to communicate, first an instance of this class is created, followed by its serialization, and finally by its submission. Figure 3 pictures what properties this class holds and what types of messages currently exist.



```
• private byte type;
• private Long seqNumber;
• private Node localNode;
• private byte[] content;
• private String action;
• private String actionData;
```

```
public static final byte PING = 0x00;
public static final byte SHOW = 0x01;
public static final byte FIND_NODE = 0x02;
public static final byte INFECTION = 0x03;
public static final byte FIND_VALUE = 0x04;
public static final byte INFORM = 0x05;
public static final byte STORE = 0x08;
public static final byte PING_REPLY = 0x10;
public static final byte STOP = 0x11;
p.static final byte FIND_VALUE_REPLY = 0x20;
p.static final byte FIND_NODE_REPLY = 0x40;
public static final byte STORE_REPLY = 0x7F;
```

Fig. 3. KademliaMessage with its properties

V. Communication Interface

A. *Messages Dynamics*

It was proposed in the classes to use the GRPC [7] to establish a communication channel between the nodes. However, GRPC flows under TCP, and in the original Kademlia article [5] the author makes use of datagrams to convey messages. So, following the original article, this implementation does not use GRPC. However, socket programming was extensively explored in this work. The socket library for java - java.nio library [1] - was used to implement the communication channels.

To allow the reception of messages issued by other nodes there is a thread listening on a port through a UDP socket. Therfore when a message when received and if it is identifies that this message is ready to be processed the thread handles this message. In the code, this thread is called `postOffice`. The handling of a message by `postOffcie` is done based on the type attribute of the serialized KademliaMessage recieved.

The process of sending a message to some other node is done using a DatagramChannel with the address of the destination node.

To clarify this communication process the authors given a visual description in Figure 4.

B. Publication and Subscription over Kademlia

A mandatory requirement for this auction using public ledger implementation is the fact that each publication of an item for auction or subscription to an item tag must be submitted using node knowledge acquired by the kademlia framework. Therefore a new message type has been added to the possible types that a KademliaMessage can assume. Figure 3 all types implemented.

The new type implemented to support this requirement is the `INFECTION` message type. `INFECTION` messages when received by a node, are processed and forwarded to all known nodes regarding the actual context. The processing of a message of type `INFECTION` is intrinsically related to its content. Unlike other KademliaMessages, the content of a KademliaMessage of type `INFECTION` is an InfectionMessage.

InfectionMessage is a message that can only be transmitted in the body of a KademliaMessage. This message has its properties illustrated in Figure 5.

VI. Security Mechanisms

A. Sybil Attack

One of the attacks that can affect the network is the Sybil Attack. This is a curious vulnerability that can have a big impact on blockchain networks, by allowing the attacker to have a greater presence on the network by obtaining false identities. A Sybill attack occurs when a system is breached by an entity that controls two or more different identities on a network. That is, when a person controls two or more nodes that should belong to different people or identities. It is an attack where a user tries to gain control of the network by creating multiple accounts, nodes or computers, which he owns. All this while trying to show the network that each of its points is a different identity, to avoid raising suspicion that a Sybil attack is taking place.

B. Impact of Sybil attack

The main impact of a Sybil Attack on a blockchain network mainly focuses on gaining influence on decisions made on the network. For this, the user creates and controls several ”pseudonyms” that allow him to gain that influence and as a result, the malicious agent gains disproportionate control, compared to other agents, over the decisions made on the network. This control can become dangerous, for example in the case of a vote on a given decision in the network, the malicious agent can influence the result of that vote by the simple fact of controlling more agents (influences the outcome of the election).

C. Protection measurements against sybil attacks

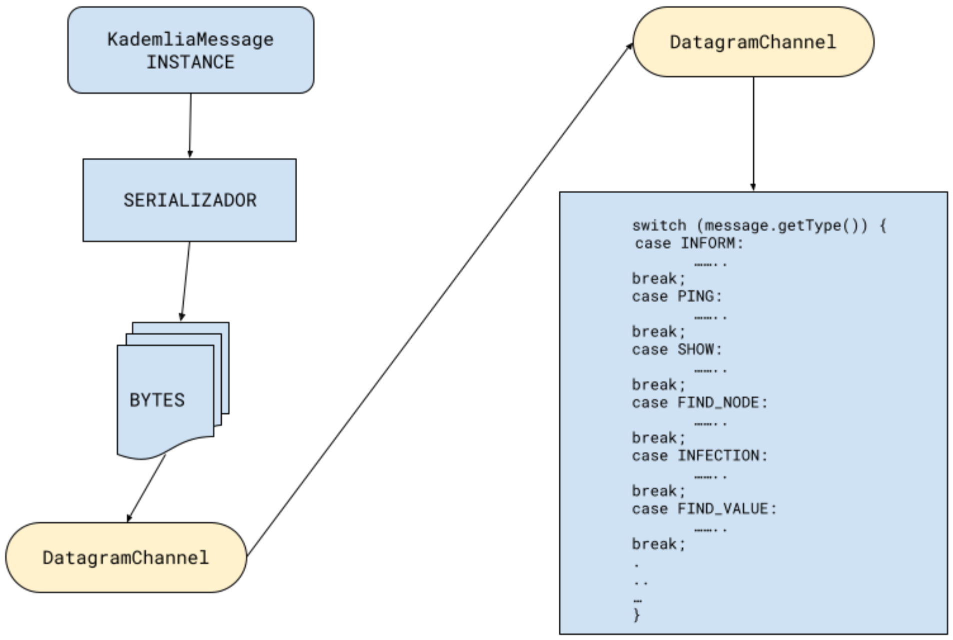
One of the most important measures at the moment is the Chain of Trust. In Bitcoin, for example, the blockchain and its transaction history are distributed across all of its nodes. They all have the same Ledger, and if only one of them tries to change it, that change will simply be rejected. So, when a node starts synchronizing on the network, it receives data from various sources. It gathers everyone's information, and if any of the nodes tries to change the data in any way, it is simply rejected and an attempt is made to get the data from another node that is trusted.

Sybil attacks are not something users can deal with or even prevent. In fact, measures to be taken are responsibility of P2P (peer-to-peer) web developers. The use of consensus protocols that incur a cost of identity or access to network resources. In this way, any action carried out on the network would have an associated cost and this would multiply in proportion to the identities usurped. While it doesn't stop the Sybil attack, making it expensive limits its potential.

D. Eclypse Attack

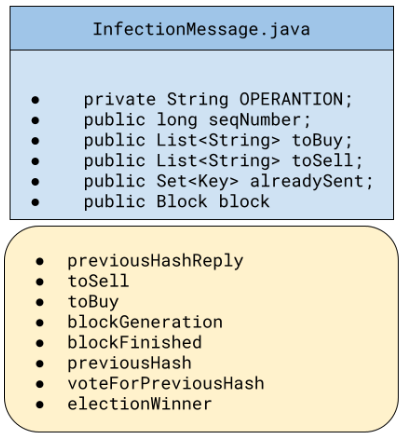
Eclipse attacks are a special type of cyberattack where an attacker creates an artificial environment around one node, or user, which allows the attacker to manipulate the affected node into wrongful action. By isolating a target node from its legitimate neighboring nodes, eclipse attacks can produce illegitimate transaction confirmations, among other effects on the network. While these types of attacks isolate individual nodes, the effectiveness of eclipse attacks at disrupting network nodes and traffic largely depends on the structure of the underlying network itself. Eclipse attacks are extremely rare in the real world; the structure of a decentralized blockchain itself tends to detect them and prevent it from causing any damage.

As stated before, Eclipse attacks involve a malicious actor isolating a specific user or node within a peer-to-peer (P2P) network. When executing an eclipse attack, the attacker attempts to redirect the target user's inbound and outbound connections away from its legitimate neighboring nodes to attacker-controlled nodes, thereby sealing off the target in an environment that's entirely separate from the actual network activity. By obfuscating the legitimate current state of the blockchain ledger, the attacker can manipulate the isolated node in various ways that can lead to illegitimate transaction



```
switch (message.getType()) {
 case INFORM:
        ……..
break;
case PING:
        ……..
break;
case SHOW:
        ……..
break;
case FIND_NODE:
        ……..
break;
case INFECTION:
        ……..
break;
case FIND_VALUE:
        ……..
break;
.
..
…
}
```

Fig. 4. Nodes communication interface in p2p network [3]



```
•    private String OPERANTION;
•    public long seqNumber;
•    public List<String> toBuy;
•    public List<String> toSell;
•    public Set<Key> alreadySent;
•    public Block block
```

```
•  previousHashReply
•  toSell
•  toBuy
•  blockGeneration
•  blockFinished
•  previousHash
•  voteForPreviousHash
•  electionWinner
```

Fig. 5. InfectionMessage with its properties

confirmations and block mining disruptions. Because eclipse attacks rely on exploiting a target's neighboring nodes, the ease with which these attacks can be successfully executed depends largely on the underlying structure of the target blockchain network. While the decentralized architecture of most cryptocurrency protocols makes them relatively difficult (and relatively rare) compared to other types of online attacks.

VII. Graphical User Interface

In addition to the functional requirements imposed by Mr Dr. Rolando Martins. A graphical interface was built to facilitate user interaction with the public ledger-based auction. This graphical interface allows subscribing to the category of an item, allows item publishing, and also through this interface it is possible to bid for published items that are known in the current context. In addition to all these more complex features, simpler features are available such as listing the rotation table, listing the values stored in the current context, listing the interests of the current context, listing the external interests as well as listing the intersection of interests. Figure X

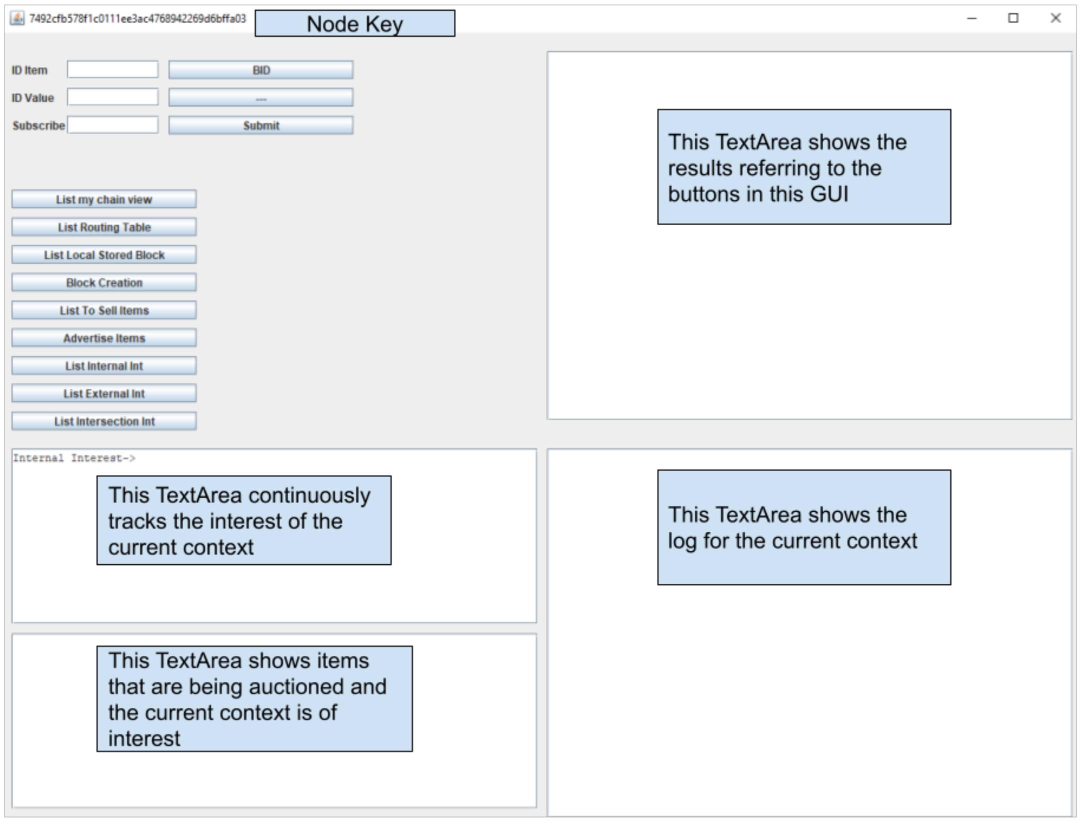
VIII. Conclusion

During the development of this work, a lot of preventive safety methods were learned. The development of an application thinking about security solutions for possible attacks had never been performed by the members of this group. And in fact, experience in building software with this mindset is judged very important by the members of this group, as security and privacy are increasingly demanding.

A. Faced Difficulties

Regarding the functional requirement difficulties faced during the development of this work, one that stands out is the fact of complexity required to complete this work. The members of this group since the beginning of the semester have tried to manage their time well so that nothing gets delayed. However, even with all this care, it seemed that there was no time to finish. Another significant difficulty that can be highlighted is the choice of using Socket Programming. This choice leads to the extensive use of threads to handle multiple requests. Thus, the code of this implementation ended up being a little complex and having a difficult debugging.

Another non-functional requirement that led to some difficulty was setting up an initial node using google cloud. The procedures for creating a VM and releasing ports on the firewall were done in accordance with the google guide. However when a second node on another network - that other


7492cfb578f1c0111ee3ac4768942269d6bffa03
Node Key
ID Item
ID Value
Subscribe
BID

Submit
List my chain view
List Routing Table
List Local Stored Block
Block Creation
List To Sell Items
Advertise Items
List Internal Int
List External Int
List Intersection Int
Internal Interest->
This TextArea shows the results referring to the buttons in this GUI
This TextArea continuously tracks the interest of the current context
This TextArea shows the log for the current context
This TextArea shows items that are being auctioned and the current context is of interest

Fig. 6. Graphical Interface

network is a vodafone home network that is behind a nat - tries to contact the initial node, it does not send any response signal. WireShark was then used to check for possible communication problems and it was found that the datagram sent by the node on the local network was not answered either with another datagram or with ICMP messages reporting some kind of error.

However, as the days went by, ideas emerged about what might not be working. And then it was discovered what was the cause of the problem. The problem was that the nodes were using a different outgoing port than the incoming port, so when the packet arrived at the VM in the cloud, they would respond to the outgoing IP and not the IP that the nodes were listening to. Figure 7 illustrates this event. The correct path for a packet flow is illustrated by ports in green and the wrong path is illustrated by red ports.

Another solution that emerged was the creation of several VM where the program used by the user would be kept. Because in order to run the code on the local machine, it is necessary to create a forwarding route, if there is a `NAT`.

B. Implementation requirements achieved

To summarize what was and what was not implemented, Table VIII-B was inserted for this purpose.

Requirements	Status
Proof-of-Work	Achieved
Proof-of-Stake	Parcial Achieved
Per-missioned blockchain	Achieved
Implement Kademlia	Achieved
Resistance to Sybil attack	Achieved
Resistance to Eclipse attack	Achieved
Implement trust mechanisms	Non Achieved
Transactions should be saved in the blockcha	Achieved
Publisher/Subscriber	Achieved

References

[1] Javanio. https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html. Accessed: 2022-05-06.

[2] Kademlia. https://kelseyc18.github.io/kademlia_vis/basics/1/. Accessed: 2022-05-06.

[3] Photodiodo. https://www.teamwavelength.com/photodiode-basics/. Accessed: 2022-05-25.

[4] Rolando martins. https://www.dcc.fc.up.pt/~rmartins/. Accessed: 2022-05-06.

[5] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[6] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[7] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. Grpc: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Operating Systems Review*, 27(3):75–86, 1993.

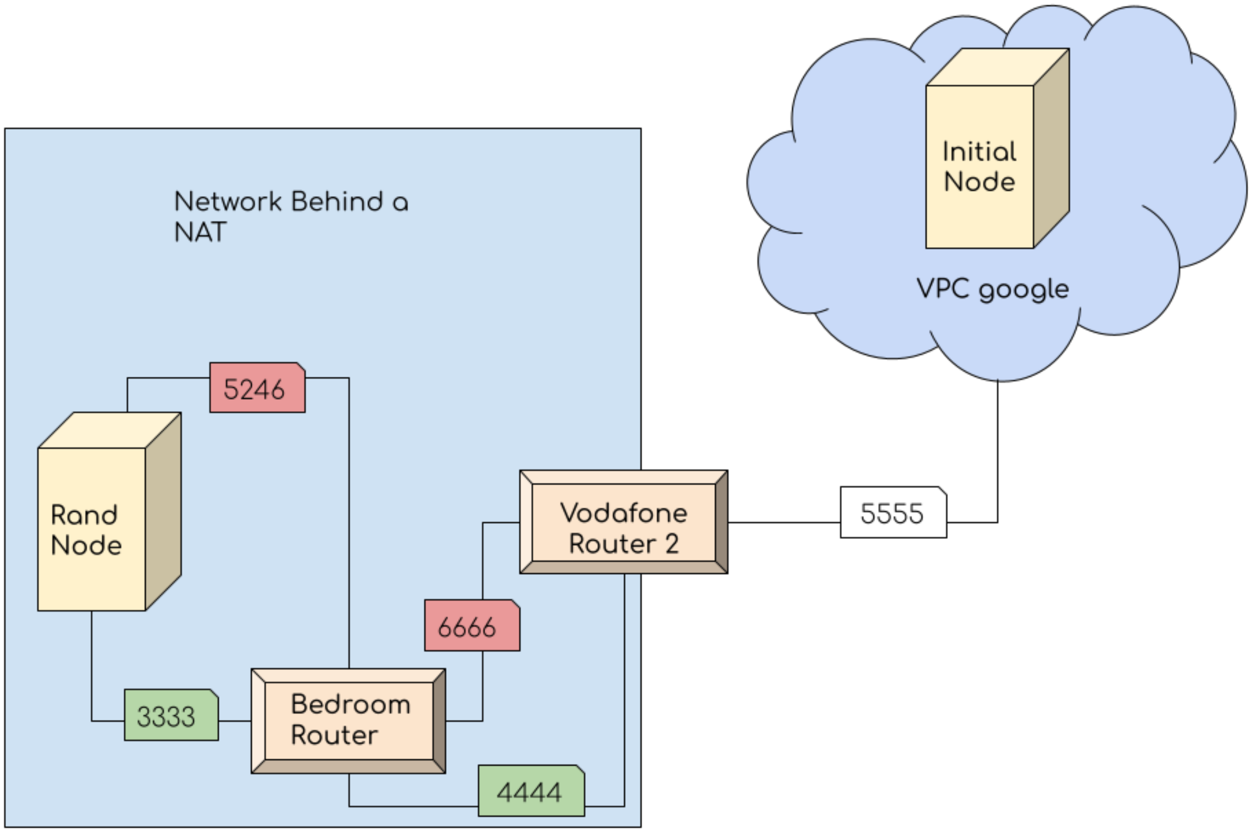


Fig. 7. NAT problem