eman ta zabal zazu

Universidad del País Vasco · Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer science

End of degree work

Distributed integrity monitoring system based on *Blockchain* technology

Author

Aitor Belenguer Rodriguez

2021

eman ta zabal zazu

Universidad del País Vasco · Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer science

End of degree work

Distributed integrity monitoring system based on *Blockchain* technology

Author

Aitor Belenguer Rodriguez

Director(s)

Jose Antonio Pascual Saiz

Summary

As a consequence of digitalization, the number of IoT devices and systems connected to the Internet has grown considerably. Those infrastructures are being constantly targeted by cybercriminals due to the large amount of valuable information they manage or their potential usage as elements of a botnet. Hence, the need to enhance the authenticity and integrity of data has become a great deal. However, the high number of gadgets make the task of monitoring security in real time very challenging. The aim of this project is to design and implement a system that could handle and monitor information of IoT devices, in a distributed, efficient and safe way. To perform it, a backbone interface which implements a P2P communication protocol will be designed. Being some of the concepts used in digital signature and Blockchain technology, the cornerstone of the development.

Contents

List of Figures

List of Tables

1. CHAPTER

Introduction

As a result of continuous technological evolving and readaptation, not securely designed systems are suffering the effects of turning them into smart devices. Although it is very convenient to have 24/7 connected IoT gadgets, a gateway to the internet turns them into a double-edged sword. Default passwords or misconfigurations transform smart systems into weak, unpredictable targets that could be used as local network accessing backdoors.

Moreover, cybercriminals try to get control over misconfigured devices to perform DDoS[1] attacks to thirds. A common example are ISP provided routers, with their default admin/admin credentials; becoming the preferred agents of botnets by antonomasia. Others, such as default credentials of Operating Systems (OS) accounts and databases, let systems to be publicly exposed; opening the doors to information dumping and modification of configuration files. Therefore, integrity preservation and its monitoring is essential, especially in environments where the amount of sensitive data is high; enterprises and corporations.

On the one hand, conventional measures to mitigate network incoming cyberattacks usually involve the use of firewalls; being the application type ones the most sophisticated systems. As part of the active security conglomerate, firewalls analyze incoming datagrams and collate their content against specified rules or information from a database. Although active security is the most important ally of cybersecurity, the effectiveness of firewalls is highly questionable. At the end of the day, threats materialize or mutate into new untracked attacks too quickly, outdating the rules of firewalls. On the other hand, the

[1] https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack

use of antivirus applications as the main pillar of passive security is globally spreaded. Those applications, use as well content collating against a record rows as threat detection main tool. As a consequence, the same fast expiration issues are inherited.

Furthermore, when integrity monitoring systems enter into the scene, it is from the hand of centralized stations. Those stations act as servers that listen to the network and send information request messages to other devices. Depending on the received answer, status of the requested elements is determined SNMP[2]. Additionally, the use of **Blockchain** technology can be a possible solution to status history simplification. Nevertheless, its use usually implies the need of blockchain storing centralized servers. Making Blockchain a not suitable technology in many environments.

The best option for an effective integrity monitoring system is to reorganize the existing technological pieces and mold them into new protocols and applications. An example of it are Blockchain based **distributed** protocols such as **Trebizond** or **Tendermint**; taken as roadmaps to develop the core of this work. In the same thread, this project tries to answer the following questions.

1. Can integrity be monitored in a distributed, secure and reduced resource cost way, using Blockchain technology and digital signature?
2. In the current scenario, are P2P networks the best approach to create a scalable and independent network protocol?

The short answer to the previous questions, based on future work, is **“yes”**.

[2]https://en.wikipedia.org/wiki/Simple_Network_Management_Protocol

2. CHAPTER

The aims of the project

The goal of the project is to create a custom integrity monitoring system based on avant-garde technologies, such as Blockchain and digital signature. Allowing an implementation of a distributed network protocol located into the application layer of the **OSI**[1] model.

As part of the security environments of enterprises, integrity monitoring is usually performed in a centralised and inefficient way. Moreover, it is typical to ensure data integrity reactively, just by hardware redundancy. On the one hand, relying all network monitoring tasks on a single computer could trigger scalability issues as well as become a weak point of the system. On the other hand, the conventional way of sending secure information throughout the network is by establishing a **TLS**[2] connection between the agents and sending a set of files to be analyzed. Those methods increase network traffic and burden communications considerably.

A possible solution could reside in monitoring integrity in a distributed way, sending just the needed data throughout the network. Therefore, no weak points will exist and integrity leaks will quickly be detected due to a scalable peer-to-peer[3] **(P2P)** auditing system. Furthermore, a digest or hash value of the file(s) to be audited could be plainly sent to the network, without needing extra security measures.

A network application protocol will be designed, implemented and tested with the aim of creating a new paradigm that could serve as a starting point of alternative usages to P2P

[1] https://en.wikipedia.org/wiki/OSI_model
[2] https://en.wikipedia.org/wiki/Transport_Layer_Security
[3] https://en.wikipedia.org/wiki/Peer-to-peer

networks and blockchain technology. At the same time, a complementary project at OS level is being developed [Herrero, 2021]. The last project will act as an underlayer application, providing a digest of a designated filesystem by **Merkle Tree**[4] parsing. Hence, an integration stage of the two projects will be carried out as well.

The main tasks of the project can be summarized into the next points:

1. Study of Blockchain technology, digital signature and sockets programming.
2. Design and implement a first stage network application protocol.
3. Perform the integration part with the OS level daemon.
4. Deploy and test the mentioned application to measure its scalability.

In spite of needing an internet connection and a pair of previously distributed asymmetric keys, the application can be used in mostly any Linux kernel running IoT device. However, constraints of the underlayer project have to be taken into consideration too. The main milestones of the project will be performed in the next way.

First, how distributed networks work and maintain consistency has to be understood as well as studying the fundamentals of Blockchain technology and digital signature. After information gathering is finished, the protocol design will begin defining system behaviour, data structures and main algorithms. Once the design is ready, the implementation juncture will start using **C++** programming language, accompanied by cryptographic libraries and sockets programming tools. After it, the integration stage with the OS level development will be carried out. To finalize, some deployment and scalability tests will be performed.

[4]https://en.wikipedia.org/wiki/Merkle_tree

3. CHAPTER

Preliminaries

3.1 Blockchain and Byzantine consensus

Blockchain technology can act as a distributed trust provider in scenarios where an incremental record of files has to be maintained. The chain is formed applying a hash function to the concatenation of the latest chain digest and the arrived block. So, each of the blocks is associated with its predecessor and any small change in any of its elements will alter the result drastically. A hash function[1] is an algorithm that processes an arbitrary sized input data and transforms it into a fixed-size output; e.g. **SHA-256**, BLAKE3, MD5. The main properties of hash functions are: deterministic behaviour, computational efficiency, irreversibility, random appearance and collision resistance.

3.1.1 Non permissioned blockchains

Any entity identified by a public key-pair can join the system, propose new blocks to be appended into the public blockchain and participate in the system status validation consensus [Fernández-Bravo, 2018]. Proof of Work[2] (PoW), mining process, is used as a worktrace of the participant nodes to avoid gossiping attacks: e.g. Sybil attack [Douceur, 2002]. However, blockchain is still vulnerable to percentage attacks, being anyone in control of more than a 50% of the network able to arbitrarily modify the chain. Non permis-

[1] https://en.wikipedia.org/wiki/Hash_function
[2] https://en.wikipedia.org/wiki/Proof_of_work

sioned blockchains are the ones used in cryptocurrency infrastructures, such as Bitcoin and Ethereum.

3.1.2 Permissioned blockchains and consensus

The process of appending a block to the blockchain and achieving consensus is performed by known entities of a trusted domain [Cachin and Vukolic, 2017], providing a consensus environment to distributed systems. Nevertheless, the agents must satisfy the following constraints:

- Each node knows the **identity** of all the nodes in the network.
- The public **key distribution** is done previously to the system start up, by a key distribution mechanism.
- It is not necessary to implement PoW (not susceptible to Sybil attacks).

The importance of specifying a fault tolerant mechanism is vital to detect problems and achieve synchrony. The following points establish a framework to maintain deterministic consensus on a distributed node set:

1. Creation of a **deterministic state machine**, which implements the logic of the service to be replicated.
2. Implementation of a **consensus protocol** which broadcasts requests among nodes, so that a node executes the same request sequence in its self service instance.

State machines are used to implement Byzantine fault tolerant services, replicating the servers and coordinating the interactions of the clients with the server replicas; accomplishing the set of replicas to perform as a unique centralized service [Schneider, 1990].

3.1.3 Fault tolerance and consensus

Blockchain based systems should countermeasure the highest amount of faults as possible, being byzantine nature faults the most dangerous ones. Byzantine faults represent arbitrary malicious behaviors acting in an undetectable way for other components of the system. In the same thread, it is recommended to deploy the replicas over different platforms and locations, which depend on distinct energy sources and network infrastructures. As far as possible, employing different design versions and implementation components to avoid hardware failures bringing down the entire system. The main problems to achieve distributed consensus with presence of Byzantine faults are the following ones: synchronization, consistency, fault detection and cryptography [Fernández-Bravo, 2018].

3.1.4 Synchronization models among agents

- **Asynchronous communication model:** No bounded time limits exist.
- **Synchronous communication model:** Bounded time limit exist.
- **Partially synchronous communication model:** The default behaviour of the system is asynchronous, until a bounded time interval from a protocol is executed, establishing time limits over message send and processing mechanisms. That exact moment is called Global Stabilization Time (GST).

3.1.5 Design decisions

In the elaboration of a protocol working over a replicated state machine the design decisions are:

1. Communications have to be established over trusted channels.
2. An atomic diffusion protocol is needed to achieve a global commitment about which value should be accepted by all the processes.
3. On a fault tolerant distributed system, a partially synchronous communication model is needed to achieve consensus.
4. To enhance protocol security and efficiency, independent consensus and application engines are needed.
5. A scalability threshold of the system has to be estimated, tolerating f *faults* in n *nodes*.

3.1.6 Similar approaches

On the one hand **Trebizond**[3] is an algorithm which maintains a shared state among a series of distributed nodes; it is appropriate for permissioned blockchain systems. To its deployment, public key sharing is assumed to be previously performed. Cryptographic keys are used to sign and verify the authenticity of distributed messages among nodes. Trebizond incorporates an explicit separation between the execution and consensus layers, amplifying the productivity of the system.

On the other hand, **Tendermint**[4] is an application formed by a Blockchain consensus engine and an application interface. In consonance with this project, the consensus engine ensures that the same transactions are recorded on every machine in the same order. Moreover, it has a tolerance threshold of $\frac{1}{3}$ of the total agents failure. Thus, the system will continue working correctly while $\frac{2}{3}$ of its agents are not compromised. As it is expected, both approaches have in common that every node acts as a replica; recording the same blockchain transactions and data structures.

3.2 Digital signature

Digital signature is a base component of many cryptographic protocols. In the same way, asymmetric cryptography is the main pillar on which digital signature is built. In short, digital signature is an algorithm that given an input message and a private key, a series of mathematical functions are applied to obtain, a signed cryptogram corresponding to the original message. The signature is then attached to the original message and sent to destination.

Digital signature tries to replicate handwritten signatures in a more secure way, providing similar security features: authenticity, integrity and non-repudiation. As a result, the original message, the signed cryptogram and the public key of the signer have to be used in the signature verification process. The next Figure 3.1 shows how the signing process is performed:

[3] https://riunet.upv.es/handle/10251/115369
[4] https://docs.tendermint.com/master/introduction/what-is-tendermint.html

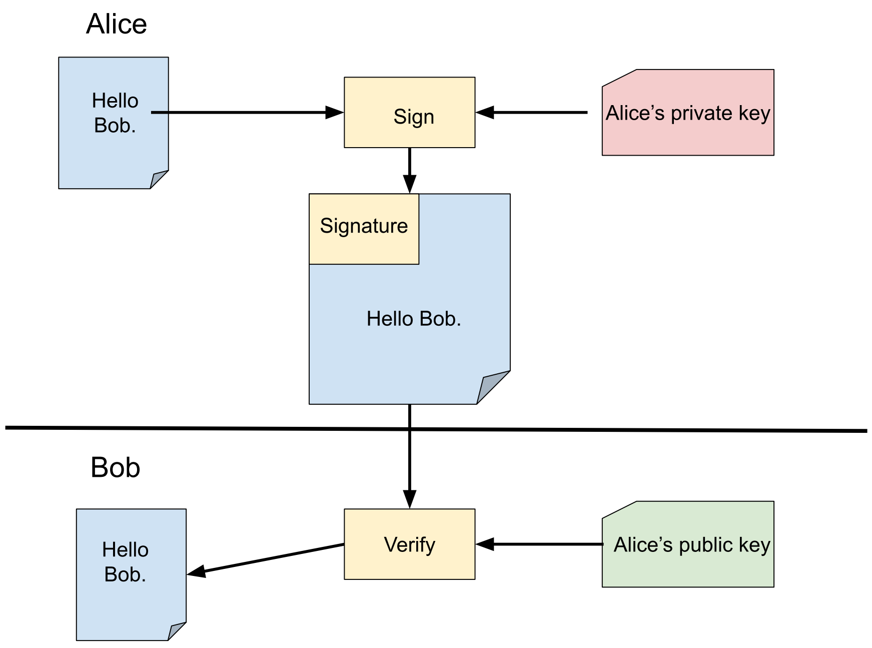


Figure 3.1: Digital signing process.

3.3 Sockets programming

Sockets are communication channel resources provided by the OS, allowing the interchange of information among applications and services using IP directives. Different programming languages such as C / C++ have libraries that make possible the usage of those socket resources. From the point of view of a developer, sockets are "files" opened in a special way. However, they are not so simple. On the one hand, an IP address and a port number are needed in order to access the desired machine and service. On the other hand, the client-server architecture has to be respected; implementing the correct socket type on each application. The following steps show how client and server sockets have to be created in C / C++:

Server

1. **Open a socket: socket()** Function returns a file descriptor.

2. **Notify the OS:** To make an association between a program and a socket, **bind()** function has to be used.

3. **Start listening in the established address: listen()** Function is used to make the OS listen to the desired socket.

4. **Accept connections: accept()** Function asks the operating system about the incoming service requester.

Client

1. **Open a socket:** Same process.

2. **Connect to a server socket: connect()** Function establishes a link with a specified application from a server.

To send and receive data, both server and client use **send()** and **recv()** functions with a fixed size buffer; being **recv() execution blocking** by default. Furthermore, once the desired data interchange concludes, memory resources have to be released by closing the connection **close()** in both interlocutors. Finally, sockets have OS established limitations; e.g. the maximum number of sockets an application can open simultaneously in Linux systems is 65535.

4. CHAPTER

Design

Although the protocol is nurtured by technologies and paradigms of previous chapters, the design is fully **custom**. Developing a secure and consistent distributed network protocol requires dealing with multiple concurrent scenarios. Moreover, Byzantine fault tolerant protocols should take into account a wide variety of possible malicious situations and have a countermeasure action to mitigate their effects. Hence, the process of designing the algorithm and structuring data, is necessary to create a robust starting point before the implementation is carried out.

The protocol is designed to operate in a **permissioned** enterprise network, with an initially trusted node set. Despite not being centralised, each node has to know the whole network topology; other nodes in the network. However, as the topology is completely virtual, the location and interconnection of the nodes is up to the disposal of system administrators.

Each node has a group of directories and files to preserve in time, in which integrity should not be altered until a system administrator, explicitly, indicates it. A **hash** or digest of the chosen filesystem is computed by an underlayer daemon [Herrero, 2021] based on Merkle Tree. The obtained hash is used as a descriptor to evaluate integrity of data and as the main element to propagate throughout the network. Hash features allow sending integrity information in a secure and light way; it is apparently random and impossible to return to source data. Notwithstanding hash functions exempt from using confidentiality providing cryptographic mechanisms, each node has to prove its identity to other nodes and demonstrate that it is trustworthy. In order to achieve it, public key cryptography in the form of **digital signature** is used.

Additionally, the network should act as a highway to transport data, control and audition messages. Every single node plays two roles at the same time, server and client. Thus, the network is really a **P2P** one, in which the actors have to access the same memory blocks preserving consistency. On the one hand, the client acts as an auditor and processes requests from the interactive console interface. On the other hand, the server must answer all the requests from the network to avoid incident accumulation.

4.1 Policies and constraints

Before starting with the deployment, a **configuration XML** file containing the network specifications must be adjusted, by a system administrator, in each of the nodes. The management file has to respect the following Figure 4.1 tree structure:

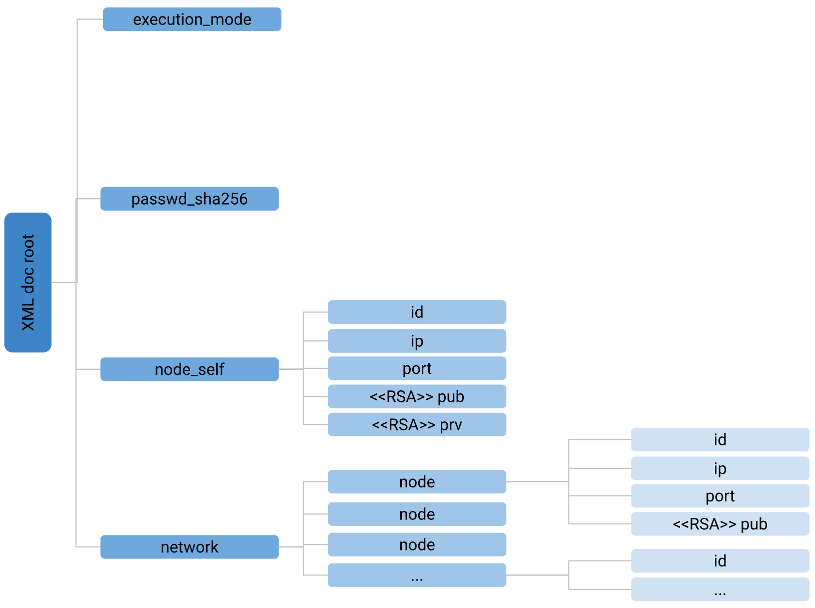


Figure 4.1: Tree structure of the XML configuration file.

Going deeper into the designing aspects of the configuration scheme. The previous Figure 4.1 establishes the main pillars on which future data structures will base. Those structures correspond to the branches, self node and network. Containing, respectively, the attributes describing the self node and a list of network nodes with their specifications. Each node must have a unique identification number $ID \in \mathbb{N}$ as well as a valid IP address (IPv4 or IPv6) and port numbers.

Besides, the self node must have a pair of public and private **RSA** keys (in DER - binary format), issued using a PKI or similar infrastructure. In the same thread, the network must contain all public keys from the rest of the nodes. Therefore, staff in charge of the deployment, has to deal with public key distribution in a secure way; deciding key propagation mechanism and channel. Moreover, the **execution mode** (0 - DEFAULT or 1 - DEBUG) and SHA256 digested hash updating password have to be specified.

The directory distribution of the application must follow the next scheme. A root directory within all the needed directories are located:

- **deploy_utils** (deploying scripts and tools): XML generator python script and C++ RSA keys generator application.
- **include** (dependent packages): Rapid-XML package.
- **logs** (log history): Log files generated during the execution.
- **RSA_keys** (RSA keys): Public keys corresponding to all network nodes and a key-pair of self node.
- **src** (main development): Application headers, source files, Makefile and network deploying shell script.
- **XML_config** (configurations): XML configuration file.

A deterministic state machine with **two states**, trusted / mistrusted, has to be defined; being the initial status of the network and all its nodes trusted. Moreover, during the execution time, a minimum number of network nodes must be trusted and operative to have a sanitized working network. The compulsory **threshold** to achieve that goal, is to have at least $\frac{2}{3}$ of the rest of the nodes not compromised.

Additionally, the minimum number of nodes to deploy the system is 3 (the main node and two network ones). Each network node must have a default trust level number that will decrease, a unit, each time a (well formulated) blaming message arrives. The trust level has to be linked to the amount of nodes in the network; $\frac{2}{3}$ times the number of network nodes. Anomalies such as sudden hash updating will contribute to point losing, until trust level reaches 0 and the node is banned from the network; declaring it as not trusted.

Linked to the mentioned trust level, each node must have a maximum number of accumulated **incidents**. In a similar way to trust level, incidents are an internal number associated with a network node. However, they are managed in an isolated way; no blaming occurs when incidents happen. Situations such as not being available, out of sync packages, signature failure, message faking and so on, will trigger incident accumulation. Surpassing the maximum amount of allowed incidents, implies losing trust in a node completely. As the natural incident number is expected to be directly proportional to the number of network nodes, the roof will be established by a $\frac{1}{3}$ of them (being 1 the minimum).

Although the designed distributed network protocol is completely **asynchronous**, some timeouts are needed to arrive at different scenarios. Auditor interval, maximum response delay, hash updating maximum and minimum time frames, network deployment time space and incident number decreasing rate are necessary.

4.2 Structures

As mentioned previously, each node knows the whole network composition. So, the information and structures are replicated per node. Nodes are supposed to be running the same implementation on independent machines. The following figures correspond to the development structure, data representation and sent datagrams.

4.2.1 Main data structures

Thinking on future implementations, the system is divided into the modules of Figure 4.2:

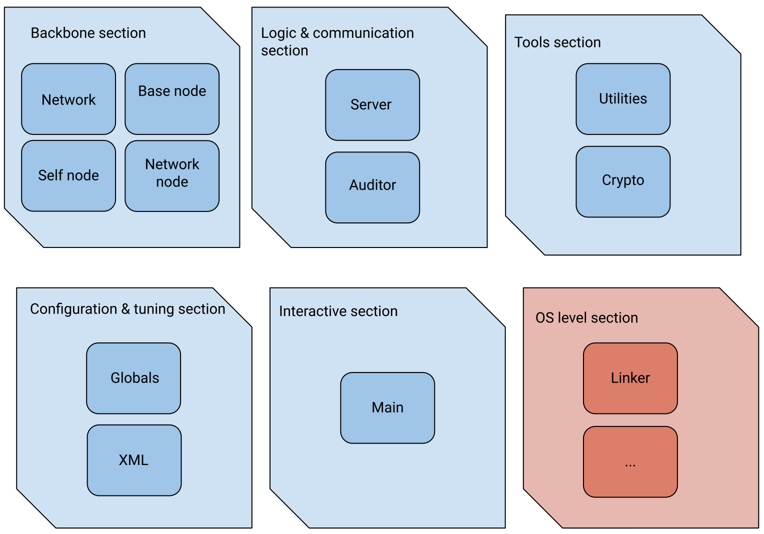


Figure 4.2: Main modules of the protocol.

- Backbone section corresponds to data structures:
  - Base node establishes all nodes common features to facilitate the implementation with inheritance.
  - Self node corresponds to the features of the local node.
  - Network node corresponds to information of a remote node.
  - The network is the cornerstone of the system. All data structures are wrapped up inside it, Figure 4.3. The network acts as an intermediate layer; the rest of the modules have to go through it to access data of nodes.

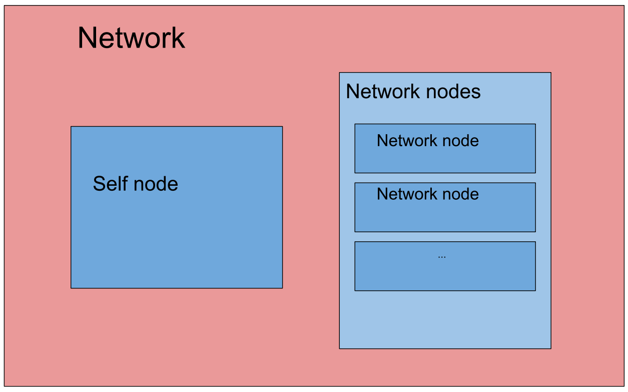


Figure 4.3: Nested data structures.

- Logic and communications section corresponds to the algorithm of the protocol:
  - Server: Gives the pertinent responses according to data in the network structure.
  - Auditor: Requests and evaluates incoming data according to the one in the network structure.
- Tools section is a tool provider for the implementation:
  - Utilities will offer random generating mechanisms, string processing methods and so on.
  - Crypto will provide other modules with public key cryptography resources.
- Configuration and tuning section is linked to setting up execution parameters:
  - Globals will fold default configuration attributes.
  - XML will represent compulsory configurations and network structure loading track.
- Interactive section corresponds to a console interface in which a system administrator can view and perform some operations in the network.
  - Main will be part of the implementation, initializing the rest of the modules.
- The OS level section is linked to the Merkle Tree implementation.

  - The linker is the only part including technology of the analogous project. It will be a single directional bridge, updating the backbone section information with the one provided by the OS level development.

4.2.2 Hash history and blockchain

The information received from other nodes is stored in three different data structures depending on the nature of the message. Although the received digest descriptor algorithm is completely modular, SHA256 is used to create the blockchains. The mentioned structures are replicated in each node and should have similar content under normal circumstances (while being trusted). The following scheme summarizes information representation per network node:

- **Good hash record:** Each time a network node hash updating is performed correctly, the received hash is stored into this **LIFO** structure. If the arrived node equals the latest value of the structure, it will not be taken into account.
- **Troublesome hash record:** Each time a blame to a node is correctly performed and does not match the latest troublesome hash in the record, the arrived hash will be saved into this **LIFO** structure.
- **Blockchain record:** Each time the good or bad hash records are modified, the latest hash of the structure is concatenated with the arrived one and the SHA256 predefined digest function reapplied. The obtained hash will become the latest element of the blockchain.

The following scheme resumes information representation in self node:

- **Good hash record:** For the self node, each hash update is considered to be OK. Having a troublesome hash record in the self node does not make sense, because integrity validation has to be done by the rest of the nodes. So, each time a monitored file is altered, the hash will be recalculated and inserted into the record.
- **Blockchain record:** Each time the previous record is updated, the blockchain structure will be updated as well, in the same way, as in the network node case.

4.2.3 Datagram structure

As the following Figure 4.4 shows, the main datagrams sent throughout the network are composed of 6 sections. **ACK** and **UPDATE_HASH** are not included in this category, because they are considered less relevant messages.

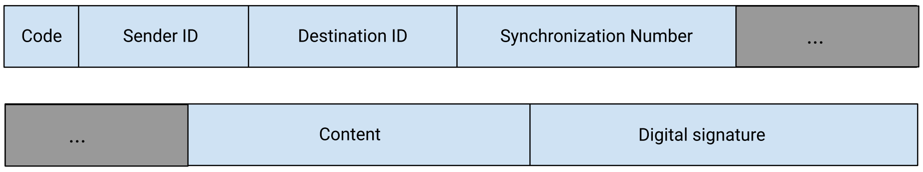


Figure 4.4: Standard structure of a datagram.

Sections of the datagram:

- **Code:** It is used as a selector of server options.
  - 0: Hash update request.
  - 1: Hash update.
  - 2: Local hash value request.
  - 2: Local hash value response. (Sent from server)
  - 3: Blaming double datagram, Figure 4.10.
- **Sender ID:** It is used as an authenticity provider measure on future verifications.
- **Destination ID:** It is needed to specify identification of the destination as a security measure on future verifications.
- **Synchronization number:** It is needed as a security measure to avoid message reusing, letting replication attacks[1]. Each self node - network node transaction will have a package counter. Every time they interchange a message, the synchronization number will increase. Hence, messages with a higher synchronization number than the latest one received will exclusively be accepted.

[1]https://ieeexplore.ieee.org/document/5948813

- **Content:** It can be empty if the message is a request. However, it can also contain a hash value or encapsulate another entire datagram (blaming case).

- **Digital signature:** It is needed to grant integrity and authenticity of all the previous fields as well to obtain a non-repudiation trace that will act as a blaming ticket. So, all the fields are concatenated and signed with the private key of the sender. The signature is encoded in hexadecimal and appended to the datagram.

4.3 Configurations

As it is mentioned previously, some configurations are set by default. Among those configurations, some are fixed values and others are dynamically linked to the dimension of the network. Whereas, others are introduced by a system administrator via **XML** file, as it is mentioned in Section 4.1.

Focusing on the internal parameters, the default trust level decrease rate is a unit as well as the default incident accumulation index. Moreover, the default hash updating maximum time frame is of 5 minutes and the minimum is of 30 seconds. In the same thread, the random waiting fixed component is of 1 minute and the default network deploying time is of 30 seconds. Next, the maximum response delay is 2 seconds and the auditor will send a request every 3 seconds (interval). Additionally, the first hash to be included in all the lists and used as first block of all SHA256 blockchains is: 0000000000000000000000000000000000000000000000000000000000000000. To end with the static values, the defined threshold to consider the network operative is $\frac{2}{3}$.

The following parameters are linked to the network node number. As mentioned previously, the trust level of each node is $\frac{2}{3}$ times the number of other nodes. Furthermore, the maximum number of incidents allowed is a $\frac{1}{3}$ of the network node number. The last dynamic parameter is the incident reset time interval, which requires more attention:

The interval is computed using a **Binomial distribution**; establishing a probability bias of a 70%, in which 2 or more times ($X > 1$ working with discrete numbers) a given network node can be repeated by a balanced random sequence (Bernoulli trials). So, the number of trials will be the number of audition requests before the incident reset mechanism decreases the incidents a unit. Additionally, to calculate the reset waiting time, the number of the mentioned audition requests have to be multiplied by the (3 sec) interval of the auditor.

- n : Number of trials?
- Probability of success (binary: YES or NO): $\frac{1}{netNodeNumber}$
- Two or more times a given network node: $X > 1$.
- Minimum probability that should be achieved: 0.7.

$X \sim Binomial(n, \frac{1}{netNodeNumber}); P(X > 1) \geq 0.7$

However, working with such sophisticated models is complex. Hence, the following statement is used as a more pessimist model to get the audition request trials number.

$2 * netNodeNumber + \frac{netNodeNumber}{2}$

The following Figure 4.5 shows the original binomial trials per network node number:

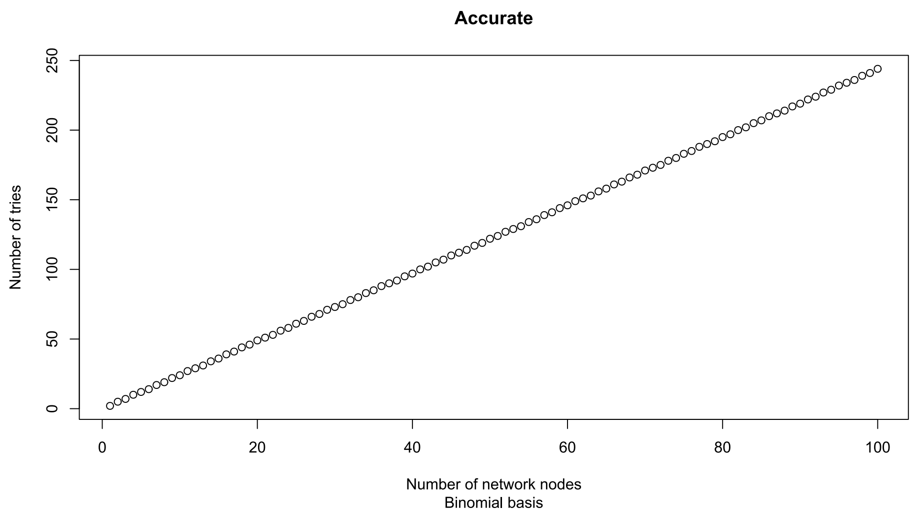


Figure 4.5: Accurate plot based on binomial trials.

The previous Figure 4.5 is obtained using the following R code, Listing 4.1. In which the number of trials is empirically incremented until the desired probability is achieved:

```r
#R–script to obtain minimum number of trials

v1 <- 1:100
v2 <- c()

for (v in v1){
 i <- 0
 p <- 0
 while(p < 0.7){
  i <- i + 1;
  p <- pbinom(1,size = i, prob = 1/v, lower.tail = FALSE)
 }
v2 <- c(v2, i)
}
plot(v1,v2, main="Accurate", sub="Binomial basis",
   xlab="Number of network nodes", ylab="Number of tries")
```

Listing 4.1: R-script to obtain minimum number of trials.

The following Figure 4.6 shows the chart of the second, more **“pessimist”** model:

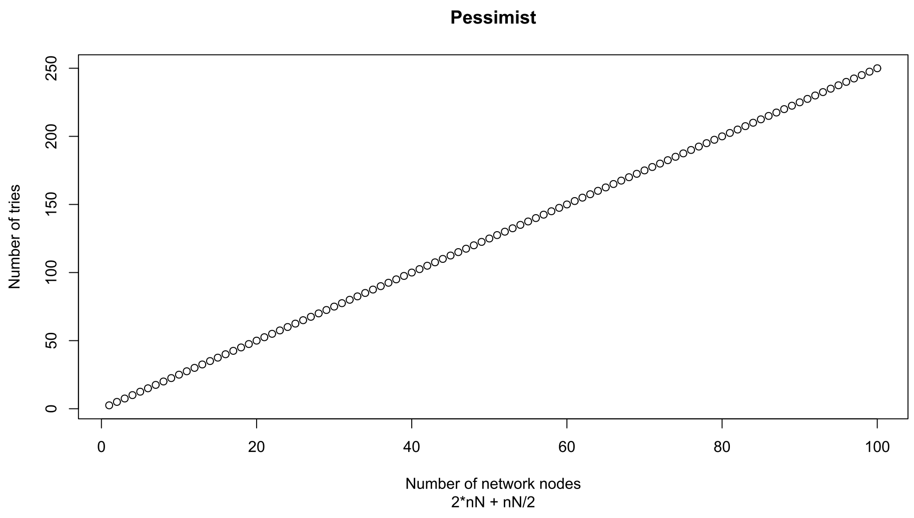


Figure 4.6: Pessimist plot based on approximation function.

The previous Figure 4.6 is obtained using the following R code, Listing 4.2:

```
#Pessimist approach to Binomial trials
v3 <- c()
for (v in v1){
 tmp <- 2*v + v/2
 v3 <- c(v3, tmp)
}
plot(v1,v3,main="Pessimist", sub="2*nN + nN/2",
   xlab="Number of network nodes", ylab="Number of tries")
```

Listing 4.2: R-script to obtain a similar approach.

4.4 Algorithm

The design is bounded to a staggered system deployment. During the starting phase, previously mentioned conditions have to be granted; key distribution, information about network nodes and a suitable system configuration. After those requirements are met, the server and linker are started up, followed by a time waiting synchronization sequence.

That synchronization sequence is used to avoid network overloads and achieve consistency before the auditor is started up. So, the delay between the two initial components launch and the first hash update broadcast has to be partially random.

As the following Figure 4.7 shows, there is a fixed component of time, representing $\frac{1}{3}$ of the waiting time and a variable component, with a balanced random function behind, representing on average a $\frac{1}{2}$ of the waiting time.

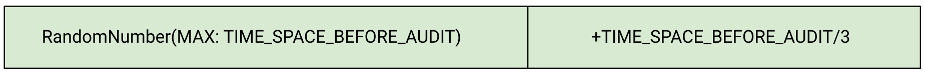

RandomNumber(MAX: TIME_SPACE_BEFORE_AUDIT)
+TIME_SPACE_BEFORE_AUDIT/3

Figure 4.7: Randomized waiting time for network deploy.

Subsequently, the system hash update phase will begin. During this phase, the local node will try to connect to all its known nodes and send them a hash updating request with a working time specification. Then, the self node will wait a pre-established response delay time, until a minimum number of $\frac{2}{3}$ of network nodes send back an ACK response. This response allows the node to test if the network is still operative and be assured that at least $\frac{2}{3}$ of the network will receive the following messages appropriately.

In short, the sender will know that a specified remote working time frame has been opened, in at least $\frac{2}{3}$ of the network nodes. If the request process ends up successfully, a local time frame will be opened as well. However, if acknowledgement messages do not cover the minimum threshold, the system will interpret that the network is compromised and network trust will be completely revoked; triggering a denial of external future interactions.

Supposing that the acknowledgment phase has performed successfully, changes in the filesystem will be carried out while the local working time frame is opened. Once the specified time window runs out, the hash describing the current state of the filesystem will be propagated throughout the network. Since other nodes have already received an updating request from the local node, the hash sent will arrive in time, as expected, and it will be recorded by the listeners. Nevertheless, if the message containing the hash does not arrive in time, or arrives without being previously notified (no request), the incident number of the sender will increase a unit.

The last part of the deploy involves waiting a predefined amount of time for messages from other nodes to arrive. Each node in the network is expected to be launched simultaneously; being the synchronization traffic balanced by the mentioned random component. Not delaying the startup of the auditor will unchain false positive blames. Hence, to achieve consistency, all nodes are given enough time to send their hashes. This last waiting stage has to be greater than the maximum possible hash updating time window, Figure 4.8. Furthermore, a deployment delay could be added as extra setting up time.

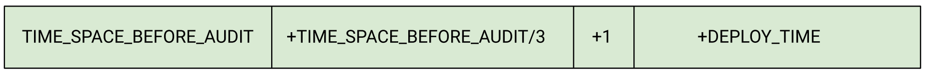


Figure 4.8: Fixed long waiting time before the auditor is launched.

Once the auditor is started up, the protocol will be in its standard stage. At this point, the network nodes and the self node are consistent. Equally, the linker, server and auditor are running. The following stages will involve, self hash update reporting, anomaly blaming, network monitoring and request serving.

While the system trusts the network and vice versa, a system administrator will be able to make a hash updating request to the network. In order to do it, the steps of Section 5.4 have to be followed. The process will continue similarly to the deployment phase. Nonetheless, this time the auditor is active and it has to be paused until the working time runs out (to avoid consistency issues). So, the server will continue working normally, responding to all the requests, but the activity of the auditor will be in standby. Then, the hash update process will proceed normally; other nodes will receive an update request, send back an ACK response and finally receive the new hash.

4.4.1 Auditor

While the system trusts the network, the auditor will be in charge of sending an audition request to a network node. Those messages are sent **atomically** to a node, by a predefined time interval. The node to be audited is computed randomly by the system. However it has to satisfy the constraints of being trusted and not having a hash update window opened. Once a node is chosen, a hash status request datagram will be sent. If the destination node is operative and the sending is performed successfully, a default response waiting time will start. If the response does not arrive in time or the selected node was not available, an incident associated with the node will be accumulated. In contrast, when a response arrives, the received datagram will be evaluated following Figure 4.9 algorithm.

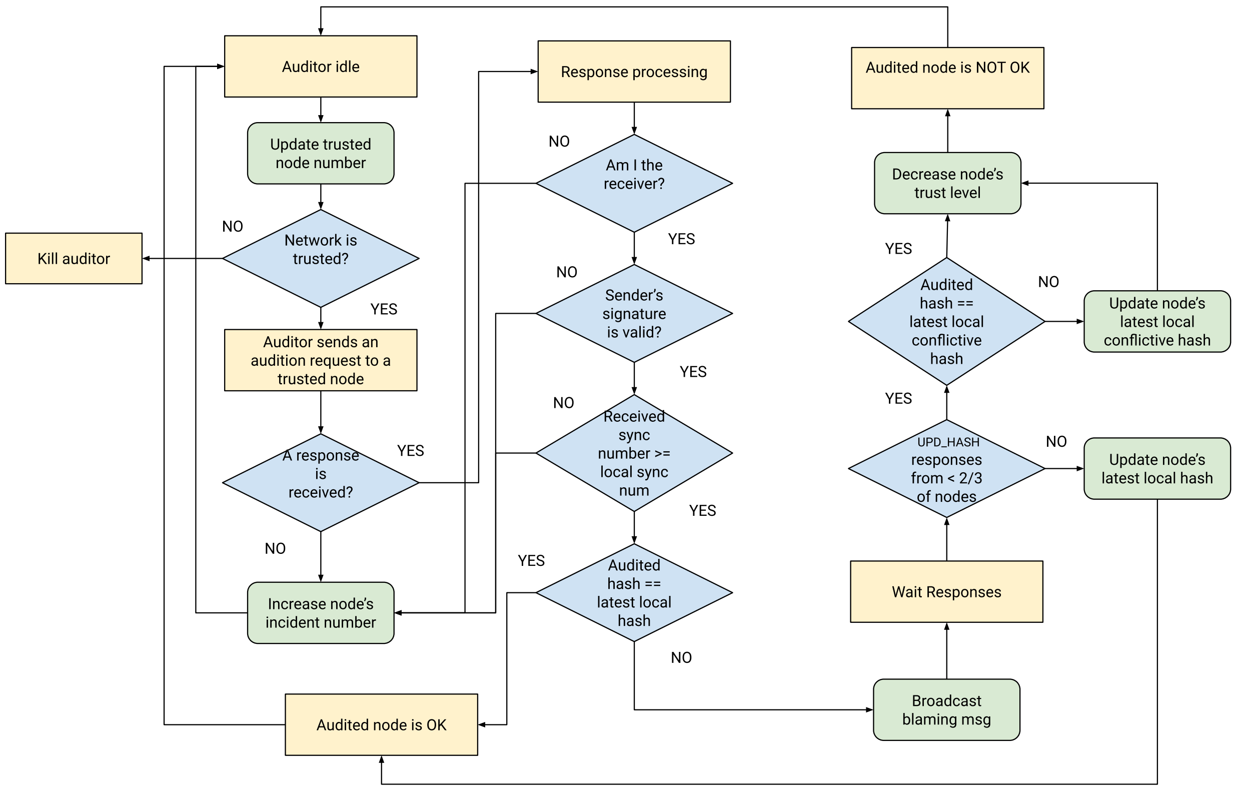

Auditor idle
Update trusted node number
Network is trusted?
NO
YES
Kill auditor
Auditor sends an audition request to a trusted node
A response is received?
YES
NO
Increase node's incident number
Audited node is OK
Response processing
Am I the receiver?
NO
YES
Sender's signature is valid?
NO
YES
Received sync number >= local sync num
NO
YES
Audited hash == latest local hash
YES
NO
Audited node is NOT OK
Decrease node's trust level
YES
Audited hash == latest local conflictive hash
NO
Update node's latest local conflictive hash
YES
UPD_HASH responses from < 2/3 of nodes
NO
Update node's latest local hash
Wait Responses
Broadcast blaming msg

Figure 4.9: Simplified algorithm of the auditor.

In the first place, response authenticity is evaluated validating the signature field and the synchronization number. When either of them is not correctly performed, the system interprets it as a soft error and an incident into the profile of the issuer is accumulated. Specifically, if the signature is incorrect, performing a blaming broadcast is impossible because no blaming ticket is achieved. Whereas, when the message is valid, the hash into it will be collated against the latest version of the one (corresponding to the audited node) in local dependencies. So, the possible scenarios are; receiving a matching hash and declaring it as correct, or receiving a mismatching hash and going to the next step of the algorithm. In which, the current datagram is broadcasted to the network, after being encapsulated inside another datagram and resigned with the signature of the auditor, Figure 4.10. As mentioned above, the non-repudiation trace of the digital signature is used as a blaming ticket.

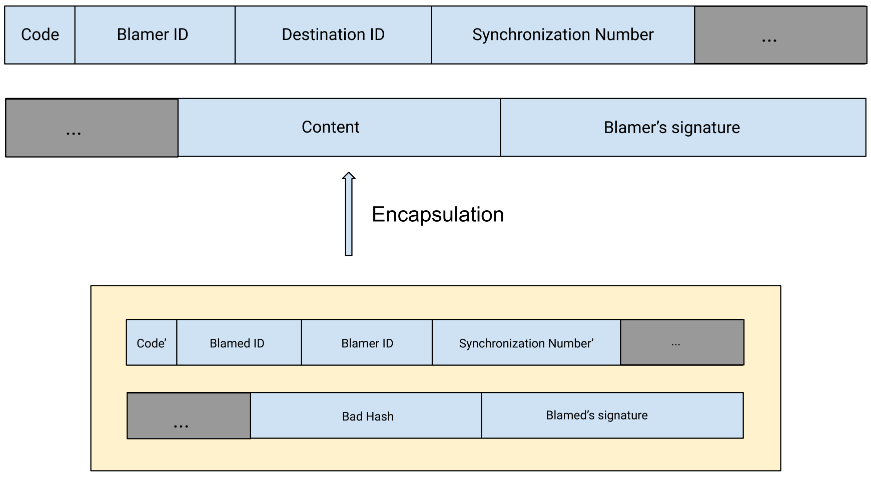


Figure 4.10: Encapsulated double datagram.

The auditor will again wait a pre-established delay time, until a minimum $\frac{2}{3}$ of the network nodes send back an UPDATE_HASH response. If the responses are received, it will mean that the formerly incorrect hash value is committed by the network. Thus, the auditor would have missed an update, having to reconsider the validity of the received hash and update its internal structures. However, the previous is not the expected situation, considering that each node hash update reaches at least $\frac{2}{3}$ of the network. Hence, when the blaming messages are successfully sent and no responses are received, the blame is considered correct. Letting a decrease in the audited node trust level and reducing network traffic in the expected scenario.

When a node makes changes in its filesystem not following the procedure of Section 5.4, the hash descriptor will be updated maliciously and other nodes will start sending blaming reports concerning it. At the end of the day, by probability, the faulty node will be exposed to random auditing requests from other nodes every auditing interval. So, as trust level is linked to the number of network nodes, as seen in Section 4.1, kicking a node from the environment is supposed to be done along with the other nodes.

As it is going to be seen afterwards, when a node blames another one correctly, trust level of both; blamer and blamed will decrease a unit in the destination(s). Broadcasting blaming messages to the network implies nodes having an altruist behaviour; they will be losing trust points during the process as well. However, it is contemplated that a single node will not kick another one by its own; reason to enhance network security adding the maximum number of nodes as possible (higher entropy). That mechanism will limit the power of the nodes by self-cancellation. Theoretically a single node will only be able to kick an analogue one, if it focuses all its blaming efforts on a single target. An important clarification has to be made at this point: The original trust level is not recovered or reset. Each time the trust in a node is decreased, it is decreased permanently.

When a node is not trusted by the network, it will be ignored by a minimum $\frac{2}{3}$ of the network nodes. Meaning that every time it sends an auditing request, no response will be received. Thus, it will accumulate incidents until the isolated node ends up not trusting more than a $\frac{1}{3}$ of the network, declaring the network as compromised.

The auditor is in charge of managing incidents of other nodes. As mentioned in Section 4.1, surpassing it means losing all trust points. However it is expected to have a minimum tolerance due to the best approach nature of the Internet. The network could be busy, some packages delayed, others lost and so on. That is why the auditor needs to reset the number of incidents little by little; decreasing the number associated with a network node after a determined time. Anyway, the incident decreasing rate has to be slower than incident accumulation owing to network isolation (when a node is not trusted by the network).

The previous statement, in Figure 4.6, shows how many audition requests have to be sent to pick up a given node, more than once, with a $> 70\%$ confidence level. In that way, the incident accumulation due to trust lost is supposed to be faster than the incident decrease interval. Therefore, the maximum number of incidents allowed can be as high as it is wanted, because it will converge to its maximum once the node is not trusted by the network.

4.4.2 Server

While the system trusts the network, the server will concurrently answer requests from other nodes based on the submitted message code. However, before deciding which decision should be taken, a series of verifications are processed. In the first place, when a client establishes a connection with the server, a countdown with a predefined time space will begin, Section 4.1. If the client does not make a request while the previous frame is opened, the server will close the connection automatically, avoiding starvation of other connections due to resource running out. Supposing that a request has arrived, its content is evaluated checking trust level of the sender, comparing synchronization numbers and verifying the veracity of the signature. If either of the elements is unsuccessfully verified, the connection will be immediately closed.

At this point, the requesting node is trusted and the received datagram is OK. So, as mentioned above, the server will respond differently depending on the received code:

- **’0’ - Hash modification request:** A node wants to open a time frame and update its corresponding hash. The requested update window will be opened and the node will be temporarily disabled as an auditing candidate. An ACK message is sent back to the requester.
- **’1’ - New hash / update:** This message must be received while the update window corresponding to the node is still opened. If it is received out of context, it will count as an incident of the requester.
- **’2’ - Audit request:** The server has to elaborate, sign and send a datagram, with its current filesystem descriptor hash to the requester.
- **’3’ - Blame from Nx (blamer) to Ny (blamed):** If the server identification corresponds to the blamed one, the package will be automatically discarded and the connection closed. Nevertheless, if the blamed node is a network type one, the encapsulated datagram signature will be checked. As seen in Figure 4.10, **Nx** has to encapsulate datagram the of **Ny** to have a non-repudiation trace. Therefore, if the encapsulated datagram signature is OK, the process will continue. However, if the signature is not OK, **Nx** will be treated as a faker and the incident number associated to it increased. Additionally, if **Ny** is not trusted already, the request will be ignored.

  Supposing that the previous steps were successful, the received hash will be checked against the locally stored one. Once again, two possibilities could happen: On the one hand, the received hash is the same as the one in the record. Suggesting that the blamer has lost an update and needs to validate the hash of **Ny**. So, an UPDATE_HASH message will be sent to the blamer and an incident accumulated (in the **Nx** structure). On the other hand, as the expected scenario, the local hash will be different from the **Ny** one. As a result, the blaming is considered correct and the trust level of **Ny** decreased. However, as mentioned previously, the trust level of **Nx** will be preventively decreased at an equal rate as well. In that way, the blamer is dissuaded from blaming other nodes arbitrarily with old stored packages, because it will end up being self-cancelled from the network.

5. CHAPTER

Implementation

5.1 Class diagram

The class diagram in Appendix A.1 shows the ins and outs of the protocol implementation. The repository containing the whole implementation is available in Appendix A.7. However, the most relevant aspects of the development are the following ones.

The Network class follows a singleton pattern. The constructor of it is private and a getInstance static method is in charge of creating the new instance or returning the existing one. So, a unique static instance is created per execution. It takes on special importance when multi thread programming is performed and every thread has to access the same memory blocks of the instance. In this case, auditor, server and linker classes, which operate in different threads, have to access network data structures.

Furthermore, the network class is in charge of initializing the whole data structure. Network initialization is in charge of reading information from “config.xml” file and creating the nested data structures inside it, Figure 4.3. Additionally, the network dimension dependent variables, declared as external integers in globals file, are assigned by the network constructor as well.

Moreover, nodes use inheritance to make the implementation of common points easier. All common features of nodes are summed up into baseNode parent class. Being the only class in which all its attributes are of protected type; attributes from the rest of the classes are of private type. Hence, netNode and selfNode classes inherit attributes and functions of baseNode; being allowed to access directly to all its elements. However, other classes such as linker, crypto, utils and globals, do not follow the same object-oriented pattern. Their duty is to serve other classes with methods or elements, without defining a constructor.

5.2 Relevant code fragments

When the program is executed, three main threads are launched: server, linker and auditor. As mentioned in previous chapters, the auditor is launched after finishing the network deployment. Moreover, each thread is able to create more threads under demand. For example, the server launches a child thread to satisfy all its requests concurrently. However, the thread propagation goes a step beyond, because those request serving threads can create time out threads to control the working time window of a node. Additionally, server, linker and auditor threads will be killed when the network is declared as not trusted. Nevertheless, the main thread will remain active displaying outdated information. The following code Listing 5.1 shows the definition of a thread function and its call from the main execution.

```
/* Server thread method */
void *serverThread(void *arg){
  network *net = net->getInstance();
  server *s = new server(net);
  s->serverUP();
  pthread_exit(NULL);
}
...
/* Initialize server thread */
int main(int argc, char *argv[]){
  ...
  if (pthread_create(&serverTid, NULL, serverThread, NULL) != 0)
  {
    ...
    Logger("Error creating server thread");
    exit(1);
  }
  ...
```

Listing 5.1: Server thread method and call in main.

Working with threads that access the same memory structures inevitably brings consistency problems. Hence, the use of mutual exclusion[1] (mutex) mechanisms gains special importance to avoid two or more processes accessing the same critical section. In the class diagram of Appendix A.1, pthread_mutex_t type attributes are defined in network and node tables. Those types of variables have to be declared, initialized and be locked

[1]https://www.cplusplus.com/reference/mutex/

/ unlocked per critical section. For example, when a node receives an update request via server thread to establish the value of changeFlag to true. The auditor thread will check the value of the variable to decide if the node is suitable to be audited or not. The same goes when the flag value is returned again to false by the timeout thread or the server thread. As the following code fragment shows, Listing 5.2, lockChangeFlag variable is used to prevent race conditions. Nonetheless, to avoid other processes starvation, it has to be locked when a thread reaches a critical section and unlocked after exiting it.

```
...
pthread_mutex_t lockChangeFlag;
...
void baseNode::setChangeFlag(bool flagValue)
{
  pthread_mutex_lock(&lockChangeFlag);
  baseNode::changeFlag = flagValue;
  pthread_mutex_unlock(&lockChangeFlag);
}
...
```

Listing 5.2: changeFlag and its mutex declarations and usage in baseNode.

A relevant part of the implementation are the steps a node must follow to send a datagram throughout the network. Working with sockets in C++ implies having a client socket and a listener (server) socket. Thus, a node has to establish a client-server connection before sending any data as well as it has to close and reassemble the socket to get it ready for future operations. Additionally, the server implementation has its mechanisms to prevent resource running out, as mentioned in Section 4.4.

5.2.1 Sender

The following rows, Listing 5.3, summarize the steps a node has to perform to broadcast a datagram:

```
...
network *net = net->getInstance();
...
net->connectToAllNodes();
...
net->sendStringToAll(0, net->getSelfNode()->getID(), to_string(timeToWork));
...
numRes = net->waitResponses(net->getNetNodeNumber() * THRESHOLD, RESPONSE_DELAY_MAX);
...
```

Listing 5.3: Steps a node has to follow to perform a broadcast.

In the first place, the node establishes connection with the rest of the trusted nodes executing the following lines, Listing 5.4, corresponding to the network class:

```cpp
bool network::connectToAllNodes()
{
  try
  {
    for (auto &i : netNodes)
      if (i->isTrusted())
        if (i->estConnection() == -1)
        {
          ...
          i->increaseIncidenceNum(INCIDENT_INCREASE);
        }
        else
        {
          FD_SET(i->getSock(), &readfds); /* Add sockets for select */
          if (i->getSock() > maxFD)
            maxFD = i->getSock();
        }
      ...
```

Listing 5.4: Network method to open a socket with every node.

Omitting C++ sockets implementation requirements. A for loop is used to iterate over all the trusted nodes and establish a connection with their running server. Moreover, a FD_SET structure is created with all the opened sockets (readfds) and the maximum file descriptor among them is stored in maxFD class variable for future uses.

The next code fragment, Listing 5.5, corresponds to the elaboration of a datagram:

```cpp
void network::sendStringToAll(int code, int sourceID, string content)
{
  ...
  try
  {
    for (auto &i : netNodes)
      if (i->isConnected())
      {
        /* Get and increment Sync Number */
        syncNum = i->getSyncNum();
        /* Specify  msgcode + sourceID + destinationID + syncNum + content */
        msg = to_string(code) + ";" + to_string(sourceID) + ";" + to_string(i->getID()) + ";" + to_string(syncNum) + ";" + content;
        signedMsg = sign(msg, std::to_string(sourceID));
        msg = msg + ";" + signedMsg + ";";
        buffer = msg;
        if (i->sendString(buffer.c_str()) == -1)
        {
          ...
          Logger("Error sending: " + i->getID());
        }
        else
          i->setSyncNum(i->getSyncNum() + 1);
      }
    ...
```

Listing 5.5: Network method to send a message to every node.

A personalized datagram is elaborated per trusted node due to previously mentioned synchronization number exclusivity requirements. Moreover, if the whole operation is performed correctly, the synchronization number is incremented to be ready for the next delivery. The datagram elements are concatenated using ';' character and signed using the sign function. Thus, before sending each datagram using the previously opened socket, the following signing procedure is applied, Listing 5.6:

```
std::string sign(std::string msg, std::string key_ID)
{
  ...
  CryptoPP::RSA::PrivateKey prv = get_prv(key_ID); /* import der priv key */
  CryptoPP::RSASSA_PKCS1v15_SHA_Signer signer(prv); /* Sign and hex encode */
  CryptoPP::StringSource ss1(msg, true,
               new CryptoPP::SignerFilter(prng, signer,
                            new CryptoPP::HexEncoder(
                              new CryptoPP::StringSink(s))) // SignerFilter
  ); // StringSource
  return s;
}
```

Listing 5.6: Crypto method to sign a given message.

With the private key of the node, the concatenated datagram is signed employing **CryptoPP** library tools. Additionally, the signed cryptogram is encoded in hexadecimal, appended to the datagram string and sent to the corresponding destination node.

Returning to the three first functions, waitResponses will be the last and more complex one, Listing 5.7. This method is used to wait selectTime seconds for a resNum incoming number of network messages to arrive. It is necessary in processes such as waiting $\frac{2}{3}$s of network nodes to answer. Hence, the following lines show how the previously created readfs structure and maxFD index are used in order to wait for some concrete opened sockets to answer; avoiding select to listen to any incoming network activity.

Moreover, the reason why **ACK** and **UPDATE_HASH** messages are not signed is because their duty is just to trigger a select statement by a previously opened socket. Therefore, it is expected that if no Man-in-the-middle attack[2] (MitM) is performed in the socket opening, ACK and UPDATE_HASH messages from the correct sockets will only be attended; socket descriptor lower or equal to maxFD by select(maxFD + 1...). However, the received response order is expected to be random. So, the following loops and temporary variables update readfds and maxFD, decreasing the maxFD number and discarding received response sockets to be listening just to the remaining connections.

[2]https://www.veracode.com/security/man-middle-attack

```
int network::waitResponses(int resNum, int selectTime)
{
  ...
  while (1)
  {
    selectStatus = select(maxFD + 1, &readfds, NULL, NULL, &tv); /* Just monitor trusted sockets; low−eq maxFD descriptor */
    counter += selectStatus;
    if (selectStatus == 0 || counter >= resNum) /* If timeout or received message number is gr eq to resNum −> break the loop */
      break;
    else
    {
      /* tmp vars for reseting values */
      FD_ZERO(&tmpFdSet); /* Clear the socket set */
      tmpMaxFD = −1;     /* Initialize tmpMaxFD */
      /* Count all received connections */
      for (auto &i : netNodes)
        if (i−>isConnected())
          if (!FD_ISSET(i−>getSock(), &readfds))
          {
            if (i−>getSock() > tmpMaxFD)
              tmpMaxFD = i−>getSock();
            FD_SET(i−>getSock(), &tmpFdSet);
          }
      /* Reset values for select */
      readfds = tmpFdSet;
      maxFD = tmpMaxFD;
    }}
  return counter;
}
```

Listing 5.7: Network method to wait $\frac{2}{3}$ of the nodes to respond.

5.2.2 Receiver

The following code, Listing 5.8, represents the most relevant aspects of **socketThread**. It corresponds to the child thread launched by the server to attend requests concurrently.

```
void *socketThread(void *arg)
{
  ...
  vectString = recvVectStringSocket(clientSocket);
  ...
  splitVectString(vectString, msgCode, clientID, selfID, syncNumReceived, content, MsgToVerify, MsgSignature);
  ...
  msgValid = net−>validateMsg(selfID, clientID, syncNumReceived, MsgToVerify, MsgSignature);
  ...
  switch (msgCode) {...}
  ...
```

Listing 5.8: Steps a server thread follow to process the incoming data.

After receiving the datagram of the requester and splitting its concatenated elements, the received message has to be validated. The first process is to check the validity of the signature. Next, the message synchronization number is checked and incremented when all the verifications go well; if not, an incident is accumulated. The following code fragment, Listing 5.9, shows how the **validation** process is done:

```
bool network::validateMsg(int selfID, int clientID, int syncNumReceived, string MsgToVerify, string MsgSignature)
{
   netNode *nN = getNode(clientID);
   if (selfID == self->getID())
   {
      int syncNumStored = nN->getSyncNum(); /* Verify if sync number is correct */
      if (verify(MsgToVerify, MsgSignature, to_string(clientID))) /* Verify if msg is correctly signed */
      {
         if (syncNumReceived == syncNumStored) /* Standard situation */
         {
            nN->setSyncNum(nN->getSyncNum() + 1); /* Increment sync number */
            return true; /* Verification successful */
         }
         else if (syncNumReceived > syncNumStored) /* Package lost occured, but greater values are valid -> resync */
         {
            nN->setSyncNum(syncNumReceived + 1);
            ...
            Logger("Package lost, resyncing - ID: " + to_string(clientID));
            return true;
         }}}return false; }
```

Listing 5.9: Network validateMsg method.

Finally, the verify function uses **cryptoPP** tools to confirm the authenticity of a message. The raw message and the signed hex encoded message have to be passed as main arguments. Identification number of the signer has to be provided as well to verify the message against a concrete public key. The following lines, Listing 5.10, correspond to the verification function:

```
bool verify(std::string msg, std::string sign_msg, std::string key_ID)
{
   CryptoPP::AutoSeededRandomPool prng;
   CryptoPP::RSA::PublicKey pub = get_pub(key_ID); /* Import public key .der */
   CryptoPP::RSASSA_PKCS1v15_SHA_Verifier verifier(pub);
   std::string decodedSignature;
   CryptoPP::StringSource ss(sign_msg, true,
                  new CryptoPP::HexDecoder(
                     new CryptoPP::StringSink(decodedSignature))); /* Decode */
   bool result = false;
   CryptoPP::StringSource ss2(decodedSignature + msg, true,
                  new CryptoPP::SignatureVerificationFilter(verifier,
                                    new CryptoPP::ArraySink((byte *)&result,
                                             sizeof(result)))); /* Verify */
   return result;
}
```

Listing 5.10: Crypto signature verification method.

5.2.3 Keys of the auditor

The following lines, Listing 5.11, show the main sections of the auditor:

```
int auditor::auditorUP()
{
   ...
   if (pthread_create(&incidentThread, NULL, resetincidentsThread, NULL) != 0) {...}
   ...
  while (1)
   {
   ...
   selfNetwork->updateTrustedNodeNumber();
   ...
   sleep(AUDITOR_INTERVAL);
   ...
   auditedID = selfNetwork->getTrustedRandomNode();
   ...
   auditNode(auditedID);
   ...
```

Listing 5.11: Main steps of an auditor.

In the first place, the incident reset thread is launched to decrease a unit the incident level of each trusted node. Next, an infinite loop is started, in which the trusted node number is updated and checked per iteration. A node will be selected to be audited if the auditor is not paused (while changeFlag of self node is false). If the network is not trusted or there are not enough nodes to be audited, the loop will be interrupted, exiting the auditor thread. The following function in Listing 5.12 is in charge of picking up a node to be audited:

```
int network::getTrustedRandomNode()
{
   int random = -1;
   if (trustedNodeNumber < netNodeNumber * THRESHOLD) /* If not enough nodes are trusted return -1 */
      return random;
   while (1) /* If there are trusted nodes, pick one among them */
   {
      random = getRandomNumber(netNodeNumber + 1 + 1); /* Self node must be counted as well +1; first node starts at 1 -> +1 */
      for (auto &i : netNodes)
         if (i->getID() == random)
            if (i->isTrusted() && !i->getChangeFlag())
               return random;
   }}
```

Listing 5.12: Network method to get a random node.

The method checks whether there are enough trusted nodes to proceed or not. Then, an infinite loop is used to obtain a random node that fits the constraints of being trusted and not having opened a hash update time frame. To get secure random numbers, getRandom-Number function is used, reading some bits from */dev/urandom* file.

Finally, when a change occurs in the filesystem hash descriptor, either server or auditor are capable of updating the data structures and blockchains of the corresponding node. As mentioned previously, the default hash function used in the elaboration of blockchains is **SHA256**. Blockchanis are computed concatenating the latest digest block of the chain with the newly arrived hash and reapplying SHA256. The following method is used to hash strings, Listing 5.13:

```
std::string hashText(std::string inputText)
{
  CryptoPP::SHA256 hash;
  std::string digest;
  CryptoPP::StringSource s(inputText, true, new CryptoPP::HashFilter(hash, new CryptoPP::HexEncoder(new CryptoPP::StringSink(digest)))); /*
      SHA256 hash and hex encode */
  return digest;
}
```

Listing 5.13: Crypto method to digest a given string.

5.3 Compilation and package dependencies

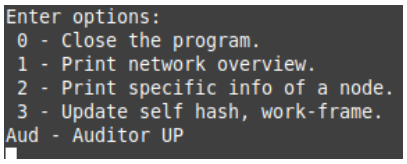
To make the compilation process easier a Makefile, Listing A.2, is created to be in charge of managing package and library dependencies. In the same thread, the code is written in C++ version 14 and g++ is used as the main compiler; from Linux build-essentials package. Moreover, the OS used to compile and run the application has to be Linux based; e.g. reading from /dev/urandom and using systemd d-bus to communicate with the OS level implementation. Additionally, the implementation has the following API dependencies:

- libcrypto++
- libpthread
- libsystemd
- rapidxml-1.13

The compiled file can be executed with and without argv parameters. If a parameter is specified, it is interpreted to be the identification number of the self node. Thus, it will be used to read its corresponding configuration file; *configID.xml*. However, if no parameter is given, the execution will get its ID from the default *config.xml* file. In future sections, this modularity will be used to build network deploying scripts.

5.4 Console interface - User interaction

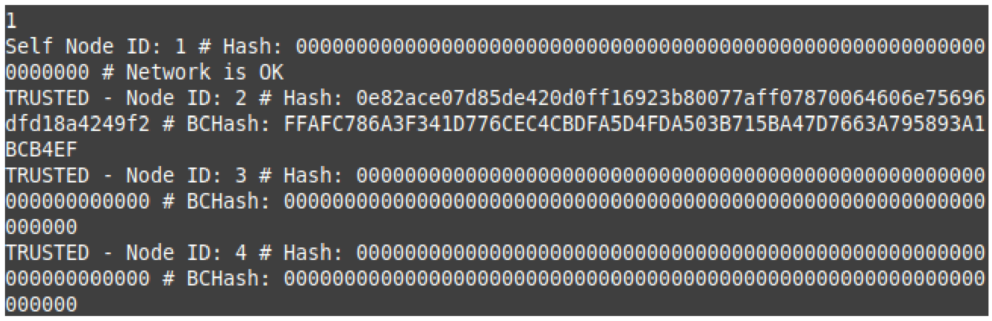
The interface is a console with four options: close the application, display an overview of the whole network, show a complete breakdown of a node and send a hash updating request. However, information shown will only be coherent if the system trusts the network, because being not trusted implies not receiving updates. To perform those operations, human interaction is needed. In this case, a system administrator is supposed to manage the console. The following Figure 5.1 shows how the menu is displayed on a terminal:



```
Enter options:
 0 - Close the program.
 1 - Print network overview.
 2 - Print specific info of a node.
 3 - Update self hash, work-frame.
Aud - Auditor UP
```

Figure 5.1: Menu shown in the console interface.

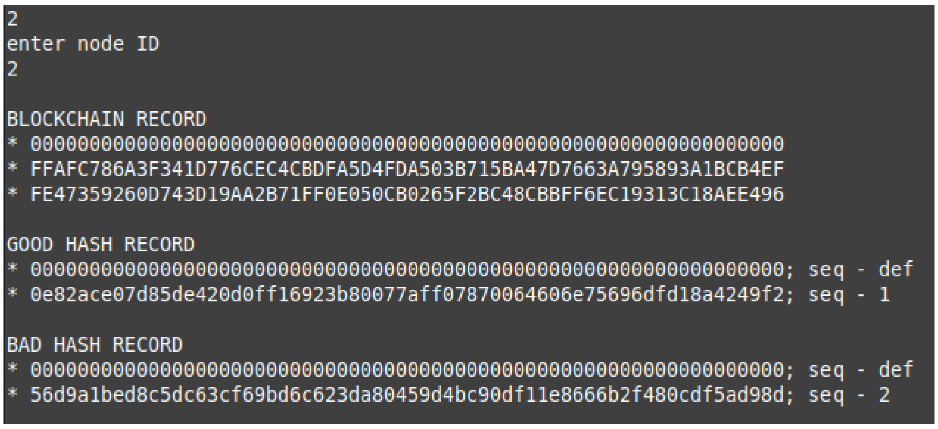
When the second option is selected, the screen of Figure 5.2 is displayed, showing the overall information of the network: whether the network is trusted or not, confidence in each of the nodes and their respectives latest valid and blockchain hashes.



```
1
Self Node ID: 1 # Hash: 000000000000000000000000000000000000000000000000000000000
0000000 # Network is OK
TRUSTED - Node ID: 2 # Hash: 0e82ace07d85de420d0ff16923b80077aff07870064606e75696
dfd18a4249f2 # BCHash: FFAFC786A3F341D776CEC4CBDFA5D4FDA503B715BA47D7663A795893A1
BCB4EF
TRUSTED - Node ID: 3 # Hash: 0000000000000000000000000000000000000000000000000000
000000000000 # BCHash: 0000000000000000000000000000000000000000000000000000000000
000000
TRUSTED - Node ID: 4 # Hash: 0000000000000000000000000000000000000000000000000000
000000000000 # BCHash: 0000000000000000000000000000000000000000000000000000000000
000000
```

Figure 5.2: Network overview.

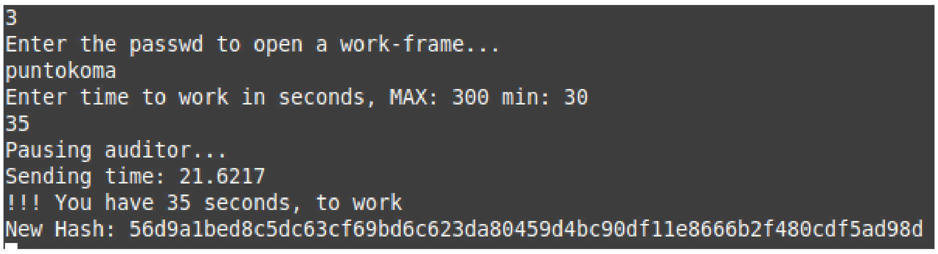
When the third option is selected, a specific node number is asked as well to perform a complete breakdown of its information. The following screen is displayed, Figure 5.3, showing the full history of the selected node: whether the node is trusted or not, a complete blockchain history of the performed actions and both good and troublesome hash records. In the same thread, those hash records are followed by a sequence number *seq - N*. That number describes the hash arrival position and can be used to manually track the blockchain. To the first default 000... values, *seq - def* traces are used.



```
2
enter node ID
2

BLOCKCHAIN RECORD
* 0000000000000000000000000000000000000000000000000000000000000000
* FFAFC786A3F341D776CEC4CBDFA5D4FDA503B715BA47D7663A795893A1BCB4EF
* FE47359260D743D19AA2B71FF0E050CB0265F2BC48CBBFF6EC19313C18AEE496

GOOD HASH RECORD
* 0000000000000000000000000000000000000000000000000000000000000000; seq - def
* 0e82ace07d85de420d0ff16923b80077aff07870064606e75696dfd18a4249f2; seq - 1

BAD HASH RECORD
* 0000000000000000000000000000000000000000000000000000000000000000; seq - def
* 56d9a1bed8c5dc63cf69bd6c623da80459d4bc90df11e8666b2f480cdf5ad98d; seq - 2
```

Figure 5.3: Detailed information of node 2 from node 1.

Finally, when the fourth option is selected, Figure 5.4, the user is asked to enter the source text of the previously loaded (SHA256 digested) password. After verifications conclude, the user is asked again to enter the amount of seconds the updating process will take (it must be between the time bounds defined in globals). The user will be able to start with the filesystem modification, after the auditor has been paused and an explicit message indicating it is displayed.



```
3
Enter the passwd to open a work-frame...
puntokoma
Enter time to work in seconds, MAX: 300 min: 30
35
Pausing auditor...
Sending time: 21.6217
!!! You have 35 seconds, to work
New Hash: 56d9a1bed8c5dc63cf69bd6c623da80459d4bc90df11e8666b2f480cdf5ad98d
```

Figure 5.4: Hash update interaction.

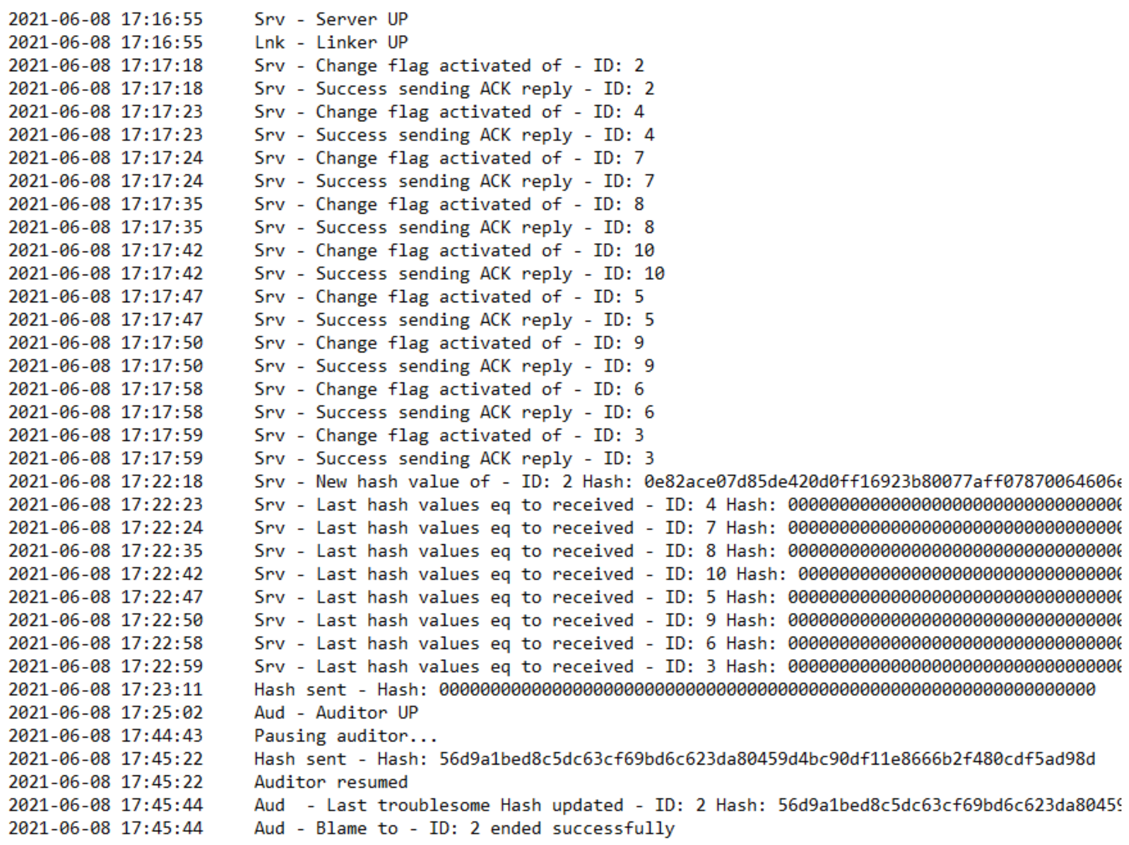
5.5 Logs and execution modes

As mentioned in previous sections, there are two execution modes: DEFAULT and DEBUG. They are in charge of managing the amount of information to be printed by the standard output. On the one hand, DEFAULT prints the minimum amount of messages to avoid poisoning the console interface with redundant feedback. It just displays the main menu and all the actions triggered by the interactor. Nonetheless, the main deploying stages and thread stops / pauses are shown as well; start sequence, thread kills (network not trusted) and auditor pauses (hash updates). On the other hand, the DEBUG mode prints every interaction among nodes, audition results, blame messages and so on.

It is recommended when a detailed representation of what is going on in the network is wanted. This mode is necessary to perform debugging tasks. In the same thread, DEBUG is the only execution mode that displays standard error outputs.

However, there is still a third element in charge of maintaining a history of the network events. Independently of the execution mode, a general overview log is formed with the main reports and issues. The previously mentioned logs directory is created, containing files with *<ID_>yyyy-mm-dd.txt* notation. Each file contains the main events happened during the execution, such as successful blame messages, successful hash update operations, package lost incidents, thread status changes and so on.

Nevertheless, the logger messages do not include all the audition and datagram interactions the DEBUG mode does. Whereas, when an error or significant event occurs, the two report modes and the logger, specify the event originator process; printing *Aud / Lnk / Srv* - <*event*>. Finally, the logger is the only reporting system recording the execution date and hour per row; the following lines in Figure 5.5 show a fragment of a log file:



```
2021-06-08 17:16:55     Srv - Server UP
2021-06-08 17:16:55     Lnk - Linker UP
2021-06-08 17:17:18     Srv - Change flag activated of - ID: 2
2021-06-08 17:17:18     Srv - Success sending ACK reply - ID: 2
2021-06-08 17:17:23     Srv - Change flag activated of - ID: 4
2021-06-08 17:17:23     Srv - Success sending ACK reply - ID: 4
2021-06-08 17:17:24     Srv - Change flag activated of - ID: 7
2021-06-08 17:17:24     Srv - Success sending ACK reply - ID: 7
2021-06-08 17:17:35     Srv - Change flag activated of - ID: 8
2021-06-08 17:17:35     Srv - Success sending ACK reply - ID: 8
2021-06-08 17:17:42     Srv - Change flag activated of - ID: 10
2021-06-08 17:17:42     Srv - Success sending ACK reply - ID: 10
2021-06-08 17:17:47     Srv - Change flag activated of - ID: 5
2021-06-08 17:17:47     Srv - Success sending ACK reply - ID: 5
2021-06-08 17:17:50     Srv - Change flag activated of - ID: 9
2021-06-08 17:17:50     Srv - Success sending ACK reply - ID: 9
2021-06-08 17:17:58     Srv - Change flag activated of - ID: 6
2021-06-08 17:17:58     Srv - Success sending ACK reply - ID: 6
2021-06-08 17:17:59     Srv - Change flag activated of - ID: 3
2021-06-08 17:17:59     Srv - Success sending ACK reply - ID: 3
2021-06-08 17:22:18     Srv - New hash value of - ID: 2 Hash: 0e82ace07d85de420d0ff16923b80077aff07870064606
2021-06-08 17:22:23     Srv - Last hash values eq to received - ID: 4 Hash: 0000000000000000000000000000000
2021-06-08 17:22:24     Srv - Last hash values eq to received - ID: 7 Hash: 0000000000000000000000000000000
2021-06-08 17:22:35     Srv - Last hash values eq to received - ID: 8 Hash: 0000000000000000000000000000000
2021-06-08 17:22:42     Srv - Last hash values eq to received - ID: 10 Hash: 000000000000000000000000000000
2021-06-08 17:22:47     Srv - Last hash values eq to received - ID: 5 Hash: 0000000000000000000000000000000
2021-06-08 17:22:50     Srv - Last hash values eq to received - ID: 9 Hash: 0000000000000000000000000000000
2021-06-08 17:22:58     Srv - Last hash values eq to received - ID: 6 Hash: 0000000000000000000000000000000
2021-06-08 17:22:59     Srv - Last hash values eq to received - ID: 3 Hash: 0000000000000000000000000000000
2021-06-08 17:23:11     Hash sent - Hash: 0000000000000000000000000000000000000000000000000000000000000000
2021-06-08 17:25:02     Aud - Auditor UP
2021-06-08 17:44:43     Pausing auditor...
2021-06-08 17:45:22     Hash sent - Hash: 56d9a1bed8c5dc63cf69bd6c623da80459d4bc90df11e8666b2f480cdf5ad98d
2021-06-08 17:45:22     Auditor resumed
2021-06-08 17:45:44     Aud  - Last troublesome Hash updated - ID: 2 Hash: 56d9a1bed8c5dc63cf69bd6c623da8045
2021-06-08 17:45:44     Aud - Blame to - ID: 2 ended successfully
```

Figure 5.5: Piece of a log generated by node 1 in a network of 10 nodes.

5.6 Leaks

Although the design and implementation are developed taking into account as many consistency issues as possible, it is nearly impossible to create a 100% Byzantine fault tolerant algorithm. Problems such as not expected attacks, hardware issues or percentage attacks, can happen out of any methodically performed development. This section will cover the main problems the algorithm has.

In spite of being an unusual situation, when an attacker gains control over $\frac{2}{3}$ of the nodes, the network will become unpredictable. The attacker could arbitrarily manipulate information of the nodes and expel the not infected ones from the network. In addition, if a private key from a node is compromised, an intruder might be able to perform *MitM* attacks, impersonating the identity of a node and signing datagrams on behalf of it.

Moreover, external DDoS attacks are not handled by the algorithm; they are supposed to be dealt by an external firewall, proxy or similar technology. However, interrupting the audition message flow could negatively affect the behaviour of the network. The incident level associated with the blocked nodes will start growing, until they are marked as not trusted. In the same way, the nodes are disconnected for being mistrusted by the network.

Furthermore, the network can suffer from scalability problems. As mentioned previously, every node must replicate the information structures of other nodes as well as store their private keys. Although not being a big problem due to current memory availability and the default needed blocks, the space required by the application is directly proportional to the size of the network.

Another aspect linked to the dimension of the network is the amount of traffic generated through it. When a node is corrupted and other nodes start sending blaming messages, the traffic will increase considerably. Each node has to propagate a message to every trusted network node. Hence, apart from skyrocketing the traffic, it could overload the sending ability of the nodes, because no explicit broadcast method is designed. As mentioned previously, each message needs a specific synchronization number per node to node communication. Therefore, broadcast can not be a unique message to all nodes. A node has to send individual messages to every trusted node.

Lastly, the previously mentioned synchronization number will increase a unit each time a message is interchanged between two nodes. So, the situation of overflowing the counter could arrive sometime; working with integers in C / C++ - **INT_ MAX** (4 bytes) limit.

6. CHAPTER

Integration

As mentioned in previous sections, this project is fully completed with *File integrity monitoring on Linux systems* underlayer implementation. The kernel level daemon uses systemd d-bus technology to send messages to the current upper layer environment. To merge both projects, knowledge from both developers is joined and a linker class created. However, the linker implementation is located in the upper layer application, because it is the only one using data from the other development. So, the linker can be described as an unidirectional bridge from the OS implementation to the network protocol. For more information about systemd and d-bus refer to [Herrero, 2021] documentation.

The most relevant aspects of the linker implementation, related to the network protocol, are shown in the following code fragments:

```c
int main(int argc, char *argv[])
{
...
if (pthread_create(&linkerTid, NULL, linkerThread, NULL) != 0) {...}
...
}
```

Listing 6.1: Launch of linker thread.

```c
int dbus_init()
{
/* Definition of the bus object and method*/
static const sd_bus_vtable linker_vtable[] = {..., SD_BUS_METHOD("updateHash", "s", "i", method_updateHash, SD_BUS_VTABLE_UNPRIVILEGED |
        SD_BUS_VTABLE_METHOD_NO_REPLY), ...};
...
/* define a specific name and path */
int dbus_init()
{
```

```
...
   r = sd_bus_add_object_vtable(bus, &slot,
                  "/net/linker/manager", /* object path */
                  "net.linker.manager", /* interface name */
                  linker_vtable, NULL);
...
}
```

Listing 6.2: Set up linker update method; object path and interface name.

The main execution process creates a thread, Listing 6.1, which configures and initializes a d-bus listener in */net/linker/manager* path, with *net.linker.manager* name, Listing 6.2. So, the underlayer daemon can send messages to the created process referring to the listening domain. Each time a message is received, the following method will be executed, Listing 6.3:

```
int method_updateHash(sd_bus_message *m, void *userdata, sd_bus_error *ret_error)
{
   ...
   network *net = net->getInstance();
   r = sd_bus_message_read(m, "s", &newHash);
   if (net->isNetworkCompromised()) {...} /* If network not trusted, kill thread */
   ...
   net->getSelfNode()->updateHashList(newHash);
   ...
}
```

Listing 6.3: Update self hash in Linker through Network.

The network instance has been created at this point by the main execution. Hence, when the linker calls getInstance method inside method_updateHash, the existing network instance is returned. The next steps in Listing 6.3 involve; hash receiving, evaluation of the network trust level and a call to updateHashList method with the received hash as parameter.

7. CHAPTER

Deployment and tests

7.1 Background

In order to test the **scalability** of the protocol, the measure chosen is package send and receiving time, for being the most restrictive one. Therefore, less restrictive measures, such as memory demanding are not taken into account. However, as shown in Section 5.6, working with super massive networks could cause memory problems too. In the same thread, OS sockets max amount could limit the size of the network as well, shown in Section 3.3. Some deployment executions are performed locally in the following machine:

- **Brand and model:** Lenovo Thinkpad E14 - year 2020.
- **CPU:** i7-10510U 1.80Ghz x 4.
- **RAM:** 16GB DDR4.
- **OS version:** Linux Mint 20.1 Cinnamon 64-bit.
  - **Kernel version:** 5.4.0-66-generic.

Although it is not a real situation in which all the nodes are distributed in remote machines, the tests show the average time a node requires to connect to the rest of the nodes, send an individual message to each one of them (broadcast) and wait until $\frac{2}{3}$ of the network responds. As mentioned in previous sections, those three steps are part of the hash updating sequence. Hence, as the maximum default waiting time to get a response is of 2 seconds, in a correctly working network, the values received should be always lower. In the same thread, it is expected to achieve linearly incremental execution times, because each node knows the whole network topology.

However, the implementation was not originally designed to deploy more than an instance per machine. Thus, some modifications and scripts are created to make large scale deployments easier. A two variant shell script is designed deploy.sh, Listing A.3, which operates differently depending on the introduced parameter number. On the one hand, if the number of nodes is the only provided field, the script interprets that the required files have been previously created and it proceeds to run the application on independent shell tabs. On the other hand, when a 'g' (generate) second parameter is contiguously provided to the node number, the script will use the tools inside deploy_utils directory to create the required files.

The script iterates node number times, calling generate_keys RSA key-pair generator (C++ application - Listing A.4) and generate_configs.py XML configurations generator (Python script - Listing A.5). Therefore, the correct usage of the script involves calling it (firstly) with the 'g' parameter and then proceed to network deployment, executing the script again with the same number of nodes, omitting the 'g' clause.

7.2 Results

The previous script is called with 10, 20, 40, 80 and 160 nodes. The data is processed making an average, among eight randomly chosen hash update times. Table A.1 shows the mean of the experiment results collected in Table A.2.

Table A.1 is used to elaborate Figure 7.1 chart. The chart shows the execution time distribution and its corresponding linear regression, computed by the *least squared method*[1].

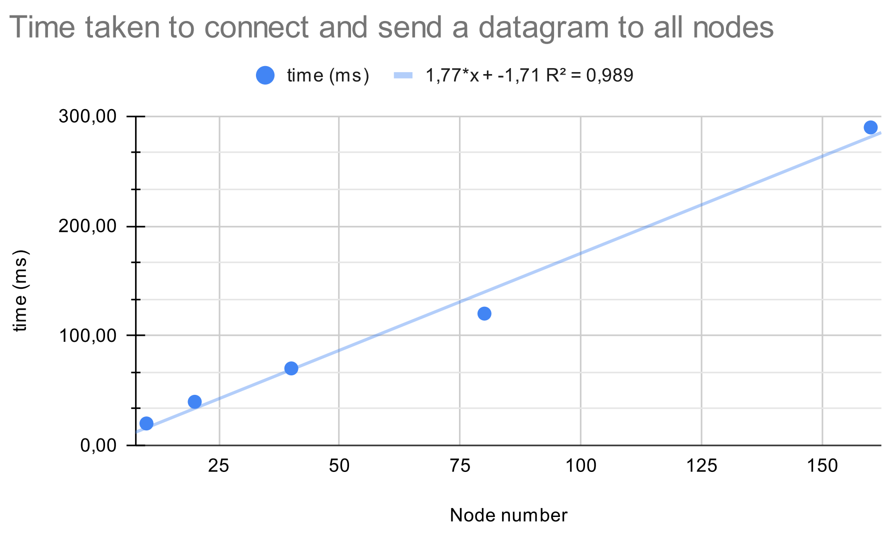


Figure 7.1: Distribution of the execution results and linear regression.

As it was hypothesized, the distribution follows a complete linear tendency. The determination coefficient of the computed linear regression is nearly 1; 0.989. Hence, the obtained values are quite optimistic, because the same growth trend is maintained independently of the number of nodes; it does not get skyrocketed from one concrete value on. Nevertheless, in a real scenario where the nodes are deployed in different locations, other factors such as bandwidth and network overload will have to be taken into account.

[1] https://mathworld.wolfram.com/LeastSquaresFitting.html

8. CHAPTER

Project management

Project management has been crucial to develop a well structured project as well as to elaborate a successful working schedule.

8.1 WBS diagram

The Work Breakdown Structure is a tree diagram of the performed work. It starts from the root (project), being the underlying levels, main development stages and work packages respectively. The following Figure 8.1 shows how the tasks have been structured.

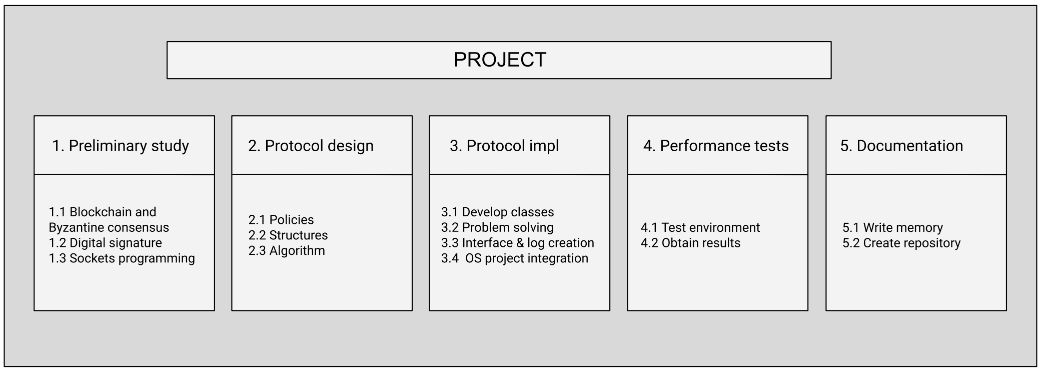


Figure 8.1: WBS diagram of the project.

8.2 Work packages

The following points correspond to the breakdown of the stages needed to achieve the goals of the project.

8.2.1 Preliminary study

The base of the project is founded on previous knowledge. Information search and synthesis have created a solid background to approach future steps.

- **Blockchain and Byzantine:** Accomplish a deep study about what Byzantine consensus is and how blockchain can be used in distributed environments.
- **Digital signature:** Determine the bases about digital signature and its usage in C++.
- **Sockets programming:** Understand how sockets work and their usage in C++.

8.2.2 Protocol design

Once the theoretical base is concluded, the design of a custom network protocol that ensures data integrity on distributed blockchain environments has been performed.

- **Policies:** Define a series of rules and constraints the protocol has to take into account.
- **Structures:** Determine an appropriate way to represent the agents of the algorithm as well as the modules for future implementations.
- **Algorithm:** Elaborate the main logics of the algorithm involving deterministic state machines.

8.2.3 Protocol implementation

After the protocol is designed, a custom C++ application has been developed.

- **Development of the classes:** Designed modules are materialized into classes. Inheritance and singleton pattern implementation techniques are used.

- **Problem solving:** Debug and little changes on the design are performed to adjust both design and implementation.
- **Interface and log creation:** The user level interaction scope is established as well as the feedback provided by the execution mode.
- **Integration with OS daemon:** Agreement between developers and knowledge sharing is performed.

8.2.4 Performance tests

The network testing has to be performed once the whole development is carried out. A good metric to measure network scalability in a local environment has to be chosen.

- **Build a test environment:** Deployment scripts and code adaptation to get datagram broadcast needed time.
- **Obtain results:** The obtained results are analyzed and conclusions extracted.

8.2.5 Documentation

Once the development is finished, an extensive documentation phase has been carried out. The documentation tries to summarize the main keys of the project.

- **Project memory:** The whole development of the project is written and documented.
- **Code repository:** The elaborated application is commented, a user manual is created and all the files are uploaded to a GitHub repository.

8.3 Time estimation and deviations

After defining work stages and packages, a time estimation can be done to evaluate the a priori required elaboration time. Once the project is concluded, the real time taken can be compared with the hypothesized one. The following Table 8.1 gathers both a priori and a posteriori times.

	Estimated time	**Final time**
Management phase	35	35
Planning	10	10
Communication	20	15
Preliminary Study	33	48
Distributed blockchain documentation	12	16
Byzantine consensus documentation	13	16
Digital signature documentation	4	8
Sockets in C documentation	4	8
Protocol design	90	90
Define policies	20	20
Create data structures	20	20
Development of the algorithm	50	50
Protocol implementation	88	125
Pose a C++ development of the design	50	60
Adapt the design to aroused problems	30	60
Integration between projects	8	5
Performance tests	20	17
Creation of a test environment	13	10
Obtain results	7	7
Documentation	38	38
Memory of the project	30	30
Code repository	8	8
Total amount of time	304	333

Table 8.1: Estimation of tasks and their final required time.

Although the idea of the project came to mind in July, 2020, the official project was formalized in December, 2020. The milestones of it are collected in Figure 8.2.

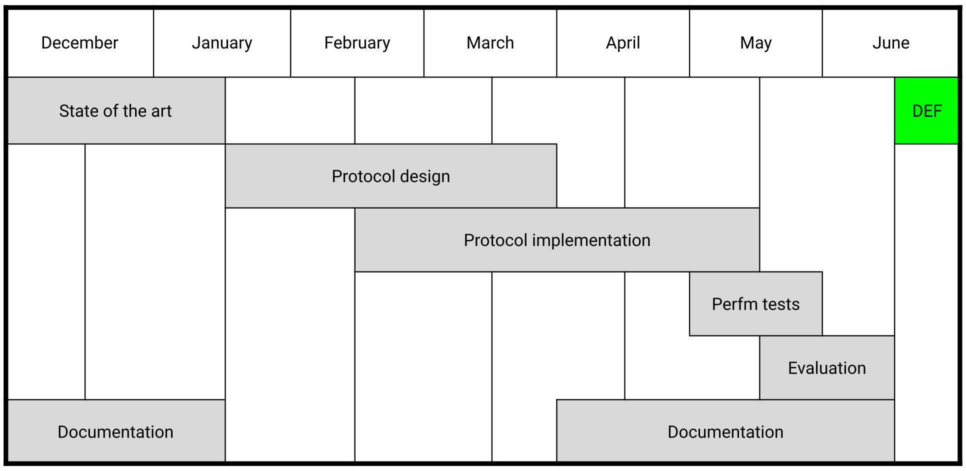


Figure 8.2: Simplified Gantt diagram of the project.

8.4 Deviation analysis

8.4.1 Complex documentation material

In the previous Table 8.1, a considerable time deviation in the preliminary study phase can be appreciated. On the one hand, the documentation about Byzantine consensus requires a high amount of knowledge in areas such as consistency and distributed systems. Therefore, extracting valuable information from esoteric texts is very time consuming. On the other hand, documentation about digital signature and OS sockets was performed from the prism of future C / C++ implementations. Hence, specific features of C++ cryptographic libraries had to be compulsorily analyzed.

8.4.2 Implementation complexity

Undoubtedly, implementation has been the most challenging part of the project. Its complexity was higher than expected and some extra hours had to be employed. Creating a fully working distributed system, taking into account as many consistency constraints as possible, has been beyond the original schedule. Furthermore, C++ programming language was chosen due to its performance and plasticity specifications. Although some aspects of C and C++ are quite similar, the initial lack of experience with C++ made the first implementation steps harder. However, research on online forums and *man* pages documentation made the development gear move on.

8.5 COVID-19 latent risk

This has been the second year affected by the SARS-CoV2, a global pandemic that caused millions of deaths and threatened the welfare state as currently known. In the race of mitigating the effects of the virus, The Spanish government maintained the state of alarm until May, 2021. That supposed a constant uncertainty about how the imposed restrictions situation could evolve. Fortunately, no global lock-downs were needed, as a result of an arduous vaccine campaign. The project was implicitly affected by COVID-19, because the working scenario was full of uncertainties; possible communication problems, lack of access to material and so on. Hence, the risk was mitigated establishing online secondary communication channels and hoarding all the needed resources with enough time.

9. CHAPTER

Conclusions and future work

Some interesting conclusions can be extracted at this point. However, having built such a complex mechanism, opens the door to an uncountable number of possible improvements and future versions. There is still a lot of work to be done in the integrity monitoring area. The following answers round up the initial approaches.

1. Can integrity be monitored in a distributed, secure and resource cost reduced way, using blockchain technology and digital signature?

Yes, it is possible. As shown in previous chapters, using reduced datagrams with the structure of Figure 4.4, is secure and simple. Data confidentiality is granted because a hash descriptor is sent throughout the network, with its random appearance and non-return properties. So, no additional confidentiality mechanisms have to be added, because anyone can read the sent data with no bad consequences. Moreover, authenticity, integrity and non-repudiation are granted by digital signature; the entire datagram is signed with the private key of the sender. Therefore, with just two datagrams (request and hash payload), the filesystem description of a node can be updated. Finally, Blockchain technology allows to maintain a simplified history of all the network activity. Each time a remarkable event happens, the chain is updated. In addition, the protocol assures that $\frac{2}{3}$ of the nodes have at least the same chain value.

1. In the current scenario, are P2P networks the best approach to create a scalable and independent network protocol?

Yes, they are. P2P networks allow not having a centralised entity in charge of controlling the network. The server and client are integrated in the same implementation and access the same memory structures, in a fast way, using threads. Hence, no databases or interchange files have to be used elsewhere. Based on performed experiments, the execution time to send and receive a datagram to and from all the network components is directly proportional to the node number. In other words, both node number and execution delay increase or decrease at the same rate. The delay increase does not change its growth rate to a more pessimistic one from a determined node number on.

As mentioned previously, the design and development carried out, are the first rock of a more bigger and promising application. Although the protocol has possible vulnerabilities, analyzed in Section 5.6, the following lines will cover possible improvements beyond the current implementation.

The main milestone of future approaches is to get a version with better scalability results. Being the memory usage of the algorithm and broadcast performance completely independent from the number of nodes; always should remain the same. In order to achieve it, maintaining the main design of the implementation, interlaced smaller networks could be created; as seen in Figure 9.1. In that way, each node will be in more than a network simultaneously. The networks and all their components will be completely independent, but they will have a common meeting point; e.g. when one of the networks is declared as mistrusted, a flag will be set, indicating that the node must isolate itself from all the networks. Then, the node will be declared as not trusted by other networks, stating that a problem occurred in the node or in a network related to it and so on.

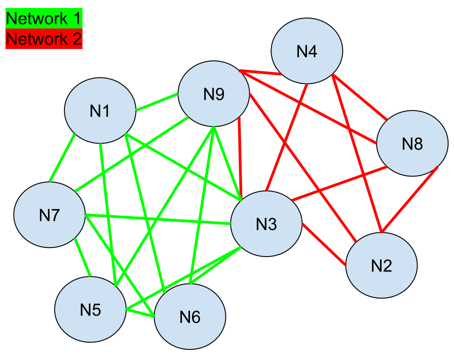


Figure 9.1: Two networks interlaced by two nodes.

Other possible improvements to take into consideration are:

- Perform staggered broadcasts using tree propagation.
- Perform datagram hashing to make the signature process lighter.
- Add a mechanism to reconnect / reset non-trusted nodes.
- Add a mechanism to add more nodes once the network is deployed.
- Add more states to the network and nodes to make a more complex deterministic state machine; not just trusted or not trusted.
- Reset synchronization number of the nodes to prevent integer overflow.
- Make an approach using PoW to extend the scope of the protocol to non permissioned environments.
- Create an aesthetic, user friendly web interface and alert system.

A. APPENDIX

Appendix

A.1 UML diagram

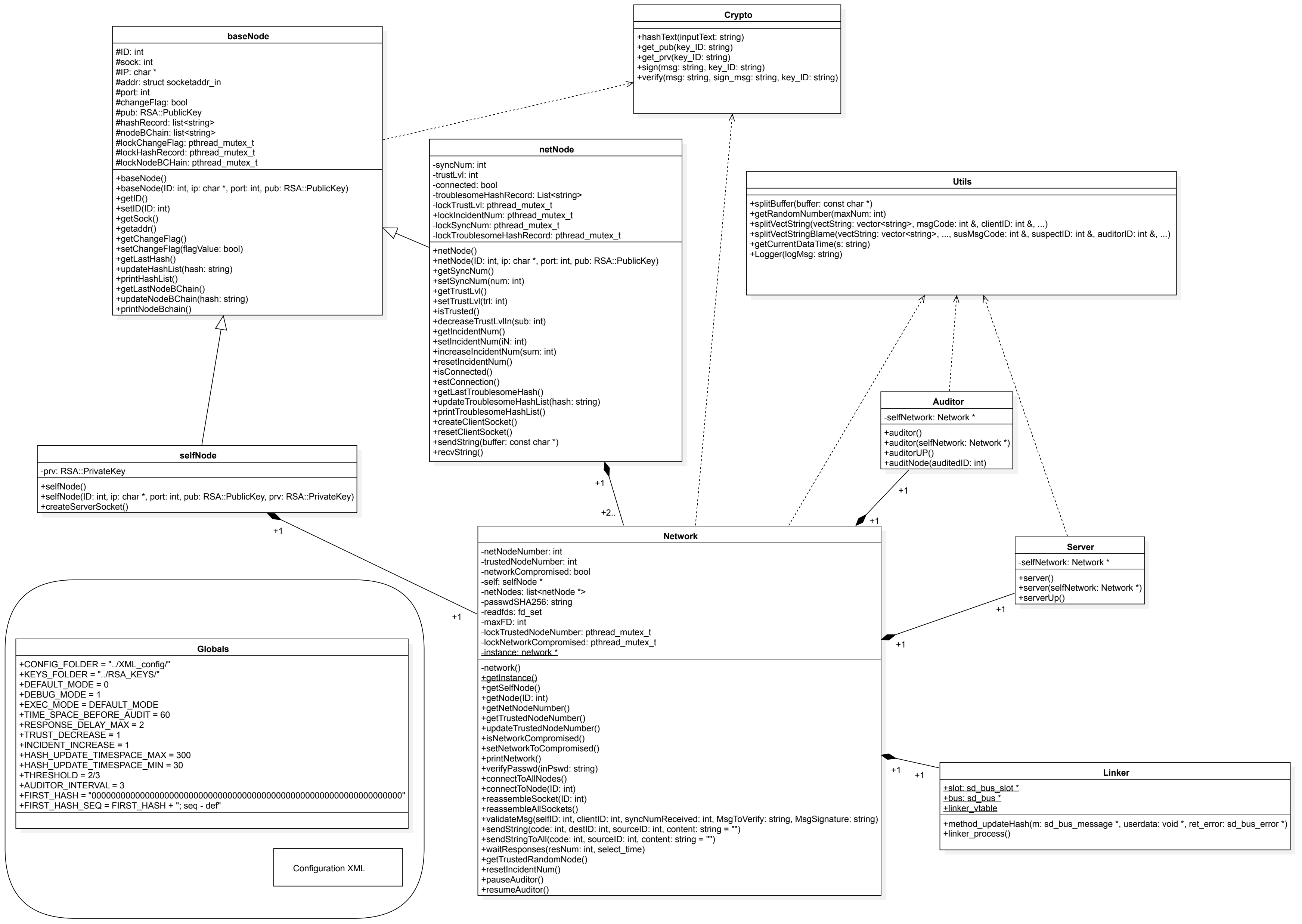

baseNode
#ID: int
#sock: int
#IP: char *
#addr: struct socketaddr_in
#port: int
#changeFlag: bool
#pub: RSA::PublicKey
#hashRecord: list<string>
#nodeBChain: list<string>
#lockChangeFlag: pthread_mutex_t
#lockHashRecord: pthread_mutex_t
#lockNodeBCHain: pthread_mutex_t
+baseNode()
+baseNode(ID: int, ip: char *, port: int, pub: RSA::PublicKey)
+getID()
+setID(ID: int)
+getSock()
+getaddr()
+getChangeFlag()
+setChangeFlag(flagValue: bool)
+getLastHash()
+updateHashList(hash: string)
+printHashList()
+getLastNodeBChain()
+updateNodeBChain(hash: string)
+printNodeBchain()
Crypto
+hashText(inputText: string)
+get_pub(key_ID: string)
+get_prv(key_ID: string)
+sign(msg: string, key_ID: string)
+verify(msg: string, sign_msg: string, key_ID: string)
netNode
-syncNum: int
-trustLvl: int
-connected: bool
-troublesomeHashRecord: List<string>
-lockTrustLvl: pthread_mutex_t
+lockIncidentNum: pthread_mutex_t
-lockSyncNum: pthread_mutex_t
-lockTroublesomeHashRecord: pthread_mutex_t
+netNode()
+netNode(ID: int, ip: char *, port: int, pub: RSA::PublicKey)
+getSyncNum()
+setSyncNum(num: int)
+getTrustLvl()
+setTrustLvl(trl: int)
+isTrusted()
+decreaseTrustLvlIn(sub: int)
+getIncidentNum()
+setIncidentNum(iN: int)
+increaseIncidentNum(sum: int)
+resetIncidentNum()
+isConnected()
+estConnection()
+getLastTroublesomeHash()
+updateTroublesomeHashList(hash: string)
+printTroublesomeHashList()
+createClientSocket()
+resetClientSocket()
+sendString(buffer: const char *)
+recvString()
Utils
+splitBuffer(buffer: const char *)
+getRandomNumber(maxNum: int)
+splitVectString(vectString: vector<string>, msgCode: int &, clientID: int &, ...)
+splitVectStringBlame(vectString: vector<string>, ..., susMsgCode: int &, suspectID: int &, auditorID: int &, ...)
+getCurrentDataTime(s: string)
+Logger(logMsg: string)
Auditor
-selfNetwork: Network *
+auditor()
+auditor(selfNetwork: Network *)
+auditorUP()
+auditNode(auditedID: int)
selfNode
-prv: RSA::PrivateKey
+selfNode()
+selfNode(ID: int, ip: char *, port: int, pub: RSA::PublicKey, prv: RSA::PrivateKey)
+createServerSocket()
+1
+1
+2..
+1
+1
+1
Network
-netNodeNumber: int
-trustedNodeNumber: int
-networkCompromised: bool
-self: selfNode *
-netNodes: list<netNode *>
-passwdSHA256: string
-readfds: fd_set
-maxFD: int
-lockTrustedNodeNumber: pthread_mutex_t
-lockNetworkCompromised: pthread_mutex_t
-instance: network *
-network()
+getInstance()
+getSelfNode()
+getNode(ID: int)
+getNetNodeNumber()
+getTrustedNodeNumber()
+updateTrustedNodeNumber()
+isNetworkCompromised()
+setNetworkToCompromised()
+printNetwork()
+verifyPasswd(inPswd: string)
+connectToAllNodes()
+connectToNode(ID: int)
+reassembleSocket(ID: int)
+reassembleAllSockets()
+validateMsg(selfID: int, clientID: int, syncNumReceived: int, MsgToVerify: string, MsgSignature: string)
+sendString(code: int, destID: int, sourceID: int, content: string = "")
+sendStringToAll(code: int, sourceID: int, content: string = "")
+waitResponses(resNum: int, select_time)
+getTrustedRandomNode()
+resetIncidentNum()
+pauseAuditor()
+resumeAuditor()
Server
-selfNetwork: Network *
+server()
+server(selfNetwork: Network *)
+serverUp()
+1
+1
+1
+1
+1
Globals
+CONFIG_FOLDER = "../XML_config/"
+KEYS_FOLDER = "../RSA_KEYS/"
+DEFAULT_MODE = 0
+DEBUG_MODE = 1
+EXEC_MODE = DEFAULT_MODE
+TIME_SPACE_BEFORE_AUDIT = 60
+RESPONSE_DELAY_MAX = 2
+TRUST_DECREASE = 1
+INCIDENT_INCREASE = 1
+HASH_UPDATE_TIMESPACE_MAX = 300
+HASH_UPDATE_TIMESPACE_MIN = 30
+THRESHOLD = 2/3
+AUDITOR_INTERVAL = 3
+FIRST_HASH = "0000000000000000000000000000000000000000000000000000000000000000"
+FIRST_HASH_SEQ = FIRST_HASH + "; seq - def"
Configuration XML
Linker
+slot: sd_bus_slot *
+bus: sd_bus *
+linker_vtable
+method_updateHash(m: sd_bus_message *, userdata: void *, ret_error: sd_bus_error *)
+linker_process()

Figure A.1: UML class diagram of the implementation.

A.2 Makefile

```
CC=g++
IDIR = ../src
DEPS = main.hpp crypto.hpp network.hpp server.hpp linker.hpp auditor.hpp baseNode.hpp netNode.hpp selfNode.hpp utils.hpp globals.hpp
OBJ = main.o crypto.o network.o server.o linker.o auditor.o baseNode.o netNode.o selfNode.o utils.o globals.o
CFLAGS=-I .
LIBS = -g -lcryptopp -lpthread -lsystemd
%.o: %.c $(DEPS)
        $(CC) -std=c++1y -c -o $@ $< $(CFLAGS)
run: $(OBJ)
        $(CC) -std=c++1y -o $@ $^ $(CFLAGS) $(LIBS)
```

A.3 deploy.sh

```
#!/bin/sh
#mkdir RSA_keys
#mkdir XML_config
node_number=$1
generate_keys_and_confs=$2
if [ $node_number -lt '4' ]; then
    echo "Error, enter valid node number (> 4) and generator g (optional) \n";
    exit 1
fi

if [ -z "$generate_keys_and_confs" ]; then
    echo "Deploting network locally \n"
    for i in $(seq 1 $node_number)
        do
            gnome-terminal -x sh -c "./run $i ; bash"
        done
else
    echo "Generating keys and configurations... \n"
    for i in $(seq 1 $node_number)
        do
            ../deploy_utils/generate_keys $i
            python3 ../deploy_utils/generate_configs.py $i $1
        done
fi
```

A.4 generate_keys.cpp

```
#include "generate_keys.hpp"

void generate_keys(std::string key_ID)
{
  CryptoPP::AutoSeededRandomPool prng;

  CryptoPP::InvertibleRSAFunction params;
  params.GenerateRandomWithKeySize(prng, 2048);
  CryptoPP::RSA::PrivateKey prv(params);
  CryptoPP::RSA::PublicKey pub(params);

```

```
    std::string name = "../RSA_keys/RSA_prv" + key_ID + ".der";
    CryptoPP::FileSink output1(name.c_str()); //Convert to char*
    prv.DEREncode(output1);

    name = "../RSA_keys/RSA_pub" + key_ID + ".der";
    CryptoPP::FileSink output2(name.c_str());
    pub.DEREncode(output2);
  }

int main(int argc, char *argv[])
{
    generate_keys(argv[1]);
}
```

Listing A.1: Server thread method and call in main.

A.5 generate_configs.py

```
#!/bin/python
import sys
import xml.etree.cElementTree as ET
ident = int(sys.argv[1])
totalNodeNumber = int(sys.argv[2])
def_ip = "127.0.0.1"
def_port = 25570
config = ET.Element("config")
ex_mode = ET.SubElement(config, "execution_mode").text = "1"
passwd = ET.SubElement(config, "passwd_sha256").text = "64E8DF328D36F9688A4AD76208CE3FC06AC582552531FF968742645C43DF1930"
node_self = ET.SubElement(config, "node_self")
idd = ET.SubElement(node_self, "id").text = str(ident)
ip = ET.SubElement(node_self, "ip").text = def_ip
port = ET.SubElement(node_self, "port").text = str(def_port + ident - 1)
pub = ET.SubElement(node_self, "pub").text = "../RSA_keys/RSA_pub" + str(ident) + ".der"
prv = ET.SubElement(node_self, "prv").text = "../RSA_keys/RSA_prv" + str(ident) + ".der"
network = ET.SubElement(config, "network")
for i in range(totalNodeNumber):
    i = i+1
    if (i != ident):
        node = ET.SubElement(network, "node")
        idd = ET.SubElement(node, "id").text = str(i)
        ip = ET.SubElement(node, "ip").text = def_ip
        port = ET.SubElement(node, "port").text = str(def_port + i - 1)
        pub = ET.SubElement(node, "pub").text = "../RSA_keys/RSA_pub" + str(i) + ".der"
tree = ET.ElementTree(config)
path = "../XML_config/config" + str(ident) + ".xml"
tree.write(path)
```

A.6 Tables

Node number	10	20	40	80	160
Mean exec time (ms)	19.9463	39.69625	70.11875	120.057625	289.733125

Table A.1: Mean values from Table A.1 columns.

Node Number	10	20	40	80	160
Exec randNode 1 (ms)	23.21	37.36	79.19	122.37	246.5
Exec randNode 2 (ms)	19.18	36.51	66.73	130.46	344.05
Exec randNode 3 (ms)	20.61	37.11	71.1	126.48	283.145
Exec randNode 4 (ms)	17.85	36.28	89.83	118.091	266.34
Exec randNode 5 (ms)	21.38	39.41	67.42	114.35	235.08
Exec randNode 6 (ms)	19.36	39.04	64.65	110.49	339.45
Exec randNode 7 (ms)	19.94	57.35	62.6	109.51	219.6
Exec randNode 8 (ms)	18.04	34.51	59.43	128.71	383.7

Table A.2: Execution time samples of 8 random nodes.

A.7 Online repository

The implementation is publicly available in the following GitHub repository:

http://github.com/aitorb16/Blockchain-based-integrity-monitoring-distributed-net-protocol

Bibliography

[Cachin and Vukolic, 2017] Cachin, C. and Vukolic, M. (2017). Blockchain consensus protocols in the wild.

[Douceur, 2002] Douceur, J. R. (2002). International workshop on peer-to-peer systems; the sybil attack.

[Fernández-Bravo, 2018] Fernández-Bravo, F. J. (2018). Consenso bizantino y blockchain.

[Herrero, 2021] Herrero, A. (2021). File integrity monitoring on linux systems.

[Schneider, 1990] Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach.