ECOLE
POLYTECHNIQUE
DE BRUXELLES

MA1 - IRIF

Communication Network

# Design and Implementation of a TOR Network

https://github.com/iQuad427/TOR_Network_Python

**Authors**

Saïd Belaroussi

Eliot Cosyn

Mickaïl Mavros

Quentin Roels

**Course**

ELEC-H417

**Professor**

Jean-Michel Dricot

Denis Verstraeten

Wilson Daubry

**Academic Year**

2022-2023

# Table of contents

# 1 Design and Architecture

## 1.1 TOR Network

### 1.1.1 Nodes topology

We decided to create a single class Node that contains all the tools required to discuss through the TOR network (client part) and be part of the network (relay part). When a node is instantiated, it can be a client, a relay or both at the same time. We can then use the node object and its tools to create any application we would want to do, such as a challenge-response authentication with a server outside of the TOR network, which is exactly what we did for this project.

### 1.1.2 Connecting to the network

We decided to implement our project in a fully peer-to-peer manner, which means that there is no authority server which communicates with each and every node of the network. When a node want to connect to the network, it does so by requesting a list of known nodes from an already-connected node. To start up the network, we instantiate the four primordial nodes, 3 relays and 1 exit node, each knowing the three others. Those nodes will always be connected, their address can thus be hard coded in the nodes.

To manage this register, we created a Phonebook, which is a custom class containing all the entries that the node want to remember. It also has a method that creates a route through the nodes of the network and ends up at an exit node.

### 1.1.3 Creating a path

The typical length of a path in the original TOR project is 3 node long (including the exit node). Our program shuffle the list of all possible nodes it knows and pops them until only 3 are remaining. It then ends the process by adding a randomly chosen exit node.

### 1.1.4 Sending and returning messages

Our TOR network works by opening a return path while sending the client's message. The exit node can then send the server's response back through this opened path. This implementation allows for basic request-response communication between a client in our network and the server they want to reach. Additionally, we can choose to close or not the path after the response is sent back.

When we choose to not close the path, we allow for faster download speeds through the TOR network, which is one of the major downsides of the original Onion Router. However, it also consumes more resources from the nodes that form the path, as they constantly have to transmit packets

## 1.2 Challenge-Response

We designed our challenge-response protocol to be compatible with any request-response communication protocol, and so to work with our TOR network, which expect to receive something only if it asked for it.

### 1.2.1 Signing up

Before the challenge-response phase, the client signs up to the server by sending their username and the hash of their password (keeping secret the possible recurrences of his password). The server can then save this tuple in their register as a new user.

Communication between the client and server is encrypted using RSA by exchanging their public keys with each other. So that the exit node can't have any information about the messages it forwards or backward.

### 1.2.2 Challenge-response

The challenge-response occurs during the sign-in process. The client asks the server to sign them in, and the server responds with a challenge, which is unique to each user and each user's connection. Both the client and the server compute the encryption of this challenge using the hash of the password as their shared AES symmetric key. The client sends its version of the encryption to the server, which compares the client's answer to its own and responds with the connection status: either positive if the authentication was successful, or negative if it failed.

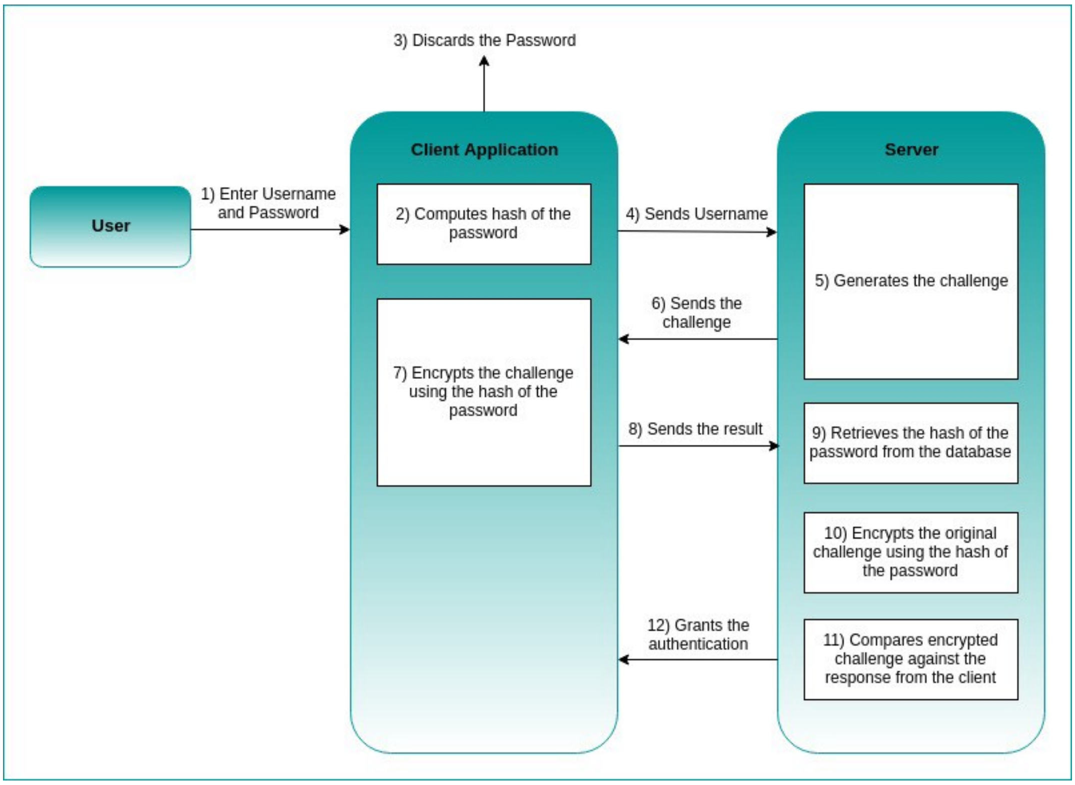
3) Discards the Password
User
1) Enter Username and Password
Client Application
2) Computes hash of the password
7) Encrypts the challenge using the hash of the password
4) Sends Username
6) Sends the challenge
8) Sends the result
12) Grants the authentication
Server
5) Generates the challenge
9) Retrieves the hash of the password from the database
10) Encrypts the original challenge using the hash of the password
11) Compares encrypted challenge against the response from the client

Figure 1.1: Sign in process to the authentication server

The client can also request to be disconnected from the server, which does not remove them from the register, but will require the next instance of the connection with the server to authenticate themselves by signing in again.

### 1.2.3 Messages topology

In order for the server to communicate with a client, we needed to ensure that they could understand each other. To do this, we designed a message structure that would be well-suited to a request-response protocol.

- First, we had to inform the server of who was talking with them, we thus added the username to the message.
- Secondly, we had to inform the server of what was the action we wanted them to do, this will be the request part of the message.
- Finally, we needed to include any additional information required to fulfill the client's request. This part would be the end of the message.

### Final topology

The final topology looks like that : **encoding:tag:username:request:content**

Where,

- Encoding tells the receiver what to do with the content. Since messages transmitted over sockets are in bytes, we need to encode them. However, some things should not be decoded - for example, an encryption should remain in bytes because it cannot be decoded without being decrypted first.
- Tag is the signature made by the sender over the username:request:content part of the message
- The username is the unique identifier of an account on the server side. It is used to determine which password to compare with, and what key to use for authenticating the message. In the end, it identifies which user sent the message.
- The request is the unique identifier of the request that is made to the receiver, telling them what to do or the status of what they requested.
- The content is the data required by the request. For example, if the request is a sign-up, the user will send their password in the content part of the message.

# 2

# Implementation

## 2.1 Technical implementation

Our implementation is entirely based on socket and thread programming using Python.

## 2.2 TOR Network

### 2.2.1 Node

As previously mentioned, a Node object can function as both a relay and a client, which means it has the tools to forward packets and also the tools to send and receive packets through/from the TOR network. When functioning as a relay, the node performs two simultaneous actions: forwarding and backwarding.

FORWARDING : It launches a loop that listens for new connections on the relay's forwarding port. For each new connection accepted, it removes a layer of the received onion to find out which is the next node to forward the onion to, and then proceeds to send it. If the relay is an exit node, it can send the message when it encounters the end of the onion string "send" after the next address.

To enable backwarding, each time a node forwards a packet, it updates its routing table. This routing table is not used for forwarding, as each onion contains the necessary information to forward it from the client to the server.

Note: we decided that any relay could also function as an exit node if desired. It is the responsibility of the client to send and receive through an encrypted route with the servers they want to communicate with, in order to make it impossible for the exit nodes to understand the nature of what is being sent.

BACKWARDING : It launches two loops: one is used to receive information on which nodes want to send something back (i.e. those in the routing table that we forwarded something to), while the other loop connects with them and sends their message back using the routing table updated during the forwarding.

On the way back, we do not encrypt the message, but we sign it. Therefore there is no onion only a simple message.

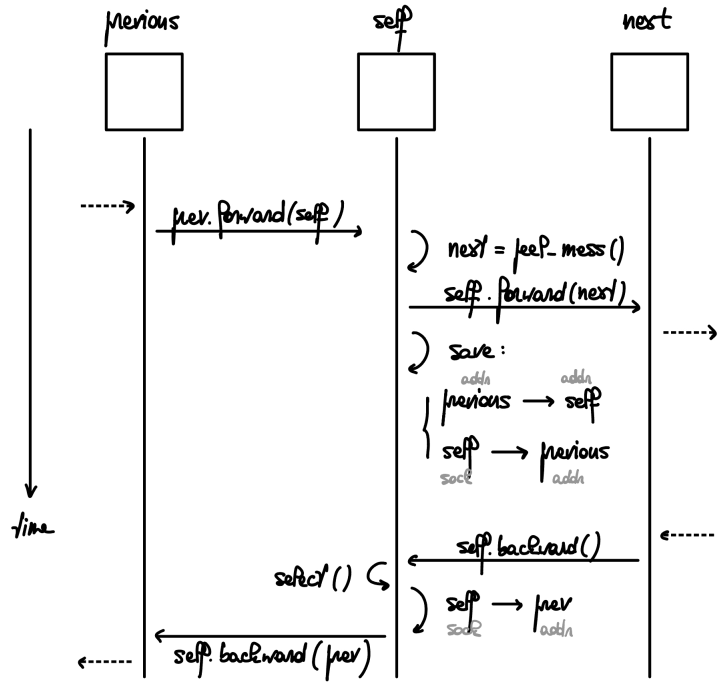

Figure 2.1: Sign in process to the authentication server

### 2.2.2 Listening

In order for the nodes to be able to update their register, all nodes need to be listening to each other. Therefore, we have a loop that accepts connections on the main port of the node in order to respond to the various requests from the network regarding their public key or Phonebook addresses.

### 2.2.3 Sending and receiving

Each node has access to a "send" and a "receive" method. To function as a client, it only needs to send a message - it's that simple. Sending a message involves two steps:

- Define the path through the network using its phonebook addresses.

- Encrypt the message as an onion to allow it to be forwarded through the TOR network. The message can then be sent by connecting to the next node in the path.

To receive a message, the node uses the "recv" method to read the first element in its receive buffer. This buffer is updated when the node receives a packet to be forwarded back that has the node itself as the destination.

## 2.3 Challenge-Response

### 2.3.1 Server loop

The server is a loop designed to respond to specific requests. It accepts a connection, receives a message, parses it, looks at the request, and executes code depending on the request. It has been built to be a request-response server, and does not keep a connection open with the exit node between two messages from the same client. It responds and then closes the connection.

To allow a client to stay connected from the server's perspective, it has been built as a state machine. It keeps the state of its users in memory and acts accordingly. For example, a user who already exists cannot sign up again, and a user must be disconnected before signing in.

While the state of the user is "connected", the server will not require the user's password.

Verifying client identity: To provide an additional layer of authentication and to ensure that no one can impersonate a client after they have log into the server, the client signs each message they send to the server with their own RSA private key, and the server verifies the messages using the client's public key.

### 2.3.2 Generating challenge

For the challenge, we wanted something that was dependent on the user and changed with each connection of the client. We therefore added a nonce based on the number of connections the user has made, which is stored as a statistic in the user's profile on the server.

Challenge : « $bonjour\_username\_nonce$ »

# 3 Innovation and Creativity

## 3.1 Resilience

While the original TOR Network has authority servers that defines the path that the user packets are going to take, we implemented our project has a fully relay based network where each node can manage its register of address as wanted and define its path by itself. This allows for the Network to still be online even if the kernel nodes were to shut down, would it be for maintenance or because of an attack.

## 3.2 Reusability

To prove that our code has a good reusabilty, we implemented the challenge-response server as an application over the lower level implementation of the TOR Network. In fact, our project is built as an API, there is no need for someone wanting to add new functionalities to the project will not need to do socket programming, he can directly use send and recv methods of the Node class to implement a communication through our TOR Network.

## 3.3 Security

We implemented exactly the same encryption scheme as for the original TOR network, i.e. hybrid encryption for forwarding of packets. This way, we ensure that an adversary sniffing packets between two nodes would get an extra hard-time to recover any information about the client.

Many of our ideas went into making a well designed protocol for the challenge-response server that would be as secure as possible. First we implemented a challenge-response that was a bit more secure than the other (cf. Implementation), then we made the client tag the messages he sends to the server and we added a pair of keys depending on the connection session to allow different computers to connect at the same account if they have the right password.

## 3.4 Backwarding strategy

One of the main innovation we added to the TOR was our backwarding strategy that allows for easy handling of many yet to come responses to everything we forwarded. The way we achieved this is explained in the corresponding section of design and implementation.

This backwarding protocol allows for two different behaviour : 1. pure request-response, where the network forget about the path after it forwarded the answer to the request. 2. one-request-many-answers, where the path created through the network for backwarding can be used multiple times before being deleted. An example for both application would be connecting to a server through request-response and start downloading from the server by one-request-many-answers.

## 3.5 Single process application

We implemented our TOR relays in a way that make any node of the network a single process program. This way, we avoided the creation of many processes while the node was working for the network (which means that it won't open as many processes as Google Chrome). It is then easier to kill the main process if it becomes too expensive for the processor. This is also positive for the quantity of thread that the operating system has to manage at the same time. If you launch a TOR relay on your computer, it will have at most 3 running thread most of the time; one checking for questions of the network peers, one forwarding the packets the node receives, and a last one that backward the responses of the forwarded packets.

The innovation comes from the fact that we could have implemented a protocol that sends a packet and wait for the response without doing anything else and use multiprocess to launch as many process as communication to forward, but because of our implementation we can keep on forwarding other packet while waiting for the responses to come back with a single process. We thus have the full client experience without being scared of getting flooded without being able to stop the relay.

# 4 Challenges

## 4.1 Python related challenges

### 4.1.1 Multi-threading

Our primary idea was to implement a TOR network through an onion routing of type “connect-communicate-disconnect”, where the network would first establish a path from the client to a server and then use that path to forward and backward packets as long as the client was not disconnecting. But multi-threading in Python can be tricky, and for some reason a blocking call in a thread would sometimes cause the whole program to stop.

To solve the problem we had take it into account and reduce to the minimum the use of threads in our program. Some other possibility were considered if the problem was to become too significant, such as using multi-processing or the "Async IO" library. We thus decided to come back to a more request-response based approach that would require less time consuming threads and avoid creating to much of them.

### 4.1.2 Blocking calls management

Blocking calls such as *select()* and *accept()* were a small issue in the implementation. Timeouts were added to prevent threads from being lock for to long.

## 4.2 Conceptual challenges

### 4.2.1 Onion routing

On onion routing itself, some information was really difficult to obtain. Everybody explains onion routing with its forwarding only (which is also the case of the course) without ever talking about the best way to backward. Onion routing take place at the forwarding phase, which is the important part to explain but backwarding was mandatory to implement a challenge-response server, which took us a bit longer than expected because we had to think about it a lot to find out what we could be doing.

### 4.2.2 Managing addresses

Keeping a well-maintained register of addresses during the whole network up-time was a bigger issue than expected. The main solution was to make Phonebook a class to allow it to manage itself. Nodes regularly exchange Phonebooks, after each exchange the updated phonebook tries to be as up-to-date as possible by asking for public keys and refreshing its list of exit nodes. Its also the Phonebook class responsibility to generate new paths of nodes every time its necessary.

### 4.2.3 Backwarding

The implementation of backwarding was a pure conceptual challenge as we had few information on existing methods and found almost no documentation on internet. When a message is forwarded, the address and port used are kept in memory, the node then listen to this port for a response. Once a response is received it gets backwarded to the sender.

## 4.3 Implementation related challenges

Nowadays it exists easier way than socket programming to implement those kind of applications, it is way more difficult to program cleanly with sockets than some other more high level options such as HTTP. Since we didn't know how to use those options, we kept on working with sockets. It required to be really cautious on which address we use, not reusing the same addresses in the same code. The biggest issue was to debug, depending on the operating system, the sockets that are not closed properly close themselves after a period of time that vary and can be really high. Most of the time it results in having to wait a bit before rerunning the code, and overall it is quite a big time loss.

# 5 Conclusion

In this project we implemented a fully functional peer-to-peer TOR network. It is scalable as a new node only needs to know one of the node of the network to be part of it. We also implemented a challenge-response authentication server whose main goal is to demonstrate that clients can communicate anonymously and authenticate through the TOR network without any issues.

If we had to startover the project we would certainly choose to use multi-processing or the ASync-IO library instead of manual multi-threading as this aspect consumed a lot of our time.

To optimise the code we should add a size variability management for the Phonebook class. In bigger networks the Phonebook size can become a problem especially for sharing. We could add a limit to the phonebook size and replace old contacts by newer one once in a while. We could as well transmit a fraction of the phonebook instead of the complete one.