Université Libre de Bruxelles
Section M-IRIFS
Academic Year 2022-2023
ELEC-H-417

COMMUNICATION NETWORKS:
PROTOCOLS AND ARCHITECTURES

TOR network project

Authors

Sacha **Duynslaeger**
Olivier **Ndinga Oba**
Florian **Noussa Yao**
Mohamed **Ouamar**

Professor and assistants

Jean-Michel **Dricot**
Denis **Verstraeten**
Wilson **Daubry**

December 23, 2022

Contents

1 Introduction **1**

2 Architecture **1**

2.1 Server-gateway 1

2.2 Clients 2

2.2.1 Launching client and requesting 2

2.2.2 Encryption 2

2.3 Relays 3

2.4 Challenge-Response based Authentication 4

3 Innovation and creativity **4**

4 Challenges **6**

5 What to do next **7**

6 Conclusion **8**

1 Introduction

TOR, short for The Onion Router, is a free and open-source network that allows users to communicate anonymously over the Internet. It does this by routing Internet traffic through a network of servers, or "nodes," that are spread across the globe. Each node in the network only knows the location of the node immediately before and after it, making it difficult to trace the origin or destination of a given communication.

One of the main benefits of using a Tor network is that it helps to protect users privacy by making it difficult for anyone to track their online activities. This is especially useful for people living in countries where Internet censorship is prevalent, or for individuals who want to keep their online activities private for other reasons.

In this project, we will be implementing the core components of a TOR network using Python. This will include building the infrastructure for routing traffic through the network, as well as implementing the necessary cryptographic protocols to ensure that communications are secure and anonymous. By the end of the project, we will have a functional TOR network that can be used to securely and anonymously communicate with local authenticate server and any internet free access API.

2 Architecture

The architecture of our TOR network is composed on several python scripts. The server/gateway (*gateway.py*), the clients (*client_TOR.py*), the relays (*relay.py*) and the authentication of a client to another server using a password (*client_auth.py* and *server_auth.py*).

2.1 Server-gateway

First, we have the 'server' or 'gateway.' Its role is to register every node that connects to the network, whether it is a relay or client, and maintain a list of the addresses of the relays connected in the network. (Note: For this project, we refer to ports as 'addresses.' Technically, the address of a node is composed of its IP address and its local port, but since we are running the TOR network on a single computer, every IP address used in this project is always 127.0.0.1. Therefore, when we talk about the address, we will be referring to the port, i.e. the only thing that distinguishes different nodes). Every relay and every client that wants to join the TOR network must first connect to the server. The server will also act as a 'gateway,' meaning that it acts like the entry point, the first step if a client want to use the TOR network. When a client requests the list of the current relays in the network, it sends it through TCP. TCP is used for all communication protocols in this project. We chose TCP because of its reliability. In the context of the

TOR network, we decided that it was better to ensure that packets are not lost during transmission, even if it is slower than UDP. Losing packets during the transmission of the encrypted message (also known as the onion) would be quite problematic.

2.2 Clients

The client is a user that connects in the TOR network willing to make requests to go on the internet. We will explain how we developed the client in our code and how it interacts with the relays, while the relays will be separately explained in section 2.3.

2.2.1 Launching client and requesting

When the client is launched, it first connects to the gateway to notify its arrival. When it is about to make a request, it asks the gateway to provide it a list of the addresses of the network's available relays. This list is important because it will allow the clients and relays to communicate directly, according to the Peer To Peer architecture. The client will receive the list from the gateway and will choose a random number of relays r and create a random path using those relays. The client then connects to all of these nodes to request their encryption keys and then successively encrypts the message using those keys as it will be explained in the next subsection. We made the decision to implement the path creation and encryption inside the client rather than in the gateway so that the gateway won't become overloaded in case of many clients requests. Each client handles its own work.

2.2.2 Encryption

For the encryption, the client first connect to each relay that were selected randomly and requests them to compute a encryption key. The relays generate a key with the $fernet$ python library, which primitives include a 128-bit AES in CBC mode and an HMAC with SHA-256, and then send them back.
Afterwards, for preparing the onion, the client encrypts its message r times (r being the random number generated earlier, i.e the length of the path) by doing the following operation :

$$Onion = Enc(k_1, [next_1, Enc(k_2, [next_2, Enc(...Enc(k_r, [next_r, client\_request]))])])$$

With :

- k_1, k_2,...,k_r the encryption keys of the 1st, 2nd, ..., rth relays of the path.
- $next_1$, $next_2$,...,$next_r$ the address of the next hop. For the first $r-1$ relays, $next_{1...r-1}$ is the address of the next relay on the path. For the last relay, $next_r$ is the string "$last\_relay$" that marks the end of the path.

We can observe that each encryption encrypts a list composed of two components : the address of the next hop and the message previously encrypted. We chose to work with list for easier manipulation.
Then, once the onion is ready, the client send it to the first relay on the path generated earlier.

Doing this encryption scheme ensure that no observer can retrieve both the address of the client and the final destination while looking at one specific relay. It is important to note that the response of the requested server will be send back to the client through the same path, and every node in it will encrypt the response to block whoever could be capturing the traffic to see the response. The client will then decrypt back the returning onion to get the response of the server.

2.3 Relays

The relays are nodes in the TOR network that are used for forwarding messages. They are an essential part of the network, as without them, we cannot speak about a TOR network. All nodes can communicate with any other node on the network without the need for an intermediary server, enabling peer-to-peer communication within the network. As previously mentioned and justified, communication between nodes uses the TCP protocol. They can react to two kind of messages :

- Message "*key_request*" : This message can only be sent by a client. When a relay receive this message, it generate a fernet encryption key and send it back to the client.
- Message "*send_to_next*" : this message can be sent by either a client or a relay. When a relay receive this message, it knows that it will receive an "onion", so it waits for it. Once it received it, it peels one layer by decrypting the message. The decrypted message is a list (as shown in 2.2.2). The relay registers the address of the next relay on the path and warns it that it will soon receive an onion (it warns it by sending a "*send_to_next*" message). Then, it sends the onion and wait for a response. Forcing each relay to wait for a response will allow the final destination to reply and send messages to the client. So, the final destination and the client will be able to communicate without the destination knowing the address of the client.

In fact the destination doesn't know the real address of the client because, it only communicate with the last node.

2.4 Challenge-Response based Authentication

The purpose of this part of the project is to enable and verify that a user of our TOR network can actually successfully connect to their account on a Challenge-Response Based Server (for example, using a password) without any leak of this password during the process. By the way the TOR network is designed, thanks to the multiple encryptions, the message (whether it is a password or not) won't be seen by any part of the path. On the surface, this may not seem like a big deal for the project. But this enables us to address two main points: a continuous communication through the tor network and a a known flaw of the TOR network: the exit node. In fact, the exit node, also called the last node, is able to see plaintext messages, which means it can see even the password. The solution for the exit note being the creation of TOR called "TOR hidden service layer" in which users never really exit the TOR network to connect to normal internet servers. In this case, the exit node won't see the message or the password because it will be encrypted for the server and only the server will be able to decrypt it and obtain the password. the case of a continuous communication will be discussed in the limitation part of this report.

For this part of the project, we have created a server that can receive and register users and then allow them to log in using a password. To do this, we send connection command to the exit node through the Tor network, the exit node will send back the response of the server to the client, then until the end of the process the exit node receive from the client the message to send to the server and exit node send back to the client every response of the server.

3 Innovation and creativity

We give here a list of functionalities that we added to our implementation in order to either give a more friendly-usable program, or to give a more realistic network.

1. A launcher : before running the client, the network needs to be launched. To do so, it is required to launch the gateway and to launch several relays. But if we want to launch several relays, it is quite annoying to open a new terminal window every time we want to add a new relay. So we came up with a launcher that will do it for us : the *lauch_network.sh* bash script. This launcher first launches the server and then launches a number of relays given by the user as an argument. Note : this functionality is working only on linux and macOS.

2. A communication interface giving choices : when launching the client, (*client_TOR.py*), the terminal will ask you what you would like to do. It gives you the choice of using the network in two ways : you can learn about the number of public repositories of a

github user, or you can visit a web page. If you do the github option, the username of the user you want information about will be asked. Then, the request sent to the network is a json request.

```
(base) Sachas-MacBook-Pro:network_project sachadune$ python client_TOR.py

Hello ! Welcome to our TOR network.

In this network, you have the choice between opening a web page, exctracting informations
about a github user or authenticate to our server.

What would you like to do ? [web/git/server] : git

Please enter the git username you want informations about : duynslaeger

 QUERY RESULT:
 the user of github duynslaeger has 0 public(s) repository(s)
```

If you chose the web page option, the terminal will ask you the paste the url link of the web page you would like to visit.

```
(base) Sachas-MacBook-Pro:network_project sachadune$ python client_TOR.py

Hello ! Welcome to our TOR network.

In this network, you have the choice between opening a web page, exctracting informations
about a github user or authenticate to our server.

What would you like to do ? [web/git/server] : web

Please paste the link of the web site you would like to visit (https:// required) :
https://www.ecosia.org/
```

We need to give some precisions here : in order to do this, we used the *webbrowser* package that allows to open a web site in the default browser of the computer. The "problem" is that this would work it in a real TOR network because it would be the computer running the last relay of the path that would have its browser opened. But we chose to do it here because we run the whole network in local and that we thought it was an interesting way of simulating an internet access through our TOR network. In a real TOR network, the connection to internet is done by a complex communication between the destination and the client, something we do not have the knowledge to do.

3. Handling relay disconnection : Beside the ability of the network to accept any new relay in the network at any time, we added the ability to handle the disconnection of a relay. Every 5 seconds, the gateway pings the relays to check if they are still available in order to maintain an accurate list of available relays in the network.

4. Registration to the authentication Server: We created a server running a login and sign up process. When a client connect to the server, it is ask to choose between login and sign to be able to login the user first have to sign in. We have also add a hashtable and so

4 Challenges

During this project, we went through several challenges, several were well addressed and other were not. Those challenges were :

1. At the start there is the understanding of the TOR protocol. While it seem simple to understand that it is a computer network, it was necessary to get the main point to get the anonymity such as encryption and the impossibility for nodes to know the rest of the network without loosing the ability to communicate with them.

2. We had a really hard time doing the Challenge Response based Authentication. At first, we did manage to have a server that we could connect and sign up / login but this first version was creating a discussion directly between the client and the server without passing through the TOR network. When we tried to pass through the TOR network, we had a lot of errors and problems. We spent many hours on this problem, for finally having a result that do not satisfy us fully. In fact our Challenge Response based Authentication is not complete. It works only with no more than two relays in the network. Intuitively, we think that the problem might come from the encryption, as if the messages exchanged between the server and the client through the TOR network were lost in the wild (because there were no problems when the messages were not encrypted). Very strange, but unfortunately we did not manage to find solutions in time.

3. The length of the ciphertext resulting from several nested encryptions can get very large, which is a concern because we have to specify to the relay the maximum size of the message that it can receive. It would be ideal to have a stable length for encrypted messages throughout the entire path through the network, but this seems to be impossible due to the fact that the ciphertext becomes larger with the encryption key, the length of the plaintext before encryption, and the number of nodes in the path. Some eventual solutions are explained in the next section.

   To ensure that the message can pass through the network, we can try two solutions: first, use a shorter key, which will result in less security; second, compress the text before or after encryption. Both of these solutions will not result in a constant length for the ciphertext, but they can help reduce its size to fit within the acceptable message size. For simplicity's sake and because of the small size of the requests we hard-coded from the client, we set the maximum message length to an arbitrarily large value.

5 What to do next

Even if our current implementation fulfills the project's requirements, it faces some limitation that we could have improve with more time. We will give here a small description of them :

- For the encrytion scheme, we used the *fernet* encryption from the *cryptography* library. The "problem" with fernet encryption is that the cipher text is bigger than the plaintext we want to encrypt, as explained in the last section. We adapted our relays so they can handle huge messages but still, there is theoretically a limit to the number of node in the network, even if this number is huge. To correct this issue, we could either find an encryption scheme that do not increase the ciphertext size (but we have not find any of them in python), or we could make the relays be able to deal with messages of size depending of the number of relays in the network (but this would have complicated the code and would have increased the size of the script.).

- In our implementation, the clients and the relays are two different entities. With more time, we could have coded the clients so they can become relays while they are not doing any request, or even more, multithread them so they can be a relay and handle their own request in the same time.

- There are several ways to make the IP addresses of the TOR gateway and authentication server available to the clients. Here are a few options:

  Hardcode the IP addresses in the client code: You can include the IP addresses of the TOR gateway and the authentication server as constants in the client code. This is the simplest solution, but it means that if the IP addresses change, we will need to update the client code and redistribute it to the users.

  Use a configuration file: a configuration file that contains the IP addresses of the TOR gateway and the authentication server can be created. The client can read this file at runtime to obtain the necessary information. This allows to easily update the IP addresses without having to redistribute the client code.

  Use a dynamic DNS service: If the IP addresses of the TOR gateway and authentication server are likely to change frequently, a dynamic DNS service to assign them stable hostnames can be used. The client can then look up the IP addresses of these hostnames using standard DNS resolution. This allows to easily update the IP addresses without having to update your client code or configuration files.

Due to time limitation, we implemented the hardcode way.

6 Conclusion

In conclusion, the implementation of a Tor network using Python has been a challenging but rewarding project. By building the necessary infrastructure and implementing cryptographic protocols, we have created a functional network that allows users to communicate anonymously over the Internet. The use of a Tor network is particularly beneficial for individuals living in countries with Internet censorship, or for anyone seeking to protect their online privacy. While there is still room for improvement and further development, the success of this project demonstrates the potential for Python to be used in the creation of secure and anonymous communication networks.