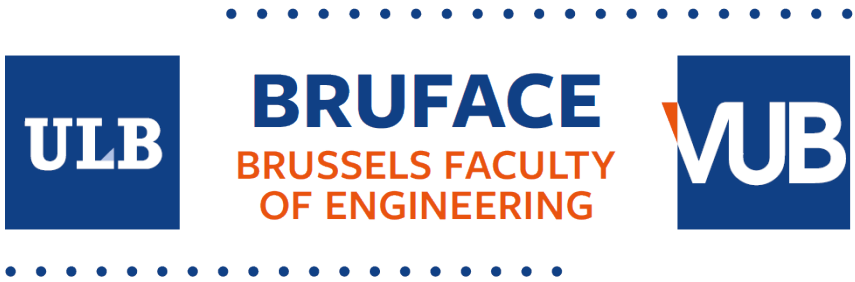

ELEC-H417

Project Report of Group 7

TOR Like Network

Authors:

Andrei ANGHEL-PANTELIMON
Bao HAN
Mouhi al din MAHMALJI
Rodolphe VALICON

Professor:

Jean-Michel DRICOT

Friday 23th December 2022

Contents

1 Introduction **1**

2 Software features and architecture **2**
2.1 Features . . . 2
2.1.1 Asynchronous based python library . . . 2
2.1.2 Peer-to-peer decentralized onion routing network architecture . . . 2
2.1.3 Stream cipher-based cryptography . . . 2
2.1.4 Pool indexing . . . 3
2.1.5 HTTP proxy server . . . 3
2.1.6 Secured simple challenge-response authentication . . . 3
2.2 Architecture . . . 4

3 Working principles and protocols **5**
3.1 Torpydo network working principles . . . 5
3.1.1 Network architecture . . . 5
3.1.2 Connection process with destination . . . 6
3.1.3 Communication process with destination . . . 7
3.2 The TorPyDo Protocol (TPDP) . . . 7
3.2.1 Handshake . . . 7
3.2.2 Error handling . . . 8
3.3 Secured simple challenge-response authentication protocol . . . 8

4 Challenges **10**
4.1 Developing thread-based code . . . 10
4.2 Cyptographic choices . . . 10

5 Conclusion **11**

1 Introduction

As part of the *Networks and Protocols* first-year MSc in Electrical Engineering course given at the Brussels Faculty of Engineering, we had the project of creating a TOR-like network system in Python.

For our project, we've decided to develop this system as a user-friendly Python library named **Torpydo**, capable of fulfilling production-wise needs. We've also designed the communication protocol from scratch, thinking about each step, to ensure maximum security and performance.

The library was developed using an asynchronous paradigm, as it eases up the development of network applications while maximizing performances.

A lot of additional features have been added, such as pool index servers to ease up client access to torpydo nodes; or a simple HTTP proxy to access web pages through a torpydo network.

Finally, aside from the library, an authentication server has been implemented with its corresponding client, connecting to the server through a torpydo network.

This report describes the different implemented features and explains the processes, protocols, and design choices.

2 Software features and architecture

2.1 Features

Before diving into the in-depth explanations of the project, let's do a quick review of the different implemented features.

2.1.1 Asynchronous based python library

As said in the introduction, we have decided to develop our project as a user-friendly python library called **Torpydo**. It seemed logical, as it lets us give simple high-level interfaces to anyone who wants to build anonymity-guaranteeing secured applications, instead of just giving a specific implementation.

Furthermore, as this project is heavily reliant on networking and connection handling, we've decided to design our library using an asynchronous paradigm. This enables a server entity to handle multiple connections on a single thread, and hence greatly simplifies development, since writing thread-safe multi-threaded programs in Python is very tough and verbose. It also increases the performance by avoiding the problem of Python's Global Interpreter Lock (GIL) bottleneck in multi-threaded programs.

2.1.2 Peer-to-peer decentralized onion routing network architecture

As required, our system is peer-to-peer. A client or a node can connect directly to any node of the network without the need for an intermediate server.

Furthermore, the system is decentralized. Anyone can host a new node for a pool. The only central points are the pool index servers that are responsible for maintaining a list of the known online nodes in their pool for clients to choose from. However, these servers are completely optional, as they are not required for a torpydo network to operate, since a client can still construct a path through the network if it knows some node addresses.

2.1.3 Stream cipher-based cryptography

One of the big weaknesses of current TOR implementations is that every node is configured to have a specific role in the network:

- **Entry nodes** are configured to let clients access the TOR network. They know for sure that their source peer is an actual client.
- **Relay nodes** are configured to forward data from node to node. They know for sure that their source and destination are nodes.
- **Exit nodes** are configured to forward data from a relay node to a client's final destination. They know for sure that they are communicating with a client's destination.

These fixed configurations are necessary because of the use of block ciphers (AES 256bits with Cipher Block Chaining mode of operation). Indeed, a block cipher can only encrypt/decrypt blocks of fixed length, and so when a segment of data is not sufficiently long to fill a block, padding needs to be added. This padding should also be removed at the end of the chain, before communicating the data segment to the final destination. Hence the last node of the chain must know that it needs to remove the padding before forwarding data to the destination server.

This is a problem since nodes know a lot of information about their source and destination peers. This vulnerability can be (and has been) exploited by hosting a massive amount of entry and exit nodes. A malicious owner of these nodes has a big statistical chance that clients' data enter and exits the TOR network through his nodes and hence, the eavesdropper can link clear-text data from his exit nodes to the identity of a client at his entry nodes by a statistical traffic analysis, effectively breaking TOR anonymization.

To fix this vulnerability, every node in the network should have the same behavior, and should only know limited information about its source and destination. Knowing that we had the idea of using stream ciphers (AES 256bits with Counter mode of operation) instead of block ciphers. Thereby, data segments of arbitrary length can transit over the torpydo network, and hence every node in the network has the same behavior, and cannot know the nature of its source and destination peer.

In this configuration, the previously mentioned attack, cannot be conducted anymore, as a node's owner eavesdropping over the network cannot tell if the traffic is entering or exiting the network.

2.1.4 Pool indexing

To connect to a final destination through a torpydo network, a client should know the addresses of some nodes. To get access to these addresses, a client can contact a pool index server, which will reply with the list of all the nodes indexed in its pool. A client can hence get multiple lists of nodes from different pools and then chose random nodes to create a path to its destination. This multi-pool feature reinforces security since the nodes from different pools don't have any link and so cannot be owned by the same person.

Pool index servers actively maintain lists of alive nodes, regularly removing nodes considered dead. The nodes of a pool should regularly send heartbeats to their pool index server to be considered alive.

2.1.5 HTTP proxy server

We also have implemented a simple HTTP proxy server, to redirect HTTP requests through a torpydo network. This enables anyone to connect anonymously to web pages simply. This proxy is given as an example code in the `examples` directory of the project's repository.

We also wanted to implement a SOCKS4 proxy server, but lacked of time to do so.

2.1.6 Secured simple challenge-response authentication

As required, we have implemented a server hosting a simple challenge-response authentication system, with a simple cryptographic handshake to prevent sensitive information leakage (like passwords or secret data reserved for authed clients).

2.2 Architecture

As the developed software is a library, we've used the standard structure for a python module. Hence, at the root of the repository, there are the documentation files, in addition to the setup configuration files enabling building and installing the library using `pip`. The actual code of the module lies in the `torpydo` directory.

There is also some example code to show basic usage of the library in the `examples` directory.

Finally, we've added the required authentication server and client in the `test` directory.

A complete overview of the software architecture can be seen in Figure 2.1.

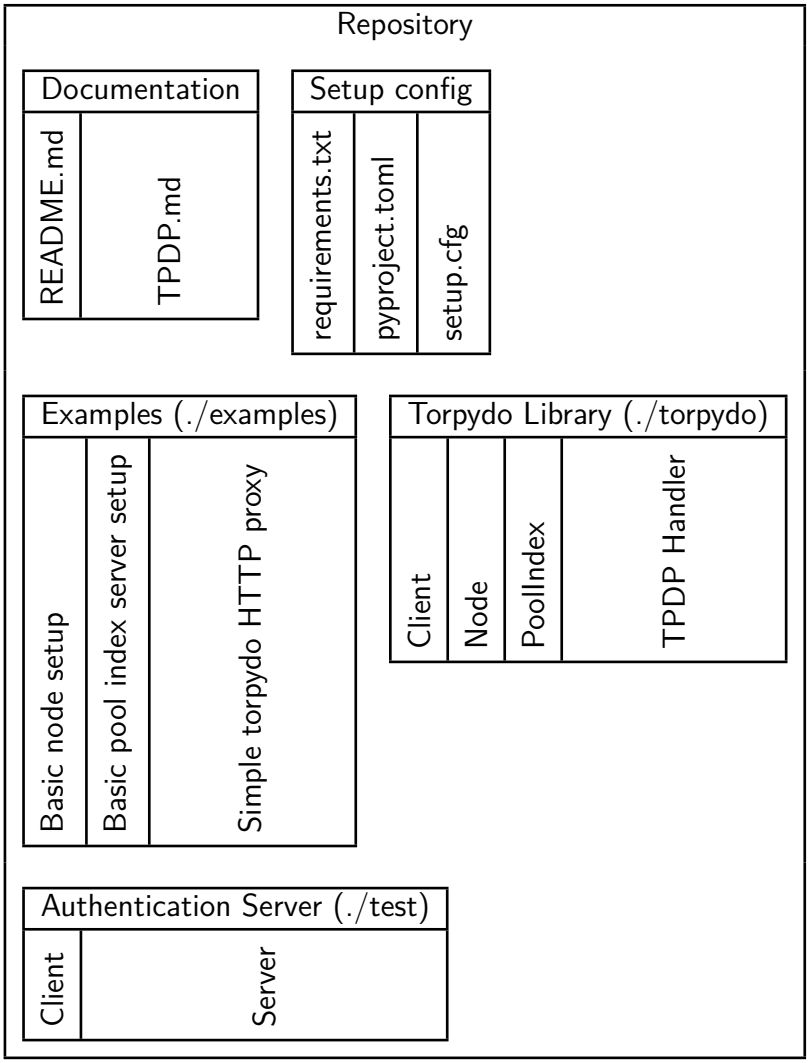

Figure 2.1: Software architecture of Torpydo

3 Working principles and protocols

3.1 Torpydo network working principles

This section describes the working principles of the system we implemented.

3.1.1 Network architecture

A torpydo network is constituted of three linked entities: Clients, Nodes, and Pool index servers. All communications are made through TCP/IP. A scheme of a torpydo network can be seen in Figure 3.1.

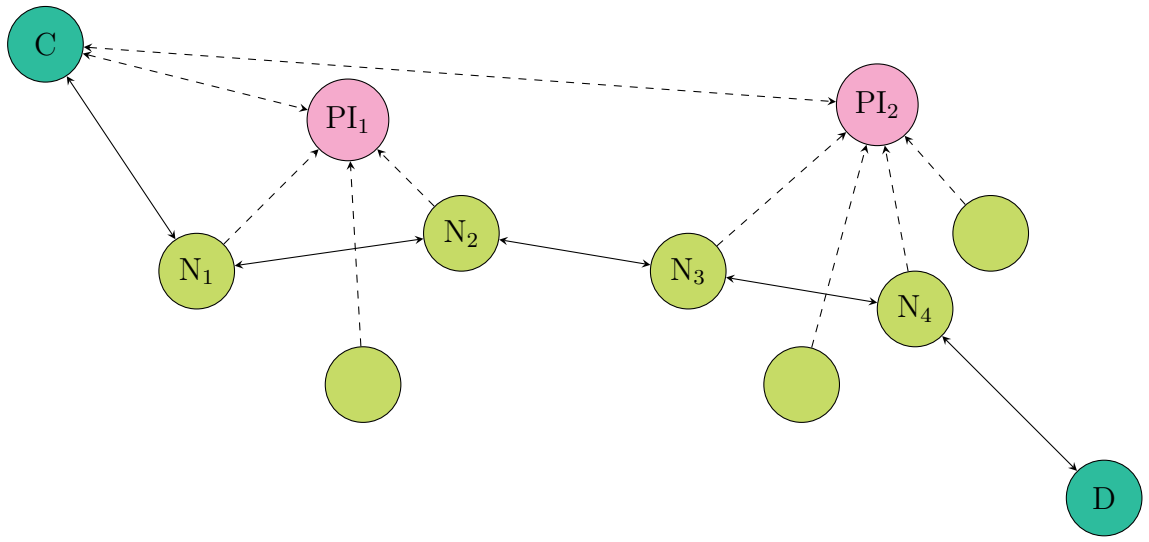

Figure 3.1: Torpydo network architecture. Client C, communicates with its destination D, with the help of two torpydo pools. The client got the addresses of the nodes N_1, N_2, N_3 and N_4 through the pool index servers PI_1 and PI_2. Nodes periodically send heartbeats to their pool index server to notify them that they are still alive.

3.1.1.1 Node

The nodes of a torpydo network are the mean of communication between clients and their desired destinations. They are optionally associated with a pool and if so, they must periodically send a heartbeat to their pool index server to be considered active. During a handshake process, they can be connected to a destination peer, at the will of a client. Once connected to its destination peer, a node will decrypt any data segment received from its source peer using the shared key generated during the handshake with the client and forward the decrypted data to its destination peer. Similarly, the node will encrypt

any data segment received from its destination peer and forward the encrypted data to its source peer.

A node can handle multiple connections at the same time and hence can have multiple linked source and destination peers.

3.1.1.2 Pool index

A pool index is a server that maintains a list of nodes. It is used by clients to get a list of nodes from which to generate a random path to their destination. Every node of a pool must send periodic heartbeats to its pool index to be considered active. Non-active nodes are removed by a garbage collector periodically.

A pool index has three different time-related parameters:

- Requested delay: period at which the nodes are requested to send heartbeats to the pool index server.
- Deprecation delay: delay after which a node is considered non-active if no heartbeat was received.
- Garbage collector cycle duration: duration between two garbage collector cycles.

3.1.1.3 Client

A client can connect to a node and then create connections between nodes further down the line to communicate with the desired destination. To do so, the client communicates with one or more pool index servers to retrieve node lists. It then generates a random path to the destination using the desired number of known nodes.

3.1.2 Connection process with destination

When a client wants to connect anonymously to a distant peer through a torpydo network, it first needs the addresses of some nodes. It can easily obtain them by contacting pool index servers. Once the client has chosen the nodes it wants, it connects to the first node and proceeds with a TPDP handshake (see 3.2) during which a shared key is exchanged. At the end of the handshake, the client requests the node to connect to the next node in the client's node list. From this point, the client communicates with the second node through the first node (see 3.1.3). The client then handshakes with the second node, exchanging a shared key and requesting the second node to connect to the third node. Now, the client communicates with the third node through the first and the second nodes. This path construction process is repeated until the final destination peer of the client is reached.

This process is guaranteeing anonymity to the client since the nodes only know the addresses of their source and destination peer. The only node directly connected to the client only receives data encrypted under several encryption layers. The only node having access to the clear-text data (the last node) is at the end of the chain and hence doesn't have access to the client address. Furthermore, as said previously, thanks to the stream ciphers used, every node has the same behavior, so no node can know for sure if they are connected to the client, to the final destination, or to another node.

3.1.3 Communication process with destination

Once the path is complete between the client and its destination peer, the two entities can communicate with each other. When the client wants to send data to its peer, it first encrypts the data multiple times with each node's shared key from the last key to the first. It then sends the encrypted data to the first node.

When the first node receives the encrypted data segment, it removes the first encryption layer with its shared key and then forwards the data to the second node. The second node receives the data, removes the second layer of encryption, and forwards the data to the third node. This process is repeated all along the path until the last node removes the last layer of encryption and forwards the clear-text data to the client's destination peer.

When the destination peer answers, it sends data to the last node, and the previous process is executed backward: The last node receives the data, adds an encryption layer, and forwards the data to its source node, etc... At the end of the chain, the client removes each encryption layer with the node's shared key from the first key to the last.

If the client, the destination peer, or a node of the path cuts a TCP connection, then all the connections between nodes, client, and destination peer are closed.

3.2 The TorPyDo Protocol (TPDP)

The TorPyDo Protocol (or TPDP) is the protocol we designed for client-node communication. The full specification of the protocol is written in the file `TPDP.md`.

When a client connects to a node, a TPDP session is created. A TPDP session is divided into two phases: handshake, and data forwarding.

During the handshake, the client and the node will exchange a shared key, to encrypt and decrypt data. Then the client communicates with the node its destination hostname and port (in an encrypted way), to which the node tries to connect. If the connection is successful, the session enters the data forwarding phase.

During the data forwarding phase, every data segment received from the source is decrypted and forwarded to the destination. Similarly, every data segment received from the destination is encrypted and forwarded to the source.

3.2.1 Handshake

During the handshake:

1. "Hello" is exchanged between the client and the node.
2. A random private key is generated on the client and the node.
3. The public keys (tied to the private keys) are exchanged.
4. A 256-bit shared key is calculated using the Elliptic Curve Diffie-Hellman key exchange algorithm.
5. The client generates a random nonce for the AES256-CTR cipher and sends it to the node.
6. The client and the node both create an AES256-CTR cipher instance using the shared key and the nonce.

7. The client sends to the node the encrypted hostname of the desired destination and the port.

8. The node tries to connect with the client-requested destination.

9. The node sends two End of Transmission Blocks (ETB) characters to the client to mark the end of the process.

Any difference with these steps will result in a `PROTOCOL_ERROR` followed by an immediate disconnection.

Additionally, a timeout (10s by default) can be set by the node administrator. If the node waits for a client response during the handshake for more than the timeout delay, it will result in a `TIMEOUT_ERROR` followed by an immediate disconnection.

3.2.2 Error handling

Errors can happen during the handshake phase. Here is a list of all the possible errors, with their signification:

- `TIMEOUT_ERROR` - The client took too much time to do a handshake step.
- `PROTOCOL_ERROR` - Node or Client did not respect the protocol.
- `DESTINATION_CONNECTION_ERRROR` - Node was not able to connect to the next node

When an error is thrown by a node, it sends a byte corresponding to the error code to the client and immediately cuts the connection. Hence the client can get the error code by looking at the last byte of the stream. When an error is thrown by a client, it cuts the connection and raises an exception.

3.3 Secured simple challenge-response authentication protocol

The implemented challenge-response authentication protocol is divided into two phases: handshake, and authentication.

During the handshake, a shared key is exchanged, to ensure secure communication of sensitive data. The handshake is similar to the beginning of the previously explained TPDP handshake:

1. "Hello" is exchanged between the client and the server.

2. A random private key is generated on the client and the server.

3. The public keys (tied to the private keys) are exchanged.

4. A 256-bit shared key is calculated using the Elliptic Curve Diffie-Hellman key exchange algorithm.

5. The client generates a random nonce for the AES256-CTR cipher and sends it to the server.

6. The client and the server both create an AES256-CTR cipher instance using the shared key and the nonce.

7. The server sends two End of Transmission Blocks (ETB) characters to the client to mark the end of the process.

During the authentication, the server receives the password of the client, encrypted with the exchanged key. The server then decrypts it, hashes it, and compares this hash with the stored password hash.

If this last step is successful, the server sends secret data to the client, encrypted with the exchanged key to avoid eavesdropping.

4 Challenges

During this project we were confronted with some challenges:

4.1 Developing thread-based code

Initially, we were not aware of the existence of async paradigm in python. So we began our project by writing multi-threaded code to be able to manage multiple connections. This became rapidly overwhelming as we had to manage some kind of semaphore variables to control the execution flow.

In then end, we've switched to asynchronous programming, when we've found, by chance, the standard `asyncio` library.

4.2 Cyptographic choices

When designing the TPDP protocol, we had to make several decisions related to cryptography.

We had to choose the kind of cipher we wanted to use (block or stream), the algorithm (public key or secret key), the mode of operation and if applicable, the key exchange algorithm.

Since these choices are critical, we have spent a lot of time studying the different possibilities, to finally find the best combination of algorithms.

5 Conclusion

During this project, we developed an asynchronous user-friendly Python library, capable of fulfilling production-wise needs for anyone that wants to implement anonymity-based security to their application.

We have designed the communication protocol from scratch, thinking about each step, and fixing some of TOR's known vulnerabilities, to ensure maximum security and performance. We've also added quality-of-life features like pool index servers.

Overall this project was really interesting. We've had the chance of working on a complex real case of networking and cryptography application. The implementation required a lot of reasoning but was, at the end, very rewarding.