Redhwane Ben Salem
Yen Nhi Nguyen
Maxime Pellichero
David Touche

GROUP 06
ELEC-H417
December 2022

ELEC-H417 - Project Assignment :

TOR Network

& Anonymous Challenge-Response Authentication

git : https://github.com/RedhwaneBS/TOR-network

Introduction

Sometimes, anonymity rather than privacy is what is preferred. A user could desire, for instance, to remain anonymous when surfing the internet or to restrict their search engine from building a profile based on their network activity. Since a VPN provider may still observe the user's traffic history, TOR aims to ensure anonymity by letting its users be anonymous without forcing them to trust any centralized providers.

Code structure

Network of relay nodes, TOR is a "onion routing protocol" which employs multiple layers of encryption to mask the source of network requests and allows clients to communicate anonymously. In this project, there is a superclass, *Element*, which inherits 2 subclasses, *Client_TOR* and *Node_TOR*. The TOR nodes are modelized as instances of the *Nodes_TOR* class, and the clients are represented by instances of the *Client_TOR* class. *Element* represents the union of the 2. It consists of the set of attributes and methods that are common to both clients and nodes. This implies that these 2 classes work in much the same way. They are both instantiated in other files called *client_manager* and *node_manager*.

In the *RSA* file, there are all the functions related to the cryptographic algorithms
Next to the code there are two csv files : *contacts.csv* contains the ip address, port and name of the clients (which are read at the beginning of the client_manager file), *private_keys* contains the ip, port and public keys of the nodes (which are generated at the beginning of *node_manager*).

Architecture of the TOR network

The network is composed of TOR nodes to which clients can connect to communicate with each other anonymously.

Entry to the TOR network

When a client wants to use the TOR network, it must know the port and the ip of a node in it. It is assumed that the list of nodes in the network with their ip/port/public key is public and accessible by everyone on the internet.

When connecting to a node, the client receives in return a list of all the nodes in the network. It will be used to build paths for sending onion messages.

When connecting to a node, the client sends its coordinates (ip, port, public key) and receives in return the list of all the nodes on the network. It will be used to build paths for sending onion messages.

Peer-to-Peer Network

TOR nodes periodically send part of the list of clients they own (obtained when connecting a client or a previous share) to each other and to clients they know. The goal is to propagate the public keys of the clients between them. As sharing is quite frequent, this propagation is done quite quickly. However, in a very large network with many clients, it is possible that the key of a client is not known by the one who wants to send him a message. We have not implemented a solution to this problem but we discuss a possible solution in section “Future possible implementation”.

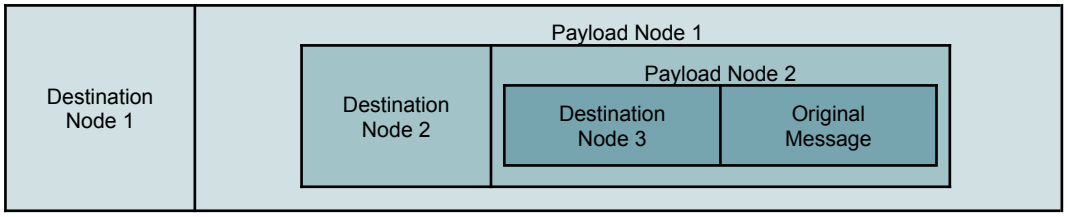
Clients communication

Clients have a contact list, which contains the ip/port/name of registered contacts. They can send messages to each other in the terminal using the name of the contact they have registered with the "contact_name message" structure. They can also send a message with the ip and port of the person with the structure "ip//port message".

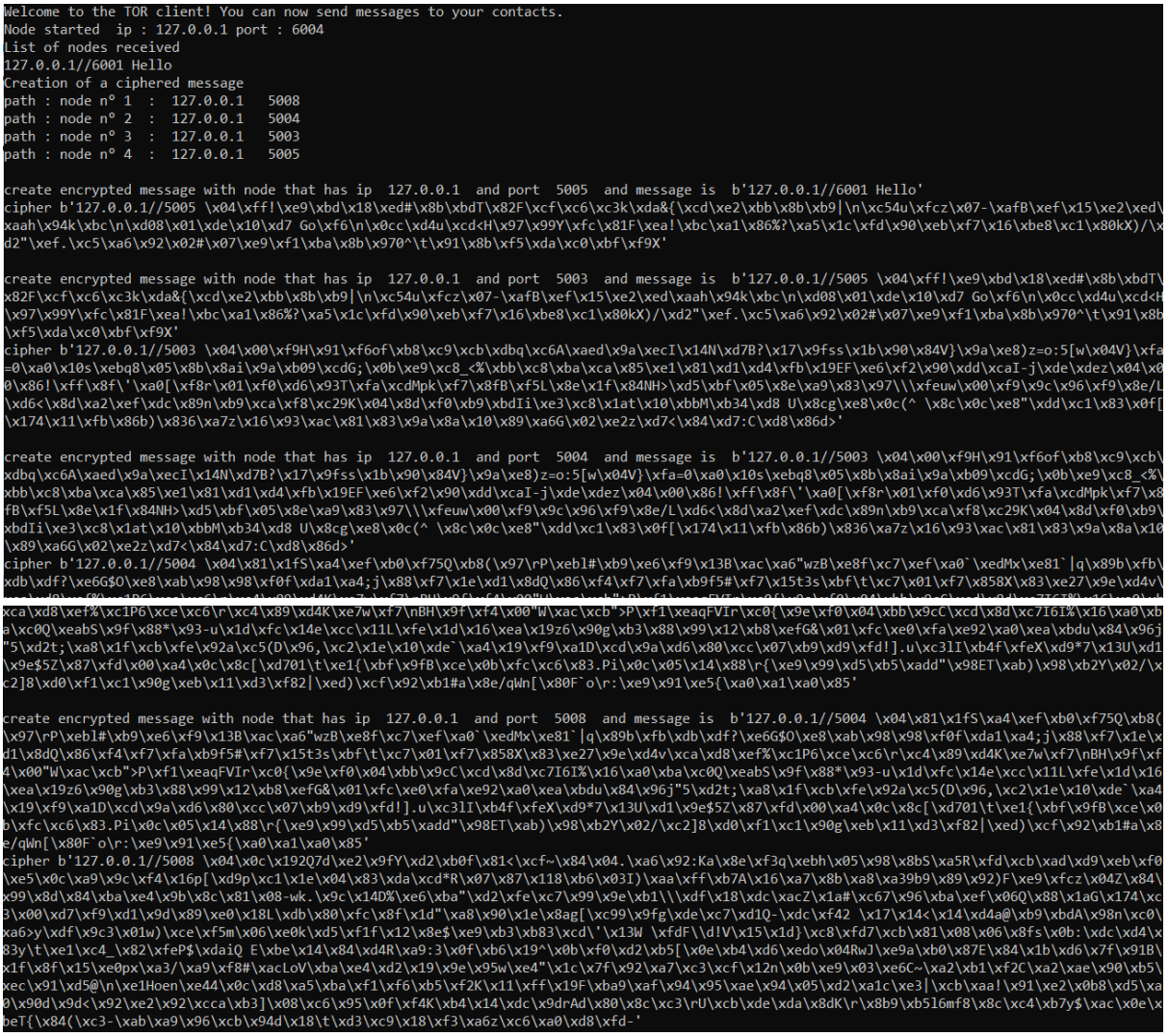
However, for this to work, the client must have received the recipient's public key via propagation.

Cryptography concerning the messages

Each client and each node use *crypt()* to generate its pair of public and private keys. From that, instances of the *Encryptor* class are generated by the client/sender/source for each public key in order to encrypt. Composed by the ip and the port of the next hop, the header is also added during the encryption. The only way to decrypt the cipher is to call *crypt.Decryptor()* that uses the private key, so each node or each client can only decrypt what was encrypted with its own key.



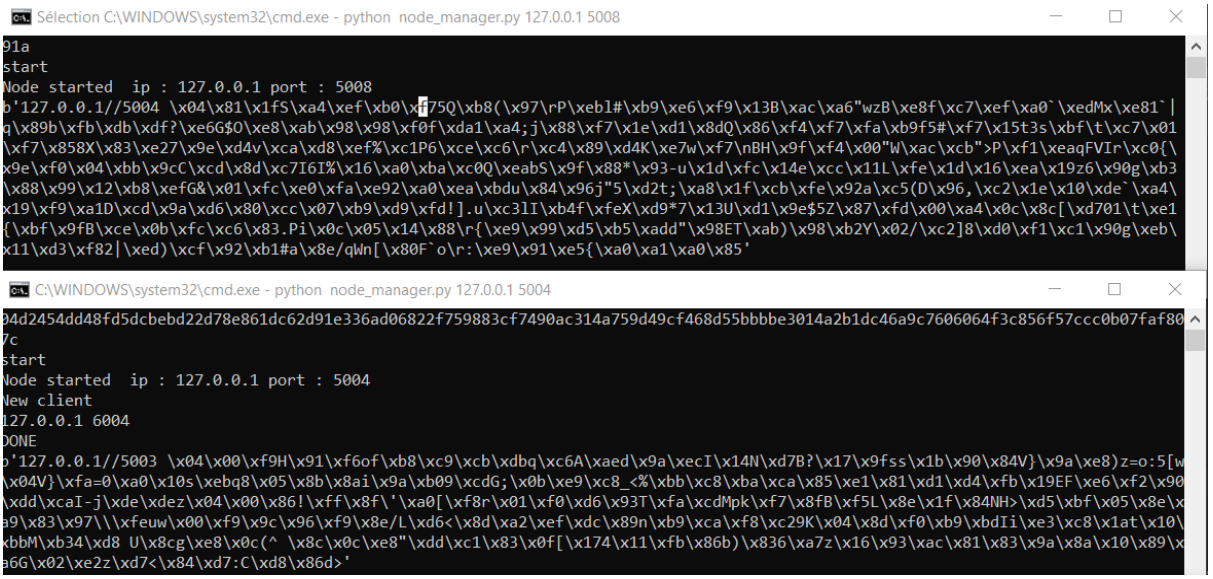
Encryption for a path of 3 nodes



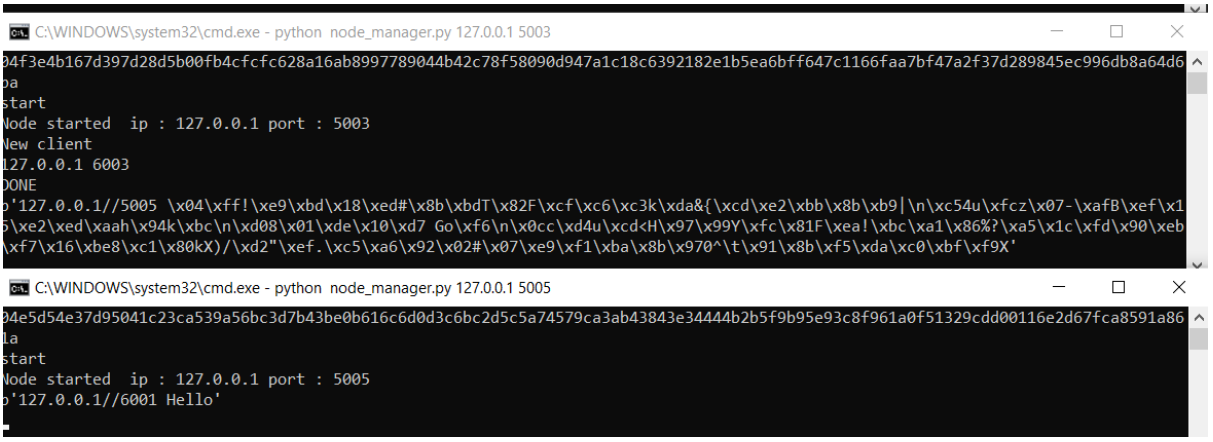
Terminal of the sender : creation of a ciphered message (with additional print() lines in the code)

The processing of a message can be seen with some *print()* lines added to the function `create_message(self, path, message)` in the `Client_TOR(Element).` For each node in the path, the message will be first encrypted and then, the header will be added.

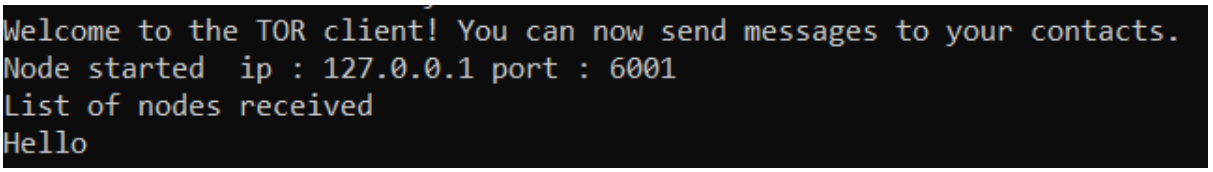
After the sending, the comparison between the ciphers of the encryption iterations and the data that each node sees (after the decryption with its own key) is the same. The receiver is also the only one that can read the original message.



Terminals of the 2 first nodes : reception of the ciphered message



Terminals of the nodes : reception of the ciphered message



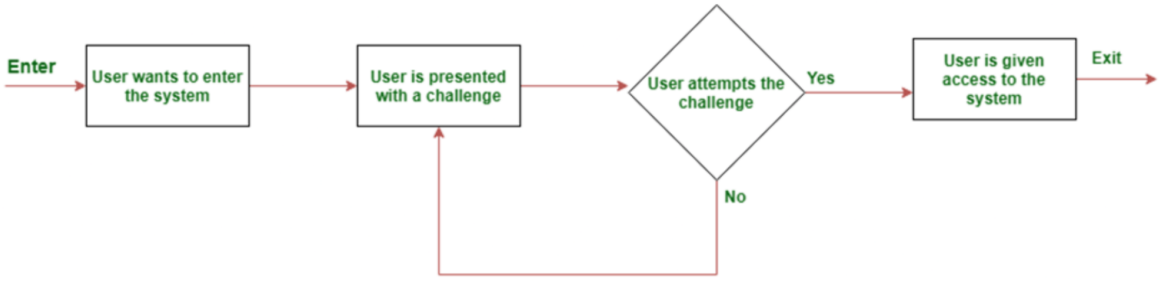
Terminal of the receiver : reception of the ciphered message

Anonymous Challenge-Response Authentication

Clients connected to the TOR network should be able to contact an outside server and start a Challenge-Response-Authentication with them.

The authentication starts as follow :

- The client contact the server and give it their username and password for registration
- Anytime later, the client can contact the server to login by sending it's username
- The server will respond with a "challenge" which is a random token .
- The client must then encrypt this random token with its password (Private Key) and send the encrypted token to the server
- The server will also encrypt the token with the registered password of this user
    - If both encrypted token match, the client is authenticated



All messages are sent through the TOR network for anonymous exchange.
For security purposes, we could also send the hash of the password instead of the password itself for authentication. The exchange would have been the same by replacing the password by its hash but the password would not have been shared directly.

In our case, the server is still not working, we have had some issues with the communication network and the sockets. That being said, the server did work individually with a direct client. The authentication followed the previous points and the server was able to confirm the identity of a client.

The authentication is done using AES encryption for the token's encryption.

Innovation and creativity

The main feature we added was the node_list sharing between the nodes and the client. Instead of simply admitting that every node and client knew the public key of each other, we decided to force them to share the public keys and the nodes they know to each other. This happens every few seconds. We decided not to have a server that distributes the public key because that would not fit with the idea of a TOR network and a peer-to-peer network.

Sharing the public key between nodes allows us to refresh the path possibility of the nodes when they decide to choose a path.

For this purpose, nodes and clients must be able to exchange contact information. To do this, special headers were used to differentiate an information exchange from an onion message.

- 300.0.0.0//0 is the mark of a TOR node list exchange.
- 300.0.0.0//1 is the mark of a clients' list exchange.

Challenges

The modelization of the TOR network is the first issue that arises. At first, it wasn't evident what made TOR nodes and clients different from one another. This results in the subclasses Client_TOR and Node_TOR as well as the superclass Element. Then, in order to accommodate the desired functionality, some functions were overridden, and some others were added.

A second problem encountered is the ‘challenge-based authentication’. The code was first implemented, but then, when merging, a problem arose : the communication is not continued between the server and the client. The solution found is to allocate a new socket of the client to the communication with the server.

The third challenge emerged when we wanted to encrypt the data. The user must have all public keys of the nodes on his path in order to encrypt it. The problem was that it would have been impossible to store the key of all nodes in each node of the TOR network. We decided that every client would have a list containing the public key of each node. The client’s public key on the other hand is distributed as previously mentioned.

Future possible implementation

A TOR network can always be improved to achieve faster or more secure communication.

The first big issue of a TOR network is the speed of requests. The random bouncing between many nodes of the TOR network makes it very difficult to optimize the speed. The goal is to reduce the delays at each node. Congestion control is difficult to implement without losing the privacy of the network however it is possible and it will make the TOR network much faster. The congestion control can reduce the number of packet loss and optimize the bandwidth usage.

A second issue is the randomness of our TOR network. Both public key sharing and routing are random. This can lead to a situation where a packet must go through all the nodes of the network which can be way too slow. Some clients may never be able to contact more than a

few nodes or a packet may never arrive at the destination. One better way of implementing the routing for example is to use an algorithm which takes some randomness into account (because it is a must for a TOR network) but still is efficient and does not lead to any critical paths. The same approach can be taken for public key sharing. Thus, upgrading our pathfinding algorithm can be very useful for our TOR network to avoid problems and to increase efficiency.