Université libre de Bruxelles

École Polytechnique de Bruxelles

Vrije Universiteit Brussel

PROJECT: ELEC-H417

Tor Network

STUDENTS: MERIEAU LUCAS, KIEFFER AXEL AND VAN BEGIN NATHAN

I. Table of contents

II. Introduction

For the P2P connection, we have used the following ressources:
https://github.com/GianisTsol/python-p2p

1. The description of the project

The goal of this project is to design and implement a TOR network that enables anonymous usage of a network. It will be an opportunity to have a practical understanding of different concepts in cryptography and networking studied in the theoretical classes.

2. Goal of the lab

Our project should fulfil the following properties. Peer-to-Peer Network, The nodes in the pool should be connected following a Peer-to-Peer architecture. We must design a protocol that can support an arbitrary large number of peers and listen for new ones which would try to join. We must run these peers locally on our computer for development purposes. Anonymous Connection from a Client A client should be able to send requests to and receive answers from a destination address through the TOR pool. The source address should be anonymous for any observer eaves-dropping the network. At this point, the destination can be anything, from an actual webpage to a simple local HTTP server replying with a Hello world! HTML file.

We also have to implement a server running a challenge-response based authentication protocol. This should allow a client to successfully authenticate to the server using a password, without leaking it. Our client should be able to contact the authentication server through the TOR network to achieve anonymous authentication.

III. Connection initialization

1. Waiting a connection

To establish a connection with a remote host over a network. We must create a function that takes two arguments: host, which is the IP address or hostname of the remote host, and port, which is the port number to which the connection should be made. The default value for the port is 5000.

First, the function needs to check if it is already connected to a node with the given host. Otherwise, it creates a new socket and attempts to connect to the specified host and port. If the connection is successful, the function must send the IP of **A** to **B** and receive the IP of **B** in return.

Next, the function should exchange public keys to establish a secure connection. It does this by sending its own public key to B and receiving the B's sym key encrypted in the public key of A in return. It then uses the private key to decrypt the received public key. The sym key of A is then encrypted using B sym key and sent back to B (this part will be more explained later).

Finally, the function creates a new thread to handle the connection and passes it the socket. If any exceptions are raised during the process, the function prints an error message.

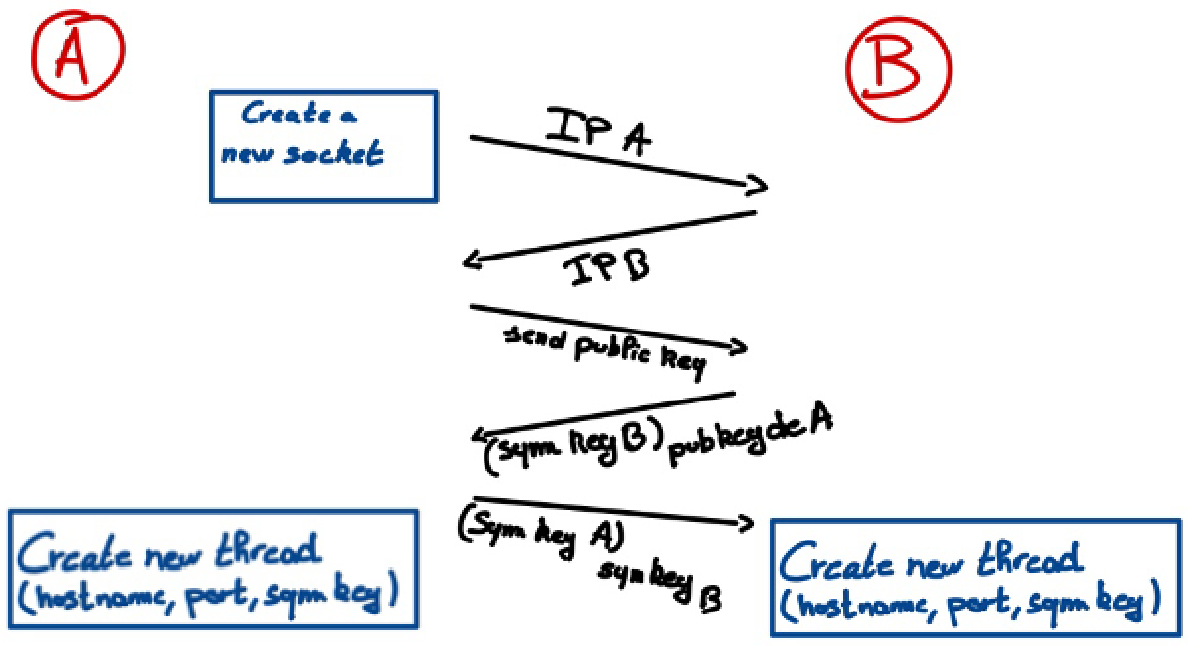


Figure: *Handshake diagram*

If a timeout occurs while waiting for a connection, the client checks if it has any nodes that it has previously tried to connect to and attempts to connect to them again.

When over, the function should stop all the threads handling connections with other nodes, close the socket, and print a message indicating that the node has stopped.

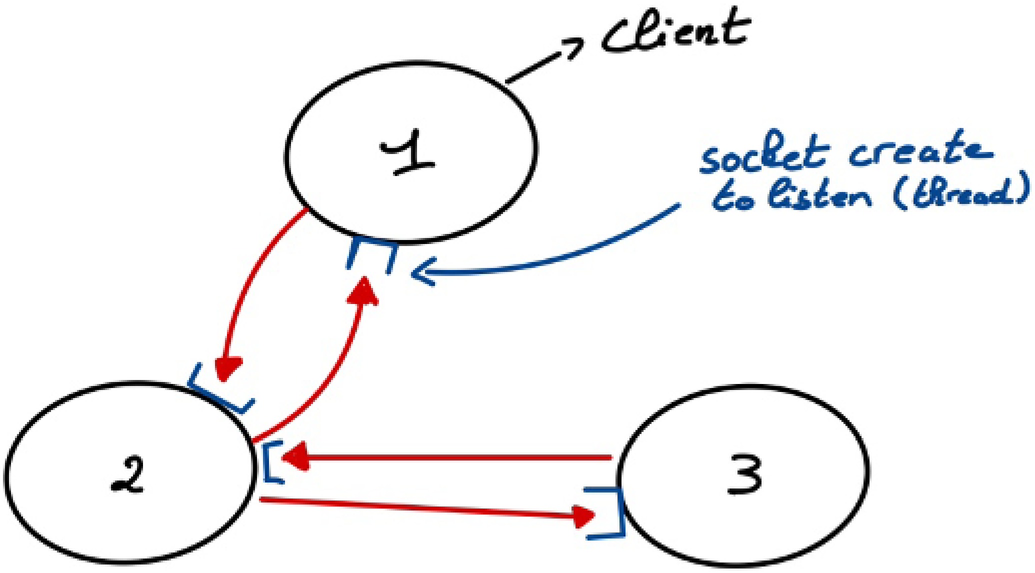


Figure: *Creation of socket diagram*

IV. Tor communication

The **main.py** file is the entry point of the program.

The first thing it does is ask for a number of peers “n” to be introduced by the user.

Then it calls the **start_nodes(n)** function. If n > 10, then we proceed to a safe start, which means that we start and connect the nodes one by one instead of starting them simultaneously, to avoid overloading the program.

The variable called "nodes" is an array of nodes representing our TOR nodes, and each node is an instance of the client class (**client.py**).

When we create a client, we must provide an IP address, a port number, and mention whether or not we want it to operate in verbose mode.

1. Client initialisation :

The client constructor initialises every thing that will be needed later on :

- Creates a private key
- Creates a public key
- Creates a symmetric key
- launches an instance of the pinger class (**pinger.py**), which will go through every node that is connected to the client and try to ping it. The pinger does this every 10 seconds. Each time a client A sends a ping to a client B, the client A includes in the ping request the list of the peers connected to him, allowing B to actualise his list of peers with the new peers from A. Thus in our case, the ping is used to know if the node is alive, but also to transmit RIP-like data about the peers and the network.

Then the client begins to wait for a connection. The client can also connect itself to another client using the **connect_to(...)** method. This method will send the IP address of the sending client, and begin a handshake with the receiving client. If the handshake succeeds, then both clients will each have an active socket (in a thread) listening to the other client, allowing bidirectional communication.

2. Handshake :

The handshake between two clients happens in the following manner :

- A sends its IP address to B.
- B sends his IP address to A.
- A sends its public key to B.
- B encrypts its symmetric key* with the public key of A and send the encrypted symmetric key to A.
- A encrypts its own symmetric key with the symmetric key of B and sends it to B.
- A starts a socket to listen to incoming messages from B.
- B starts a socket to listen to incoming messages from A.

* symmetric key : the same key is used for encryption and decryption

The sockets verifies that incoming packets :

- Are valid
- Then the packet is decrypted with the clients own key
- Then the packet is analysed :
  - If it is a ping packet
  - If it is a normal message, then it print the message
  - If it is a TOR message, then it forwards the message to the next node and wait for an answer if there is one

If a client receives a ping packet, then it goes through the packet looking for peers to which he is not yet connected. If he finds any, then he adds them to his list of peers.

At each new ping, the client can only add 10 new peers to his list of peers, otherwise the client is overloaded.

A client can send two types of messages to one of its peers :

- A cleartext message using the **send_message(...)** method.
- An encrypted message using the **send_tor(...)** method.

The **send_message(...)** method is trivial so we will not expand on it.
However the **send_tor(...)** method is more complex so we will explain it in the next section.

3. Sequence of operations of an encrypted message through our TOR program :

If A wants to send an encrypted message to B through :

- A converts the data to be send form E to B into bytes, written **data**
- A chooses randomly 3 peers (for example C, D, E) among his peers.
- A creates a path from A to B using C, D and E, for example A -> C -> D -> E -> B.
- A places **data** in the datagram DE (source D, destination E) -> **DE(data)** encrypted with key E
- A places **DE(data)** inside CD -> **CD(DE(data))** encrypted with key D
- A places **CD(DE(data))** inside AC -> **AC(CD(DE(data)))** encrypted with key C
- A sends **AC(CD(DE(data)))** to C

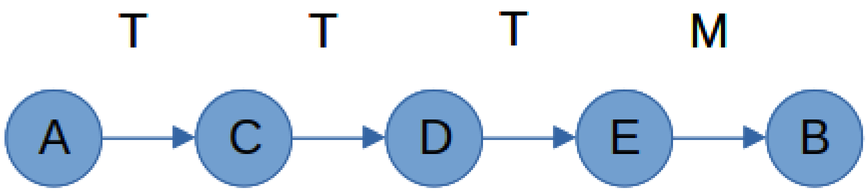
- C decrypts **AC(CD(DE(data)))** with key C -> **CD(DE(data))**
- C sends **CD(DE(data))** to D

- D decrypts **CD(DE(data))** with key D -> **DE(data)**
- D sends **DE(data)** to E

- E decrypts **DE(data)** with key E -> **data**
- E sends **data** to B

- B receives **data**

The nested (onion) encryption of datagrams happens through the use of the tor_builder(...) method



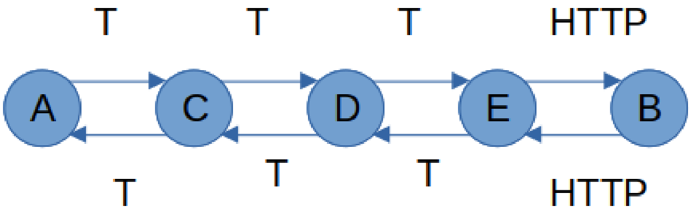
T denotes the use of the **send_tor(...)** method to send the message.
M denotes the use of the **send_message(...)** method to send the message.

When a client receives a TOR packets, then :

- The client can decrypt the package because it has been encrypted with his key
- If the client is not the last relay, then after decryption of the data, he can only see the next destination node. The data is still encrypted. The client then forwards the data to the next destination.
- If the client is the last relay, then he sends the data to the destination using the **send_message(...)** method.

4. Sequence of operations of an HTTP request through our TOR program :

if A wants to send an HTTP request to B through the TOR tunnel C -> D -> E :



- A creates a normal data packet containing the HTTP request with written **HTTPr** with source E and destination B
- A places **HTTPr** in a datagram DE (source D, destination E) written **DE(HTTPr)** , and encrypts it with key E
- A places **DE(HTTPr)** in a datagram CD (source C, destination D) written **CD(DE(HTTPr))** , and encrypts it with key D
- A places **CD(DE(HTTPr))** in a datagram AC written **AC(CD(DE(HTTPr)))** and encrypts it with key C
- A sends **AC(CD(DE(HTTPr)))** to C

- C receives and decrypts **AC(CD(DE(HTTPr)))** with key C and obtains **CD(DE(HTTPr))**
- C sends **CD(DE(HTTPr))** to D
- C blocks itself waiting for a response from D

- D receives and decrypts **CD(DE(HTTPr))** with key D and obtains **DE(HTTPr)**
- D sends **DE(HTTPr)** to E
- D blocks itself waiting for a response from E

- E receives and decrypts **DE(HTTPr)** with key E and obtains **HTTPr**
- E sends **HTTPr** to B and waits for a response

- B receives the **HTTPr** request (normal HTTP request)
- B responds to E with an **HTTPa** answer

- E receives the **HTTPa** response and forwards it to D
- E encrypts **HTTPa** with key E

- D receives the **HTTPa** response and forwards it to C.
- D encrypts **HTTPa** with key D
- D can now unblock himself

- C receives the **HTTPa** response and forwards it to A.
- C encrypts **HTTPa** with key C
- C can now unblock himself

- A receives the **HTTPa** response it was waiting for.
- A can decrypt **HTTPa** using symmetric keys C, D, E

V. Useful functions:

Packet_builder: This function build a packet with a custom packet header

The packet header we have defined includes these fields:

- Type: This field specifies the type of packet.
- Total length: This field specifies the total length of the packet in bits. It is calculated by adding up the lengths of the other fields in the header and the payload.
- Time to leave (TTL): This field specifies the maximum number of hops that the packet is allowed to take before it is discarded.
- Padding: This field ensures that the total length of the packet is a multiple of 8 bits.
- IP_Source: This field specifies the source IP address. It is converted from a string representation of the IP address to a BitArray using the ip_to_BitArray function.
- IP_Destination: This field specifies the destination IP address. It is converted from a string representation of the IP address to a BitArray using the ip_to_BitArray function.

Tor_builder: This function creates a TOR packet by adding layers of encryption and encapsulation to a payload of data.

The function takes these inputs:

- Ip_source: The IP address of the source of the packet.
- Path: A list of 3 tuples, each containing an IP address and a key.
- Data: The payload of the packet.
- Time to leave: The time to leave the packet, which specifies the maximum number of hops that the packet is allowed to take before it is discarded. The default value is 255.

The function first encrypts the payload using the key from the third tuple in the path list. It then builds a packet using the packet_builder function, with the source and destination IP addresses set to the IP addresses in the second and third tuples, respectively. The TTL of this packet is set to 0.

This packet is then encrypted using the key from the second tuple in the path list, and a new packet is built using the packet_builder function, with the source and destination IP addresses set to the IP addresses in the first and second tuples, respectively. The TTL of this packet is set to 1.

Finally, the resulting packet is encrypted using the key from the first tuple in the path list, and a new packet is built using the packet_builder function, with the source and destination IP addresses set to the ip_source and the IP address in the first tuple, respectively. The TTL of this packet is set to 2.

TTL will be used to define if the receiver is the exit point of the TOR network.

Start nodes: This function starts a set of nodes as clients in a network and establishes connections between them.

It takes a single input, n, which specifies the number of nodes to start. If n is greater than 10, the function starts the nodes in a safe manner and waits until all of the nodes are connected before adding another node. If n is less than or equal to 10, the function starts all the nodes and wait for all of the nodes to be connected. Once all of the nodes are started, the function calculates and prints the time it took to start and connect all of the nodes.

Connect to:

It takes as input the IP address and port of the node to connect to, and the node calling the function. The function first checks to see if the calling node is already connected to the node it wants to connect to.

If not, the function creates a new socket and tries to connect to the target node using the specified IP address and port. It then exchanges public key information with the target node and uses that information to exchange an encryption key.Then the function creates a new thread to handle the connection to the target node.

Send Tor: This function sends a message from one node to another in our TOR network.

It takes as input the message to send and the IP address of the destination node. The function scans the list of peers and finds the node with the corresponding IP address (the destination node). It removes this node from the peer list and stores it in a variable called *receiver*.

The function then selects three nodes at random from the peer list and adds them to the path list. Finally, it adds the receiver node to the end of the path list. The function then encodes the message data as bytes and builds a data packet.

This packet is then passed to the "tor_builder" function which builds a TOR packet containing the data packet and the routing information. The function then scans the list of nodes to which the sending node is connected and, when it finds the node with the IP address specified in the first element of the list, it calls the send method.

VI. Challenge

Let's take a client A and a server S. The server wants to know if A and S share the same password without sending the password over the TOR network. To do this, A chooses a diversifier randomly, which is an integer on 16 bits. Then A will concatenate his password with the diversifier and compute the hash of the result.

$$\text{hash}_A = \text{abs}(\text{hash}(\text{password}_A \mid\mid \text{diversifier}))$$

Where || denotes the concatenation operation.
Then A will send the computed hash $_A$ and the diversifier to S.

S then computes the hash of his password concatenated with the diversifier.

$$\text{hash}_S = \text{abs}(\text{hash}(\text{password}_S \mid\mid \text{diversifier}))$$

S then compares hash$_A$ with hash$_S$:

If hash$_A$ = hash$_S$ then the passwords of A and S can be considered equal with very high probability, otherwise the passwords are different.

VI. Conclusion

In this project, we have learned how to create a TOR network.

It was interesting to apply the knowledge that we have acquired in class to a “real life” example.

We have encountered some difficulties during the project regarding the realisation of a good communication P2P between the hosts but also to make the TOR requests come back in the other direction.

TOR increased online privacy by hiding the user's IP address and encrypting data sent over the internet. The TOR project uses a network of relay nodes called "onion layers" to hide the origin of the data and prevent tracking.

However, it is important to note that TOR does not guarantee absolute privacy and some users may be identified through advanced tracking techniques (If the 3 nodes are compromised by the same entity).

In summary, TOR is a useful tool for protecting online privacy, but it should not be considered a foolproof solution for completely protecting a user's identity on the internet.