Communication Networks : Protocols and Architecture

Tor Network

Students: *AGYEMANG Serge, GURLUI Octavian, LADNER Navid and SALIM Anthony*
Professor: *DRICOT Jean-Michel*
Teaching Assistants : *DAUBRY Wilson, VERSTRAETEN Denis*

December 23, 2022

Contents

1 Introduction

This project aims to showcase the design and implementation of a TOR network. The task at hand was to implement a peer to peer network, on top of which a layer of encryption on each of its constituent nodes was added to allow for anonymous usage.

The first step to this approach is to initiate a peer to peer network consisting of four nodes, out of which one of the is the *genesis node*, which acts as a tracker of all nodes IP addresses and ports. Within this network, nodes can either broadcast their message to every other node, or send a to a specific one (unicasting).

For two ends, from now on called *clients*, to be able to communicate anonymously through the network, a TOR protocol has been added. In short, each node computes its own key. The keys are negociated with the clients such that the latter ones will have all the keys. Whenever one of the clients wants to send a message (consisting of a payload and the address of the destination), it randomly picks three of the nodes in the network to form a path to the other client. Knowing their three keys, the client starts a process that adds layers of encryption over each node (as shown in fig.1): the payload and the destination address get encrypted with the key corresponding to the last node in the path. To this encrypted message, the address of the last node is added, and the result is encrypted with the 2^{nd} node's key. Lastly, to this message the address of the 1^{st} node is attached, and it all gets encrypted using the first node's key.

For the end client to send back a message, the process above runs backwards (fig. 1), each node using its own key to decrypt the message received. After decryption, the node gets revealed the address to which it should send the payload next, but since it does not have the other keys, can't decrypt the message any further. The purpose of a TOR protocol is that the nodes within the network can only communicate with the adjacent ones, and whatever happens after that is completely hidden from them.

The implementation of the network was done locally using python, different ports of the localhost being attribuited to the nodes. In addition, the AES encryption algorithm was also implemented in python using a dedicated library.

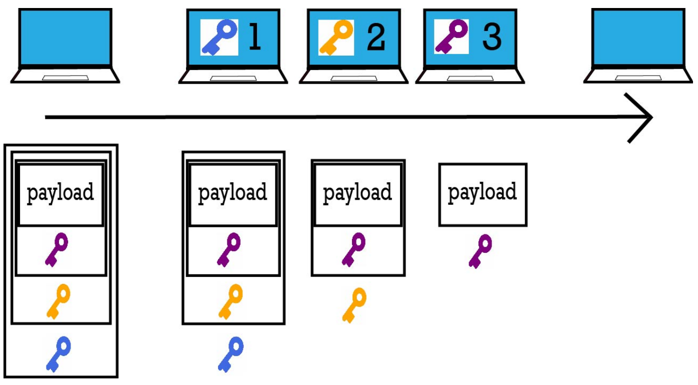


Figure 1: TOR network encryption/decryption system.

2 Peer To Peer Network

The peer to peer network is a type of network were all the nodes are connected and can send direct message to each other. If a client wants to join the network, it needs first to connect to a *tracker*. The *tracker* will be able to send the IP addresses of the nodes already connected to the network.

2.1 Development

The structure of the code for the network is highly inspired by a Github folder [1]. In this code, the *GenesisNode.py* act as the tracker and every instantiations of the class Node should first send a message to the *GenesisNode.py* by the python consol in order get the ports of the other nodes. The reason the only thing the node needs is the port is because the network is run within the computer and therefore, the IP address is the one of the local host. In the code, the IP address **127.0.0.1** is used.

However, the code has to be adapted. Indeed, the nodes and client (which, here, is a type of node) can only communicate through broadcasting, furthermore, the messages send from one node (or client) to the network can not be saved and therefore manipulated (for instance, encrypted or decrypted).

In order to palliate to the first issue, a function *unicastTOR()* has been defined. This function allows a node to send direct messages to another node by putting in arguments the message to send and the port of the destination in the local network.

```python
def unicastTOR(self,data,port,isJson=False):
    ip = '127.0.0.1'

    conn = self.create_connection(ip, port)


    if isJson==False:
        msg = self.short_json_msg("#UNIDCAST",data)
    else:
        msg=data

    try:
        index = self.message_logs.index(msg["id"])

    except:
        index=-1
        self.message_logs.append(msg["id"])

        try:
            self.send(conn,msg)
        except:
            print_colored("MESSAGE COULDN'T SEND, RECIEVER MAY BE
                DISCONNECTED ","red")
```

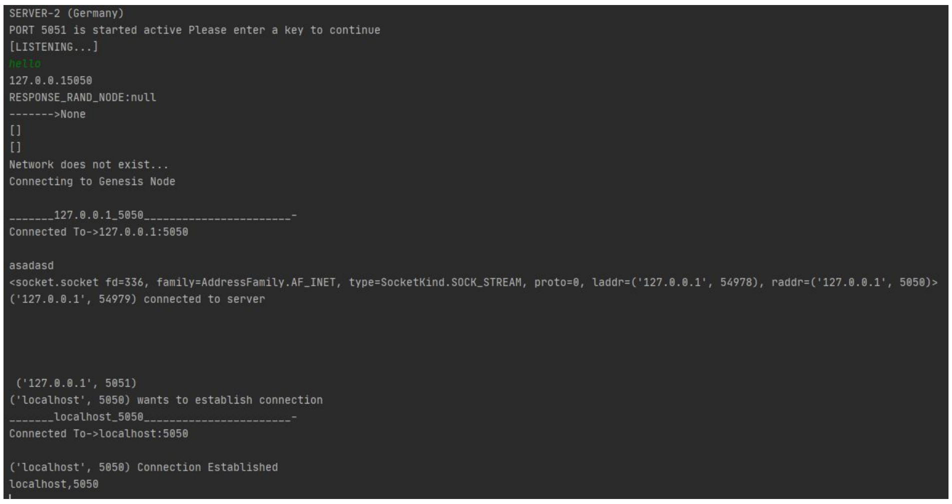
The second issue can be solved by adding an attribute to the class *Node* : *MSGTOSAVE = None*. Then in the file *Network.py*, the following command must be added in the broadcast and ”else” part in the function *handle* : $self.MSGTOSAVE = f"msg"$. This allows a node to save a message received from any other nodes, whether it is on broadcast or unicast mode.

```
    if("#BROADCAST" == msg["title"]):

            self.broadcast(msg,True)

            print(f"{msg}")
            self.MSGTOSAVE = f"{msg}"

            if(msg['message']=="#JOINED_IN_NETWORK"):

                msg_sender_ip =msg['sender_ip']
                msg_sender_port =msg['sender_port']
                if(msg_sender_ip == "" or msg_sender_ip ==None or
                    len(msg_sender_ip) ==0):
                    msg_sender_ip = conn.getpeername()
                    msg_sender_ip= msg_sender_ip[0]

                print(conn.getpeername())


                print_colored(f"{msg['sender_ip']}:{msg['sender_port']}
                    has joined to network","green")


                self.nodes_in_network.append({"ip_addr":msg_sender_ip,
                    "port":msg['sender_port']})
                print("NODES IN NETWORK ")
                print(self.nodes_in_network)
      else:

          print(msg["message"])
          self.MSGTOSAVE = msg
```

2.2 Test and Results

2.3 Creation and connection to the network

In this section, the peer to peer network is tested with 3 nodes (node1, node2 and node3) and a client. As said earlier, the first step for the previous four is to communicate to the tracker (here, GenesisNode) to tell him that they want to be in the network as shown in figure 2. In this case, node1 is the first node to connect to the network. When a second node is connecting, it receives the list of nodes already in the network as shown in figure3.

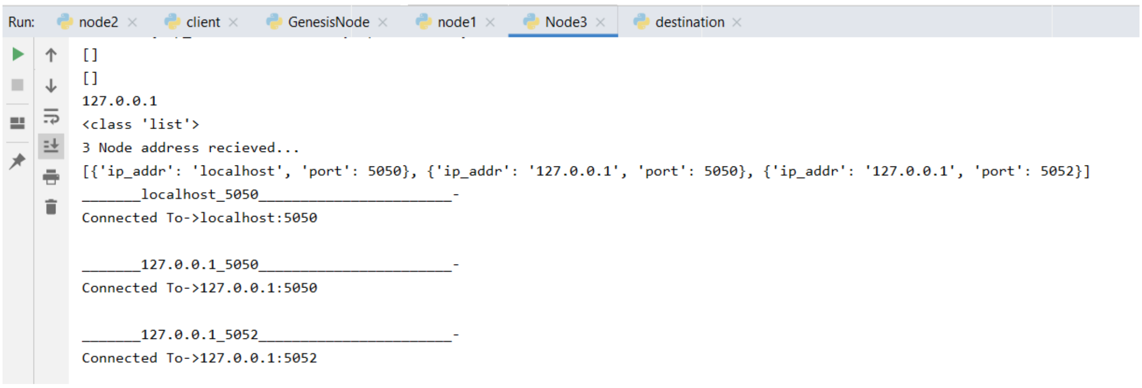


```
SERVER-2 (Germany)
PORT 5051 is started active Please enter a key to continue
[LISTENING...]
hello
127.0.0.15050
RESPONSE_RAND_NODE:null
------->None
[]
[]
Network does not exist...
Connecting to Genesis Node

_______127.0.0.1_5050______________________-
Connected To->127.0.0.1:5050

asadasd
<socket.socket fd=336, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 54978), raddr=('127.0.0.1', 5050)>
('127.0.0.1', 54979) connected to server




 ('127.0.0.1', 5051)
('localhost', 5050) wants to establish connection
_______localhost_5050______________________-
Connected To->localhost:5050

('localhost', 5050) Connection Established
localhost,5050
```

Figure 2: Node 1 connecting to the peer to peer network by communicating with Genesis.Node.

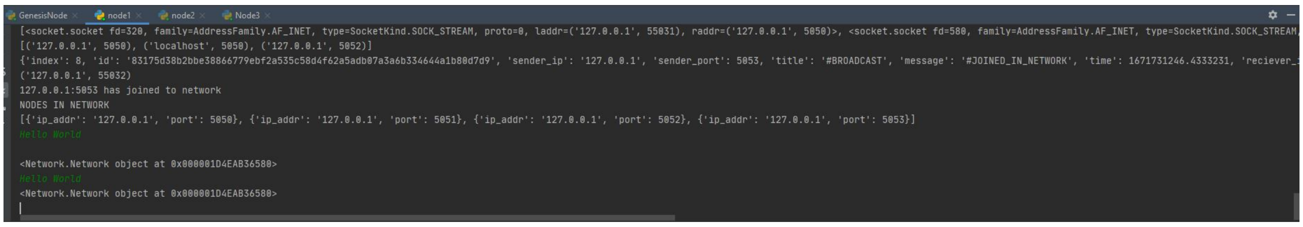
2.3.1 Broadcasting

At this moment of the test, there is no client. To test the broadcasting of the nodes, a message from node1 is sent (figure 4). As shown in figure 5 and 6, the message is well received in node2 and node3. The same operation can be done from either node2 or node3. Hence, the broadcasting function works.



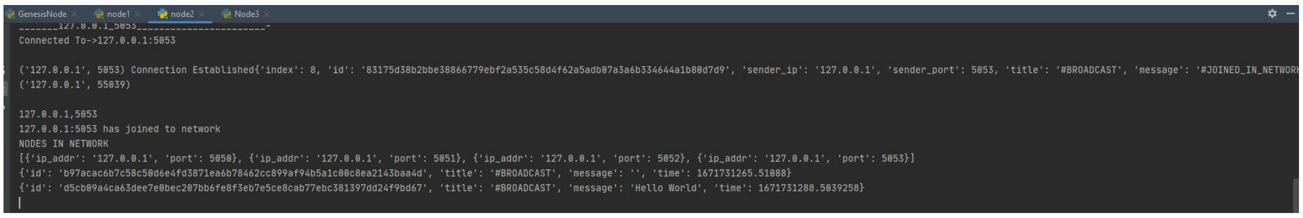
```
[]
[]
127.0.0.1
<class 'list'>
3 Node address recieved...
[{'ip_addr': 'localhost', 'port': 5050}, {'ip_addr': '127.0.0.1', 'port': 5050}, {'ip_addr': '127.0.0.1', 'port': 5052}]
_______localhost_5050______________________-
Connected To->localhost:5050

_______127.0.0.1_5050______________________-
Connected To->127.0.0.1:5050

_______127.0.0.1_5052______________________-
Connected To->127.0.0.1:5052
```

Figure 3: Node 3 is joining the network after node1.



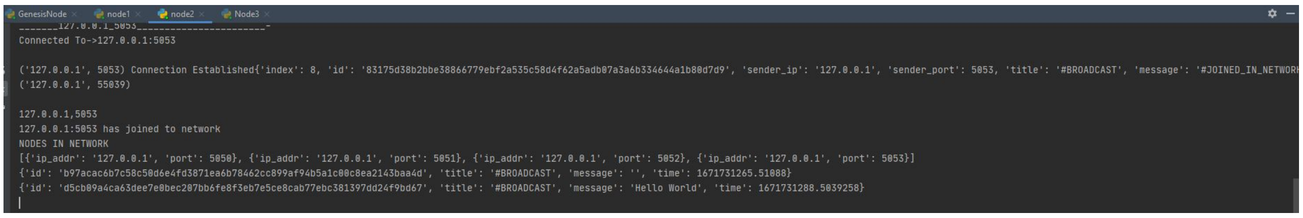
```
[<socket.socket fd=320, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 55031), raddr=('127.0.0.1', 5050)>, <socket.socket fd=580, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM,
[('127.0.0.1', 5050), ('localhost', 5050), ('127.0.0.1', 5052)]
{'index': 8, 'id': '83175d38b2bbe38866779ebf2a535c58d4f62a5adb07a3a6b334644a1b80d7d9', 'sender_ip': '127.0.0.1', 'sender_port': 5053, 'title': '#BROADCAST', 'message': '#JOINED_IN_NETWORK', 'time': 1671731246.4333231, 'reciever_
('127.0.0.1', 55032)
127.0.0.1:5053 has joined to network
NODES IN NETWORK
[{'ip_addr': '127.0.0.1', 'port': 5050}, {'ip_addr': '127.0.0.1', 'port': 5051}, {'ip_addr': '127.0.0.1', 'port': 5052}, {'ip_addr': '127.0.0.1', 'port': 5053}]
Hello World

<Network.Network object at 0x000001D4EAB36580>
Hello World
<Network.Network object at 0x000001D4EAB36580>
```

Figure 4: Node 1 is sending "Hello World" to all the nodes



```
_______127.0.0.1_5053______________________-
Connected To->127.0.0.1:5053

('127.0.0.1', 5053) Connection Established{'index': 8, 'id': '83175d38b2bbe38866779ebf2a535c58d4f62a5adb07a3a6b334644a1b80d7d9', 'sender_ip': '127.0.0.1', 'sender_port': 5053, 'title': '#BROADCAST', 'message': '#JOINED_IN_NETWORK
('127.0.0.1', 55039)

127.0.0.1,5053
127.0.0.1:5053 has joined to network
NODES IN NETWORK
[{'ip_addr': '127.0.0.1', 'port': 5050}, {'ip_addr': '127.0.0.1', 'port': 5051}, {'ip_addr': '127.0.0.1', 'port': 5052}, {'ip_addr': '127.0.0.1', 'port': 5053}]
{'id': 'b97acac6b7c58c50d6e4fd3871ea6b78462cc899af94b5a1c00c8ea2143baa4d', 'title': '#BROADCAST', 'message': '', 'time': 1671731265.51088}
{'id': 'd5cb09a4ca63dee7e0bec207bb6fe8f3eb7e5ce8cab77ebc381397dd24f9bd67', 'title': '#BROADCAST', 'message': 'Hello World', 'time': 1671731288.5039258}
```

Figure 5: Node2 receives the message "Hello world" from node 1.

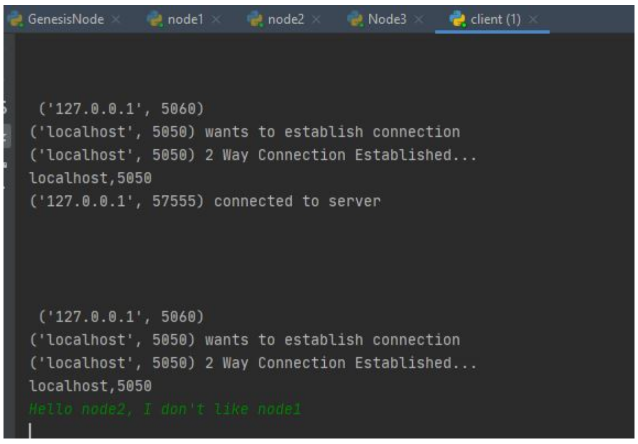


```
_______127.0.0.1_5053______________________-
Connected To->127.0.0.1:5053

('127.0.0.1', 5053) Connection Established{'index': 8, 'id': '83175d38b2bbe38866779ebf2a535c58d4f62a5adb07a3a6b334644a1b80d7d9', 'sender_ip': '127.0.0.1', 'sender_port': 5053, 'title': '#BROADCAST', 'message': '#JOINED_IN_NETWORK
('127.0.0.1', 55039)

127.0.0.1,5053
127.0.0.1:5053 has joined to network
NODES IN NETWORK
[{'ip_addr': '127.0.0.1', 'port': 5050}, {'ip_addr': '127.0.0.1', 'port': 5051}, {'ip_addr': '127.0.0.1', 'port': 5052}, {'ip_addr': '127.0.0.1', 'port': 5053}]
{'id': 'b97acac6b7c58c50d6e4fd3871ea6b78462cc899af94b5a1c00c8ea2143baa4d', 'title': '#BROADCAST', 'message': '', 'time': 1671731265.51088}
{'id': 'd5cb09a4ca63dee7e0bec207bb6fe8f3eb7e5ce8cab77ebc381397dd24f9bd67', 'title': '#BROADCAST', 'message': 'Hello World', 'time': 1671731288.5039258}
```

Figure 6: Node3 receives the message "Hello world" from node 1.

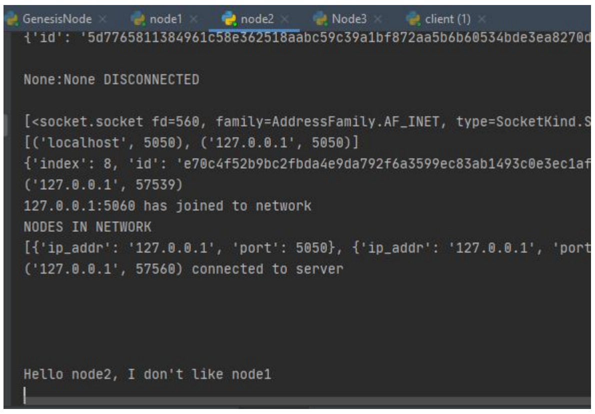
2.3.2 Unicasting

Now a client is connected to the network just like the other nodes. However, this time the client wants to send a message only to one node. In this experiment, the client wants to send a message to node2 as shown in figure 8. However, as shown in figure 9 and 10, node1 and node3 did not receive the message from the client which means that the unicasting works.



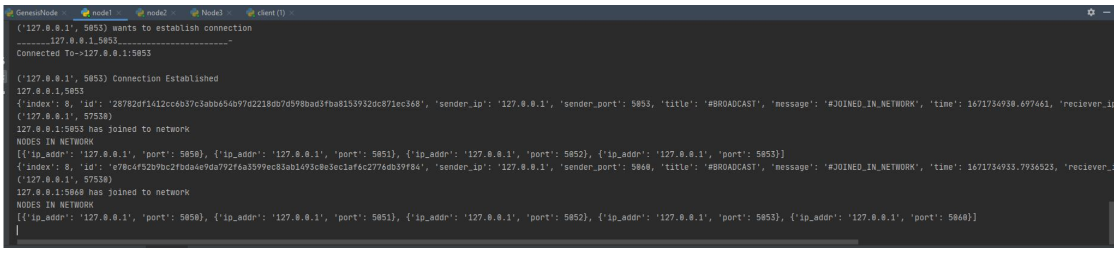
```
 ('127.0.0.1', 5060)
('localhost', 5050) wants to establish connection
('localhost', 5050) 2 Way Connection Established...
localhost,5050
('127.0.0.1', 57555) connected to server



 ('127.0.0.1', 5060)
('localhost', 5050) wants to establish connection
('localhost', 5050) 2 Way Connection Established...
localhost,5050
Hello node2, I don't like node1
```

Figure 7: The client sends a message to node2.



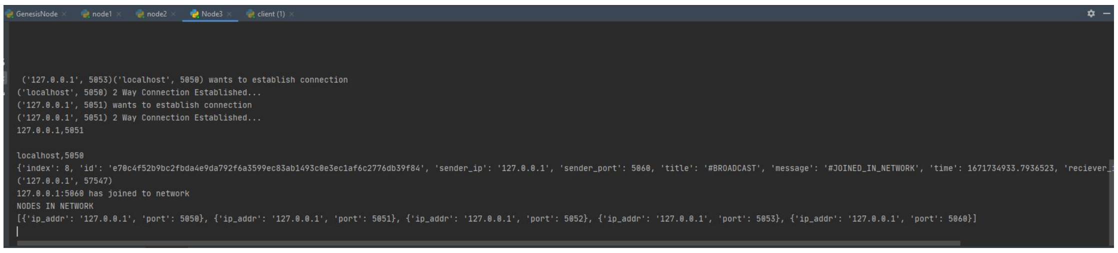
```
{'id': '5d7765811384961c58e362518aabc59c39a1bf872aa5b6b60534bde3ea8270d

None:None DISCONNECTED

[<socket.socket fd=560, family=AddressFamily.AF_INET, type=SocketKind.S
[('localhost', 5050), ('127.0.0.1', 5050)]
{'index': 8, 'id': 'e70c4f52b9bc2fbda4e9da792f6a3599ec83ab1493c0e3ec1af
('127.0.0.1', 57539)
127.0.0.1:5060 has joined to network
NODES IN NETWORK
[{'ip_addr': '127.0.0.1', 'port': 5050}, {'ip_addr': '127.0.0.1', 'port
('127.0.0.1', 57560) connected to server



Hello node2, I don't like node1
```

Figure 8: Node2 receives the private message from client.



```
('127.0.0.1', 5053) wants to establish connection
_______127.0.0.1_5053_______________________-
Connected To->127.0.0.1:5053

('127.0.0.1', 5053) Connection Established
127.0.0.1,5053
{'index': 8, 'id': '28782df1412cc6b37c3abb654b97d2218db7d598bad3fba8153932dc871ec368', 'sender_ip': '127.0.0.1', 'sender_port': 5053, 'title': '#BROADCAST', 'message': '#JOINED_IN_NETWORK', 'time': 1671734930.697461, 'reciever_ip
('127.0.0.1', 57530)
127.0.0.1:5053 has joined to network
NODES IN NETWORK
[{'ip_addr': '127.0.0.1', 'port': 5050}, {'ip_addr': '127.0.0.1', 'port': 5051}, {'ip_addr': '127.0.0.1', 'port': 5052}, {'ip_addr': '127.0.0.1', 'port': 5053}]
{'index': 8, 'id': 'e70c4f52b9bc2fbda4e9da792f6a3599ec83ab1493c0e3ec1af6c2776db39f84', 'sender_ip': '127.0.0.1', 'sender_port': 5060, 'title': '#BROADCAST', 'message': '#JOINED_IN_NETWORK', 'time': 1671734933.7936523, 'reciever_
('127.0.0.1', 57530)
127.0.0.1:5060 has joined to network
NODES IN NETWORK
[{'ip_addr': '127.0.0.1', 'port': 5050}, {'ip_addr': '127.0.0.1', 'port': 5051}, {'ip_addr': '127.0.0.1', 'port': 5052}, {'ip_addr': '127.0.0.1', 'port': 5053}, {'ip_addr': '127.0.0.1', 'port': 5060}]
|
```

Figure 9: Node 1 did not receive the message the client sent to node2.



```
 ('127.0.0.1', 5053)('localhost', 5050) wants to establish connection
('localhost', 5050) 2 Way Connection Established...
('127.0.0.1', 5051) wants to establish connection
('127.0.0.1', 5051) 2 Way Connection Established...
127.0.0.1,5051

localhost,5050
{'index': 8, 'id': 'e70c4f52b9bc2fbda4e9da792f6a3599ec83ab1493c0e3ec1af6c2776db39f84', 'sender_ip': '127.0.0.1', 'sender_port': 5060, 'title': '#BROADCAST', 'message': '#JOINED_IN_NETWORK', 'time': 1671734933.7936523, 'reciever_
('127.0.0.1', 57547)
127.0.0.1:5060 has joined to network
NODES IN NETWORK
[{'ip_addr': '127.0.0.1', 'port': 5050}, {'ip_addr': '127.0.0.1', 'port': 5051}, {'ip_addr': '127.0.0.1', 'port': 5052}, {'ip_addr': '127.0.0.1', 'port': 5053}, {'ip_addr': '127.0.0.1', 'port': 5060}]
|
```

Figure 10: Node 1 did not receive the message the client sent to node2.

2.3.3 Conclusion

The peer to peer network developed is operational and can either broadcast to all nodes or unicast to one particular node. It is scalable to any number of nodes needed, however, in order to do so, the nodes have to be created (by creating a file and adding a specific port number intrinsic to the node) and connected manually to the tracker (by sending manually a message, for instance ”Hello there”).

3 Encryption and Decryption in the TOR Network

In this section, the encryption of the messages is developed. As explained in the introduction, in the TOR network, the messages are encrypted by the client with the keys of pre-selected nodes. The nodes will then decrypt the message layer per layer in order to get the next node. Indeed, each node knows the previous and the next step but not the one before or after. In order to do so, first a way to encrypt the messages is selected. Then cryption of a message and then its decryption are tested. After developing this simple case, the encryption is applied on the peer-to-peer network seen earlier (section 2).

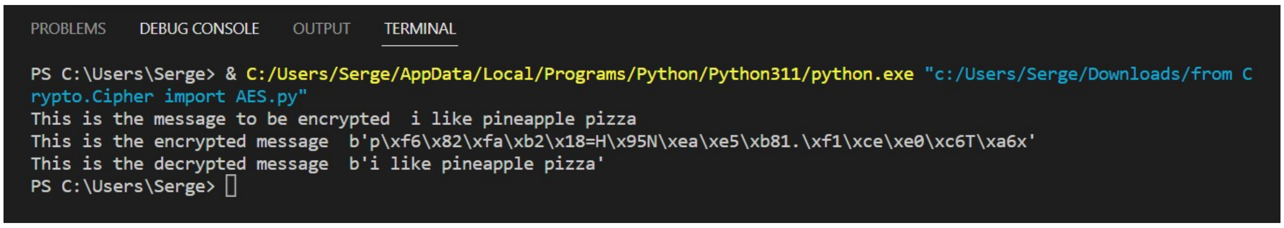
3.1 Basic Encryption and Decryption

First of all, the library used to encrypted the messages is *pycryptodome* [2]. This library uses AES Cipher encryption method. The 2 functions needed are *encrypt_and_digest()* to encrypt the message and *decrypt_and_verify()* to decrypt it. In order to execute those functions, a key of 16 bits is needed. Then, a *Nonce* and a *Tag* are generated and stored in file. Those 2 variables are needed for the decryption of the message.

The encryption of a message from one end of a network can be as follows :

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

destination_port_number=""


encrypted_message3= "i like pineapple pizza"+destination_port_number
print("This is the message to be encrypted ",encrypted_message3)

key3 = b'\xce0\xdd"\xef\x1fV.\x1c\x835\xbcD\x87\xf5\xad'
# text file for node 3

cipher3 = AES.new(key3, AES.MODE_EAX)
ciphertext3, tag3 =
   cipher3.encrypt_and_digest(encrypted_message3.encode('ASCII'))
f = open("tag_node_3.txt", "wb")
f.write(tag3)
f.close()
print("This is the encrypted message ",ciphertext3)

f = open("tag_node_3.txt", "wb")
writing=tag3
f.write(writing)
f.close()


f= open("tag_node_3.txt", "rb")
tag3_read=f.read();
```

```
cipher3.nonce=' '.join(map(str,list(cipher3.nonce)))

cipher3.nonce=bytes(map(int,cipher3.nonce.split(' ')))

cipher3 = AES.new(key3, AES.MODE_EAX, cipher3.nonce)
decrypt1=cipher3.decrypt_and_verify(ciphertext3, tag3_read)
print("This is the decrypted message ",decrypt1)
```



```
PS C:\Users\Serge> & C:/Users/Serge/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/Serge/Downloads/from C
rypto.Cipher import AES.py"
This is the message to be encrypted  i like pineapple pizza
This is the encrypted message  b'p\xf6\x82\xfa\xb2\x18=H\x95N\xea\xe5\xb81.\xf1\xce\xe0\xc6T\xa6x'
This is the decrypted message  b'i like pineapple pizza'
PS C:\Users\Serge> 
```

Figure 11: Original, encrypted and decrypted message

It is important to note that the *Nonce* and *Tag* stored in the files are written in binary in order to be sure that the other end can read it properly. The figure 11 displays what the code above does : encrypt the message and then decrypt it from another end of the network.

3.2 Encryption and Decryption Applied in the TOR Network

3.2.1 Encryption

To implement the encryption seen above in the TOR network, some assumptions were made. The most import one was that the *Tag* and the *Nonce* of each node is saved in a binary file accessible by every nodes and client. However, the key is saved in the nodes that created it and can not be accessed by everyone therefore the security is not compromised.

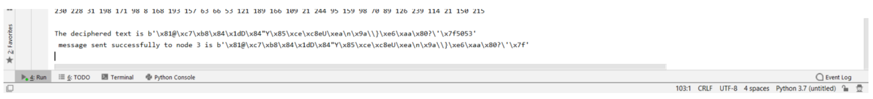
At first the *Client* will need to do the encryption in the following order:

1. Add the port number of the *destination* to the message he want to send.
2. Encrypt using *key3* of the *end port* .
3. Add the port number of the *end port* to the encrypted message.
4. Encrypt again using *key2* of the *middle node* .
5. Add the port number of the *middle port* to the encrypted message.
6. Encrypt for the third time *key1* of the *end node* .

After encrypting the message using all three keys, the client will send the result to the *entry port* using the *unicastTOR(message,port number)* function but before sending, the message should be converted from bytes to string because the *message* parameter of the function *unicastTOR()* has the type string.

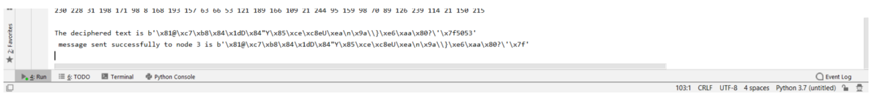
3.2.2 Decryption

Once the message is received, the decryption using *key1* will start inside the *entry port*. To decrypt the message, as mentioned in Section 3.1, 4 parameter are needed, *key, cipher.nonce, tag* and the *encrypted message*. Both *tag* and *cipher.nonce* are retrieved from the binary file and each *key* is written inside its own node port so it won't be accessible to all the node (each node can access only his key for security reasons).
The result will be an encrypted message which has at the end the *port number* of the next node as seen in figure 12 and 13.



```
230 228 31 198 171 98 8 168 193 157 63 66 53 121 189 166 109 21 244 95 159 98 70 89 126 239 114 21 150 215

The deciphered text is b'\x81@\xc7\xb8\x84\x1dD\x84"Y\x85\xce\xc8eU\xea\n\x9a\\}\xe6\xaa\x80?\'\x7f5053'
 message sent successfully to node 3 is b'\x81@\xc7\xb8\x84\x1dD\x84"Y\x85\xce\xc8eU\xea\n\x9a\\}\xe6\xaa\x80?\'\x7f'
```

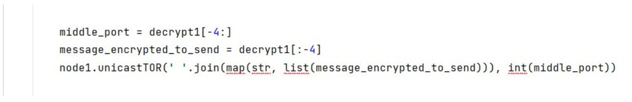
Figure 12: Decrypting the message using *key1* in the *start node*.



```
230 228 31 198 171 98 8 168 193 157 63 66 53 121 189 166 109 21 244 95 159 98 70 89 126 239 114 21 150 215

The deciphered text is b'\x81@\xc7\xb8\x84\x1dD\x84"Y\x85\xce\xc8eU\xea\n\x9a\\}\xe6\xaa\x80?\'\x7f5053'
 message sent successfully to node 3 is b'\x81@\xc7\xb8\x84\x1dD\x84"Y\x85\xce\xc8eU\xea\n\x9a\\}\xe6\xaa\x80?\'\x7f'
```

Figure 13: Decrypting the message using *key2* in the *middle node*.

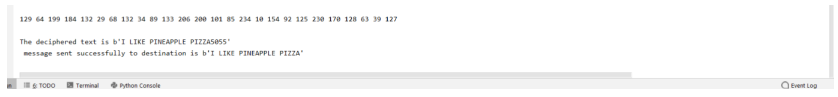
The *port number* and the *encrypted text* will be separated and they will be used as parameters for the next *unicastTOR(message,port number)* function as shown in figure 14.



```
middle_port = decrypt1[-4:]
message_encrypted_to_send = decrypt1[:-4]
node1.unicastTOR(' '.join(map(str, list(message_encrypted_to_send))), int(middle_port))
```

Figure 14: Separating the next port number from the encrypted text.

Once the encrypted message arrivese to the *end port* and decrypted ,the result will be the clear message sent from the client at the beginning, with the port number for the destination network. In this way, each node only knew the next port number and the *port end* was the only one who knew the *destination network* as shown in figure 15.



```
129 64 199 184 132 29 68 132 34 89 133 206 200 101 85 234 10 154 92 125 230 170 128 63 39 127

The deciphered text is b'I LIKE PINEAPPLE PIZZA5055'
 message sent successfully to destination is b'I LIKE PINEAPPLE PIZZA'
```

Figure 15: Node 1 did not receive the message the client sent to node2.

4 Challenge Response

The challenge response happens when a client wants to login to a server through a TOR network. To verify that this is the client that is trying to connect and not any one else, the server sends a *challenge* to the client. The client will be able to answer because he is the only one with the password.

In order to implement the protocol, it was assumed that the password was already known by the client and the server. The following protocol is illustrated in the sketch of code under :

For the server :

1. The server has to create a challenge.
2. The server joins the network to send the challenge to client.
3. The server computes the combination of the challenge and the password with the *hash()* function.
4. The server waits for the answer from the client and will check if the answer is the same as the one the server computed.

For the client :

1. The client waits for the challenge from the server.
2. The client computes the combination of the challenge received and the password with the *hash()* function.
3. The client sends the value returned by the *harsh* function to the server.

Server

```
print("SERVER-4 (LONDON)")
print_colored("PORT 5055 is
    started active Please enter
    a key to continue", "green")
input()
challenge = "is_cool"
password =
    'communication_networks'
server.join_network()
server.unicastTOR(challenge,
    5060)
print("Challenge is", challenge)
print("Server: the hash is ",
    hash(password+challenge))
```

Client

```
password =
    'communication_networks'
print("Wait to receive the
    challenge then press enter")
test = True
while(client.MSGTOSAVE):
    print(client.MSGTOSAVE)
    donothing = 1
challenge = client.MSGTOSAVE
print("Challenge is ", challenge)
tosend = hash(password+challenge)
print("Client: the hash is ",
    hash(password+challenge))
input()
encrypted_message = str(tosend)
```

The general idea of the challenge response is understood, however, during the implementation, there were issues with the *hash()* function. Indeed, the function returned different values for different script (for instance, a script to emulate the client and a script to emulate the server) even though the challenge and the password were the same.

References

[1] Peer to Peer Network from Github https://github.com/vnknt/p2p_network

[2] Cryption library used from Github https://github.com/Legrandin/pycryptodome