Designing & Implementing a TOR Network

23.12.2022

Ianna FREITAS, Ingrid HASANAJ, Jawade BACHIR, Ming-Chen YU

Contents

Section 1: Introduction

Target of the project

The target of this project is the creation of a simple but functional TOR network that can accommodate a large number of users and offers the exclusive feature of anonymity together with a challenge-response authentication scheme for clients.

TOR Network

TOR stands for: "The Onion Routing" project and it is an open-source privacy network based on the TOR protocol that was built long before TOR browsing became accessible. This type of network enables anonymous web browsing by using secure, encrypted protocols to ensure that users' online privacy is protected. Practically, the traffic will be routed through three random nodes, encrypted, in order to make it impossible to track or read. The name 'Onion Routing' is related directly to the encryption that the data undergoes before being routed, namely, the data is encrypted three times, consecutively, creating an 'onion-like' layered packet. With each hop, one layer of encryption is "peeled off" letting the handling device know where the data is coming from and towards where it is supposed to relay the packet but not revealing anything more.

This does not break anonymity since there are intermediate steps, and none of the involved parties can have information on both the source and destination. In fact for the destination server, the last Tor node appears as the origin of the data.
Consequently, we can conclude that traversing the TOR network means going through a random node trio where the first one that the user contacts is called: the entry nodes or guard node, the middle node, and the exit node.

What happens is that:

1. The TOR daemon on the user's machine has to select the three nodes it wants in its circuit, ask about their availability, and then negotiate a symmetric encryption key with each of them (The user knows all three keys).
2. After having created a circuit, TOR will be using that for all of the traffic for the following 10 minutes before creating a new one.

3. Afterwards the new TCP streams are routed through the new circuit while existing TCP streams go through the old one.
4. The selection of the nodes is based on information on available nodes collected from directory nodes. That is why the TOR daemon needs access to fresh lists of all present TOR nodes and their capabilities.

Section 2: Architecture

2.1 Peer-to-Peer Connection

To create a peer-to-peer connection, we need two scripts. One is a sever, creating the small network; another is the client, the hosts that join this network and communicate with each other.[1]

2.1.1 Server

We first create a server class. This server here is only used to manage the network. For instance, when a new host tries to join the network, it will be connected through the server, but the communication among clients does not rely on the server.

First, we create two lists, one for connections, and one for the peers. Then we build the constructor of this class. In the constructor, the socket is set to internet socket(AF.INET), TCP connection(SOCK_STREAM). In the case server is disabled, we need to allow a client to use the same address to become a new server. Then, bind the server to the host and port. Then use the threading library to append the address of the peers that connect to this network.

Next, we define a handler to handle to peers. If clients are speaking, we receive data, all the peers would receive the message from the speaker. If a client is disabled, the server would remove the information of this host.

The last thing for the server, we define the function to send the list of peers to all hosts that connect to this server. In order to differentiate the message and the list of peers, we use "x11" at the beginning of the list.

2.1.2 Client

In the class client, we define a function "sendMsg" to send the message. In the constructor, the socket is set to internet socket (AF.INET), TCP connection (SOCK_STREAM) same as sthe erver. In the threading, we use the function we just defined to send the message. Then start an endless loop to make the client always listening to each other. When a client receives a message, it will use the beginning of the message to identify if the received message is a list of peers. There is also a function to update the peers.

2.1.3 P2P

All the peers who join this network will be added to the list. Initially, we put "127.0.0.1" in the list, which refers to the local host.

Implementation

We use an endless loop to realize the peer-to-peer connection. For all peers in the list p2p, we first try to become a client. If there is no server in this network, this attempt would fail. We pass this error and move on to become a server.

Since every host can become a server, it creates a problem when more than one host join the network at the same time. Therefore, we let new comer sleep for a random tie between 1 and 3 seconds. If more than one computer joins the new network at the same time, the one that wakes up earliest will become the server.

2.2 Encryption and Decryption

After creating a peer-to-peer connection, the next step is to do the encryption and decryption. We choose to implement AES (Advanced Encryption Standard)[2]. This is included in the library PyCryptodome.

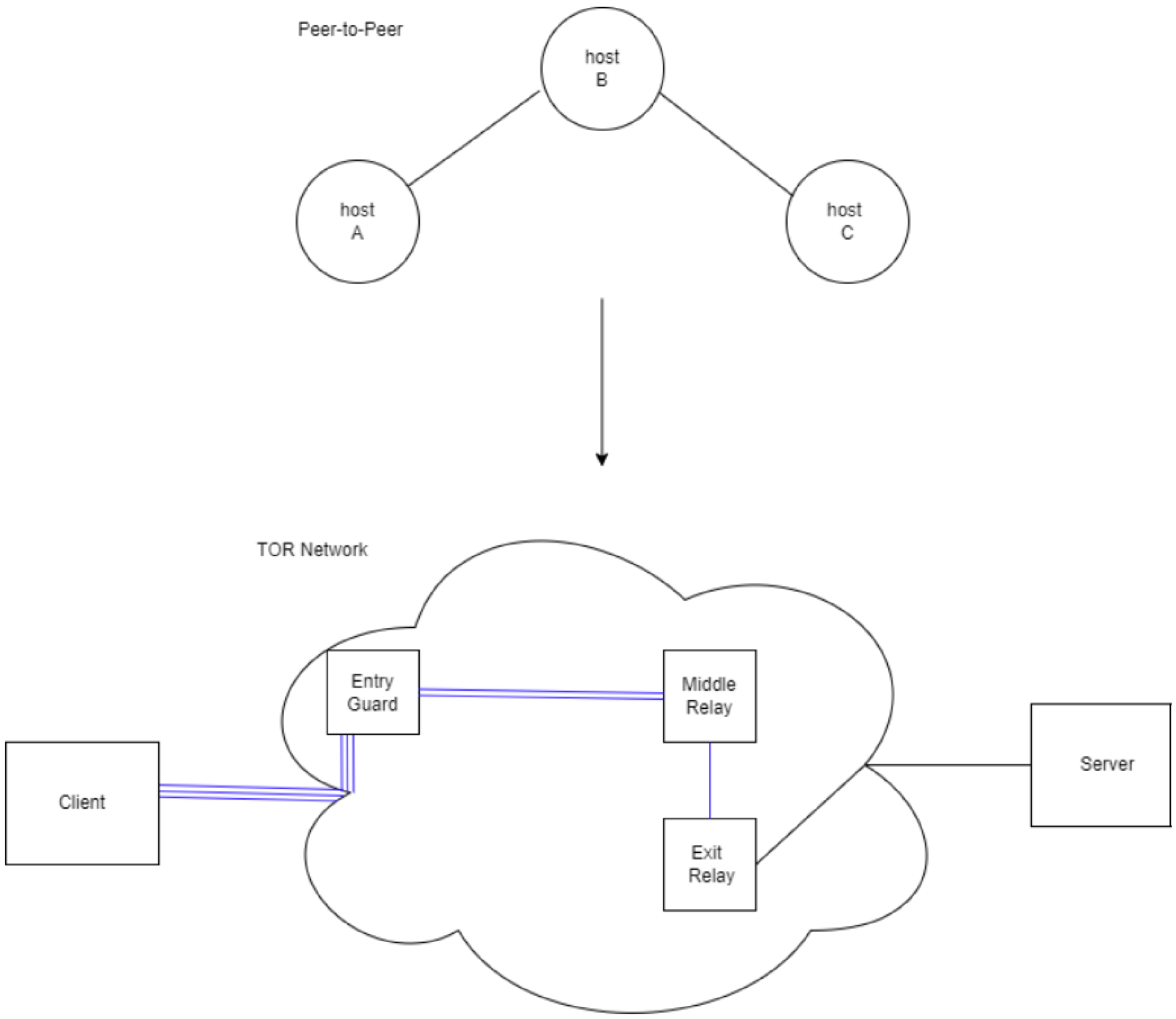
In the code, we first define a key that will be used to do the encryption and decryption. For the encrypt function, use the key and the mode MODE_EAX to do the encryption, then use the function in the library to encode the message to utf8. The function encrypt will return nonce, tag, and ciphertext (encrypted message).

For the decryption, we use the same mode as encrypt, and decode with utf8. The mode of decoding must comply with encoding such that the message can be recovered correctly.

After defining the encrypt and decrypt, we ask the user to enter a message. The output would show the encrypted message and the decrypted version of the message, which should be identical to the original message.

The next step is to apply encrypt and decrypt function to the peer-peer network.

An explanatory schematic is shown below.



Blue = encrypted

Section 3: Challenges

To start, a simple TCP chat program was made which connects multiple clients to one server. The clients can communicate between them within a public chatroom. Here we simply have a connection of multiple clients to one server using the socket and threading libraries in python. Quickly it was realized that this was not really peer-to-peer communication since the clients don’t communicate directly with each other but rather through a common server.

A new file that had real P2P communication was made. Once this was established and messages could be sent well from one node to another and also from one node to another while going through an intermediate node it was time to implement that which gives the TOR network its name.

Messages had to be encrypted thrice by the client and then at each node it arrived at one layer of encryption had to be removed so that when arriving at the third node the computer knew to which webserver it had to connect. For the encryption and decryption, the AES method was chosen. Some functions inside the client files were made to encrypt and decrypt a message in function of a given key. Each one of the nodes should share one of the keys with the client. This was the part that was difficult to implement for us. Since none of us were really experienced with Python it was difficult to link the different files of P2P communication and encryption/decryption to create the TOR network.