Communication Network and protocols

Project Report

-

ELEC-H417

Group 1

Ulysse Denis 493954
Nicolas Seznec 493632
Hamoumi Mohcine 494462
Salvatore Carluzzo 494506

-

December 23, 2022

Introduction

This project consists in designing a peer to peer TOR network, able to admit and manage a variable number of peers, and allow those peers to anonymously send requests to a server.

The following sections develop the implementation of the peer to peer TOR network. They explain how a peer establishes a circuit in the network, how the message is encrypted to ensure confidentiality and anonymity, and how the relays in the circuit handle received messages. Additionally, a section details the implementation of a server providing a challenge-response authentication scheme for clients.

In the following, a peer is also referred to as a *relay, hop* or *node*.

1. General architecture and TOR characteristics

The TOR network ensures anonymity by doing so-called onion encryption. Each time a node wants to transmit a message through the network, it first randomly generates a route composed of several active nodes from the network, constituting the hops of the circuit. The first and last hop are respectively called the entry and exit node. In this project, a fixed number of three nodes is used in a circuit, where the second hop is the middle node. This number was chosen as less hops do not provide sufficient anonymity and more hops do not necessarily improve anonymity and only increase the time necessary for the communication [1].

For each hop, a symmetric key is defined and shared between the considered hop and the sender node using a Diffie-Hellman key exchange protocol. The messages are then encrypted and decrypted using this key with the AES algorithm. More specifically the ECB mode of the AES encryption algorithm is used. This choice was made because it does not require an iv, unlike the other AES algorithms.

The message is then encapsulated and encrypted (more on the message construction in section 4) using the keys shared beforehand, once for each node in the circuit, in reverse order, starting from the exit node. When a message is sent, it is decrypted at each hop and forwarded to the following node, until it reaches the exit node, at which point the message can be fully decrypted and the request can be processed.

When the receiver wants to answer back to the sender, the response goes through the same hops in the inverse order. Instead of decrypting the message, the relays encrypt it with the corresponding key.

Currently, two major problems remain. First, how can the sender and the nodes of the path agree in a safe way on the key they will use. Secondly, as a relay can be part of several circuits, it means that it will have to encrypt (or decrypt) messages with the right key and transfer them to the right node. These two problems need a well structured protocol which will be the aim of the two next sections.

2. Peer to peer

Each **Node** has a **NodeServerThread** thread that will handle the sending and receiving of its messages. Each NodeServerThread is bound to a host and a port that constitute its address. They also have an id but that is just for reading facilities. When the **NodeServerThread** is started, it listens in an infinite loop for communication demands on his address. More specifically, it has a socket binded to its address that listens. When it hears that another Node wants to connect to him (socket.accept) it creates a new **ConnectionThread** that will handle the communication with this node through the socket used to contact him. ConnectionThreads are responsible for sending and receiving data and are associated with the node that created the NodeServer that spawned them. When a NodeServer wants to connect to another NodeServer, it uses the method *connect_to(address)* which creates a socket that connects to the wanted **NodeServerThread**. Then it gives this socket to a ConnectionThread that will handle this socket. This way, the communications are handled by the **ConnectionThreads** that receive and send data.

When a ConnectionThread receives data, it gives it to its bound Node through the handle_messages() method that puts the received messages in a queue to avoid data concurrency.

Then the main analysis of the data received is handled in the data_handler() method. It is there that the Node decides what to do with the message. For example, if the message received has the ping type, it updates its list of active Nodes (see Directory Node).

To avoid having too many ConnectionsThreads running at the same time. Each ConnectionThread has a timeout defined in their initialization. If the ConnectionThread is inactive during this timeout, it stops. This way, inactive connections are closed and active connections are kept running.

3. Diffie-Hellman Key Exchange

When a path is determined, the sender must agree with the three other nodes on specific symmetric keys that they will use in order to encrypt and decrypt the message (each node shares a key with the sender). Since the key is a critical element, the sender and the node must agree on a key without compromising it. This can be done with a *Difffie-Helfman key exchange method* as represented in the following figure:

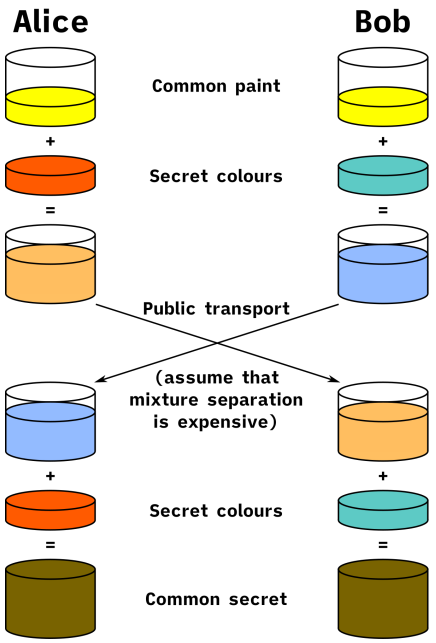

Figure 1 : Diffie Hellman key exchange [4]

The two nodes start with a common element (parameters of the algorithm, in this project common to every node) then add on it their secret component, share to each other this component and finally add again their secret element on the received packet. They obtain after this the same secret element, in our case the shared symmetric key.

More specifically in this project the common elements are : the prime p used in the modulo operation and the generator g that is put to the power of the secret key. In the code, the generate_shared_keys() function returns a hash of the shared key because the ECB encryption requires a 16 bytes long key. The hash allows you to get a fixed length output.

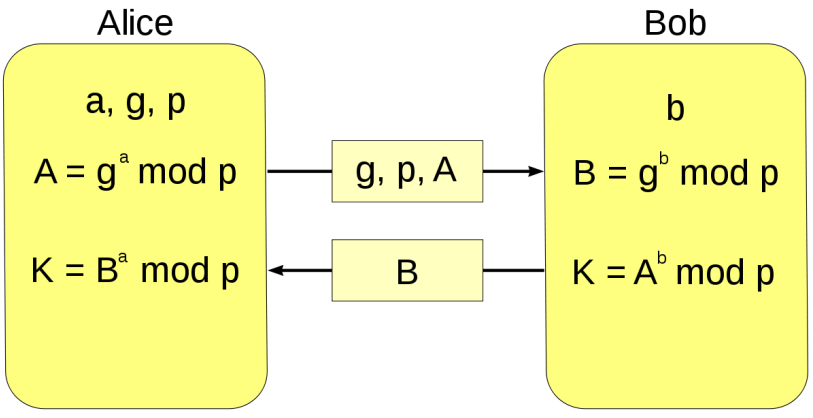

Figure 2 : Diffie Hellman exchange [2]

In order to exchange keys with each relay in the circuit, the sender follows a certain protocol. The sender first executes the key exchange with the entry node, without any encryption since no shared secret exists between the two peers at the beginning. From this point on, any message exchanged between the sender and the entry node can be encrypted. The encryption is performed using AES, as it is well known and widely used.

The sender then carries on the key exchange with the second hop through the entry node, first encrypting the message for the first hop. The message is decrypted by the first hop and sent to the second hop which answers the key exchange request. His response is first sent to the first hop which encrypts it with its key and then sends it back to the client[1]. As it can be seen, it follows the same logic of the TOR implementation but only with a partial circuit. When the sender proceeds to the key exchange with the third and last hop, it encrypts the message two times before sending it.

The answer follows the same logic. The figure below illustrates this mechanism:

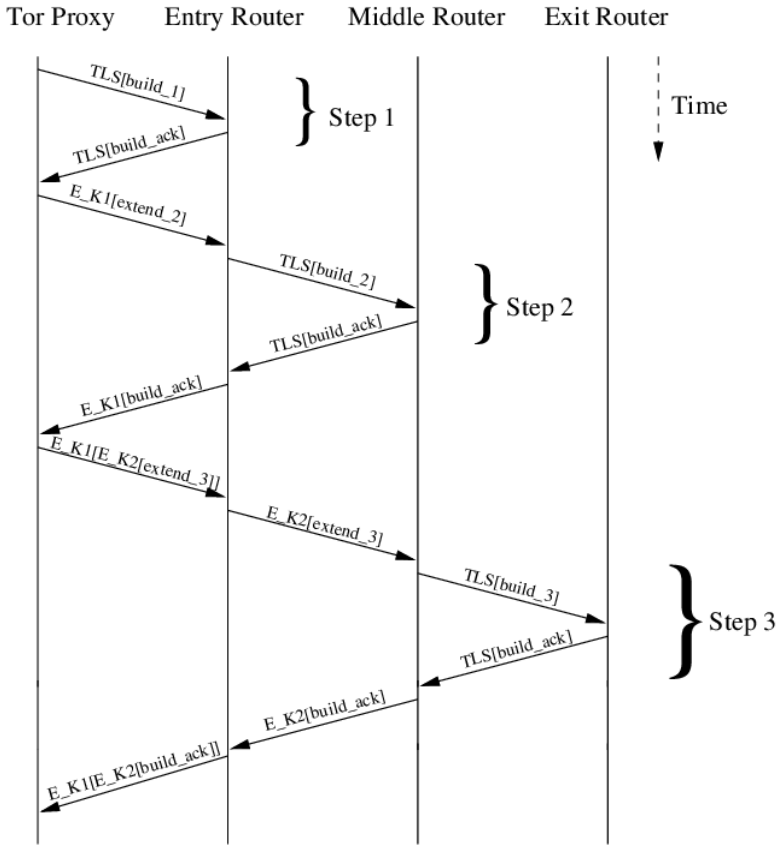

Figure 3 : Illustrating the protocol used to generate a circuit through the network. [3]

This protocol allows to obtain a safe key exchange and to keep the anonymity of the sender when exchanging the key (as it should be done in a TOR network). In parallel with the key agreement between the sender and each one of the three hops, a message id is randomly computed by the sender. That id is necessary as it will help the relays to distinguish the messages from each other, and know which key to use in order to encrypt or decrypt a message.

In the code, the key exchange is initiated by the *launch_key_exchange()* method, which returns the shared key list. To wait for a key to come back, it starts a **WaitingThread** that waits until the pending_key_list has changed. This list changes when the node gets the public key back in the diffie-hellman protocol.

[1] *The user relying on the tor network to transmit its requests is appointed as 'client'.*

3. Message Routing

As mentioned, different message ids are created during the key generation. To better showcase their utility, let's use an illustration:

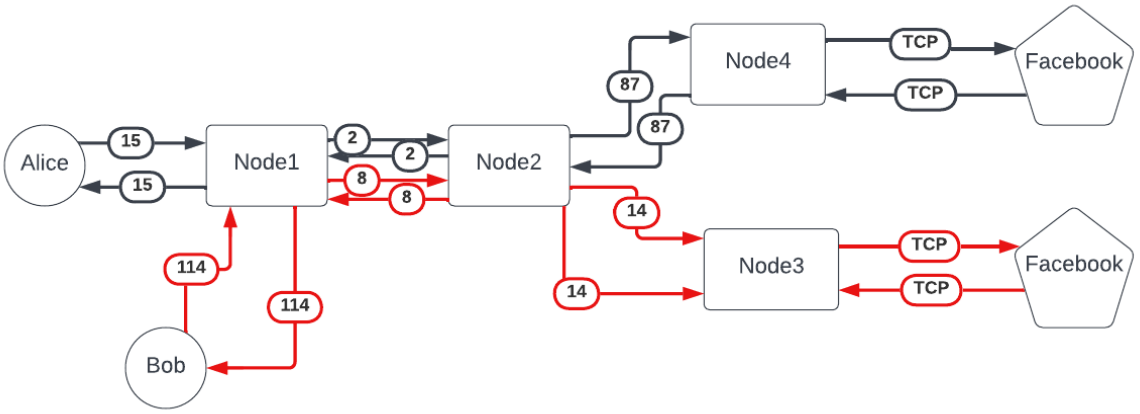

Figure 4 : Message routing using ids

In the figure above, two different circuits contain the same two nodes. This means that Node2 must be able to differentiate between a message sent from Alice and one sent from Bob (without knowing the source, obviously), even though they both come from Node1. In order to achieve this, when the key exchange takes place, the client assigns an id to each message in the circuit, that the relay can associate to the shared key.

When a relay encounters the same id from the same source when receiving the actual message, it knows which key to use for decryption. It also associates the key to the id of the forwarded message, so that when the response comes back, it knows how to encrypt it and where to send it back.

In the example, Node2 receives a message from Node1 containing the id 8. As Node2 associated the id 8 with the key shared with Bob during the key exchange, it knows what to use for decryption. From the decrypted message, it retrieves the next node (Node3) and the id that will be used on the way back (14). When the response comes back with the id 14, Node2 knows to send it with the id 8 and to encrypt it using the key shared with Bob.

The message ids are uuids. The chance of two messages having the same Universally Unique IDentifier (uuid) is extremely small, but both the uuid and the destination of each message are checked to ensure that they are not mistaken for one another. This is a precautionary measure, as the likelihood of two messages having the same uuid and being intended for the same destination is considered to be practically nonexistent.

Each Node has a **Table** that works like a database with two tables : the key table and the transfer table. The key table is a dictionary with tuples (message_id, address) as keys and the shared encrypting/decrypting keys as value. These tuples and keys are stored during the key exchange. So each time a message is received and is a transferable type (e.g. not “ping” or “key”), the node looks in its key table which key to use to decrypt/encrypt the data of this message.

The transfer table is a dictionary with tuples (message_id, address) as keys and values. The key tuple corresponds to “the exit node side” and the value to “the client side”. This table goal is to be able to determine to which Node the data has to be transferred on its way back. Indeed, in the “decryption” way, the node knows where to transfer the message because the information is contained in the decrypted data. But on the way back, the node has no way to know which way to forward the message.

Concretely, when a message is received, the node first checks if the (message_id, address) is in its transfer table. If it is, it knows that it is on its way back and that it has to encrypt it and then transfer it to the right node (found in its table). If it is not, it knows that it should decrypt it and then analyze what to do with the decrypted data.

In the key table, only the “client side” id and address are necessary because if it is coming from the “exit node side”, the node can easily retrieve the “client node side” id and address and thus the key.

To keep only the useful data, when a path will not be used anymore, the exit node puts a mark on the message. This mark indicates to the nodes receiving this mark that they should delete the information linked with this path. In clear, it deletes one the corresponding lines from the key and transfer tables.

4. Message Construction

Each message is a dictionary that has a defined pattern. They all have an id, a type, a destination (receiver), a source (sender) and of course the content of the message itself (data). Each of these information will allow the method handle_data() to know what to do when a message is received. For example, the type can be *“key”*, the node understands then that the coming message is Difffie-Helfman key exchange.

The id is a random uuid generated by the original sending node and is specific to an exchange between two nodes for one TOR path as explained above.

Because the sockets only accept bytes-typed data, it is not possible to send a dictionary. To be able to send a dictionary through a socket, the pickle dumps and loads methods have been used. They respectively convert an object in a pickle object and the inverse. A pickle object is Python specific data. This package has been chosen because it allows to convert dictionaries with bytes in it (instead of json for example).

When a user wants to make an anonymous request through the tor network, it tags it with the “request” type and then sends it. It has to follow a definite pattern. Since the request will then be achieved with the Request package, the method of the request has to be one of the methods of this package ("get", "delete", "head", "post").

The request will be done by the exit node. It is the connection of our tor network with the outside world. The result of the request is then sent back to the user. If the request has encountered an error, it returns the error.

5. Directory Node

Up until this point, it was assumed that the peers knew each other. However, when a new peer joins the network, it has no knowledge about any node. To allow the peers to make initial contact, a directory authority keeps track of every active node. The directory node is a node that acts as the directory authority, and informs the nodes of the network of the currently running relays. A newcomer can therefore contact the directory node periodically to both retrieve the updated list of active nodes, and signal its own presence, to be added to said list.

In the scope of the project, there is only a single directory node that the peers know because its address is hardcoded. In a more elaborate system, there could be several directory authorities that would need to agree with each other and publish a consensus of the active nodes. Accessing this list could be done via the internet. On top of that, there would be measures to ensure the nodes can be trusted, especially entry nodes, as they are the ones directly interfacing with clients.

6. Challenge-Response authentication

Since the whole point of the peer-to-peer tor network is to attain a certain level of confidentiality and anonymity, it should be possible for any client relying on the network to perform anonymously a *'Challenge-Response' (C-R)* authentication protocol based on a password, without leaking this password at any step of the authentication of course.

As the exit node[2] has the ability to read the unencrypted data provided by the client, a basic shipment of the password to the server leaks it whenever the password has to go from that last node to the server. Any eavesdropper could then usurp the client's identity.

At this point we can define mandatory specifications that our *C-R* must meet. Firstly, no password needs to be known on the server side, which increases the confidentiality of the password in case of a security breach. Secondly, no leak of the client's password. And finally, the response of the client should give valid proof that the sender knows the password.

[2] *Last node of the circuit, it decrypts the last layer of encryption before forwarding the request on the internet.*

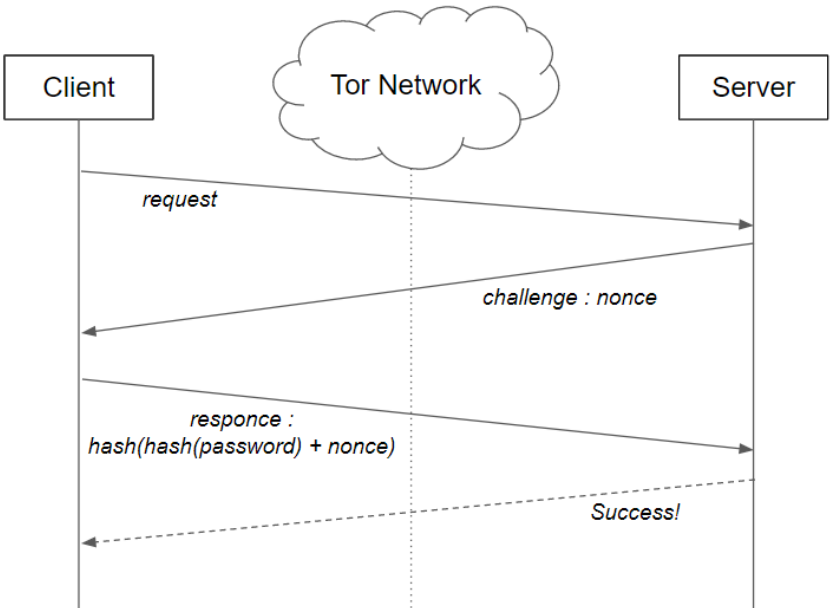

Figure 5 : Challenge-Response exchange.

In order to authenticate that the request comes from a known user, the server gives a nonce[3] as a challenge. When received at client side, the client will compute the response as followed : $Response = SHA256(SHA256(password) + nonce)$.

The server receives the response and then compares it with its own computed value. At this point, both client and server should have computed the same expected response and the only way for a client to give that response is to actually know the password, or more precisely the *hash of the password*.

Indeed, this particular scheme of *C-R* respects all the three constraints enumerated previously. The password is never sent nor leaked at any step of the communication. Nor is the password stored in the server database since it only needs $SHA256(password)$ to verify the client.

A relevant review of this *C-R* protocol is that it is sensible to *eavesdrop* and *man-in-the-middle* attack. Indeed, any MiM could potentially place itself between the exit node and the server, listen to the response sent by the client and modify the request. Fortunately, if the path is changed each new request sent in the tor network, it is not very likely to happen because the exit node changes each time. But if a user has the control of multiple nodes it could be a problem.

The main problem lies in the registration process because if the exit node sees the hash of the password (eavesdrop), it can then authenticate with the client's identity. In this project, a solution has been found based on the actual tor network that uses especially https secured connections (see Client Registration). Moreover, if an exit node steals the hash of the password, it is not a problem because both client and server could agree on another

[3] *Random number used once in a lifetime.*

hash function[4] and redo all their *C-R* protocol with that new function. This last solution will need to update the server database with $Hash_{new}(password)$ instead of the corrupted one.

6.1. Client Registration

Whether it is to register a new client on the server database or to replace a compromised hash with a new one, both client and server will need to confidentially share sensitive information : the hash of the password. In this project, whenever a client wants to register itself in the server database, it creates an additional key exchange request with the server *cf.[1.2]* and proceeds to send the hash of the password as encrypted data. The channel created is secured and thus the hash of the password does not leak. After decryption, the server reads and stores $Hash(password)$ and wait for any new request coming from that user to run a *C-R* authentication. Note that the user can also register by sending its $Hash(password)$ in clear. Both options are available in the register() method in the Node class.

6.2. Authentication implementation

The Authentication class represents the authentication server. It inherits from Node only for practical reasons but it is **absolutely not a Node**. It means that it does not ping the directory node and thus does not appear in the active nodes. Therefore it will never be a relay in a generated path when trying to send a message through the tor network. However, it handles its tcp communication the same way as the nodes.

The server has a user table which stores the usernames and password hashes.

Conclusion

This project represents a tor network where the clients can make anonymous requests. The challenge-response implementation is the representation that a user can anonymously authenticate. The main challenges were :

- To implement a peer to peer network : a solution has been found using threads.
- To be able to exchange keys with the other nodes anonymously.
- To be able to send a request anonymously.
- To be able to authenticate to a server without leaking the password using a challenge response based scheme.

[4] *SHA512 or SHA384 for example.*

Annex:

1. Sources

[1] https://community.torproject.org/relay/types-of-relays/
[2] https://fr.m.wikipedia.org/wiki/Fichier:Diffie-Hellman-Schl%C3%BCsselaustausch.svg
[3] Brown, Michael & Brown, Michael & Little, Herbert & Kirkup, Michael. (2013). Challenge response-based device authentication system and method. https://www.researchgate.net/publication/302659833_Challenge_response-based_device_authentication_system_and_method