ULB UNIVERSITÉ LIBRE DE BRUXELLES

Faculty of Engineering
1st Master Electrical Engineering

Communication Networks: Protocols and Architectures

-

Project: Shallot Routing

Mathias De Roover, Quinten Deliens,
Guylian Molineaux, Remco Royen

Prof. Dr. Ir. Jean-Michel Dricot
Ir. Soultana Ellinidou

December 22, 2017

Contents

1 Getting started **2**
1.1 Adding the libraries . . . 2
1.2 Initiating the network . . . 2
1.3 Running the network . . . 3

2 Code walkthrough **4**
2.1 Shallot Client . . . 5
2.1.1 Initializing Shallot Client . . . 5
2.1.2 Sending messages . . . 5
2.1.3 Receiving messages . . . 7
2.2 Shallot Relay . . . 9
2.2.1 Relay running . . . 9
2.3 Sequence diagram . . . 12

3 Faced problems and solutions **13**
3.1 AES key and block size . . . 13
3.2 Changing IP addresses . . . 13
3.3 Handling errors . . . 13

4 Example of the network in action **14**

1 Getting started

This section contains everything you need to know in order to correctly run the code, which is written in Python 3.6.3.

1.1 Adding the libraries

First of all, besides the default modules which come with the Python 3.6.3 installation process, external modules are used and thus should be installed before running the code. This section explains how to install them.

Cryptography

In the command line type: "`pip install cryptography`". [2]
(Or "`py -3 -m pip install cryptography`" or "`pip3 install cryptography`" when running multiple versions of Python.)

Pycrypto

In the command line type: "`pip install PyCryptoDome`". [3]
(Or "`py -3 -m pip install PyCryptoDome`" or "`pip3 install PyCryptoDome`" when running multiple versions of Python.)

1.2 Initiating the network

Important note on addressing

For the network to work on your computer, the correct IP address needs to be inserted in the configuration files. This should be done automatically when running the code (lines 8 to 10 in "`Shallot_Network.py`"). However, should this not work for some reason, one can always do this by manually editing both the configuration files "`config\host.ini`" and "`config\topology.ini`" with an editor of choice. Just change every IP address to the IP address of your computer and you are good to go. Also make sure to remove lines 8 to 10 in "`Shallot_Network.py`"!

The reason why every IP address has to be the same, is that we are simulating the network on one computer and thus the role of the IP address in the simulation will be taken over by the ports of the computer.

Node configuration and network topology

The default network topology is given by Figure 1.

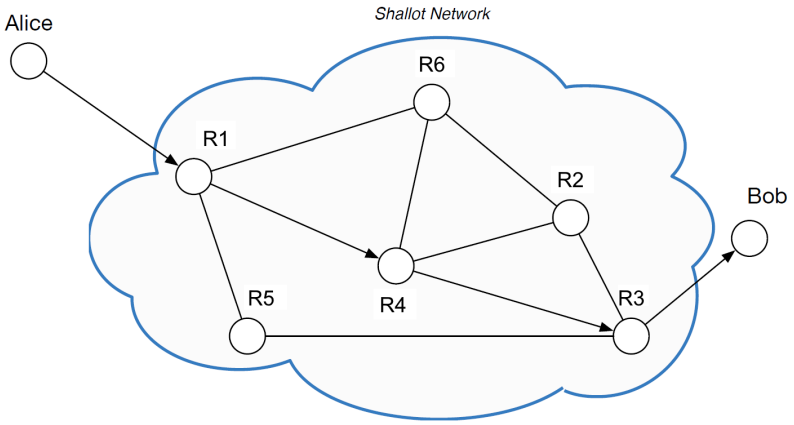

Figure 1: Default topology [1]

If one would like to change the IP and port of a client or add or remove one, this should be done inside both "`config\host.ini`" and "`config\topology.ini`" under `[clients]`. For relays, one should edit both "`config\host.ini`" and "`config\topology.ini`" under `[relays]`. Note that in order for the changes in IP address to remain, one should remove or comment lines 8 to 10 in "`Shallot_Network.py`".

Furthermore a topology of choice can be implemented in "`config\topology.ini`". Under `[topology]`, one can change for each client and relay their corresponding neighbors. One should however always make sure a client or relay is a neighbor of itself, which should be the first element of the neighborlist, as well as making sure the order of the elements under `[topology]` is the same as the order of the clients and relays under `[clients]` and `[relays]` inside "`config\topology.ini`".

When editing these files, one should always make sure the addresses of the clients and relays are the same in both the configuration files. There should be no unconnected nodes and a path (via relays) between the clients should always be present.

1.3 Running the network

Inside the project folder, one should find the file "`Shallot_Network.py`", which reads in the configuration files and initiates the hosts and relays (as separate threads) and also provides a user friendly interface for sending messages between two hosts of choice. Run this file from the command line and the network is ready to use! (Make sure the file is run with Python version 3!)

2 Code walkthrough

For ease of usage and overview, the code is split into 9 different parts, which all in someway are connected to the main file "`Shallot_Network.py`". Figure 2 gives a schematic representation of these relations, while also briefly explaining the function of each code block.

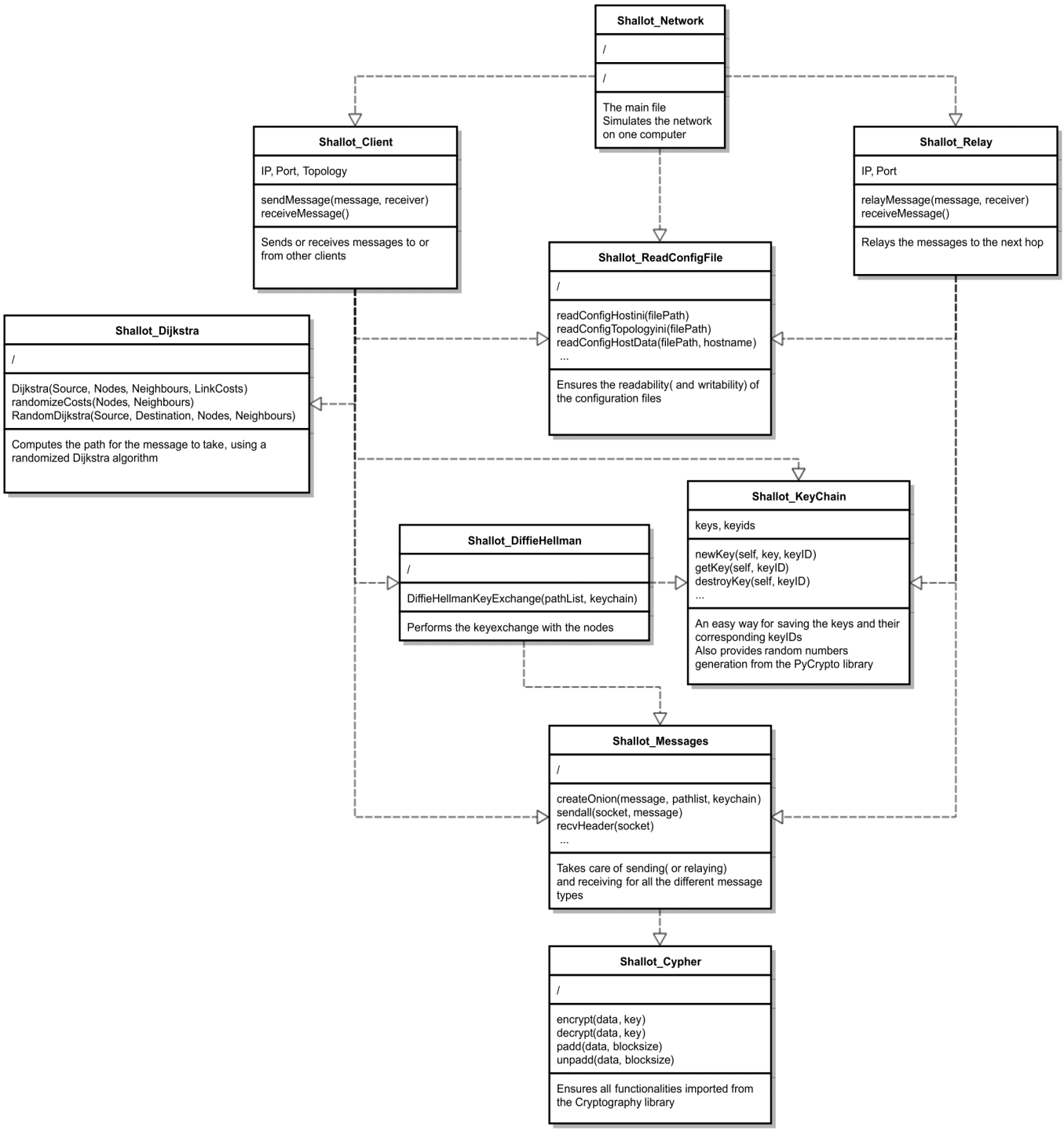

Figure 2: Code structure [4]

2.1 Shallot Client

2.1.1 Initializing Shallot Client

When a client is initialized, a keychain is made and a socket service is started. Last but not least the IP-address and port of the client are read (by the function `shallot_readConfigHostData`) from the hostfilepath that is given as an argument. It is important to notice that every client (and also every relay) only uses "`host.ini`" to determine their own IP-address and port. This ensures that if the code would be used to work on different machines, in every "`host.ini`" only the names of the clients (and relays) at that machine should need to be mentioned.

This also means that all the information about the network must be derived from "topology.ini" and hence entities with no access to this file cannot discover the actual structure of the network and information about the others.

```
    def __init__(self,hostfilepath,topologyfilepath,hostname):
        self.hostname = hostname
        self.listenKeychain = Shallot_KeyChain.shallot_KeyChain()
        self.IPs,self.Ports,self.Neighbours =
            Shallot_ReadConfigFile.shallot_readConfigTopologyini(topologyfilepath);
        self.hostIP, self.hostPort =
            Shallot_ReadConfigFile.shallot_readConfigHostData(hostfilepath,
                self.hostname);
        self.listensock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.listensock.bind((self.hostIP,self.hostPort))
        self.listensock.listen(1)
        self.isRunning = True;
        threading.Thread.__init__(self)
```

Listing 1: Code for initiating Client

2.1.2 Sending messages

Clients in the shallot network, like Alice and Bob, send their messages using the function `sendMessage`, found in the `Shallot_Client` module. This function takes as an argument the plaintext message and the recipient address (IP and port). The code is given in Listing 2.

```
    def sendMessage(self,message,recipient):
    #============================================================
    # Executed to send a message
        message_bytes = str.encode(message)

        print("Computing path...")
        source = [self.hostIP, self.hostPort]
        destination = [recipient.hostIP, recipient.hostPort]
        pathlist = Shallot_Dijkstra.shallot_RandomDijkstra(source,destination,
            self.IPs,self.Ports,self.Neighbours)
        # pathlist nested list of this form: [[recipient.hostIP,recipient.hostPort]]
        # (in reversed order!)
        print("Path found:")
        print(list(reversed(pathlist)))

        onionKeychain = Shallot_KeyChain.shallot_KeyChain();
        Shallot_DiffieHellman.shallot_DiffieHellmanKeyExchange(pathlist,
            onionKeychain)
        print("Keys exchanged")
```

```
        payload = Shallot_Messages.shallot_createOnion(message_bytes,pathlist,
            onionKeychain)
        onionKeychain.clear()

        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((pathlist[-1][0],pathlist[-1][1]))
        Shallot_Messages.shallot_sendall(sock,payload)
        sock.close()

        print(self.hostname+" sent: "+message)
    #============================================================
```

Listing 2: Code for the sendMessage function

To start: the plaintext message is encoded in a sequence of bytes. Then the path will be computed using the `shallot_RandomDijkstra` function from the `Shallot_Dijkstra` module. This function takes following inputs:

- `source` and `destination`: containing the IP address and port number of respectively source and destination
- `IPs` and `Ports`: 2 list containing respectively the IP addresses and port numbers of all relays in the network as well as the IP address and port number of both source and destination
- `Neighbours`: a nested list containing at each entry the list of neighboring nodes to the node corresponding to that entry.

All these inputs are obtained by the sending host by reading the `config/topology.ini` file.

The `shallot_RandomDijkstra` function implements the modified Dijkstra algorithm that assigns a random cost between 1 and 16 to each link, before calculating the path using Dijkstra. The result is saved in `pathlist` which is a nested list containing the sequence of addresses (IP and port) of the nodes in the path in reversed order, i.e. from destination to source.

Then a `shallot_KeyChain` object, `onionKeychain`, is created. Using the Diffie-Hellman key exchange from the `shallot_DiffieHellman` module, this objects' `keys` and `keyids` lists are filled with respectively a key and corresponding key ID for every node in the computed path.

Once the key exchange is finished, the shallot will be built. This is done using the `shallot_createOnion` function from the `Shallot_Messages` module. This function builds the payload starting from the byte encoded message and executing the following steps:

1. The byte encoded message is the payload for the first shallot layer
2. Using the function `shallot_constructOnionPeel` from the `Shallot_Messages` module, the next layer of the shallot is built.
   (a) The previous shallot becomes the payload of this new message
   (b) To this payload, the IP and port of the next hop are added
   (c) This data is encrypted with the AES algorithm using the `shallot_encrypt` from the `Shallot_Cypher` module
   (d) The correct message header and key ID of the used key are added before the encrypted data

3. Step 2 is repeated until the source node is reached

Once the message is fully encoded the keys are no longer needed, thus the negotiated keys are removed from the `keys` list and their corresponding key IDs from the `keyids` lists of `onionKeychain`.

Finally a socket at the sending client's side (with a connection to the first node from the path, which is the last node in `pathlist`) is opened and used to send the shallot message. After this the socket is closed again and the sending process is completed.

2.1.3 Receiving messages

Of course a client must not only be able to send a message but also to receive one. This is done by using the function `run`, found in `Shallot_Client.py`. The code of this function can be found in Listing 3. Since every client that is initialized is a thread, this function can be executed continuously until `isRunning` is put to False by the function `close` without blocking the rest of the program.

The main goal of this function is to continuously listen to possible entities that are trying to connect to this client. When a connection is made, the data is received and the type of message is determined. This is done by examining the header. Every type of message is handled in a different way. Afterwards the connection through the socket needs to be closed. In the following paragraphs these types of messages and the way they are handled are examined. To make it less confusing, the receiving client will be called 'Bob' and the sending client 'Alice' (although the program is written in such a way that Bob can also send messages to Alice or every other client that is added to the topology).

Wrong message format When receiving a message with a wrong header format, an error message is generated and send to the entity that sent the faulty message, who is waiting at the other side of the connection.

Key initialization message These messages are received when Diffie–Hellman is being executed by Alice. When receiving it, a check needs to be done whether or not the keyID that Alice proposes is already present in Bob's key-chain. When Bob already has the keyID an $INVALID\_KEY\_ID$ error is send to Alice. If the keyID can still be used, then Diffie–Hellman is executed by Bob and a key reply message is send to the Alice.

Key reply message Normally Bob (a client in receiving mode) should not be able to receive these messages. This is because the Diffie–Hellman code is never initialized while listening but only when sending (see paragraph 2.1.2). Upon the case that a key reply message is received Bob prints a warning to the console and continues normal operations.

Relay message When Bob receives a relay_message, he first needs to check if the message is in a correct form and whether or not he is the final destination of the message. If the message is not meaningful, an error message is send to the sender of the message and the connection is closed. When this is not the case and it is determined that Bob is the final destination, the message must be decoded. This is done by `decode()`. At this moment Bob has received the message from Alice as it was intended. As a last step this message is printed. If the message was meaningful but Bob was not the final destination that means that something went wrong and a warning is printed before continuing normal operations.

Error message Upon receiving an error message, a corresponding message is displayed on the console and Bob continues its normal operations.

```
    def run(self):
        #=======================================================
        # To Receive messages
        while(self.isRunning):
            timeout = 1
            readable,writable,exceptional =
                select.select([self.listensock],[],[],timeout)
            for sockread in readable:
                conn,addr = sockread.accept()
                msg_version,msg_type,msg_length =
                    Shallot_Messages.shallot_recvHeader(conn)

                if not(msg_version==1 and  msg_type>=0 and msg_type<=3):
# If the format of the header is wrong, generate and send error message
                    Shallot_Messages.shallot_sendErrorMessage(conn,0)

                elif msg_type == 0:
# A KEY_INIT message has been received, calculate key and send KEY_REPLY message
                    keyID,g,p,A =
                        Shallot_Messages.shallot_recvKeyInit(conn,msg_length)
                    if self.listenKeychain.hasKey(keyID):
                        Shallot_Messages.shallot_sendErrorMessage(conn,1)
                    else:
                        b = Shallot_KeyChain.randomInt(1024)
                        B = pow(g,b,p)
                        s = pow(A,b,p)
                        self.listenKeychain.newKey(s,keyID)
                        Shallot_Messages.shallot_sendKeyReply(conn,keyID,B)

                elif msg_type == 1:
# A KEY_REPLY message has been received, generate key for commuication with relay
                    print("You are not supposed to receive these messages here")

                elif msg_type == 2:
# A Relay_Message has been received,
# decrypt, generate new header and send to next relay
                    nextHop,nextHopPort,message_received_bytes =
                        Shallot_Messages.shallot_recvMessageRelay(conn,msg_length,
                            self.listenKeychain)
                    if nextHop == None and message_received_bytes == None:
                        error_code = 1
                        Shallot_Messages.shallot_sendErrorMessage(conn,error_code)
                        print("Key not found\n An error message has been sent
                            (INVALID_KEY_ID)")
                        conn.close()    #because of the continue
                        continue
                    if nextHop == self.hostIP:
                        message_received = message_received_bytes.decode()
                        print(self.hostname+" received: "+message_received)
                    else:
                        print("Error: "+self.hostname+" is not final destination")

                elif msg_type == 3:
# An error message has been received. No idea yet what to do then
                    error_code =
                        Shallot_Messages.shallot_recvErrorMessage(conn,msg_length)
                    if error_code == 0:
```

```
                    print("an error message arrived: INVALID_MESSAGE_FORMAT")
                if error_code == 1:
                    print("an error message arrived: INVALID_KEY_ID")
                else: print("an error message arrived: UNKNOWN")

            conn.close()
        #============================================================
        #executed upon closing the thread
        self.listensock.close()
        print("closed client "+self.hostname)
```

Listing 3: Code for the run function

2.2 Shallot Relay

The relays are coded in a similar way to the clients and hence there shall be often references to Chapter 2.1. Every relay on its own is a thread which has its own socket listening for incoming messages. The thread has to be initialized which happens with the code from Listing 4. This gets the address for the relay, a key-chain and starts up a socket service. The code to stop a relay is given in Listing 5.

```
    def __init__(self,hostfilepath,hostname):
        self.hostname = hostname
        self.listenKeychain = Shallot_KeyChain.shallot_KeyChain()
        self.hostIP, self.hostPort =
            Shallot_ReadConfigFile.shallot_readConfigHostData(hostfilepath,
                self.hostname);
        self.listenSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            # listenSocket = socket that listens for incoming requests
        self.listenSocket.bind((self.hostIP,self.hostPort))
        self.listenSocket.listen(1)
        self.isRunning = True;
        threading.Thread.__init__(self)
```

Listing 4: Code for the relay initiation

```
    def close(self):
        self.isRunning = False
```

Listing 5: Code to stop the relay

The main functionalities of every relay consist of the ability to agree on a key with the client who is sending a message (Alice) and to be able to relay the message forward to the next relay. These functionalities are given by the main function of the relay: "`shallot_Relay.run()`". (As one will notice, this function will have a lot of similarities with Listing 3 and Chapter 2.1.3.)

2.2.1 Relay running

Every time the code (see Listing 6) is run through, a check is done to see whether someone is trying to connect to the relay. A connection is made in this case and a message is sent to the relay. Upon receiving the message only the header is read. Using the information in the header it's decided what kind of message arrived using the `message type` field of the header. All different kind of messages are handled differently. The type of messages are the same as in Chapter 2.1.3 and since some types are handled in a completely analogous way, only a reference to this chapter will be given.

Wrong message format See Section 2.1.3.

Key initialization message Analogous to Section 2.1.3.

Key reply message Normally a relay should not be able to receive these messages. This is because the Diffie–Hellman code is never initialized on a relay, a relay can only respond to the code run by a client. Upon the case that a key reply message is received the relay prints a warning to the console and continues normal operations.

Relay message When a relay receives a message it should relay it and thus `shallot_recvMessageRelay()` is called. This function automatically decrypts the message and peels one layer of the onion. Besides the decrypted message that needs to be forwarded, it also returns the IP and port of the next hop. A connection is then made with this next hop and the message is sent.

Error message See Section 2.1.3.

```
    def run(self):
        #============================================================
        while(self.isRunning):
            timeout = 1
            readable,writable,exceptional = select.select([self.listenSocket],
                [],[],timeout)
            for sockread in readable:
                conn,addr = sockread.accept()
                msg_version,msg_type,msg_length =
                    Shallot_Messages.shallot_recvHeader(conn)

                if not(msg_version==1 and  msg_type>=0 and msg_type<=3):
# If the format of the header is wrong, generate and send error message
                    Shallot_Messages.shallot_sendErrorMessage(conn,0)

                elif msg_type == 0:
# A KEY_INIT message has been received, calculate key and send KEY_REPLY message
                    keyID,g,p,A =
                        Shallot_Messages.shallot_recvKeyInit(conn,msg_length)
                    if self.listenKeychain.hasKey(keyID):
                        # KeyID already in use
                        Shallot_Messages.shallot_sendErrorMessage(conn,1)
                    else:
                        # Apply Diffie-Helmann
                        b = Shallot_KeyChain.randomInt(1024)
                        B = pow(g,b,p)
                        key = pow(A,b,p)
                        self.listenKeychain.newKey(key,keyID)
                        Shallot_Messages.shallot_sendKeyReply(conn,keyID,B)

                elif msg_type == 1:
# A KEY_REPLY message has been received, generate key for commuication with relay
                    print("Relays arent supposed to receive these messages")

                elif msg_type == 2:
# A Relay_Message has been received,
# decrypt, generate new header and send to next relay

                    nextHopIP,nextHopPort,payload =
                        Shallot_Messages.shallot_recvMessageRelay(conn,msg_length,
```

```
                            self.listenKeychain)

                    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                    sock.connect((nextHopIP,nextHopPort))
                    Shallot_Messages.shallot_sendall(sock,payload)
                    sock.close()

                    print("Message relayed")

                elif msg_type == 3:
# An error message has been received. No idea yet what to do then
                    error_code =
                        Shallot_Messages.shallot_recvErrorMessage(conn,msg_length)
                    if error_code == 0:
                        print("an error message arrived: INVALID_MESSAGE_FORMAT")
                    if error_code == 1:
                        print("an error message arrived: INVALID_KEY_ID")
                    else: print("an error message arrived: UNKNOWN")

                conn.close()
        #============================================================
        #executed upon closing the thread
        self.listenSocket.close()
        print("closed relay")
```

Listing 6: Code to run the relay

2.3 Sequence diagram

The previously explained workings are summarized in Figure 3.

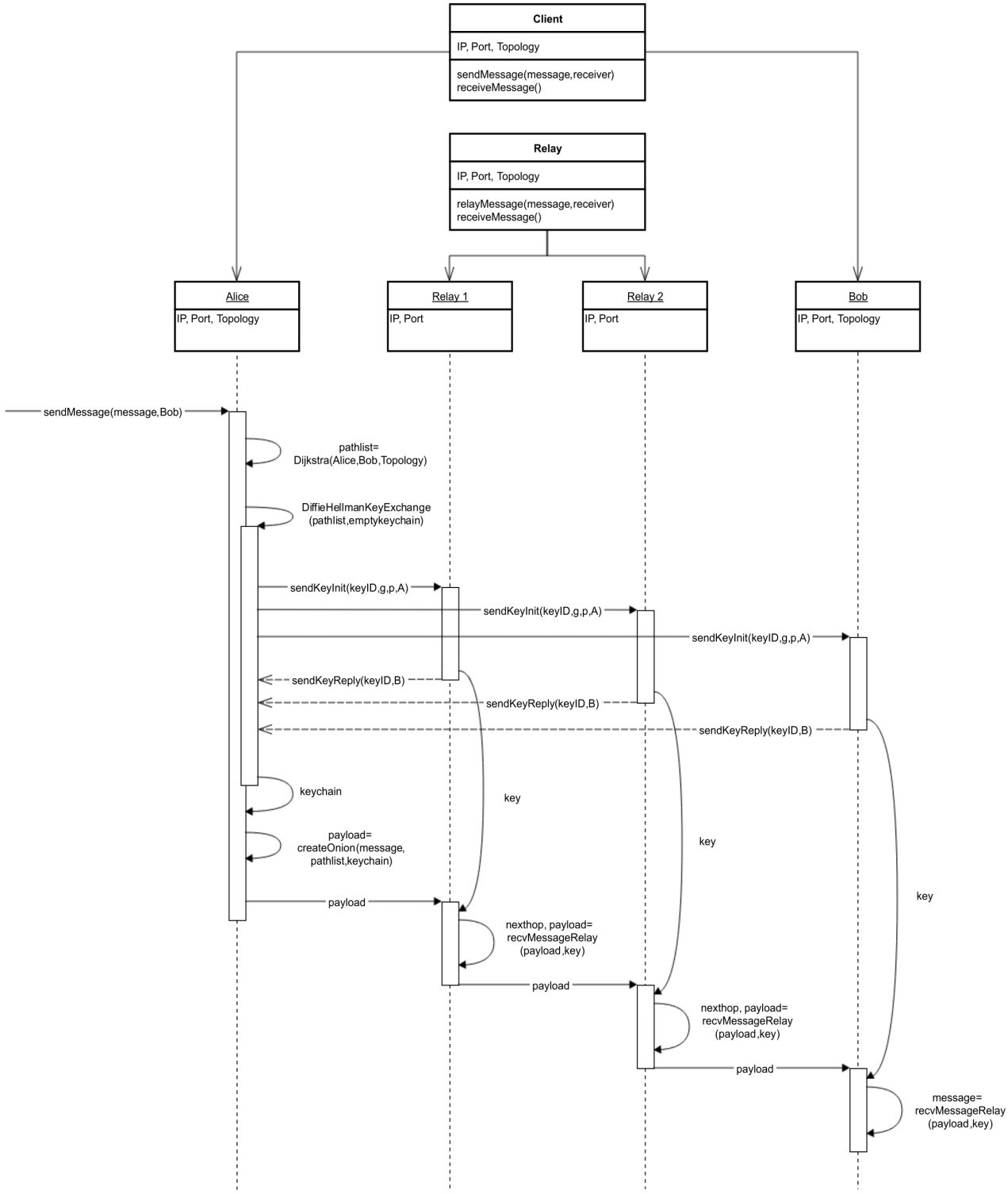

Figure 3: Sequence diagram [4]

3 Faced problems and solutions

3.1 AES key and block size

AES is a standard algorithm with predefined key and block sizes. The possible key sizes for AES are 128, 192 or 256 bits. The only possible block size is 128 bits.[2] A faced difficulty was to adapt the 1024 bit key size and 32 bit block size from the assignment to the AES algorithm. The way this is solved for the keys is by using only part of the longer keys. The least significant 256 bits of the 1024 bit key are used for the AES algorithm. The most significant bits of the key are also used in the code. The 128 most significant bits of the key are also used as the initialization vector of the AES algorithm, since this vector needs to have the same size as the block size.

To change the 32 bit block length of the assignment to the 128 bit needed to encrypt, padding was used. Before encrypting, the message is padded to the correct size and after decrypting, the padding is removed to get the original message.

3.2 Changing IP addresses

Originally every time the code was run on a different computer, the IP address of that computer had to be changed in all the configuration files. This was a lot of extra work and it was decided to make this process automated. Code was written that on starting up the program, it automatically changes all the IP addresses in the configuration files to the one of the computer running the code. Should one not want to execute this process, one can simply remove lines 8 to 10 inside "`Shallot_Network.py`".

3.3 Handling errors

When an error occurs within the network not much is done at the moment to automatically correct it. Currently the error is only send towards the node in the network probably causing the error, which would be the one of which an erroneous package was received. A message is printed to the console saying an error occurred and thats all that happens at the moment. Other approaches have been tried to get to correct errors automatically, for example when a message is received with a wrongly formatted header it was tried to automatically send an error message to the sender which in its turns remakes the package and resends it. This however brought a lot of problems since during the time the sender needs to wait to be certain no error occurred on the other side, he would be unable to do other actions.

Another problem was that with relay messages the error might be from an earlier node in the path instead of the node receiving the error message. In this case it would be impossible to correct the message using only the node that receives the error message and it would continuously send an erroneous message and an error message over the link between the node originally sending the wrong message and the one receiving it, blocking both from doing any other action. A possible way to prevent this is to give a message an ID upon receiving it and every node remembering the previous and next node for that message. Then it would be possible to give an error message the same ID and so it could be send all the way back to the original sender. But this method would be in direct contradiction with the original goals of a TOR network which would have optimum security and privacy without placing trust in the relays, every relay only knowing it's own neighbors and not knowing where a message comes from and where it is send to.

4 Example of the network in action

In this paragraph an example of the network in action will be shown. A client, 'Alice', will send a message to another client, 'Bob', by using the topology seen in Figure 1. (Although it is important to notice that the program is written in such a way that Bob can also send a message to Alice or to any other client that is part of the topology in "`topology.ini`".) The images below show what is printed in the console and give a good idea about what is happening.

When `Shallot_Network` is started, the network starts. First the clients are initialized and their information is printed. After this, the same thing is done for the relays. This can be seen in Figure 4.

```
Starting up the Shallot network:

Host alice running at IP, Port: 94.224.151.210, 50001
Host bob running at IP, Port: 94.224.151.210, 50002
relay1 at IP, Port: 94.224.151.210, 51001
relay2 at IP, Port: 94.224.151.210, 51002
relay3 at IP, Port: 94.224.151.210, 51003
relay4 at IP, Port: 94.224.151.210, 51004
relay5 at IP, Port: 94.224.151.210, 51005
relay6 at IP, Port: 94.224.151.210, 51006
```

Figure 4: Network Startup

Now that everything is initialized, the user-interface starts (see Figure 5). At this point, there will be asked who the sender is and who the receiver. In our case these are respectively 'Alice' and 'Bob'. Once this is done the message that needs to be transmitted can be written. Please note that before the actual sending of the message, the application can be exited at anytime by simply typing 'quit'.

```
Who are you? (Type quit to exit application)
alice


Hello alice, who do you want to send a message to? (Type quit to
exit application)
bob


Every message you type is send from alice to bob (Type quit to exit
application)
Hi Bob, this is Alice. How are you doing?
```

Figure 5: User-interface

When this message is written and entered, the network can begin to work, as can be seen on Figure 6. First the path is determined. Then the keys are exchanged such that the encryption can be done. When this encryption is done, the message is sent and Alice prints this to confirm that everything went according to plan. The same is done by every relay where the message passes and Bob also prints to confirm that he has received the message correctly.

```
Computing path...
Path found:
[['94.224.151.210', 51001], ['94.224.151.210', 51004],
['94.224.151.210', 51003], ['94.224.151.210', 50002]]
Keys exchanged
Message relayed
alice sent: Hi Bob, this is Alice. How are you doing?
Message relayed
Message relayed
bob received: Hi Bob, this is Alice. How are you doing?
```

Figure 6: The technical aspect of the network in action

After this confirmation message, a new message can be written. If this is not necessary and you want to exit the application, 'quit' can be typed. As can be seen on Figure 7 the clients and the relays then close, after which you exit the application.

```
bob received: Hi Bob, this is Alice. How are you doing?
quit
closed relay
closed relay
closed relay
closed client bob
closed client alice
closed relay
closed relay
closed relay
```

Figure 7: Network shutdown

References

[1] J-M. Dricot. *Communication Networks : Protocols and Architectures*. ULB, Brussels, 2017. [Online]. Available: `https://owncloud.ulb.ac.be/index.php/s/eg2T0I99DoTLp51`.

[2] Documentation on the cryptography library:
[Online] Available: `https://pypi.python.org/pypi/cryptography`
[Online] Available: `https://cryptography.io/en/latest/`

[3] Documentation on the pycrypto library:
[Online] Available: `https://pypi.python.org/pypi/pycrypto`
[Online] Available: `https://pycryptodome.readthedocs.io/en/latest/`

[4] Schematics made with Gliffy Editor:
[Online] Available: `https://go.gliffy.com/`