Final report

Jan-Cedric Anslinger Jörn von Henning

August 13, 2018

Contents

1 Software Instructions

We have chosen `GoLang` as progamming language in order to implement the `Onion Module`. In the following, dependencies and instructions for running our Module are stated.

1.1 Dependencies

In order to run our `Onion Module` one needs to install the following dependencies first.

- `GoLang`

  GoLang

- `INI`: Open-Source implementation for reading `.INI` config files.

  go get github.com/Thomasdezeeuw/ini

- `DHKX`: Open-Source implementation of the `Diffie-Hellman Key Exchange` algorithm for `GoLang`

  go get github.com/monnand/dhkx

1.2 Installation and Running

In order to run our software one have to install the dependencies stated above first. One of them is providing a 4096 bit `RSA Keypair` in `.pem` format:

```
openssl genpkey -algorithm RSA -out keypair.pem -pkeyopt
rsa_keygen_bits:4096
```

Furthermore, a `config file` is required; a sample `config.ini` file is given in the repository.

```
;Config file
Name = "P2PProject - 61"

[onion]
p2p_port = 3000
p2p_hostname = 127.0.0.1
hostkey = keypair1.pem
```

Figure 1: Sample config file

In order to run the the `Onion Module`, one can either build the project with `go build peer.go` and than run the runnable file with `./NAME -C path/to/config`, or by using the `go run peer.go -C path/to/config` command.

1.3 Testing

For testing purposes, we implementend a `Test case` which is located in the repository `(testCaseOne.go)`. Before running the test case with `go run testCaseOne.go`, one must adapt the available Hosts in the same file as well as starting the peers.

2 Current State

At this point, we have two branches in our repository that represent our solution to the `Onion Module`. The `master branch` contains a runnable version of our implementation without encryption. Unfortunately, some issues concerning the encryption part occured during the development phase.. We intended to encrypt the `Construct Tunnel Message` and `Confirm Tunnel Instruction Message` (stated in the Protocols-Messages section) by using the `public RSA key` of the receiver peer. Since Encryption is a complete new topic for both of us, we were not aware of the `maximum encryption size` of 256 bytes minus padding/header data. Hence, we decided to create `ephemeral keys` between each peer in order to encrypt Data/Messages that need to be interpreted by the receiver peer - data like `Tunnel ID`, `Message Type`, etc. or messages like `Construct Tunnel` and `Confirm Tunnel`

Construction. By now, the encryption branch contains an almost-runnable version of the project including the encryption part. Admittedly, we still have some issues with a few RSA public keys in Tunnel Instruction and Confirm Tunnel Instruction message, which where used to encrypt Tunnel ID and Command Type. Unfortunately, we did not find a proper alternative saving us the trouble of implementing said RSA keys.

Another issue presented itself to us when we tried to obtain the ephemeral key which is required to encrypt/decrypt the data contained in the message. For this, we needed the Tunnel ID and Command Type. The latter one is required to determine the reaction to the received message. Ephemeral keys are stored in a hashmap - this is where the Tunnel ID comes into play: together with the sender's hostkey it extracts the ephemeral key from the hashmap. However, we have been unable to retrieve the ephemeral key until now, henceforth both, Tunnel ID and Command Type, have been encrypted with RSA instead. This enables the user to encrypt and decrypt all the data sent via our Onion Module - no matter what size the message is of.

As displayed before, the sender's hostkey serves a vital function in the retrieval of the ephemeral key. This is where we encountered one of the biggest challenges in this project which we unfortunately failed to solve: Due to the fact that those sender's hostkeys somehow have not been correctly set, we ran into a nil pointer exception. Therefore, we were unable to decrypt the data sent since we could not obtain the required ephemeral key.

Nevertheless when hardcoding the sender's hostkeys for testing purposes, we encountered no further errors and issues and our Onion Module worked as expected.

3 Protocols

A diagramm of the Tunnel Construction Process is shown in Figure 2 - this is the final architecture which is used in the encryption branch, not in the master branch.

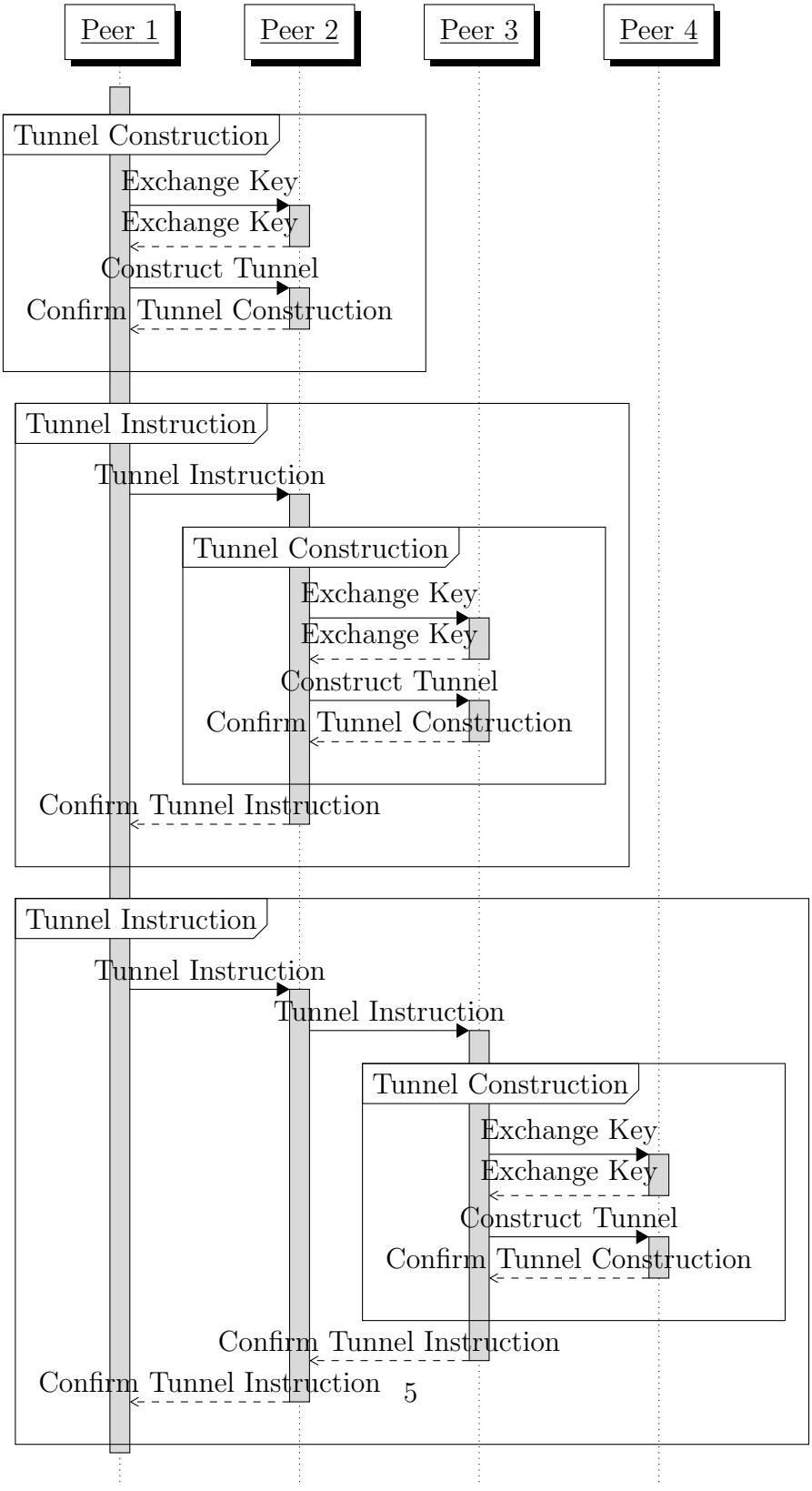

Figure 2: Tunnel Construction Process

3.1 Messages - Master Branch

3.1.1 Construct Tunnel

This message is sent by the onion module in order to construct a tunnel between two peers. This message is identified by `CONSTRUCT TUNNEL` Message Type (567). The version of the network address is specified by the flag V; it is set to 0 for IPv4, and 1 for IPv6. Moreover, this message contains an onion port - the port on which the onion module is listening for incoming UDP messages, a TCP port , a Tunnel ID, the size of the destination hostkey, the destination hostkey, the size of the origin hostkey, the origin hostkey (first's peer hostkey) and the pre-master secret, which is used to generate an ephemeral key between the first peer and the peer that is to be added to the tunnel.

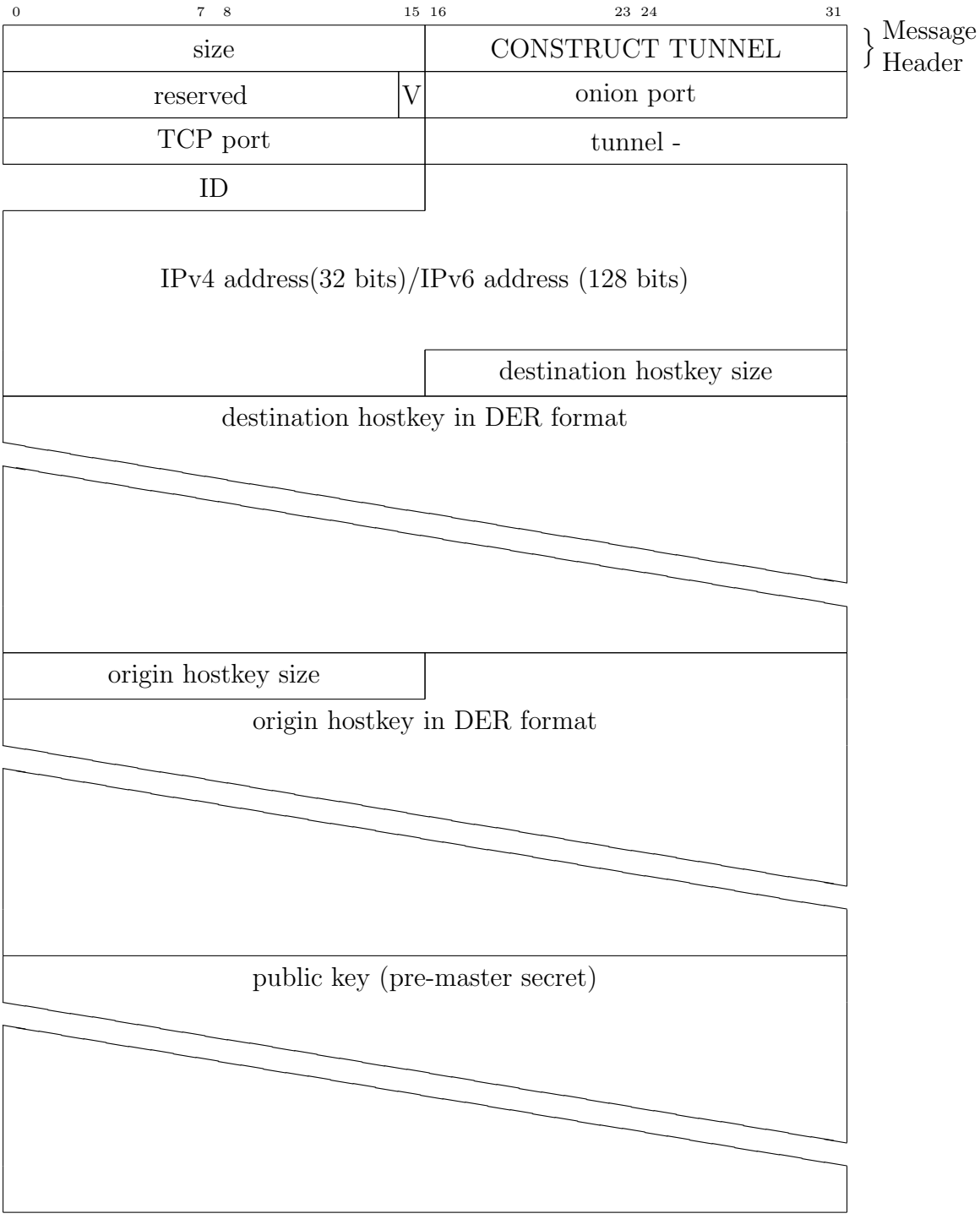

Figure 3: Construct Tunnel Message

3.1.2 Confirm Tunnel Construction

This message is sent by the onion module in order to confirm the requested tunnel construction. This message is identified by `CONFIRM TUNNEL CONSTRUCTION` Message Type (568). It contains the onion port - the port on which the onion module is listening for incoming UDP messages, a tunnel ID, the size of the destination hostkey, the destination hostkey and the pre-master secrect which is used to generate an ephemeral key between the first peer and the peer that is to be added to the tunnel.

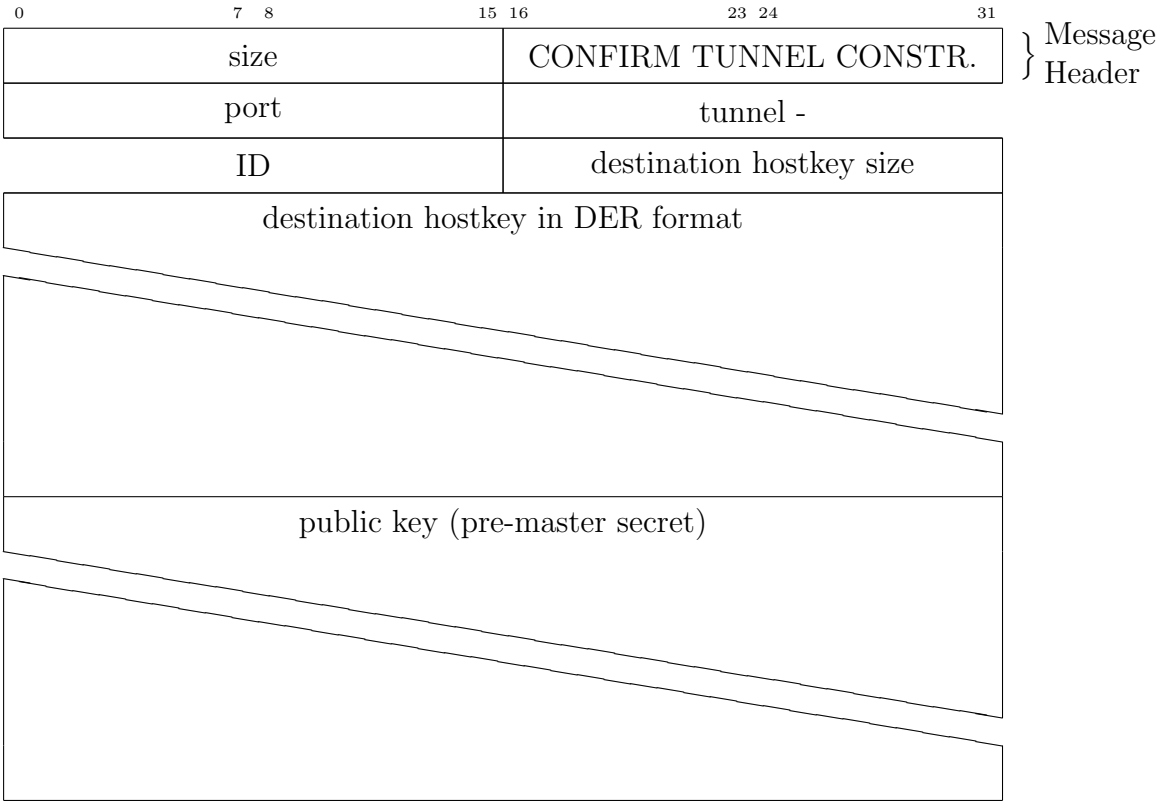

Figure 4: Confirm Tunnel Construction Message

3.1.3 Tunnel Instruction

This message is sent by the onion module in order to request other peers to execute the specified command. This message is identified by `TUNNEL INSTRUCTION` Message Type (569). This message contains the tunnel ID and data.

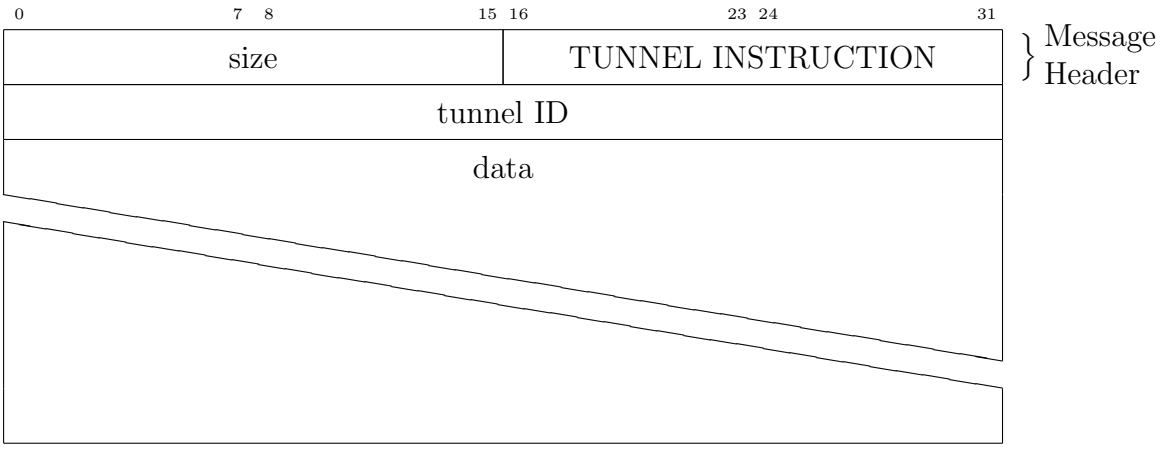

Figure 5: Tunnel Instruction Message

3.1.4 Confirm Tunnel Instruction

This message is sent by the onion module in order to confirm the requested tunnel instruction. This message is identified by `CONFIRM TUNNEL INSTRUCTION` Message Type (570). It contains the tunnel ID and data.

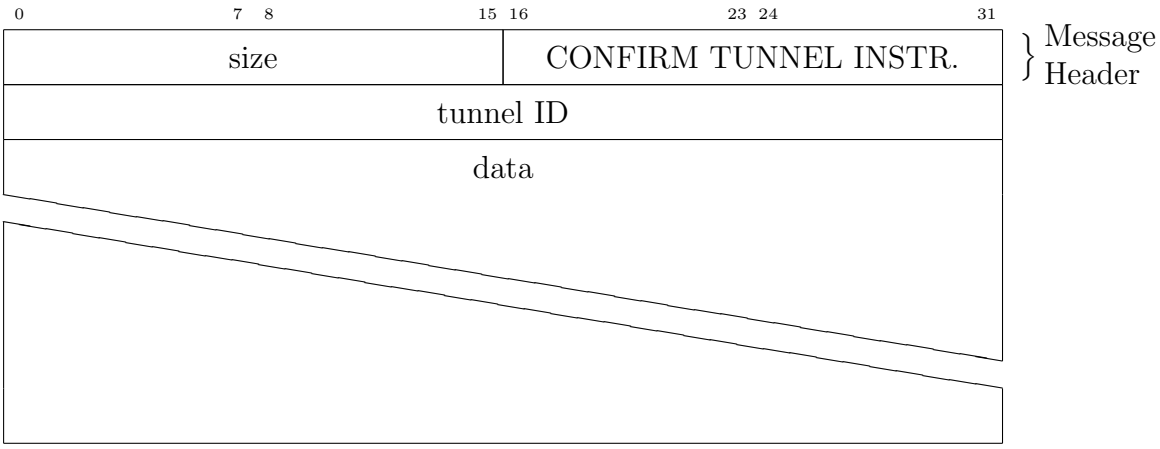

Figure 6: Confirm Tunnel Instruction Message

3.2 Messages - Encryption Branch

3.2.1 Construct Tunnel

This message is sent by the onion module in order to construct a tunnel between two peers. This message is identified by `CONSTRUCT TUNNEL` Message Type (567). The version of the network address is specified by the flag V; it is set to 0 for IPv4, and 1 for IPv6. Moreover, this message contains an onion port - the port the onion module it is listening for incoming UDP messages, a Tunnel ID, the size of the destination hostkey, the destination hostkey, the size of the origin hostkey, the origin hostkey (first's peer hostkey) and the pre-master secret, which is used to generate an ephemeral key between the first peer and the peer that is to be added to the tunnel.

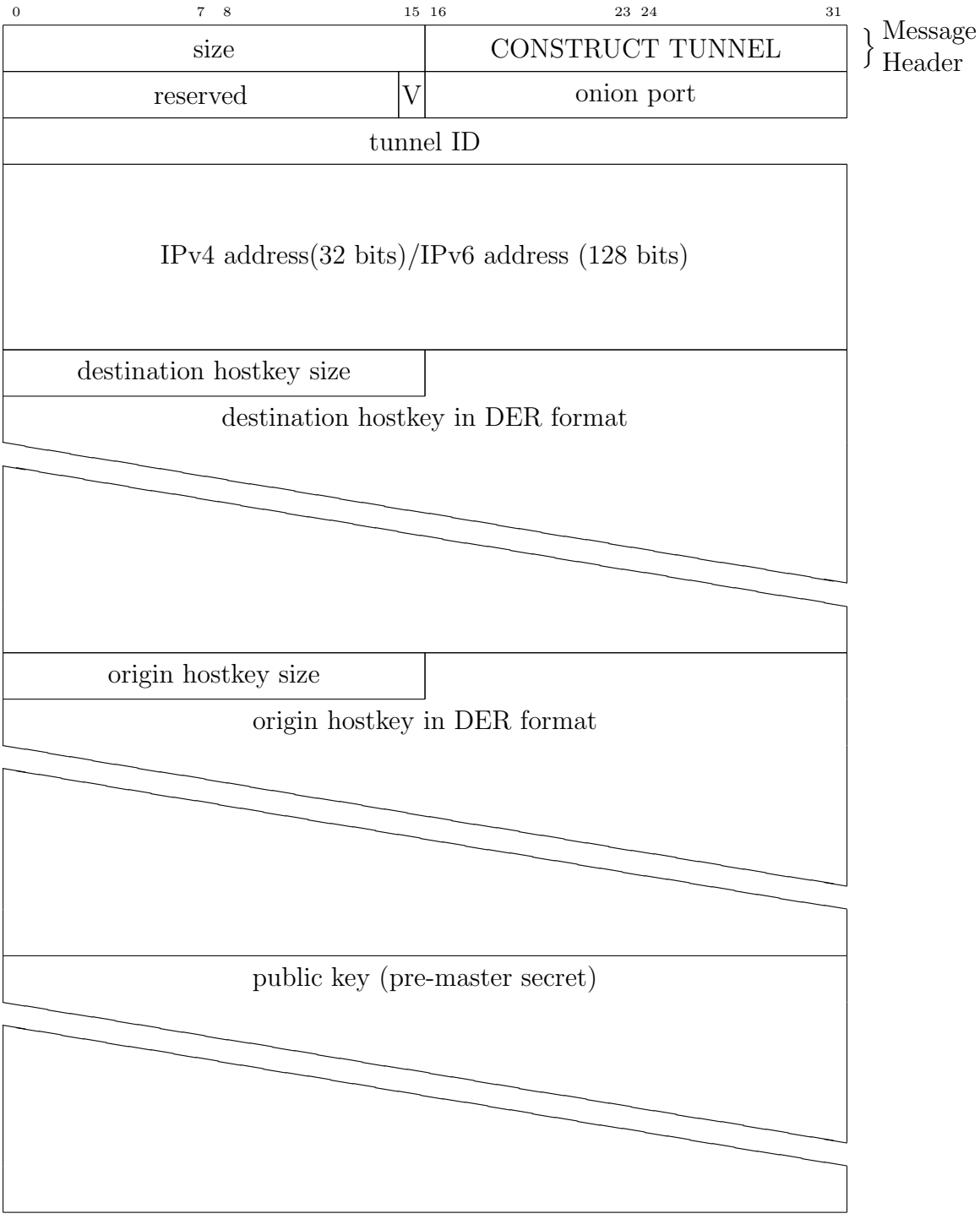

Figure 7: Construct Tunnel Message

3.2.2 Confirm Tunnel Construction

This message is sent by the onion module in order to confirm the requested tunnel construction. This message is identified by `CONFIRM TUNNEL CONSTRUCTION` Message Type (568). It contains a tunnel ID, the size of the destination hostkey, the destination hostkey, an onion port - the port the onion module it is listening for incoming UDP messages and the pre-master secrect, which is used to generate an ephemeral key between the first peer and the peer that is to be added to the tunnel.

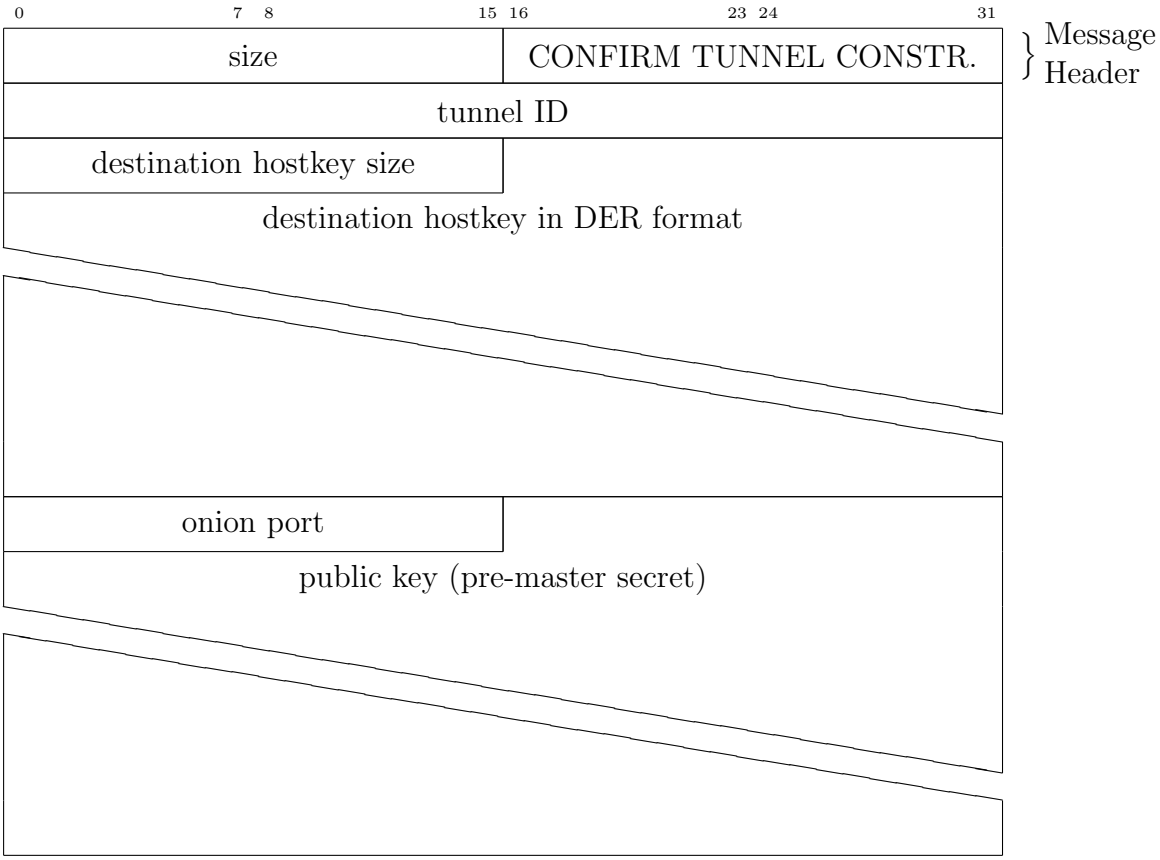

Figure 8: Confirm Tunnel Construction Message

3.2.3 Tunnel Instruction

This message is sent by the onion module in order to request other peers to executed the specified command. This message is identified by `TUNNEL INSTRUCTION` Message Type (569). This message contains the tunnel ID, the size of the origin hostkey, the origin hostkey and data.

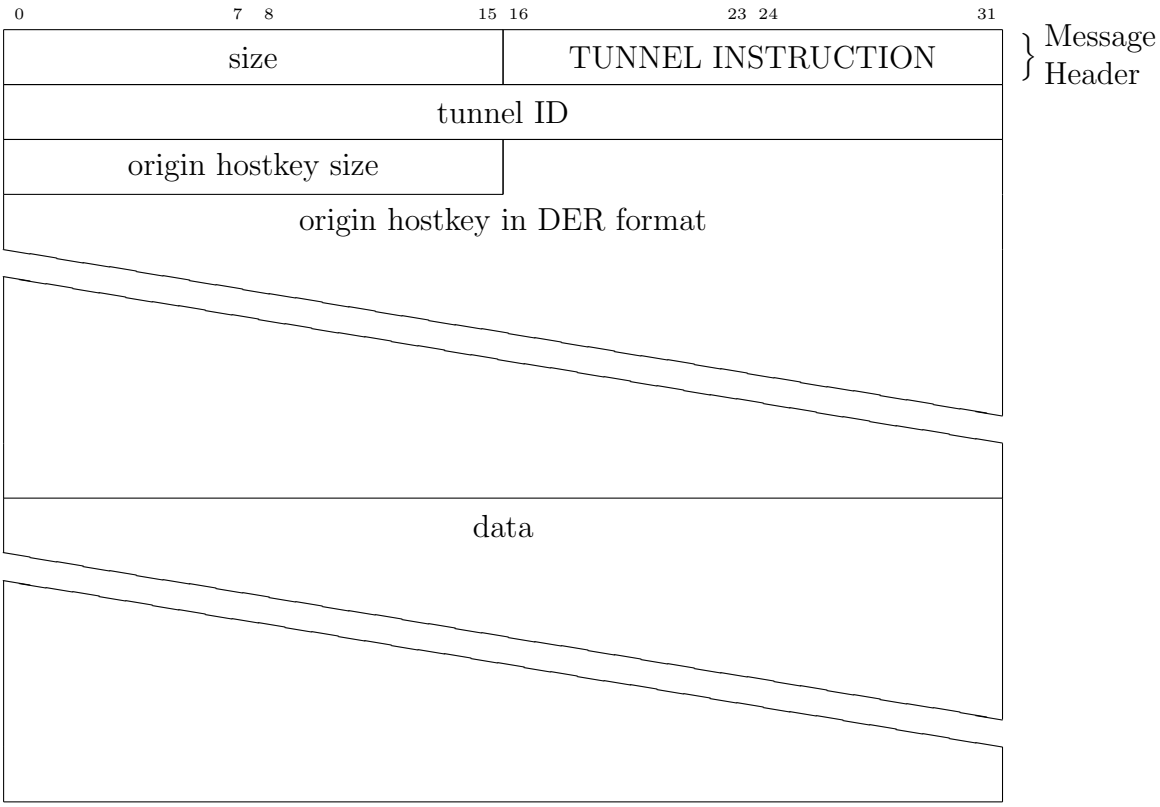

Figure 9: Tunnel Instruction Message

3.2.4 Confirm Tunnel Instruction

This message is sent by the onion module in order to confirm the requested tunnel instruction. This message is identified by `CONFIRM TUNNEL INSTRUCTION` Message Type (570). It contains the tunnel ID and data.

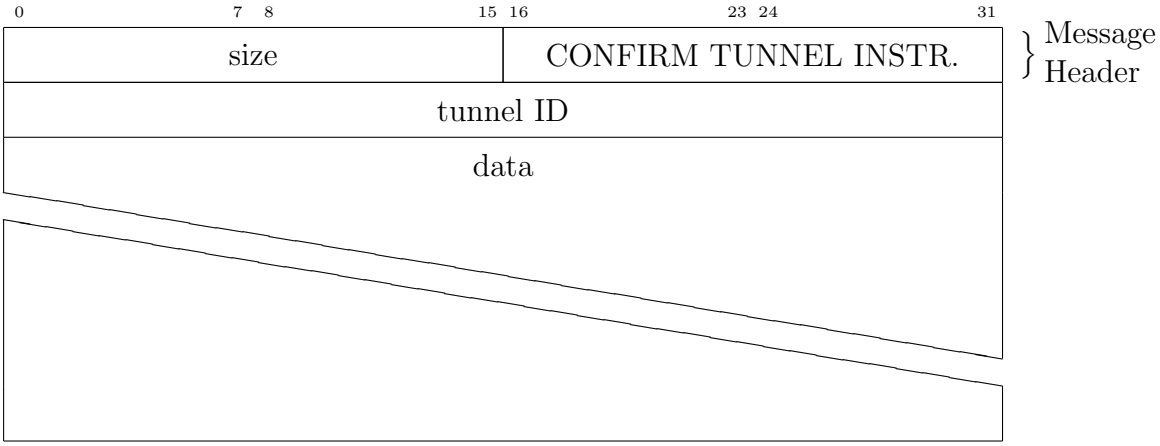

Figure 10: Confirm Tunnel Instruction Message

3.2.5 Key Exchange

This message is sent by the onion module in order to confirm the requested tunnel instruction. This message is identified by `Key Exchange` Message Type (571). It contains a TCP port, the status of the key exchange (represented by the V flag), the size of the (with RSA) encrypted tunnel ID, the tunnel ID, the size of the destination hostkey, the destination hostkey and and the pre-master secrect, which is used to generate an ephemeral key between the peers.

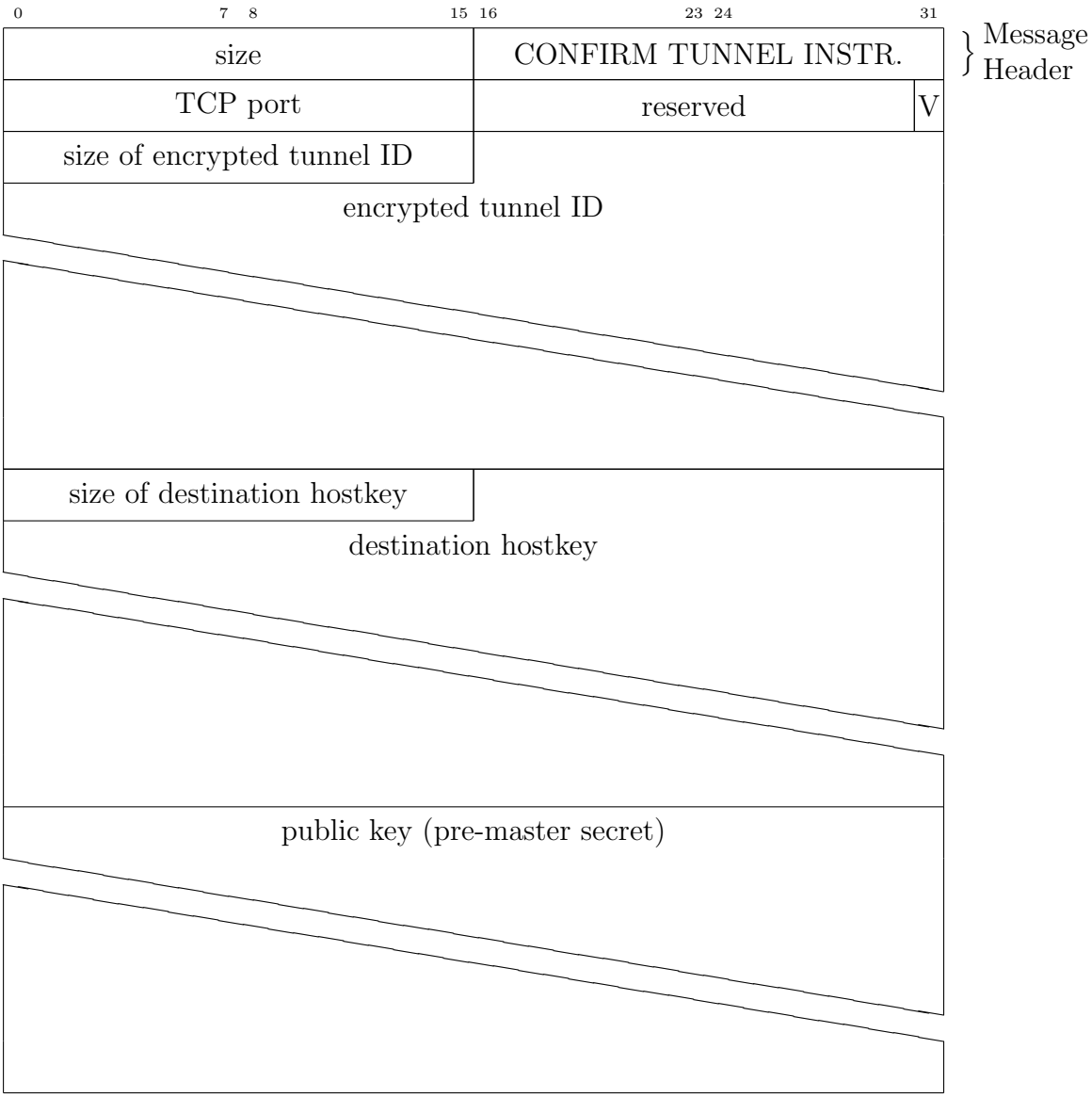

Figure 11: Confirm Tunnel Instruction Message

4 Future Work

To accomplish a further performance enhancement of the `Onion Module`, we are aiming to fix the hostkey issue on the `encryption branch` and test the encryption one more time. As soon as the encryption works we will hence merge the `master branch` and the `encryption branch`.

Furthermore, we still have to accomplish some other issues. To start with, we need to implement the `Onion Tunnel Data` functionality and tether the `RPS` handling; due to the fact that we did not understood the `testing framework`, we implemented our ow testing suite and hence we could not use the `RPS messages`. Nevertheless, the `RPS Peer` message handling is implemented - only the peer requesting part is not implemented at this point.
What is more, we still need to finish the delete method which is called when a destroy message arrives. Also concerning traffic jamming, there are still several steps to do. Especially the UDP traffic jamming needs to be implemented because currently, only TCP jamming is working. In the end, we could still improve the runtime by enhancing the asynchronous performance of the peer. We are already using event loops based on asynchronous channels, but there are still other improvements possible.

5 Workshare

5.1 Jan-Cedric Anslinger

- Development of the general structure of the system
- Configuration of the system (Config Object)
- Basic functionality of the peer
- Event handling
- TCP Listening and message handling
- UDP Listening and message handling

5.2 Jörn von Henning

- TCP Listening
- Messages
- Message Factories
- Handle Messages (TCP Message Controller)
- Encryption

  Key Exchange (DHKX)

  Encryption/Decryption
- Testing

6 Effort

6.1 Jan-Cedric Anslinger

The effort for this subject was quite high. Despite iLab2 where we had to code a bit java but not on such a "deep level", I never had any contact to this kind of programming. Furthermore, I knew the language `go` from some small examples, but never actually coded any huge project in it, so subject and language were both new to me. The first "milestone" was to read in the config file. I never used the .ini format before, but thanks to the plugin I found, it was quite manageable. Afterwards, I had to walk my way through the asynchronous programming in go enabled by go functions and channels. Like that, I was able to design the basic architecture of the main function where TCP and UDP listening where running asynchronously so that they are not blocking all other functions. It was especially a nice experience to see the event loops working and sending the data based on the peer and connections model and forwarding everything autonomously, I've never done something like that before. In the end, when finishing the work at the TCP and UDP Controller after way too many bugs, it was amazing to see everything working as expected, even if we should have done probably a bit more planning concerning the encryption.

6.2 Jörn von Henning

Since distributed network software is a rather new topic for myself, it took quite a bit of time to get along with the architecture (including the messages). First of all, I spent some reading on `transfer protocols`. We came to the conclusion, after having had a discussion whether to use simple `Byte Arrays` or `Protocol Buffers`, that we would like to use `Byte Arrays` in order to spread the messages. Developing the send and receive functionality `(Handle Messages)` was a time consuming activity, because a lot of bugs (some quite weird ones) and errors occured, which needed to be fixed. Nevertheless, the by far most effort and time consuming part was that dealing with encryption: I had to come a long way to understand the mechanisms behind the operation of crypto algorithms. After a lot of reading and even more experimental implementations, I finally acomplished to get the crypto part started. Unfortunately, as we had to learn later on, we should have started with the crypto section instead, since the whole architecture is somehow related to the encryption/decryption functionality. This is why we did not manage to include the encryption in our `master branch`, although it works perfectly fine. To be honest, this is certainly the most important piece of advice for future development work that I was able to take from this project - never to underestimate the importance of encryption, especially during the concept and software architecure development.