TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Stanislav Grebennik 184943IADB

IMPLEMENTATION OF A NETWORK PROTOCOL

Documentation and analysis of a protocol implementation

Supervisor: Olaf M. Maennel
Ph.D.

Author’s declaration of originality

I hereby certify that I am the sole author of this work. All the used materials, references to the literature and the work of others have been referred to. This work has not been presented for examination anywhere else.

Author: Stanislav Grebennik

26.05.2020

Table of Contents

List of Figures

1 Introduction

This documentation is aimed to provide a better overview of the process of implementing the designed network protocol into Python console chat application. The written work describes the structure of a written code, lists advantages and disadvantages of named implementation as well as provides a proof of concept.

2 Protocol implementation analysis

The Python chat application was written as a part of TalTech Network Protocol Design (*ITC8061*) course. More information about the course can be found on the courses page [1] .

The application demonstrates the implementation of a protocol design, which was developed during the course. It uses *libnacl* library to encrypt and decrypt the messages transferred between nodes [2] . The encrypted payload then is encoded with base64, as agreed during the protocol designing phase. All UDP packets sent over the network are encoded using ascii standard.

It has been decided, that the packet's payload should have a maximal length of 100 bytes. Packets with larger payload must be segmented. That said, the python implementation uses a socket buffer size of 256 bytes to accommodate packets header, too. The buffer size can be changed in the application configuration file.

The way nodes are getting information about each other is by sending update packets containing the sender nodes entire routing tree. Upon receiving an update packet it has to be forwarded to all direct neighbours except the one the packet was received from.

Every node is keeping its routing trees version number, which must be incremented upon updating and which must be included in the update packet. The version number can only be five characters long, so the version number will get overflowed. To prevent this situation, the update packet with version number significantly lower than stored version number should be treated as a valid update packet.

In this python application, each known node in a routing topology is represented by a *Node* object. The *Node* object is containing all information about given node, including the cryptographic keys, addresses, and routing information, i.e nodes direct neighbours and a ways we can access the node. The latter is implemented using *Route* objects. Each known node can contains multiple *Route* objects. Each *Route* object contain the

destination of a first hop and entire routes weight. That way, if we want to send a packet to the node without knowing anything about its location or network address, we simply look into nodes *Routes* list, find the one with the smallest weight and send the packet to the first hop destination of that *Route*.

2.1 Features and advantages

2.1.1 No network addresses neede

The application can work and exchange messages with other nodes even without our UDP port or IP address defined in the configuration file. It is possible to configure all of the other known nodes without even specifying their IP addresses, too!

It should be noted that in order to establish communication with somebody, you have to configure the networking address of at least one participant. This means that in order to connect to a larger network of nodes, you only have to know a single nodes address from the desired network and start sending routing update packets to and receiving from this address. That way you will eventually receive a routing update packet from your single defined neighbour containing the topology of an entire network, thus allowing you to start communicating with any node on the network.

2.1.2 Automated discoverability

All nodes can change their statuses and be rediscovered automatically, no application restart is needed. The application can detect any disappeared or unresponsive node and must be able to adjust its routing table accordingly.

2.1.3 Heartbeat

Heartbeat class allows for a very flexible periodic updates and checks.

2.1.4 Code visibility

This protocol implementation is specifically made to be more visual and simple to grasp. There are multiple places of conversions from packet structure in bytes to python lists of strings, messages cache is represented by python dictionary object, as well as other techniques used to make the code relatively simple to read. Some places definitely

need improvement. It should be mentioned that named techniques are not always suitable for production and were made specifically for illustrative purposes.

2.2 Problems and disadvantages

2.2.1 File transfer is in its infancy

Right now the implementation can send only textual files, sending a binary file must be implemented in a future release.

Large files must be cached on the disk during downloading. Right now the entire file is stored in memory and is written to disk only when the receiver accepts it.

2.2.2 Application logging

Logging to a log file should be improved further.

2.2.3 Menu functionality is tiny

Menu functionality must be improved by adding additional features, like drawing the entire route tree, etc.

2.2.4 Automated configuration validation

A wrong application configuration can cause instability of connections to neighbours. Configuration must be checked on startup.

2.2.5 The code is messy in some places

Messenger class is way too big. It will make more sense to split it into smaller classes in future releases.

2.2.6 The speed of discoverabilty

The protocol can suffer from slow speeds of discoverability and updates between nodes.

3 Classes

This chapter describes classes the application is currently made of.

3.1 App

The main class of the program from where the application is started. It launches the application, initializes Config, Messenger, and Heartbeat classes. Those three initialized classes are then passed as objects to other classes where they are needed.

3.2 Config

This class loads information from configuration file about itself as a node in a routing topology, all neighbours defined in the configuration file, and holds discovered information about other nodes in topology. Methods of this class are used everywhere in the application to interact with its configuration.

3.3 Heartbeat

This class contains methods that have to be called each heartbeat. The method *run_routine()* from this class is called from the main App class with the pre-configured interval. Here the app checks for not acknowledged packages and resends them, verifies incoming messages and makes sure the entire message is received, sends periodic routing updates to the neighbours, checks for neighbours connection statuses.

3.4 Message

This class represents an incoming or outgoing message. Each message is stored in memory as a separate object. The main purpose of this class is to be a cache for message segments. The cache can be used in case of outgoing messages for waiting to acknowledge packets from destination, or in case of incoming messages for waiting until the message has been fully received. After that, the message object is deleted from memory by garbage collection process.

3.5 Messenger

This is the main messenger engine of the chat application. All of the packets sending, receiving, routing procedures and decisions are made here.

3.6 Node

This class represents a single node. The class is used for describing all known nodes including ourselves. It stores all information about a node and a list of routes, which tell us how to connect and send a message to this node.

Node can store multiple routes leading to it, which can be used for altering the packets path in case some primary connection is lost. Each node has its own crypto box, which is initialized from nodes public key and which is used for encrypting and decrypting payloads [2] .

The designed protocol specification states that the public key needs to be obtained manually. This offloads the key exchange procedure from the application. All known nodes public keys should be present in a configuration file. If some nodes key is not found in the configuration, no messages exchange can occur between ourselves and the node because there will be no way to encrypt or decrypt the messages.

3.7 Options

This is a menu class for the console application. Every possible menu option should be added here and called from this class.

3.8 Route

Each route to the node consists only from the first hop destination, which is always our direct neighbours node object, and the weight of the entire route.

The protocol doesn't define whether we should store and somehow use the entire routes to the desired destination. So this implementation is storing only the first hop the packet should be sent to.

We don't have to store the entire route because all the other nodes know which node has to be the first hop to forward the packet to. Despite that, we still know the entire topology, which can be obtained from node objects in Config classes *known_nodes* dictionary.

4 Proof of concept

This chapter describes a couple proofs of concept, which are based on testing the interoperability of the application between different devices as well as between different protocol implementations.

4.1 Mobile and PC interoperability

For this test, the python application described above was installed on PC and on an android phone using *Termux* software, which imitates *Linux* console on an android mobile device [2] . The process of installing an application from *Linux* console is explained in Appendix 1.

The mobile phone used only a mobile internet during this test, so there were no local IP address nor UDP port defined in the configuration file. A node with a name "*PC*" was added to configuration as a direct neighbour. This nodes information contained an IP address and a port, which will allow the mobile to start communication first. The full configuration of a mobile node can be seen in Appendix 2.

The PC application configuration has its local IP address and port defined. The node with the name "*MOBILE*" has been added as known node in PC application configuration, with only the public key defined. The full configuration of a PC node can be seen in Appendix 3.

During the test, both applications quickly discovered one another and marked each other as online nodes. Nodes were successfully exchanging messages and files. For PC screen capture see Appendix 4, for mobile screen capture see Appendix 5.

4.2 Interoperability with other groups implementation

For this test my chat application was installed, just as for the first test, on PC with the username "*STAS*" and on an android phone with the username "*MOBILE"*.

The other group members are Dariana Khisteva and Ilker Furkan Kahyalar with their implementation written in python. Their application used the username "*FURKAN*".

Just as before, the mobile phone used only a mobile internet during this test, so there were no IP address nor UDP port of a mobile phone defined in any configuration files. The same is true with the other groups node. Its IP address and UDP port were unknown during the test because the node was located behind the dormitory firewall which was unaccessible and unconfigurable, so it was impossible to do port forwarding.

So, during this test, the only known address was the address of the node "*STAS*", which had nodes "*FURKAN*" and "*MOBILE*" as its direct neighbours.

During the test all three applications were able to quickly discover one another, start sending and flooding periodic routing update packages, and establish communication. For PC screen capture see Appendix 6, for mobile screen capture see Appendix 7.

5 Summary

This piece of work has been made to illustrate the real-world use of a designed protocol. It shows the advantages of the design as well as its possible problems and caveats, leaving plenty of room for further improvements.

References

[1] Network Protocol Design (ITC8061, 6 ECTS). https://courses.cs.ttu.ee/pages/ITC8060 (25.05.2020)

[2] libnacl: Python bindings to NaCl, Creating Sealed Box. https://libnacl.readthedocs.io/en/latest/topics/sealed.html#creating-box (25.05.2020)

[3] Termux – Android terminal emulator and Linux environment app. https://termux.com/ (25.05.2020)

Appendix 1 – Installing the python application

```
0. Start from the project root directory:
cd ~/app

1. Make sure you have python3 installed and pip version is up to
date:
python3 -m pip install --upgrade pip

2. Install virtualenv module, if you don't yet have one:
pip3 install virtualenv

3. Create new venv:
python3 -m venv ./venv

4. Activate venv:
source venv/bin/activate

5. Install requirements:
pip3 install -r requirements.txt

6. Configure the application:
vi app.ini

7. Launch the chat app from project root directry:
python3 chat
```

Figure 1. Installing the python application.

Appendix 2 – Mobiles configuration for first proof of concept

```
[CONFIG]
# App main configuration
log_file = log/app.log
files_dir = files/

# Packets related settings
packet_payload_size_bytes = 100
max_packet_hops = 100
buffer_size = 256
encoding = ascii

# Heartbeat settings
heartbeat_seconds = 5
update_heartbeat_frequency = 1
nodes_online_check_heartbeat_frequency = 3
message_resend_heartbeat_frequency = 1
message_resend_tries = 1
message_receive_check_heartbeat_frequency = 1
message_receive_expired_heartbeats = 4

# Connection info
username = MOBILE
public =
eeb07798e22e1897aae8e992402d506f23739a2560e3591a1ded8aef1e95211f
private =
524867f3ef91696497d03a5615f42d3ea26961a72a4c1c7ae3618686e4128777

[PC]
ip = 90.191.117.228
port = 32002
public =
370a2172b8b5ee1a93a50e7cc6f5db80d120e5190f09f9583b975da57ccc7925
```

Figure 2. Mobiles configuration for first proof of concept.

Appendix 3 – PCs configuration for first proof of concept

```
[CONFIG]
# App main configuration
log_file = log/app.log
files_dir = files/

# Packets related settings
packet_payload_size_bytes = 100
max_packet_hops = 100
buffer_size = 256
encoding = ascii

# Heartbeat settings
heartbeat_seconds = 5
update_heartbeat_frequency = 1
nodes_online_check_heartbeat_frequency = 3
message_resend_heartbeat_frequency = 1
message_resend_tries = 1
message_receive_check_heartbeat_frequency = 1
message_receive_expired_heartbeats = 4

# Connection info
username = PC
ip = 192.168.1.149
port = 32002
limit = 0.0.0.0
public =
370a2172b8b5ee1a93a50e7cc6f5db80d120e5190f09f9583b975da57ccc7925
private =
21249251adc3e0f2ee5229c9c30d3d814f3d23ef1968d4bd12d86a66cebfd334

[MOBILE]
public =
eeb07798e22e1897aae8e992402d506f23739a2560e3591a1ded8aef1e95211f
```

Figure 3. PCs configuration for first proof of concept.

Appendix 4 – PCs screenshot of a first proof of concept

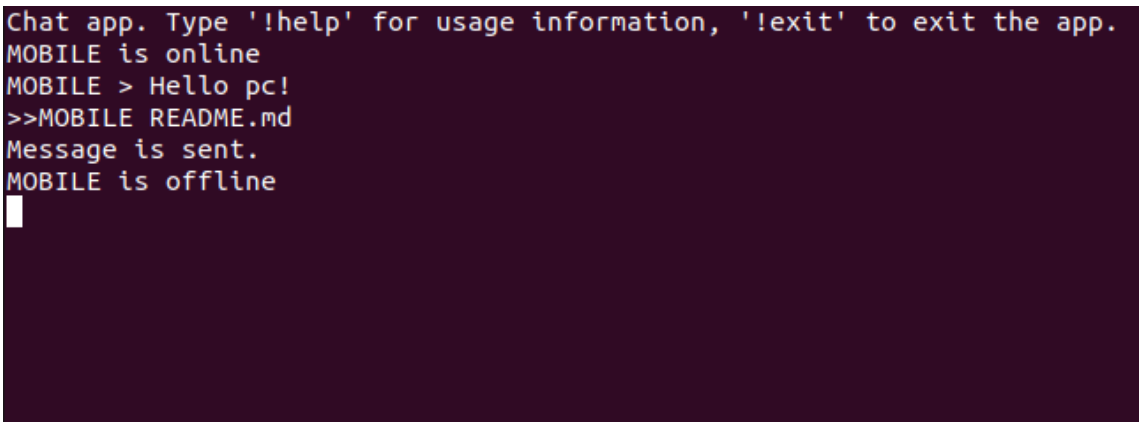

Figure 4. PCs screenshot of a first proof of concept.

Appendix 5 – Mobiles screenshot of a first proof of concept

Figure 5. Mobiles screenshot of a first proof of concept.

Appendix 6 – PCs screenshot of a second proof of concept

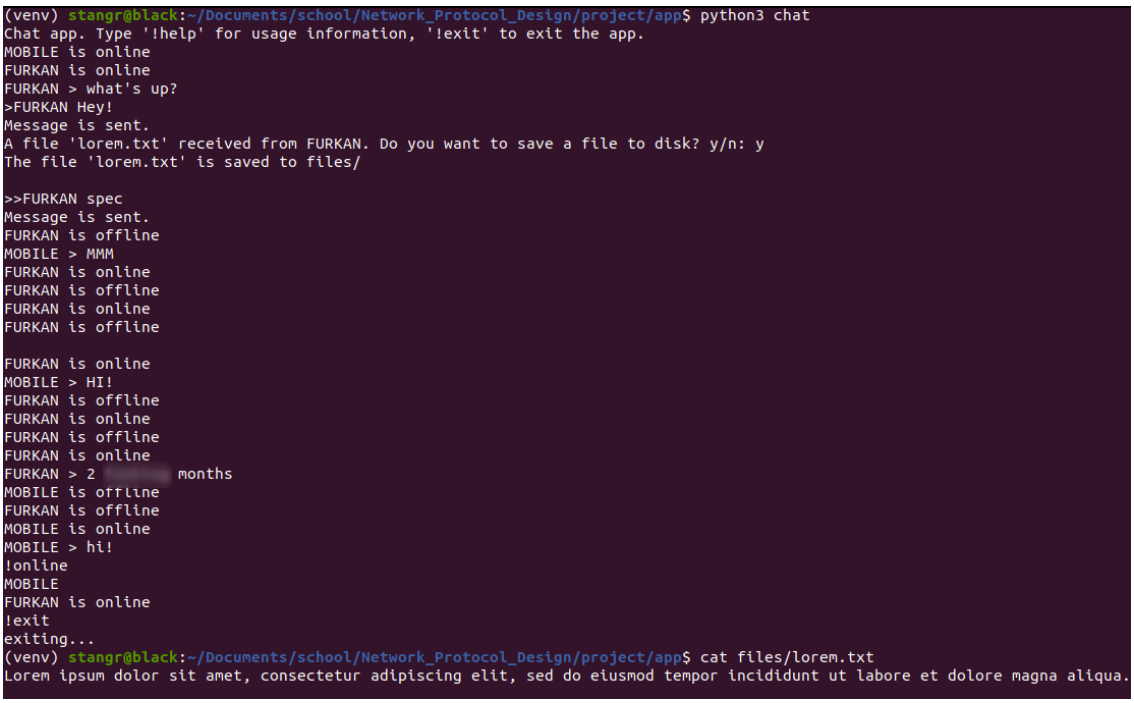

Figure 6. PCs screenshot of a second proof of concept.

Appendix 7 – Mobiles screenshot of a second proof of concept

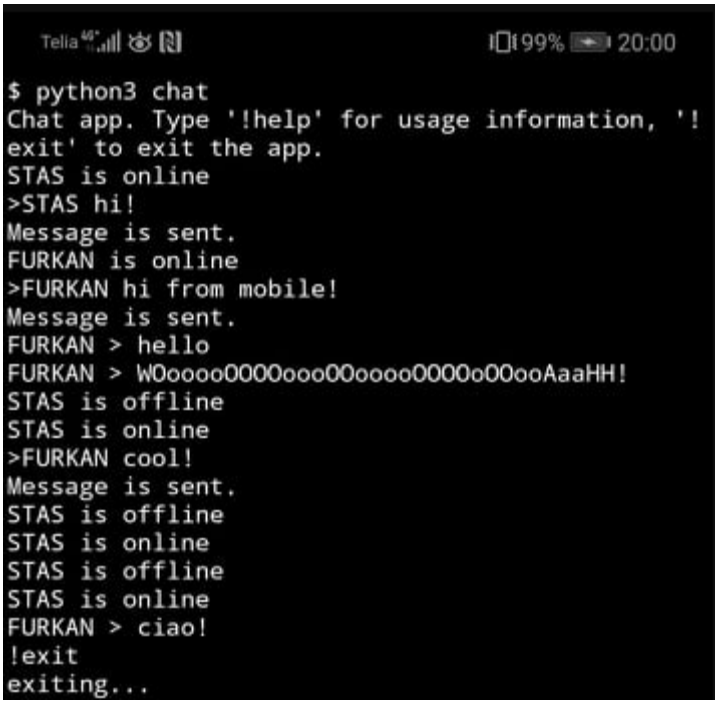

Figure 7. Mobiles screenshot of a second proof of concept.