IaS - Exercise 4

Task 1 - Routing - Are We There Yet?! AWTY?! ...

Basic Setup

We implemented for the basic setup a *peer* and a *control* node.

Peer: The *peer* implementation is divided in multiple files, located in the *peer* folder. There is a *protocol* that defines different message types that are correctly managed by the peer, *routing* is responsible to manage the routing table and to save the ip addresses of the neighbour nodes, *sender* establishes a new connection with the correct receiver node and sends the desired message to it, and *peer* is the "main program" that listens to messages and parse them correctly.

Control: The *control* is similarly implemented, also in his folder. There is again a *protocol* for the possible messages that are managed, *commands* defines the possible commands to enter in the *cli*, that is the "main program" were you can enter the different commands, started by *control*. The *sender* is responsable to store the peer addresses and to send them all type of messages. Last but not least we have the *parser* which reads from a net_config file and parses the correct parts of the file to the correct peer. It is also responsable to send the FINALE keyword, to let the peers start updating their network table.

To run everything just follow the *README* file for the commands. The best way to test the program, is in 2 terminals. One for the controller and one for the peers. On the peers side, you can terminate the main program (launcher.py) by `ctrl-c` and than, to terminate also the running peers as subprocesses, run *terminator* script. On the controller side you can just enter `END` in the cli, the programm will terminate just fine.

On controller side you can enter the following commands:

- `MSG <source_id> <destination_id> <message>`: send message from source to destination
- `NTU <net_config file>`: set a new network

Observation and Discussion

b) The file *task_1_capture.pcapng* is the capture of our program.

Setup:

- Host: started as controller with *net_config_1.nconf*
- Alpine VM: started with some peers, according to *alpine_1.nconf*
- Debian VM: started with some peers, according to *debian_1.nconf*

To see just handshake, ACK, set as filter: `tcp.port != 22 && tcp.flags.ack == 1`.
The first part is just to esclude SSH related traffic, and the second is to filter

message related packets. You can observe a handshake to every peer (port number 5001-5009) comming from the controller (over the bridge, with ip 192.168.122.1).

To see on the other hand the message related trafic, filter settings are: `tcp.port != 22 && (tcp.flags.push == 1 || tcp.flags.fin == 1)`.
NTU goes to every peer, and the last becomes a FIN message (not to be confused with the flag FIN) that lets start the serie of NUs.
No. 397 shows the input from the controller to let peer 2 send a message to peer 6.
No. 407 shows how peer 2 sends directly (see ip addresses) to peer 6 the message "hello".
Then the controller sends a new NTU and everything restarts.
Interesting is also that, thanks to the filter with flag fin set, you can see that after exchanging some packets, the peers close the connection again.

c) To utilize UDP you should make some major changes to ensure that packets arrive, and in the correct order. The most important thing is that every peer becomes the NTU, or the connection will never be established. An other important aspect is that the packets arrive as one unit, or that they are managed to be reassembled correctly by the receiver.

Task 2 - Wireshark - Snooping as a Profession

"I Spy with My Little Eye. . . "

The first point has been covered in the answer in the previous exercise.

Increasing the message beyond the packet limit of 65535 bytes, the only the first part of the message arrives.. In the wireshark interface you can see an entry, marked in red, with the flags "RST and ACK". In some way it is notified that something is wrong. I suppose it is because the connection is already terminated (FIN) when the first packet, with the first part of the message, arrives; and then the other packages can not be delivered and are lost. I think the problem could be solved implementing a delay before closing the connection, then you should see the long message packet split in some minor packets arriving to destination.