UDP-over-TCP

Ihor Mykytenko, s18288

The project implements the functionality described in the task as UDP-over-TCP tunnel. From the perspective of a local process, UDP packets are being sent out and also UDP packets are received in response. However, the transmission of the process data is implemented with the two-way TCP-based communication between the agent, the relay, and the remote host. This document will describe both the normal flow of the program and the implementation details, which allow achieving this kind of behavior.

Normal flow*

*This is just a general description, a practical instruction for testing, the communication protocol and implementation details will be provided below.

1. The remote host, with which the TCP tunnel will be established, should be running before all the other processes. (Otherwise, the relay will not be able to connect to it when provided with corresponding arguments by the agent). So, the remote host mock should be launched and waiting for the relay connection.

2. The relay is launched and starts waiting for incoming agent connection. It cannot yet connect to the remote host, because it’s location is supplied as configuration data from the agent.

3. The agent is launched with the following arguments: IP of the relay, IP of the remote host, and an arbitrary number of ports, on which listeners will be waiting for UDP packets from local processes. Now the agent can connect to the relay and provide it with the data about the remote host, so the relay connects to it as well.

4. Now we can attach a local process to one of the ports on which the agent is listening. The local process

is launched, and it prompts for a port number, to which UDP packets will be sent (we're assuming that the address is localhost, as agent and process should be running on the same machine). After that, the local process mock allows for entering some string, which will be used to illustrate data transfer. This string is passed to agent, then to the relay, then to the remote host, which replies somehow and this reply travels through the chain back to the local process, which receives it as the UDP packet data again.

Practical testing

1. Run the remote process mock. No args.
2. Run the relay. No args.
3. Run the agent. Args: [127.0.0.1 127.0.0.1 12342 14643 14444]. The last three numbers are ports, which could be different or have a different quantity. Say, two or five will do. More ports means more processes can communicate with the remote host.
4. Run the local process mock. No args, but it will prompt for a port number, which should be one of those supplied as an argument to the agent.
5. Enter some string into the local process mock's prompt. See it logged across the other communication participants and decoded into string back again on the remote host side. See the response coming back to the local process as the array of bytes. (The remote host always responds with "`Yes, I've received that.`").

The order of launching things is important here, please follow it.

Communication protocol

When the agent receives a datagram from the user process, it converts this datagram to the following string: PORT:XXXXX&[X, X, X, X, …] where X are numbers. For example, sending the string "Hi!" from the local process, attached to the agent on port 14444 creates the following:

PORT:37591&[72, 105, 33, 0, 0, 0, 0, 0, 0, …]. 37591 is the process own port.
Zeros inevitably appear due to the buffer size.
This data is passed to the remote host, which keeps the port and replaces the data with it's own response. When it comes back to the agent, it sends the UDP datagram with the remote host's response to that port's attached local process mock. Thus, for the process, the communication is purely UDP, but everywhere else it is based on TCP sockets and buffered readers and writers to pass the plain text data around.

Implemented classes

1. Remote host mock. This simulates the remote host, which in fact could be handling data in more complex ways, but in the scope of this tasks it just receives the data, converts it into string, logs and sends a response. It accepts TCP connection from the relay, so has a Socket, BufferedReader and BufferedWriter to talk to it.

2. **Relay.** It opens Sockets with both the remote host and the agent, creates BufferedReader and BufferedWriter for both of them, and then attaches hadlers for incoming traffic from both sides.

3. AgentTrafficHandler. This is a Runnable class with has a reference to agentReader(to catch the incoming data from it), and a remoteHostWriter(to forward that data). It is started as a separate thread along with the RemoteHostTrafficHandler, and it handles the agent - remote host direction of data flow without blocking anything else.

4. RemoteHostTrafficHandler. Pretty much the same as the AgentTraffic handler, but this one gets the data from remoteHostReader and places it into the AgentWriter.

5. **Agent.** Opens a Socket with the relay with parameters given from the args[]. Communicates with the relay over BufferedReader and BufferedWriter based on that socket. Sends parameters of the remote host to the relay this

way, too. Also, runs a LocalProcessListener thread for every port specified on launch.

6. LocalProcessListener. Runnable class, which has a DatagramSocket for receiving UDP datagrams from the local process and an BufferedWriter made upon the Agent's socket to write the data from the local process to the relay. It's run as a separate thread as well as the other Runnable classes here (i.e. all the handlers).

7. RelayTrafficHandler. Runnable class, which receives the response from the remote host via relayReader, decomposes it into port and data, packs the data into the datagram and sends it to the corresponding pots.

8. LocalProcessMock. A simulation of the local process, which gets attached to one of the Agent's listeners, sends UDP datagrams to it and receives UDP datagrams in response.

Difficulties and imperfections

My first thought approaching this project was to just serialize the UDP datagram and pass it around via the ObjectOutputStream and ObjectInputStream. It was truly upsetting to get to know that it is not serializable. Thus, I had to implement the quazi-serialization of data from the UDP packet by myself, which I did with Arrays.toString(byte[]), then passing it to I/O streams, working with plain text. The challenging part was trying to recover the string from the byte array, represented by a string. This can be seen in the remote host mock, where I am trying to do so. For some reason after the fourth or fifth string received it starts overlapping in strange ways, although the relay logs the received data properly. Nevertheless, strings are just the media I've chosen to illustrate the data transfer, so those could just be plain bytes and those should probably work just fine.

Also, I've established the UDP communication between the user process and the agent, while agent and relay communicate with TCP. However, I believe that the UDP-over-TCP functionality is more or less fully achieved for

the following reasons:
1. The user process sends and receives only UDP packets, which it can work with.
2. The packets are forwarded to the final destination via TCP, and sent back the same way.
3. The communication is asynchronous and can handle both directions of data flow due to the use of traffic handlers, listening for IO streams from the sockets in separate threads.

Thanks for reading this all and please contact me in case of an ambiguities arising.