Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU33031 Computer Networks
Assignment #2: Flow Forwarding

Dmitry Kryukov

December 3, 2023

Contents

1 Introduction

This report provides the solution for the given assignment 2 of the Computer Networks module, the main problem of which is to design a flexible protocol, that can allow forwarding packets through different routers and networks from several endpoints without knowing an actual address. There are only two types of actors in the given assignment: endpoints and routers. Addresses of endpoints are presented in the form of special IDs, so even if the machine is replaced, and a new one still retains the old ID, the protocol will work the same way without any troubles. The communication between the elements of the assignment is done via UDP Datagram and sockets. It supports any number of networks, routers, or endpoints, as long as those endpoints have valid IDs, and tries to minimize time usage for the data transfer. Below will be a detailed explanation of how every major part of this project works, including the routing table and header design, their pros and cons, and what is required in order to replicate it.

2 Background

This section is designated for the base of the project, the core of it. It will cover several technologies and tools that were used to complete the said project, such as language, IDE, WSL, and UDP. I will quickly cover the chosen language for this assignment and a few reasons why it was that language in particular. Then I will slightly touch upon the IDE and WSL, the latter is crucial to test the functionality of the complete protocol. At last, there will be a short description of UDP datagrams and sockets.

2.1 Java

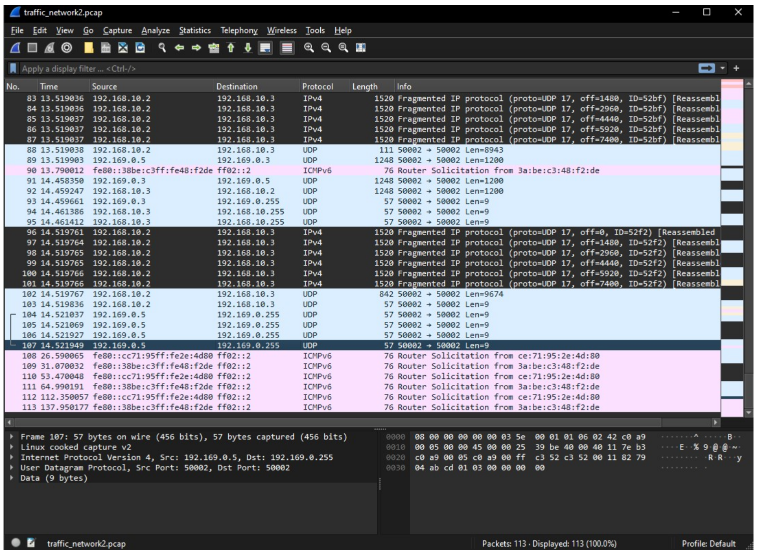
The process of choosing the language for this assignment consisted of several steps. First of all, I should know this language. I was sure of my skills in Java, Python, and C. Second, what will cover all the needed functionality? Because I used Java for the first assignment and did not have to use anything apart from it and several libraries, I already knew that it covers all the needs that might arise. From the assignment description, I could not see anything apart from what was used for the first assignment, except for the routing table, upon which I will touch later on. I mainly used Visual Studio Code to write the protocol. It provides most of the tools that you might need, and if you require anything else, you can always download an extension for it.

2.2 Docker and Wireshark

In order to test the project, several isolated machines with separate networks are required. The easiest way to accomplish this from home, with only one real-life machine, is to use some sort of virtualization platform. This was achieved with the help of Docker. It allows for the fast creation of multiple containers and different networks. Every component in the topology runs on its own container, only connected by networks. In order to speed up the testing process, I used Docker Compose .yml file. It helps with creating additional networks and containers and lets you start the whole project with just one command. To see the traffic, I open a tcpdump file with Wireshark. Traffic for router 2 can be seen in figure 1.
This snapshot shows the traffic on the Wireshark app, with every packet being a separate line.

2.3 Topology

I will provide a topology I used for testing the final version of the assignment in a figure 2 below. Because the task was to make a semi-universal protocol, topology does not matter much here. It will work on any number of networks, endpoints, and routers, as long as endpoints have their own unique IDs.



```
traffic_network2.pcap
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help
Apply a display filter ... <Ctrl-/>
No.  Time        Source                      Destination      Protocol  Length Info
  83 13.519036   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=1480, ID=52bf) [Reassembl
  84 13.519036   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=2960, ID=52bf) [Reassembl
  85 13.519037   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=4440, ID=52bf) [Reassembl
  86 13.519037   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=5920, ID=52bf) [Reassembl
  87 13.519037   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=7400, ID=52bf) [Reassembl
  88 13.519038   192.168.10.2                192.168.10.3     UDP        111 50002 → 50002 Len=8943
  89 13.519903   192.169.0.5                 192.169.0.3      UDP       1248 50002 → 50002 Len=1200
  90 13.790012   fe80::38be:c3ff:fe48:f2de   ff02::2          ICMPv6      76 Router Solicitation from 3a:be:c3:48:f2:de
  91 14.458350   192.169.0.3                 192.169.0.5      UDP       1248 50002 → 50002 Len=1200
  92 14.459247   192.168.10.3                192.168.10.2     UDP       1248 50002 → 50002 Len=1200
  93 14.459661   192.169.0.3                 192.169.0.255    UDP         57 50002 → 50002 Len=9
  94 14.461386   192.168.10.3                192.168.10.255   UDP         57 50002 → 50002 Len=9
  95 14.461412   192.168.10.3                192.168.10.255   UDP         57 50002 → 50002 Len=9
  96 14.519761   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=0, ID=52f2) [Reassembled
  97 14.519764   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=1480, ID=52f2) [Reassembl
  98 14.519765   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=2960, ID=52f2) [Reassembl
  99 14.519765   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=4440, ID=52f2) [Reassembl
 100 14.519766   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=5920, ID=52f2) [Reassembl
 101 14.519766   192.168.10.2                192.168.10.3     IPv4      1520 Fragmented IP protocol (proto=UDP 17, off=7400, ID=52f2) [Reassembl
 102 14.519767   192.168.10.2                192.168.10.3     UDP        842 50002 → 50002 Len=9674
 103 14.519836   192.168.10.2                192.168.10.3     UDP         57 50002 → 50002 Len=9
 104 14.521037   192.169.0.5                 192.169.0.255    UDP         57 50002 → 50002 Len=9
 105 14.521069   192.169.0.5                 192.169.0.255    UDP         57 50002 → 50002 Len=9
 106 14.521927   192.169.0.5                 192.169.0.255    UDP         57 50002 → 50002 Len=9
 107 14.521949   192.169.0.5                 192.169.0.255    UDP         57 50002 → 50002 Len=9
 108 26.590065   fe80::cc71:95ff:fe2e:4d80   ff02::2          ICMPv6      76 Router Solicitation from ce:71:95:2e:4d:80
 109 31.070032   fe80::38be:c3ff:fe48:f2de   ff02::2          ICMPv6      76 Router Solicitation from 3a:be:c3:48:f2:de
 110 53.470048   fe80::cc71:95ff:fe2e:4d80   ff02::2          ICMPv6      76 Router Solicitation from ce:71:95:2e:4d:80
 111 64.990191   fe80::38be:c3ff:fe48:f2de   ff02::2          ICMPv6      76 Router Solicitation from 3a:be:c3:48:f2:de
 112 112.350057  fe80::cc71:95ff:fe2e:4d80   ff02::2          ICMPv6      76 Router Solicitation from ce:71:95:2e:4d:80
 113 137.950177  fe80::38be:c3ff:fe48:f2de   ff02::2          ICMPv6      76 Router Solicitation from 3a:be:c3:48:f2:de

> Frame 107: 57 bytes on wire (456 bits), 57 bytes captured (456 bits)
> Linux cooked capture v2
> Internet Protocol Version 4, Src: 192.169.0.5, Dst: 192.169.0.255
> User Datagram Protocol, Src Port: 50002, Dst Port: 50002
> Data (9 bytes)

0000  08 00 00 00 00 00 03 5e  00 01 01 06 02 42 c0 a9   ·······^ ·····B··
0010  00 05 00 00 45 00 00 25  39 be 40 00 40 11 7e b3   ····E··% 9·@·@·~·
0020  c0 a9 00 05 c0 a9 00 ff  c3 52 c3 52 00 11 82 79   ········ ·R·R···y
0030  04 ab cd 01 03 00 00 00  00                        ········ ·

traffic_network2.pcap          Packets: 113 · Displayed: 113 (100.0%)          Profile: Default
```

Figure 1: This snapshot shows the traffic on the wireshark app, with every packet being a separate line.

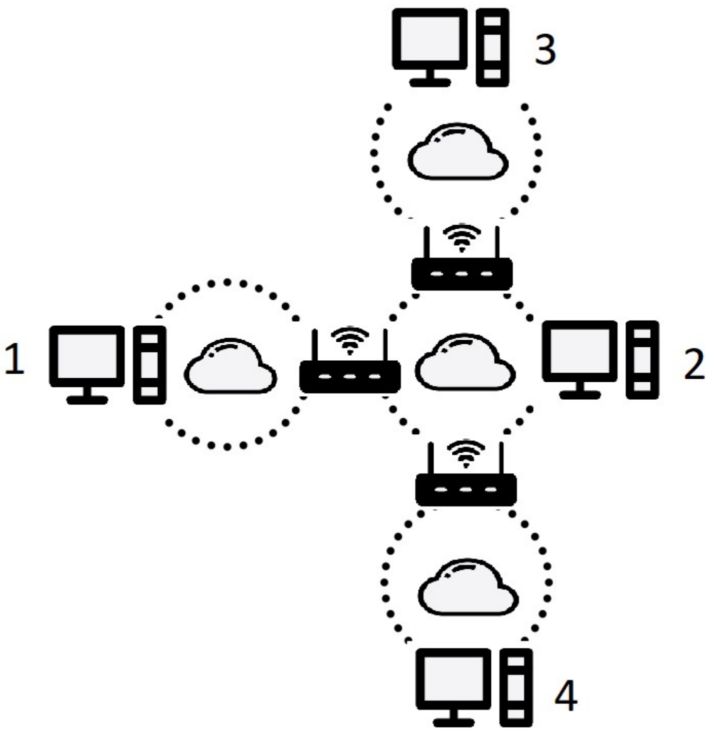


Figure 2: For testing I used a topology with 4 networks, 4 endpoints, and 3 routers. Endpoint 1, which has ID "ABCD0101" tries to send files to endpoint 3, with the ID "ABCD0103"

3 Problem Statement

As mentioned above, the main focus of this assignment is to learn how to properly forward information between different entities, which can be located anywhere in the topology. This consists of making working endpoints and routers. Endpoints serve as clients. They send packets to each other, containing any information, without knowing the actual address, only a specific unique ID, which is predetermined. To reach the destination, those packets have to go through several networks, which are connected by routers. Routers are used as a forwarding tool. They look for the destination in the topology, and, once they find it, they set up a path between two endpoints, which lets them communicate effectively without any additional and redundant strain on those routers. Whenever a path is established, it gets added to the routing table of each router. Those routing tables consist of a previously mentioned ID and an address of the next entity. If an endpoint wants to remove itself from the routing table, for example, if it goes offline, it can ask the connected router for it, which will then ask every other router in the topology to also remove any information about the path to that endpoint. The requirement for this assignment is to use UDP Datagram and Sockets.

4 Design

This section will provide the overall design of the implantation for this assignment. It will quickly cover basic parts of it which will later be properly discussed in the separate section. The main idea is to have two different types of entities. First is an Endpoint, which can send and receive packets. Depending on the type of the packet, which can be determined from the header of it, it will behave differently. When the packet is sent to the first router from the endpoint, if this router does not know the way to the desired destination, it will ask every other router in a different network if they have this address in their routing table. If they do, they send an acknowledgment packet back. If they do not know it, they will continue looking in other networks, until either they find a router who has it or if the final endpoint itself responds to it. Whenever an endpoint sends any packet for the first time, it will be remembered in the routing table of the router it sends the packet to. Endpoints also can request the deletion of their path. For example, if an endpoint goes offline, it might ask a router to delete any information about it in the routing table. After the first router does it, it will ask the same from every router, and so on. The idea is the same as with the path establishment, using the broadcast method. Routers send broadcast packets to each other, which every device in the network will get.

5 Implementation

This section will consist of several parts that were mentioned earlier in the design and a few others. I will quickly cover the header, which is used to determine what to do with the received packet. Unique ID structure, the encoding and decoding process of it. The implementation of the routing table mechanism, which is the main idea behind the project, including acknowledgment packets. Removal from the said routing table on request from the endpoint. The broadcasting method is used to contact every device in the network. The base for the endpoint and router were taken from the previous assignment and the example provided on Blackboard. Modified Broker is used as a router and endpoint is a Producer with some small changes. A simple storage of packets when the next node in the path is still unknown.

5.1 Endpoint

The role of endpoints here is mainly to work as dummy users. All they do is send packets and respond with acknowledgment packets sometimes. Important parts are how the header is encoded and that they have their own IDs.

5.1.1 Header and ID

Every packet is represented as an array of bytes. First, several bytes are used as a header, which helps entities in the system decide what to do when they receive it. I made the header 9 bytes, which is split into 3 parts: Type of the packet, source ID, and final destination ID. Type only takes one byte, while IDs take 4 bytes each.

```
byte[] header1 = new byte[9];
header1[0] = PacketContent.FILESEND;
for (int i = 0; i < 4; i++) {
    header1[i + 1] =
    (byte) Integer.parseInt(myID.substring(i * 2, i * 2 + 2), 16);
}
for (int i = 0; i < 4; i++) {
    header1[i + 5] =
    (byte) Integer.parseInt(dstID.substring(i * 2, i * 2 + 2), 16);
}
// byte[] address = FINAL_ADDRESS.getBytes(StandardCharsets.UTF_8);
byte[] data1 = new byte[header1.length];
System.arraycopy(header1, 0, data1, 0, header1.length);
// System.arraycopy(address, 0, data1, header1.length, address.length);
packet = new DatagramPacket(data1, data1.length);
packet.setSocketAddress(SERVER_PORT);
socket.send(packet);
// wait for 10 seconds
TimeUnit.SECONDS.sleep(5);
stream1live = true;
streams[0] = 1;
startStreaming(0);
```

Listing 1: PacketContent contains all the types of packets needed, such as FILESEND, FINISHSEND, ACKNOWLEDGMENT, etc. myID and dstID are hardcoded strings with unique IDs of endpoints. For test purposes, myID for example is “ABCD0101”, dstID is “ABCD0103”. Streams at the bottom are a part of the previous assignment and are used for testing the functionality of the system. It starts a separate thread that sends frames every second.

```
byte[] data = packet.getData();
StringBuilder srcIDBuilder = new StringBuilder();
StringBuilder dStringBuilder = new StringBuilder();
for (int i = 1; i <= 4; i++) {
    srcIDBuilder.append(String.format("%02X", data[i]));
}
for (int i = 5; i <= 8; i++) {
    dStringBuilder.append(String.format("%02X", data[i]));
}
String srcID = srcIDBuilder.toString();
String dstID = dStringBuilder.toString();
if (!addressMap.containsKey(srcID) && data[0] != PacketContent.FINISHSEND) {
    addressMap.put(srcID,
    new InetAddressAndTime(sourceAddress, System.currentTimeMillis()));
}
```

Listing 2: When a router or endpoint receives a packet, it takes the first 2-9 bytes of it, which is a header part with the ID in it, and with the help of StringBuilder transforms it to the String, which is later checked as a key in the HashMap routing table.

5.1.2 Path request

When the path to the request endpoint is unknown to any router in the system, it will eventually reach the endpoint. When this happens, the endpoint checks if the required ID is equal to that endpoint's own ID. If so, it will send a response.

```
switch (data[0]) {
        case PacketContent.PATHREQUEST:
                // check if the ID is correct
                if (destID.equals(myID)) {
                        dstID = srcID;
                        // send and ACK packet back to the router
                        System.out.println("Received path request");
                        byte[] ackData = new byte[HEADER_LENGTH];
                        ackData[0] = PacketContent.ACKPACKET;
                        System.arraycopy(data, 5, ackData, 1, IDlength);
                        System.arraycopy(data, 1, ackData, 5, IDlength);
                        DatagramPacket responsePacket = new DatagramPacket
             (ackData, ackData.length, routerAddress, DEFAULT_PORT);
                        try {
                                socket.send(responsePacket);
                                System.out.println("Sending ACK packet");
                                // start the stream
                                System.out.println("Starting the stream");
                                stream1live = true;
                                streams[0] = 1;
                                startStreaming(0);
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                } else {
                        System.out.println("Wrong ID");
                }
                break;
```

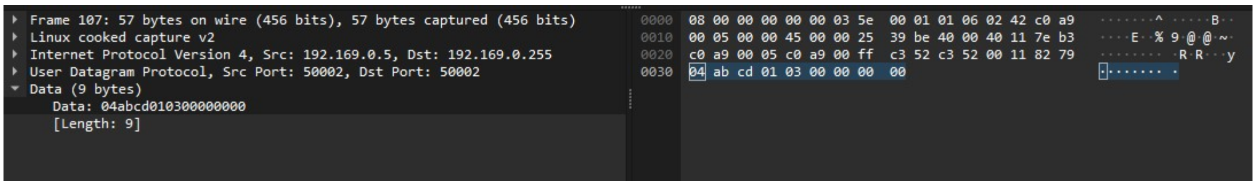
Listing 3: data[] array contains all the bytes from the received packet. If the type byte is 8, which is PATHREQUEST, it will check if the required ID is the same as the myID String variable. Every endpoint has its own ID. Then, if the answer is yes, it will switch the ID bytes of the packet with arraycopy and send an ACK packet back. For testing, I also made it so it will start streaming and sending frames, to check that two-way connection works.

5.1.3 Removal

In my solution, I made it so that after a certain number of frame packets, an endpoint will send a removal request and close its socket.

```
public static void stopStreaming(int a) {
        if (a == 0) {
                stream1live = false;
                streams[0] = 0;
        } else if (a == 1) {
                stream2live = false;
                streams[1] = 0;
        }
        DatagramPacket packet;
        byte[] header = new byte[HEADER_LENGTH];
        header[0] = PacketContent.FINISHSEND;
        for (int i = 0; i < 4; i++) {
                header[i + 1] =
                (byte) Integer.parseInt(myID.substring(i * 2, i * 2 + 2), 16);
        }
        byte[] data = new byte[header.length];
        System.arraycopy(header, 0, data, 0, header.length);
        packet =
        new DatagramPacket(data, data.length, routerAddress, DEFAULT_PORT);
        try {
                socket.send(packet);
                System.out.println("Sending finish packet");
                // close the socket and stop to listen for packets
                socket.close();
        } catch (IOException e) {
                e.printStackTrace();
        }
}
```

Listing 4: This snippet shows the stopStreaming() function, which is called by the stream thread when it wishes to stop. It sets the type of the packet to FINISHSEND and puts the ID of the endpoint in the bytes 2-5. Then it sends the packet and closes the socket.



```
▸ Frame 107: 57 bytes on wire (456 bits), 57 bytes captured (456 bits)
▸ Linux cooked capture v2
▸ Internet Protocol Version 4, Src: 192.169.0.5, Dst: 192.169.0.255
▸ User Datagram Protocol, Src Port: 50002, Dst Port: 50002
▾ Data (9 bytes)
    Data: 04abcd010300000000
    [Length: 9]

0000  08 00 00 00 00 00 03 5e  00 01 01 06 02 42 c0 a9   ·······^ ·····B··
0010  00 05 00 00 45 00 00 25  39 be 40 00 40 11 7e b3   ····E··% 9·@·@·~·
0020  c0 a9 00 05 c0 a9 00 ff  c3 52 c3 52 00 11 82 79   ········ ·R·R···y
0030  04 ab cd 01 03 00 00 00  00                        ········ ·
```

Figure 3: The first byte of the packet is 4 here, which means that an endpoint, in this case, ABCD0103, requested to get removed from the routing table. The next 4 bytes are the ID of that endpoint.

5.2 Router

Routers are the core of this project. They forward the data and are responsible for keeping the system clean from any unnecessary packets. They have routing tables in order to accomplish it. They do not send any packets themselves outside of path searching.

5.2.1 Broadcasting

.

```
// reset the enumeration
networkInterfaces = NetworkInterface.getNetworkInterfaces();
// System.out.println("----------------------------------------");
while (networkInterfaces.hasMoreElements()) {
        // if the network interface is not the one the packet came from,
        // then send the packet to the network interface
        NetworkInterface networkInterface = networkInterfaces.nextElement();
        if(badNetworks.contains
        (networkInterface.getInterfaceAddresses().get(0).getBroadcast()))    {
                continue;
        }
        // reset the enumeration of the getInetAddresses
        System.out.println("Sending a broadcast packet");
        InetAddress broadcastAddress =
        networkInterface.getInterfaceAddresses().get(0).getBroadcast();
        System.out.println("Broadcast address: " + broadcastAddress);
        if (broadcastAddress == null) {
                continue;
        }
        // send the packet
        byte[] broadcastData = new byte[HEADER_LENGTH];
        broadcastData[0] = type;
        System.arraycopy(data, 1, broadcastData, 1, HEADER_LENGTH-1);

        DatagramPacket broadcastPacket =
        new DatagramPacket
        (broadcastData, broadcastData.length, broadcastAddress, DEFAULT_PORT);
        ignoredAddress = ((InetSocketAddress)
        broadcastPacket.getSocketAddress()).getAddress();

        socket.send(broadcastPacket);
}
```

Listing 5: This code snippet provides the main part of the broadcastPacket() method functionality, which is used for sending packets to every device in the network. This function takes in the data to put in the packet, the type of the packet, and the address that the last packet came from. badNetworks are networks that should be ignored, such as the device's own network and the network that the packet came from. NetworkInterface in Java allows to work with networks the current device is connected to.

One of the key points of this assignment was to connect routers and endpoints without using actual addresses. In order to connect endpoints to each other, routers between them need to find a way across multiple networks. Therefore, they are required to ask every router they can if they know the required endpoint. This request packet is being sent as a broadcast packet to a specific broadcast address, which can be obtained from the network interface.

5.2.2 Forwarding

The main purpose of routers is to forward packets between endpoints across different networks. In my solution, As can be seen on listing 6, I take the data from the received packet and check if the path to the desired destination is established.

```
case PacketContent.FILESEND:
// if the path is not established, ask every router for the path
System.out.println("Received file");
// say if the path is established or not
if (!addressMap.containsKey(dstID)) {
        // store the packet until the path is established
        packetStorage.add(data);
        broadcastPacket(data, sourceAddress, PacketContent.PATHREQUEST);
}
// if the path is established,
// forward the packet to the next router in the path
else {
        // if the storage is not empty, forward packets from the storage first
        if(packetStorage.size() > 0) {
                for(byte[] packetData : packetStorage) {
                        DatagramPacket forwardPacket =
                        new DatagramPacket(packetData, packetData.length,
                        addressMap.get(dstID).getAddress(), END_PORT);

                        socket.send(forwardPacket);
                }
                packetStorage.clear();
        }
        System.out.println("Forwarding the packet");

        System.out.println("Forwarding the packet to: " +

        addressMap.get(dstID).getAddress());
        DatagramPacket forwardPacket =
        new DatagramPacket(data, data.length,
        addressMap.get(dstID).getAddress(), END_PORT);

    socket.send(forwardPacket);
}
break;
```

Listing 6: This snippet provides the actual forwarding of packets by the router. If the path is not established, it will store incoming packets in the storage ArrayList. The path is established, it will empty it and forward every other incoming packet to the desired destination by getting the address from the routing table map.

5.2.3 Routing table

The routing table consists of 2 parts. The first one is an ID of the endpoint, which in my implementation is in the form of 4 bytes and is stored as a String. The second is the next node in the path with the associated time, so whenever a packet is sent and bytes 6-9 are converted to String to find a corresponding key value in the HashMap, it can send the packet straight to the next router in the path, instead of broadcasting it again and using the whole system. Listing 7 and listing 8 provide examples of how it is used in my solution.

```
// a hashmap, where the key is the inetaddress
// and the value is inetAddress and the timing
Map<String, InetAddressAndTime> addressMap = new HashMap<>();
ArrayList<byte[]> packetStorage = new ArrayList<>();
```

Listing 7: In the snippet, you can see a Map that has a String as a key and an object with InetAddress and Time in it. The next line after it is the storage in the form of an ArrayList for packets when the next node is still unknown. When a path is established, they will be quickly sent and clear the list.

```
if (!addressMap.containsKey(srcID) && data[0] != PacketContent.FINISHSEND) {
        addressMap.put(srcID, new InetAddressAndTime(sourceAddress,
        System.currentTimeMillis()));
}


...


addressMap.remove(srcID);


...


case PacketContent.PATHREQUEST:

if (addressMap.containsKey(dstID)) {
        byte[] response = new byte[HEADER_LENGTH];
        response[0] = PacketContent.ACKPACKET;
        System.arraycopy(data, 1, response, 5, IDlength);
        System.arraycopy(data, 5, response, 1, IDlength);
        DatagramPacket responsePacket =
        new DatagramPacket(response, response.length,
                        addressMap.get(srcID).getAddress(),
                        DEFAULT_PORT);
        socket.send(responsePacket);
        return;
}
```

Listing 8: These are examples of how routers work with the mentioned above table. The first one is a check at the start of onReceipt() function. Whenever a packet is received, the router looks up the source of the packet in the map. If it can not find a matching key, it adds a new element to the map, which contains the source address and the time. The second is how information about the router is deleted from this map. The third is a forwarding part of the path-searching process. If the router knows the requested endpoint, it will send an ACKNOWLEDGMENT packet back, with IDs switched, so routers on the way back can also put the path in their tables.

6 Discussion

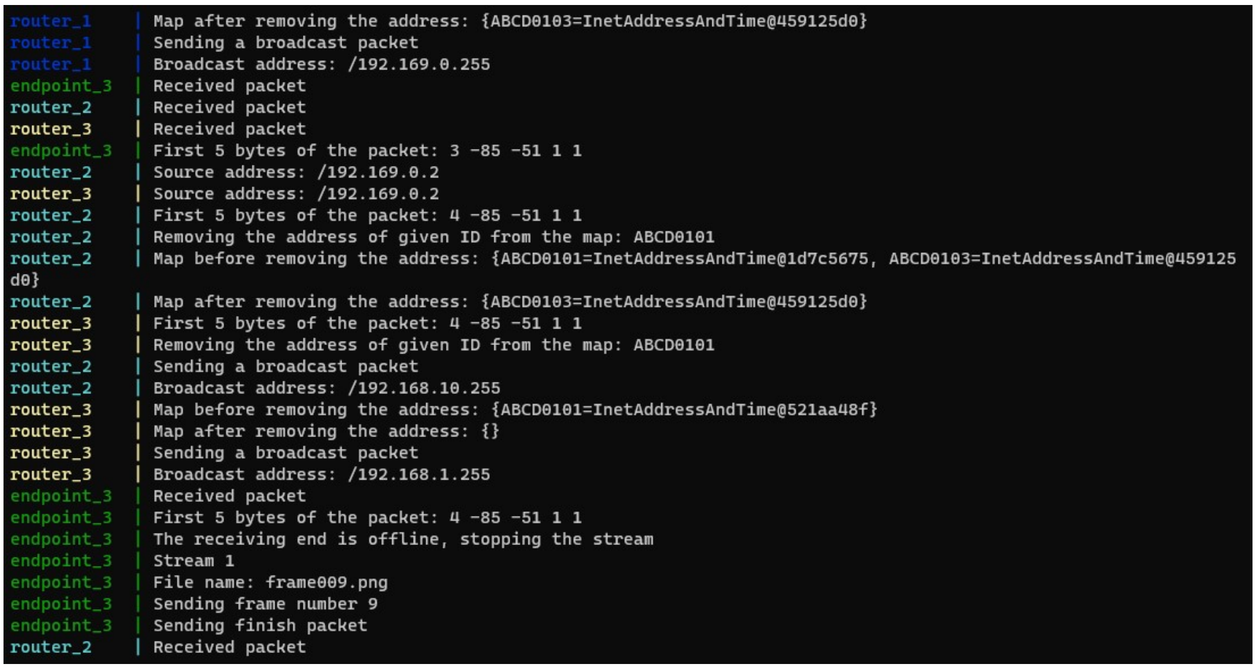
The solution that I provided covers all the requirements from the assignment description. It can be used on any number of entities and networks, which makes it flexible. In the previous assignment, I used ArrayLists in order to store IDs, which made it pretty slow. In this one, I used a hashmap instead, so I do not need to go through the whole list every time I request an address from it. There are still some downsides to it. Endpoints need to know where to send the first packet in the beginning, which in my solution is given to them. This can be changed to the broadcast method if needed, but then if there is more than one router in the network they will all receive the request and build multiple paths, which might not be desirable. Another downside is that I did not consider any shortest path algorithms, so the fastest router to respond will be the one that the path goes through. I also did not consider any loops in networks in my solution, but whenever a path is found it will be locked in and stop the search. In order to send and receive broadcast packets, the device should have broadcast enabled. This can be accomplished using socket.setBroadcast(true). By default, in my solution, all devices have it enabled.

7 Summary

This report covers my solution for the flow forwarding assignment, what was used to write the protocol, and how it was tested. Every crucial part of the protocol that is required to replicate it is explained, with code snippets provided. The solution functions as intended and follows the requirements of the assignment. HashMap routing table and header work well together, providing a necessary forwarding for any topology. Endpoints which are several networks apart are able to communicate with each other through routers, which try to minimize any unnecessary strain on the system.

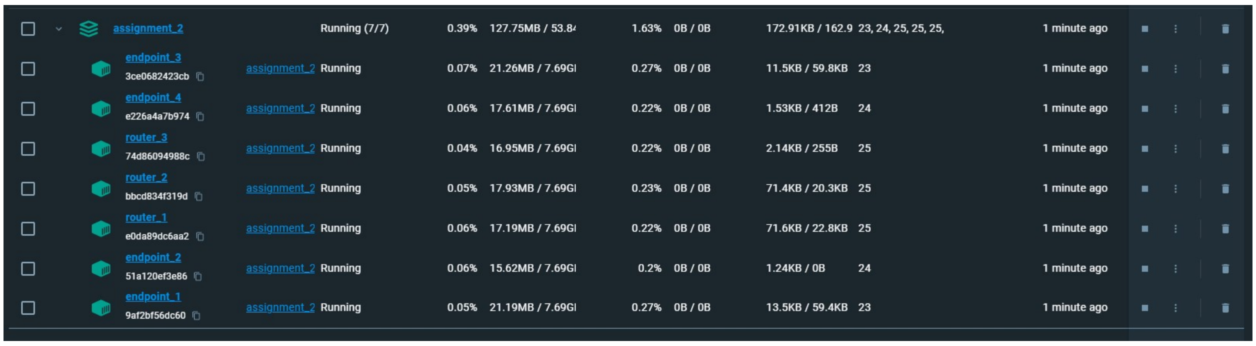
8 Reflection

At first glance, this assignment looked almost the same as the first one, with some minor tweaking. The routing table part seemed strange at first, but when I grasped the idea behind the assignment, I immediately figured out how to do it. The complete solution was almost ready for the first submission. I spent around 35 hours overall on this project, not including the report. It took a bit more time than I thought to test it because starting several containers takes some time. Overall, I am satisfied with how everything turned out and how much time I spent on it. Figures 4 and 5 show what was used in order to debug the program.



```
router_1     | Map after removing the address: {ABCD0103=InetAddressAndTime@459125d0}
router_1     | Sending a broadcast packet
router_1     | Broadcast address: /192.169.0.255
endpoint_3   | Received packet
router_2     | Received packet
router_3     | Received packet
endpoint_3   | First 5 bytes of the packet: 3 -85 -51 1 1
router_2     | Source address: /192.169.0.2
router_3     | Source address: /192.169.0.2
router_2     | First 5 bytes of the packet: 4 -85 -51 1 1
router_2     | Removing the address of given ID from the map: ABCD0101
router_2     | Map before removing the address: {ABCD0101=InetAddressAndTime@1d7c5675, ABCD0103=InetAddressAndTime@459125
d0}
router_2     | Map after removing the address: {ABCD0103=InetAddressAndTime@459125d0}
router_3     | First 5 bytes of the packet: 4 -85 -51 1 1
router_3     | Removing the address of given ID from the map: ABCD0101
router_2     | Sending a broadcast packet
router_2     | Broadcast address: /192.168.10.255
router_3     | Map before removing the address: {ABCD0101=InetAddressAndTime@521aa48f}
router_3     | Map after removing the address: {}
router_3     | Sending a broadcast packet
router_3     | Broadcast address: /192.168.1.255
endpoint_3   | Received packet
endpoint_3   | First 5 bytes of the packet: 4 -85 -51 1 1
endpoint_3   | The receiving end is offline, stopping the stream
endpoint_3   | Stream 1
endpoint_3   | File name: frame009.png
endpoint_3   | Sending frame number 9
endpoint_3   | Sending finish packet
router_2     | Received packet
```

Figure 4: This snapshot shows how devices communicate with each other with the help of several print statements in the code. This helps with debugging the program.



assignment_2		Running (7/7)	0.39%	127.75MB / 53.8	1.63%	0B / 0B	172.91KB / 162.9	23, 24, 25, 25, 25,	1 minute ago
endpoint_3 3ce0682423cb	assignment_2	Running	0.07%	21.26MB / 7.69GI	0.27%	0B / 0B	11.5KB / 59.8KB	23	1 minute ago
endpoint_4 e226a4a7b974	assignment_2	Running	0.06%	17.61MB / 7.69GI	0.22%	0B / 0B	1.53KB / 412B	24	1 minute ago
router_3 74d86094988c	assignment_2	Running	0.04%	16.95MB / 7.69GI	0.22%	0B / 0B	2.14KB / 255B	25	1 minute ago
router_2 bbcd834f319d	assignment_2	Running	0.05%	17.93MB / 7.69GI	0.23%	0B / 0B	71.4KB / 20.3KB	25	1 minute ago
router_1 e0da89dc6aa2	assignment_2	Running	0.06%	17.19MB / 7.69GI	0.22%	0B / 0B	71.6KB / 22.8KB	25	1 minute ago
endpoint_2 51a120ef3e86	assignment_2	Running	0.06%	15.62MB / 7.69GI	0.2%	0B / 0B	1.24KB / 0B	24	1 minute ago
endpoint_1 9af2bf56dc60	assignment_2	Running	0.05%	21.19MB / 7.69GI	0.27%	0B / 0B	13.5KB / 59.4KB	23	1 minute ago

Figure 5: A snapshot of docker containers used for testing