



CSU33031 Computer Networks

Assignment #2: Flow Forwarding

Louie Somers Std# 21363644

December 1, 2023

Contents

1	Introduction	1
2	Background	2
2.1	Flow Forwarding.....	2
2.2	Topology.....	2
2.3	Technical Background.....	2
2.4	Closely-Related Projects.....	3
2.2.1	Summary.....	3
3	Problem Statement	3
4	Design	3
4.1	Header.....	2
4.2	Flow Control.....	7
5	Implementation	9
5.1	Node.java.....	9
5.2	Router.java.....	10
5.3	Endpoint.java.....	12
6	Discussion	15
7	Summary	16
8	Reflection	16

1 Introduction

The focus of this assignment is designing a reactive routing protocol. This requires first establishing routing information through broadcasts, and then using that routing information stored in forwarding tables to send packets between endpoints without broadcasting. For my implementation, I decided to have endpoints take user input through the console and have router's take no user. Another problem this assignment addresses is

the storing of forwarding information for an indefinite period of time. To solve this I gave lifetime to entries as well as giving a command to endpoints to remove their data from all routers.

2 Background

This section will describe the theory underlying the assignment. It is important to understand the basis on which my solution is built. I will discuss some general information regarding flow forwarding, the topology of my solution, the technical background of some of the software I used and some closely-related projects.

2.1 Flow Forwarding

The internet is essentially a network of networks. Flow forwarding is a method of transporting messages across networks using forwarding tables. What I am implementing in this assignment is a flow forwarding protocol which reacts to new endpoints being discovered through broadcasts, and storing the information received from those broadcasts in order to directly send messages in the future.

2.2 Topology

Below is an example topology my solution works on. My solution could work larger topologies provided the relevant docker setup and the correct networks were placed in Node.java to be accessed by the endpoints and routers.

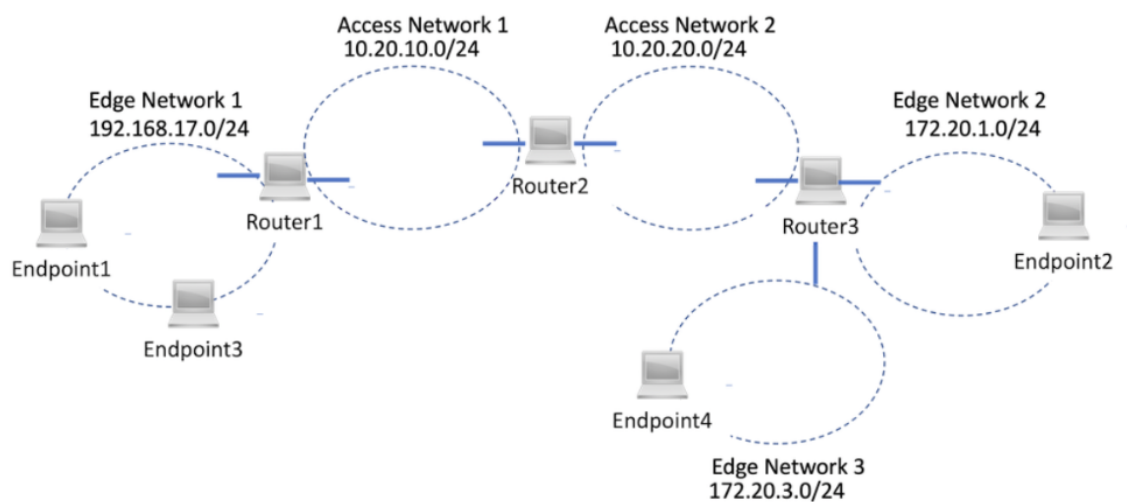


Figure 1 This is an example of a network topology this system that could be implemented using my code and it what is built from the bash script in my submission. There are 4 sub networks, 4 endpoints and 3 routers. Packets can be sent between any of the endpoints including ones within the same network.

2.3 Technical Background

This section briefly summarizes technologies that formed the basis of a project.

- I choose Java as a programming language for this assignment. I found it particularly useful as it is an object-oriented language which allowed me to have an abstract superclass Node.java for methods and global variables required by both endpoints and routers, and for the java.net library which allows for socket programming.
 - I used Docker to set up containers for nodes and facilitate networking between them in a virtual environment.
 - I used Visual Studio Code and my Integrated Development Environment(IDE) because of my familiarity with it and its useful docker integration.
-
- I used Wireshark to view captured network traffic in the form of pcap files.

2.4 Closely-Related Projects

This section will examine the Address Resolution Protocol(ARP) as an existing option to this problem today, as well as the previous Publish-Subscribe Protocol I made and how it was used as basis for the structure of this assignment.

Aspect 1: ARP ->

This is a protocol used to find the media access control(MAC) address of a given internet protocol(IP) address. When a new network element joins a network, it will receive a unique IP address to use for identification and communication. Packets of data arrive at a gateway, destined for a particular host machine. The gateway, or the piece of hardware on a network that allows data to flow from one network to another, asks the ARP program to find a MAC address that matches the IP address. The ARP cache keeps a list of each IP address and its matching MAC address.

Aspect 2: Publish-Subscribe Protocol ->

The structure of this project was based on my Publish-Subscribe Protocol, with the abstract superclass Node.java and the network elements who were extended this. The basis of the routers comes from Broker.java and the basis for endpoints comes from Producer.java. For packets I removed the sending of images and instead just used Strings as the data.

2.2.1 Summary

While the ARP connects an IP address to a MAC address, in this project I connect the IP address of the endpoints a custom 4 byte hexadecimal decided by the endpoint via user input. The routers store these pairs in their forwarding tables when receiving a packet from a source not already in their table.

3 Problem Statement

This assignment focuses on implementing a routing protocol to learn about flow forwarding development and the routing information that is stored in forwarding tables. The problem faced in this assignment is reduce the communication necessary to establish the necessary routing information from endpoints not listed in the routers' forwarding tables.

4 Design

In this section I will provide a description of my header and give an example of the flow of packets in my solution.

4.1 Header

Header = [Packet Type|Dest ID|Source ID|Router no.]

- **Packet Type -> 1 byte**
- **Destination ID -> 4 byte**
- **Source ID -> 4 byte**
- **Router number -> 1 byte**

I included my own 10 byte header. This is attached at the start of a packet. The rest of the packet, or the payload, is a String message. As discussed in 2. Background, the 4-byte Source ID is what is mapped to the IP address of the next hop to that ID in the router's forwarding table. The Destination ID is initially checked by the router to see if it is in its forwarding table. If it is it sends the packet to next hop, else it broadcasts across its networks. It is also used by endpoints to check if the packet is meant for it. The router number is simply used by router's to ignore broadcasts from themselves. The packet types can be the following:

No.	Time	Source	Destination	Protocol	Length	Info
134	21.255114	172.20.20.3	172.20.20.255	UDP	1408	51000 → 51000 Len=65000
178	21.265786	172.20.20.4	172.20.20.3	UDP	1408	51000 → 51000 Len=65000
224	41.682879	172.20.20.2	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
268	41.731012	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
312	41.731826	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
358	41.845470	172.20.17.2	172.20.17.3	UDP	1408	51000 → 51000 Len=65000
402	41.862821	172.20.20.4	172.20.20.2	UDP	1408	51000 → 51000 Len=65000
450	52.623334	172.20.20.3	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
494	52.641997	172.20.20.4	172.20.20.2	UDP	1408	51000 → 51000 Len=65000
538	52.679377	172.20.20.2	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
582	52.681493	172.20.20.4	172.20.20.3	UDP	1408	51000 → 51000 Len=65000
630	72.198881	172.20.17.2	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
674	72.206439	172.20.20.4	172.20.20.3	UDP	1408	51000 → 51000 Len=65000
718	72.233492	172.20.20.3	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
762	72.240180	172.20.17.3	172.20.17.2	UDP	1408	51000 → 51000 Len=65000
808	81.057885	172.20.20.2	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
852	81.064546	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
896	81.064999	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
940	84.937704	172.20.20.3	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
984	84.943525	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
1028	84.944119	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
1074	84.944945	172.20.20.3	172.20.20.255	UDP	1408	51000 → 51000 Len=65000

> Frame 268: 1408 bytes on wire (11264 bits), 1408 bytes captured (11264 bits) on interface 0	0000	c7 38 c7 38 fd f0 01 d6 01 dd dd dd dd aa bb cc	8:8...
> Linux cooked capture v2	0010	dd 01 74 68 69 73 20 69 73 20 31 00 00 00 00 00	...this is 1...
> Internet Protocol Version 4, Src: 172.20.17.3, Dst: 172.20.17.255	0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
> User Datagram Protocol, Src Port: 51000, Dst Port: 51000	0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
Source Port: 51000	0040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
Destination Port: 51000	0050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
Length: 65008	0060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
Checksum: 0x01d6 [unverified]	0070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
[Checksum Status: Unverified]	0080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
[Stream index: 4]	0090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
> [Timestamps]	00a0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
UDP payload (65000 bytes)	00b0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
> Data (65000 bytes)	00c0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
Data [truncated]: 01dddddddaabccdd017468697320697320310000000000000000	00d0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
[Length: 65000]	00e0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	00f0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	0100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	0110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	0120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	0130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	0140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	0150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	0160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	0170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	0180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	0190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	01a0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Figure 3 Example of a packet of type MESSAGE, first byte in header underlined in yellow(01), captured in Wireshark. This is the Router 1 broadcasting to the network it didn't receive the packet from, after receiving a packet from Endpoint 1, with ID AABBBCCDD, to Endpoint 4, with has ID DDDDDDDDD. Byte 1-4 in the header is the destination ID, which is set to the ID of Endpoint 4. Byte 5-8 in the header is the source ID, which is set to the ID of Endpoint 1. The router number, or the 9th byte in the header is 01 as it is Router 1 broadcasting. The payload, underlined in blue, holds the encoding string, "this is 1".

No.	Time	Source	Destination	Protocol	Length	Info
134	21.255114	172.20.20.3	172.20.20.255	UDP	1408	51000 → 51000 Len=65000
178	21.265786	172.20.20.4	172.20.20.3	UDP	1408	51000 → 51000 Len=65000
224	41.682879	172.20.20.2	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
268	41.731012	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
312	41.731826	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
358	41.845470	172.20.17.2	172.20.17.3	UDP	1408	51000 → 51000 Len=65000
402	41.862821	172.20.20.4	172.20.20.2	UDP	1408	51000 → 51000 Len=65000
450	52.623334	172.20.20.3	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
494	52.641997	172.20.20.4	172.20.20.2	UDP	1408	51000 → 51000 Len=65000
538	52.679377	172.20.20.2	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
582	52.681493	172.20.20.4	172.20.20.3	UDP	1408	51000 → 51000 Len=65000
630	72.198881	172.20.17.2	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
674	72.206439	172.20.20.4	172.20.20.3	UDP	1408	51000 → 51000 Len=65000
718	72.233492	172.20.20.3	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
762	72.240180	172.20.17.3	172.20.17.2	UDP	1408	51000 → 51000 Len=65000
808	81.057885	172.20.20.2	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
852	81.064546	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
896	81.064999	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
940	84.937704	172.20.20.3	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
984	84.943525	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
1028	84.944119	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
1074	93.560005	172.20.20.3	172.20.20.255	UDP	1408	51000 → 51000 Len=65000

> Frame 358: 1408 bytes on wire (11264 bits), 1408 bytes captured (11264 bi
 > Linux cooked capture v2
 > Internet Protocol Version 4, Src: 172.20.17.2, Dst: 172.20.17.3
 > User Datagram Protocol, Src Port: 51000, Dst Port: 51000
 Source Port: 51000
 Destination Port: 51000
 Length: 65008
 Checksum: 0x9cbc [unverified]
 [Checksum Status: Unverified]
 [Stream index: 5]

> [Timestamps]
 UDP payload (65000 bytes)

Data (65000 bytes)
 Data [truncated]: 02aaabccddddd0244444444444444420686173207265f3e
 [Length: 65000]

Offset	Hex	ASCII
0000	c7 38 c7 38 ff f0 9c bc	
0010	dd 02 44 44 44 44 44 44	.DDDDDD DD has t
0020	65 63 65 69 76 65 64 20	eceived your mes
0030	73 61 67 65 20 61 74 20	sage at its assi
0040	67 6e 65 64 20 64 65 73	gned destination
0050	00 00 00 00 00 00 00 00	.
0060	00 00 00 00 00 00 00 00	.
0070	00 00 00 00 00 00 00 00	.
0080	00 00 00 00 00 00 00 00	.
0090	00 00 00 00 00 00 00 00	.
00a0	00 00 00 00 00 00 00 00	.
00b0	00 00 00 00 00 00 00 00	.
00c0	00 00 00 00 00 00 00 00	.
00d0	00 00 00 00 00 00 00 00	.
00e0	00 00 00 00 00 00 00 00	.
00f0	00 00 00 00 00 00 00 00	.
0100	00 00 00 00 00 00 00 00	.
0110	00 00 00 00 00 00 00 00	.
0120	00 00 00 00 00 00 00 00	.
0130	00 00 00 00 00 00 00 00	.
0140	00 00 00 00 00 00 00 00	.
0150	00 00 00 00 00 00 00 00	.
0160	00 00 00 00 00 00 00 00	.
0170	00 00 00 00 00 00 00 00	.
0180	00 00 00 00 00 00 00 00	.
0190	00 00 00 00 00 00 00 00	.
01a0	00 00 00 00 00 00 00 00	.

Frame (1408 bytes)
Reassembled IPv4 (65008 bytes)

Figure 4 Example of a packet of type REPLY, first byte in header lined in yellow(02), captured in Wireshark. This is the Router 2 forwarding on a packet to Router 1. Byte 1-4 in the header is the destination ID, which is set to the ID of Endpoint 1, AABBBCCDD. Byte 5-8 in the header is the source ID, which is set to the ID of Endpoint 4, DDDDDDDDD. The router number, or the 9th byte in the header is 02 as it is Router 2 forwarding the packet. The payload, shown in blue, holds the encoding string, "DDDDDDDD has received your message at its assigned destination". This is the reply to the message above, and since the routers have added Endpoint 1 to their forwarding tables, they no longer need to broadcast the packet but can simply forward it to the next hop.

No.	Time	Source	Destination	Protocol	Length	Info
134	21.255114	172.20.20.3	172.20.20.255	UDP	1408	51000 → 51000 Len=65000
178	21.265786	172.20.20.4	172.20.20.3	UDP	1408	51000 → 51000 Len=65000
224	41.682879	172.20.20.2	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
268	41.731012	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
312	41.731826	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
358	41.845470	172.20.17.2	172.20.17.3	UDP	1408	51000 → 51000 Len=65000
402	41.862821	172.20.20.4	172.20.20.2	UDP	1408	51000 → 51000 Len=65000
450	52.623334	172.20.20.3	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
494	52.641997	172.20.20.4	172.20.20.2	UDP	1408	51000 → 51000 Len=65000
538	52.679377	172.20.20.2	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
582	52.681493	172.20.20.4	172.20.20.3	UDP	1408	51000 → 51000 Len=65000
630	72.198881	172.20.17.2	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
674	72.206439	172.20.20.4	172.20.20.3	UDP	1408	51000 → 51000 Len=65000
718	72.233492	172.20.20.3	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
762	72.240180	172.20.17.3	172.20.17.2	UDP	1408	51000 → 51000 Len=65000
808	81.057885	172.20.20.2	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
852	81.064546	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
896	81.064999	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
940	84.937704	172.20.20.3	172.20.20.4	UDP	1408	51000 → 51000 Len=65000
984	84.943525	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
1028	84.944119	172.20.17.3	172.20.17.255	UDP	1408	51000 → 51000 Len=65000
1074	93.560045	172.20.20.3	172.20.20.255	UDP	1408	51000 → 51000 Len=65000

> Frame 852: 1408 bytes on wire (11264 bits), 1408 bytes captured (11264 bits) on interface v2

> Linux cooked capture v2

> Internet Protocol Version 4, Src: 172.20.17.3, Dst: 172.20.17.255

> User Datagram Protocol, Src Port: 51000, Dst Port: 51000

- Source Port: 51000
- Destination Port: 51000
- Length: 65008
- Checksum: 0x6f23 [unverified]
- [Checksum Status: Unverified]
- [Stream index: 4]
- > [Timestamps]
- UDP payload (65000 bytes)
- Data (65000 bytes)
 - Data [truncated]: 04fffffffaabccdd01456e64706f696e7431205265717565737 [Length: 65000]

```

0000  c7 38 c7 38 fd f0 6f 23 04 ff ff ff ff aa bb cc 08-0-g#
0010  65 73 74 20 70 61 63 6b 65 74 31 20 52 65 71 75 ..Endpoint1 Requ
0020  6d 6f 76 65 20 74 68 65 69 72 20 69 6e 66 6f 20 est pack et to re
0030  66 72 6f 6d 20 66 6f 77 61 72 64 69 6e 67 20 74 move the ir info
0040  61 62 6c 65 00 00 00 00 00 00 00 00 00 00 00 from fow arding t
0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 able...
0060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00c0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00f0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
01a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Frame (1408 bytes

Figure 5 Example of a packet of type REMOVE, first byte in header seen in yellow(04), captured in Wireshark. This is the Router 1 broadcasting to the network it didn't receive the packet from, after receiving a packet from Endpoint 1, with ID AABBCDD, to Endpoint 4, with has ID DDDDDDDD. Byte 1-4 in the header is the destination ID, which is set to the special ID FFFFFFFF, which for my system allows the packet to be send to all routers without an endpoint receiving the packet, as no endpoint can have ID FFFFFFFF. Byte 5-8 in the header is the source ID, which is set to the ID of Endpoint 1. The router number, or the 9th byte in the header is 01 as it is Router 1 broadcasting. The payload, underlined in blue, holds the encoding string, "Endpoint1 Request Packet to remove their info from forwarding table". This will be broadcast across all networks so that all routers remove the ID of Endpoint 1 from their forwarding tables.

4.2 Flow Control

Below is an example of a potential flow of packets between 2 endpoints and 3 routers. Each endpoint initially sends a Broadcast to find its gateway. Endpoint1 then send a packet to endpoint 2. Since endpoint 2 is not in either router 1 or router 2's forwarding tables, they each broadcast a request across the network they didn't receive the packet from trying to find it. Once router 3 receives the packet, since it has endpoint 2 in its forwarding table it does not need to send a broadcast but simply forwards the packet to the next hop, which in this case is the endpoint itself. Endpoint 2 the sends a reply with the destination ID of endpoint 1's ID. All routers now have this ID in their forwarding tables so no broadcasts are sent and each forwards the reply packet onto the next hop.

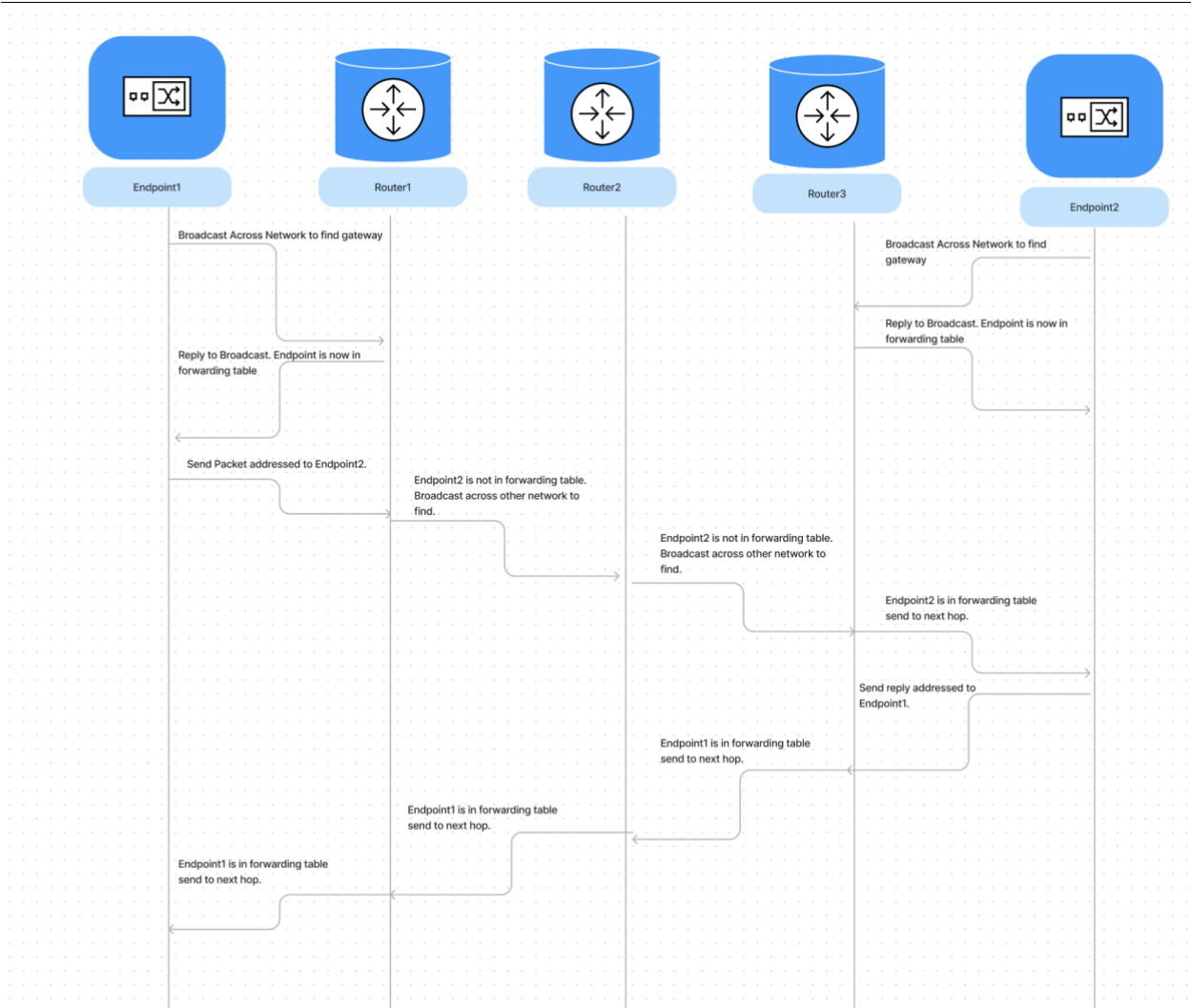


Figure 6 A representation of the initial flow of packets between endpoints and routers, including establishing a connection between endpoint and gateway, sending a message to another endpoint, and replying to said message.

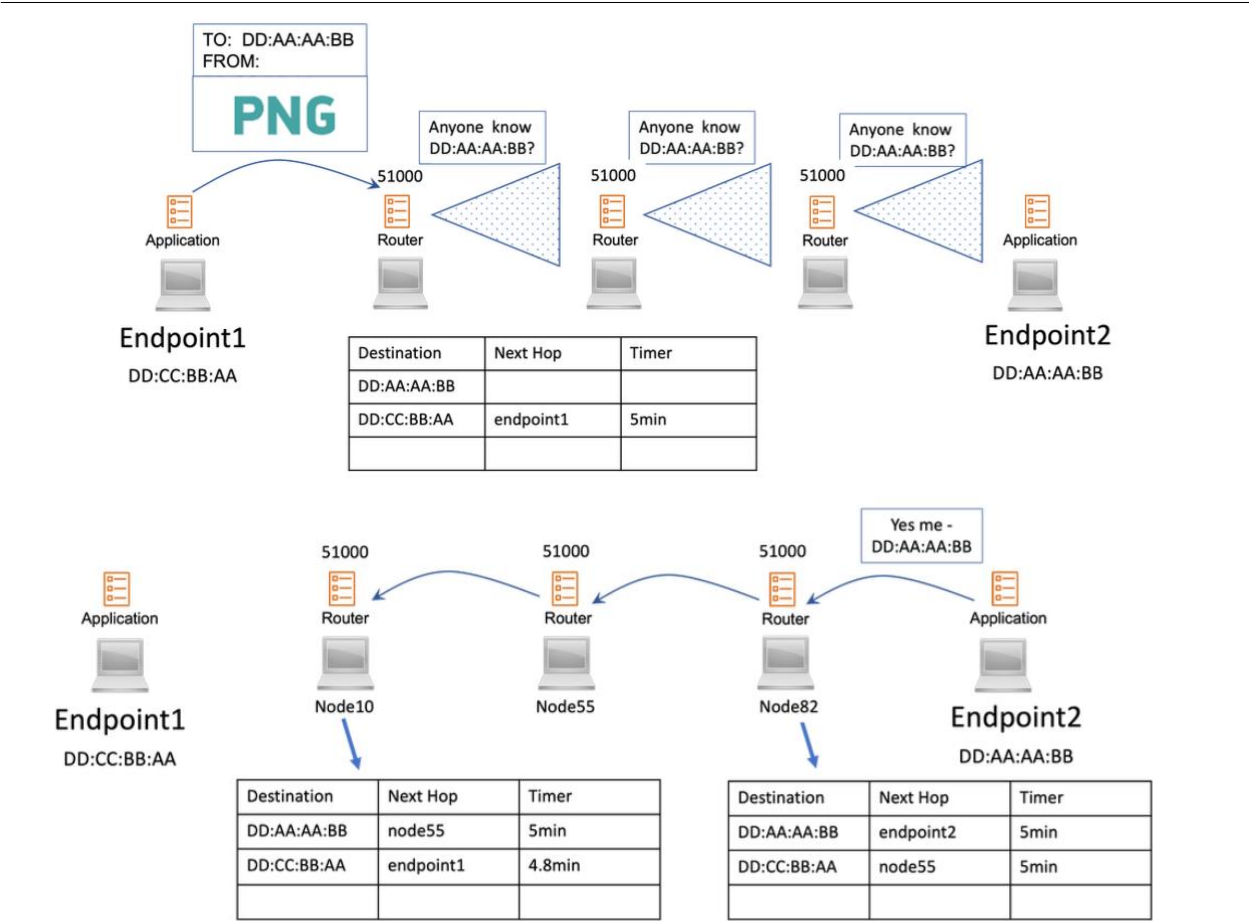


Figure 7 A further look at the communication between endpoints this time with forwarding tables visible, an clearly showing the updated forwarding tables after the broadcasts.

5 Implementation

This section provides the breakdown of the code for my solution. I explain my implementations of the network elements of the Endpoint and Router and their superclass Node and provide examples of their output in the command line interface.

5.1 Node.java

A basic version of this class was provided to us as a sample and importantly contains a nested Listener class which constantly listens for any incoming packets, calling the onReceipt function on arrival. I have updated the class to contain various useful functions to decode and encode packets. The class starts with a list of constants, seen in Listing 1, which include the packet-size and a list of potential packet types. The most important addition from my previous assignment is the HashMaps containing network information that routers and endpoints access when they initialise.

```

public abstract class Node {
    static final int PACKETSIZE = 65000;
    static final byte MESSAGE = 1;
    static final byte REPLY = 2;
    static final byte BROADCAST = 3;
    static final byte REMOVE = 4;
    static final int ROUTER_PORT = 51000;
    static final String DEFAULT_DST_NODE = "localhost";
    static final int NUMBEROFENDPOINTS = 4;
    static final int IDSIZE = 4;
    public static Map<String, String> routerAddresses;
    public static Map<String, String> endpointAddresses;
    static {
        routerAddresses = new HashMap<>();
        endpointAddresses = new HashMap<>();
        routerAddresses.put(key:"1", value:"172.20.17.255,172.20.20.255");
        routerAddresses.put(key:"2", value:"172.20.17.255,172.20.19.255");
        routerAddresses.put(key:"3", value:"172.20.19.255,172.20.18.255,172.20.1.255");
        endpointAddresses.put(key:"1", value:"172.20.20.255");
        endpointAddresses.put(key:"2", value:"172.20.18.255");
        endpointAddresses.put(key:"3", value:"172.20.20.255");
        endpointAddresses.put(key:"4", value:"172.20.1.255");
    }
}

```

Listing 1: Constants at the start of Node.java. The two HashMaps are to store network information for the router and endpoints respectively

I have modified the method to encode packets to incorporate the new header described in 4.1. The method in Listing 2 shows how the header is encoded into a byte array and the remaining space is used to encode the payload. The first four parameters are for the header I created, while the last parameter are for the payload. The payload is a String because this is the communication I decided was best for the endpoints.

```

protected byte[] createHeader(byte packet_type, byte[] dest_endpoint_id, byte[] source_endpoint_id, byte currentRouter, String msg) {
    byte[] data = new byte[PACKETSIZE];
    data[0] = packet_type;
    for (int i = 0; i < 4; i++) {
        data[i + 1] = dest_endpoint_id[i];
    }
    for (int i = 0; i < 4; i++) {
        data[i + 5] = source_endpoint_id[i];
    }
    data[9] = currentRouter;
    byte[] msg_array = msg.getBytes();
    for (int i = 0; i < msg_array.length && i < PACKETSIZE; i++) {
        data[i + 10] = msg_array[i];
    }
    return data;
}

```

Listing 2: Method to encode given data into packet. Byte 0 is the packet type, byte 1-4 is the destination ID, byte 5-8 is the source ID and byte 9 is the current router, the rest of the bytes is allocated to the message String.

5.2 Router.java

This class is used for the router node in my implementation. The broker takes no user input and is initialised through the first argument passed to it. Node.java contains a Hashmap which holds its forwarding information, where the endpoint ID is the key and the InetAddress of the next hop is the value. Below is the code for the router's onReceipt method. It first checks if the packet is from its self by checking the current router number in the header, if it is it ignores it. Otherwise it processes the packet based on its type seen in the switch statement. Messages will either be forwarded directly to the next hop or broadcasted across the router's other

networks depending on whether or not the destination ID in the header is in its forwarding table. Reply's and Broadcasts are always forwarded as the destination will always be in the router's forwarding table. Finally REMOVE requests remove the endpoint's routing information from the forwarding table and the router's broadcasts across its other networks in order for all other routers to do the same.

```
public synchronized void onReceipt(DatagramPacket packet) {
    System.out.println("Received packet from " + packet.getAddress());
    printForwardingTable();
    try {
        byte[] data = packet.getData();
        System.out.println("Current Router " + getCurrentRouter(data) + " router number " + number);
        System.out.println("Request recieved from " + getSource_id(data) + " with dest " + getDest_id(data));
        if(!getCurrentRouter(data).equals(number)) {
            System.out.println("Request recieved from to send TEXT: " + getMessage(data) + " packet type: " + getType(data));
            switch(getType(data)) {
                case MESSAGE:
                    System.out.println("Request recieved from " + getSource_id(data) + " to send TEXT: " + getMessage(data));
                    System.out.println("Do I know " + getSource_id(data) + "?");
                    if(!forwardingTable.containsKey(getSource_id(data))) {
                        addEntry(getSource_id(data), (InetSocketAddress)packet.getSocketAddress());
                        printForwardingTable();
                    }
                    System.out.println("Do I know " + getDest_id(data) + "?");
                    if(forwardingTable.containsKey(getDest_id(data))) {
                        forwardMessage(data, getDest_id(data));
                    } else {
                        createMessage(data, packet.getAddress());
                    }
                    break;
                case REPLY:
                    System.out.println("Request recieved from " + getSource_id(data) + " to send Reply to " + getDest_id(data));
                    if(!forwardingTable.containsKey(getSource_id(data))) {
                        addEntry(getSource_id(data), (InetSocketAddress)packet.getSocketAddress());
                        printForwardingTable();
                    }
                    forwardMessage(data, getDest_id(data));
                    break;
                case BROADCAST:
                    System.out.println("Request recieved from to send Address to " + getDest_id(data));
                    if(!forwardingTable.containsKey(getDest_id(data))) {
                        addEntry(getDest_id(data), (InetSocketAddress)packet.getSocketAddress());
                    }
                    forwardMessage(data, getDest_id(data));
                    break;
                case REMOVE:
                    System.out.println("Request recieved from to " + getSource_id(data) + " to delete info from forwarding table");
                    if(forwardingTable.containsKey(getSource_id(data))) {
                        forwardingTable.remove(getSource_id(data));
                    }
                    printForwardingTable();
                    createMessage(data, packet.getAddress());
                    break;
            }
        }
    }
}
```

Listing 3: Code snippet showing onReceipt method for router. Processed based on the packet type, first byte of the header. Receiving packets with destination endpoints not known to the router prompt a broadcast, while endpoints in its forwarding table allow the router to simply forward the packet to the next hop

Below in Listing 4 is the code for the sending methods discussed above. The forwarding method simply get the value of the next hop, which is a InetSocketAddress, from its forwarding table based on the key of the destination ID of the packet, and sends on the packet to this address, making sure to change the router number. The createMessage method sets broadcast on and loops through the router's networks it's connected to, making sure to excluded the network it received the packet from.

```

public synchronized void forwardMessage(byte[] sendData, String dst) {
    setCurrentRouter(sendData, (byte)Integer.parseInt(number));
    // Forward to next hop
    try {
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, forwardingTable.get(dst));
        socket.send(sendPacket);
        System.out.println("Router Request packet forwarded to next hop: " + forwardingTable.get(dst));
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public synchronized void createMessage(byte[] sendData, InetAddress receiverAddress) {
    System.out.println(receiverAddress);
    setCurrentRouter(sendData, (byte)Integer.parseInt(number));
    // Find the server using UDP broadcast
    try {
        socket.setBroadcast(true);
        for(int i = 0; i < addresses.size(); i++) {
            if(!split(addresses.get(i)).equals(split(receiverAddress.toString().replace(target:"/", replacement:"")))) {
                try {
                    DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, InetAddress.getByName(addresses.get(i)), ROUTER_PORT);
                    socket.send(sendPacket);
                    System.out.println("Router Request packet sent to: " + addresses.get(i));
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

Listing 4: Methods showing the different kinds of ways the router can communicate, either forward directly to the next or broadcasts across all other networks from the network it received the packet from.

```

Received packet from /172.20.20.2
Printing current forwarding table Router 1 receives initial Broadcast from Endpoint 1 to establish connection.
Current Router 0 router number 1
Request recieved from 00000000 with dest AABBCDD
Request recieved from to send TEXT: packet type: 3
Request recieved from to send Address to AABBCDD
Router Request packet forwarded to next hop: /172.20.2:51000
Received packet from /172.20.20.4
Printing current forwarding table Router 1 receives initial Broadcast from Endpoint 3 to establish connection.
AABBCDD /172.20.2:51000
Current Router 0 router number 1
Request recieved from 00000000 with dest DDDDDDD
Request recieved from to send TEXT: packet type: 3
Request recieved from to send Address to DDDDDDD
Router Request packet forwarded to next hop: /172.20.4:51000
Received packet from /172.20.20.2
Printing current forwarding table Router 1 receives message from Endpoint 1 to Endpoint 2.
AABBCDD /172.20.2:51000
DDDDDDD /172.20.4:51000
Current Router 0 router number 1
Request recieved from AABBCDD with dest DDDDDDD
Request recieved from to send TEXT: this is 1 packet type: 1
Request recieved from AABBCDD to send TEXT: this is 1
Do I know AABBCDD?
Do I know DDDDDDD?
Router Request packet forwarded to next hop: /172.20.4:51000
Received packet from /172.20.20.4
Printing current forwarding table Router 1 receives reply from Endpoint 3 to Endpoint 1.
AABBCDD /172.20.2:51000
DDDDDDD /172.20.4:51000
Current Router 0 router number 1
Request recieved from DDDDDDD with dest AABBCDD
Request recieved from to send TEXT: DDDDDDD has received your message at its assigned destination packet type: 2
Request recieved from DDDDDDD to send Reply to AABBCDD
Router Request packet forwarded to next hop: /172.20.2:51000
AABBCDD has been removed from forwarding table
DDDDDDD has been removed from forwarding table Lifetimes over

```

Figure 8: Command Line interface snipped from Router 1. Shows initialisation of endpoints and the establishment of their connection to Router 1 which acts as their gateway. They then communicate with each other as Endpoint1 sends a message and receives a reply from Endpoint 3. Finally both their forwarding information is removed from Router 1's forwarding table after the entries lifetimes expire. Once the endpoints want to re-establish connection another broadcast will be sent out before they send any packets to other endpoints.

5.3 Endpoint.java

Endpoints have three instructions they can carry out:

1. Send a message to another endpoint.
2. Wait to receive messages from other endpoints
3. Remove its data from the forwarding tables of all routers.

```
Which endpoint are you?  
1  
What is your ID?  
aabbccdd  
Endpoint1 Request packet sent to: 172.20.20.255 (DEFAULT)  
Endpoint1 Done looping over all network interfaces. Now waiting for a reply!  
Reply to Broadcast Received received: /172.20.20.4:51000  
Instructions:  
>SEND  
To send a message to another Endpoint  
>WAIT  
To wait for messages  
>REMOVE  
To remove your info from routers  
send  
What ID do you want to send to?  
dddddddd  
Enter text you want to send  
this is 1  
Endpoint 1 Request packet sent to: /172.20.20.4:51000  
REPLY received: DDDDDDDD has received your message at its assigned destination
```

Figure 9 Command Line interface snipped from Endpoint 1. Shows initialisation of endpoint and the establishment of their connection to Router 1 which acts as its gateway. It then sends a packet to endpoint ID DDDDDDDD, and receives a reply from that endpoint.

I will discuss 1 and 2 first. The methods below facilitate the two-communication from endpoint to endpoint. createMessage asks the user to input an ID to send to, which it then checks if it is 8 digit hexadecimal. It then asks for the message to be in encoded and sends the packet through its gateway router. Once an endpoint receives a packet of type MESSAGE, it will call response message with a new reply String which will be sent to the source ID of the packet it just received.


```

public void createMessage(Scanner scan) {
    byte[] endpoint_dst;
    while(true) {
        System.out.println(x:"What ID do you want to send to?");
        String endpoint = scan.nextLine();
        if(endpoint.matches(regex:"[a-zA-Z0-9]*")) {
            byte[] test = HexFormat.of().parseHex(endpoint);
            if(test.length == IDSIZE) {
                endpoint_dst = test;
                break;
            }
        }
        System.out.println(x:"Invalid ID. ID must be 8 digit hexadecimal number.");
    }
    System.out.println(x:"Enter text you want to send");
    String message = scan.nextLine();
    // Find the server using UDP broadcast
    try {
        byte[] sendData = createHeader(MESSAGE, endpoint_dst, endpoint_id, (byte) 0, message);
        try {
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, dstAddress);
            socket.send(sendPacket);
            System.out.println("Endpoint " + number + " Request packet sent to: " + dstAddress);
        } catch (Exception e) {
            e.printStackTrace();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void responseMessage(String message, byte[] endpoint_dst) {

    // Find the server using UDP broadcast
    try {
        byte[] sendData = createHeader(REPLY, endpoint_dst, endpoint_id, (byte) 0, message);
        try {
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, dstAddress);
            socket.send(sendPacket);
            System.out.println("Endpoint" + number + " Request packet sent to: " + dstAddress);
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("Endpoint " + number + " sent reply");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

Listing 5: Code snippet showing the sending and replying methods in the endpoints. The error-handling

The third instruction to remove data from forwarding table is done by the `removeInfo` function below in Listing 6. It simply sends a new request to its gateway router with packet type REMOVE.


```

public void removeInfo() {
    // Find the server using UDP broadcast
    try {
        byte[] sendData = createHeader(REMOVE, removeInfoID, endpoint_id, (byte) 0,
            "Endpoint" + number + " Request packet to remove their info from forwarding table");
        try {
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, dstAddress);
            socket.send(sendPacket);
            System.out.println("Endpoint" + number + " Request packet sent to: " + dstAddress + " to remove their info from forwarding table");
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("Endpoint " + number + " sent reply");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void sendBroadcast() {
    // Find the server using UDP broadcast
    try {
        socket.setBroadcast(true);
        byte[] sendData = createBroadcastHeader(BROADCAST, endpoint_id, (byte) 0);
        try {
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, InetAddress.getByName(networkAddress), ROUTER_PORT);
            socket.send(sendPacket);
            System.out.println(x:"Endpoint1 Request packet sent to: 172.20.20.255 (DEFAULT)");
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println(x:"Endpoint1 Done looping over all network interfaces. Now waiting for a reply!");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

Listing 6: Methods to remove is routing information for routers' forwarding tables and to send broadcast across network to find gateway router. The method sendBroadcast is sent when the endpoint is initialised and whenever the endpoint wants to re-establish communication after removing it routing information using the first method above.

6 Discussion

I am happy with my solution to this assignment, particularly in comparison to the previous one. I have learned from the previous assignment that a bash script is the easiest way for me to set up docker and that is not necessary to have classes for each node when you can take the information in on the command line. I also implemented error-handling for the endpoints. The strengths of my solution lie in the minimal packets sent out to establish routing information. The endpoints don't have hardcoded gateways but instead find them through broadcasts and routers only send broadcasts on networks they did not receive a packet from. Since forwarding information is established between endpoints and their gateway router immediately, this means that endpoints sending packets to each other over the same network do not need any initial broadcasts as they are both in the forwarding table of the shared router. Endpoints can also only send packets to other endpoints who have been initialised, meaning there can be no hanging waiting for a reply from an endpoint that doesn't exist.

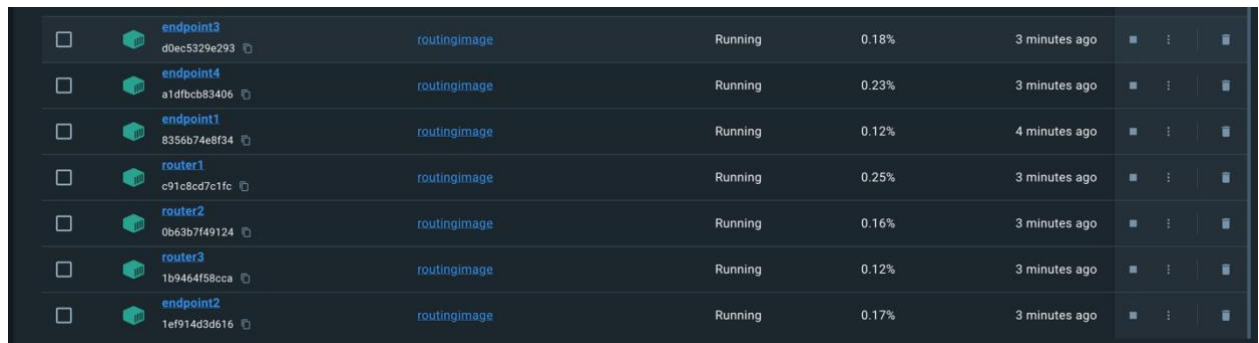
However, this solution is far from ideal. One disadvantage is that in my current header the number of routers is limited to 1 byte. This could easily be fixed by increasing the bytes this part of the header takes up but I feel that a more elegant solution is possible for the problem of determining the current router, I just was not able to get anything working in this time-frame. Another element of my solution that could be worked on is the lifetime for the routing information. I did not put a huge priority on this so I feel like I could have explored this feature more. Acknowledgements are implemented in my solution for the initial endpoint broadcast and for the removing of routing information but are not for the sending of packets between endpoints, instead endpoints only wait for

a reply from the destination of their packet. Another possible addition is a controller for the protocol which

could help implement error-handling for endpoints and routers so that there would be now duplicate numbers or IDs.

7 Summary

This report has described my attempt at a solution to the Flow Forwarding Protocol. The description of the implementation in this document provides the essential components and design elements required for my solution to work. The end result functions well, allowing simple establishment of routing information through broadcasts, and then using that routing information stored in forwarding tables to send packets between endpoints without broadcasting. I have also added additional functionality such as lifetimes of forwarding table entries.



Container Name	ID	Image	Status	Usage	Updated	Actions
endpoint3	d0ec5329e293	routingimage	Running	0.18%	3 minutes ago	Stop, Restart, Logs, etc.
endpoint4	a1dfbcb83406	routingimage	Running	0.23%	3 minutes ago	Stop, Restart, Logs, etc.
endpoint1	8356b74e8f34	routingimage	Running	0.12%	4 minutes ago	Stop, Restart, Logs, etc.
router1	c91c8cd7c1fc	routingimage	Running	0.25%	3 minutes ago	Stop, Restart, Logs, etc.
router2	0b63b7f49124	routingimage	Running	0.16%	3 minutes ago	Stop, Restart, Logs, etc.
router3	1b9464f58cca	routingimage	Running	0.12%	3 minutes ago	Stop, Restart, Logs, etc.
endpoint2	1ef914d3d616	routingimage	Running	0.17%	3 minutes ago	Stop, Restart, Logs, etc.

Figure 9 This shows my solution working in Docker: 4 endpoints and 3 routers are running.

My solution contains 3 classes:

- **Node.java:** Abstract class which contains methods to receive and encode packets as well as helper functions to decode packets. Also holds networks that can be accessed by endpoints and routers.
- **Router.java:** Router node that keeps track of forwarding information and facilitates communication between endpoints.
- **Endpoint.java:** Can send messages to other endpoints and remove their routing information from routers.

8 Reflection

I found this assignment very interesting to reason about, and fairly easy to understand the basics of. I appreciated the opportunity to create my own protocol from the ground up, and I believe I really built upon my knowledge from the last assignment. I realized when coming up with my solution that I was much more comfortable with the java.net library and once I worked out broadcasts I felt my solution fell into place. I found using a bash script was better for me for this assignment when testing user input but I would like to have used and tested docker compose as it seems like a useful tool.