



CS2031 Computer Networks

Assignment #2: Flow Forwarding

Adriana Hrabowych, 19304296

December 4, 2022

Contents

1	Introduction	1
2	System Design	2
2.1	Openflow	2
2.2	Components	2
2.3	Packet Description	3
2.4	Models	4
3	Implementation	7
3.1	Node	7
3.2	Endpoint	8
3.3	Forwarder	9
3.4	Controller	11
3.5	Packet Encoding	11
4	Discussion	13
5	Summary	14
6	Reflection	14

1 Introduction

The focus of this assignment was to learn about forwarding tables and routing UDP packets over multiple networks, specifically in an Openflow method. This is accomplished through the creation of a flow forwarding protocol, where forwarders on a network can contact a central controller to ask for the path a packet should take to make it to its desired endpoint. For this assignment, I took an approach of keeping forwarding decisions simple as to not have the controller be too overloaded while still being able to handle a variety of different packets in different orders with no packet ever being truly dropped.

In the following report I will discuss my solution in greater detail, from a broad explanation of theory to a description of the network layout and packet types I implemented. This will then be followed by specific descriptions of my components and how they were implemented.

2 System Design

In this section, I will go over the design behind the implementation, and how that design was reached.

Before beginning the assignment, through reviewing lecture material and independent sources, I researched the basic concepts of forwarding tables, openflow, packet routing, and communicating over different networks with docker.

I based my system model on the sample topology given in the assignment description, with only a few small differences, mainly that I added more endpoints and forwarders in order to increase complexity in my sample. After I had an idea of what components I wanted to include and how they would communicate, I then focused on what data was necessary in order to facilitate communication and how it could be stored efficiently in packets. In the end, my solution ended up with 3 unique components and 1 type of UDP packet, albeit one that is highly customizable.

2.1 Openflow

Using UDP packets was a requirement and through research, I decided that the best course of action was to very loosely base my solution on the Openflow protocol.

The Openflow protocol was recommended as something to research and potentially use as a base when working on the assignment. Openflow is based upon Software-Defined Networking (SDN); a networking concept which splits the architecture into three layers. The application layer, control layer, and the data layer, which is where endpoints, controllers, and forwarders lie respectfully. Openflow defines how the control and data layer, or the controller and 'switches', interact.

A switch will open communication with the controller with a 'Hello' packet which the controller responds to with another 'Hello'. The switch and controller then inform each other of their capabilities, the version of Openflow they support, the number of flow tables they support, etc through the use of Feature Request and Feature Reply packets. Once everything is decided, the connection is established and the switch can now request modifications for its flowtable from the controller upon receiving packets not in its own flow table.

2.2 Components

As mentioned previously in the report, my solution has three unique components that act in the protocol. The functionalities of which are as follows,

Endpoint: Whenever a new endpoint is started, it automatically sends a request to the controller asking for all of the forwarders on the same network as itself and stores the list of forwarders received in return. An endpoint then becomes open for configuration, it can be configured to either send or receive. If an endpoint is configured to receive it informs all of the forwarders on its network, waits for an acknowledgement from each forwarders and for any previously dropped messages to be received, and then becomes available to receive a new message. If an endpoint is configured to send it only informs one forwarder on its network, waits for an acknowledgement, and then sends on a packet containing the message and the destination of the message to that forwarder. After the endpoint completes the task its configured to do, either received a message or sent one, it then becomes available to be configured again.

Forwarder: Forwarders facilitate messages between endpoints. Whenever a new forwarder is started, it automatically sends a packet to the controller with a list of the networks it's on and becomes active when it receives an acknowledgement in response. Within the forwarder is a forwarding table, which stores where packets need to be sent based on requested destinations. Whenever an endpoint informs the forwarder that it's ready to receive, the forwarder will add it to its table and sends on any previously dropped message packets it was storing for that endpoint. If the forwarder receives a message for a destination within its forwarding table, it forwards that message to the destination and removes the connection.

However, if a forwarder receives a message packet for a destination not within its table, it stores the dropped message and requests a path to the destination from the controller. Upon receiving a flow update from the controller, the forwarder forwards all dropped packets for that destination using the updated path. It then also stores that new path in its forwarding table for future use.

Controller: The controller is the central hub of communication in the network, all other components must interact with it in order to set up and it stores all the connections between components. A controller will have many forwarders/endpoints connected to it at once. The controller handles connection requests from forwarders and stores which networks that forwarder is on, then sends an acknowledgement. It also handles connection requests from endpoints and informs that endpoint what forwarders are on its network.

If a forwarder doesn't know the path to an endpoint, it will request it from the controller. The controller will then either respond with the path or if there isn't one, 'drops' the packet. This is done via a routing table stored within the controller that keeps track of all the forwarder to forwarder and forwarder to endpoint connections. This table is updated with every connect and disconnect request received from forwarders, which the controller will then respond to with an acknowledgement. Each time the table is updated, the controller checks if any of the previously dropped packets now has a path and will update forwarders accordingly via a flow control packet.

2.3 Packet Description

There are many different types of requests/responses needed for protocol functionality, each with different data necessities. As making individual packet classes for each would be time consuming and unstable, I decided to have only one type of UDP packet but make use of the Type Length Value (TLV) formatting in order to be able to heavily customize packets. These TLV packets consist of a large TLV which contains many small TLV's within. The first section of the packet header contains the packet type, a number which then determines what the rest of the header will contain. The length defines how many small TLVs are within the value. Then the value is a sequence of small TLVS, each with their own TLV type, byte length of the value, and some data for the value.

There are six TLV types, each of them have some value and length that depend on input. They are as follows:

TLV Type	Type ID
Message	1
Network ID	2
Port Number	3
Sender Container Name	4
Destination Container Name	5
Container Name	6

There are eight TLV Packet Types, each of which will be made up of some amount of TLV types within its value. They are as follows¹:

ID	TLV Type	Length	Value
1	Acknowledgement	1	[Message]
2	Message	3+	[Message][Destination Container Name][Sender Container Name][ContainerName]*
3	Connect Endpoint	2	([Port Number][Message])[Container Name]
4	Connect Forwarder	2+	[Container Name][Network ID]+
5	Flow Request	2	[Destination Container Name][Container Name]
6	Flow Response	2+	[Destination Container Name][Container Name]+
7	Forwarder List Request	1	[Container Name]
8	Forwarder List	1+	[Container Name]+

¹The value field in the table follows Regular Expression rules. '|' is logical or, '+' is 1 or more, and '*' is 0 or more.

Some examples of various packets:

Type: 1 Length:1 V: 13ACK

This is an acknowledgement packet, as specified by the type, it has a length of one TLV packet in its value. The value is a message TLV (type 1), with a length of three characters, "ACK".

Type: 6 Length:2 V: 54ENDA64fone

This is an flow request packet, as specified by the type, it has a length of two TLV packets in its value. The value is made up of a destination container name TLV of length four and value "ENDA", and a container name TLV of length four and value "fone".

Type: 3 Length:25 V: 15Hello54ENDD34ENDA64ftwo66fthree

This is a complex message packet, instead of the base length of three it has five TLVs within its value. The first three define the message, destination, and sender. Then there are two more container name TLVs at the end which informs the protocol that in order to get from ENDA to ENDD this packet needs to be send to ftwo and then fthree.

2.4 Models

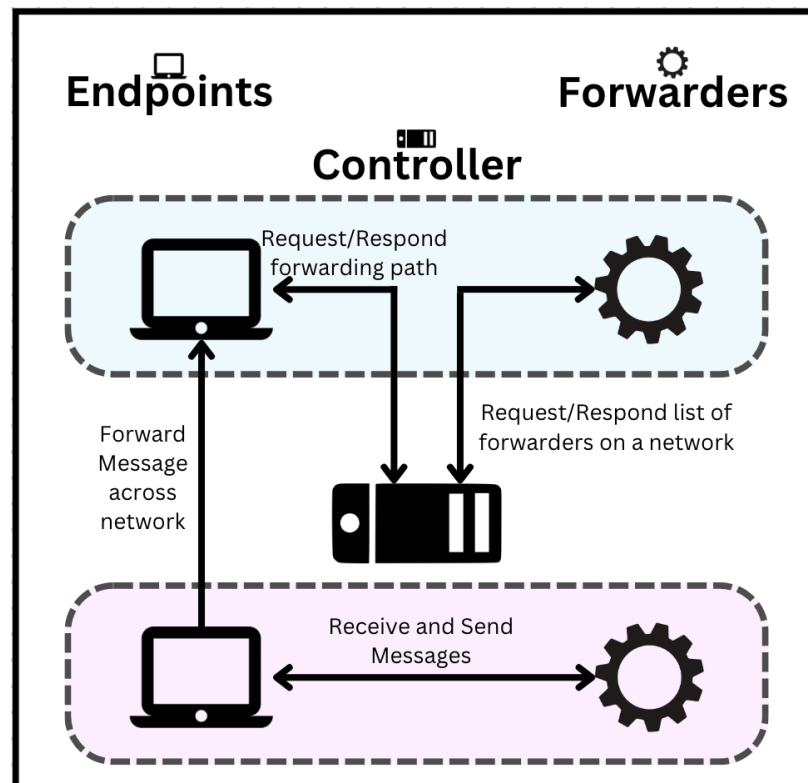


Figure 1: A diagram showcasing the fundamental design of the solution.

Figure one highlights the base design of my solution, with endpoints and forwarders on different subnetworks communicating with a global controller in order to send and receive messages. Figure two describes how each of the components were connecting using docker. Each component was ran within its own container on a subnet on the default Docker Bridge network, except for the controller which is just ran on the default network. Each subnet needs at least one forwarder/endpoint and forwarders typically sit on two subnets at once (though it is possible in my protocol to have them be connected to more). The IP addresses of each container were not included as depending on the order in which the containers are started each time, the IP addresses can vary.

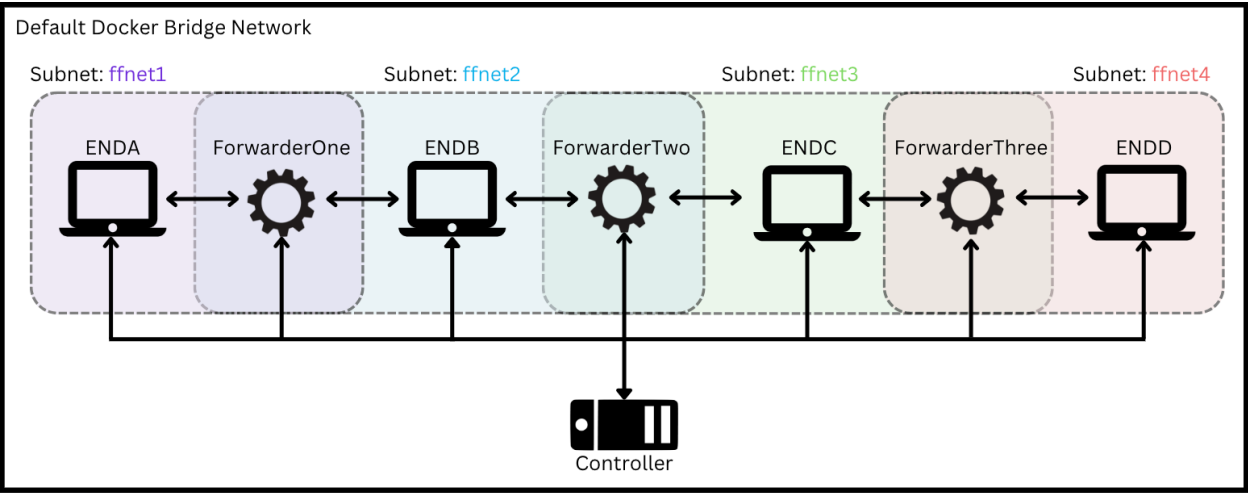


Figure 2: A sample network diagram of the solution in Docker.

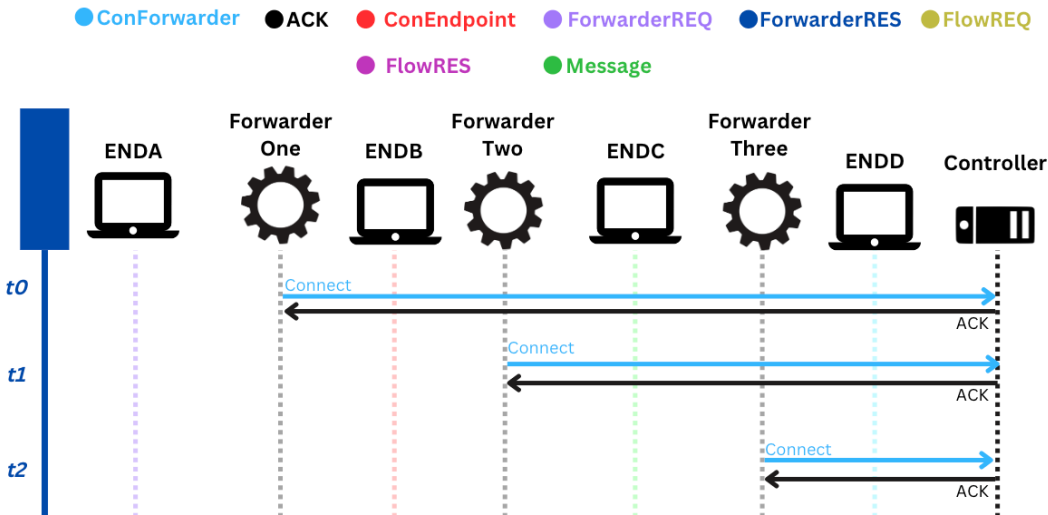


Figure 3: A flow diagram that shows each of the active forwarders connecting to the controller upon startup.

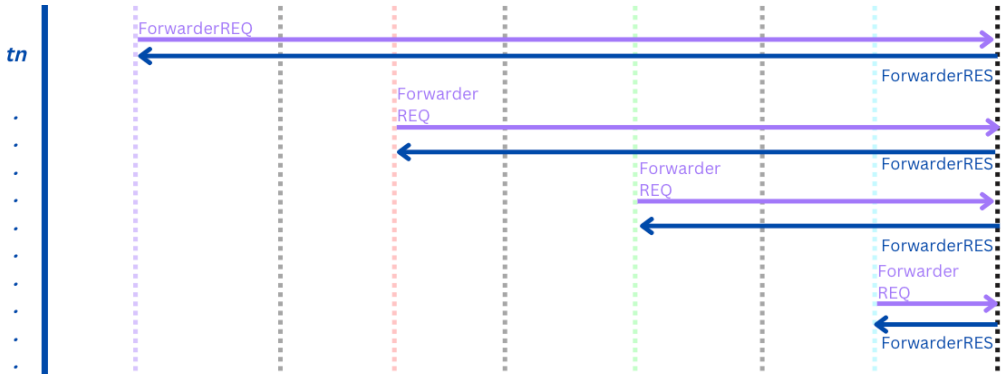


Figure 4: A flow diagram that shows each of the active endpoints connecting to the controller upon startup.



Figure 5: A flow diagram that shows a basic message protocol with no need for flow requests. Connection is added, message forwarded, connection removed.

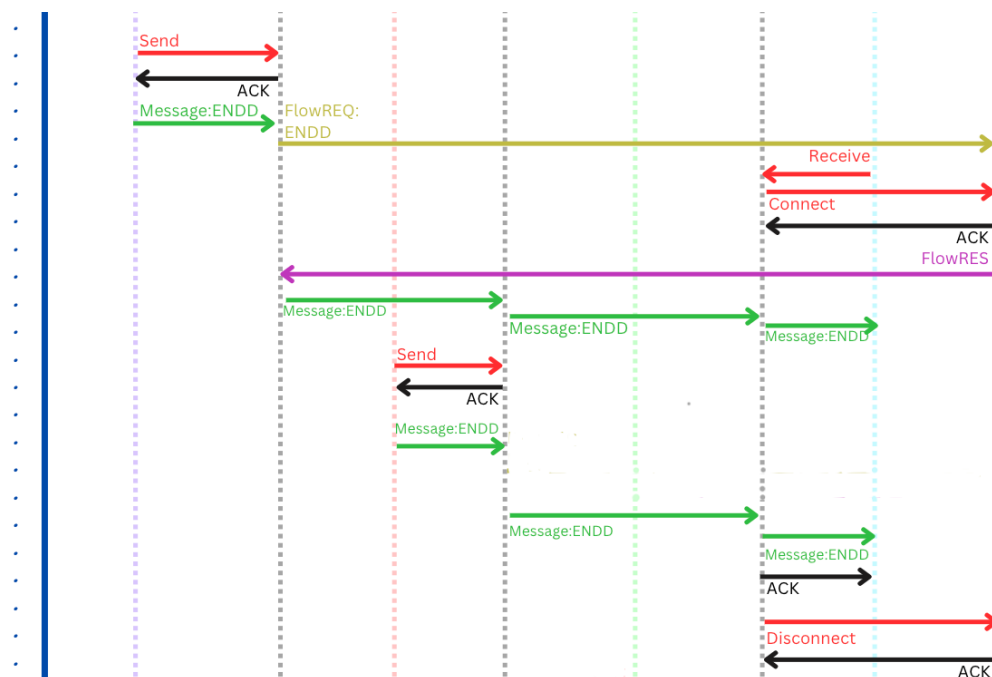


Figure 6: A flow diagram that shows a more complex message protocol. ENDA wants to send a packet to ENDD, but ENDD is not ready to receive yet, so fone requests apath and stores the dropped packet. When ENDD is available the controller updates fone with the path and the dropped packet is forwarded accordingly. Then ENDB also send a packet, ftwo stored the previous path and the message is forwarded.

3 Implemenation

This section discusses the details of the implementation of the prior theory. The solution was written using Java datagram packets and ran on Docker containers. XQuartz and wireshark were used to monitor and visualise the packets being sent. I used the sample Java code provided on blackboard as a base point for my solution.

3.1 Node

All of the components are extensions of the Node class, this class is exactly the same as in the sample except for the addition of more variables in order to make reading and encoding TLV packets simpler. Also the more variables are used over hardcoded integers, the easier the code is to read.

```
//TLV TYPES
public static final String ACK_PACKET = "1";
public static final String MESSAGE_PACKET = "2";
public static final String CON_ENDPOINT = "3";
public static final String CON_FORWARDER = "4";
public static final String FLOW_CONTROL_REQ = "5";
public static final String FLOW_CONTROL_RES = "6";
public static final String FORWARDER_LIST_REQ = "7";
public static final String FORWARDER_LIST = "8";

public static final String T_MESSAGE = "1";
public static final String T_NETWORK = "2";
public static final String T_SENDER_NAME = "3";
public static final String T_PORT = "4";
public static final String T_DEST_NAME = "5";
public static final String T_CONTAINER = "6";
```

Listing 1: Global variables within the node class that every other component can access. The integers are the same as the ID's defined in the report previously.

There are two more variables defined in the node not to do with TLVs, these are ip addresses of the default docker bridge and host network. These are defined here and not in the forwarder, despite only the forwarder requiring them, in order to make finding and editing them easier as it is needed in order to run the protocol on different machines.

```
//EDIT THESE NEXT TWO VARIABLES BASED ON DEVICE
public static final String BRIDGE_NET_IP = "/172.17.0";
public static final String HOST_NET_IP = "/127.0.0";
```

Listing 2: Two global variables that define the ip addresses of the default docker bridge network and host network.

The last addition to the Node class is two non-final variables that are defined in the constructor. These are the alias of the container the file is currently running in, and the length of that name. These are set here as to reduce repeated code between components and to allow for more versatality as previously container names were hardcoded.

```
String containerAlias;
String aliasLength;
Node() {
    try{
        containerAlias = InetAddress.getLocalHost().getHostName();
        aliasLength = Integer.toString(containerAlias.length());
```

Listing 3: Using methods available in the Java.net library, it's possible to get the running container alias (not the container name) from within a file.

3.2 Endpoint

The functionalities of an Endpoint are exactly as described previously in the theory section, it can handle receiving acknowledgements, forwarder list, and message packets. To make the protocol more real, endpoints don't always have the same port number. The port number is defined in the constructor and depends on the final char in the container name for an endpoint.

```
Endpoint(Terminal t, String dstHost, int dstPort) {
    try {
        int lastChar =
            containerAlias.charAt((containerAlias.length() - 1));
        if(lastChar == 'A')
            DEFAULT_SRC_PORT = 50001;
        else if(lastChar == 'B')
            DEFAULT_SRC_PORT = 50002;
        else if(lastChar == 'C')
            DEFAULT_SRC_PORT = 50003;
        else
            DEFAULT_SRC_PORT = 50004;
        endpointTerminal = t;
        controllerAddress = new InetSocketAddress(dstHost, dstPort);
        socket = new DatagramSocket(DEFAULT_SRC_PORT);
        listener.go();
    }
}
```

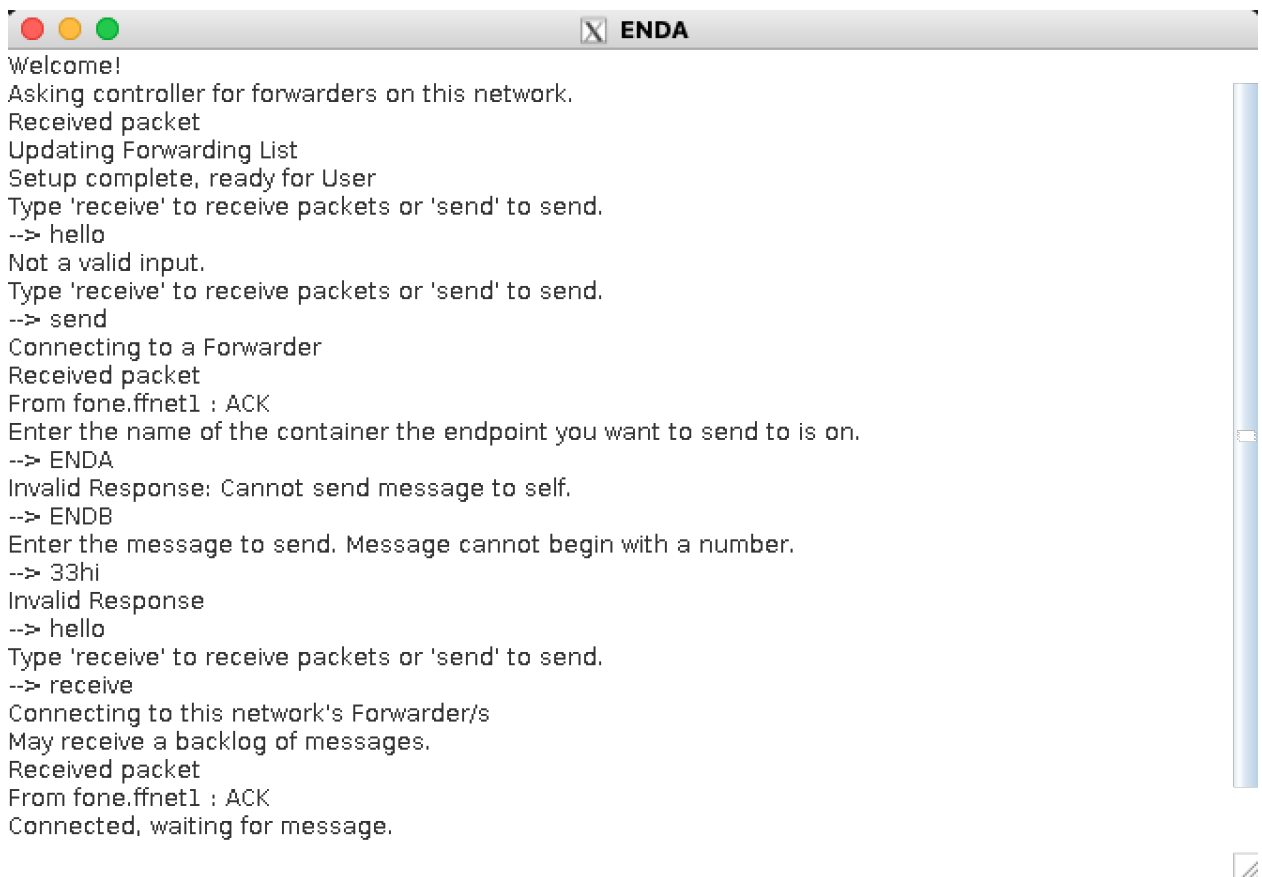
Listing 4: Using the containerAlias, the port number for the endpoints socket is set to a port between 50001 and 50004.

Upon startup, an endpoint will immediately send a forwarder list request to the controller and then waits to receive a list of forwarders in return. The first forwarder in the received list is set as the default forwarder, the endpoint will only use this forwarder to send messages but will inform all the forwarders in the list when it wants to receive. If there are no forwarders on an endpoints network and the list is empty, the endpoint will throw an error and quit as its necessary for there to be at least one forwarder for the endpoint to be able to do any other functionality.

```
if(forwarders.size() < 1)
{
    System.out.println("ERROR: no forwarders on this endpoints network,
    please run all forwarders before endpoints.");
    System.exit(0);
}
else
    defaultForwarderAddress = new InetSocketAddress(forwarders.get(0)
    FORWARDER_DST_PORT);
```

Listing 5: *forwarders* is the name of local ArrayList<String>, an endpoint uses to store its list of forwarders, this is set to empty upon construction and is updated only once upon receive of a forwarder list packet.

An Endpoint makes use of a terminal class in order to allow for user interaction. As the protocol goes on an endpoint will be set between sending and receiving packets by the user multiple times. As such there is some error handling in place in order to make sure no unviable inputs are entered, one example of this is that if a user attempts to make the endpoint send a packet to itself a warning will be printed. It is important to mention that endpoints do not wait for an acknowledgement that their message made it to its destination before being able to send/receive a new one. This is because the packet may have been dropped and it might take some time before a path is found and I think it is more complex for an Endpoint to be able to send multiple packets at once.



```

Welcome!
Asking controller for forwarders on this network.
Received packet
Updating Forwarding List
Setup complete, ready for User
Type 'receive' to receive packets or 'send' to send.
--> hello
Not a valid input.
Type 'receive' to receive packets or 'send' to send.
--> send
Connecting to a Forwarder
Received packet
From fone.ffnet1 : ACK
Enter the name of the container the endpoint you want to send to is on.
--> ENDA
Invalid Response: Cannot send message to self.
--> ENDB
Enter the message to send. Message cannot begin with a number.
--> 33hi
Invalid Response
--> hello
Type 'receive' to receive packets or 'send' to send.
--> receive
Connecting to this network's Forwarder/s
May receive a backlog of messages.
Received packet
From fone.ffnet1 : ACK
Connected, waiting for message.

```

Figure 7: Example of the terminal for an Endpoint. The endpoint asks and receives a list of forwarders from the controller. Then sends the message 'hello' to container ENDB and sets itself to receive. Note the error checking for the user input.

3.3 Forwarder

The forwarders functionalities are exactly as described previously in the theory section, it can handle receiving acknowledgement, connect endpoint, message, and flow control update packets. It can send connect forwarder, connect endpoint, message, and flow control request packets. All forwarders use the same hard-coded port number, 54321. A forwarder has two local variables within it, the forwarding table and a list of dropped packets. The forwarding table is a hashmap which maps the name of an endpoint to either its port number or the name of the next forwarder in a path to the destination. The list of dropped packets is a hashmap of a datagram packet to the destination name, packets are frequently added and removed to this list.

```

HashMap<String, String> forwardingTable = new HashMap<String, String>();
HashMap<DatagramPacket, String> droppedPackets
    = new HashMap<DatagramPacket, String>();

```

Listing 6: In order to facilitate flow forwarding both of these variables are necessary for the protocol.

Upon startup, the forwarder will create a list of docker networks its on and sends that list to the controller, waiting for an acknowledgement before being ready to receive any other type of packet.

```

DatagramPacket connectSend;
String val = T_CONTAINER + aliasLength + containerAlias;
int length = 1;

```

```

Enumeration<NetworkInterface> nets = NetworkInterface.getNetworkInterfaces();
for (NetworkInterface netint : Collections.list(nets))
{
    Enumeration<InetAddress> inetAddresses = netint.getInetAddresses();
    for (InetAddress inetAddress : Collections.list(inetAddresses)) {
        String address = inetAddress.toString();
        address = address.substring(0, address.length()-2);

        if (!address.equals(BRIDGE_NET_IP) && !address.equals(HOST_NET_IP))
        {
            val = val + TNETWORK +
                Integer.toString(address.length()) + address;
            length++;
        }
    }
}

connectSend= new TLVPacket(CONFORWARDER, Integer.toString(length), val)
               .toDatagramPacket();
connectSend.setSocketAddress(dstAddress);
socket.send(connectSend);
}

```

Listing 7: Part of the start method for a forwarder, using methods available in the Java.net library it's possible to get all the IP addresses of a container. The addresses are then edited to remove the last two digits to get the generic IP and not the container specific one. The default addresses are then also removed from the list before configuring the list as a forwarder list packet and sending it to the controller.

```

adriana — root@fone: /compnets/ffnet — com.docker.cli • docker start -i f...
root@fone:/compnets/ffnet# java Forwarder
Connecting to Controller
Received packet
From Controller: ACK
Received packet
Ready to forward, sending ACK to ENDA
Received packet
ENDD not in this forwarders forwarding table, informing controller of need for path.
Received packet
Received path for ENDD
Forwarding message hello from ENDA via ftwo
Received packet
Adding ENDA to forwarding table of forwarder
Informing Controller, waiting for backlogged messages to come through.
Received packet
From Controller: ACK
Sending message backlog (if any) to ENDA
Sending ACK to ENDA
Received packet
ENDDA found in forwarding table - Sending packet.
Sending ACK to ENDA
Removing Connection

```

Figure 8: Example of a forwarder running within a container. It connects to the controller, and then forwards two messages.

When an endpoint informs a forwarder it wants to receive, the forwarder informs the controller of the connection and then waits three seconds before sending previously dropped packets for that endpoint and then an acknowledgement. This waiting time is to allow time for dropped packets from other forwarders to

make it to the forwarder so it can send them before the acknowledgement. The acknowledgement after the dropped packets are sent is to inform the endpoint that the dropped packets are finished and it can stop receiving after the next message is sent.

3.4 Controller

The controllers functionalities are exactly as described previously in the theory section, it can handle receiving flow control request, forwarder list request, connect endpoint, and connect forwarder packet types. It can send acknowledgements, forwarder lists, and flow control updates. The controllers port number is hardcoded as 50000 and the container its run on must be named and aliased 'controller' as both the endpoints and forwarders have that information hardcoded.

A controller has three local variables within it, each of the Graph type, a class made specifically for this protocol. A graph is made up of an ArrayList of Links, a Link is then made of up a string for an origin, and a string for the end. The networksToF graph, which stores connections between a forwarder and an IP address it is on. This graph is used to inform endpoints which forwarders are on its network and to create forwarder to forwarder connections for the next graph which is the routing table. The routing table graph is comprised of links between forwarders and endpoints, and forwarders and forwarders on the same subnet. This graph is the heart of the protocol, as it is what is maintained and then used in order to find the path between separate endpoints. The final graph is the dropped packets graph, which stores links between forwarders and endpoints to represent that a forwarder is waiting for the path to the connected endpoint.

```
static final int DEFAULT_SRC_PORT = 50000;
static final int DEFAULT_DST_PORT = 54321;

Graph networksToF = new Graph();
Graph routingTable = new Graph();
Graph droppedPackets = new Graph();
```

Listing 8: The local variables in a controller, two final ints and three graphs. Note the empty constructor for the Graph class, all of them start empty.

The controller uses a Depth First Search (DFS) algorithm in order to find the shortest path between a forwarder and requested endpoint. This was chosen as each link is unweighted and so more complex algorithms like Dijkstra were not viable.

3.5 Packet Encoding

As mentioned previously in the report there is technically only one kind of packet, the TLV packet, which uses the packetContent class given in the java example as a base.

```
TLVPacket(String t, String length, String value) {
    type= TLV_PACKET;
    this.t = t;
    this.length = length;
    this.encoding = value;
}
```

Listing 9: The constructor for a TLV packet, despite being made up of multiple TLVs the value is a single string.

When components receive a TLV packet, in order to organize the small TLVs within the value for easy reading they will call the readEncoding method within the TLV packet class which will return the value as a String to String Hashmap. Each entry represents a single TLV, with the key being the type and the value being the value.

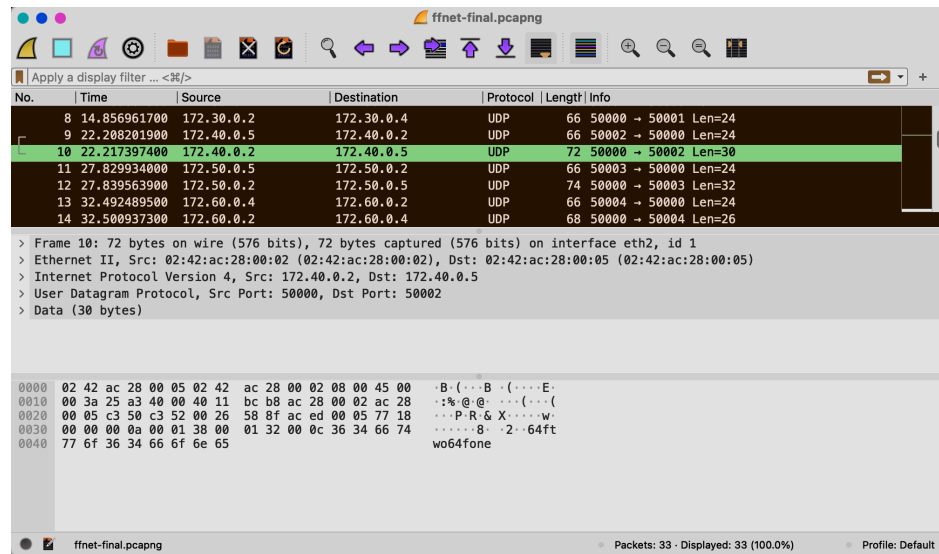


Figure 9: Example of a Forwarding List TLV packet as its recorded as a pcap file on wireshark.

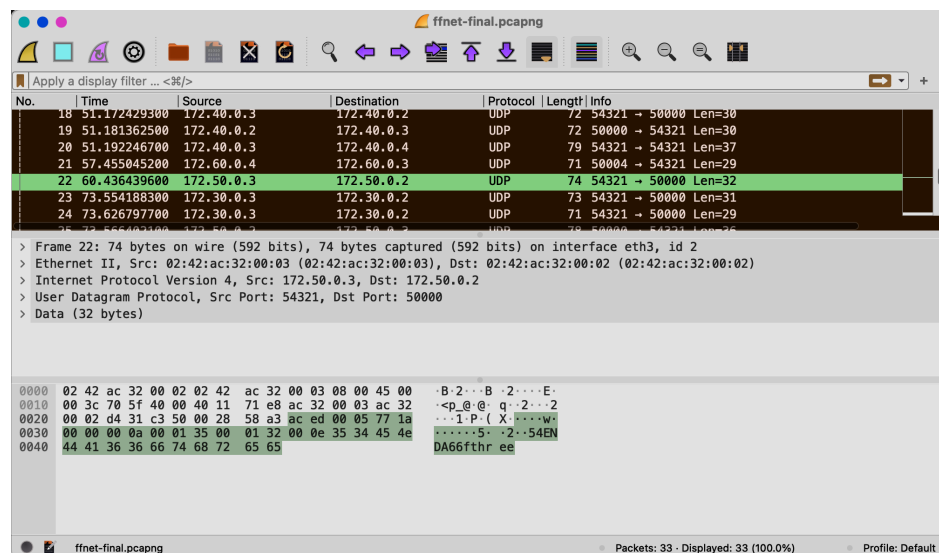


Figure 10: Example of a Flow Control Request packet as its recorded as a pcap file on wireshark.

4 Discussion

In this section I will discuss the various decisions that were made in the process of creating the implementation, and why the conclusions were made.

Using docker instead of localhost, kubernetes, or other virtual infrastructure softwares was decided due to the fact that it was what I used for the last assignment and thus had a basis from which to work off of. Also, as I use a mac for most of my assignments, docker seemed like the most lightweight option that would work for my computer.

Including User input was something I personally wanted to implement, as something I wanted to focus on with this assignment was eliminating as much hardcoding as I could. It wasn't a completely necessary feature and it did increase the amount of error handling needed as human interaction leads to human error.

As I said previously I wanted to have the least amount of hard coding possible, in order to facilitate being able to have more complex topologies with any number of forwarders and endpoints. In order to accomplish this it was necessary to have more complex connection packets between the controller and other components to send the necessary information between the two. While this did increase the amount of TLV packet types I needed, I felt like this inclusion made my protocol more unique.

TLV packets may seem like overkill, though I felt it was a better option rather than having eight different packet types each with their own class file. Also it afforded me the chance to learn about how TLV packets work and make my protocol more similar to real world examples. TLV packets also made reading packets off of wireshark simpler, as with the first integer of the encoding its possible to know exactly what the packet is meant to be.

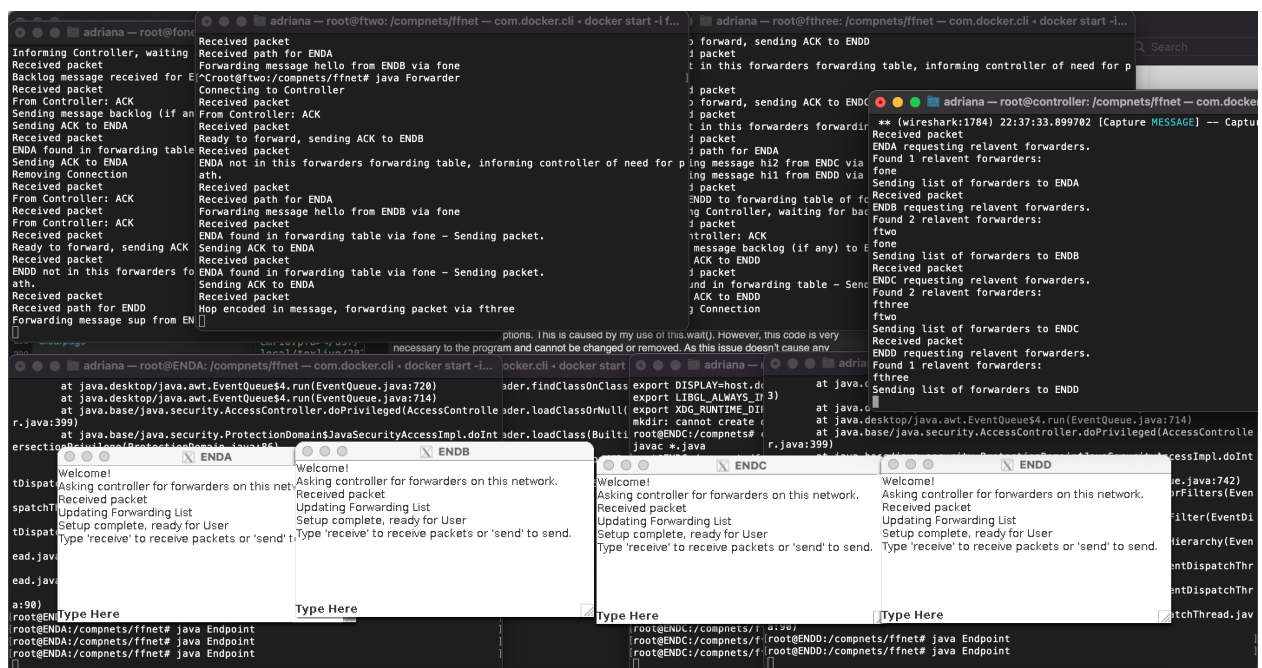


Figure 11: Screenshot showcasing a working example of the solution with 1 controller, three forwarders, and four endpoints.

Having the endpoints stop receiving after getting one message was a choice I made because of two reasons mostly. Reason one being that due to the shortcomings of the terminal class, its not possible to be able to receive terminal inputs and registers received packets at the same time, and so it was necessary to break the two functionalities (sending and receiving) up. But, reason number two, it didn't make sense to me that an

endpoint would have to be permanently stuck doing a single action the entire protocol once it decided on one. So, in order to reach a compromise, I decided to have a cap on the messages an endpoint could receive before its no longer a receiving endpoint and allowed to choose between the two options again.

5 Summary

This report has described my research, my idea, and finally my solution for a flow forwarding protocol using Docker and an Openflow approach. My final solution included three components communicating over multiple different subnetworks, sending TLV packets in order to achieve both the assignment brief and my proposed topology. The solution showcases all that I have learned about protocols and routing tables, and with that knowledge I have created a sophisticated network program that messages to be sent across networks through the use of flow paths.

6 Reflection

At the end of this assignment, I find myself thoroughly proud of my completed work. Though some might find my solution simplistic in some senses, given that this was the first time I've ever worked with multiple networks at once, the code I've accomplished here is quite well done and meets all the requirements set for the assignment. Overall I'd say I spent about 35 hours researching, coding, being in labs, and writing this report. If I had more time, or perhaps less assignments, I would have liked to add more user interaction, or at the very least try to perfect the user interaction I've included already as the current terminals are a bit clunky. Also, the way the controller and forwarder stores routing table could have been more streamlined and if I had to start the assignment over again I think I would have completely changed the way I did my forwarding tables. Lastly, one feature I had thought of but didn't end up implementing was making it so that if a new forwarder were to come online halfway through the protocol, the controller would inform the affected endpoints of the new forwarder they could connect to.

Overall, despite the occasional frustration, I learnt a lot about socket programming, TLV packets, and routing tables due to this assignment. I liked having the two smaller submissions for the project before the final as it provides useful feedback going forward, though I do feel as if the marking scheme for the interim submissions was not always as clear and a bit slow at times. Personally, though I understand the need for them, I feel as though the videos are quite time consuming and elaborate to create and I found myself spending just as much time making them as coding the actual solution. Also, despite receiving a mark for my assignment one report, there were no comments along with it and so I didn't know where I could have improved.