



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

## **CSU33031 Computer Networks**

### **Assignment #2: Flow Forwarding**

**Alice Doherty**

December 3, 2021

## **Contents**

1 Introduction.....	2
2 Theory of Topic .....	2
2.1 Internet Protocol, Ports and Sockets .....	2
2.2 User Datagram Protocol and Datagrams.....	3
2.3 Software Defined Networking .....	4
2.4 OpenFlow.....	4
3 Implementation .....	4
3.1 Overall Topology .....	4
3.2 Node.....	5
3.3 EndNode .....	6
3.4 Router.....	7
3.5 Controller .....	9
3.6 Packet and Header Design .....	12
3.6.1 User Message Packets.....	13
3.6.2 Router and Controller Packets .....	14
3.7 Docker.....	16
4 Discussion .....	16
4.1 Header Design.....	16
4.2 Other Design Decisions .....	17
4.3 Features .....	17
5 Summary.....	18
6 Reflection.....	20
7 References.....	20

# **1 Introduction**

The purpose of this assignment was to design a protocol that makes forwarding decisions based on an overall table of routes maintained by a central controller. The controller sends each router in the network an individual forwarding table so that each router knows where to forward incoming packets. The user can specify the destination and message to be sent across the network through an application running at each endpoint. When a message is sent from one endpoint, it will be forwarded through a number of routers and will arrive at another endpoint depending on the destination specified.

This report will first outline the theory that underpins this assignment. The specific details of the implementation will next be covered, followed by a discussion on the decisions made when designing this protocol. Finally, a summary will be provided, as well as a reflection on the overall assignment.

## **2 Theory of Topic**

This section will cover the essential theory that underpins the protocol and its implementation. The motivation behind this section is so that the reader does not need to make assumptions about the concepts used and that any reader should be able to follow what is covered in this report without referring to other sources. Specifically, this section will cover the Internet Protocol, the User Datagram Protocol, Software Defined Networking, and OpenFlow. Note that sections 2.1 and 2.2 are taken from the report written for the first assignment as the theory is relevant for both.

### **2.1 Internet Protocol, Ports and Sockets**

When we talk about the Internet Protocol (IP), we are referring to the network layer in the Open Systems Interconnection (OSI) model. IP is used to deliver packets from a source host to a destination host which are identified using IP addresses. IP addresses are a unique and universal sequence of 32 or 128 bits depending on whether IPv4 or IPv6 are used. For this assignment,

each container that is used to run each end node, router, and controller has its own IP address which is used to identify where to send packets to.

A port represents a communication endpoint. Whereas an IP address is used to identify a host in a network, a port number is used to identify a process or service. In this solution, the port number is always 51510 and is used to identify my protocol and the forwarding service.

A socket is used as an endpoint for sending and receiving packets across a network. In this implementation, each end node, router, and controller instance has its own socket. This is defined using the `DatagramSocket` class, from the `java.net` library [1], which listens for incoming packets and sits on port 51510. This implementation also uses the `InetSocketAddress` class to represent the destination addresses of packets. An `InetSocketAddress` represents an IP socket address and takes in an IP address and a port number as parameters.

## 2.2 User Datagram Protocol and Datagrams

The User Datagram Protocol (UDP) is a protocol that runs on top of IP and is part of the transport layer in the OSI model. It is connectionless and is mainly used for low-latency and loss-tolerating connections, due to its unreliability. No flow or error control is built-in so it is better suited for applications that can provide their own. Packets sent over UDP are referred to as datagrams. A UDP datagram consists of a header and the data section; see figure 1. The header is relatively small at 8 bytes and contains the source and destination port number, the length of the datagram, and a checksum field.



Figure 1: The basic structure of a UDP packet, or datagram. The header is made up of four sections: the source port number, the destination port number, the length of the packet, and an optional checksum field. The rest of the datagram consists of the payload to be transported.

For this protocol implementation, only the data section is relevant and the custom header and payload will be sent in this section. The DatagramPacket class in the java.net library is used to represent these packets.

## **2.3 Software Defined Networking**

Software Defined Networking (SDN) is an architecture that separates the network control layer and the forwarding layer. With SDN, you can have a number of applications that can talk to a control layer that dictate how network devices are configured. Rather than each device configuring itself, it is handled by the one or more controllers in the control layer. Some of the benefits of SDN include improved network performance, management, and flexibility [2].

## **2.4 OpenFlow**

OpenFlow is one of the first SDN standards and is a communications protocol that allows the control layer to interact with the forwarding layers of multiple routers in a network [3]. OpenFlow is relevant to this assignment, as the protocol designed for this assignment is heavily influenced by the functionality offered by OpenFlow, albeit at a much more basic level. At its most basic level, an OpenFlow switch, or router, consists of one or more flow tables which tells the router what to do with an incoming packet. The switch communicates with a controller which can update the switch's flow table depending on what is required. There are a number of different packet types defined by the OpenFlow standard and the ones relevant to this protocol implementation will be outlined in section 3.6.2.

# **3 Implementation**

This next section of the report will focus on the implementation details of the protocol designed. The protocol was implemented in Java and the network elements were simulated using Docker. First, a general overview of the network topology will be given, followed by a breakdown of each of the Java classes involved in the implementation, and finally a detailed explanation of the header and packet layout will be outlined.

## **3.1 Overall Topology**

There are three main network elements: the end nodes, the routers, and the controller. The end nodes, described as end-users in figure 2, are part of the network that interact with the user, and is made up of an application and a forwarding service. The application that runs on the end nodes gets input from the user and passes the requested message onto the forwarding service. In other words, the end node gets user input and then forwards it onto the first router in the network. There are a number of routers within the network and depending on the overall forwarding information the controller has, the message packet will be forwarded through a certain path of routers, eventually reaching another end node. The controller has an overall view of the network and can communicate with any of the routers, sending them up to date forwarding information.

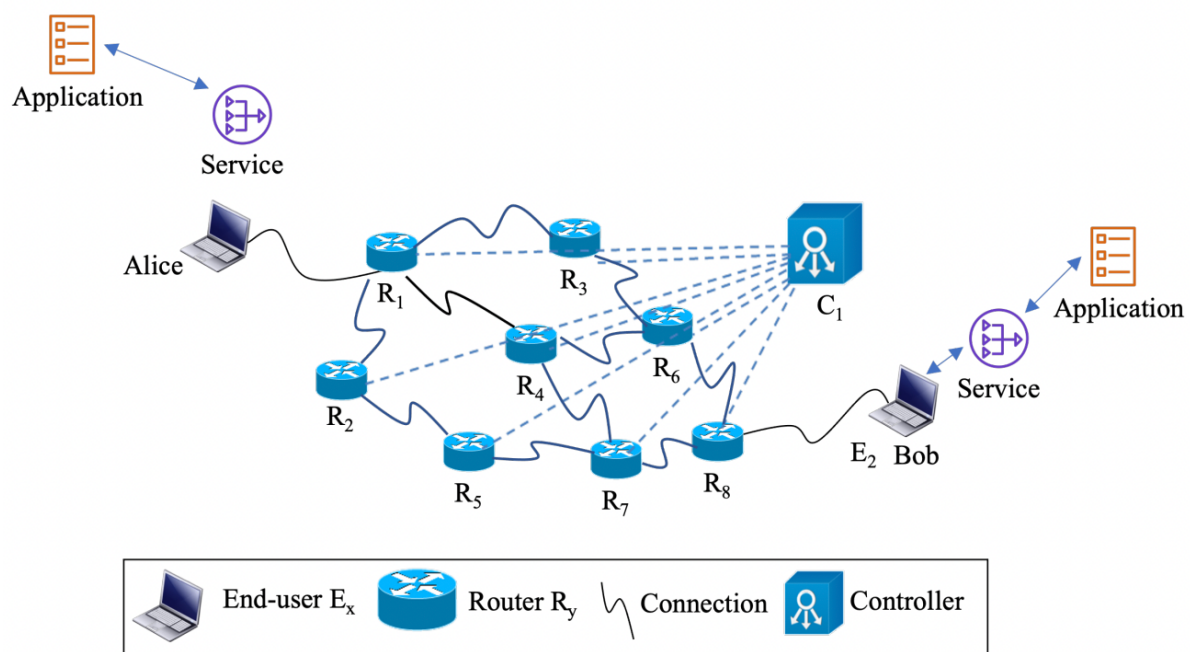


Figure 2: An end node, or end-user, runs an application that takes in the destination and message a user wishes to send across the network. The message packet then gets sent into the forwarding service that starts at R1. The packet will then be sent through a number of routers before it reaches another end node, or its destination. The controller has an overall view of the network and sends each router up to date forwarding information. Note that this diagram is taken directly from the assignment description.

### 3.2 Node

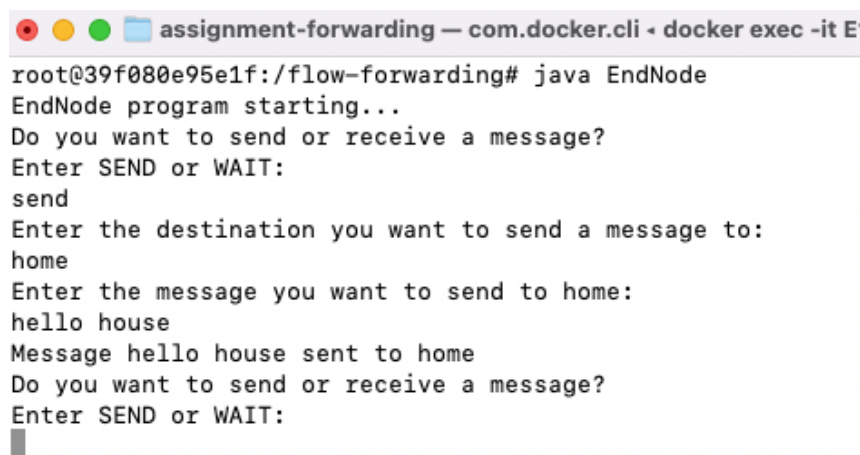
My implementation consists of four Java classes, Node, EndNode, Router, and Controller. The abstract Node class was provided on Blackboard and the EndNode, Router, and Controller all extend this class. The main functionality provided by the Node class is the Listener thread

which continuously listens for incoming packets on a datagram socket. When a packet is received, it calls `onReceipt`. Depending on the class, there is a different implementation of this method which determines what will happen when that component receives a packet. The `Node` class also contains some basic helper functions and defines constants used by its three subclasses. For example, the constants used for different OpenFlow packet types and the protocol's port number are defined here.

### 3.3 EndNode

In this implementation the `EndNode` class refers to the application that an end user interacts with to either send a message or wait for a message through the network. When it is run, the user will first be prompted to enter whether the application should send a message, or wait to receive a message. If the user selects to wait for a message, the application will simply sit and wait until a packet arrives to it, if it is defined as the end node for a certain destination string. Otherwise, if the user decides to send a message they will be prompted to enter the destination that they want to send a message to. Note that there is no error handling at this stage that will catch whether a destination exists or not. This is because the router deals with this and this will be covered in section 3.4. Next the user will be asked to input the message they want to send.

The destination and message will be put into a byte array of type "Network ID" and will get forwarded to the first router in the network. The exact contents of the packet and its design will be discussed in section 3.6.1.

A screenshot of a terminal window titled "assignment-forwarding — com.docker.cli • docker exec -it E". The terminal shows the execution of the `java EndNode` command. The output is as follows:

```
root@39f080e95e1f:/flow-forwarding# java EndNode
EndNode program starting...
Do you want to send or receive a message?
Enter SEND or WAIT:
send
Enter the destination you want to send a message to:
home
Enter the message you want to send to home:
hello house
Message hello house sent to home
Do you want to send or receive a message?
Enter SEND or WAIT:
█
```

Figure 3: A sample interaction with the `EndNode` class. In this example, the user first enters that they wish to send a message through the network. They then set the destination of the message to "home",

and send the message “hello house”. They are then prompted again if they want to send or wait for another message.



```
assignment-forwarding — com.docker.cli ◀ docker es
root@ddffca431efc:/flow-forwarding# java EndNode
EndNode program starting...
Do you want to send or receive a message?
Enter SEND or WAIT:
wait
Waiting for messages...
Received message: hello house
```

Figure 4: This screenshot shows the EndNode, set as the destination for “home”, receiving the message sent in figure 3. Note that this EndNode can also send messages but in this example it was set to wait.

### 3.4 Router

The basic functionality of the router is to receive incoming packets, look at the header, and depending on the destination encoded in the header forward the packet to the next router or end node. Note that in OpenFlow, these network components are referred to as switches but in this report the terms switches and router can be used interchangeably.

Each router has its own forwarding table. This forwarding table is what tells the router where the next hop for the packet it is handling is. It consists of three columns. The first is the destination, this is the string that the user specified as the final destination for the packet. The next column represents which network element the packet came in from, and the final column represents what network element to next forward the packet to; see table 1. The forwarding table is a 2D array of strings and the “in” and “out” columns store the name of the router or end node’s corresponding Docker container.

Destination	In	Out
trinity	E1	R2
home	E1	R3
lab	E3	R4

Table 1: An sample forwarding table for a router in the network. For example, if the router receives a packet from E1 and the destination “trinity”, then it should forward that packet to R2. The string “R2” refers to the name of the container for that network element, and therefore can be used as the address of the packet.

When the router program is run, it first prints out its current forwarding table, which should be empty as it has made no contact with the controller yet and therefore has no knowledge of the network. The next thing the router will do is send a “Hello” packet to the controller. This initial hello serves as a simple introduction between the router and the controller. When the controller receives this hello packet it responds by sending the forwarding table relating to that router back to the router in the form of a “Flow Mod” packet. These two packets will be covered in more detail in section 3.6.2. When the router receives this it will update its forwarding table accordingly.

```
assignment-forwarding — com.docker.cli • docl
root@6c377383c3e5:/flow-forwarding# java Router
Router program starting...
Current Forwarding Table:
DEST      | IN  | OUT
-----
null      | null | null

Hello sent to controller
Received request to update forwarding table
Current Forwarding Table:
DEST      | IN  | OUT
-----
trinity   | E1  | R2
home      | E1  | R3
lab       | E3  | R4
```

Figure 5: A sample view of the router running. Notice that when the router code is first run it prints out its current forwarding table which is empty. It then sends a hello to the controller and receives back a request to update its forwarding table. The updated forwarding table is then printed out.

At this stage the router now has information about the routes in the network and where to forward incoming packets. When a router receives a message packet, which is identified if the packet has the type “Network ID”, see section 3.6.1, it will first go to look up what network element the next hop is. To do this, it will extract the destination of the packet, for example “trinity”. It will then search through its forwarding table for a row whose destination, the first column, is set to “trinity”, if there is a match it will then check that the element the packet came from, matches the “in” column in the table. The router can establish who sent the packet to it by extracting the Docker container name from the DatagramPacket; see listing 1. If both the destination and “in” values match a row it will then take the value stored in the “out” column for that row which is the container name for the next hop. Then it simply sets the



InetSocketAddress, using the Docker container name as the IP address and the default port 51510, and sends the packet to that address. If the destination cannot be found in the router's forwarding table the router will forward the packet onto the controller to see whether it has any information on the destination. This feature will be covered in section 3.5.

```
private String getNextHop(DatagramPacket packet) {
    String destination = getDestination(packet);

    // Gets packet's source address (container name) and trims
    // e.g trims "E1.assignment-forwarding_flow-forwarding" to "E1"
    String source = packet.getAddress().getHostName().substring(0,2);

    System.out.println("The final destination of this packet is: " +
destination);
    System.out.println("Packet came from container: " + source);

    for(int i = 0; i < forwardingTable.length; i++) {
        if(destination.equals(forwardingTable[i][DEST])) {
            if(forwardingTable[i][IN].equals(source)) {
                return forwardingTable[i][OUT];
            }
        }
    }
    return "error";
}
```

Listing 1: Code from Router.java that searches through its forwarding table for a row that matches the packets destination, and if the network element in the "in" column matches the source, then the container name in the "out" column is returned.

### 3.5 Controller

The controller is one of the key network components when it comes to SDN. The controller, as its name suggests, controls the network elements which in this protocol just refers to the routers. It does this by initialising the flow tables of each of the routers and updating route destinations as the network changes. The controller contains an overall view of the network and has a table, represented as a 2D string array, that contains all the information about the routes in a network. This table has five columns which outline the source and the hops for a packet being sent to a specific destination; see listing 2.

```
// Preconfig Info Layout:
// {DEST, SRC, ROUTER, IN, OUT}
String[][] preconfigInfo = {
    {"trinity", "E1", "R1", "E1", "R2"},
    {"trinity", "E1", "R2", "R1", "R4"},
    {"trinity", "E1", "R4", "R2", "E4"},
    {"home", "E1", "R1", "E1", "R3"},
    {"home", "E1", "R3", "R1", "E2"},
    {"lab", "E3", "R1", "E3", "R4"},
    {"lab", "E3", "R4", "R1", "R3"},
    {"lab", "E3", "R3", "R4", "E1"}
};
```

Listing 2: This is the overall view of the routes in the network that the controller has. The first column represents the destination of the packet. The second column represents the network element that was the original source of the packet. The third column specifies which router the row is relevant for. The final two columns then represent where the packet will come into the router from, and where the router should forward it to next. For example, the final row in the table tells us that for the destination “lab”, where the source of the packet is “E3”, if “R3” receives a packet from “R4” with the destination “lab”, then it should forward it onto “E1”.

When the controller starts up, it will print out its current view of the network. It will then sit and wait until it receives a packet. The first packet it will receive will be a “Hello” packet from one of the routers. When the controller receives this, it will respond by sending the forwarding table for that router back. This exchange can be thought of as the router registering with the controller and getting initialised. As mentioned, the controller has one large table, with the preconfiguration information, but rather than sending the whole table to the router, it will extract only the information relevant to that specific router. This is best explained with an example. The router “R2” starts up and sends a “Hello” packet to the controller. The controller can then extract the router’s name, or its container name, “R2” using built in functions from the package `java.net.InetAddress` mentioned previously in this report. The controller then loops through the 2D array with the preconfiguration information and searches for rows where the third column contains “R2”. If there is a match, the destination address, the router in, and the router out are added to another table, which is represented as an `ArrayList` of strings. This table then gets sent to the router who can then process it and update its own flow table with the destination, router in, and router out details.

```
assignment-forwarding — com.docker.cli • docker exec -it
root@3786dd96fb19:/flow-forwarding# java Controller
Controller program starting...
The current view of the network is:
DEST      | SRC   | ROUTER | IN    | OUT
trinity   | E1    | R1     | E1    | R2
trinity   | E1    | R2     | R1    | R4
trinity   | E1    | R4     | R2    | E4
home      | E1    | R1     | E1    | R3
home      | E1    | R3     | R1    | E2
lab       | E3    | R1     | E3    | R4
lab       | E3    | R4     | R1    | R3
lab       | E3    | R3     | R4    | E1
Received hello from router R3
Updated forwarding table has been sent to R3
Received hello from router R1
Updated forwarding table has been sent to R1
```

Figure 6: The overall view of the network that the controller has can be seen printed to the terminal. This controller then receives a "Hello" packet from R3 and R1 and sends them both back the latest forwarding information relevant to them.

Another feature implemented in this protocol is the dropping of packets if the destination cannot be resolved. For example, the end user may say that they want to send a message to the destination "hamilton", this packet then gets forwarded onto the first router in the network. The router will then look up that destination string in its forwarding table, as outlined in section 3.4. When it finds that there is no match, it will contact the controller and ask if it knows where to next forward the packet. It does this by taking the received packet, changing its type to "Packet In" and then sending that onto the controller. When the controller receives a packet of type "Packet In", it knows it is a request from the router to try and establish the next hop. The controller then searches its table for the destination and if it cannot be found the packet is dropped, otherwise the controller will send the router an updated forwarding table. The reason why a packet of type "Packet In" was used is because in OpenFlow it is used to transfer the control of a packet from the router to the controller if the router does not know what to do with it.

```

assignment-forwarding — com.docker.cli • docker exec -it R1 /bin/t
root@6c377383c3e5:/flow-forwarding# java Router
Router program starting...
Current Forwarding Table:
DEST      | IN  | OUT
-----
null      | null | null

Hello sent to controller
Received request to update forwarding table
Current Forwarding Table:
DEST      | IN  | OUT
-----
trinity   | E1  | R2
home      | E1  | R3
lab       | E3  | R4

The final destination of this packet is: home
Packet came from container: E1
Next hop for packet is: R3
Message forwarded.
The final destination of this packet is: unknown
Packet came from container: E1
Next hop cannot be established - forwarding packet to controller

```

Figure 7: This router receives a packet with the destination set to "unknown". It cannot find any information relating to this destination in its forwarding table and forwards the control of the packet to the controller.

```

assignment-forwarding — com.docker.cli • docker exec -it co
root@3786dd96fb19:/flow-forwarding# java Controller
Controller program starting...
The current view of the network is:
DEST      | SRC  | ROUTER  | IN  | OUT
-----
trinity   | E1   | R1      | E1  | R2
trinity   | E1   | R2      | R1  | R4
trinity   | E1   | R4      | R2  | E4
home      | E1   | R1      | E1  | R3
home      | E1   | R3      | R1  | E2
lab       | E3   | R1      | E3  | R4
lab       | E3   | R4      | R1  | R3
lab       | E3   | R3      | R4  | E1

Received hello from router R3
Updated forwarding table has been sent to R3
Received hello from router R1
Updated forwarding table has been sent to R1
Received packet with unknown next hop from R1
Searching for the destination...
Destination does not exist - packet has been dropped

```

Figure 8: The controller receives the packet with the unknown destination from figure 7 and searches for any information about that destination. However, no information is found and the packet is dropped.

### 3.6 Packet and Header Design

The previous sections have described the overall implementation of each of the main network elements involved in this protocol. This section will now provide a detailed explanation of the different packet types and the header design used.

The packet types involved in this protocol can be divided into two main types. The first type relates to the packets that carry the user’s message from one end node to another. The second type relates to packets being sent between routers and the controller.

### 3.6.1 User Message Packets

This subsection will cover the design of the packets used to send a user’s message from one end node to another. The header of these packets are encoded as type-length-value (TLV) format. This means that the first byte encodes the packet type, the second byte encodes the length of the value, and the following bytes represent the value. In this case, the value is the destination. The only packet type supported for these messages is NETWORK\_ID, which is encoded as 1. The rest of these packets carry the payload, or the message the user wants to send across the network; see table 2.

Byte (if fixed)	0	1		
Content	Type	Length	Value	Payload
Example	1	7	trinity	hello world

Table 2: This table shows the packet structure of a user message being sent across the network. The first byte represents the type and the second byte represents the length of the value. The value is the destination and the payload is the message the user wants to send. In this example, the packet is of type “NETWORK\_ID”, with the destination “trinity”, which is 7 bytes long. The message being sent is “hello world”.

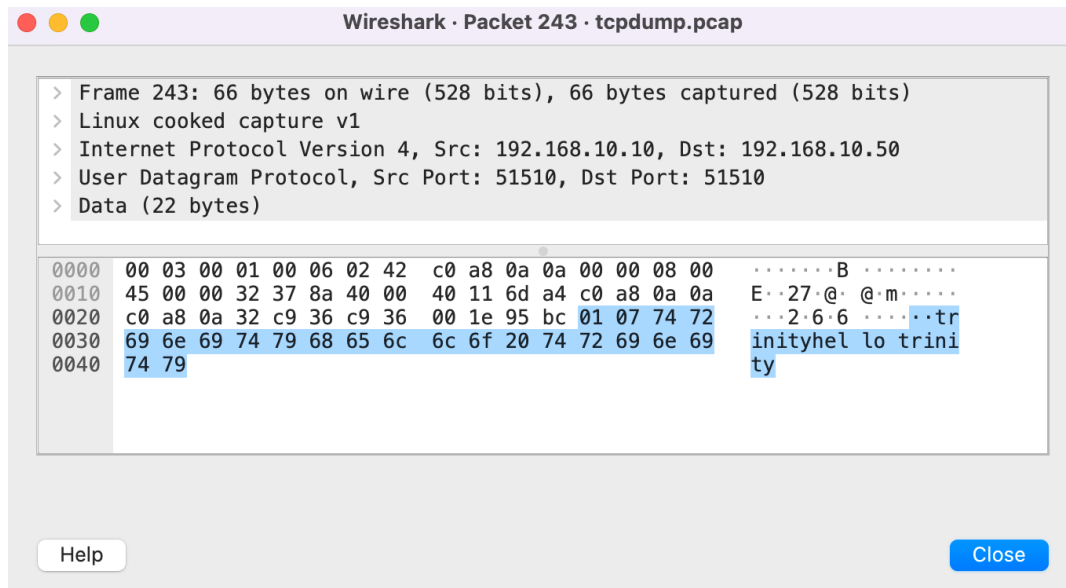


Figure 9: A sample message packet captured in Wireshark. Notice that the first byte is 1, indicating that it is a message packet carrying a Network ID. The second byte is 7 which is the length of the value "trinity". Following "trinity", the payload "hello trinity" can be observed.

### 3.6.2 Router and Controller Packets

The packets sent between the controller and routers will next be described. There are three different types of these messages implemented in this protocol: Hello packets, Flow Mod packets, and Packet In packets. In OpenFlow, these types of packets are split into three subtypes: controller-to-switch messages, asynchronous messages, and symmetric messages. There is an example of each one of these in this protocol.

The header design of each of these packets is incredibly simplistic and consists only of a single byte. In other words, the header of these packets consist only of the first byte of the packet. The value of the byte determines the type of packet. Note that some of the values used may seem arbitrary but they are taken directly from the OpenFlow standard, and a list of all the values can be seen in Node.java.

```
protected static final byte OFPT_HELLO = 0;
protected static final byte OFPT_PACKET_IN = 10;
protected static final byte OFPT_FLOW_MOD = 14;
```

Listing 3: The constants used for each of the different OpenFlow packet types. Note that the numbers seem random but they are taken directly from the OpenFlow standard.

## Hello Message

This is a message sent from the router to the controller, or vice versa as an introduction or keep-alive message. In OpenFlow these packets are sent both ways and are therefore categorised as a symmetric message. However, for simplicity in this protocol it is only sent from the router to the controller when a router starts up. This packet is simply made up of only the header, and consists of one byte set to 0.

0000	00 03 00 01 00 06 02 42	c0 a8 0a 32 00 00 08 00	.....B ...2....
0010	45 00 00 1d de 4e 40 00	40 11 c6 a4 c0 a8 0a 32	E...N@ @.....2
0020	c0 a8 0a 5a c9 36 c9 36	00 09 95 f7 00	...Z·6·6 ....·

Figure 10: A "Hello" packet captured in Wireshark. Note that it consists only of one byte which is set to 0.

## Flow Mod Message

This is a message sent from the controller to a router and is used to add, delete, or update the router's flow table. In OpenFlow, this is categorised as a controller-to-switch message. In this protocol, it is used to identify packets sent from the controller to a router with the router's forwarding table information. Again, the header of this packet is one byte and is set to 14. The rest of this packet is the payload which can vary in length depending on the length of the forwarding table being sent.

0000	00 03 00 01 00 06 02 42	c0 a8 0a 5a 00 00 08 00	.....B ...Z....
0010	45 00 00 47 dd 3b 40 00	40 11 c7 8d c0 a8 0a 5a	E·G·;@ @.....Z
0020	c0 a8 0a 32 c9 36 c9 36	00 33 96 21 0e 74 72 69	...2·6·6 ·3·!·tri
0030	6e 69 74 79 2c 20 45 31	2c 20 52 32 2c 20 68 6f	nity, E1 , R2, ho
0040	6d 65 2c 20 45 31 2c 20	52 33 2c 20 6c 61 62 2c	me, E1, R3, lab,
0050	20 45 33 2c 20 52 34		E3, R4

Figure 11: A "Flow Mod" packet captured in Wireshark. Note that the first byte is set to 14. The rest of the packet is the payload that consists of the forwarding information being sent to the router.

## Packet In Message

This is a message sent from the router to the controller to transfer the control of the packet to the controller. In OpenFlow, this type of message is categorised as an asynchronous message as it is a message from the router updating the controller of network events. As covered in section 3.5, this packet is used when a router cannot resolve the destination and forwards the packet to the controller so that it can look up the next hop. This packet is simply the original

message packet described in section 3.6.1, but with the first byte changed from 1 to 10, in other words changing it from type “Network ID” to type “Packet In”.

0000	c9	36	c9	36	fd	f0	e5	b1	0a	03	75	63	64	74	65	73	·6·6····	··ucdtes
0010	74	20	75	6e	6b	6e	6f	77	6e	20	64	65	73	74	69	6e	t unknow	n destin
0020	61	74	69	6f	6e	00	00	00	00	00	00	00	00	00	00	00	ation···	········
0030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	········	········
0040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	········	········
0050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	········	········

Figure 12: A "Packet In" packet captured in Wireshark. Note that the first byte is set to 10 to indicate. The second byte is the length of the destination, "ucd", which is 3. In this screenshot the packet was set to the maximum size which is why it is buffered with the 00s. This has since been fixed but due to time constraints, I didn't have time to recapture the packet.

### 3.7 Docker

Docker [4] is used to run my solution, with each end node, router, and controller instance being run inside its own Docker container. The main advantage of this is that each container will have its own unique IP address allowing the different network elements to be simulated realistically. To create the network and containers, Docker Compose was used, and the YAML file containing the configurations can be found in the code bundle submitted to Blackboard. As an example, four end nodes, four routers, and one controller container were defined. However, my solution is able to support many more routers and end nodes. The only constraint is that only one controller is supported with the current implementation. To capture the network traffic in a PCAP file, a docker container with the image “tcpdump” [5] was used.

## 4 Discussion

The previous section touched on the actual implementation of the assignment protocol, this section will now go into a discussion on the design decisions made and why the protocol was implemented how it was.

### 4.1 Header Design

Compared to the previous assignment, the header design for this protocol was much more simplistic. The first reason for this was because the desired header layout for the message packets was described in the assignment description, which outlined that it should be encoded



as type-length-value format. This made sense for the purpose of the protocol as destinations of varying length need to be encoded in the header, so storing the length of the value was a good way to achieve this.

However, due to time constraints my implementation does not support a combination of network ID. For example, the assignment description gave the example “trinity.scss”, where both “trinity” and “scss” were network IDs which could be combined. My current protocol implementation can support simply if the destination was set to “trinity.scss” and there was a destination called “trinity.scss” in the controller’s information, however, the protocol would have no notion that “trinity” and “scss” are two different IDs related to one another. If there was more time to work on this more support would have been implemented for destinations consisting of patterns such as these.

In terms of the header design for packets sent between routers and the controller. I decided to keep these as simple as possible and just have the first byte represent the packet type, and the rest of the packet be interpreted as payload. For the purpose of this assignment, I think this is more than enough, and by designing a more complicated header it would only increase the overhead without any real performance benefits.

## **4.2 Other Design Decisions**

Another design decision made was to hardcode the routes in the network into the controller. I initially was planning on implementing a shortest path algorithm, such as Dijkstra, so that once the controller had a view of the network elements it would calculate the shortest path between the end nodes itself. However, the advice from the lecturer, Stefan Weber, was to not to this as it is not relevant for the purpose of this assignment and that having the controller have predefined routes meets the requirements.

## **4.3 Features**

This subsection will quickly touch on some of the main features of this protocol that go above the basic requirements set out in the assignment description.

First, when a router does not know where next to forward a packet, it contacts the controller to try and resolve the destination. This results in either the packet being dropped or the router's forwarding table being updated. This could have just been implemented more simplistically by the router dropping the packet if it cannot find the next hop but I think having the router contact the controller is much more suitable and mimics the feature in OpenFlow. It also allows for more expansion if I were to continue working on this in the future.

Another feature is that the implementation supports multiple destinations and that different router paths are taken depending on the destination. Each end node also can both send and wait for a packet and is not limited to one function, depending on where the user wants to send a message to.

## 5 Summary

In summary, the purpose of this assignment was to design a protocol that allows a central controller to communicate with a number of routers, giving them up to date forwarding information so that messages can be sent from one end node to another.

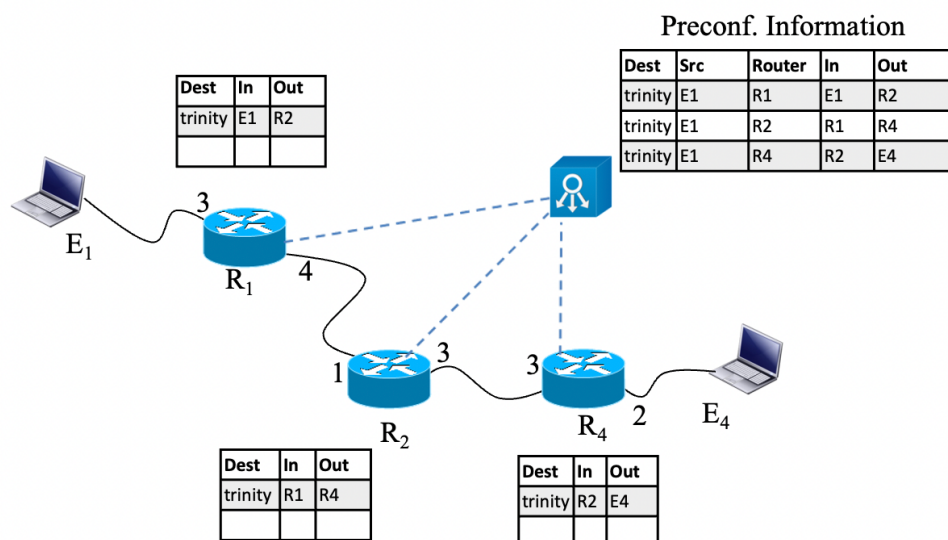


Figure 13: This diagram shows a controller with sample preconfiguration route information. Each router then gets sent the forwarding information relevant to them by the controller. E1 can then send a message to E4, using the destination "trinity". Note that for simplicity this diagram has only one destination but this protocol can support multiple.

## End Node

The end node, is the part of the protocol that communicates with the user. It either can send or wait for messages. The user can input the destination and message they want to transport through the network. This message is then sent in a packet with type “Network ID”. Multiple end nodes are supported in this protocol.

## Router

When a router starts up it has no forwarding information and sends a “Hello” packet to the controller. The controller responds with an up to date forwarding table. After the user enters the message they want to send and its destination the packet gets sent to the first router in the network. The router then looks up the destination in its forwarding information to decide where to next send the packet. If the destination cannot be established the packet is forwarded to the control of the controller in the form of a “Packet In” message.

## Controller

The controller has an overall view of the network. When a router starts up it sends the router its relevant forwarding information. If a router does not know where to forward a packet, it can contact the controller who will look up the destination. If the destination cannot be resolved the packet is dropped. There is only one central controller in this protocol.

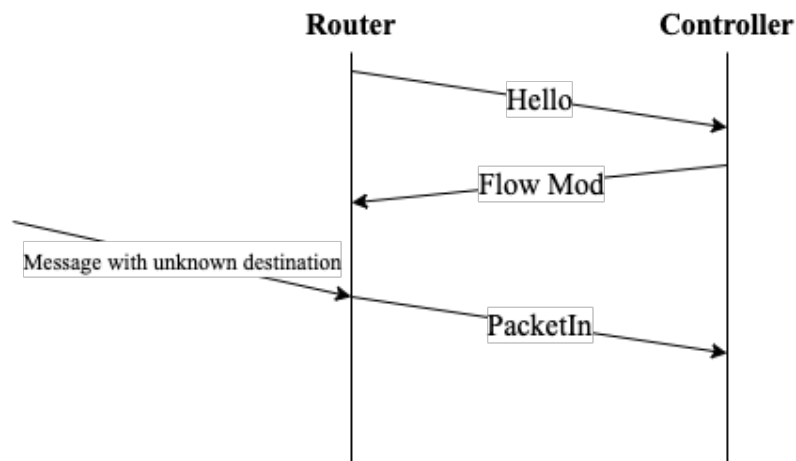


Figure 14: This flow diagram shows the communication between the routers and the controller in the network. When the router starts it sends a "Hello" packet to the controller which responds with a "Flow Mod". If a message with an unknown destination is received by a router, the packet will get forwarded onto the controller as a “Packet In” message, where it will either get dropped or the router will receive an updated table.

## 6 Reflection

Overall I am happy with the protocol I have implemented for this assignment. I gained a greater understanding of how network elements communicate and how routers forward network traffic. As well as that, I also was able to look more into OpenFlow and SDN. Following the previous assignment, this implementation was a lot quicker getting off the ground as I already had learned much of the foundational knowledge, such as how to use Docker and send UDP packets. If I was given more time for this assignment I would have liked to add more features, like those mentioned in section 4.1, but due to the nature of college towards the end of term and the current climate I am very happy with the features I was able to implement.

## 7 References

- [1] Oracle. Package java.net. <https://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html>, visited Nov 2021.
- [2] Cisco. Software-Defined Networking. <https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html>, visited Nov 2021
- [3] Overlaid. OpenFlow – Basic Concepts and Theory. <https://overlaid.net/2017/02/15/openflow-basic-concepts-and-theory/>, visited Nov 2021.
- [4] Docker. Docker project page. <https://www.docker.com/>, visited Nov 2021.
- [5] Docker Hub. kaazing/tcpdump Docker Image. <https://hub.docker.com/r/kaazing/tcpdump>, visited Nov 2021.