

# CS2031 Telecommunications II

## Assignment #2: OpenFlow

Lexes Jan Mantiquilla - -1

December 2, 2019

### Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Problem Statement . . . . .	3
1.2	Approach . . . . .	3
<b>2</b>	<b>Overall Design</b>	<b>3</b>
2.1	Features . . . . .	3
2.1.1	Controller . . . . .	3
2.1.2	Switch . . . . .	4
2.1.3	End-Node . . . . .	4
2.2	OpenFlow . . . . .	4
2.3	Design of the Protocol . . . . .	5
2.4	Packet Descriptions . . . . .	5
<b>3</b>	<b>Implementation</b>	<b>5</b>
3.1	Packet Classes . . . . .	6
3.1.1	FeatureRequestPacketContent . . . . .	7
3.1.2	FeatureResultPacketContent . . . . .	8
3.1.3	FlowModPacketContent . . . . .	9
3.1.4	HelloPacketContent . . . . .	10
3.1.5	PacketInPacketContent . . . . .	11
3.1.6	PayloadPacketContent . . . . .	12
3.2	Node Classes . . . . .	12
3.2.1	Controller . . . . .	14
3.2.2	EndNode . . . . .	15
3.2.3	Node . . . . .	15
3.2.4	Switch . . . . .	16
3.3	Path Finding class . . . . .	16
3.3.1	Graph . . . . .	17
3.4	Miscellaneous Classes . . . . .	17
3.4.1	Terminal . . . . .	17
3.4.2	Tokenizer . . . . .	17

<b>4</b>	<b>Advantages and Disadvantages</b>	<b>17</b>
4.1	Advantages . . . . .	17
4.2	Disadvantages . . . . .	18
<b>5</b>	<b>Program Usage</b>	<b>18</b>
<b>6</b>	<b>Reflection</b>	<b>18</b>

# 1 Introduction

## 1.1 Problem Statement

In this assignment we are tasked to design a protocol which implements our own version of the OpenFlow software defined networking standard. We are tasked to create a protocol consisting of **End-Nodes**, **Switches** and a **Controller** in which the **End-Nodes** can send messages between each other. However we must emulate the OpenFlow standard which means that we must send the payload packet to a switch which will then forward the packet to another **Switch** until the **Switch** forwards it to the destination **End-Node**.

## 1.2 Approach

Using my knowledge from the previous assignment, I was able to immediately setup the skeleton code for many classes that I believed I would need.

When designing the packets, I used the website [flowgrammable.org](http://flowgrammable.org) for inspiration and helped me greatly to understand how the OpenFlow standard is actually implemented. I also researched on various websites on the OpenFlow standard to help me understand and grasp the general concept of software defined networks.

When designing the protocol, I attempted to implement what I understood about the OpenFlow standard. Upon finishing the basic implementation, I then researched about graphs and graph path finding algorithms, specifically Dijkstra's algorithm.

# 2 Overall Design

This section contains the features implemented, my understanding of the OpenFlow standard and the protocol I have decided to implement to solve the problem statement.

## 2.1 Features

I have implemented the required features described in the problem statement and the Link State Routing as a part of the advanced implementation features. The features that I have implemented are the following:

### 2.1.1 Controller

- Ability to reply to hello packet and issue a feature request.
- Ability to accept packet in packets.
- Ability to respond to packet in packets with a flow mod packet.
- Ability to find a path from one **End-Node** to another with any valid network configuration.

### 2.1.2 Switch

- Ability to make contact with the **Controller** and reply to a feature request packet.
- Sends out packet in packets when the flow table does not contain an entry for the final packet destination
- Ability to receive and update the flow tables according to the flow mod packet received.
- Ability to receive payload packets from the end nodes.
- Ability to forward packets based on a flow table.

### 2.1.3 End-Node

- Can send messages to another **End-Node**.
- Can receive messages forwarded by a **Switch**.
- Implements a simple command system.
- Can run commands such as “clear”, and “help”.

## 2.2 OpenFlow

A software defined network is an approach to network management that allows one to program network configurations and abstract away the control plane from the data plane. This means that the data moving specifications will still function as before with but there would be the ability to control and program the all the network devices from a central control unit.

OpenFlow is a software defined network standard which has three basic components in the network. The **Switch**, the **Controller** and the **End-Node**.

The **Controller** is the node which does most of the work. The **Controller** is connected to all the **Switches**. When a **Switch** does not know where to forward the incoming packet i.e. a table miss occurred, the switch sends a packet in packet to the **Controller**. The **Controller** is able to program the **Switches**. In OpenFlow, there are many functions that the **Controller** may perform such as add and delete entries in the **Switch's** flow table. Such a packet to change the **Switch's** flow table is called a flow mod packet. Both the **Controller** and the **Switch** may send hello packets to each other signalling their online status.

The **Controller** may also ask the **Switches** for their capabilities. The **Controller** sends a feature request packet to do so, to which the **Switch** will reply with a feature result packet.

The **Switches** do not have much to do. They simply keep a flow table of all the routes to forward any incoming packets. They have the ability to request the **Controller** for the incoming packet's next hop if they do not have the entry

in their flow table. In Link State Routing, the **Switches** also have the ability to send information about their neighbours.

The **End-Nodes** have two basic functions, to send a message to a **Switch** which will then be forwarded to the desired **End-Node** or print out the message which has been sent by another **End-Node**.

## 2.3 Design of the Protocol

In my implementation of OpenFlow, I have attempted to implement the standard. However there are many changes or missing features as implementing OpenFlow in its entirety would be huge task and is very difficult to complete by myself.

The design of the protocol's nodes is similar to the description of my understanding of OpenFlow as described above. However, the **Controllers** are unable to remove flow table entries from the **Switches**.

The protocol uses Link State Routing to create find the path to the destination specified. Link State Routing uses Dijkstra's algorithm to find the shortest path to the destination, using the latency between nodes as the weights of the graph. However, I found it difficult to measure the latency between the nodes using Docker and thus used breadth first search as the path finding algorithm.

## 2.4 Packet Descriptions

There are many types of packets which I have created and is use when transferring data between the nodes in the network. The feature request packet is used to ask the **Switches** to send their specifications to the **Controller**. The feature result packet contains information about the **Switches** and is used when creating a graph of the network configuration. The flow mod packet is used to modify the **Switches'** flow tables. The packet in packet is used to ask the **Controller** the next hop the payload packet has to take to reach the destination. The payload packet stores the message as well as the name of the switch and the destination node. This packet is the one which is forwarded between the nodes in the network. The unknown destination packet is used to indicate to the switches that destination does not exist or a path could not be found. The hello packet is used in initialisation of the switches and to set the version number.

## 3 Implementation

This section contains an explanation of the implementation. It will describe how each of the classes used implement the design described above.

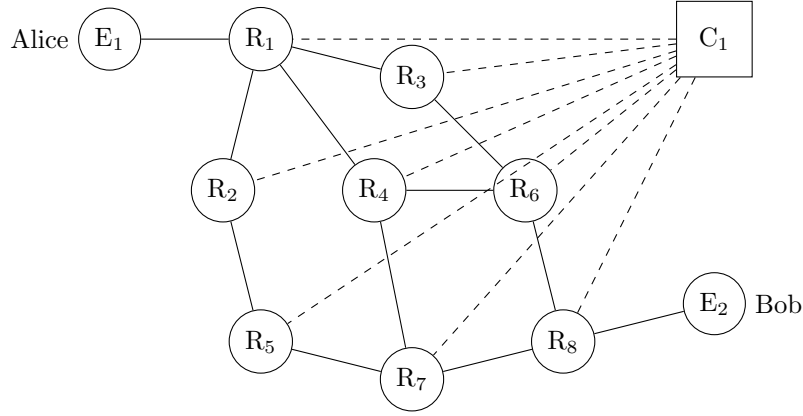


Figure 1: The figure shows shows the topology implemented by the “start\_sample\_route.sh”

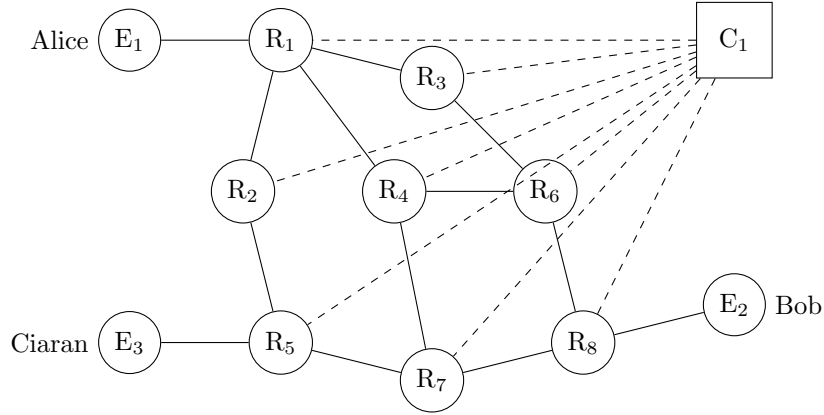


Figure 2: The figure shows shows the topology implemented by the “start\_sample\_route2.sh”

### 3.1 Packet Classes

The packet classes are encoded using the method `ObejectInputStream` and `ByteArrayInputStream`. This allows us to create a byte array which is then used to create a `DatagramPacket`. The abstract class `PacketContent` and the classes which inherit it abstract the creation of `DatagramPackets`. The abstract methods `toByteArray` and `fromDatagramPacket` do the encoding of the `DatagramPacket`. Each implementation of the `PacketContent` class has their own implementation of the abstract methods `toByteArray` and `fromDatagramPacket`. The abstract `PacketContent` abstract class and the `AckPacketContent`

class have been provided to us in the Advanced sample code. I have created other packet classes which implement the PacketContent abstract class.

### 3.1.1 FeatureRequestPacketContent

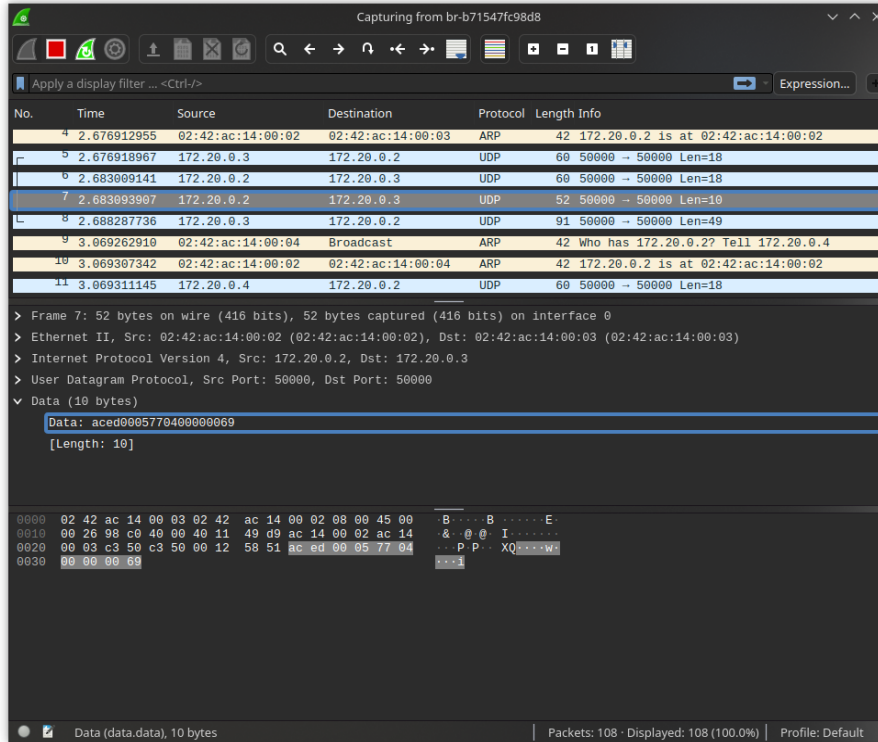


Figure 3: This figure is a feature wireshark capture of a feature request packet. The code for a feature request packet is 105 that is 0x69.

This class is used to create a packet which represents a feature request packet. The feature request packet is sent by the **Controller** to a switch upon receiving a Hello packet. The feature request packet does not contain any data and only contains its type.

### 3.1.2 FeatureResultPacketContent

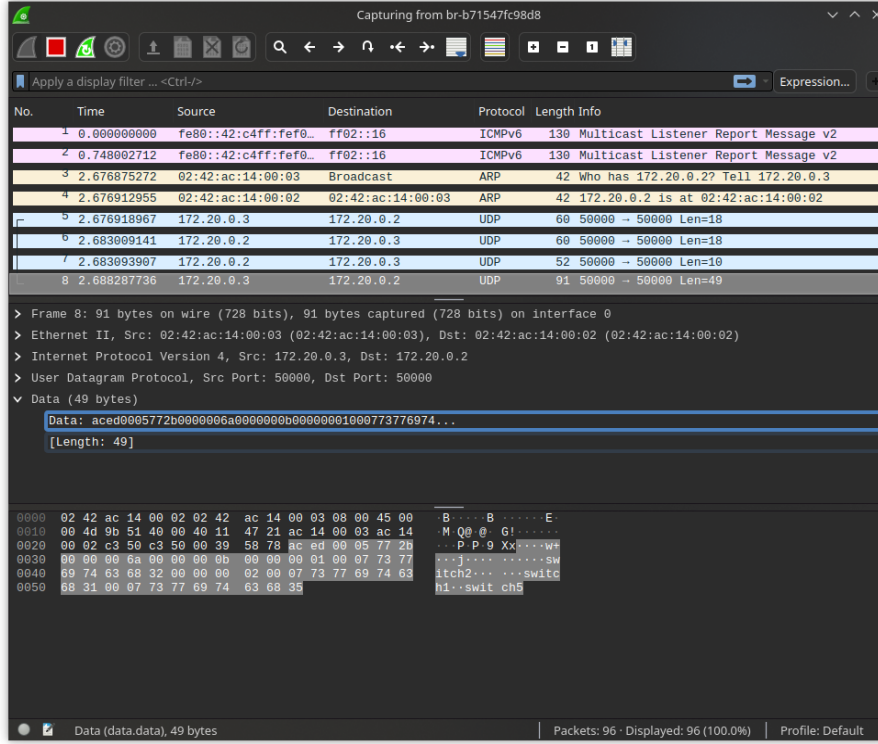


Figure 4: This figure is a feature wireshark capture of a feature result packet. The code for a feature request packet is 106 that is 0x6A.

This class is used to create a packet which represents a feature result packet. The feature result packet is sent by the **Switch** to the **Controller** in response to a feature request packet from the **Controller**. The feature request packet contains information about the specifications of the **Switch** such as the number of tables it has and the size of the buffer. The buffer size and number of tables currently have no use and is only included as it was a part of the OpenFlow specification. The feature result packet also contains the **Switch's** name and its connections. The switch connections are used when calculating the path to the specified destination.



### 3.1.3 FlowModPacketContent

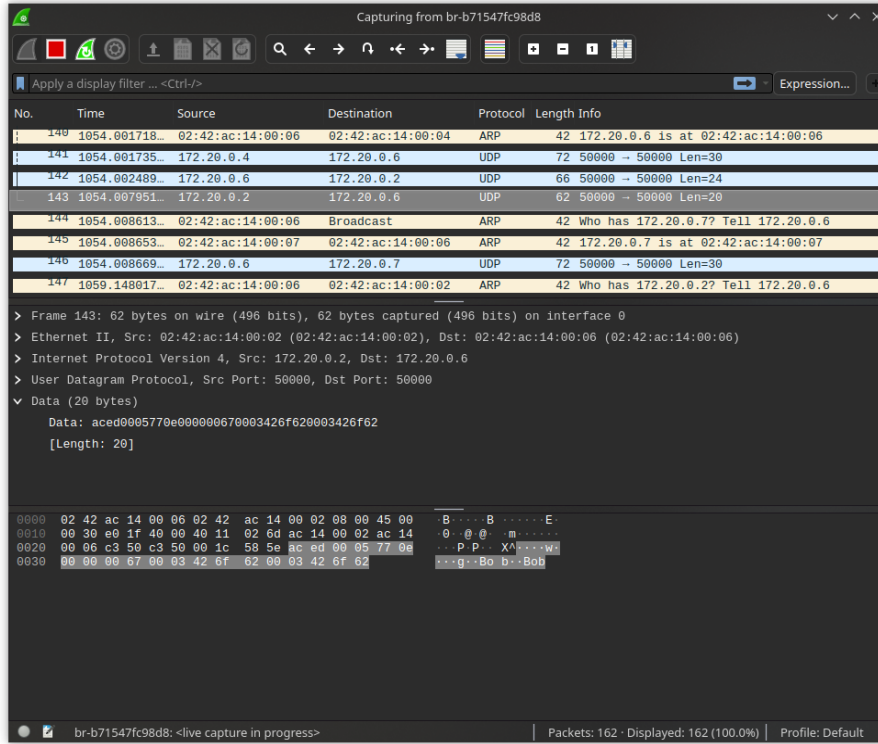


Figure 5: This figure is the wireshark capture of a flow mod packet sent by a **Controller**. The code for a flow mod packet is 103 that is 0x67.

This class is used to create a packet which represents a flow mod packet. The flow mod packet is sent by the **Controller** to a **Switch**. The flow mod packet is used to change a **Switch's** flow table. The flow mod packet contains two strings, the destination and the next hop. Upon receiving a flow mod packet, the **Switch** update it's flow table with the destination as the key and the next hop as the value.

### 3.1.4 HelloPacketContent

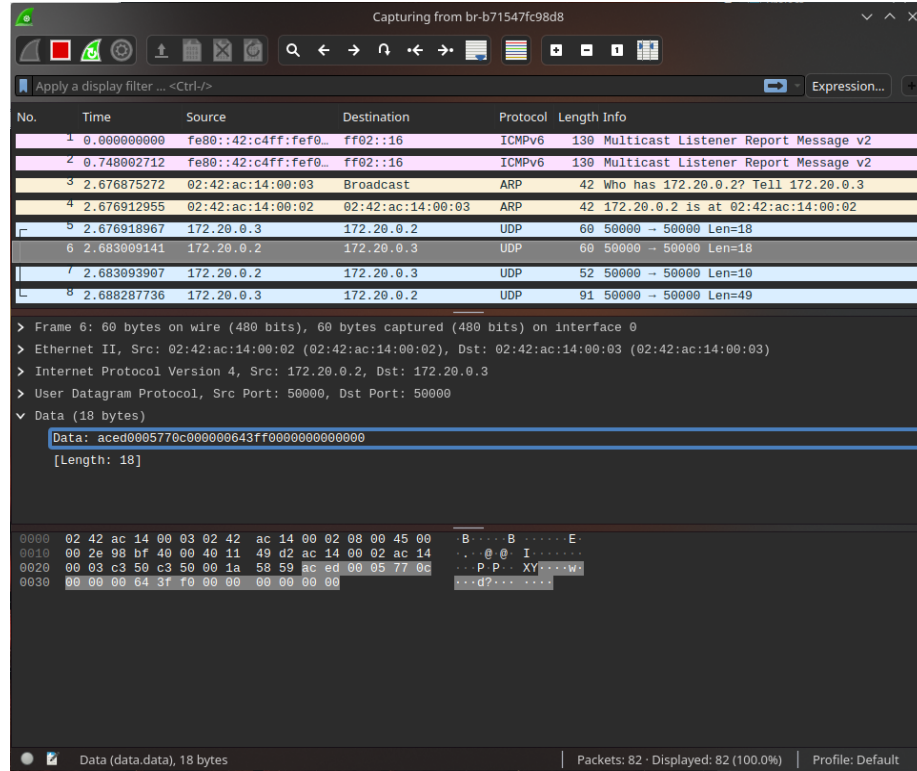


Figure 6: This figure is the wireshark capture of a hello packet sent by a **Switch**. The code for a hello packet is 100 that is 0x64.

This class is used to create a packet which represents a hello packet. The hello packet is sent by both the **Switch** and **Controller**. The hello packet contains the version number. When receiving a hello packet, the lower version of the protocol is set. In my assignment, the default version number is set to 1. In my version of OpenFlow, the hello packet is sent first by the **Switch** to the **Controller** upon initialisation. The **Controller** then responds with it's own hello packet back to the **Switch**. This differs from the actual OpenFlow specification as the hello packet should be asynchronous.

### 3.1.5 PacketInPacketContent

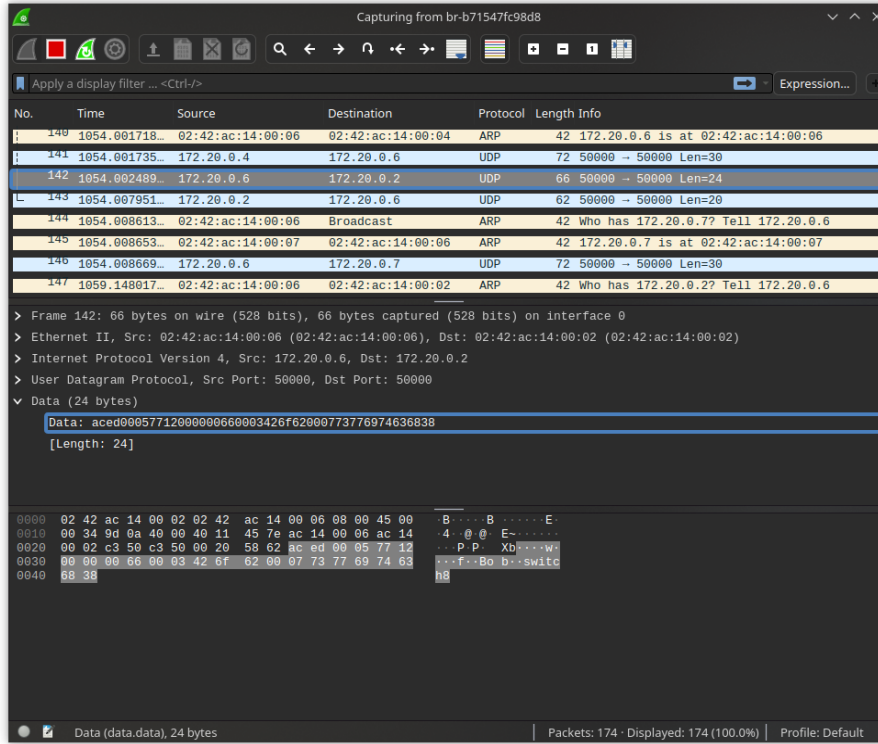


Figure 7: This figure is the wireshark capture of a packet in packet sent by a **Switch**. The code for a packet in packet is 102 that is 0x66.

This class is used to create a packet which represents a packet in packet. The packet in packet is sent by the **Switch** to the **Controller**. This packet is used to ask the **Controller** for a flow mod packet. This packet is sent in the case where a table miss occurs and the switch does not know where to forward the packet. The packet in packet contains the final destination of the packet and the switch name. The destination and the switch name are both keys in the **Controller's** flow tables.

### 3.1.6 PayloadPacketContent

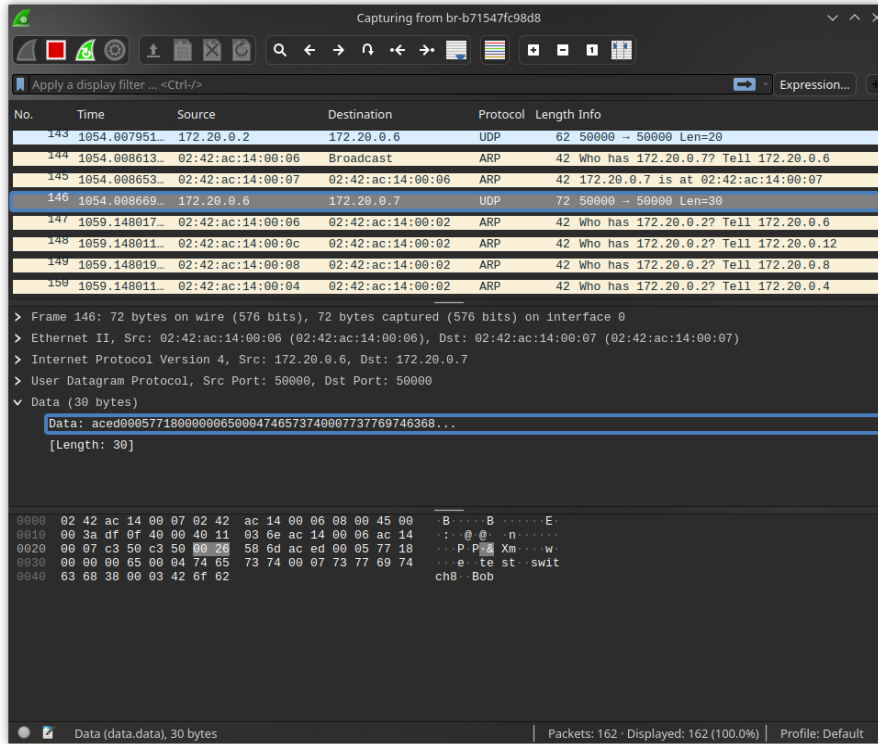


Figure 8: This figure is the wireshark capture of a payload packet sent by a **Switch**. The code for a payload packet is 101 that is 0x65.

This class is used to create a packet which represents a payload packet. The payload packet contains the payload i.e. the string containing the message and final destination of the packet. This packet is sent by the **End-Nodes** as well as the **Switches**. This packet is the packet which is forwarded from **Switch** to **Switch** and to the final destination **End-Node**.

### 3.2 Node Classes

This section includes a description of each of the nodes classes which are interfaced by the terminal. They represent any type of node in the network.

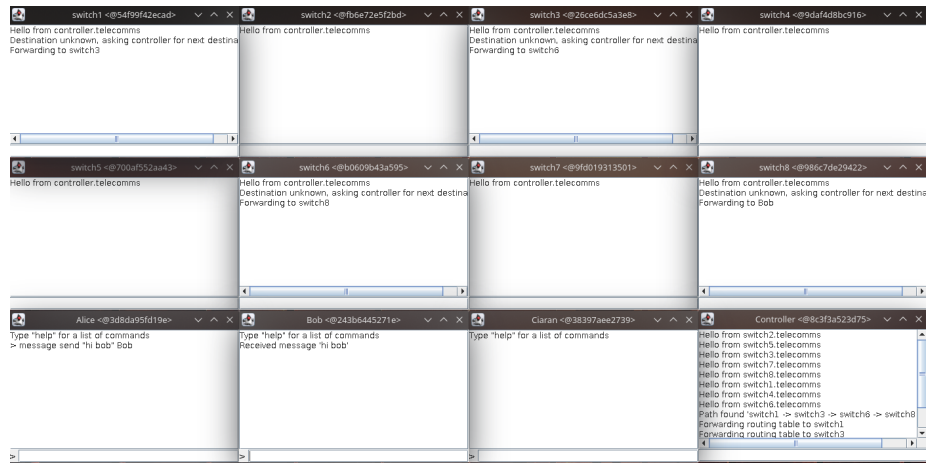
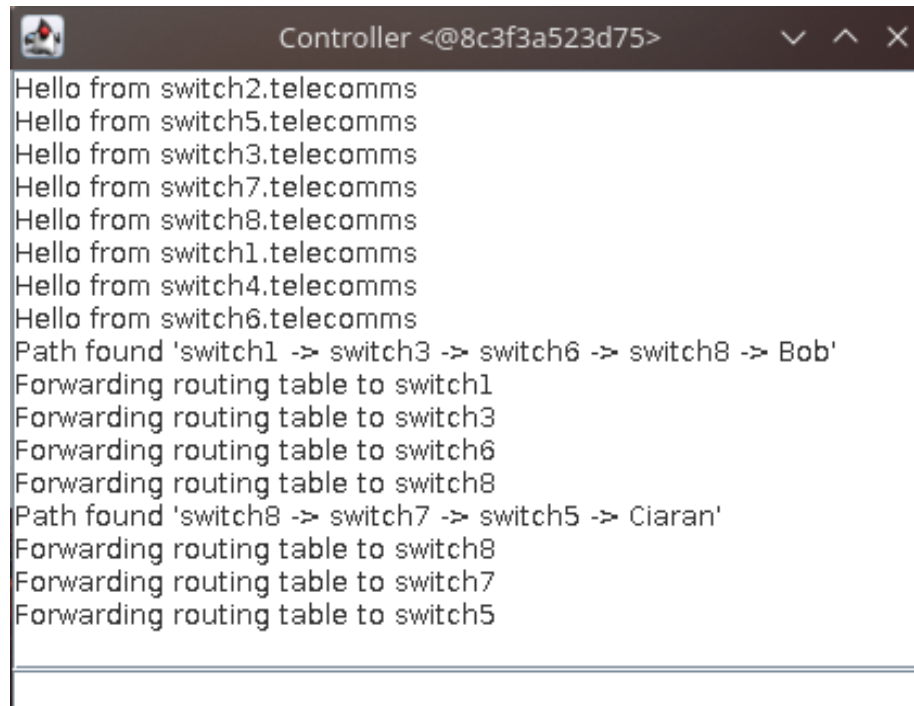


Figure 9: The figure shows a message being sent from Alice to Bob. The hops the packet took and the path generated by the **Controller** is shown.

### 3.2.1 Controller

A screenshot of a terminal window titled "Controller <@8c3f3a523d75>". The terminal displays a series of messages: seven "Hello from switchX.telecomms" messages for switches 2, 5, 3, 7, 8, 1, and 4; a path found message "Path found 'switch1 -> switch3 -> switch6 -> switch8 -> Bob'"; four "Forwarding routing table to switchX" messages for switches 1, 3, 6, and 8; another path found message "Path found 'switch8 -> switch7 -> switch5 -> Ciaran'"; and three "Forwarding routing table to switchX" messages for switches 8, 7, and 5.

```
Controller <@8c3f3a523d75>
Hello from switch2.telecomms
Hello from switch5.telecomms
Hello from switch3.telecomms
Hello from switch7.telecomms
Hello from switch8.telecomms
Hello from switch1.telecomms
Hello from switch4.telecomms
Hello from switch6.telecomms
Path found 'switch1 -> switch3 -> switch6 -> switch8 -> Bob'
Forwarding routing table to switch1
Forwarding routing table to switch3
Forwarding routing table to switch6
Forwarding routing table to switch8
Path found 'switch8 -> switch7 -> switch5 -> Ciaran'
Forwarding routing table to switch8
Forwarding routing table to switch7
Forwarding routing table to switch5
```

Figure 10: The figure shows the **Controller** finding a path to the destination required and forwarding the appropriate routing tables. The figure shows is the result of Alice sending a message to Bob and Bob sending a message to Ciaran.

The **Controller** is used to direct the **Switches** when they have no flow table entry for a packet. The **Controller** sends a flow mod packet upon receiving a packet in packet. The paths are stored in a hash table. The key is a string which is the final destination the packet to be forwarded. The value is another hash table of which the key is the switch. The value is the next hop the switch has to forward the packet to.

The **Controller** used to contain a predefined path. However I have since changed it to use Link State Routing. The **Controller** uses the Graph class in order to calculate a path to the destination. If the destination is not in the hash table, the path is calculated and is then stored into the hash table. The path is only generated when a **Switch** requests for the next hop and the destination is not found in the flow table. If no path is found, the **Controller** sends an unknown destination packet to the **Switch**.

### 3.2.2 EndNode

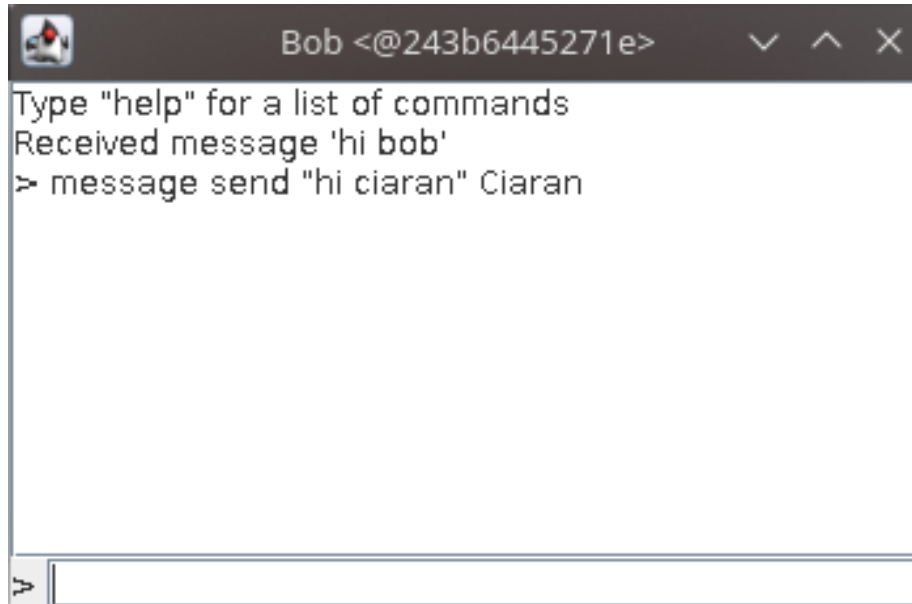


Figure 11: The figure shows a message being sent and received

The **End-Node** is where the user can send and receive messages. The user is able to choose another **End-Node** to send the message to. The message it takes in is of type string and is encapsulated in a payload packet. This payload packet is sent to a **Switch** which is then forwarded to another **Switch** or to the destination **End-Node**. When an **End-Node** receives a payload packet, the message is printed out to the terminal.

The **End-Node** implements a command system. This command system contains the commands “help”, “clear” and “message send <‘message’><destination>”. The “help” command is used to display all the commands available. The “clear” command is used to clear the terminal screen and the “message send” command is used to send a string message to the destination **End-Node**.

### 3.2.3 Node

This class is an abstract class which all of the **Node** classes inherit. This class has been provided to us from the advanced sample code. It has been modified slightly in the previous assignment, the changes of which is carried through to this assignment.

### 3.2.4 Switch

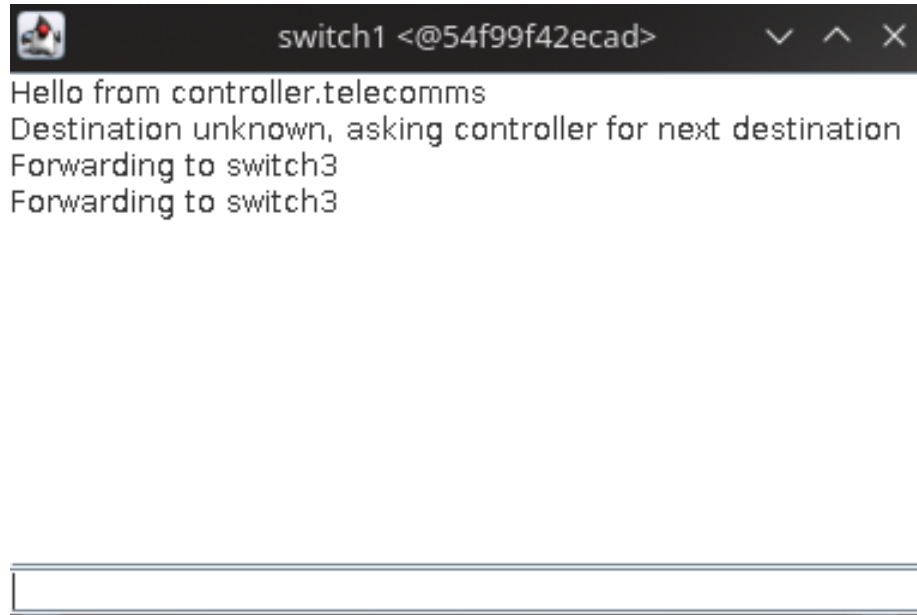


Figure 12: The figure shows a payload packet forwarded by the **Switch**, first case where table miss occurs, second case the packet is forwarded directly to the next hop without contacting the **Controller**.

The **Switch** class uses a hash table to store the payload packet's next hop. When the **Switch** receives a payload packet, the destination is looked up in the hash table. If the destination in the hash table exists, the packet is then forwarded to the next hop. If the destination does not exist, then the **Switch** sends a packet in packet to the **Controller**. A countdown latch is used to wait for a flow mod packet to arrive. Upon the arrival of a flow mod packet, the payload packet is then forwarded to the next hop.

A queue called the packet buffer is used to deal with payload packet arrivals. This allows the payload packets to be handled one at a time. An infinite loop is used to process the payload packets in a thread.

## 3.3 Path Finding class

This path finding class is used to implement the link state routing. It implements the algorithm which computes the path to the destination.



### 3.3.1 Graph

The graph class contains a node class. This node class represents nodes in the network in a graph. The node class contains the name of the node as well as the node's adjacent nodes. This abstract data structure is used when calculating the path to the destination.

All of the nodes are stored in the graph's hash table. The key of the hash table is a string which is the name of the node. The value of the hash table is the reference to the node which has that name.

I found it difficult to measure the latency between the connections in Docker and thus was unable to obtain the weights of the edges in the graph. I used a breadth first search algorithm instead of Dijkstra's algorithm for this reason.

## 3.4 Miscellaneous Classes

This section contains an explanation of the classes which did not fit into the other categories.

### 3.4.1 Terminal

The terminal class was provided to us in the sample code. I have slightly modified the terminal class to have the ability to clear the terminal screen as well as read a boolean from the user. The boolean function was implemented using JOptionPane to display a yes or no question. The boolean function however was not used in this assignment but was used in the first assignment.

### 3.4.2 Tokenizer

The tokenizer class was used to split up the command into tokens and has the ability to separate messages in quotations. This was created instead of using the string split function as it allows the message sent to have spaces instead of being a single word.

## 4 Advantages and Disadvantages

### 4.1 Advantages

- Implements Link State Routing thus allowing any valid switch and end-point connection combinations to be used.
- Can add more routers to the network on the fly.
- The network graph can be directed i.e. **Switches** can have one way connections
- Does not allow the use of multiple **Controllers**.
- It is asynchronous due to the use of threads.

- The implementation is abstracted well using Java’s object orientated nature and the use of classes.

## 4.2 Disadvantages

- Does not deal with dead routers.
- Does not deal with invalid or unconnected graphs.
- Is not an accurate implementation of the OpenFlow specification.
- Does not implement acknowledgements.
- Command system may be confusing to the user.
- Certain network configurations may cause the path to be calculated multiple times.

## 5 Program Usage

The following are some instructions to run the project:

- This program was built around the use of Docker.
- To run the program, the bash script “start\_sample\_route.sh”, “start\_sample\_route2.sh” or the “start\_straight\_line.sh” may be used.
- The script must be run with root due to the nature of Docker and it’s permissions required to run.
- The script “start\_straight\_line.sh” may take an integer parameter to indicate the number of switches between the endpoints e.g. “sudo start\_straight\_line.sh 3” to have three switches between the endpoints.

The program has only been tested on a Linux system and is not guaranteed to run on a Windows or Mac machine.

## 6 Reflection

This assignment has taught me many things about the OpenFlow software defined networking standard. Doing this assignment has also taught me more things about Java threads and concurrent programming as well as Docker and bash scripting. I also learned a lot about implementing graph algorithms and about graphs in general.

Overall I am very happy with my implementation of the OpenFlow standard. I would like to improve on my bash scripting as I feel that the code I produced for the bash scripts are somewhat hard to read. If I were to complete the project

again I would attempt to implement Dijkstra's algorithm or perhaps even more complex graph algorithms.

I have spent overall 40 hours on the assignment in researching, designing, implementing the design and writing the report.