



---

# CS2031 Telecommunications II

## Assignment #2: OpenFlow

---

Jack Gilbride, Student ID #17340868

December 14, 2018

### Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Theory of Topic</b>	<b>2</b>
2.1	Software Defined Networking . . . . .	2
2.2	OpenFlow . . . . .	2
<b>3</b>	<b>Implementation</b>	<b>2</b>
3.1	Node . . . . .	2
3.2	Controller . . . . .	4
3.3	Switch . . . . .	8
3.4	EndNode . . . . .	11
<b>4</b>	<b>Discussion</b>	<b>12</b>
4.1	Distance Vector Routing . . . . .	12
4.2	Link State Routing . . . . .	13
4.3	Dynamic Change of Flow Tables . . . . .	13
4.4	Threading . . . . .	13
<b>5</b>	<b>Summary</b>	<b>13</b>
<b>6</b>	<b>Reflection</b>	<b>14</b>

## 1 Introduction

The task of this assignment was to design and implement our own semi-realistic version of OpenFlow. In this report I will describe my approach towards this assignment, starting with the theory of the topic, then describing my own implementation of the protocol in Java. I will finish my report with a discussion, a summary and a reflection on the overall assignment.

## 2 Theory of Topic

In this section I will describe the concepts that were used to realise a solution to my final OpenFlow protocol. Some theory, such as Datagram Packets and Maximum Transmission Units, are left out as they were already discussed in the documentation for Assignment 1.

### 2.1 Software Defined Networking

Software Defined Networking (SDN) is a move away from traditional networking. In a software-defined network, a network engineer or administrator can shape traffic from a centralized control console without having to touch individual switches in the network. The network consists of three "planes"; an application plane in the application layer, a controller plane in the control layer and a data plane in the infrastructure layer. The application layer contains typical networking applications that an organization or business may use. The control layer contains SDN control software. This layer provides network services which manage the flow of traffic in the system. The infrastructure layer is made up of physical network devices called switches which route traffic through the network along paths defined by the control layer. Interfaces are needed between the layers; typically application programming interfaces (APIs) exist between the application and control layers. Between the control layer and infrastructure layer there is a Control Data Plane interface, such as OpenFlow.

### 2.2 OpenFlow

As mentioned above, OpenFlow is a Control Data Plane interface between the control layer and infrastructure layer in SDN. The original OpenFlow paper, "OpenFlow: Enabling Innovation in Campus Networks", was published in 2008 and was written by very influential people in telecommunications from various universities. OpenFlow works between OpenFlow switches and a controller with a secure connection. OpenFlow switches consist of a software layer containing an OpenFlow client and a hardware layer. One of the most important aspects in the hardware layer of an OpenFlow switch is the use of configurable flow tables. These define what should be done with received packets based on many factors such as the hardware address, Internet Protocol (IP) address or the port number of the source or destination. When a switch is unsure of what to do with a packet it may query the controller with information in the packet. The controller then tells the switch what to do with the packet. Flow tables in the infrastructure layer may be altered by the controller.

## 3 Implemenation

### 3.1 Node

In Assignment 1 we designed a Publish-Subscribe protocol which defined communication between three types of Nodes; a Publisher, a Subscriber and a Broker. Each of these Nodes extended an abstract Node class that we were given on Blackboard. The Node already has some functionality which involved listening for packets, and I decided to expand that to add other functionality that would be common to all three Node types. That method worked well for me in Assignment 1 so I decided to use it again in this assignment. Once again communication was done using DatagramPackets. Listener was a class inside of Node which the Node used to listen for these DatagramPackets and hand them over to onReceipt, an abstract function that each subclass would have to implement itself. So if a Node was waiting, its Listener would wait for packets and hand them over to onReceipt which each Node would deal with differently.

The first thing added to the controller was constants representing OpenFlow packet types. These were given to us in lecture notes on Blackboard. I decided to implement them as bytes so that they would not take up unnecessary space as fields in a packet. I also defined constants for two different types of packets outside of the OpenFlow protocol; messages between EndNodes and configuration between EndNodes and their nearest switch. Some more constants were also included in the class such as the maximum packet size, indexes for the flow table, the number of Switches and the number of EndNodes in the system.

A base port number was also added to the constants, set to 5000. I wanted my port numbers to start at 5000 because they did in the sample Client/Server code we were given on Blackboard so I knew that these ports would always work on localhost. Having a base port number as a constant meant that I could represent

socket numbers in flow tables using a one-byte field; e.g. socket number 5001 could be represented as 1 in the table to fit into one byte.

In terms of defining a packet layout, I first looked at OpenFlow's packet layouts. I realised that types of packets had different layouts as they would need to convey different types of information. All that they had in common was a fixed header, containing the OpenFlow version, the packet type, the length and the xid, or transaction identifier, used to match requests to responses. I decided to only keep the type field in my header as there would only be one version, the length would be defined in the DatagramPacket length field already and upon sending a message my Nodes would wait for the response, so there was no need to identify which request it was a response to. As a result the only general use of packet fields in the Node would be setting and getting the type. The code in listing 1 shows the simple getters and setters for the packet type.

```
protected byte getType(byte data[]) {  
    return data[0];  
}  
  
protected void setType(byte[] data, byte type){  
    data[0] = type;  
}
```

Listing 1: Functionality of the Node class to set and get the packet type, which may be an Openflow Packet type, an initialisation between an EndNode and a Switch, or a message between two EndNodes.

I also defined a packet layout for messages between Nodes. Index 1 would contain the source port number and Index 2 would contain the destination port number, minus the base port number. The rest of the data would contain the message. This was done because the destination and source port fields in the DatagramPacket may change as the packet is routed between EndNodes, Switches and the Controller. Therefore the original source and final destination needed to be in the data field. Listing 2 shows the functionality of Node to access the fields of a message between EndNodes.

```
/** The following three methods are for parsing messages between End Nodes. */  
protected byte getMessageSource(byte data[]) {  
    assert (getType(data)==NODE.MESSAGE);  
    return data[1];  
}  
  
protected byte getMessageDest(byte data[]) {  
    assert (getType(data)==NODE.MESSAGE);  
    return data[2];  
}  
  
protected String getMessageContent(byte data[]) {  
    assert (getType(data)==NODE.MESSAGE);  
    byte[] content = Arrays.copyOfRange(data, 3, data.length);  
    String messageContent = new String(content).trim();  
    return messageContent;  
}
```

Listing 2: Functionality of the Node class to set and get the fields of messages between EndNodes. The methods also assert that the packets are indeed messages.

The rest of the functionality of the system is implemented by individual subclasses, as there is much more functionality implemented by only one class than in Assignment 1.

### 3.2 Controller

My Controller contained a flow table for the entire network, defined as a constant 2-Dimensional Array. Each row of the flow table contained 5 bytes. Index 0 contained the destination port of the packet. Index 1 contained the source port. Index 2 contained the port that the packet was being routed through. Index 3 contained its previous hop and index 4 contained its next hop. I decided to implement the network given to us as an example in the Assignment briefing, pictured in figure 1. I also added two more EndNodes, E3 and E4, connected to Switch 5 and Switch 3 respectively.

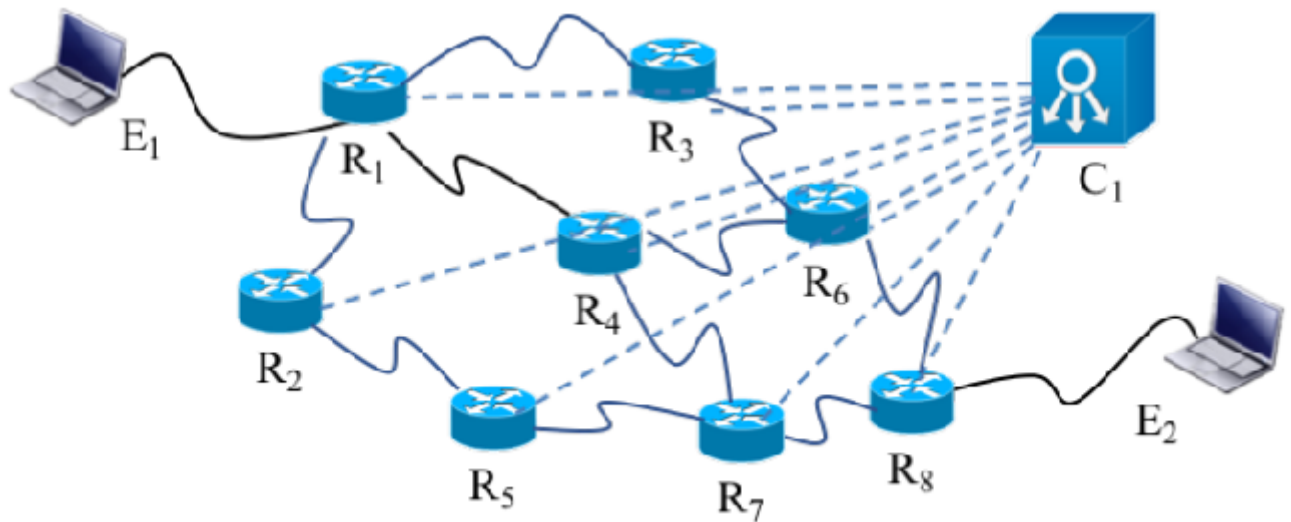


Figure 1: Fixed layout of my OpenFlow network. Not pictured; EndNode E3 connected to R5 and EndNode E4 connected to R3.

This overall view of the network was built into the flow table of the Controller using constants. This can be seen in 3.

```
/**
 * 2D Array storing the preconfiguration information of the controller. Column
 * 0 = destination port numbers, column 1 = source port numbers, column 2 =
 * router number, column 3 = input port of switch, column 4 = output port of
 * switch
 */
private static final byte[][] PRECONFINFO = { { E1, E2, R1, E1, R4 },
{ E1, E2, R4, R1, R7 }, { E1, E2, R7, R4, R8 }, { E1, E2, R8, R7, E2 },
{ E1, E3, R1, E1, R2 }, { E1, E3, R2, R1, R5 }, { E1, E3, R5, R2, E3 },
{ E1, E4, R1, E1, R3 }, { E1, E4, R3, R1, E4 }, { E2, E1, R8, E2, R7 },
{ E2, E1, R7, R8, R4 }, { E2, E1, R4, R7, R1 }, { E2, E1, R1, R4, E1 },
{ E2, E3, R8, E2, R7 }, { E2, E3, R7, R8, R5 }, { E2, E3, R5, R7, E3 },
{ E2, E4, R8, E4, R6 }, { E2, E4, R6, R8, R3 }, { E2, E4, R3, R6, E4 },
{ E3, E1, R5, E3, R2 }, { E3, E1, R2, R5, R1 }, { E3, E1, R1, R2, E1 },
{ E3, E2, R5, E3, R7 }, { E3, E2, R7, R5, R8 }, { E3, E2, R8, R7, E2 },
{ E3, E4, R5, E3, R2 }, { E3, E4, R2, R5, R1 }, { E3, E4, R1, R2, R3 },
{ E3, E4, R3, R1, E4 }, { E4, E1, R3, E4, R1 }, { E4, E1, R1, R3, E1 },
{ E4, E2, R3, E4, R6 }, { E4, E2, R6, R3, R8 }, { E4, E2, R8, R6, E2 },
{ E4, E3, R3, E4, R6 }, { E4, E3, R6, R3, R4 }, { E4, E3, R4, R6, R7 },
{ E4, E3, R7, R4, R5 }, { E4, E3, R5, R7, E3 } };
```

Listing 3: The preconfiguration information of the controller represented as a 2D array. For any node at index 2 in a row, that row belongs to its flow table. R1 to E4 are constants from 1 to 12, which are added onto the base port number to give the port number of that Node.

A problem that I had in Assignment 1 was having too many mainlines. Each time that I wanted the system to run I had to run the Publisher, Broker and Subscriber separately. This also confused me in terms of threading, and how to run two different Subscribers by running the Subscriber twice. As a result, there is only one mainline in my OpenFlow program. This simply starts the Controller. The constructor of the Controller initialises every Node in the program, as seen in listing 4.

```
Controller(Terminal terminal) throws SocketException {
    // Initialise Controller
    Controller.terminal = terminal;
    this.socket = new DatagramSocket(BASEPORTNUMBER + CONTROLLER.PORT);
    listener.go();
    // Initialise Switches
    switches = new Switch[NUMSWITCHES + 1];
    for (byte i = R1; i <= NUMSWITCHES; i++) {
        switches[i] = new Switch(i);
    }
    // Initialise EndNodes
    endNodes = new EndNode[NUMENDNODES];
    for (byte j = 0; j < NUMENDNODES; j++) {
        endNodes[j] = new EndNode((byte) (j + NUMSWITCHES + 1));
    }
}
```

Listing 4: The constructor of the Controller. Initialises itself as well as all other Nodes in the system. The Nodes will assign themselves the passed port number plus the base port number. The initialised Nodes also create a Terminal and Listener themselves.

After this the Controller's start method is called. This starts the first Switch as seen in listing 5.

```
public synchronized void start() throws Exception {
    startSwitch(R1);
}

private synchronized void startSwitch(int routerNumber) {
    Switch s = switches[routerNumber];
    s.start();
}
```

Listing 5: The start method, which starts Switch 1. The rest of the Switches are started sequentially later in the program.

As seen in listing 5, only one Switch is started. This is because the other Switches will be started later. I wanted to follow the diagram in figure 2, however between steps 4 and 5 I wanted to also send a FlowMod message to give the Switch its flow table before moving onto the next Switch. Once this is done, the Controller undergoes the same process with the next Switch.

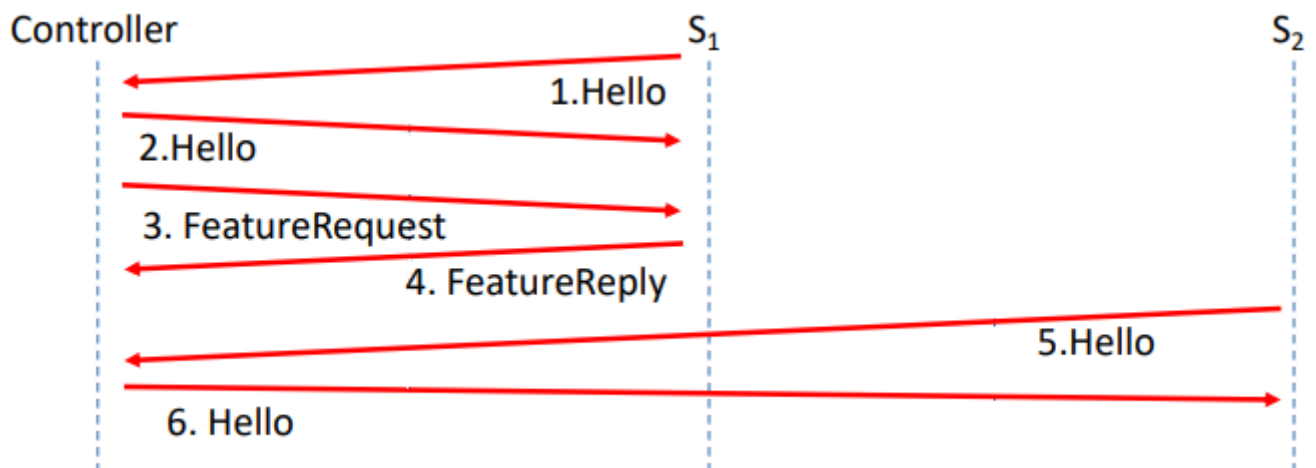


Figure 2: How the Controller interacts with the Switches upon startup. Not shown: the FlowMod message and reply before starting the next Switch.

In the Controller class, this was all done in the `onReceipt` method. After a Switch is started the Controller just waits for incoming packets. When it receives a Hello, it will send a Hello back, then a Feature Request, and wait again. When it gets a Feature Reply it will send the relevant Flow Mod table. When it receives a Flow Mod reply, it moves on to the next Switch. This is all shown in listing 6.

```

public synchronized void onReceipt(DatagramPacket packet) {
    byte[] data = packet.getData();
    int port = packet.getPort() - BASEPORTNUMBER;
    byte type = getType(data);
    switch (type) {
        case OFPT_HELLO:
            terminal.println("Hello_packet_received_from_switch_number_" + port + ".");
            sendHello((InetSocketAddress) packet.getSocketAddress());
            sendFeaturesRequest((InetSocketAddress) packet.getSocketAddress());
            break;
        case OFPT_FEATURES_REPLY:
            terminal.println("Features_reply_received_from_switch_number_" + port + ".");
            if (data[1] == BASIC_FEATURES) {
                terminal.println("Switch_" + port + "_has_basic_features.");
            } else {
                terminal.println("Features_of_switch_" + port + "_unknown.");
            }
            sendTable((byte) port);
            break;
        case OFPT_FLOW_MOD:
            terminal.println("Flow_mod_complete_for_switch_number_" + port + ".");
            if (port < NUMSWITCHES) {
                startSwitch(port + 1); // flow mod is complete, start the next switch
            }
            break;
        case OFPT_PACKET_IN:
            setType(data, OFPT_FLOW_REMOVED);
            packet.setData(data);
            packet.setSocketAddress(packet.getSocketAddress());
            try {
                socket.send(packet);
            }
    }
}
  
```

```
terminal.println("Told switch_" + port + "_to_drop_packet.");  
} catch (IOException e) {  
    e.printStackTrace();  
}  
break;  
}  
}
```

Listing 6: The onReceipt method, which handles incoming packets. When it receives a packet it will move onto the next stage of setting up the Switches.

There are some things to note about listing 6. Firstly, certain methods such as sendHello, sendFeaturesRequest and sendTable are called. These all do as expected; send a packet of the relevant type to the relevant Switch. sendTable also puts the Switch's router table into the data of the packet. This is every row of PRECONF\_INFO where the Switch's port number is at index 2. These are packed into a 1D array in the packet, so every five bytes contains a new row of the Switch's router table.

Secondly, when getting a Features Reply, the Controller only checks if the Switch has "basic features" and continues on. This is because each Switch in the program will have the same features, but Feature Requests are still necessary to include as part of the protocol.

Thirdly, there is a fourth case in the switch statement which handles the Packet In type. This is when a Switch does not know what to do with a packet so forwards it to the Controller. The only possible packets that a Switch would not know what to do with are those from an EndNode sending to itself. In this case the Controller tells the Switch to drop the packet, as there is no route specified from an EndNode to itself in the router table. It is also because the user would not have time to put it directly into a waiting state before the packet is sent back, as we will see later. This means that the packet would be dropped anyways so there is no point sending it across the medium.

Figure 3 shows the Terminal output for the Controller.

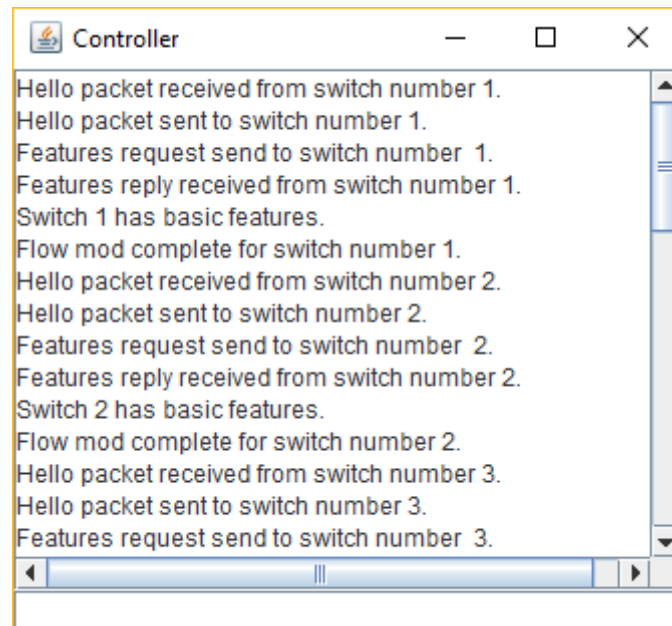


Figure 3: Part of the output for the Controller. Any relevant information is printed to the Terminal to show the user the order of the setup. This output continues until Switch 8.

### 3.3 Switch

When a Switch is constructed, it assigns itself the passed port number and constructs a Terminal. Upon startup it will send a Hello packet to the Controller, and follow the pattern shown in figure 2. In this pattern, the Switch will always say that it has "Basic Features". After this it parses its router table from a FlowMod packet. This is the packet's data starting at index 1, as index 0 contains the packet type. Nothing additional is done in relation to the Switch and Controller that is not explained in the Controller section.

However the Switch does have other interactions in the system. These are interactions with the other Switches and EndNodes. When a Switch has received its Flow Mod packet, i.e. has finished its setup, it will call setEndNodeAddress. This is a way for EndNodes to figure out which Switch they are connected to, as seen in listing 7. The Switch parses its router table. If any port numbers of the previous or next immediate hops are greater than the maximum switch number, then that hop is an EndNode. The Switch will send a packet of type NODE\_INITIALISE\_SWITCH. The EndNode then sets this Switch's address as its destination address. Of course the Switch may not be connected to any EndNodes, in which case no packet of this type is sent.

```

/**
 * This method lets the switch figure out which end node it is connected to
 * by checking its flow table. If at any point in the table its immediate
 * input or output is beyond the switch index, then it is an end node. Each
 * switch is connected to a maximum of one node in the system so the check
 * can end once if the end node is found.
 */
private synchronized void setEndNodeAddress() {
    boolean addressSet = false;
    int i = 0;
    while (i < flowTable.length && !addressSet) {
        if (flowTable[i][INPUT_INDEX] > NUMSWITCHES) {
            int portNumber = flowTable[i][INPUT_INDEX] + BASEPORTNUMBER;
            this.endNodeAddress = new InetSocketAddress(LOCALHOST, portNumber);
            addressSet = true;
            terminal.println("This switch is connected to end node" +
                (portNumber - BASEPORTNUMBER - NUMSWITCHES) + ".");
        } else if (flowTable[i][OUTPUT_INDEX] > NUMSWITCHES) {
            int portNumber = flowTable[i][OUTPUT_INDEX] + BASEPORTNUMBER;
            this.endNodeAddress = new InetSocketAddress(LOCALHOST, portNumber);
            addressSet = true;
            terminal.println("This switch is connected to end node" +
                (portNumber - BASEPORTNUMBER - NUMSWITCHES) + ".");
        }
        i++;
    }
    if (!addressSet) {
        terminal.println("This switch is not connected to an end node in the"
            + " network.");
    } else {
        byte[] data = { NODE_INITIALISE_SWITCH };
        DatagramPacket initialisation = new DatagramPacket(data, data.length);
        initialisation.setSocketAddress(endNodeAddress);
        try {
            socket.send(initialisation);
            terminal.println("Switch port number sent to the connected end node.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```



```
}
}
```

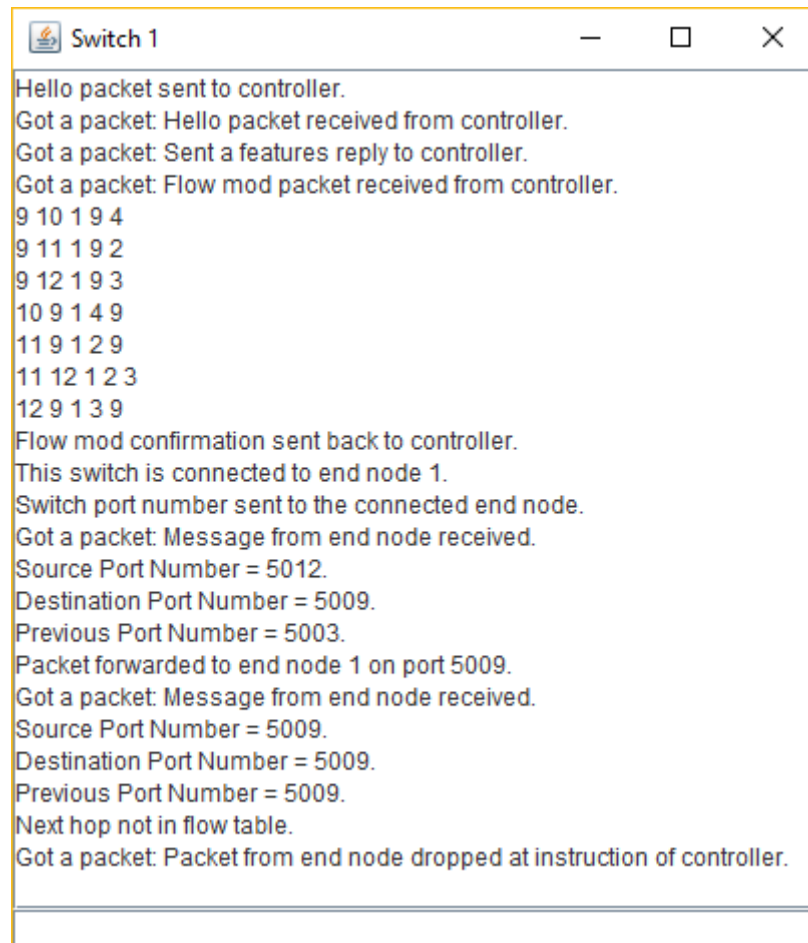
Listing 7: The setEndNodeAddress method. This method parses the Switch's router table to find any directly connected EndNodes. If it finds an EndNode, it will contact it. The EndNode will then set the Switch's address as the destination address for all packets.

After all of the setup the Switch is ready to serve as a router for messages between EndNodes. A message may come directly from an EndNode or a Switch that has forwarded it. On receipt of any packet, the Switch checks its source address. If it is the Controller's address, it handles the packet as mentioned earlier. Otherwise, it calls the method handleEndNodePacket. This method is shown in listing 8.

```
private synchronized void handleEndNodePacket(DatagramPacket packet) {
    byte[] data = packet.getData();
    if (getType(data) == NODEMESSAGE) {
        terminal.println("Message_from_end_node_received.");
        byte nextHop = checkFlowtable(data, packet.getPort());
        if (nextHop == CONTROLLERPORT) {
            terminal.println("Next_hop_not_in_flow_table.");
            byte[] unrecognised = new byte[data.length];
            setType(unrecognised, OFPT_PACKET_IN);
            for (int i=1; i<unrecognised.length; i++){
                unrecognised[i] = data[i-1];
            }
            DatagramPacket packetIn = new DatagramPacket(unrecognised,
                unrecognised.length);
            packetIn.setSocketAddress(controllerAddress);
            try {
                socket.send(packetIn);
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else {
            InetAddress outputAddress = new InetAddress(LOCALHOST,
                BASE_PORT_NUMBER + nextHop);
            packet.setSocketAddress(outputAddress);
            try {
                socket.send(packet);
                if (nextHop <= NUMSWITCHES) {
                    terminal.println("Packet_forwarded_to_switch_" + nextHop + "_on_port_" +
                        outputAddress.getPort() + ".");
                } else {
                    terminal.println("Packet_forwarded_to_end_node_" + nextHop % NUMSWITCHES
                        + "_on_port_" + outputAddress.getPort() + ".");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Listing 8: The `handleEndNodePacket` method, which forwards the packet onto the next hop according to its routing table. This is done using simple loops and array indexing in `checkFlowtable`. If the next hop is not in the flow table then the packet is forwarded to the controller in a `Packet In OpenFlow` message.

The functionality of the Switch has been described. All that is left to look at is its output. This is shown in figure 4.



```
Switch 1
Hello packet sent to controller.
Got a packet: Hello packet received from controller.
Got a packet: Sent a features reply to controller.
Got a packet: Flow mod packet received from controller.
9 10 1 9 4
9 11 1 9 2
9 12 1 9 3
10 9 1 4 9
11 9 1 2 9
11 12 1 2 3
12 9 1 3 9
Flow mod confirmation sent back to controller.
This switch is connected to end node 1.
Switch port number sent to the connected end node.
Got a packet: Message from end node received.
Source Port Number = 5012.
Destination Port Number = 5009.
Previous Port Number = 5003.
Packet forwarded to end node 1 on port 5009.
Got a packet: Message from end node received.
Source Port Number = 5009.
Destination Port Number = 5009.
Previous Port Number = 5009.
Next hop not in flow table.
Got a packet: Packet from end node dropped at instruction of controller.
```

Figure 4: The output of Switch 1. Each Switch has a similar output. Pictured; Setup stage with Controller, receipt of router table, identification of connected EndNode, forwarding a message from a Switch, dropping a message sent from an EndNode to itself.

As seen in figure 4, a Switch prints all relevant output to the user. This includes setup information and information on forwarding packets. The router table of the Switch is also printed to show the user, which may be verified by looking at how packets are forwarded later in the program. Much like the Controller, this Terminal takes no input and is purely there to show the user the functionality of the object without having to look at the code.

### 3.4 EndNode

As mentioned in the theory section of this report, OpenFlow is a way to define the interactions between the Controller and Switches in an SDN. So with these two classes created I had essentially implemented my version of OpenFlow. However, the whole point of networking is forwarding data between end users, which had not been implemented yet. So I created an EndNode class, which would take user input and send messages to other EndNodes through the network. It would only need to know its nearest Switch to be able to access the network and would have no view at all of the network's topology.

When constructed, the EndNode initialises its Terminal, initialises its Listener and assigns itself a passed socket number. It then waits for an initialisation message from any Switch. When this is received, the EndNode sets its destination address to the source of the initialisation message. This is the address of the EndNode's nearest Switch and the gateway of the EndNode into the network for all incoming and outgoing messages.

Waiting for its destination address is the only setup needed by the EndNode. After this it calls the start method, seen in listing 9. This method was influenced by an issue that I had in Assignment 1, where I could not make the Subscriber wait for user input and incoming packets at the same time. This was important because a Subscriber had to be able to subscribe and unsubscribe due to user input but also print incoming publications due to incoming packets. It was also important this time as an EndNode had to send messages but also print received ones. My solution last time was to let the Subscriber subscribe to one topic and then always wait for publications. This time my solution involved two states; WAIT and SEND. If the user wanted to receive messages they would input "WAIT". The EndNode would break out of this state once a message is received and printed. If a user instead wanted to send a message they would input "SEND", the number of the EndNode to send to and the message to send. This state would be broken out of once the message is sent and the user is asked again whether they want to send or wait. Any incoming packets are dropped when the user is either selecting their state or sending, as the listener is only active during the waiting state. This was the best solution that I could think of to let the user send and receive messages whenever they wanted without using threading, which I found very complicated and could not implement due to time constraints.

```
public synchronized void start() {
    while (true) {
        String chosenState =
            terminal.read("Please_enter_SEND_or_WAIT_to_continue:_").toUpperCase();
        terminal.println("Please_enter_SEND_or_WAIT_to_continue:_ " + chosenState);
        if (chosenState.contains(WAIT)) {
            terminal.println("Waiting_for_messages.");
            return;
        } else if (chosenState.contains(SEND)) {
            sendMessage();
        } else {
            terminal.println("Invalid_input.");
        }
    }
}
```

Listing 9: The start method of the EndNode class. Allows the user to either send a message by calling sendMessage or wait for input by breaking out of the function, so that the listener is active. This function is called again at the end of sendMessage and onReceipt, representing the break from the sending and waiting states respectively.

sendMessage is a typical function that takes the destination EndNode and message from the user and sends the relevant packet. onReceipt is a typical function that either prints messages from incoming packets or sets the destination address of the EndNode, depending on the packet type. The EndNode was by far the easiest class to implement as it did not involve networking, only user interaction and sending and receiving packets with its nearest Switch. A sample output for an EndNode is shown in figure 5.

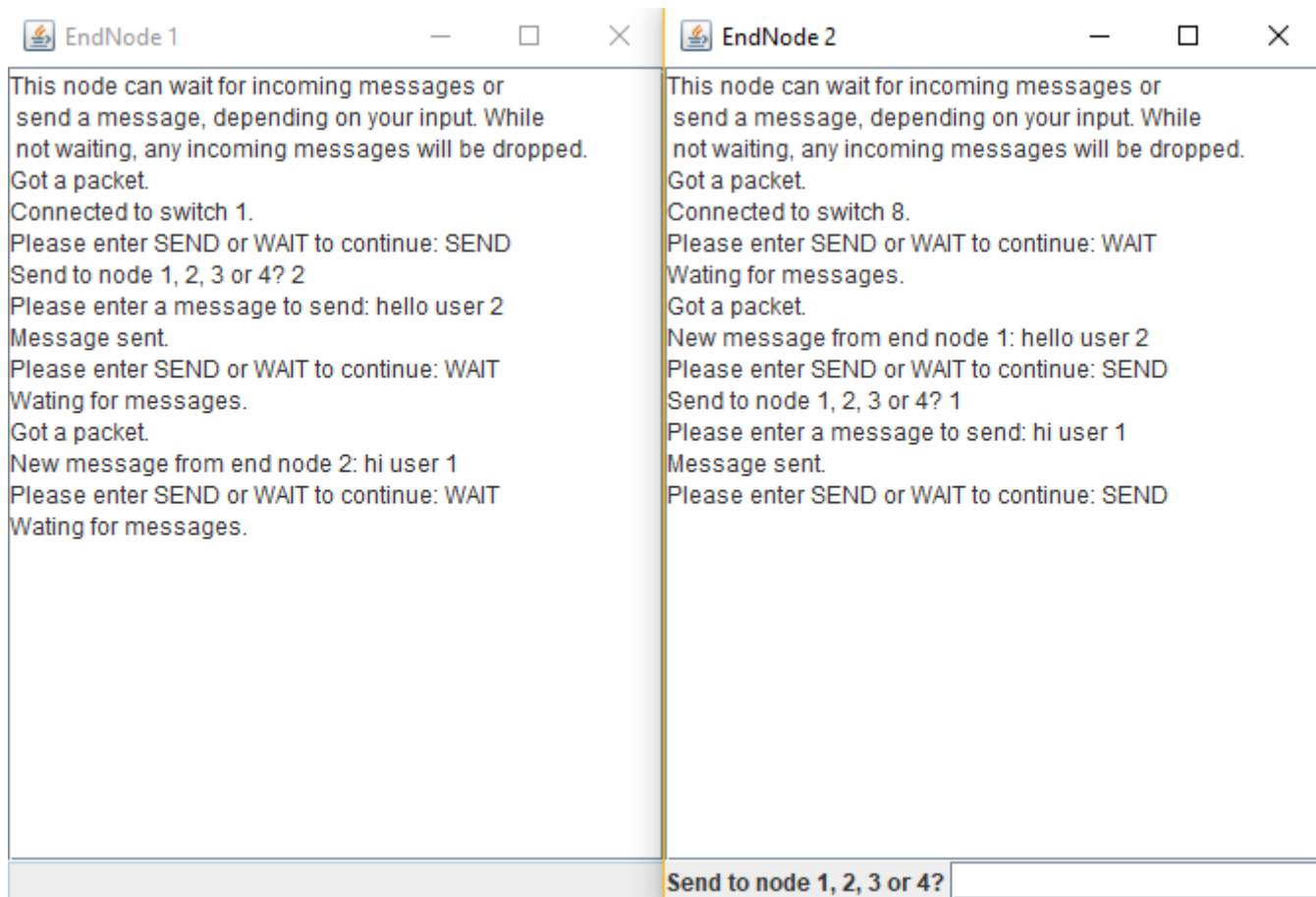


Figure 5: Interaction between two EndNodes. Pictured; user warning that packets will drop if not waiting, resolution of nearest Switch, sending messages, waiting on and receiving messages.

## 4 Discussion

In this section I will discuss my protocol and possible expansions that I would make if I had more time. With this assignment I made sure to implement a protocol that I could finish within time constraints, as I used up too much time in Assignment 1. It is therefore easy for me to identify areas which I understand in theory but would not have time to implement in Java.

### 4.1 Distance Vector Routing

One suggested extension to the assignment that I did not have time to implement was Distance Vector Routing. In this, each node contains a table of triples; Destination, cost and next hop. Each node exchanges its table with directly connected neighbours either periodically or whenever the table changes. If the cumulative cost for a destination via information from a received routing table is lower, the node alters the cost and next hop for that destination in its own routing table. Implementing this through OpenFlow could involve each Switch contacting the Controller when it receives a routing table from a neighbour. The Switch could send the received table and its own table to the Controller, who could carry out Distance Vector Routing and send the new flow table back to the Switch.

## 4.2 Link State Routing

Another suggested extension for the assignment was to implement Link State Routing. In this a node shares knowledge about its immediately connected nodes with every router in the network. Messages from every node in the network allow each node to build a view of the complete topology. The node then uses Dijkstra's Shortest Path algorithm to figure out a list of tentative paths and save the paths of least cost to each node as permanent paths. Implementing this through OpenFlow could involve the Switch knowing the cost of a hop to its immediately connected neighbours. When a Switch receives information from all other Switches about their neighbours, it could forward that information to the Controller. The Controller could then build a complete view of the topology for that Switch, use Dijkstra's Shortest Path algorithm and send the flow table back to the Switch.

## 4.3 Dynamic Change of Flow Tables

The current implementation of the assignment involves a fixed flow table for each Switch based on the preconfiguration information of the Controller. An extension to this assignment could allow the flow tables to change during the course of the program. For example, the Controller could take user input which defines a new path between Switches. Another example is that a Switch that is not directly connected to an EndNode could be removed from the network. After this the Controller would have to work out new paths replacing paths in the network that included the deleted node.

## 4.4 Threading

Once again fully understanding threading in Java would have been useful to me in this assignment. I managed to get multiple Switches and EndNodes to work in the network concurrently without using threading too much, but I still had the issue of an EndNode not being able to wait on both input and incoming packets at the same time. Ideally the EndNode could have two threads; one which waits on user input and always allows the user to send messages. The second thread would wait on incoming packets. When a packet is received this thread would pause the program for a short amount of time to print the message to the terminal, before handing control of the EndNode back over to the input thread.

## 5 Summary

This report has described my attempt to create my own version of the OpenFlow Protocol using Java. Overall my implementation consisted of a Terminal class from Blackboard, an abstract Node class, a Controller class, a Switch class and a EndNode class. The three concrete Node classes all inherit from the abstract Node class, which provides common functionality to all three. Multiple instances of the Switch and EndNode classes are created in the program. The features of the overall system can be summarised as the following:

**Abstract Node Class:** Constants for message types and other constants for the protocol, the Listener Class, abstract onReceipt method, setting and getting packet types, getting the source, destination and content of an EndNode message

**Controller:** Constants for the preconfiguration information of the network, initialising all classes in the program, starting Switches, handling Hello packets, sending Hello packets, sending Feature Requests, handling Feature Replies, sending flow tables, handling Packet Ins, telling a Switch to drop a packet, displaying user output

**Switch:** Sending Hello packets, handling Hello packets, handling Feature Requests, sending Feature Replies, handling Flow Mod packets to get its flow table, parse routing table to inform a neighbouring EndNode of connection, forward packets according to flow table, drop packets at request of controller, displaying user output

**EndNode:** Parsing initialisation message to work out its destination address, take user input, handle errors in user input, send messages to other EndNodes, wait for incoming packets, displaying user output

## 6 Reflection

When I started this assignment I took the decision to keep my implementation relatively simple. In the last assignment I had tried to implement advanced features such as breaking up packets that were longer than the maximum size and sequence numbers with the view of implementing flow control. As a result I spent too much time on the assignment and was not happy with the outcome, as my subscriber could not subscribe and unsubscribe from multiple topics as I wanted. Keeping my implementation for assignment 2 simple certainly helped me and I was pleasantly surprised that I was able to get it finished on time.

In this assignment I also had the benefit of working with DatagramPackets and DatagramSockets for almost a full semester. As a result I did not run into any of the same errors that took up much of my time over Assignment 1.

Making all Nodes initialise from one mainline really helped me to understand the program. I could see exactly where I was getting any errors on startup and also could see the exact sequence of events in the program. It also made it much more pleasant to test as I only had to press run once every time, as opposed to three times in Assignment 1.

I certainly feel like I reinforced the knowledge learnt in lectures while doing this assignment. Software Defined Networking and OpenFlow are only truly understandable to me when you have to implement it yourself. It taught me a lot about the importance of the Controller defining the paths in the network and of OpenFlow as an interface between the Controller and Switches. Overall I am very happy with how this assignment went and am proud of the work done.