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

CS2031 Telecommunication II
Assignment #2: Openflow

Ciaran Coady, 17326951

December 29, 2018

Contents

1 Introduction

The problem description for this assignment outlined a network consisting of one controller, a number of switches and a number of endpoints. This network is to operate much like an openflow network does and is to incorporate some of the features of an openflow network.
In this report, I will discuss the main features of my solution, along with some theory of how it works. Followed by the implementation and an explanation of the code used.

2 Overall Design

Looking at the project specification, it was required that the following features were implemented:

Controller: Accept contact from switches and issue feature requests. Accept PacketIn messages. Send out FlowMod messages.

Switch: Make contact with the controller and reply to feature requests. Send out PacketIn messages. Receive and incorporate instructions from FlowMod messages. Receive messages from end-nodes. Forward messages depending on the flowtable.

End-Node: Send messages addressed to another end-nodes. Receive messages.

2.1 Openflow

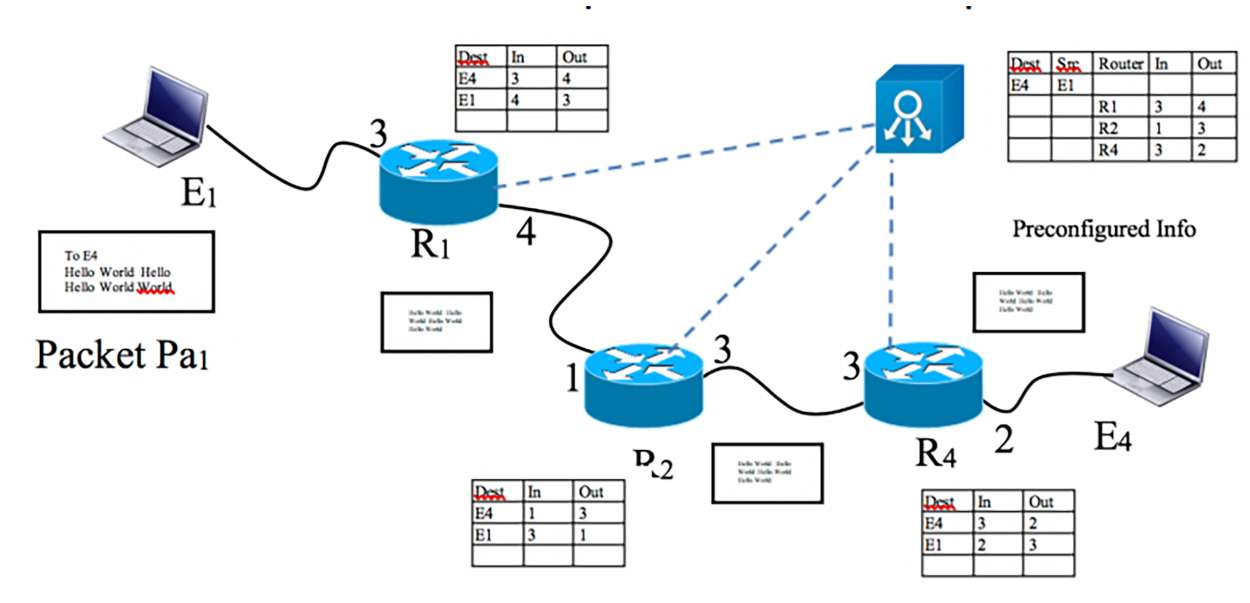

Figure 1: The figure above demonstrates a simple topology of a network using Openflow. The switches in the network are connected to a controller. A switch will contact this controller when an incoming packet has no matching rules in the flowtable.

Openflow is a networking method that uses switches and controllers to route packets from one endpoint to another. Each switch is connected to a controller, other switches or end-nodes. Each switch uses a flowtable which it has received from the controller to route packets to the correct destination. This flowtable consists of a number of rules that influences where incoming packets get routed. If an incoming packet matches a field in the flowtable, that rule gets used. If a switch received a packet that is not in its flowtable it will contact the controller to modify its flowtable so it knows where to route this packet. An End-node will want to send packets to other end-nodes. When a packet is sent it moves from the end-node to the switch that it

is connected to. The switch then routes the packets based on its flowtable. The packet moves from hop to hop in this manner until it reaches its destination or gets dropped.

The Openflow standard defines a number of packet types:

Hello: Is an announcement from a network element to a controller that it is alive and assumes to be under the control of the controller.

FeatureRequest: A FeatureRequest is sent by the controller, after a hello packet is received from a switch in the initialization stage. The purpose of this packet is for the controller to gather information about the switch, which it will use to configure flowtables.

FeatureReply: This is the response to a FeatureRequest and is sent by a switch to the controller. The switch will send its identity and basic capabilities, such as what connections it has to the controller.

PacketIn: This packet is sent from a switch to the controller if a packet is received that doesn't match its flowtable. The packet received by the switch is forwarded to the controller either in full or partially for further action. The controller will then decided where the destination of the packet is and send a FlowMod or PacketOut packet to the switch.

PacketOut : These are used by the controller to send packets out of a specified port on the switch, and to forward packets received via Packet-in messages.

FlowMod: These are used by the controller to update the FlowTable of a switch.

2.2 Link State Routing

Link State routing consists of two steps: Establishing a view of the topology and executing a shortest path algorithm such as Djikstra's Shortest Path. In the first step, routers in a network would measure their connectivity to their neighbours e.g. the latency on each port to a router or endpoint connected to that port, and broadcast this information to all other network elements in the network. Every router will gather this information for a certain period and then execute a shortest path algorithm on the data that has been received during that period.

Due to time constraints the system that was implemented only operates using hard coded routing tables, but for a small scale implementation Djikstra's Shortest Path and other link state routing methods are not really necessary.

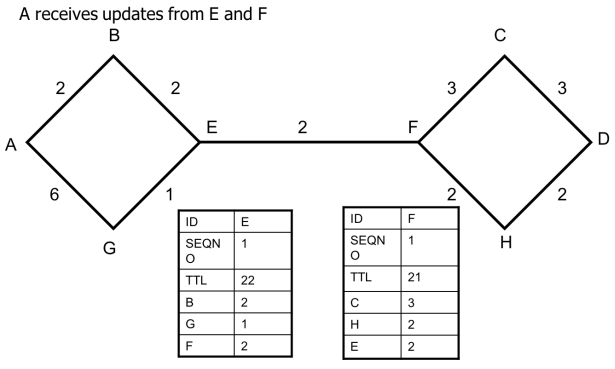

Figure 2: An example of Link State topology

Djikstra's Shortest Path algorithm is a greedy algorithm that pursues in every step the subsequent shortest path, starting with the router where it is executed as origin. In a first step, the current node e.g. router itself would be added to the routing table, then the neighbours of this node would be added to a list of nodes, which have not yet been added to the routing table. The cost to reach these pending nodes is the

cost from the origin to the current node, plus the cost from the current node to the neighbour. In the next step, it will pick the node with the lowest cost out of the list of pending nodes, add it to the routing table and repeat the first step with this node as current node.

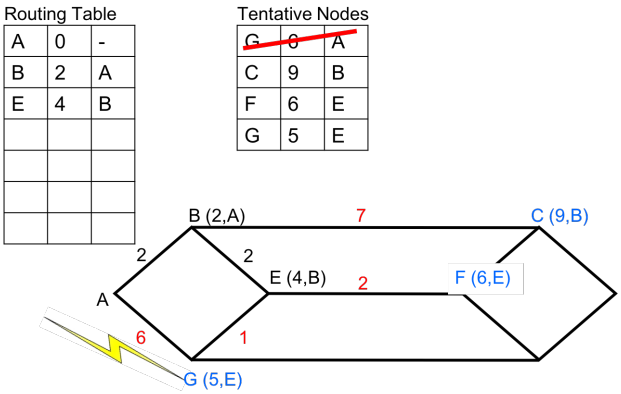

Figure 3: An example for Djikstra's Shortest Path

2.3 Design of Network Elements

For this assignment it was crucial that each network element had the facility for user interaction. Based on my experience from assignment 1, I decided to use the tcdIO library again, allowing me to make use of the terminal. This gave good separation between each network element and allowed for an easy to follow, easy to debug program flow.

To allow for scalability in the future, each network element is configured at run-time using the terminal. Switches are initialized using a switch number, as are the end-nodes. This way extra network elements could be added for future expansion. When each network element is initialized, it gets allocated a port number by adding its number to a constant. e.g. the starting switch port number is 5002, so switch number 1 will be allocated port number 5002+1=5003.

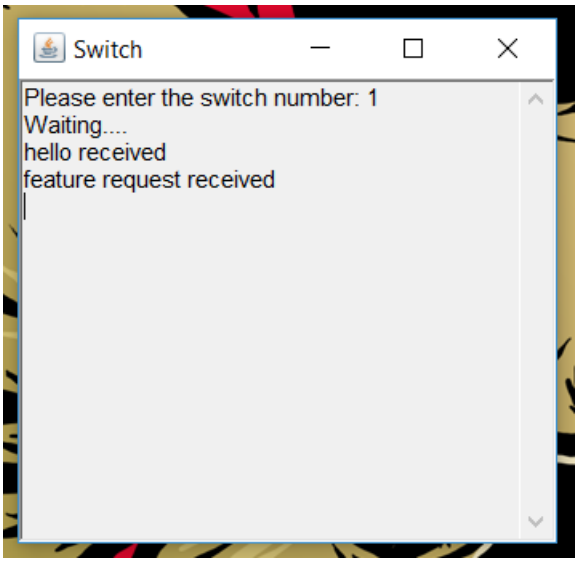

Figure 4: This shows the initialization process for switch number 1

2.4 Packet Descriptions

Each packet is created using the packet class. A packet is first given a header, byte 0 is what type of packet it is e.g. FlowMod, PacketIn.
Byte 1 is the end destination that the packet is trying to get to.
Byte 2 is the overall length of the packet.

The remaining bytes in the packet then contain the message that is being sent. In the case of a PacketOut packet the message is the string that the end-user has entered in the terminal. In the case of a FlowMod packet, the message portion will contain the updated flowtable that is being sent from the controller to the switch.

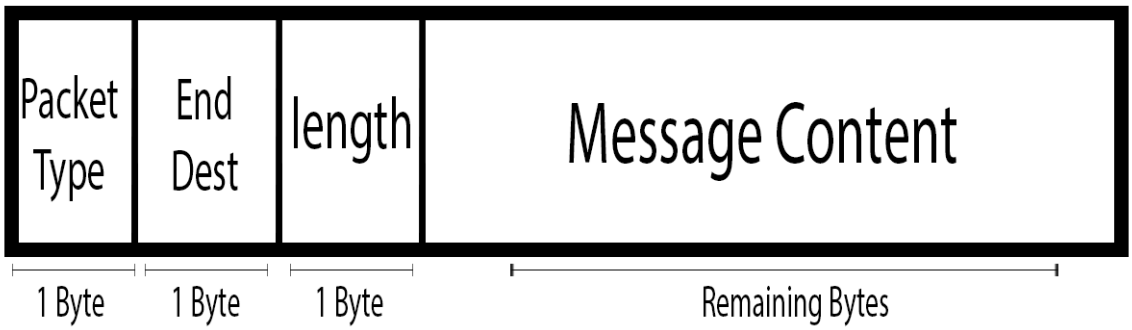

Figure 5: The packet layout that was used in this assignment

3 Implemenation

This section presents the implementation details of the individual network elements, the controller, the endpoints and switches. It also discusses the layout of the messages between the endpoints and the packets exchanged between the switches and the controller.

3.1 Node

Identical to the last assignment, All network elements are an extension of the node class. In this case the only pieces of functionality contained in the Node is the initialization of a listener thread that continuously monitors the node's port waiting for packets. The Node also includes the functionality to send a packet, by using socket.send() to send the packet to the destination. Putting this code in the Node made for neater code elsewhere.

3.2 Packet

The packet class holds all of the packet formatting and encoding functionality. A Packet object has all of the included methods of the Packet class and also a private DatagramPacket which can be accessed using the getPacket() method.

The first method included is the addHeader() method. This takes the header information and puts it into a passed byte array in the correct locations.

The addMessageContent() method takes a packetContent byte array and a String message. It converts the String into a byte array and places it into the packetContent byte array using a for-loop.

The featureRequestPacket() method takes the portNumber of the switch that is getting sent the featureRequest, the number of the switch and the entire flowtable from the controller. Using a for-loop the rows of the flowtable that are relevant to the switch are converted to a string. The string is then bundled into a new packet that will be sent to the switch.

```
//feature request packet
public void featureRequestPacket(int portNumber, int switchNumber,  int configTable[][]) {
    DatagramPacket packet;
    //processing config table so it can be sent to the switch
    String content = "";
    for(int i = 0; i < configTable.length; i++) {
        if(configTable[i][2] == switchNumber) {
            content = content + configTable[i][0] +
                    "," + configTable[i][3] + "," + configTable[i][4] +
                    ",";
        }
    }
    //setup the array of bytes to be put in the packet
    byte[] packetContent = new byte[content.length() + 3];
    //add the header
    addHeader(packetContent, Constants.FEATURES_REQUEST, Constants.CONTROLLER_PORT_NUMBER, packetContent.length);
    //add the content
    addMessageContent(packetContent, content);
    //get the address of the packet
    InetSocketAddress dstAddress = new InetSocketAddress("localhost", portNumber);
    //create the packet
    packet = new DatagramPacket(packetContent, packetContent.length, dstAddress);
    this.packet = packet;
}
```

Figure 6: The featureRequestMethod() a described above

The featureReply() method, helloPacket() method and packetIn() method all share the same code but were made into separate methods for future expansion and also helping make the code more understandable. These methods take in a portNumber and nodeNumber as parameters. The node number is used when making the header of the packet and the port number is used when setting the packet destination.

```
byte[] packetContent = new byte[3];
// add header
addHeader(packetContent, Constants.PACKET_IN, nodeNumber, packetContent.length);
InetSocketAddress dstAddress = new InetSocketAddress("localhost", portNumber);
//create the packet
packet = new DatagramPacket(packetContent, packetContent.length, dstAddress);
this.packet = packet;
```

Figure 7: This shows the method body that is shared by the featureReply(), helloPacket() and PacketIn() methods

The packetOut() method is used when a node is sending a packet to the next hop in the network. The parameters are the portNumber that packet is being sent to, the nodeNumber which is the endDestination of the packet and the message that is to be sent. Much like the other methods, a byte array gets created, the header is added and the message is inserted. After this the destination address is set and the packet is stored in the local DatagramPacket variable.

```
// packetOut packet
public void packetOut(int portNumber, int nodeNumber, String message) {
    //TODO add error checking if the message is too big to fit into a packet
    DatagramPacket packet;
    byte[] packetContent = new byte[message.length()+3];
    // add header
    addHeader(packetContent, Constants.PACKET_OUT, nodeNumber, packetContent.length);
    addMessageContent(packetContent, message);
    InetSocketAddress dstAddress = new InetSocketAddress("localhost", portNumber);
    //create the packet
    packet = new DatagramPacket(packetContent, packetContent.length, dstAddress);
    this.packet = packet;
}
```

Figure 8: This shows the packetOut() method

The last method included in the packet class is the flowMod() method. This has identical functionality to the featureRequestPacket() method, with the exception that packet type is different. Much like with other duplicate methods, I decided to keep them as separate functions in case I had time to implement extra features.

3.3 EndNode

The EndNode uses similar code to the subscriber class from the last assignment. The interface of the EndNode is used to read the message to be sent and the EndNode that the user wishes to send the information to. When an EndNode is first started it asks what EndNode number it is and what switch it is connected to. This information is then passed into the constructor of the EndNode and its port number along with other information gets intialised.

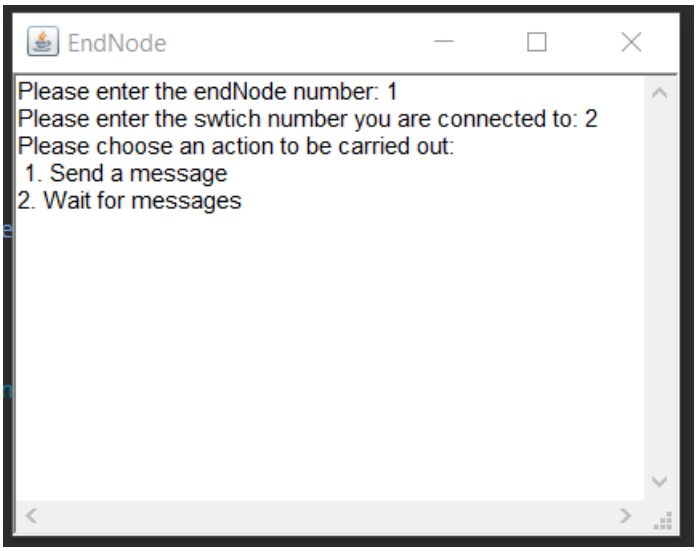

Figure 9: The interface that the user interacts with when an endnode is first started

```
int endNodeNumber = Integer.parseInt(terminal.readString
                ("Please enter the endNode number: "));
int connectedSwitch = Integer.parseInt(terminal.readString
("Please enter the swtich number you are connected to: ")) +
          Constants.SWITCH_BASE_PORT;
(new EndNode(terminal, endNodeNumber, connectedSwitch)).start();
```

Listing 1: Upon intialising the Endnode, the user is being asked to input the Endnode number and the switch it is connected to

The constructor then takes these values and initializes the EndNode with its own port and listener thread. If an Endpoint were to receive a packet it would run the onReceipt() method. This method prints the received string to the terminal.
The composeMessage() method takes no parameters and is called when a user wants to send a message. It asks the user what they would like to send and to who they want to send it to. These strings are read in from the terminal and used to create a messagePacket which is a PacketOut packet.

```
public void composeMessage() {
    int destination = Integer.parseInt(terminal.readString("Please enter the end user you want to send to: \n"));
    String message = terminal.readString("Please enter the message you wish to send: \n");
    Packet messagePacket = new Packet();
    messagePacket.packetOut(this.connectedSwitch, destination, message);
    sendPacket(messagePacket);
}
```

Figure 10: The composeMessage() Method

Once constructed, this packet is sent on to the Switch that this Endnode is connected to using the sendPacket() method included in the node class.

3.4 Switch

The Switch is initialized identically to the EndNode. When a switch is first started up, it asks the user what switch number it is and then proceeds to use this information in the constructor.
After the switch has been initialized there is no more interaction from the user, the terminal is only used to display information about what packets are being sent from the switch and received by the switch.

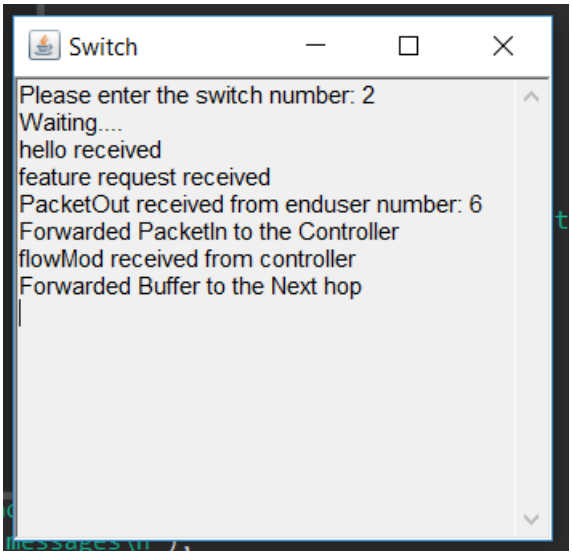

Figure 11: Above is the output from a switch in its normal operation

After a switch gets initialized it sends a hello packet to the controller. The controller then responds with a hello packet followed by a FeatureRequest packet. In this implementation the controller has the overall flowtable for the example topology and sends the relevant flowtable information in the FeatureRequest packet. When the Switch receives the FeatureRequest packet it processes the flow table and stores it locally using the processFlowTable() method. Once this is complete the Switch responds with a FeatureResponse packet. If I had implemented link state routing this packet would contain information about the Switch that would allow the controller to calculate shortest paths.

```
public synchronized void onReceipt(DatagramPacket packet) {
    byte[] packetContent = packet.getData();
    if(packetContent[0] == Constants.HELLO) {
        //our hello has been acked
        terminal.println("hello received");
    }
    else if (packetContent[0] == Constants.FEATURES_REQUEST) {
        //send feature response
        terminal.println("feature request received");
        //process received table
        processFlowTable(packetContent);
        Packet response = new Packet();
        response.featureReply(packet.getPort(), this.switchNumber);
        sendPacket(response);
    }
```

Figure 12: This shows the onReceipt() method snippet that is involved with the intialisation of the switch

In the processFlowTable() method the flowtable is first removed from the message portion of the packet. Then due to the fact that any unused space in the packet gets padded with 0 bytes the java String class is used to remove and 0's. The string is then converted back to the UTF-8 character set. After this the string is then inserted into the local flowtable using a for-loop and some modulo arithmetic.

```
public void processFlowTable(byte[] packetData) {
    byte[] temp1 = new byte[packetData.length-3];
    for(int i = 0; i < temp1.length; i++) {
        temp1[i] = packetData[i+3];
    }
    byte[] temp = new String(temp1).replaceAll("\0", "").getBytes();
    String flowTableString = "";
    try {
        flowTableString = new String(temp, "UTF-8");
    } catch (UnsupportedEncodingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    String[] flowTableRows = flowTableString.split(",");
    flowTable = new int[flowTableRows.length/3][3];
    int row = 0;
    for(int i = 0; i < flowTableRows.length; i++) {
        if(i != 0 && i%3 == 0) {
            row++;
        }
        flowTable[row][i%3] = Integer.parseInt(flowTableRows[i]);
    }

}
```

Figure 13: This shows the processFlowTable() method

When a Switch receives a packetOut packet from another network node, it caches this packet to a buffer and sends a packetIn packet to the controller. Then the controller makes a decision on where the switch should forward the packet. The controller sends back a FlowMod packet, which the switch implements and then using this updated flowtable, forwards the packet. In this implementation the the flowtable is hard-coded and doesn't change after the initialization stage of the program. Therefore packetIn packets and this functionality are not really used, but would allow for easy expansion if the feature set of the code was to be improved in the future.

```
else if (packetContent[0] == Constants.PACKET_OUT) {
    terminal.println("PacketOut received from enduser number: " + (packet.getPort() - Constants.ENDUSER_BASE_PORT));
    //save it to the buffer
    if(buffer == null) {
        buffer = packet;
    }
    //convert to packetIn and send to controller
    Packet packetIn = new Packet();
    packetIn.packetIn(Constants.CONTROLLER_PORT_NUMBER, packetContent[1]);
    sendPacket(packetIn);
    terminal.println("Forwarded PacketIn to the Controller");
}
```

Figure 14: This shows the section of the onReceipt() method that handles a packetOut packet. The packet gets saved to a buffer followed by the switch sending the end destination of the received packet to the controller in a new packetIn packet

```
else if (packetContent[0] == Constants.FLOW_MOD) {
    terminal.println("flowMod received from controller");
    //processFlowTable(packetContent);
    forwardPacket();
}
```

Figure 15: This shows the section of the onReceipt() method that handles a FlowMod packet. When a FlowMod packet is received, the updated flowtable it includes gets implemented, then this flowtable is used to forward the packet that is contained in the buffer

```
public void forwardPacket() {
    byte[] packetContent = this.buffer.getData();
    boolean found = false;
    for(int i = 0; i < this.flowTable.length && !found; i++) {
        if(packetContent[1] == flowTable[i][0] && this.buffer.getPort() == flowTable[i][1]) {
            this.buffer.setPort(flowTable[i][2]);
            found = true;
        }
    }
    try {
        socket.send(this.buffer);
        terminal.println("Forwarded Buffer to the Next hop");
        this.buffer = null;
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 16: This shows the forwardPacket() method. This method takes no parameters and has no return values. It extracts the byte array from the packet stored in the buffer. Then using a for-loop moves through the flowtable until it finds the rule that corresponds to the end destination of the packet. If a rule is found the destination of the next hop gets updated and the packet is sent to the next hop. After this the buffer gets cleared

3.5 Controller

The initialization of the controller is very similar to a basic version of the broker from the last assignment. The constructor sets the port number for the controller which is based on a constant and starts its listener thread.

The broker has no interaction with the user, but instead acts much like a receiver, receiving packets from various sources and deciding what to do with them. The mainline of the controller calls its constructor and after this continuously loops using a while(true) loop, to wait for incoming packets.

The controller holds the main flowtable for this system, which is stored in the form of a 2D array of integers. Unfortunately I didn't get time to implement link state routing or any distance based routing algorithm, as a result the flowtable in the controller is set before run-time and doesn't change.

```
private final int[][] preconfigTable = {
        //{Dest                    source    SWITCH              in                      out  }
        {Constants.ENDUSER2, Constants.ENDUSER1, 1, Constants.ENDUSER1_PORT_NUMBER, Constants.SWITCH2_PORT_NUMBER},
        {Constants.ENDUSER2, Constants.ENDUSER1, 2, Constants.SWITCH1_PORT_NUMBER, Constants.SWITCH3_PORT_NUMBER},
        {Constants.ENDUSER2, Constants.ENDUSER1, 3, Constants.SWITCH2_PORT_NUMBER, Constants.ENDUSER2_PORT_NUMBER},
        {Constants.ENDUSER1, Constants.ENDUSER2, 1, Constants.SWITCH2_PORT_NUMBER, Constants.ENDUSER1_PORT_NUMBER},
        {Constants.ENDUSER1, Constants.ENDUSER2, 2, Constants.SWITCH3_PORT_NUMBER, Constants.SWITCH1_PORT_NUMBER},
        {Constants.ENDUSER1, Constants.ENDUSER2, 3, Constants.ENDUSER2_PORT_NUMBER, Constants.SWITCH2_PORT_NUMBER},
};
```

Figure 17: This shows the 2D array that stores the master flowtable in this topology

Most of the functionality of the controller occurs in the onReceipt() method shown below in figure 18. If the controller receives a hello packet from a switch, it send a hello packet in response to that switch. Following this the controller sends a featureRequest packet which contains the relevant section of the flowtable for that Switch. All of the functionality behind formatting these packets is contained in the packet class, leaving the controller code easy to read.

```
byte[] packetContent = packet.getData();
if(packetContent[0] == Constants.HELLO) {
    terminal.println("Hello received from switch number: " + (packet.getPort() - Constants.SWITCH_BASE_PORT));
    //send hello in response
    Packet response = new Packet();
    response.helloPacket(packet.getPort(), packetContent[1]);
    sendPacket(response);
    //send feature request
    Packet featureRequest = new Packet();
    featureRequest.featureRequestPacket(packet.getPort(), packetContent[1], this.preconfigTable);;
    sendPacket(featureRequest);
}
```

Figure 18: This shows the section of the onReceipt() method that deals with a hello packet as described above

In this implementation if the controller receives a packetIn packet from a switch, it just takes the switch's number and returns its flowtable as done previously when sending a feature request packet. A packetIn packet from a switch signifies that the switch either doesn't have an entry in it flowtable to forward this packet or is asking the controller what is the best place to send it. If a more advanced algorithm had been implemented, the controller would do some calculations and decide where to send the packet based on various factors. The controller sends this flowtable using a FlowMod packet.

```
else if (packetContent[0] == Constants.PACKET_IN) {
    terminal.println("packetIn received from switch number: " + (packet.getPort() - Constants.SWITCH_BASE_PORT));
    Packet flowMod = new Packet();
    flowMod.flowMod(packet.getPort(), packetContent[1], this.preconfigTable);
    sendPacket(flowMod);
    terminal.println("flowMod send to switch number: " + (packet.getPort() - Constants.SWITCH_BASE_PORT));
}
```

Figure 19: This shows the section of the onReceipt() method that deals with a packetIn packet as described above.

4 Discussion

The strengths of this program are its simplicity and expand-ability. I only used a small sample topology, but to expand the system all that would need to be altered would be the master flowtable that is contained in the controller and a few minor pieces of code. It also has most of the structural work laid down when it comes to sending a receiving packets, this means that it would not be too difficult to implement a more advanced routing technique.

This implementation does not include any error checking or Automatic Repeat Request technique. As a result, if packets are sent simultaneously it will result in packets being dropped and the system will break. This could be improved upon by adding some of the Stop and Wait functionality that was included in assignment 1 and extending it to work in this scenario. This would add some extra complication as timeouts and Acknowledgments would have to be introduced to the system, therefore I decided not to implement it.

Another weakness is that all network elements must be initialized one at a time, otherwise the system will fail as hello packets to/from the controller will get dropped.

There is a problem with FlowMod packets in this system. If a switch receives a FlowMod packet from the controller, it does not get parsed correctly, cannot be successfully converted to a flowtable and as a result corrupts the existing flowtable in the switch. This prevents packets from being routed properly. I couldn't track down the cause of this problem as the same method works perfectly for featureRequest packets. After spending hours trying to solve the bug I decided to not process the flowtable received in the flowMod packets. This works fine in this system as nothing changes in the flowtable at any point.
.

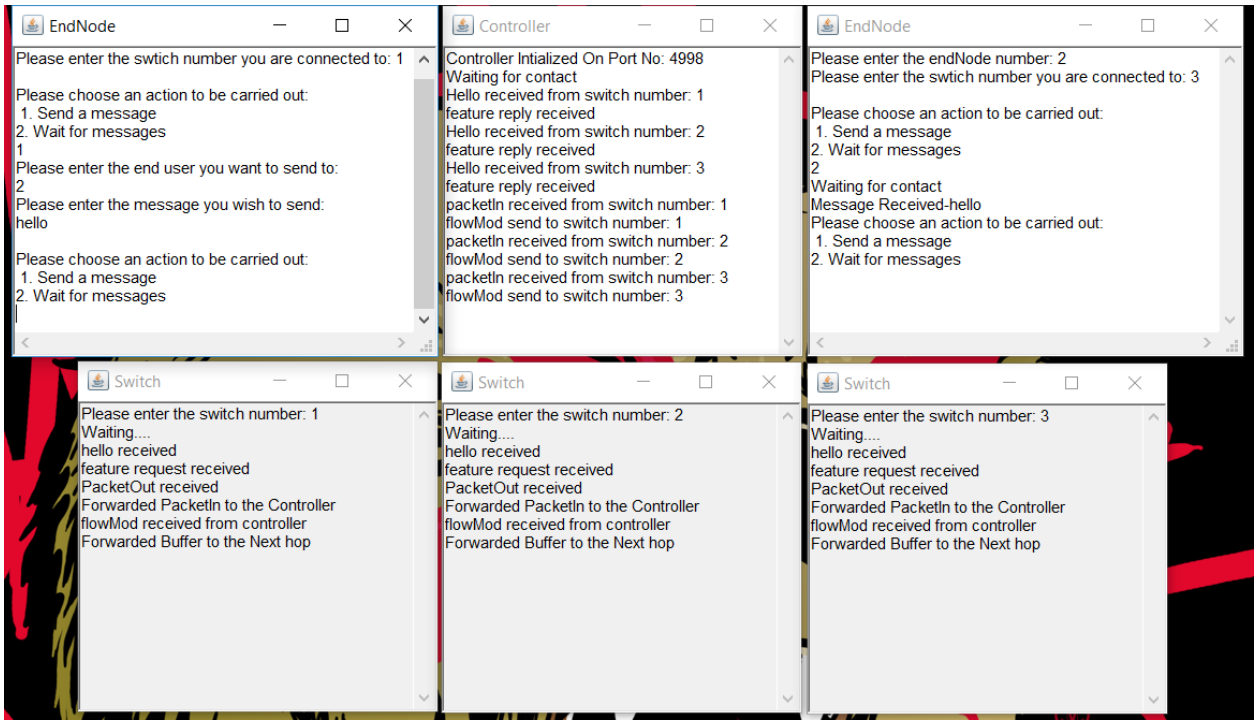

Figure 20: This figure shows terminal windows for each network element in the sample topology while in operation

5 Summary

This report has described my attempt at a solution to address the routing of messages in a small topology based on an Openflow-like approach. I've outlined the background research I did before beginning the project and some of the theory behind features I didn't get to implement such as link state routing. I discussed the implementation details along with some of the design choices that I made through-out the development of this solution. It does have some flaws but I feel like it does a good job of representing how an openflow-like system works.

6 Reflection

I'm much happier with the outcome of this assignment compared to the first one. I learned from the mistakes I made in assignment 1 resulting in this assignment taking much less time, all in all taking about 25 hours to complete. I decided not to use as many code snippets in the place of explainations, only including code snippets where anything interesting or difficult was being carried out. Much like the last assignment I found the labs to be a cruicial way to expand my understanding on the topic and also get help when I was stuck or unsure at any point. I handled the development process much better than before, implementing small chunks of code at a time, slowly building up to the final solution. While I didn't implement any link state routing, I now understand how it works after completing this assignment and am very pleased with my general understanding of openflow.

7 References

-CS2031 notes and example documents.
-Java Documentation - www.oracle.com
-Openflow specifications and documentation - https://www.opennetworking.org