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

CS2031 Telecommunications II

Assignment 2: Routing Protocols

Claire Cassidy

16325301

November 27, 2019

Contents:

1. Introduction

The goal of this assignment was to model a small network topology and implement our own version of the OpenFlow routing protocol to allow efficient routing and communication between nodes on the network. Communication in the network takes place between a number of routers with arbitrary direct connections to other routers, a single controller node which coordinates and optimizes traffic in the network, and a number of endpoints, each with a direct connection to a single router. As per the specifications of the OpenFlow protocol, each router maintains its own routing table which, given an incoming packet's final destination, dictates where next to send the packet next. These routing tables are dynamically updated by messages from the Controller with the intent to optimise traffic in the network given its topology.

In the following sections, this report will first discuss the overall design of my implementation, with attention given to the overall components required to produce a rudimentary simulation of a network using the OpenFlow protocol. Next, I will discuss my implementation in depth, how the classes representing these components are created and how their instances interact. In the subsequent section, I will justify my approach by showing packet captures of the network in action and discuss the strengths and weaknesses in the overall design. This section will be followed by a summary of my implementation and finally my reflections on the project.

2. Overall Design

As the purpose of this project is to simulate a rudimentary version of the OpenFlow routing protocol, it is important to firstly discuss the concepts of Software Defined Networking and the principles of OpenFlow themselves. Next, the concept of Distance Vector Routing will be examined, along with the benefits of including such a concept in the design of a network. The broad components or network elements that needed to be modelled for this assignment will be discussed, namely the network's controller, routers and endpoints. I will also briefly outline the different types of packets the OpenFlow network needs to be able to handle to fulfill the protocol, followed by a brief explanation of the Application class which will be used to run my implementation.

2.1 Software-Defined Networking

Software-Defined Networking is intended as a solution to the emerging shortcomings of traditional network architecture, which is becoming inefficient and obsolete as the modern environment of mobile devices and cloud services expands. It creates an abstract model for switching (moving information in the network to its intended recipient) that can be tailored to a given network's needs on the fly instead of relying entirely on the manual configuration of a network's hardware. Through Software-Defined Networking, admins can achieve a central control point over the entire network and configure network devices remotely, as well as have more flexible control over how the components in their network operate.

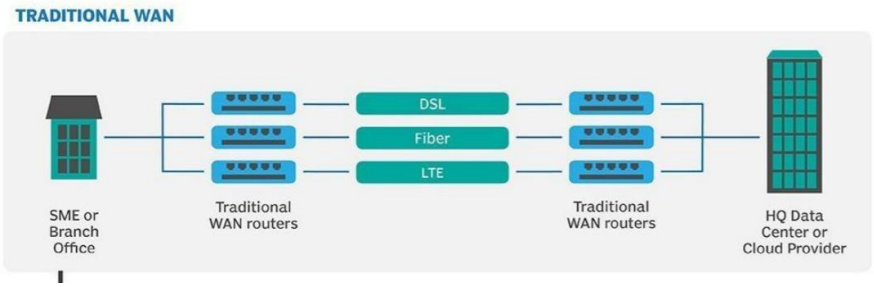


Figure 2.1 1: Traditional network architecture is restricted to conform to the routing specifications defined by the protocols of its individual nodes, and can be difficult to manage on the fly or in large-scale operations

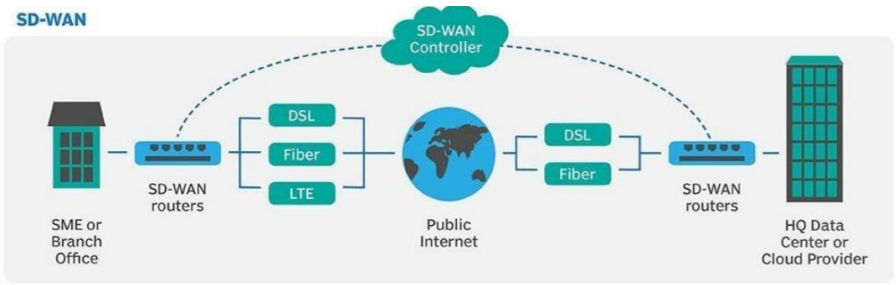


Figure 2.1 2: A SDN allows network configuration to be altered from a single point. Since it is software-defined, configuration possibilities are much more flexible than working with the limitations of the individual nodes.

2.1.1 OpenFlow

OpenFlow is a manifestation of the principles of Software-Defined Networking. It gives an abstract specification of what a Software-Defined Network should look like in terms of components, and what requirements the communications protocol(s) should have. My implementation of OpenFlow uses a controller, multiple routers and multiple endpoints as components. The protocol specifies that any packet received by a router in the network should be forwarded based on the information in the router's routing table. The routing table contains a number of classifiers used to match an incoming packet to a set of instructions on how to route

the packet. In the case of my implementation, the classifier is the final destination (an endpoint) port of the packet, and the set of instructions is simply a port number for the next hop in the transmission. Information in the routing table, and by extension any decision a router makes, comes entirely from the controller; whenever the router finds it does not know how to proceed with an incoming packet (i.e. when there is no classifier in the flow table matching the packet), it contacts the controller for instructions on how to proceed. The controller may then send out an update to the router's flow table with routing instructions, and an update to the flow tables of all subsequent routers along the path as well. In this way, all routing procedures can be tuned precisely to a network's needs and the controller is free to update the routing information at any time by issuing updates to the routers.

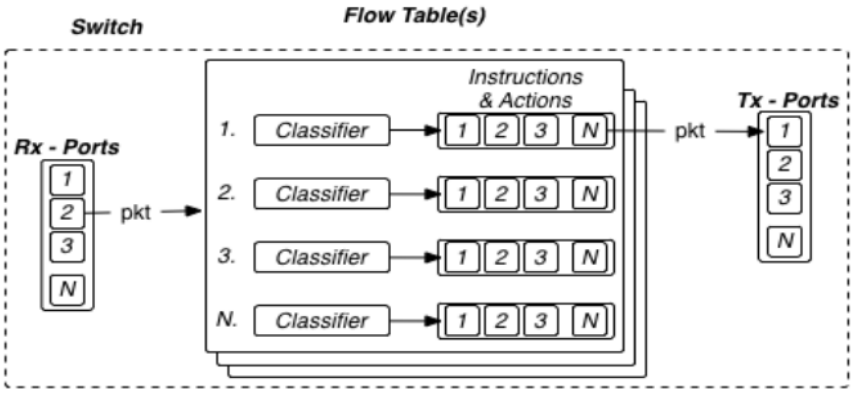


Figure 2.1 3: Figure showing the process a router undertakes when deciding how to route a packet; packets entering any port are matched to a flow table entry to find the actions to be performed based on the packet's information.

2.2 Distance-Vector Routing

It is expected that the controller makes intelligent decisions on the routing instructions it disseminates based on the topology of the network. In my implementation, this is achieved through Distance-Vector Routing (DVR). In DVR, the network determines a best route for any packet by choosing the route with the minimum number of hops required to reach its destination. This is convenient to implement in conjunction with OpenFlow, as the controller is designed such that it has an overview of the topology of the network and how individual nodes are connected. This means it can determine the best route independently without having to communicate with other nodes in the network, given only a start node and an end node. My implementation of DVR depends on a breadth-first search of connected nodes represented by an adjacency matrix, and will be discussed in detail in the section on Implementation.

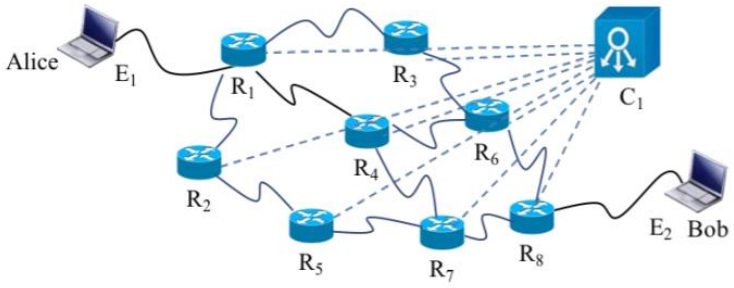


Figure 2.2 1: Distance-Vector Routing would allow us to favour the path <R1, R4, R7, R8> over the path <R1, R3, R6, R4, R7, R8>, for example, in the sample topology

2.3 Design of Network Elements

As stated previously, the network elements involved in the OpenFlow protocol are controllers, routers, and endpoints. For simplicity, each of the network elements is designed with a single port for incoming and outgoing packets, and all communicate using Java sockets and Java DatagramPackets containing the whole packet (header and frame) in its buffer.

2.3.1 Controller

The controller serves as the brain of the network, and is responsible for determining routing instructions for each router based on an implementation of DVR and its overview of the network topology. It establishes this topology by sending a Feature Request to any router that has contacted it, and obtaining the port number of the router and any endpoints that are directly connected to it. The controller stores the topology as an adjacency matrix, or the set of edges that exist between nodes a graph of the network.

Once a router contacts the controller about a packet's endpoint for which it has no routing instructions, the controller will generate the best possible path for the packet given the current state of the topology in the adjacency matrix. By running a breadth-first search of the graph, the controller is guaranteed to find the shortest route to the packet's destination. If the controller cannot find a path to the destination, it sends back an *Error* message to the router that contacted it, and allows it to pass this information on to the client in whatever way it sees fit. If a path is found, the Controller sends an update to all routers along the path in the form of an update to their routing tables. This means that every router on the path doesn't need to send individual messages to the controlling regarding the given packet and slow down the transmission. The information provided by the controller consists simply of a key-value pair of endpoint and next hop on the route for that particular router. If the update has a *null* value for the next hop, it means that that router is the last router in the path and has a direct endpoint connection to pass the packet along.

2.3.2 Router(s)

The network consists of many RouterGeneric instances at different ports. Each router knows the port address of the Controller and what Endpoints it is directly connected to, but it is unaware of any other routers in the network or their connections. It's decision on what ports to forward particular packets to is based entirely on the information given to it by the controller. This helps remove any potential autonomy for the router and ties control directly back to the central control point.

On startup, a router initiates contact with the controller with a *Hello* message. The controller responds and sends a Feature Request, to which the router responds by sending the port number it will use to send and receive packets and the list of endpoints it is connected to. This information will be used by the controller to build routes to particular endpoints.

At this point, the router enters an endless loop of waiting to receive packets either from endpoints, other routers or the controller. Any messages from the controller will be flow table modifications, and give the router information on how to handle packets with a particular endpoint. Packets from other routers will be forwarded message packets, whose endpoint the router will check against its routing table to see how to proceed. If a router receives a packet whose endpoint is not listed in its routing table, it will contact the controller and store the packet while it wait for a response. Once the response is received, the packet is sent on its way. Under correct operation, only the first router along a path to an endpoint will have to contact the controller; never one part-way along the path. The same start-endpoint to destination-endpoint packet route will never require contacting the controller again, as routing information is available in the flow tables.

2.3.3 Endpoint(s)

Endpoints represent the end-users of a network. End-users can request to contact other endpoints within the network (any receiver endpoint without a route to access it through is not part of the network from the sender endpoint's perspective). The implementation of the network (such as router connections and the controller) is hidden from the endpoint; all it needs to know to send a message is the port address of its router (the post-office) and the port address of the endpoint to send it to (the message recipient's home address).

The endpoint sends out a single kind of packet; the *message* type. This packet's destination address is marked as the endpoint address and is parsed by routers to move the packet along the path to the recipient. In my implementation, messages are created via the terminal by asking the user for the String message they wish to send and the port number of the

recipient endpoint. If they choose an endpoint the is not in the network, the router will know and issue an error to the user. The endpoint is also constantly listening for incoming packets from other endpoints that arrive via its router, which will be printed to the screen.

2.4 Packet Descriptions

The OpenFlow documentation describes numerous different types of packets that are exchanged between the controller and routers to implement the protocol correctly. These packet types were encoded within my packet layout (to be discussed in Implementation: Packet Encoding) and checked by nodes in the network to determine the correct course of action to proceed with based on the specifications of the OpenFlow protocol. Of use in my implementation were the following types:

- ***Hello* message:** this message is sent between each router and the controller to initiate contact
- ***Feature Request* message:** upon establishing contact with a router, the controller sends it a *Feature Request* message to ask for the port number it intends to use to send and receive messages, along with what endpoints its directly connected to.
- ***Feature Response* message:** this message is sent by the router on receipt of a *Feature Request* to provide the requested information
- ***Message* message:** (not defined in OpenFlow) this is the generic packet containing the String message to be routed from one endpoint to another. It contains a field for the recipient endpoint used by routers to check their flow tables for instructions.
- ***Flow Request* message:** (named *PacketIn* in documentation) message sent from router to controller in the event that the router has no flow table entry for an incoming packet.
- ***Flow Mod* message:** the response the controller sends to a *Flow Request* message. It is issued to one or more routers (with information changed accordingly) along a path from endpoint to another and is used to update flow tables.
- ***Error* message:** message sent between nodes for a variety of reasons when an error occurs carrying out the protocol. The context in which an error arises is enough to know how to proceed in this limited simulation of OpenFlow, so only one type of error message is defined.

2.5 Using the Application

To allow the demonstration to run all windows at once, the 'Application' class should be launched. This class runs three threads: the Controller thread, the EndpointGenerator thread and the RouterGenerator thread. The generator classes contain the hardcoded information that generate the sample

topology network and generate a number of threads to run each of the routers and endpoints in the graph.

Note that by default the Controller is configured to be at port 50000, the routers R1-R8 are located at ports 50007-50014, and the endpoints are at ports 50001 and 50002. More endpoints and routers can be added by adding new threads to the generator classes and ensuring the RouterGeneric class is updated with the new endpoint connection.

Each terminal window will provide the port number of the node it is associated with. To send a message from the endpoint at 50001 to the endpoint at 50001, you should open the 'Endpoint @ 50001' window, and follow the prompts, ensuring you pass '50002' as the destination endpoint. You will then see the message show up on the terminal window for the endpoint at 50002.

For clarity, I have shown how my implementation references the sample topology in the graphic below:

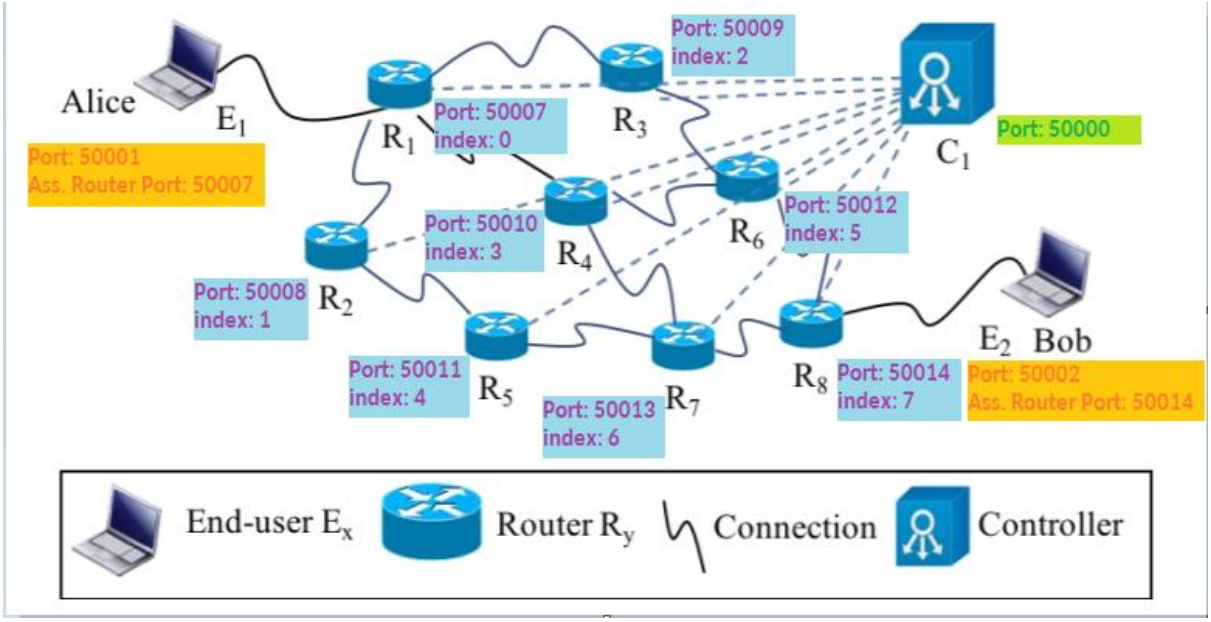


Figure 2.5 1: Figure showing how the implementation references the elements of the sample topology. Note that 'index' stands for the index of the router (row and column) in the adjacency matrix

3. Implementation

In this section, I will discuss in detail the code that facilitated my solution. I will examine each of the Java classes in my project and the purpose they serve in the overall implementation.

3.1 Node

This class came as part of the sample code on Blackboard. Each of the network elements in my implementation extends the functionality of this

class. It has not been modified in any way beyond adding a constant for the minimum router port number I defined for the implementation, which is not necessary for functionality but makes operations for calculating the offset of routers in the Controller's adjacency matrix more clear. The Node class provides each of its subclasses with a thread to listen for packets addressed to its socket.

3.2 PacketContent and PacketHelper

These classes are used to help define and implement the packet encoding I have chosen to use for this project. PacketContent is an interface implemented by PacketHelper and the other classes to access a number of useful constants relating to the packet encoding and the various packet message types.

My implementation uses Java's DatagramPacket class to send my user-defined packets from one node to another through their defined ports, by taking advantage of its compatibility with the Java networking libraries. The DatagramPacket class constructor accepts a byte array ***buf*** and its length, and can be sent from a node's socket by specifying the target port of the packet. My packets are encoded as byte arrays, containing fields shown in the following image:

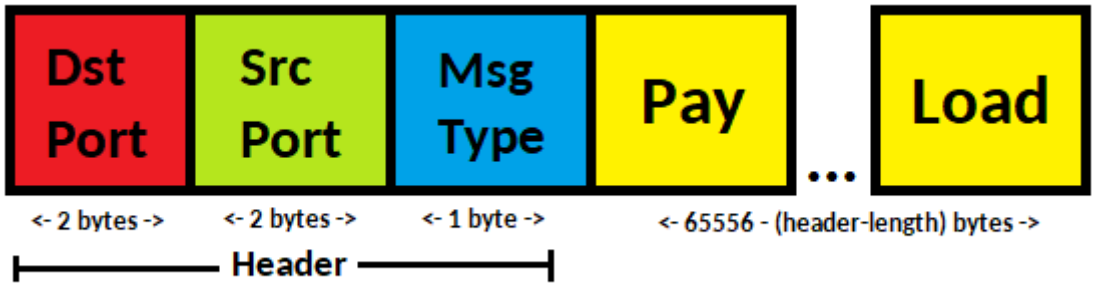


Figure 3.2 1: Diagram showing the byte array encoding the packet information. The header consists of five bytes containing the destination port, the source port and the message type, while the remaining bytes are used to store the payload

The entire frame (header and payload) can then be packaged into the *buf* of a DatagramPacket and sent via methods in the networking library.

The job of extracting the byte array from the DatagramPacket and parsing the individual bytes is offloaded to the PacketHelper class, which provides a number of static methods that can be used by the node classes when dealing with packets. This allows any interaction at the byte level to be hidden from the implementation of these node classes and leads to less complicated code. Additionally, since each node is guaranteed to parse the byte array in the exact same way as they are all using a common set of

methods to create and parse packets, it forces communication to adhere to a common protocol.

3.2.1 createPacket:

The PacketHelper class contains a method for creating a DatagramPacket from a given String payload, source port, destination port, and message type (as defined by the constants in PacketContent).

First, we create a byte array of the appropriate size for the header. We need two bytes to store the port number in binary, since it is in the range 50000. We convert the type-int argument for the source port to its byte-equivalent, taking into account where the most-significant and least-significant bytes should be placed in the array to be parsed correctly, and place it in the correct position in the header array. The same is done for the destination port and the message type arguments.

A second byte array is created to store the payload. The complete packet byte array is created by copying the header and the payload byte arrays into a new byte-array of size 65556; the maximum size of the packet. As discussed previously, the entire packet is placed inside the buffer of a Java DatagramPacket so that it can be sent using the Java networking libraries. We must therefore pass the byte array as the *buf* argument of the DatagramPacket constructor, and return the result.

3.2.2 getPacketSrcPort

This method extracts the two bytes from the given DatagramPacket's *buf* and return the int value they represent. First, the two bytes are extracted into a byte array of size 2, and the int values of the most-significant and least-significant bytes are added to produce the required int value for the packet's source port.

3.2.3 getPacketDstPort

This method is identical to 'getPacketSrcPort' but extracts and returns the destination port of the packet argument.

3.2.4 getPayload

This method extracts the trailing *n* bytes of the packet's byte array that constitute the payload and returns the String corresponding to these bytes.

3.2.5 getMsgType

This method extracts and returns the int-value of the message type of the packet. In the node classes, this can then be tested for equality against the constants defined in the PacketContent class to choose the correct course of action for a particular packet type.

3.2.6 getMsgTypeAsString

This method returns a String detailing the message type of the passed packet argument. I use this method to print relevant packet information to the nodes' terminals.

3.3 Controller

When the Controller class's main method is called, it creates a new thread for the controller. The controller has two synchronized methods: *start* and *onReceipt*.

On start-up, the start method is called which calls the initialiseAdjacencyMatrix method. The adjacency matrix is an 8x8 grid of Boolean values representing whether two routers in the sample topology have a direct connection between one another. initialiseAdjacencyMatrix uses hard-coded values reflecting the state of the sample topology to populate the adjacency matrix.

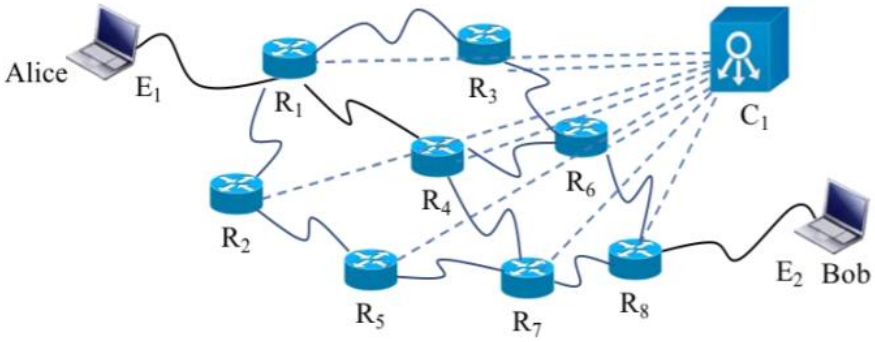


Figure 3.3 1: Sample topology provided for the assignment

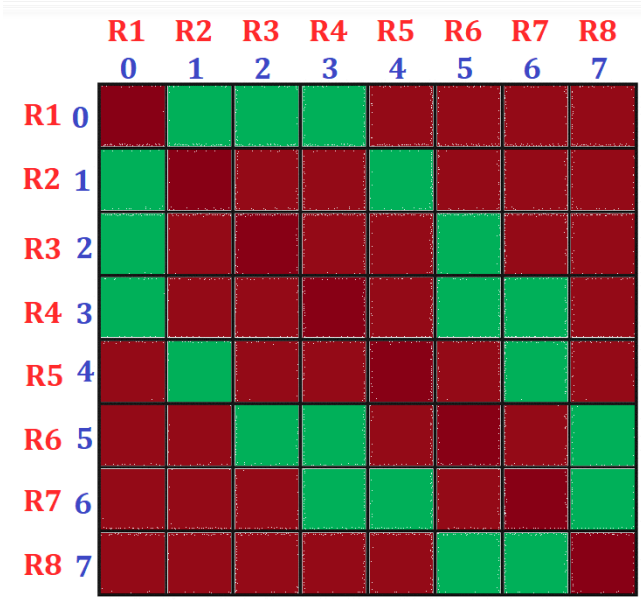


Figure 3.3 2: Corresponding Adjacency Matrix for the sample topology in Figure 3.3 1. Green cells indicate that the routers in row i and column j are directly connected. Red cells indicate that the two routers are not connected. Note that by convention we do not say a router is directly connected to itself.

Since the routers' port numbers are assigned in increasing order, the routers' index in the adjacency matrix maps nicely to their port numbers; (RouterPortNumber) % 50007. I included a constant for this value in Node (MIN_ROUTER_PORTNO) to improve readability. I will discuss how this adjacency matrix allows us to calculate the best route for a packet shortly.

The start thread then waits to allow control to be passed to the onReceipt method. The method only executes when a packet is received. Firstly, the controller prints out the source port and message type of the packet. Then it chooses how to handle the packet with adherence to the OpenFlow protocol by examining the message type.

- If it is a *Hello* message, the controller sends back a *Hello* message to establish contact, followed by a *Feature Request*.
- If the incoming message is a *Feature Response* message from a Router, the controller extracts the information stored as CSVs in the payload of the packet. The first piece of data is the port number the router will be using, and the remaining *n* values are the port numbers of the endpoints directly connected to the router. First the controller stores the port number of this router in the connectedRouterEndpoints ArrayList for reference. The controller needs to store a map of endpoints to routers in order to create a correct path to the endpoint and know at which port number to stop when searching the adjacency matrix. These are stored in the endpointToRouterMap HashMap.
- If it is a *Flow Request* message, the endpoint port of the packet the router is looking for flow information for will be stored in the payload and is extracted by the controller. It then generates a valid path through the network by calling generatePath on the start port and end port of the route, and storing this path as an ArrayList of integer port numbers to be traversed.

  The generatePath method tries first to obtain the port of a router that is directly connected to the endpoint. If this returns *null*, the exception is caught and the controller sends an error message to the router who sent the flow request. If not, the coordinate of the router we have found in the adjacency matrix is computed.

  An adjacency matrix can be navigated by choosing either the row or column matching the start router. Since our graph is undirected, the resulting adjacency matrix is symmetrical about the main diagonal and as such we can choose either the row or column of a router as the start point and obtain the same result. Any green cells in that row or column denote another router that can be directly accessed from (i.e. is adjacent to) the current router. For example, suppose my starting router was R3 and I wished to route a packet to R6. If I choose row 2 of the matrix (which corresponds to R3), I can see that there are two green cells in that row belonging to R1 and R6. This means that R3 is directly connected to

both R1 and R6 (which is reflected in the sample topology diagram) and we can route to R6 successfully in one hop. We can continue the process by changing row to 5 and examining any green cell columns at this row. If at any point the router we seek is in green, it means that we have found a path from our start point to our desired end point, however all possible paths that present themselves may have to be traversed. That is, if we end up in a scenario where more than one column has a green cell, we need to branch and traverse both possibilities.

A breadth-first search is used to traverse all possible paths of a graph in search of a desired node. The 'branching' discussed above takes place, and we move one node further in each branch at a time. This means that all branches at any iteration will have the same length and guarantee that whenever we find a path it will be the shortest possible path to the desired node.

A breadth-first search also needs to mark when it has visited a node before to prevent looping and creating long or unending paths. Since in our adjacency matrix we only move in one direction (take a row as starting router and consider columns, or take a column as starting router and consider rows), we can use a 1-D array of size 8 (the number of routers in the graph) to mark routers we have already visited. This is the Boolean array *visited* is used for.

The concurrent paths of the breadth first search are stored in a LinkedList, which functions as a queue, since I use only the *removeFirst* and *add* methods of the LinkedList. Each concurrent path is stored as an ArrayList of router coordinates in the order they were visited. The object representing this LinkedList of paths is *bfsPaths*.

Our loop deques the first item in the LinkedList and obtains the last router in the path (which is the most recently visited router and where our search should continue from). It marks the router visited, and then checks the rows at this router's column coordinate to see if any are eligible to path to. If the adjacency matrix shows that they are directly linked, we only proceed if the succeeding router has not yet been visited. We update the current path to contain this router and check if it is the endpoint router. If it is, we return the path ArrayList after converting from the adjacency matrix coordinates to the routers' actual port numbers (i.e. adding 50007). This loop will continue while there are still paths to proceed with in the depth-first search. If we are trying to get to a port to which there is no path, we will eventually exhaust all the possible paths in the search and the loop will terminate without returning a successful path. It returns *null* instead to indicate the failure to find a path.

When the method returns, a check is first done to determine if a *null* path was returned. In this case, an error is reported to the router that sent the *Flow Request*. Otherwise, each router along the returned path,

including the one which sent in the original *Flow Request*, is sent a *Flow Mod* message. These are messages containing two CSVs in the payload. These two pieces of information are used to populate an entry in the flow table. The first value is the port number of the endpoint, the second is the router port number that any packet with that endpoint should be routed to; the port number of the next hop. If the router receiving the flow mod is the last hop before the endpoint, the next hop value will be null.

3.4 EndpointGeneric

The EndpointGeneric class creates an instance of an endpoint. The constructor for the endpoint specifies a source port number for the endpoint as well as a port number for the router this endpoint is directly connected to. The EndpointGeneric class also contains two synchronized methods: *start* and *onReceipt*.

The start method is called on instantiation, and consists of an endless loop that accepts input from the user at the terminal. This input consists of two parts: the message to be sent and the port number of the endpoint the message should be sent to. When both pieces of information are entered, they are packetized using PacketHelper to a DatagramPacket and sent to the associated router for that endpoint.

On receipt of a message the control transfers to the *onReceipt* method. There are two types of messages the Endpoint can receive:

- An Error message forwarded on from the controller by the router in the event that the endpoint port passed did not exist in the network or in the case that there was no route to the endpoint.
- A packet routed from another endpoint on the network.

In either case, the appropriate message is printed on the terminal for the user's notice.

3.5 RouterGeneric

The RouterGeneric class creates router instances. On creation, the router creates a HashMap flowTable that maps an integer endpoint port to an integer representing the port number of the next hop. On start-up, the start method is invoked. The router will first wait for one second to ensure the controller has been launched and is listening before sending its *Hello* message and initiating contact. The router will also call initialiseConnectedEndpoints, which is where the hard-coded information about which endpoints connect to which routers is contained. Directly connected routers are stored in the *connectedEndpoints* ArrayList. Any new endpoints added to the application should also be referenced here.

After sending its *Hello* message, the router waits for a message to be received. This waiting period is only interrupted by invoking the onReceipt method.

The first message a router should receive is a *Hello* message from the controller, followed by a *Feature Request*. The router extracts the information from its *connectedEndpoints* list, and sends these port numbers along with its own source port number to the controller.

If the message received is of type *msg* it is an endpoint-to-endpoint packet. The router must first check its flow table for an entry relating to the endpoint listed in the packet. If there is an entry, the router looks at the instructions specified by the flow table. If the next hop is -1, it sends it directly to an endpoint connected to the router, otherwise it sends it on to the port specified. If there is no entry for the endpoint, a *Flow Request* must be sent to the controller. While the router waits for a response to this flow request, it saves the packet information in a global variable and sets the *awaitingFlowMod* flag to true. Control is passed back to the start method, which checks the flag to see if we are awaiting a flow mod. If true, this again launches another period of waiting which is interrupted by onReceipt, and this loop continues until the correct type of packet arrives.

If the type of packet is a flow mod, the information in the packet is extracted and inserted into the flow table for future reference. The awaitingFlowMod flag is set to false. There is necessarily a saved packet ready to be sent, and control is returned to the start method.

The awaitingFlowMod flag is false so the waiting loop can be escaped. The saved packet is then forwarded using the new information in the routing table. The start loop enters again and we wait for the receipt of new packets.

In the case that the router receives an Error message from the controller instead of a flow mod, a message is sent to the endpoint that initially sent the packet indicating a delivery failure. We are no longer expecting a flow mod so the flag is set to null and the saved packet is discarded.

3.6 Application, RouterGenerator & EndpointGenerator

These classes and their purposes were discussed in section Overall Design: Using the Application.

4. Discussion

In this section, I will show some WireShark captures of packets sent by my program while it is executing, and conduct an analysis of the advantages and disadvantages of my implementation.

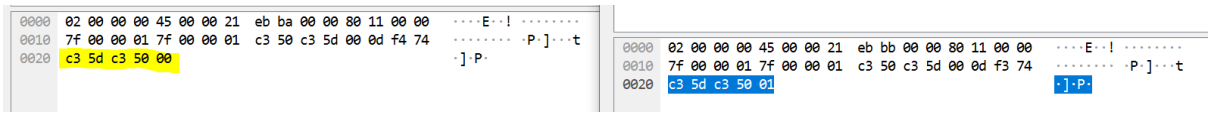
4.1 Packet Captures

I used WireShark to capture packets passing through ports 50000-50014. On start-up of the program, a slew of packets are sent as the automatic sequence of *Hello->Hello->Feature Request->Feature Response* begins between each router and the controller.

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	UDP	37	50013 → 50000 Len=5
2	0.038761	127.0.0.1	127.0.0.1	UDP	37	50000 → 50013 Len=5
3	0.038950	127.0.0.1	127.0.0.1	UDP	37	50000 → 50013 Len=5

Here we see the router at 50013 initiate contact with the controller at 50000 by sending a hello message. This message is followed by a hello response and a feature request, both from controller to router. Each of these messages have a length of 5 bytes, since they contain no payload information due to their message types; they contain just the 5-byte header.



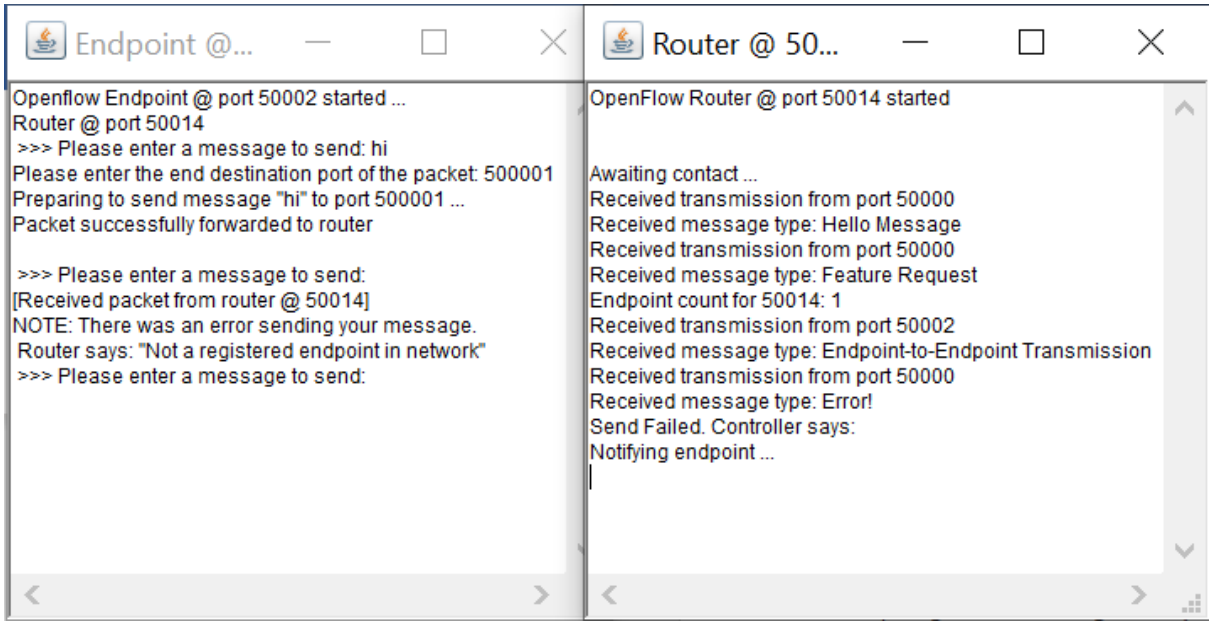
The above image the last 5 bytes of the *Hello* and *Feature Request* packets from the controller to the router, which is where my packet encoding lies within the DatagramPacket. As expected, these headers are identical, except for the final byte which encodes the message type, which differ by 1. This correlates exactly with the constant integer values for the message types denoting a *Hello* message and a *Feature Request* message; they are 0 and 1 respectively.

1	0.000000	127.0.0.1	127.0.0.1	UDP	37	50013 → 50000 Len=5
2	0.038761	127.0.0.1	127.0.0.1	UDP	37	50000 → 50013 Len=5
3	0.038950	127.0.0.1	127.0.0.1	UDP	37	50000 → 50013 Len=5
4	0.043542	127.0.0.1	127.0.0.1	UDP	37	50012 → 50000 Len=5
5	0.054193	127.0.0.1	127.0.0.1	UDP	37	50000 → 50012 Len=5
6	0.054262	127.0.0.1	127.0.0.1	UDP	37	50000 → 50012 Len=5
7	0.056147	127.0.0.1	127.0.0.1	UDP	37	50010 → 50000 Len=5
8	0.062452	127.0.0.1	127.0.0.1	UDP	42	50013 → 50000 Len=10

We can see in the above image that it takes some time before the *Feature Response* is returned to the controller, presumably due to the time taken by the router to prepare the information and place it in a datagram packet. The message length is longer this time since it contains a payload with information about the router (5 bytes for the five characters of the port number).

22	0.142958	127.0.0.1	127.0.0.1	UDP	48	50014 → 50000 Len=16
23	0.157634	127.0.0.1	127.0.0.1	UDP	48	50007 → 50000 Len=16

The two feature responses from the two routers that have direct endpoint connections shown above are longer. This conforms to their payload, which consists of two 5-digit port numbers and a comma character separating them.

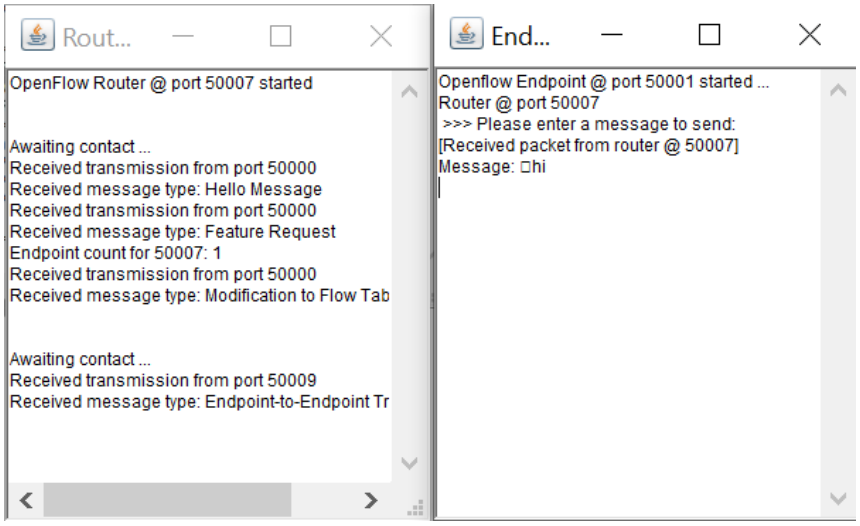


Here we see the activity of the program when I accidentally enter the wrong port number into the terminal. The packets sent are shown below:

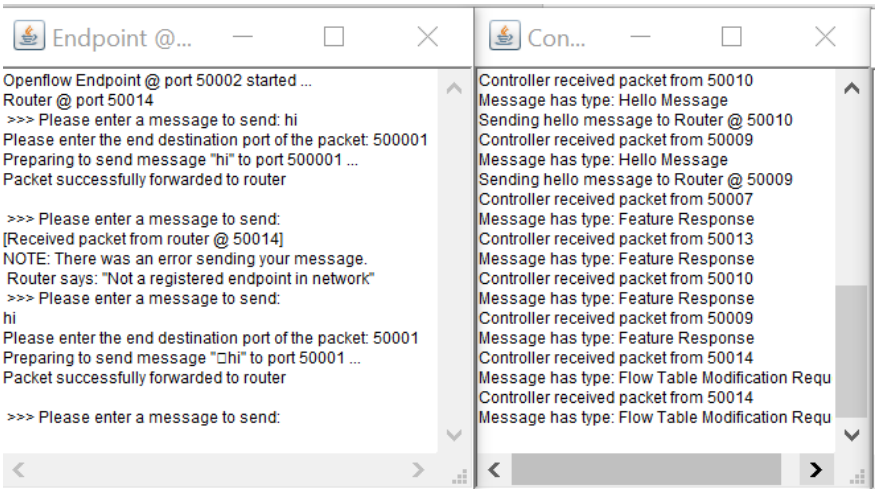
```
 70 1617.726403   127.0.0.1          127.0.0.1          UDP       39 50002 → 50014 Len=7
 71 1617.741611   127.0.0.1          127.0.0.1          UDP       42 50014 → 50000 Len=10
 72 1617.749273   127.0.0.1          127.0.0.1          UDP       73 50000 → 50014 Len=41
 73 1617.759931   127.0.0.1          127.0.0.1          UDP       73 50014 → 50002 Len=41
```

1. Endpoint sends message to router @ 50014
2. Router sends flow request to controller
3. Controller sends error message with error description payload to router
4. Router forwards this message to the endpoint

A message successfully sent from 50002 to 50001 returns the following result:



74	1807.673172	127.0.0.1	127.0.0.1	UDP	42	50002 → 50014 Len=10
75	1807.686889	127.0.0.1	127.0.0.1	UDP	42	50014 → 50000 Len=10
76	1807.695125	127.0.0.1	127.0.0.1	UDP	48	50000 → 50014 Len=16
77	1807.695301	127.0.0.1	127.0.0.1	UDP	48	50000 → 50012 Len=16
78	1807.695467	127.0.0.1	127.0.0.1	UDP	48	50000 → 50009 Len=16
79	1807.695648	127.0.0.1	127.0.0.1	UDP	47	50000 → 50007 Len=15
80	1807.722440	127.0.0.1	127.0.0.1	UDP	42	50014 → 50012 Len=10
81	1807.739838	127.0.0.1	127.0.0.1	UDP	42	50012 → 50009 Len=10
82	1807.746944	127.0.0.1	127.0.0.1	UDP	42	50009 → 50007 Len=10
83	1807.751400	127.0.0.1	127.0.0.1	UDP	42	50007 → 50001 Len=10



1. Endpoint @ 50002 sends packet to router
2. Router sends a flow request to the controller
3. Controller sends flow mods to all routers on the path to the endpoint

4. We can see the packet passed between routers sequentially until we hit the router at 50007 that the destination endpoint is directly connected to
5. The router at 50007 delivers the packet to the endpoint

Sending a packet along the same route again does the same steps but without the flow request or flow mods:

```
 84 2496.146744   127.0.0.1       127.0.0.1       UDP      42 50002 → 50014 Len=10
 85 2496.168561   127.0.0.1       127.0.0.1       UDP      42 50014 → 50012 Len=10
 86 2496.174909   127.0.0.1       127.0.0.1       UDP      42 50012 → 50009 Len=10
 87 2496.180504   127.0.0.1       127.0.0.1       UDP      42 50009 → 50007 Len=10
 88 2496.184667   127.0.0.1       127.0.0.1       UDP      42 50007 → 50001 Len=10
```

4.2 Advantages and Disadvantages of Implementation

A few disadvantages are apparent in my implementation. To ensure adherence to the OpenFlow module, my approach consists of a controller that already knows its network topology in advance. It presides over a number of routers who do not know of each other's existence outside of the controller's routing instructions. Because of this, my approach is fitted to the example topology for the assignment, and the addition of new network routers would require re-working how the adjacency matrix is stored and created. Since the adjacency matrix representation needs to be entirely refreshed each time a new router is added, and the routing tables of each of the routers may become obsolete if distance vector routing needs to be strictly enforced. Tying the adjacency matrix to a more dynamic data structure instead of an 8x8 2D array would be helpful in the long run, however when dealing with a small fixed topology I believe it is the simplest and cleanest implementation.

Another weakness is that the end users need to know the port addresses of other endpoints in advance. If a user enters a port number which is not an endpoint in the network an error is issued simply notifying the user of a failure to send, rather than a helpful way to proceed. It would be advantageous in an extended implementation to give each endpoint a memorable String id and include some way of querying their associated routers for a directory of other endpoints they can send to instead of memorizing individual port numbers. This would be implemented by keeping another HashMap in the controller which maps endpoint ports in the network (which it already has a list of) to their unique String identifiers and delivering this information to the routers in the network to when required copies for them to keep.

Other than that, I think my implementation has many strengths. The packet encoding is simple and straight-forward, and the PacketHelper class makes working with the individual bytes of a packet simple. New OpenFlow message types can be easily added by simply defining a new constant in the PacketHelper class and providing the desired implementation in the nodes. My code also handles cases where the user enters invalid data and notifies them. The main features of the OpenFlow protocol are implemented throughout and the different message types are handled easily and clearly within a switch on the message type in each type of network component. My code is easily expandable through the generator classes and can handle new endpoints being added. While adding new routers implies changes to the adjacency matrix and will require a few more tweaks, the necessary changes to the code needed to expand the implementation are confined to the methods that deal with the matrix and make sense contextually. I feel like my understanding of the difference between the start and onReceipt methods is better now than it was when completing assignment 1, as the code is much more concise, and the implied function of each method is better reflected in my code.

5. Reflection

I was happy to discover that the work I put in to the first assignment in this module was highly transferrable to this assignment, and I did not need to tweak much when devising how packet structures should be represented in the code. Since I already had the PacketHelper and PacketContent classes made for Assignment 1, it made devising and implementing the different OpenFlow packet types, and the overall protocol, much quicker. In terms of the theory for this topic, I found understanding the necessity of OpenFlow tricky at first as I don't have much practical knowledge of how networks operate on a large scale and their specific challenges, however reading up on SDN in preparation for this project gave me a better overview of networking as a whole. The main technical challenge of this assignment was implementing Distance Vector Routing through a breadth-first search of an adjacency matrix representing the topology of the network, as I had never worked with graphs in programming before. Overall I am happy with the work that I have done for this project and spent roughly 20 hours coding.