Telecommunications II
Assignment 2 – OpenFlow Routing

Stephen Rowe – ID 14319662

Date: 28th November 2019

Assignment Approach

Our assignment required us to implement OpenFlow with a system of Hosts, Routers and a Controller, and create our own version of the Protocol in order to outline how the different nodes interact with each other. I chose to develop my system with Link State Routing, using Dijkstra's Shortest Path First Algorithm, as opposed to a preconfigured network. Similar to Assignment 1, I also chose to develop this network entirely within Eclipse, and was able to reuse much of the code for communicating between nodes from the previous assignment, meaning that this Network utilizes a Go-Back-N Flow Control.

OpenFlow Description

OpenFlow is a protocol for communication that requires (i) a number of Routers which each consist of a Flow Table of destinations and how to access them (i.e. through other Routers), (ii) a Controller that provides the shortest path for the Router to reach its destination (that is, by sending it an updated Flow Table), and (iii) Hosts that will send and receive packets from other Hosts.

When a Router does not recognize the name of the intended destination of a packet it received, it will request a new Flow Table from the Controller. The Controller has developed an overall view of the connected and active nodes on the network, and is able to tell the Router how to proceed. After the Router receives their new Flow Table, the Router knows how to access this destination from this point forward, and as a result any subsequent packets to this destination do not require the Router the request another Flow Table.

The only real function that a Router has is to forward packets, but it must extract the header from each packet it receives in order to understand what it should do with it.

In my implementation, I was able to reduce the Flow Tables down to two headings, (a) Hosts available on network, (b) the next hop in the network in order to reach this Host. This greatly simplified the process of communicating a new Flow Table (from Controller to Router).

In my implementation of OpenFlow, I reused the SNDContent class from Assignment 1, in order to categorize the different types of packets, but modified this class to reflect the new types of packets that this assignment would require:

Hello ("HELLO") – This packet will be sent from the Router to the Controller to let them know of their presence. The Controller will ACK this, and then send another Hello packet back. The Controller uses this to create a NodeData for this Router (i.e. start the Go-Back-N window for communication between the two).

While not in the original assignment description, I decided to implement a Hello process between the Router and the Host, too. When the required setup between a Router and a Controller is completed (i.e. with Feature Request and Reply), then the Router will send a Hello to the Host to tell it that it may start creating and sending strings across the network, i.e. tell the host that it is fully

connected to the network and is online. The Host will send back a Hello, too. I also decided that, if the Host has not sent a Hello back (i.e. the program for that host is not running, it is offline), but the Router receives a packet for that Host, then the Router will discard this packet entirely, so as not to create a massive ArrayList of messages to send to the Host while it is waiting for it to come online, and also not to overburden the Host when it does eventually connect.

Feature Request ("FETRQ") – After receiving a Hello from the Router, the Controller will send a Feature Request to the Router. This is essentially asking the host to send the Controller a full description of all of its immediate connections.

Feature Reply ("FETRP") – This is sent from a Router once it receives a Feature Request. The content of the Feature Request packet comes in the following form:

> [Router Name] [Host Connected Name] [Router Cnctd 1 Name] [Router Cnctd 1 Distance] [Router Cnctd 2 Name] [Router Cnctd 2 Distance]…

The Controller does not need to know the sockets of all the connections that the Router has, only the names and the distances. The Controller uses the class "ControllerFlowTable". In this class, each Router is designated its own class element of "DirectConnectionsPerRouter", which stores the above information for easy access.

Packet In ("PACIN") – All nodes on the network use this packet type in their own way. When a Host sends a packet, it will send it with this header, telling the router that it needs to be forwarded. When a Router receives a PACIN, it will open the packet and read its header, and determine if it can forward it on to another router or not, judging by its Flow Table. If it can, it will do so. If not, it sends a PACIN packet to the Controller. The Controller interprets this as a request for a new Flow Table, which it will generate specifically for the requesting router, and send it back as a FLWMD.

Flow Modification ("FLWMD") – Following the generation of a new Flow Table for a specific router, the Controller will send a FLWMD packet to the Router, consisting of the results. This packet will list all the Hosts currently connected on the network, and which Router that the Requesting Router should forward a packet on to in order to eventually reach this destination. The content will have the following form:

> [Host 1] [Access Router For Host 1] [Host 2] [Access Router for Host 2]…

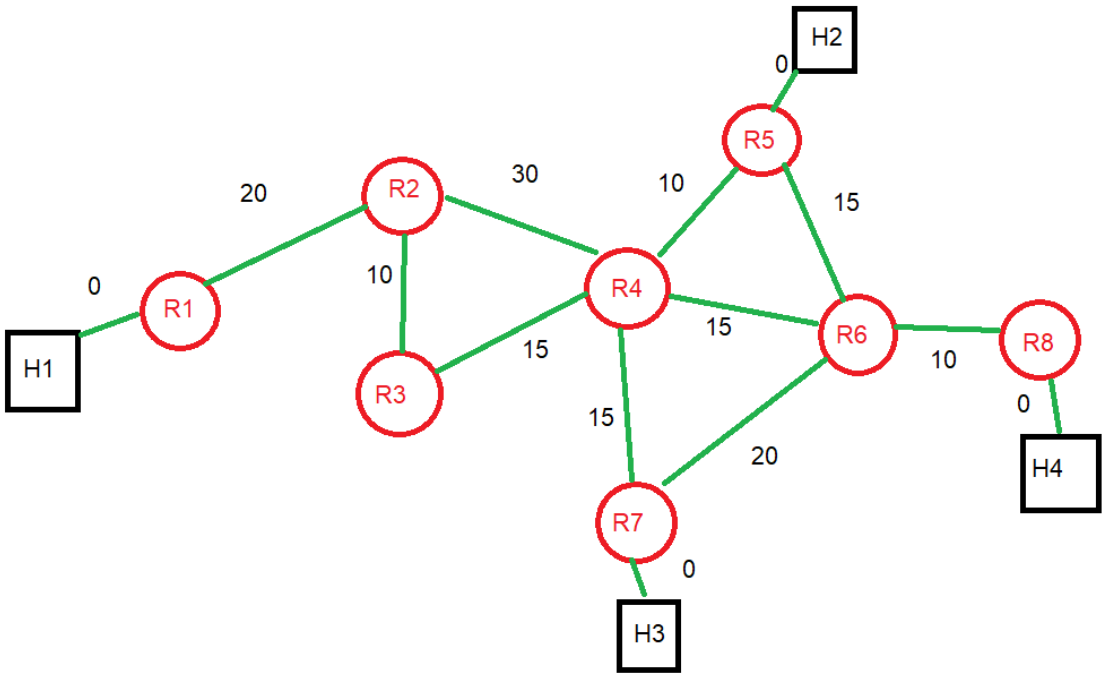
Overall Design

Due to the fact that I used Dijkstra's Algorithm, the Controller in my system does not retain a preconfigured flow Table that defines the entire network, but instead requests any Router that has contacted the Controller to provide information on the Routers and Hosts directly connected to it, as well as the distances. This means that a new Flow Table will be created for a given Router every time one is requested, but this Table is only based on the Routers that are present and active on the network. This allowed for some flexibility in demonstrating the functions of the network, and allowed me to just choose a small section of the overall network graph that I had initially designed to see that the communication was being carried out properly.

Each Router knows about the Host it is connected to (if any), and Routers it is directly connected to, but nothing else. This is achieved by passing some specific information as arguments when running the program in Eclipse (which I will elaborate on later).

I designed the following network topography to base my demonstration and the arguments of the Hosts and Routers on. It includes distances between routers (distances between Hosts and Routers

are always 0, since distance doesn't really matter between a Host and its Router, and doesn't affect the algorithm). I should also point out that, in my program, the distance between Routers don't actually mean that the packet will take longer to arrive, because I didn't create a way to impede packets proportional to their distance. Distances are only used by the algorithm and in telling the router how to forward the packet, but will actually arrive at the same speed regardless.



In order to implement the above network, I had to design my Router and Host programs to accept parameters that would describe the nodes that it is directly connected to. There were 8 Debug Configurations created in Eclipse for Routers, and 4 for Hosts.

For a Host, these parameters come in the form

> [Host Name] [Number of other Hosts on Network] [Host Port] [Router Port]

For a Router, these parameters come in the form

> [Router Name] [Router Port] [Host Name] [Router Cnctd Name 1] [Router Cnctd Port 1] [Router Cnctd Distance2] [Router Cnctd Name 2] [Router Cnctd Port 2] [Router Cnctd Distance 2]…

There is no limit to the number of other Routers that a Router can be connected to, but I limited the number of Hosts it can connect to by 1. This was simply to make the parameters and resulting code to accept these parameters more readable, but it would be possible to implement more hosts by adjusting the code a bit.

As an example, here is the parameter I used for H1:

```
H1 4 50031 50021
```

The "4" in each parameter tells the Host how many hosts are on the network in total, even before connecting to the network. The reason I did this was because the Host may want to send a packet to a Host that the Router does not know about, kind of like how we may want to visit a website that

our real router will have to make a DNS Lookup for. The Host will create an array of host names (in this case it will be “H1”, “H2”, “H3”, and “H4”), and randomly select one of these Hosts as its destination. An unintended consequence of this was that a Host can actually send itself a packet, but this actually ended up working fine, so I didn’t change anything.

To make it a little easier to read, 5002X socket numbers are Routers, and 5003X socket numbers are Hosts.

To demonstrate a Router’s parameters, I will just take the first two:

```
R1 50021 H1 50031 R2 50022 20

R2 50022 00 0 R1 50021 20 R3 50023 10 R4 50024 30
```

So, Router 1’s parameter will make reference to Router 2, and Router 2’s parameter will make reference to Router 1, as they are connected to each other, and the distances are the same for both. You can also see that in R2, the value “00” is passed for connected hosts name, and the hosts socket is set to “0”. This just means that there is no Host connected, and our code will recognize this.

If you want to recreate this network in Eclipse, here are the full parameters:

HOSTS:

```
H1 4 50031 50021

H2 4 50032 50025

H3 4 50033 50027

H4 4 50034 50028
```

ROUTERS:

```
R1 50021 H1 50031 R2 50022 20

R2 50022 00 0 R1 50021 20 R3 50023 10 R4 50024 30

R3 50023 00 0 R2 50022 10 R4 50024 15

R4 50024 00 0 R2 50022 30 R3 50023 15 R5 50025 10 R6 50026 15 R7
50027 15

R5 50025 H2 50032 R4 50024 10 R6 50026 15

R6 50026 00 0 R4 50024 15 R5 50025 15 R7 50027 20 R8 50028 10

R7 50027 H3 50033 R4 50024 15 R6 50026 20

R8 50028 H4 50034 R6 50026 10
```

In the project instructions, Routers were described as having 2 or more sockets. The node class that I am using (from Assignment 1) is built around the assumption that there is one socket, but it would be very time-consuming to make the changes to that class to implement more than one socket variable for this given router, which would ultimately have the same result in my program as just using a single socket that deals with all the communication. I understand the theory behind having multiple sockets, but the use of my NodeData class essentially acts as multiple sockets, as it identifies which port it must send the packet to and then does so, and also considers both sides of the router.

I was able to develop the program in such a way that it does not matter what nodes start up first, you can start up the hosts, controller and routers in any order you wish.

Packets

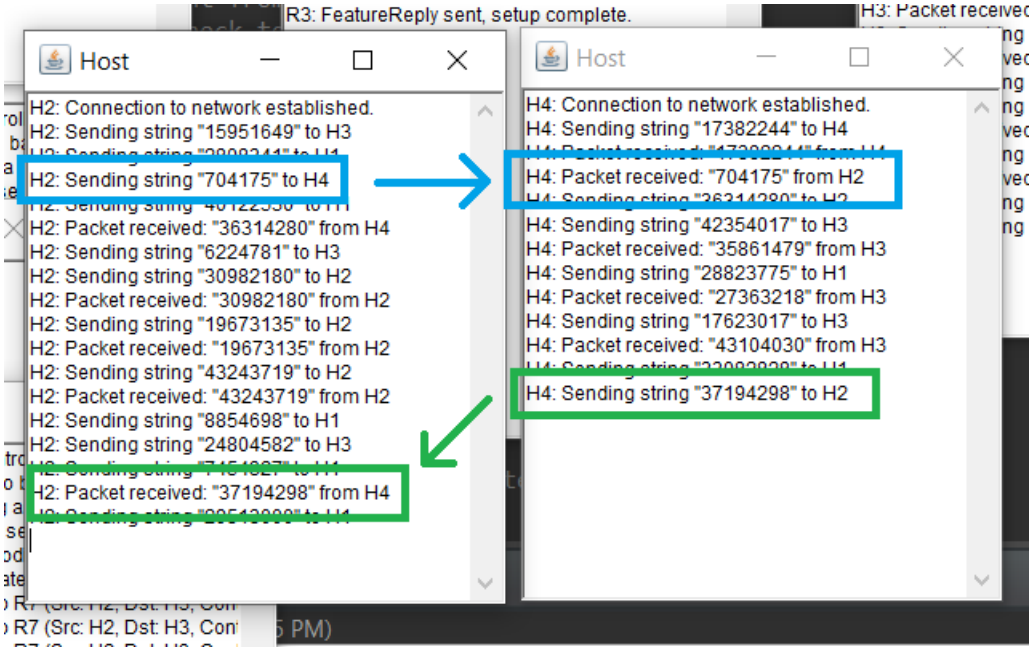
The SNDContent class has been modified from Assignment 1 to reflect the types of packets required in the OpenFlow section, described above. I replaced the "Originating C&C" header with two new ones: Source Host and Destination Host. These will be used by all three node types (Controller, Router, Host).

Host

Host was the easiest Node type to develop, as it has two simple purposes:

1. Create a randomly generated string, select a Host on the network randomly, create a packet and send this to the router
2. Receive packets from the Network, and display its content on screen and where the packet came from.

Here are two Hosts in action, H2 and H4 (all nodes on the network are connected and active here):



The Host has a timer that will activate between every 5 and 10 seconds:

```
doJobTimer.schedule(doJobClass, 0, new Random().nextInt(5000) + 5000);
```

This timer is very similar to the doJobTimer from the Worker class in Assignment 1, in that it calls a method within the Host class at regular intervals. It will generate a random string of numbers and forward it to the Router.

I also didn't want the Host to create and send a string every single time the Timer went off, so I created a small condition in the sendAPacketRandomly method:

```
if(new Random().nextInt(5)<2)
```

When the Host receives a packet, there are two options:

- It is a "Hello" message from the router, meaning that connection to the network is fully established and the Boolean connectionToNetworkEstablished is set to true, meaning that the host can start creating and sending packets
- It is a "PACIN" message, meaning that the Host will print the content of the packet (i.e. the string randomly generated by another host) and print that to screen, along with the name of the Host that sent it.

There is not much else to say about the Host, as the rest of the methods in this Class are purely for sending packets and receiving ACKs, similar to the programs in Assignment 1.

Router

The Router implements the NodeData class for other Routers it is connected to, so that it can communicated to each individually through Go-Back-N. These are stored in an ArrayList. NodeData variables are also created for the Host (connectedHost) and Controller (connectedController) for the same purpose, but there is no need to create an ArrayList for these, are there is only one of each.

The Constructor for the Router accepts the String array of arguments (that we entered in Eclipse) as parameters. The first element of this Array is the router's name. If the second element is anything other than "00", then the connectedHost variable is assigned a NodeData with the second and third element of this array as parameters. If is it "00", then there is no need to create a NodeData, and this variable will be left as null, which will be useful in if-statements in the onReceipt method (if it is null, then we immediately know that a "PACIN" packet is from another router).

We then iterate through the rest of the array, creating NodeData's for each connected router and adding them to our ArrayList.

Concurrent to all of this, we are updating the flowRequestInformationToSendToController string. This is the String that we will send to the Controller when we send our FeatureReply. It consists of the name of our router, the name of the host, the name of the router connected and the distance to this router (last two points repeated if there is more than one router connected.

The Router uses the RouterFlowTable class. If we assume that all nodes are connected and active in the above network, then the FlowTable for R7 will look like this:

HOST NAME	ACCESS ROUTER
H1	R4
H4	R6
H3	R7
H2	R4

Note that for H3 the value is R7. This doesn't mean that the packet will go on an endless loop, as R7 will check if the packet is destined for its host (H3) every time it receives a packet, and will forward it to H3 every time.

The names of all known hosts are stored in the allHosts String ArrayList. This works in conjunction with the isDestinationHostKnown method, which is a simple ArrayList.contains() call. If the destination is contained in the allHosts ArrayList, then we know that we have a HostAndAccess element for this Host. The HostAndAccess class tells us which Router name we should forward to packet on to (by matching it up with the NodeData ArrayList in the Router class). The getAccessRouter simply retrieves the name of the relevant router from our ArrayList.

The onReceipt method in router determines if

- It is an ACK (in which case we stop sending that packet)
- If it is from the Controller (so call the processControllerInstruction method)
- If it is from a Router (so call the processRouterInstruction method)
- If it is from our Host (so call the processHostInstruction method)
- If the packet was already received and processed (so we send an ACK with the next expected packet number from that Node)

processControllerInstruction has three possible scenarios:

- Controller said Hello – we write a notification of this to terminal
- Controller sent a Feature Request – we send back a Feature Reply, consisting of the flowRequestInformationToSendToController string we created earlier.
- Controller sent a Flow Modification – we clear our current Flow Table and create a new one from the string that the Controller sent us.

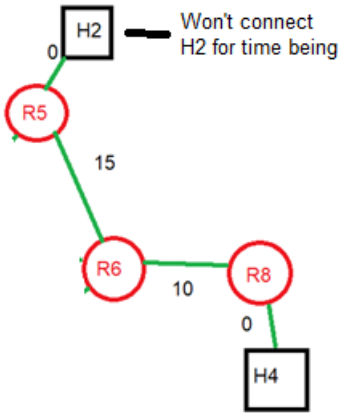
processRouterInstruction has two possible scenarios:

- Router has sent us a PACIN, and we recognize the destination host name. We check our Flow Table, identify the router we should forward the packet to, and do so.
- Router has sent us a PACIN, and we do not recognize the destination host name. We send the Controller a PACIN (i.e. request for a new Flow Table), and store the packet received in the waitingToSend ArrayList, as we have no where to send it immediately.

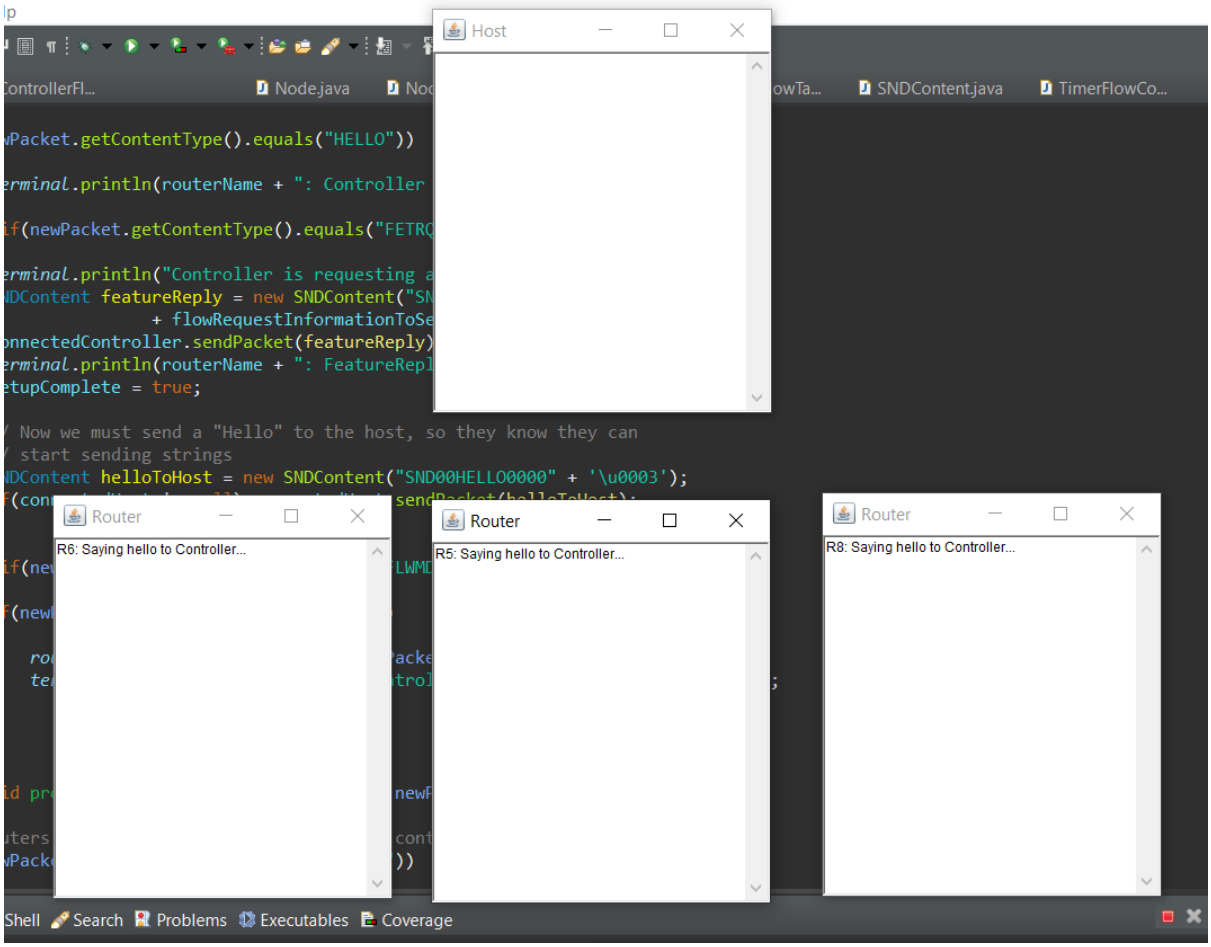
processHostInstruction has the same scenarios as processRouterInstruction, the only difference is what we actually print to the terminal.

The waitingToSend ArrayList is a list of all the packets that we have received, but don't know where to send yet. We must wait for a FLWMD from the Controller. The Router's start() method deals with this. Assuming that a FLWMD has been received in the time between the packet being stored in the waitingToSend ArrayList, and this current call of the start() method, then we iterate through waitingToSend, call the isDestinationHostKnown method from RouterFlowTable class, and if is it present, then we can send the packet and mark this packet for removal from the waitingToSend ArrayList.

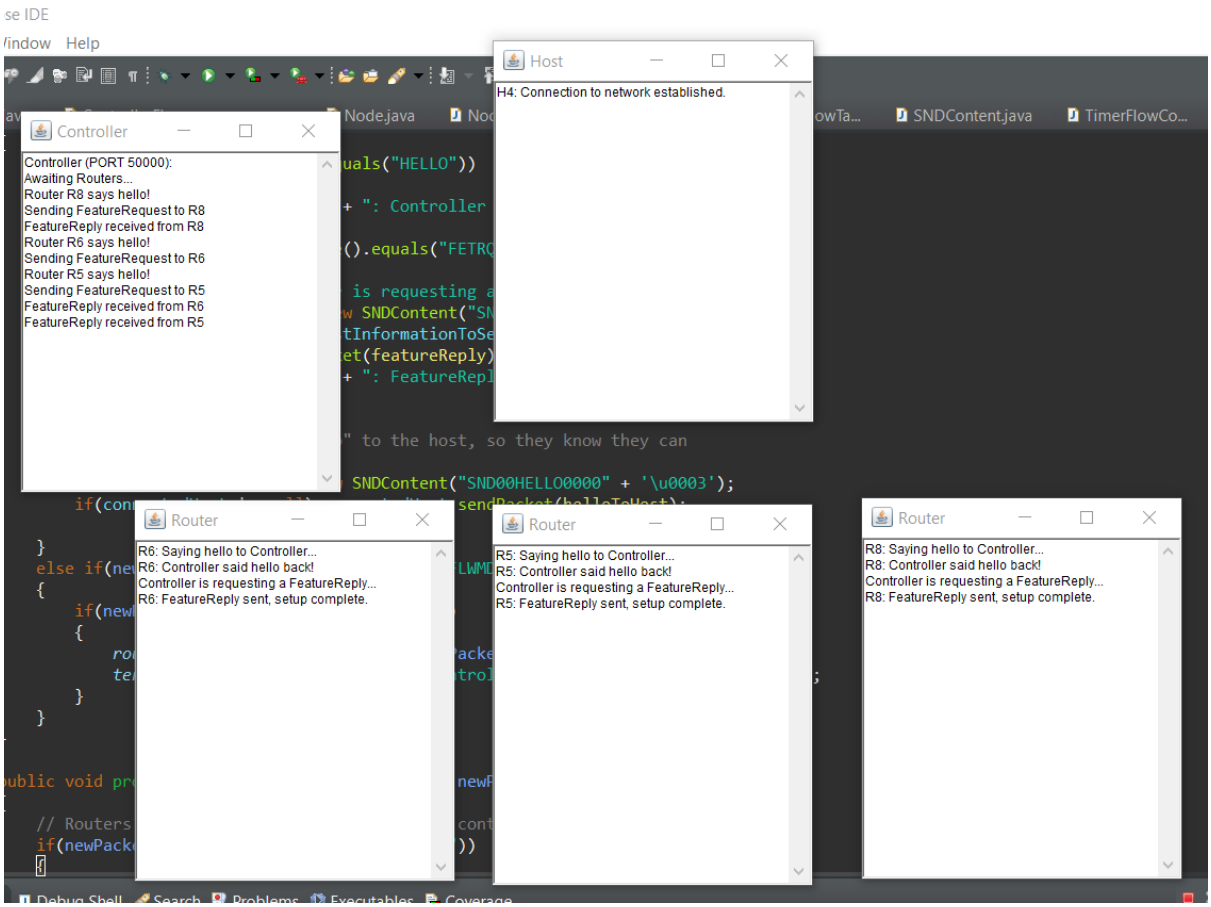
We will now demonstrate some features of the Router. For this demonstration, I will only use a section of the full network:



First, we'll show that the host will not generate any strings or send any packets until it gets the confirmation of access to the network from the Router. To do this, I started up the routers and host, but did not start the Controller (The windows will move around a bit in these screenshots, it took me a few tries to capture exactly what I wanted, sorry!):



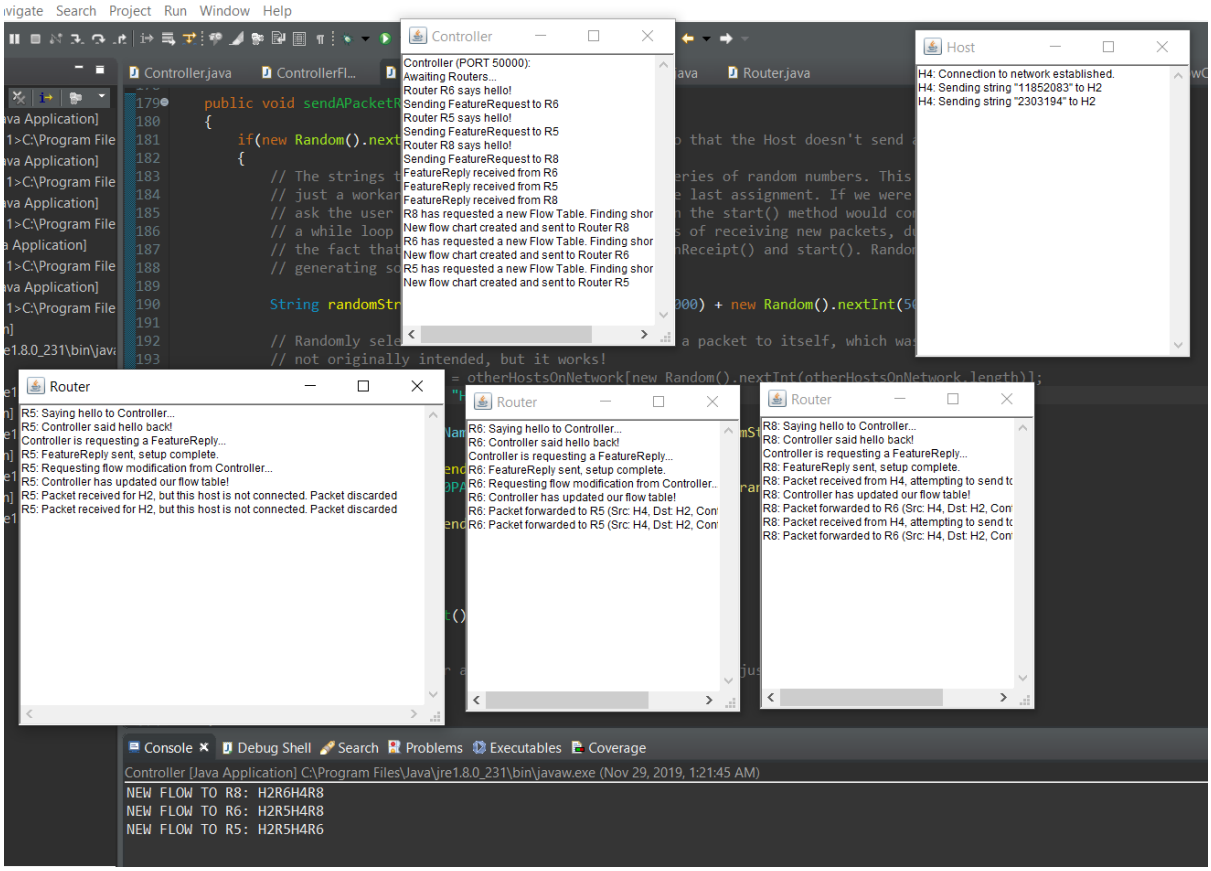
Now I will start up the Controller:



We can see a few things have happened:

- Each of the Routers have said hello to the Controller
- The controller has sent a Feature Request to each Router
- Each Router has sent a Feature Reply to the Controller
- R8 has sent a Hello to H4, telling it that connection has been established.

Now the Host has sent a string:

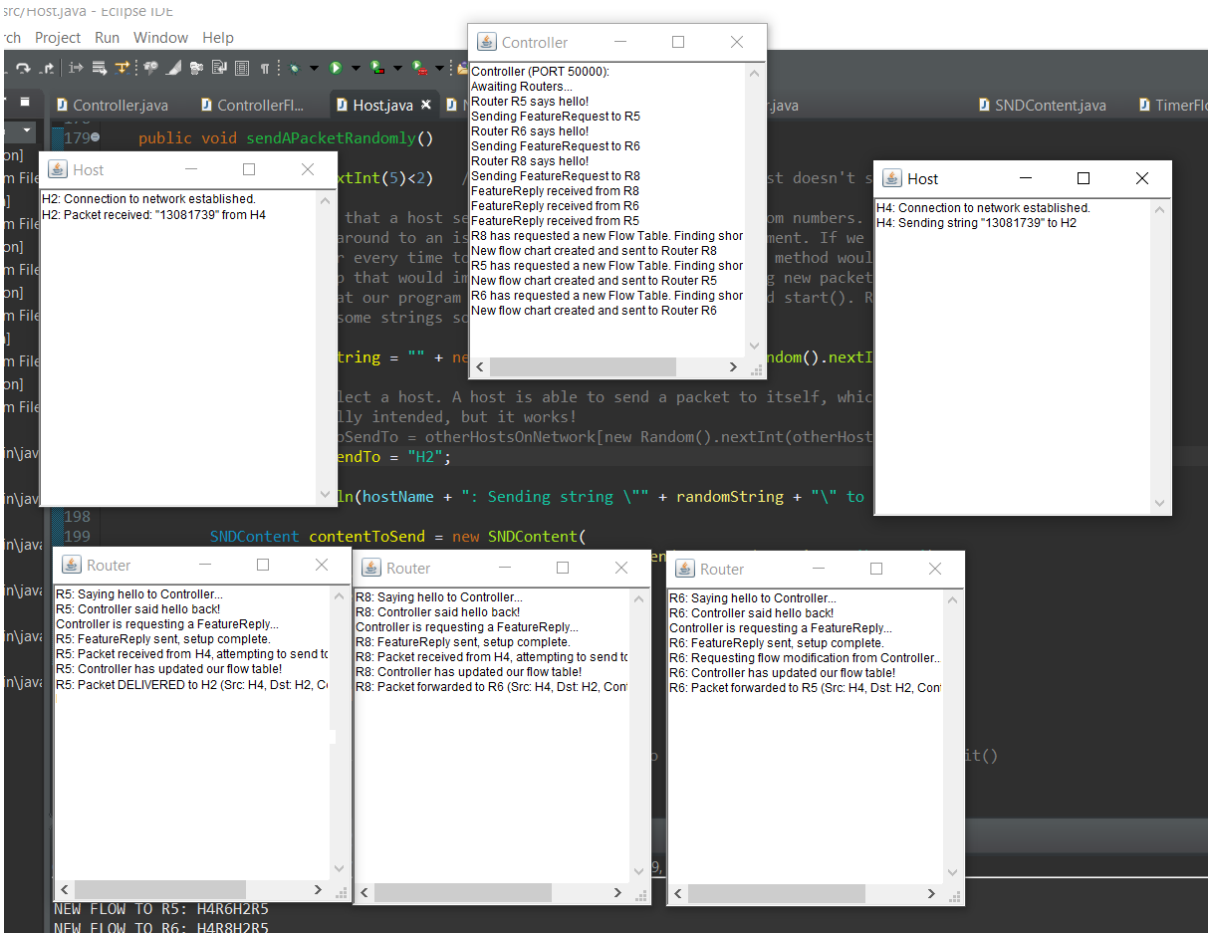


We can see that:

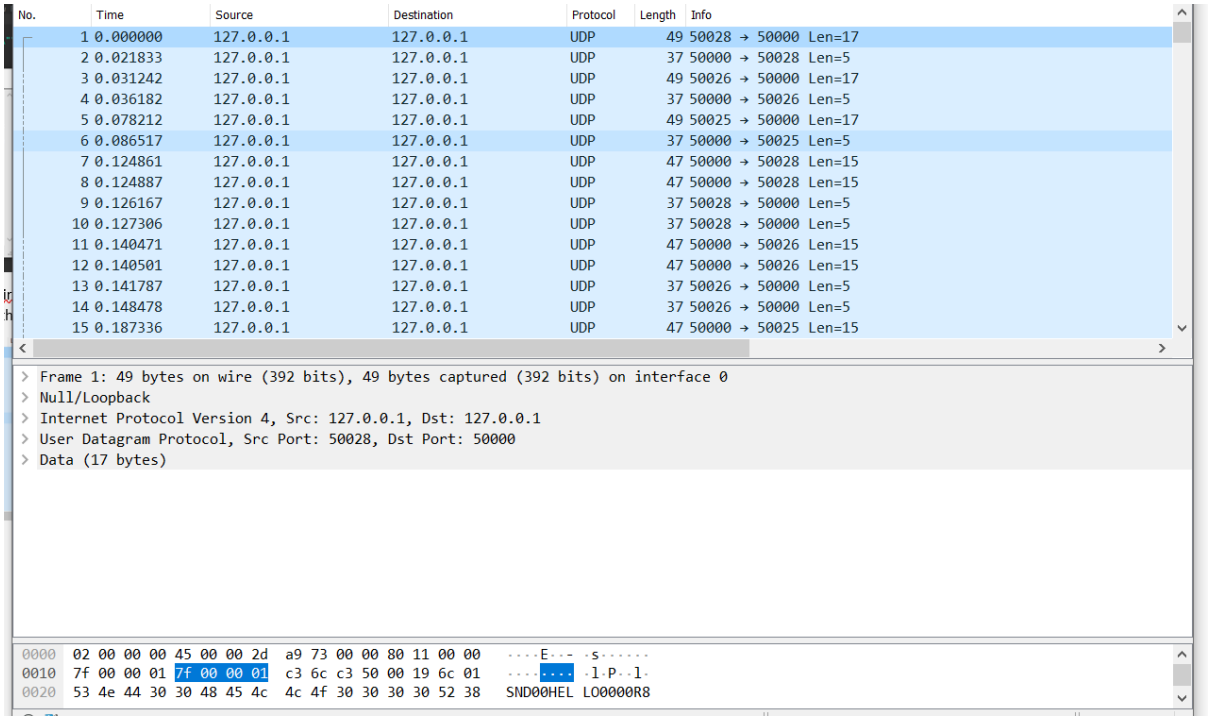
- R8 has received the packet, but doesn't know where to send it
- It stores this packet in its WaitingToSend ArrayList, and requests a Flow Modification from the Controller.
- Since R5 told the Controller that is has H2, then the Controller sends a FlowMod to R8 (you can see the FlowMod string in the Eclipse terminal at the bottom of the screenshot).
- R8 now forwards the packet to R6
- R6 requests FlowMod, gets one, forwards to R5
- R5 has received the packet and knows that its host is H2. But, since we haven't started up H2 (i.e. no hello has been received from this host), then R5 knows the Host is offline, so it discards the packet).
- All communication has been fully ACKed between Routers, so there is no issue that the Go-Back-N Flow Control is clogged up with packages that cannot get to their final destination (H2).

If you look at the terminal in Eclipse here, you can see that there are only two known hosts on the network, so the Flow Table that the Controller can create is relatively short compared to the Flow Table that will be generated on the full network. This will be elaborated on further in the Controller section.

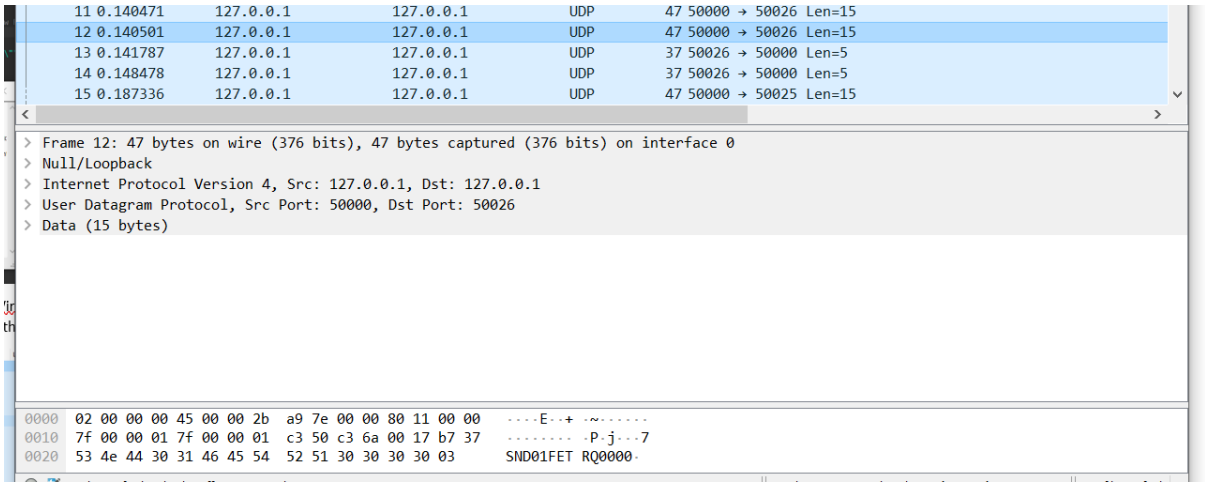
Finally, we will start up H2, and show the packet being delivered successfully:



I'll take this opportunity to capture the packets in WireShark, as there are very few packets being delivered here compared to when we're operating the full network:



We can see in the Info section of the above graph that 50028 (i.e. R8) sent a Hello packet to the Controller (I've included the contents of this packet at the bottom of the screenshot). Then 50000 (i.e. the Controller) sends an ACK back. 50026 also sends a Hello, and this is ACKed. After the Hellos are done, the Controller starts to send a Feature Request to each Router:



Controller

The Controller was the most difficult class to develop. It has its own sister class, ControllerFlowTable.

The construction of the controller is very simple, it only needs to set its own socket and declare a controllerFlowTable. The onReceipt method is also straightforward, as the only type of node that can deliver to the Controller is a Router.

When it receives a packet there are three possibilities:

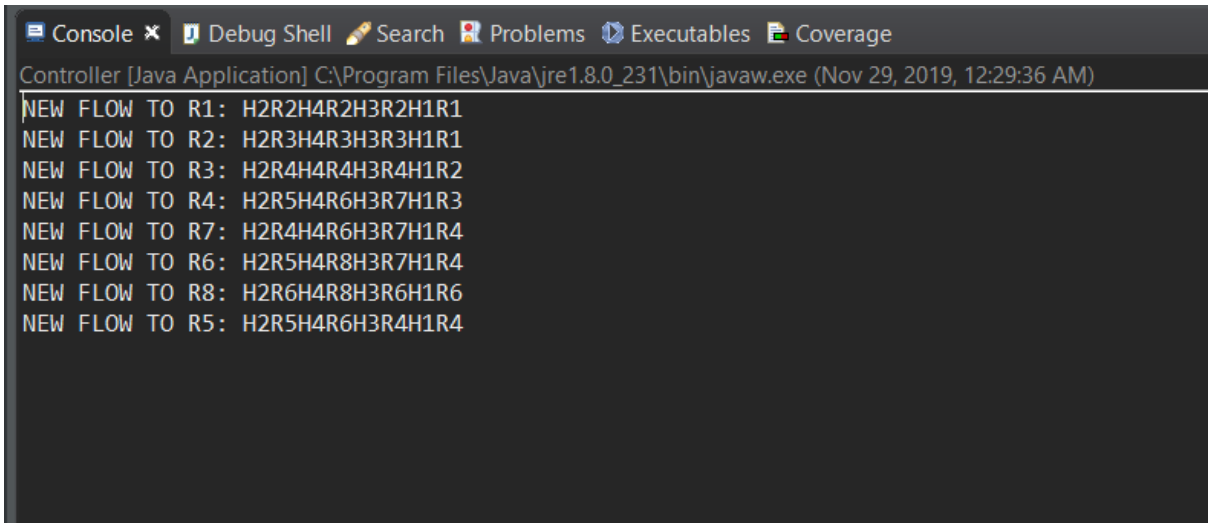
- It receives "HELLO", meaning it must set up a NodeData for the Router.
- It receives a Feature Reply, which means it must add an element of all this router's data to the ControllerFlowTable class (calling generateRoutersConnectionsFromFeatureReply method)
- It receives a PACIN, so it must generate a new Flow Table for the requesting router (calling createNewFlowTableForRouter method).

generateRoutersConnectionsFromFeatureReply will create a String Array from the string that the Router has sent in its Feature Reply. It will then call the addANewRouter in the controllerFlowTable class using this as a parameter. The controllerFlowTable contains an ArrayList of DirectConnectionsPerRouter elements (which is a class contained within the controllerFlowTable file), and a new element is added to an ArrayList every time the addANewRouter method is called. Within *that* class, is an ArrayList of the distances to each router connected to our new router.

If the router is requesting a new Flow Table, the createNewFlowTableForRouter method that is called gets the name of the Router requesting a new Flow Table, and passes just that name as a parameter to our ControllerFlowTable class's updateFlowChartForRouter method. This method is my implementation of Dijkstra's algorithm. Please have a look at the comments in this method in my submitted Java files for a full breakdown of this method.

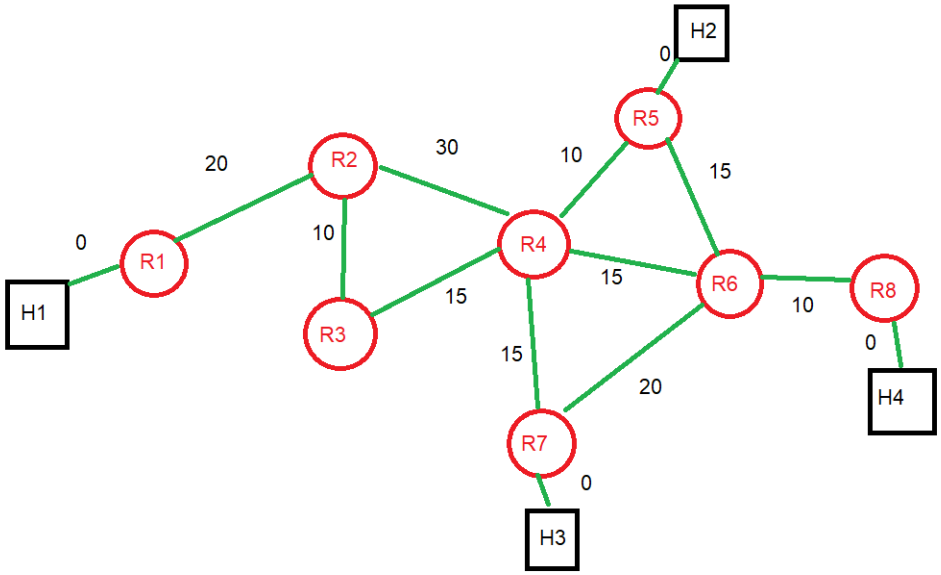
I decided to print all Flow Modifications to the Eclipse terminal so as to not clog up the program terminals. For the following screenshot, I started up every node on the network at once, and waited

for all the hosts to start sending strings. The following was printed to terminal. I left all the programs running for a few minutes, just to see if any of the routers would request another flow modification, which they shouldn't, as they already have all the information they need. Fortunately, no more flow modifications were requested:

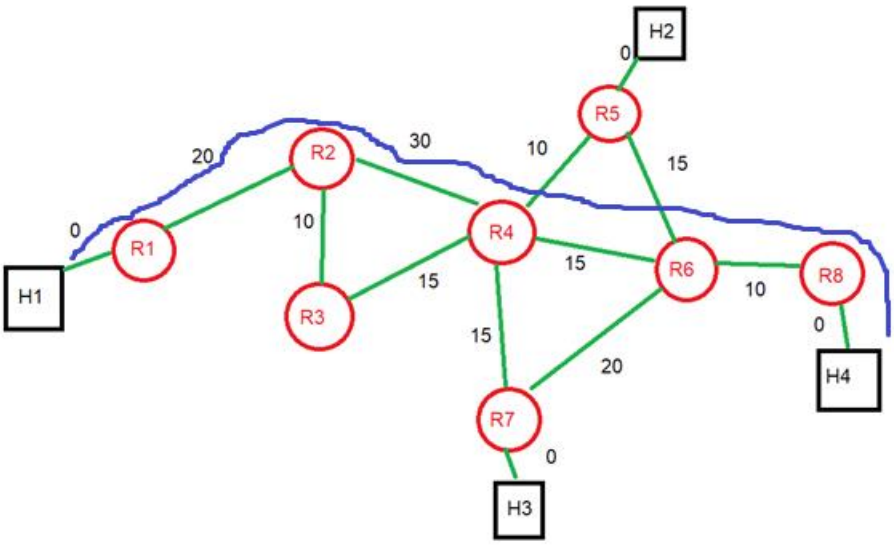


Demonstrating Full Network

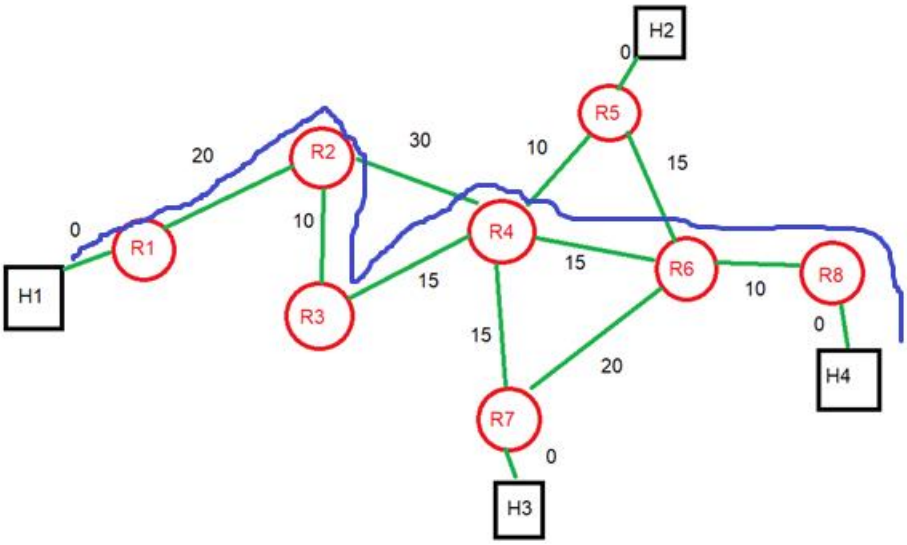
In the below demonstration, all the Router and Hosts are active and connected. We will be following a packet from H4 to H1. First, we have to look at the topography again:



If we were to go by least number of hops, then the packet would take the following route from H4:

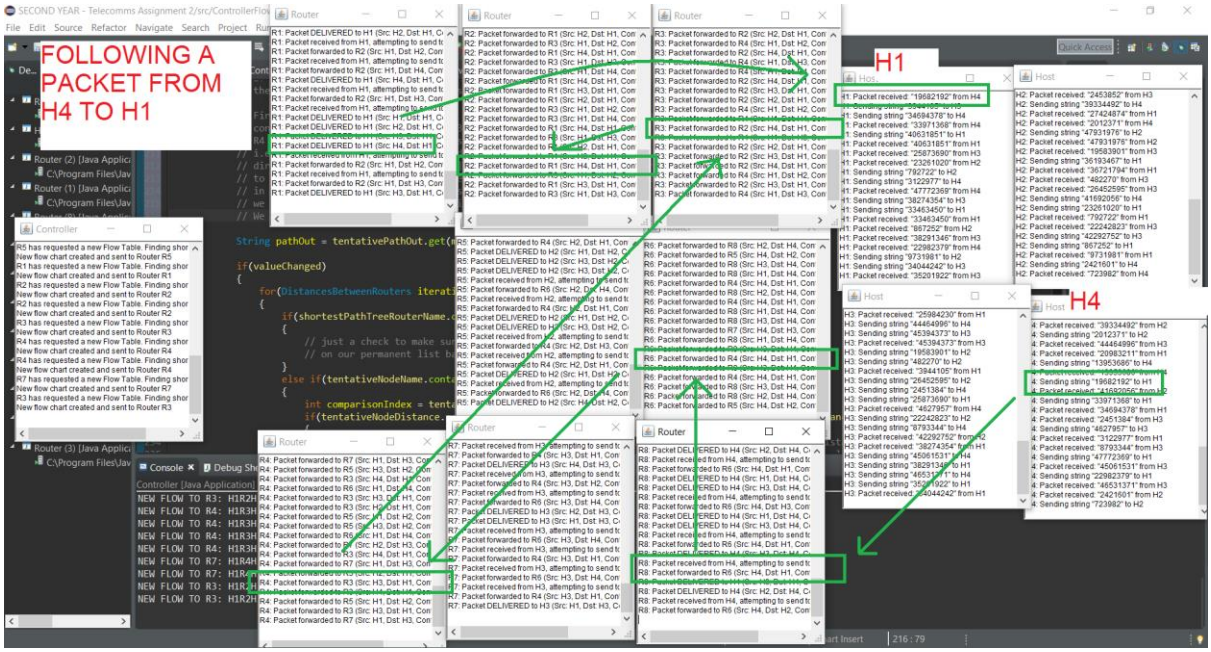


However, if we look closely at R2, R3 and R4, the distance between R2 and R4 is greater than the distance to get from R4 to R3 + R3 to R2. So, this is the shortest route:



The route will be H4 → R8 → R6 → R4 → R3 → R2 → R1 → H1

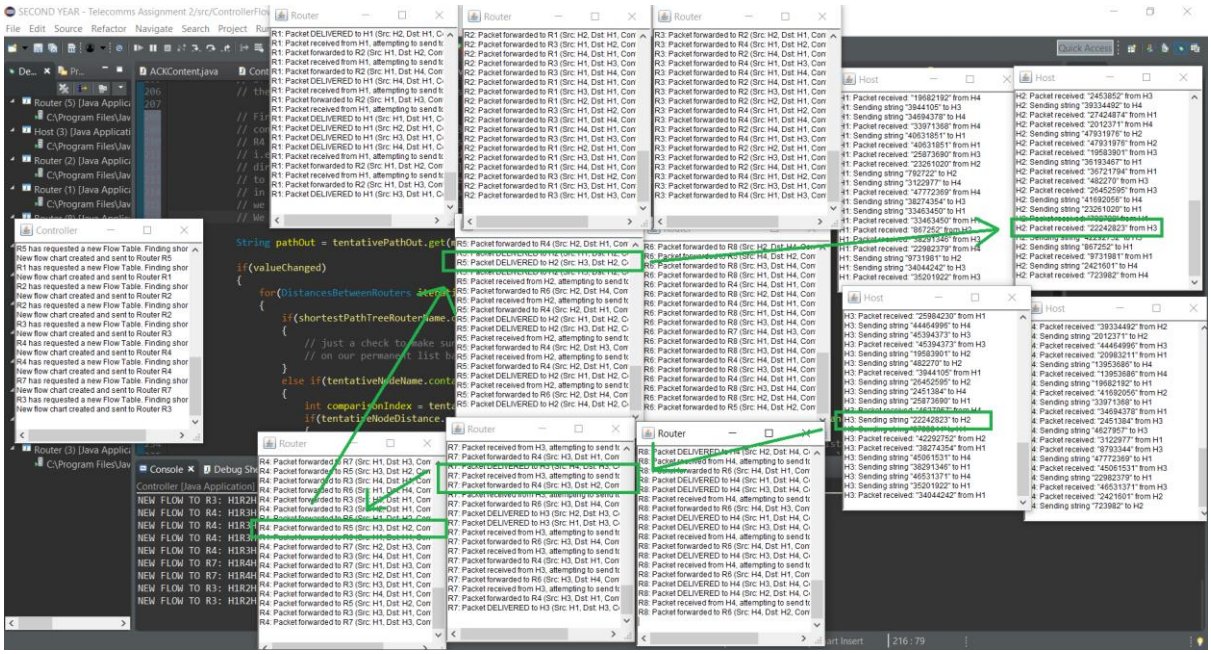
And when we run the program, we can see that this is the case (you're going to have to zoom right in for this one!):



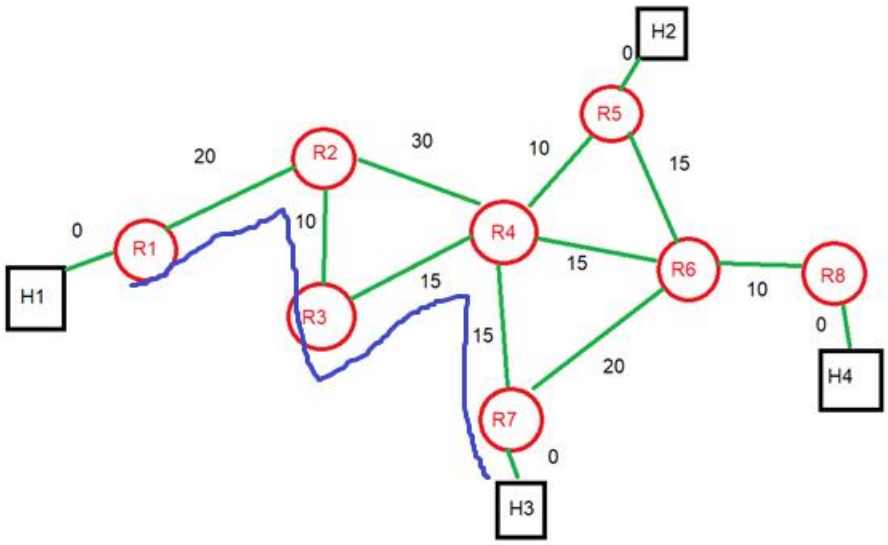
Also judging by the Topography, a packet going from H3 to H2 should have the following route:

H3 → R7 → R4 → R5 → H2

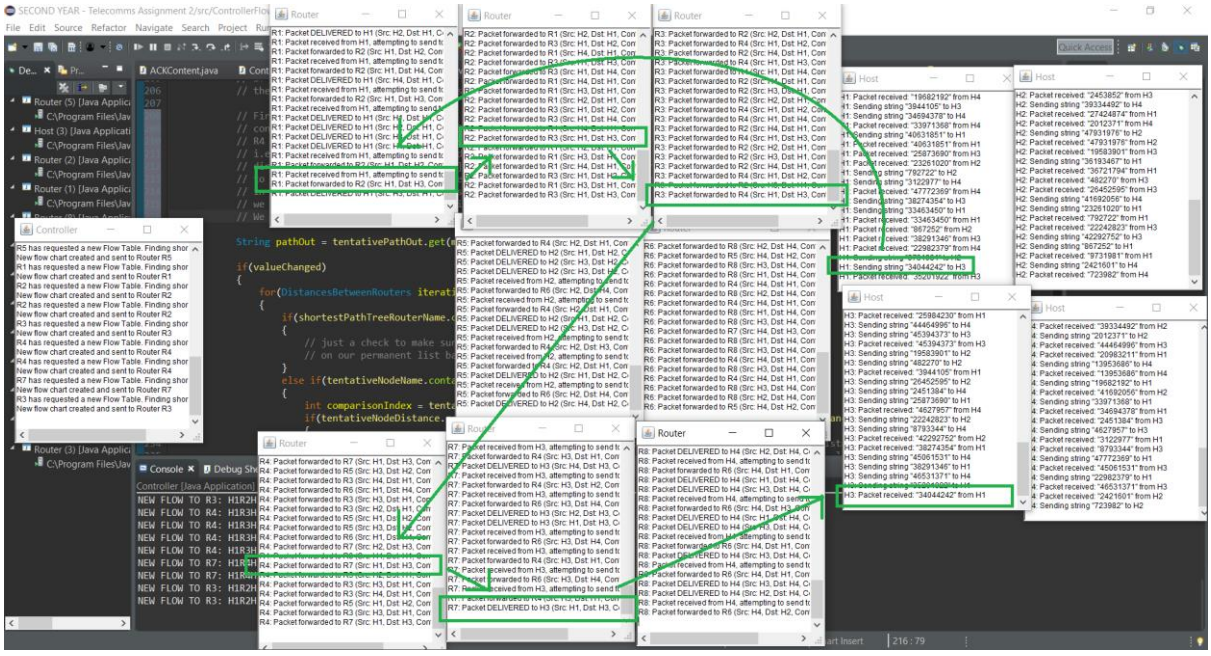
And again, we can see this is the case:



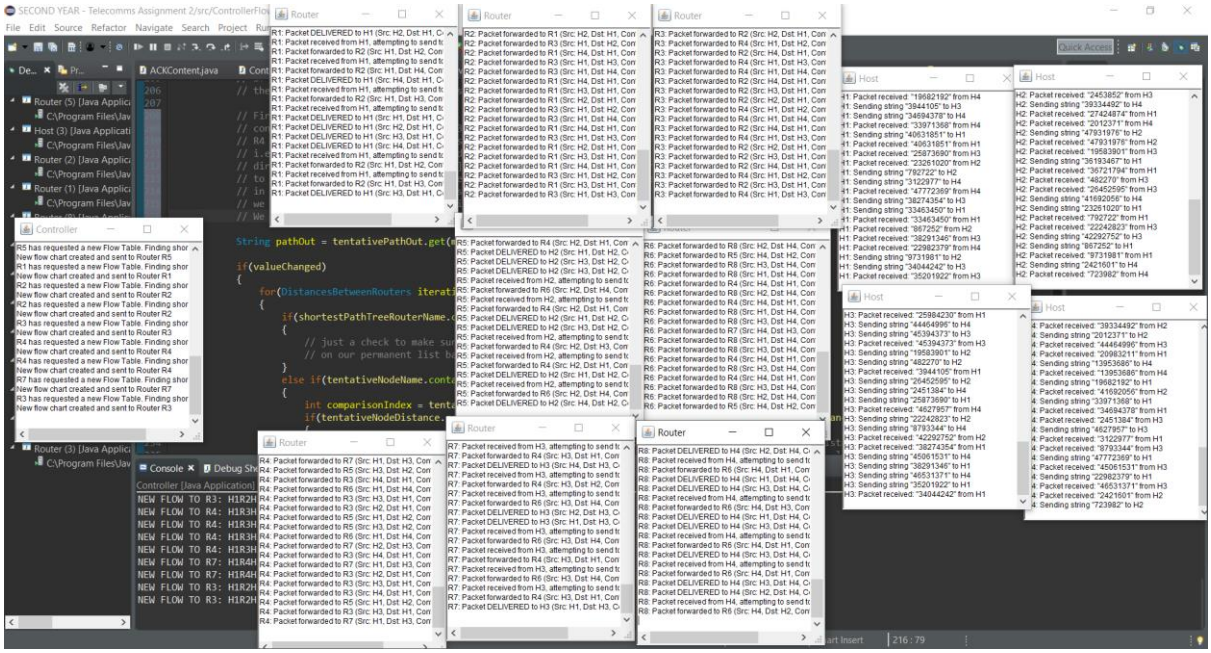
One last example, we will follow a packet going from H1 to H3. Judging by the topography, the route with the shortest distance is as follows:



The route will be H1 → R1 → R2 → R3 → R4 → R7 → H3



Here is the screenshot with no arrows, if you want to follow any other packet's routes:



Conclusion

I really enjoyed this assignment, as I was able to develop my understanding of routing, and I was able to carry over much of what I learned from Assignment 1. I also found that having the packets follow the exact route I intended was very rewarding. I was surprised by how straightforward Dijkstra's algorithm was to actually implement once I had all the Controller's recordings of the connected routers in place. Overall, this program took me about 25 hours to develop.