CS2031 Telecommunications II

Assignment 2 - OpenFlow-Inspired Routing

Seán Hassett

Contents

1. Theory **3**
1.1 Objective 3
1.2 OpenFlow Protocol 3
1.3 Link-State Routing 3
1.3.1 Dijkstra’s Algorithm 4
1.4 Distance-Vector Routing 4

2. Implementation **5**
2.1 Introduction 5
2.1.1 Note on Usage 5
2.2 Router Map 6
2.3 Main Classes 7
2.3.1 packet_utils.py 7
2.3.1.1 create_packet() 7
2.3.1.2 unpack() 8
2.3.1.3 print_packet() 8
2.3.2 controller.py 9
2.3.2.1 update_router_status() 9
2.3.2.2 configure_routes() 9
2.3.2.3 listen() 9
2.3.3 dijkstra.py 11
2.3.3.1 get_shortest_path() 11
2.3.4 router.py 13
2.3.4.1 send_request() 13
2.3.4.2 receive_messages() 13
2.3.5 clientN.py 15
2.3.5.1 send_messages() 15
2.3.5.2 receive_messages() 15
2.3.6 launcher.py 16

3. Testing **18**
3.1 Adding Weight to Paths 18
3.2 Wireshark Capture 19

4. Reflection **21**
4.1 Improvements 21
4.2 Conclusion 22

1. Theory

1.1 Objective

The goal of this assignment was to design a OpenFlow-like controller which handled the flow of traffic in a network of routers. Clients are paired with some of the routers and can send messages to each other. When a message is sent, it arrives at that client's router which then sends it through the network to the destination client's router which passes it to the destination client. The routers communicate with the controller when they don't know the optimal path for a packet to take. The controller uses Dijkstra's Algorithm to calculate the shortest paths by building a graph of the network using information sent from each router about the router's neighbouring connections.

1.2 OpenFlow Protocol

OpenFlow is a network protocol which involves controllers managing traffic through a network of routers or switches. Each router has a flow table which defines rules for forwarding packets and these rules are managed by the controller. The implications of OpenFlow are that it is much easier to modify the rules for forwarding packets since only the controller needs to be modified and not every individual router. This leads to cheaper routers since they do not have to be so sophisticated as they aren't doing any heavy calculations, all the work is handled by the controller. Once routers have the required rules from the controller, packet forwarding takes place with little delay.

OpenFlow makes modifications to the rules easier to effect since developers aren't dealing with proprietary hardware but with open source software. OpenFlow is associated with Software-Defined Networking, an approach to networking which seeks to separate the process of forwarding packets from the routing process. In OpenFlow, the forwarding of packets is handled by the routers and the routing process for all the routers is handled by the controller.

1.3 Link-State Routing

In Link-State Routing, routers communicate with each other, sending information about the routers they are connected to and the distances between them. In this way, the routers are able to construct a map of the network and know every connection. They can then calculate the shortest paths through the network for specific destinations. In our implementation, the routers send the information about their connections to the controller and each router communicates with the controller to receive pathing information. The controller makes use of Dijkstra’s Algorithm to calculate the shortest paths and send the information to the routers. The routers have their own routing tables and once they receive a path from the controller once, they store it in the table so the next packet can be sent without having to contact the controller.

1.3.1 Dijkstra’s Algorithm

Dijkstra’s Algorithm is a method of calculating the shortest distance between two nodes in a graph. We start off with a source node and set the distance from that node to every other node as infinity and mark all these nodes as unvisited. Starting with the node that is the shortest distance from the source node, we consider all of this node’s neighbours in turn. For each considered node, we compare the tentative distance to the currently stored distance value and if it is smaller, we replace the stored value with the new one as well as recording the node that this distance is associated with. When all of the neighbours of the node are considered, the node is marked as visited. We move to the next node which is the unvisited node with the shortest tentative distance to the source and repeat the process until we have visited every node. We end up with an optimal path from the source to any other node in the graph.

1.4 Distance-Vector Routing

With Distance-Vector Routing, each router communicates with its immediate neighbours, sending part or all of its routing table. Each router’s table informs the router which “direction” it needs to send packets in order to get to the destination. Routers aren’t aware of the full network, instead only knowing their neighbouring routers and which of those it needs to forward to for a given destination. Routers routinely send updates to each other which they use to keep their routing tables and will automatically send an update when a change to the network is made.

2. Implementation

2.1 Introduction

Having used Python for the first assignment, I decided to build on what I had learned with that and continue to use Python for this assignment. The main library used is the sockets library which contains everything necessary for creating sockets and sending and receiving information with them.

Following feedback from the first assignment, I decided in this assignment to use datagram sockets as opposed to stream sockets, which gave me greater freedom in binding sockets to specific addresses. With the stream sockets, once a connection request was accepted, a new socket was created with a port number assigned by the library outside of my control. With the datagram sockets, I could instead choose which port number the client sockets were to exist on which made it easier to make use of my packet header information.

2.1.1 Note on Usage

To start using the system, the launcher.py script needs to be executed as well as any or all of the client scripts. It's not important to start the launcher before the clients but the launcher must be run before any messages are sent between clients and of course, each client must be run before they can receive messages. There is more on the launcher script below.

It is also important that all the scripts are run from the same directory as some classes import others and can only see them if they are in the same directory.

2.2 Router Map

For testing my implementation at all stages of its development, I decided to design a system of clients and routers such that there were many different source & destination combinations and multiple paths through the system for each combination. The map I designed has 16 routers and 5 client endpoints, making 20 different combinations of clients (10 combinations in two directions) and many different paths for traffic to go between them.

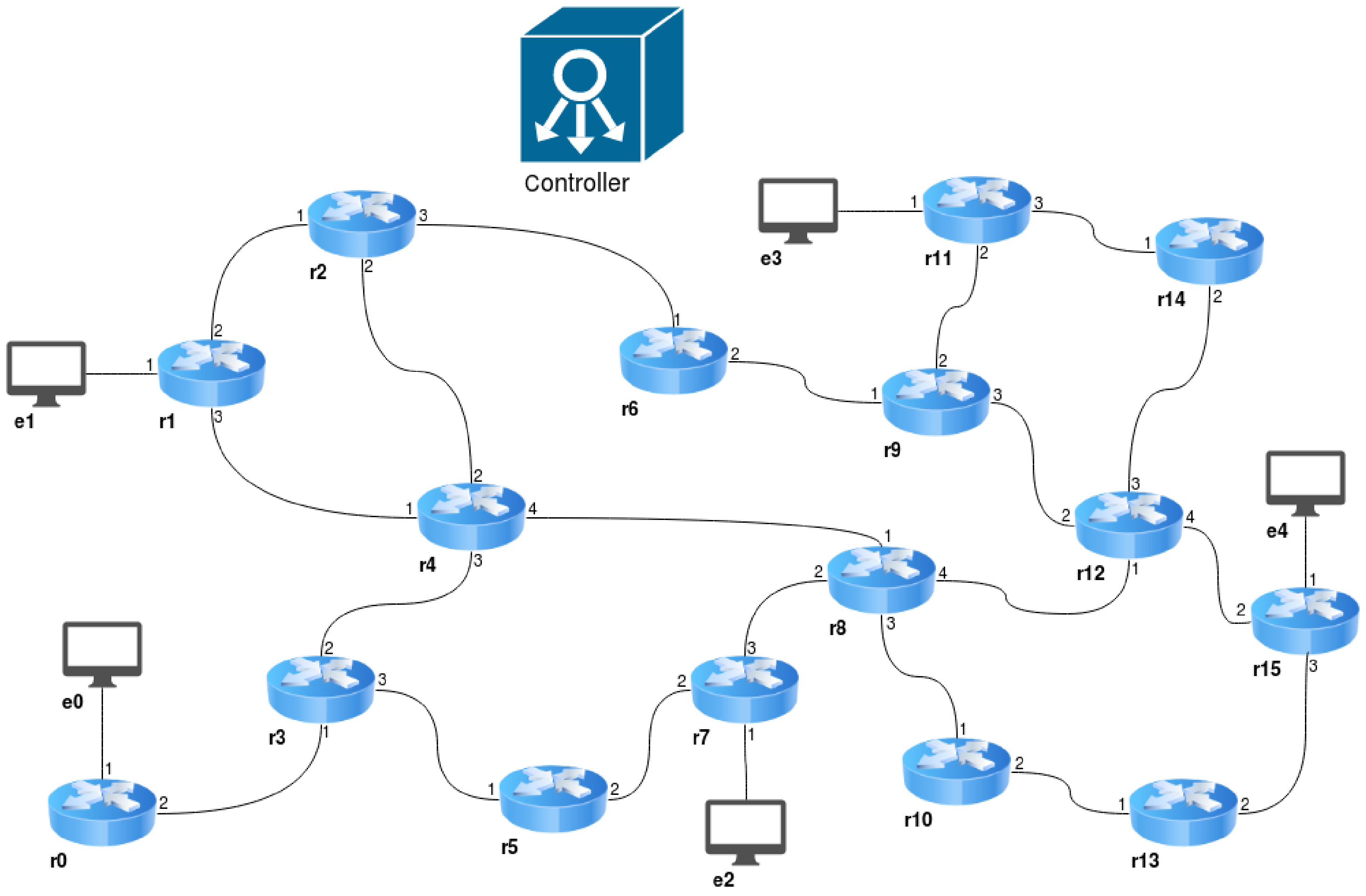


Fig. 2.1. Router Map

This map I felt was complex enough to allow for more thorough testing of my implementation, especially when it came to the controller calculating the shortest paths between clients.

2.3 Main Classes

I will now give a rundown of all the classes utilised in my implementation.

2.3.1 packet_utils.py

This class contains methods for conveniently dealing with packets according to the protocol I have defined. It is implemented by the clients, routers and controller.

The packet structure I have designed is as follows:

0	1	2-5	6-7	8-11	12-13	14-
seq no.	p type	source IP	src port	destination IP	dst. port	data

As seen, the packet consists of a 14 byte header with the payload starting from byte 14. The sequence number uses 1 byte. For this implementation I didn't make use of sequence numbers so each packet sent has a sequence number of zero.

The packet type is the next byte. This is a single integer which informs the receiver of what kind of information is in the payload and thus allows the receiver to behave in different ways depending on the packet type. The packet types I have used are as follows:

- Hello: Contains router configuration information, sent to the controller at router startup
- Client Message: Contains a message sent between clients
- Link State Request: Contains a source and destination that the router needs a route for, sent to the controller from each router
- Link State Update: Contains routing information, sent by the controller to each router

The IP's are 4 bytes each, one byte for each element of the IP address. The port numbers are 2 bytes each since the maximum value for port numbers is 2^16.

2.3.1.1 create_packet()

The create_packet() method takes in the seven pieces of information for a packet as parameters and constructs a hex representation of them, using leading zeroes where necessary to preserve the indexing of the bytes.

The sequence number, packet type and port numbers are treated as integers and formatted as hex strings with leading zeroes. The IP values are treated as four separate integers. Each integer is formatted as a hex string and the dots are ignored (but replaced by the unpack method). The data is passed in as a stream of bytes. Each byte is converted to a hex string and added to the end of the overall hex string. The hex representation is really stored as a string and so is primarily used for visualising the created packet and preserving indexing. This hex string is encoded as bytes and the encoded version is what is returned as the packet.

2.3.1.2 unpack()

The unpack method takes in a packet (stream of bytes) as a parameter and returns a namedtuple containing the seven pieces of information stored as plaintext strings. The namedtuple allows for direct referencing of the packet segments by name instead of by abstract numbering. For example, we can access packet.destination_ip instead of accessing packet[4]. This improves the readability of the code in many places.

2.3.1.3 print_packet()

This prints a hex representation of the packet and prints each of the packet segments as plaintext. This can be used for verifying that the packets are being sent with the correct structure.

2.3.2 controller.py

The controller class contains the data and methods needed for routing information through our network. When initialised, it binds a socket to a hardcoded address and then creates empty lists and dictionaries which it will populate when it receives Hello packets from the routers. It also contains a boolean called stable which is used to monitor the status of the router network. It is declared as True and whenever a new router is introduced, stable is set to False.

2.3.2.1 update_router_status()

This method runs on a thread started at initialisation and checks every second to see if the stable boolean has been set to False, meaning a new router has been introduced. If that is the case, the method will call for the routing information in the controller to be updated to reflect the new router. Checking every second gives time for a batch of routers to be introduced -- as happens with our launcher script -- and the controller will update only once instead of many times redundantly.

2.3.2.2 configure_routes()

This method populates the routing table according to the clients and routers that the controller has been made aware of. Its first step is to take the list of known clients and find each possible pairing of them. For efficiency, each pairing is only entered into the table in one direction, i.e. E1-E3 will be in the table but E3-E1 won't be. This halves the number of routes that need to be calculated since half of the routes are simply the other half in reverse.

The method then runs Dijkstra's Algorithm, imported from another script, using each of the clients as the source in turn and generating all the shortest paths necessary for the routing table. If a router is directly connected to the destination client it will automatically send the packet straight to the client and as such, the routes that the controller calculates only need to get the packet as far as the router that the client is connected to, the clients themselves are not part of the paths.

2.3.2.3 listen()

This method runs on a thread started at initialisation and listens for all incoming traffic on the controller's socket. On receipt of a packet, it uses packet_utils to unpack it and check the packet type. The controller expects two types of packets, a Hello or a Link State Request. If it receives a Hello, it adds in the information for that router to the appropriate data structures. It also checks to see if the router is connected to a client and if so, adds the client information as well. This is because the clients don't send their configuration information to the controller at startup.

If the controller receives a Link State Request, it expects the payload to be a route which it looks up in the routing table. If it can’t find that route, it reverses the route order and searches again. It is theoretically impossible for a router to send an invalid route so there is no error checking here, the controller assumes the route received will be in the table either as it was received or in reverse.

Having found the route in the table, the controller proceeds to send the routing information to each router in the route barring the very last one since that one will already know how to send the packet to the client as they are connected directly to it. If the route had to be reversed to find it in the table, the controller will reverse the routing information that it sends.

2.3.3 dijkstra.py

This script contains the function for calculating the shortest path using Dijkstra's Algorithm. This could be included as a class method in the controller class but since it was a specific part of the assignment brief, I decided to keep it separate to make it clearer and easier to find.

2.3.3.1 get_shortest_path()

This function is a Python implementation of the pseudocode for Dijkstra's Algorithm found on Wikipedia and shown below:

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:                 // Initialization
        dist[v] ← INFINITY                      // Unknown distance from source to v
        prev[v] ← UNDEFINED                     // Previous node in optimal path from  source
        add v to Q                              // All nodes initially in Q (unvisited nodes)

    dist[source] ← 0                            // Distance from source to source

    while Q is not empty:
        u ← vertex in Q with min dist[u]        // Node with the least distance will be selected
                                                   first
        remove u from Q

        for each neighbor v of u:               // where v is still in Q.
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:                   // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

The graph is passed in as a tuple containing a list, *vertices*, and a dictionary, *edges*. Vertices is a list of each router number and client. Edges is a dictionary where each key is one of the vertices and each value is another dictionary. In the nested dictionary, each key is another vertex that the parent is a neighbour of and the value is the distance between those two vertices. A sample from edges would look like so:

```
{'4': {'3': 2, '5': 4}}
```

Here we see that vertex '4' has a neighbour, vertex '3' which is a distance of 2 away, and another neighbour '5', which is a distance of 4 away.

Instead of returning both dist and prev, only prev is returned since we don't strictly need to know the distance of each path. The structure of prev is again a dictionary, with each key being a destination vertex and the value being the previous vertex you would have to reach to get to the destination. A sample prev output would be:

```
{'0': 'E0', 'E0': None, '1': '4', 'E1': '1', '2': '4', '3': '0',
 '4': '3', '5': '3', '6': '2', '7': '5', 'E2': '7',...}
```

This is part of the actual prev generated according to the route map above when taking E0 as the source. We can derive the route from E0 to E2 by starting at the end and working backwards:

E2	←	r7
r7	←	r5
r5	←	r3
r3	←	r0
r0	←	E0

Eventually, working backwards leads you to your source and you can see the path given to us by the algorithm: E0 → r0 → r3 → r5 → r7 → E2.

The controller takes the prev output and does this working backwards method for each route building up the path in reverse. At the end it reverses the full path so it is the right way around and adds it to the route table.

2.3.4 router.py

The router class allows for the creation of Router objects. Each router requires a router number to be passed in as a parameter as well as a list of nodes and list of neighbours. The list of nodes is a list of addresses and for each address, the router creates a socket. Each of these sockets is used to communicate with one of the router's neighbours. Each router is pre-configured to know each of its neighbours, as well as which node it uses to communicate with each neighbour, and the address of the node it contacts on the neighbour. They also have a node dedicated to contacting the controller.

The routers know the "distance" value between themselves and each of their neighbours. For the purposes of this assignment, the distance between every router is set to 1 since they are all on the same host and there is effectively zero latency between any of them. In the Testing section we will see how our implementation of Dijkstra's Algorithm handles weight on the edges between routers if we were to implement distances.

At initialisation, the router takes the list of its neighbours and sends the information to the controller. It then starts a thread for each socket it has and sets each of them to listen for incoming messages.

2.3.4.1 send_request()

This method simply takes in a source and destination client, formats them in such a way that the controller will understand and then sends a Link State Request packet on the router's controller socket for that route. It doesn't wait for a reply since the receive method handles Link State Update packets.

2.3.4.2 receive_messages()

This method is called on a separate thread for each socket that the router creates and listens on each socket for incoming packets. If a Link State Update packet is received then the information from the packet is entered into the router's routing table.

If a Client Message packet is received, the router unpacks the packet and extracts the source and destination addresses from the header. It then checks these against a hardcoded list of clients and their addresses to figure out the route that the packet is travelling. Once it knows the destination client, the first thing it does is check to see if the destination client is a neighbour and if so it sends the packet straight to that client.

If not, it checks its own routing table to see if it knows how to handle traffic for that destination. If so it sends the packet to the next router and if not, it calls send_request() and then sleeps for a tenth of a second to give the controller time to send the Link State Update. Since the Client Message packet arrived on a separate node to the one used to communicate with the controller and since each node operates on its own thread, the router is freely able to send and receive the Link State packets while the node receiving the client message waits.

After receiving the Link State Update, the router checks to see which node it should use to send out the packet. Since unpacking the packet doesn’t destroy or disrupt the original client packet, this original packet is simply forwarded on once the destination router is known.

2.3.5 clientN.py

These are the client endpoints from the router map. Each one is almost identical to the next with the exception of some configuration. Originally I wanted the clients to be included in the launcher script, however I had a problem with that as each client needs to have its own shell window to take in input. Finding a way to launch a client instance in a new shell window proved to be a challenge and finding a way to do so that would reliably work on another computer and potentially another OS was an even bigger challenge so unfortunately I had to settle for having five separate scripts.

Each client has a socket at a predefined address and sends and receives messages on this socket. Sending and receiving is done on separate threads and so the clients can freely send and receive on a constant basis without interruption, similar to an instant messaging system. In this vein, each client has a name associated with it which is used primarily as an aid for seeing which client instance is sending a message.

2.3.5.1 send_messages()

This method takes in input and stores it in a variable. The user is then asked for a destination client and once that is received the packet is constructed with the message as the payload and sent to whichever router the client is connected to.

2.3.5.2 receive_messages()

This method is very straight forward. On receipt of a packet, it unpacks it and prints the payload to the screen, as well as the name of whichever client sent it.

2.3.6 launcher.py

This is a convenience script used to initialise the controller and all the routers. It is hardcoded with the configurations for the routers according to the router map I specified. It creates a Controller object and then for every configuration it has creates a Router object. This makes it much easier to set up the system. All that needs to be done after running this is to run any or all of the client scripts and they will be able to communicate with each other.

The launcher also enables a neat way of tracking packets through the system, as every router is hooked into the same shell window. As such, I had each router print out its number upon receipt of a packet as well as the node it received the packet on and the controller also prints some information. If we send a packet from E1-E4, the output to the launcher window is as follows:

```
1:1

Controller sending route information…
E1-E4: [1, 4, 8, 12, 15]

1:0
4:0
8:0
12:0
4:1
8:1
12:1
15:2
```

Stepping through this, we can see that at first, router 1 receives a packet on node 1, which is the client packet from E1. The next thing we see is the controller printing out the route it received as a request as well as the path it calculated for that route. We then see each of the routers in the path receive a packet on node 0, which is the controller sending the Link State Update to each router. Router 15 doesn't get an update since it's connected to E4 and knows where to send the packet already.

After the Link State Update, we see each router in the path receiving the client packet at the appropriate node until it arrives at the final router which, we can see from the client window, sends the packet on to the client. Sending another packet from E1 to E4 yields the following output:

```
1:1
4:1
8:1
12:1
15:2
```

This time, as desired we see each router in the path receiving the packet in turn and no request to the controller, since the routers already know this route. Finally if we send a packet back from E4 to E1 we get the following:

```
15:1

Controller sending route information...
E4-E1: [15, 12, 8, 4, 1]

15:0
12:0
4:0
8:0
12:4
8:4
4:4
1:3
```

This time we see the controller giving the route in reverse. All the same routers are involved but they receive the client packet on different nodes since it’s travelling the opposite way.

3. Testing

3.1 Adding Weight to Paths

To test the implementation of Dijkstra's Algorithm taking the weight of paths into consideration, I added a weight of 10 to all paths leading to router number 4. This was done in the controller while setting up the edges dictionary by adding a check to see if the neighbour router being added was r4 and if so, setting the weight to 10 instead of 1.

Earlier we saw that the optimal path from E1-E4 was:

$$\boxed{\mathbf{1 \rightarrow 4 \rightarrow 8 \rightarrow 12 \rightarrow 15}}$$

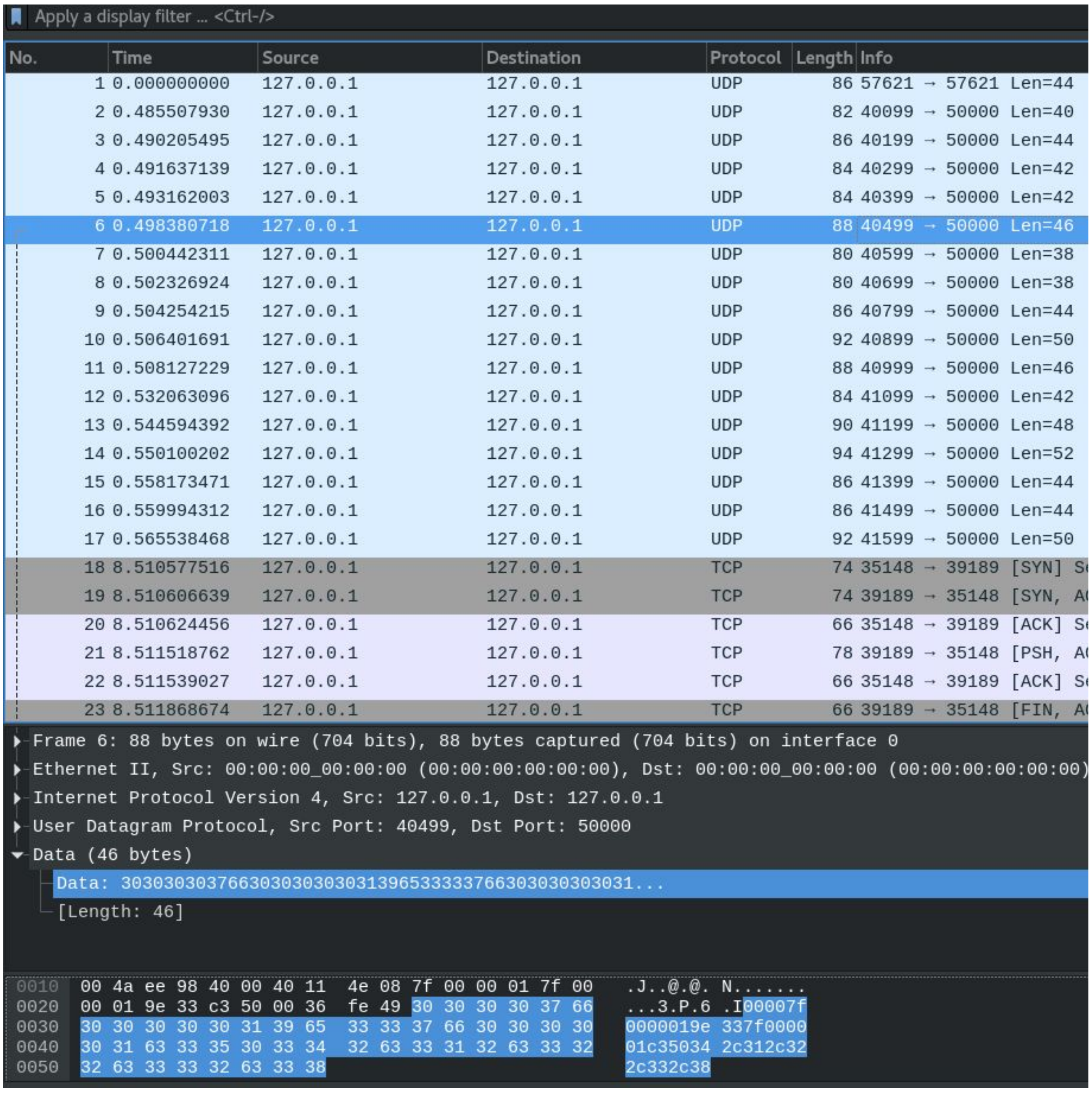
With the hop from r1 to r4 now costing ten times as much as before, we should see a new route calculated. And indeed, with this configuration the controller returns:

$$\boxed{\mathbf{1 \rightarrow 2 \rightarrow 6 \rightarrow 9 \rightarrow 12 \rightarrow 15}}$$

as the optimal route, bypassing the now expensive hop to r4 in favour of an extra hop to go around it. So if we were to calculate the latency between each of the routers and factor that in as distance values, our controller would still be fully capable of calculating the optimal routes based on the new information.

3.2 Wireshark Capture

To see packets travelling through the network and verify that the paths being printed to the launcher shell window were in fact the paths being taken, I captured some traffic using Wireshark. First of all I started the launcher script and monitored the resulting traffic:



```
Apply a display filter ... <Ctrl-/>
No.   Time         Source      Destination  Protocol Length Info
    1 0.000000000  127.0.0.1   127.0.0.1    UDP      86 57621 → 57621 Len=44
    2 0.485507930  127.0.0.1   127.0.0.1    UDP      82 40099 → 50000 Len=40
    3 0.490205495  127.0.0.1   127.0.0.1    UDP      86 40199 → 50000 Len=44
    4 0.491637139  127.0.0.1   127.0.0.1    UDP      84 40299 → 50000 Len=42
    5 0.493162003  127.0.0.1   127.0.0.1    UDP      84 40399 → 50000 Len=42
    6 0.498380718  127.0.0.1   127.0.0.1    UDP      88 40499 → 50000 Len=46
    7 0.500442311  127.0.0.1   127.0.0.1    UDP      80 40599 → 50000 Len=38
    8 0.502326924  127.0.0.1   127.0.0.1    UDP      80 40699 → 50000 Len=38
    9 0.504254215  127.0.0.1   127.0.0.1    UDP      86 40799 → 50000 Len=44
   10 0.506401691  127.0.0.1   127.0.0.1    UDP      92 40899 → 50000 Len=50
   11 0.508127229  127.0.0.1   127.0.0.1    UDP      88 40999 → 50000 Len=46
   12 0.532063096  127.0.0.1   127.0.0.1    UDP      84 41099 → 50000 Len=42
   13 0.544594392  127.0.0.1   127.0.0.1    UDP      90 41199 → 50000 Len=48
   14 0.550100202  127.0.0.1   127.0.0.1    UDP      94 41299 → 50000 Len=52
   15 0.558173471  127.0.0.1   127.0.0.1    UDP      86 41399 → 50000 Len=44
   16 0.559994312  127.0.0.1   127.0.0.1    UDP      86 41499 → 50000 Len=44
   17 0.565538468  127.0.0.1   127.0.0.1    UDP      92 41599 → 50000 Len=50
   18 8.510577516  127.0.0.1   127.0.0.1    TCP      74 35148 → 39189 [SYN] Se
   19 8.510606639  127.0.0.1   127.0.0.1    TCP      74 39189 → 35148 [SYN, AC
   20 8.510624456  127.0.0.1   127.0.0.1    TCP      66 35148 → 39189 [ACK] Se
   21 8.511518762  127.0.0.1   127.0.0.1    TCP      78 39189 → 35148 [PSH, AC
   22 8.511539027  127.0.0.1   127.0.0.1    TCP      66 35148 → 39189 [ACK] Se
   23 8.511868674  127.0.0.1   127.0.0.1    TCP      66 39189 → 35148 [FIN, AC
▸ Frame 6: 88 bytes on wire (704 bits), 88 bytes captured (704 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ User Datagram Protocol, Src Port: 40499, Dst Port: 50000
▾ Data (46 bytes)
    Data: 303030303766303030303031396533333766303030303031...
    [Length: 46]

0010  00 4a ee 98 40 00 40 11  4e 08 7f 00 00 01 7f 00   .J..@.@. N.......
0020  00 01 9e 33 c3 50 00 36  fe 49 30 30 30 30 37 66   ...3.P.6 .I00007f
0030  30 30 30 30 30 31 39 65  33 33 37 66 30 30 30 30   0000019e 337f0000
0040  30 31 63 33 35 30 33 34  32 63 33 31 32 63 33 32   01c35034 2c312c32
0050  32 63 33 33 32 63 33 38                            2c332c38
```

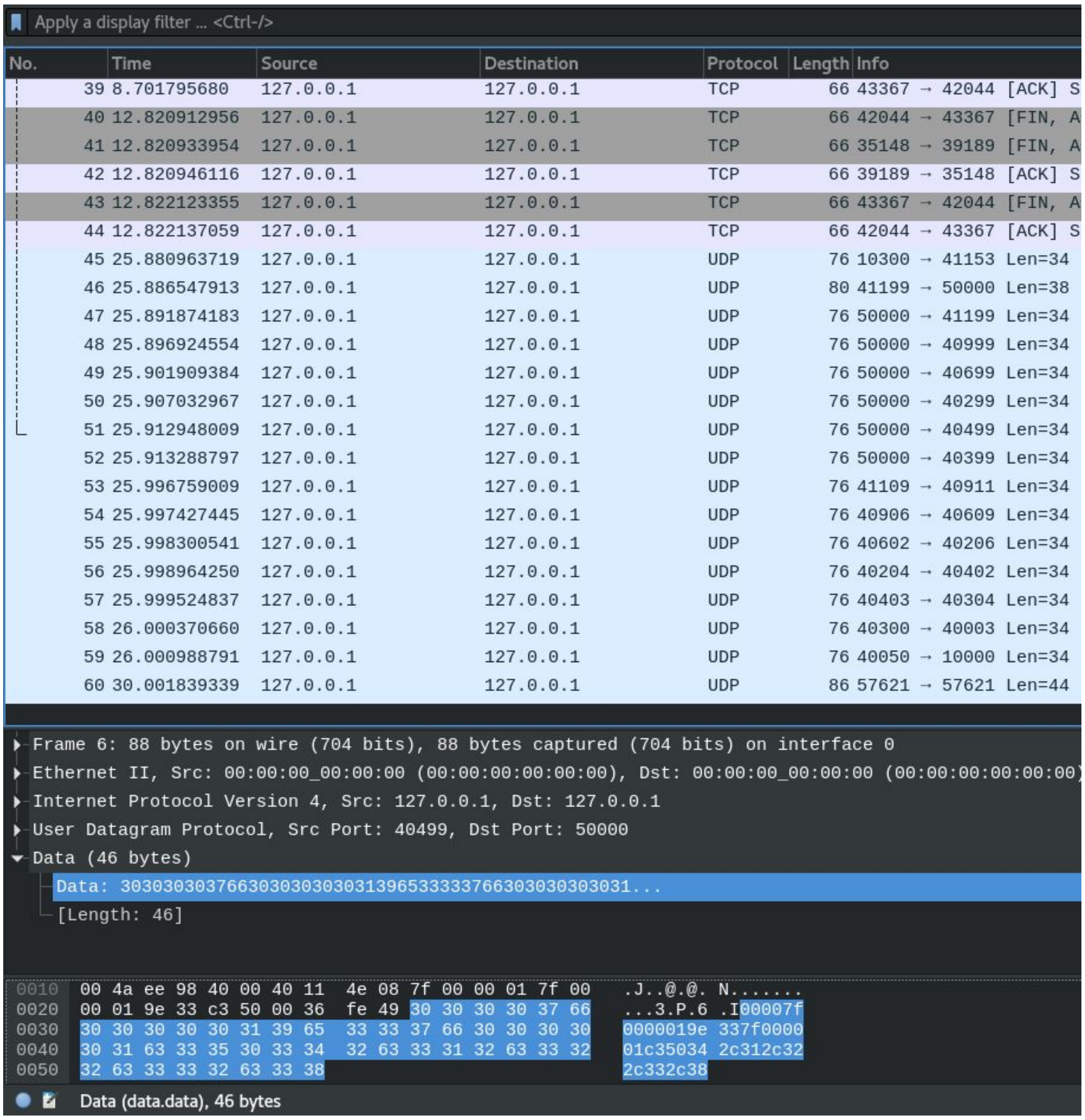
Fig. 3.1. Router startup

As seen in the capture, on startup each router sends a packet to the controller at port 50000. The payload of these packets is our own packet and we can see from our packet header that the packet type is 0, hence these are Hello packets as expected.

Next, I sent a packet from E0 to E3 which yielded the following path from the controller:

$$\boxed{11 \rightarrow 9 \rightarrow 6 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 0}$$

Examining the traffic that we captured with Wireshark and cross-referencing the router configuration information from the launcher script, we can see that the packet moves through the network exactly as expected.



```
Apply a display filter ... <Ctrl-/>
No.   Time          Source      Destination  Protocol Length Info
   39 8.701795680   127.0.0.1   127.0.0.1    TCP      66 43367 → 42044 [ACK] S
   40 12.820912956  127.0.0.1   127.0.0.1    TCP      66 42044 → 43367 [FIN, A
   41 12.820933954  127.0.0.1   127.0.0.1    TCP      66 35148 → 39189 [FIN, A
   42 12.820946116  127.0.0.1   127.0.0.1    TCP      66 39189 → 35148 [ACK] S
   43 12.822123355  127.0.0.1   127.0.0.1    TCP      66 43367 → 42044 [FIN, A
   44 12.822137059  127.0.0.1   127.0.0.1    TCP      66 42044 → 43367 [ACK] S
   45 25.880963719  127.0.0.1   127.0.0.1    UDP      76 10300 → 41153 Len=34
   46 25.886547913  127.0.0.1   127.0.0.1    UDP      80 41199 → 50000 Len=38
   47 25.891874183  127.0.0.1   127.0.0.1    UDP      76 50000 → 41199 Len=34
   48 25.896924554  127.0.0.1   127.0.0.1    UDP      76 50000 → 40999 Len=34
   49 25.901909384  127.0.0.1   127.0.0.1    UDP      76 50000 → 40699 Len=34
   50 25.907032967  127.0.0.1   127.0.0.1    UDP      76 50000 → 40299 Len=34
   51 25.912948009  127.0.0.1   127.0.0.1    UDP      76 50000 → 40499 Len=34
   52 25.913288797  127.0.0.1   127.0.0.1    UDP      76 50000 → 40399 Len=34
   53 25.996759009  127.0.0.1   127.0.0.1    UDP      76 41109 → 40911 Len=34
   54 25.997427445  127.0.0.1   127.0.0.1    UDP      76 40906 → 40609 Len=34
   55 25.998300541  127.0.0.1   127.0.0.1    UDP      76 40602 → 40206 Len=34
   56 25.998964250  127.0.0.1   127.0.0.1    UDP      76 40204 → 40402 Len=34
   57 25.999524837  127.0.0.1   127.0.0.1    UDP      76 40403 → 40304 Len=34
   58 26.000370660  127.0.0.1   127.0.0.1    UDP      76 40300 → 40003 Len=34
   59 26.000988791  127.0.0.1   127.0.0.1    UDP      76 40050 → 10000 Len=34
   60 30.001839339  127.0.0.1   127.0.0.1    UDP      86 57621 → 57621 Len=44

▸ Frame 6: 88 bytes on wire (704 bits), 88 bytes captured (704 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ User Datagram Protocol, Src Port: 40499, Dst Port: 50000
▾ Data (46 bytes)
    Data: 30303030376630303030303139653333376630303030303031...
    [Length: 46]

0010  00 4a ee 98 40 00 40 11  4e 08 7f 00 00 01 7f 00   .J..@.@. N.......
0020  00 01 9e 33 c3 50 00 36  fe 49 30 30 30 30 37 66   ...3.P.6 .I00007f
0030  30 30 30 30 30 31 39 65  33 33 37 66 30 30 30 30   0000019e 337f0000
0040  30 31 63 33 35 30 33 34  32 63 33 31 32 63 33 32   01c35034 2c312c32
0050  32 63 33 33 32 63 33 38                            2c332c38

Data (data.data), 46 bytes
```

Fig. 3.2. Packet transfer

First of all, we see a packet transferred from E3 at port 10300 to r11 at port 41153. Since this is the first packet sent since startup, we then see r11 make a request to the controller at port 50000. We then see packets arriving from 50000 to each of the routers in the path and finally we see the routers passing the packet along the path until it reaches E0 at port 10000. Examining the payload of these packets lets us verify the packet type for each one.

The router ports in the launcher script are set up in such a way that makes it easy to see which routers and nodes are being used. Every router exists on a port number in the range 40000 - 49999. The “hundreds” part of the port number is determined by the router number, so router 8 uses the range 40800 - 40899. The “units” part of the port number is determined by the connecting object at that node. If the node connects router 8 to router 9 then the address will be 40809, if it connects router 8 to client 3 it will be 40853.

This system makes it convenient for identifying nodes by the port number alone, but for a real-world implementation where any of these ports could be already taken, it may be more practical to have the routers attempt to connect to a range of ports until it successfully identifies a free port. This would also require routers to derive the address of connected nodes by sending packets instead of having it hardcoded in.

4. Reflection

4.1 Improvements

While I am happy overall with my implementation of the assignment, there are a few areas where I would have liked to improve.

A) Including the clients in the launcher script.

This would make it extremely simple to start up the system for testing but the problems of having a separate window for each client were not easily solved. Trying to open a new terminal window for each client may have worked on my computer but it would be very difficult to do it in such a way that it would work on any computer and any OS. There was another potential solution using Python's GUI library Tkinter to design a terminal window for the clients to run in but this approach seemed a bit too time-intensive for the purposes I needed.

B) Adding routers dynamically.

This would be much better than having to hardcode a lot of configuration into the launcher script. Since the controller can already receive new routers without having to restart, only the Router class would have to be modified. Doing this would involve having the routers send Hello packets to their neighbours on startup as well as the controller so that each router would know it has a new neighbour without having to restart. It would also mean routers resending their configuration to the controller on receipt of a Hello packet from another router and we would also need an interface for configuring a new router and entering it into the system after running the launcher.

C) Error handling.

Much of the code in the implementation depends upon everything going to plan. There are currently no contingencies for dropped packets or blocked ports which should be checked. It would also be better to have a method of closing the launcher and client instances without having to use a keyboard interrupt.

4.2 Conclusion

Overall I am pleased with how the assignment went. I think it was a good idea to design the router map very early on as it gave a solid testing ground for building up the implementation. Some of the early parts like coming up with the router configurations were slow going at times but once I got to the controller implementation and especially the use of Dijkstra’s Algorithm and seeing that working I began to really enjoy making the program.