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

CS2031 Telecommunication II
Assignment #2: OpenFlow

Euan Leith, Std# 18323530

November 28, 2019

Contents

1 Introduction

The goal of this project is to create an OpenFlow system that transmits packets between nodes in Java using routing.

The system is made up of a number of nodes with various connections between them; see figure 1 These nodes can send packets to other nodes in the network to which they are connected. Each node is supplied with a routing table by which it can route packets to the appropriate destinations.

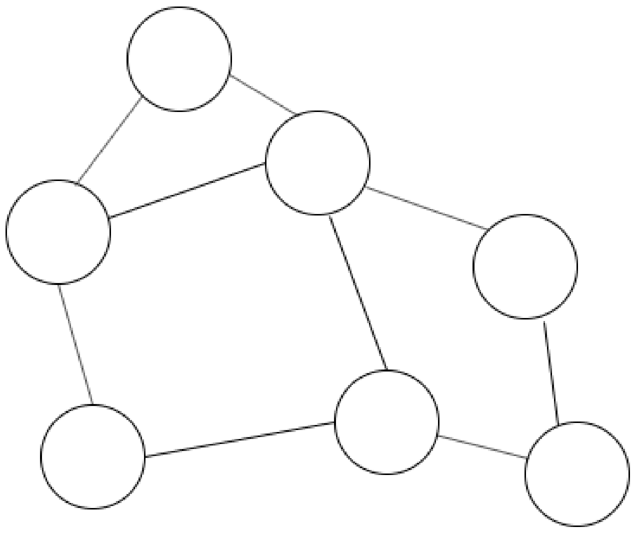


Figure 1: Shows an example topology of a network with interconnected nodes.

Note that parts of this project were re-used (and improved upon) from the previous project, and thus will not be explained in this report; namely nodes, packets, timers and listeners, as well as the basic processing of packets.

2 Theory

This section will explain routing, OpenFlow, and the Link State and Distance Vector routing protocols.

2.1 Routing

In a system containing nodes with connections to multiple other nodes, each node must know which of these nodes to forward the incoming packet to. This is so that the packet reaches its destination, but also so that it does so by the fastest possible route. This information is stored in each node in the form of a routing table; see table 1. This table contains the packet's destination, which maps to the node to which the packet is to be forwarded. This destination will be information stored in the packet. Dynamic routing protocols will also contains the distance taken to get to the destination in each routing table. This is so that the current path can be compared to any new paths to decide which is the shortest.

Destination	Distance	Next Port
50001	0	-
50002	1	50002
50003	2	50002
50004	1	50004
50005	2	50004

Table 1: Shows the routing table for the router with port 50001 connected directly to routers 50002 and 50004, and indirectly to routers 50003 and 50005.

2.2 OpenFlow Protocols

A system can have hard-coded routing tables for each router, however it can be more useful to have routing tables that are calculated dynamically. This is known as OpenFlow. This approach allows for nodes to be added to and removed from the system, and links to break without issue, as well as being more scalable. In large systems such as the internet, dynamic routing is effectively required as a result. This project will be exploring two of such routing protocols; Link State Routing and Distance Vector Routing.

Link State Routing
In this protocol, each router shares information about its neighbours with everyone. This information contains any neighbours that router may have, and the distance to those neighbours, as well as any information about those neighbours. Because of this, each router doesn't have to get information directly from every other router in the system to get a full picture of the system's structure. Once a router has received information about all other routers in the system, it can calculate the shortest path from each endpoint to all other endpoints. This is done using Dijkstra's Shortest-Path Algorithm:

1. Start with the local node (router): the root of the tree.
2. Assign a cost of 0 to this node and make it the first permanent node.
3. Examine each neighbour node of the node that was the last permanent node.
4. Assign a cumulative cost to each node and make it tentative.
5. Among the list of tentative nodes;
   (a) If a node can be reached from more than one direction, select the direction with the shortest cumulative cost.
   (b) Find the node with the smallest cumulative cost and make it permanent.
6. Repeat steps 3 to 5 until every node becomes permanent.

Whenever a node is added or removed from the network, or a link is broken, the routes are updated with regards to that new node.
Distance Vector Routing
In this protocol, each router shares information about the system with its neighbours. This information contains any endpoints the router is connected to, and the number of hops taken. This is performed repeatedly, and with each subsequent cycle, each router gets information from other routers one more hop away. Note that the number of hops is incremented with each cycle, and that each router need only remember information regarding the first instance of each endpoint it received. This is because any information received later will be from paths that took longer than the first to reach that router. Eventually each router will have information about all other routers in the system, in the form of endpoints which map to the neighbour to forward a given packet to. This process must then be repeated any time a new router is added to or removed from the system, as well as periodically so that if a connection to a router is lost, the other routers can calculate a new path.

Note that this approach is imperfect, as it can run into the 'Count to Infinity' problem: Consider three nodes A, B, and C, as in figure 2, where all nodes know how to get to all other nodes in the network. A knows that it can get to C via B. If the link between B and C is lost, B will no longer know how to get to C. It is possible that B can send any updates, A sends an update saying it can to C via B at a cost of 2. B will then update its route to C via A at a cost of 3. A will then receive updates from B updating its cost to 4. This will happen forever, with the cost counting to infinity.

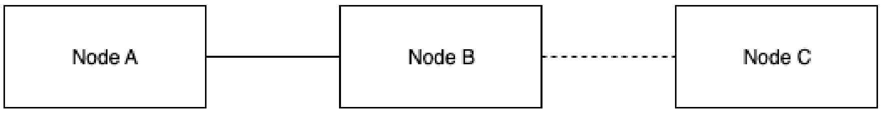


Figure 2: Shows interconnected nodes. Link between nodes B and C breaks, and the 'Count to Infinity' problem occurs.

3 Implemenation

This project was undertaken in two parts; creating a system with static routing, and then with dynamic routing. The former is much easier to implement as no routing algorithms are required, and thus was useful to implement first so as to understand the details of routing. However, the latter is required in order to maintain a system which is prone to change.
By either method, routing tables were created for each router. Once these routing tables are created, packets can be sent from one node to another across the network.

3.1 Static Routing

The topology for this system is a controller, a number of routers with various connections between them, and endpoints connected to various routers; see figure 3. The controller has hard-coded routes for each router. These routes are sent to each router to form routing tables. The endpoints can send packets to other endpoints via routers. Note that this could also have been implemented without endpoints, where routers send packets to other connected routers.

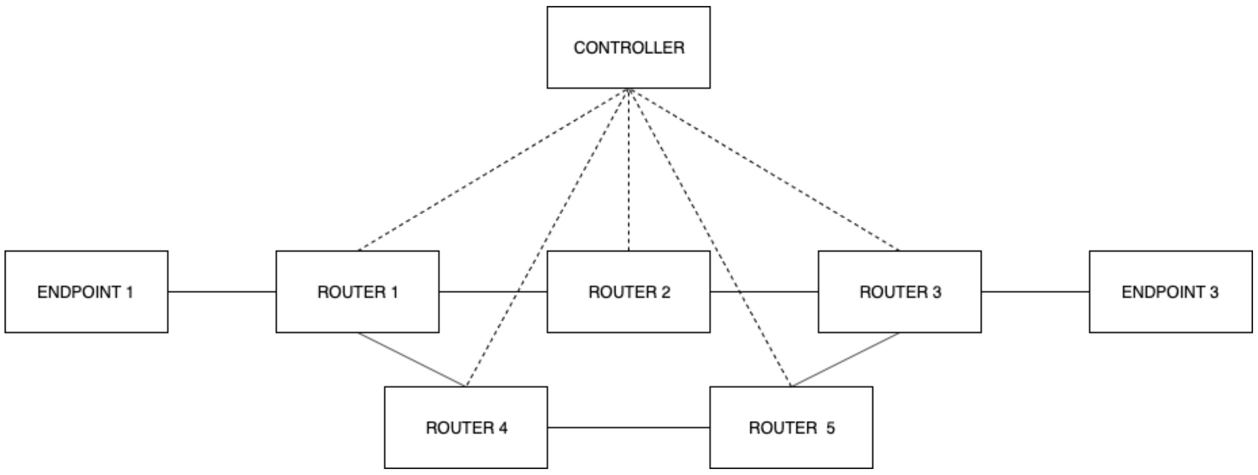


Figure 3: Shows an example topology of a network with a controller connected to all of the routers, routers with various connections, and endpoints connected to routers one and three.

The controller has a routing table, which maps the port receiving the packet and the port of the destination endpoint to the next port to forward the packet to; see listing 1

```
TwoKeyMap<Integer, Integer, Integer> routingTable; // srcPort, dstPort:nextPort
```

Listing 1: Controller's routing table for a static routing system which maps the previous port and the destination port to the next port.

The map 'TwoKeyMap' which takes two keys and returns the associated value was created for this purpose.

Each endpoint has a single, hard-coded router port to which it is connected. The controller's routing table is hard-coded into the system by simply inputting the ports manually; see listing 2. Note that this is the implementation of the example topology in figure 3.

```
// E1 -> E2
routingTable.put(E1_PORT,E2_PORT,R1_PORT);
routingTable.put(R1_PORT,E2_PORT,E2_PORT);
routingTable.put(R2_PORT,E2_PORT,R3_PORT);
routingTable.put(R3_PORT,E2_PORT,E2_PORT);
```

```
// E2 -> E1
routingTable.put(E2_PORT,E1_PORT,R1_PORT);
routingTable.put(R3_PORT,E1_PORT,R2_PORT);
routingTable.put(R2_PORT,E1_PORT,R1_PORT);
routingTable.put(R1_PORT,E1_PORT,E1_PORT);
```

Listing 2: Controller's routing table is hard-coded.

The inputted routing is the shortest path for this example system, with router's four and five being ignored. However one could hard-code paths other than the shortest path. Thus the designer must manually calculate the shortest paths for each case.

The controller then sends the destination port - next port maps to the associated routers. These routers then construct a routing table with these maps; see listing 3.

```
HashMap<Integer,Integer> routingTable; // dstPort:nextPort
```

Listing 3: Router's routing table for a static routing system which maps the destination port to the next port.

3.2 Dynamic Routing

The topology for this system is a number of routers with various connections between them; see figure 4 Each of these routers has a routing table, and routers can send packets to other connected routers. No controller is required with this approach, and all routers are endpoints that can send their own packets, as well as forward other packets.

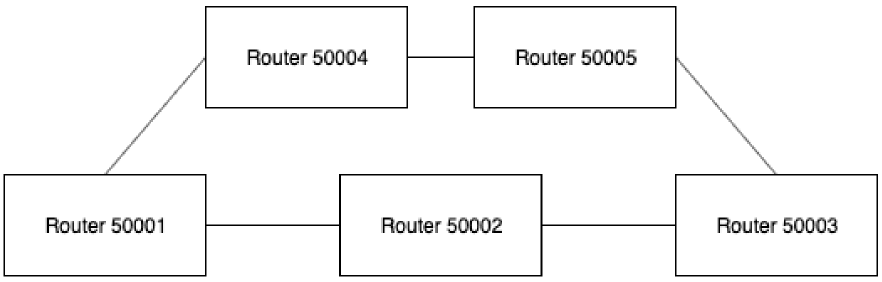


Figure 4: Shows the example topology of a network with routers with various connections.

Each router has its neighbours and all other routers in the network hard-coded upon creation; see listing 4. Note that this is the implementation of the example topology in figure 4.

```
int[] routers = new int[]{50001,50002,50003,50004,50005};
Router r1 = new Router(50001,new int[]{50002,50004},routers);
Router r2 = new Router(50002,new int[]{50001,50003},routers);
Router r3 = new Router(50003,new int[]{50002,50005},routers);
Router r4 = new Router(50004,new int[]{50001,50005},routers);
Router r5 = new Router(50005,new int[]{50004,50003},routers);
```

Listing 4: Information about each router in the network, and each router's neighbours is hard-coded

This information is then stored for each router; see listing 5.

```
Router(int srcPort, int[] neighbours, int[] routers) {
        // init portToNeighbours with routerPort : <empty>
        // for all routers in network
        portToNeighbours = new HashMap<>();
        for (int routerPort : routers) {
            portToNeighbours.put(routerPort, new HashMap<>());
        }

        // add info for this router
        for (int neighbourPort : neighbours) {
            portToNeighbours.get(SRC_PORT).put(neighbourPort, 1);
        }
}
```

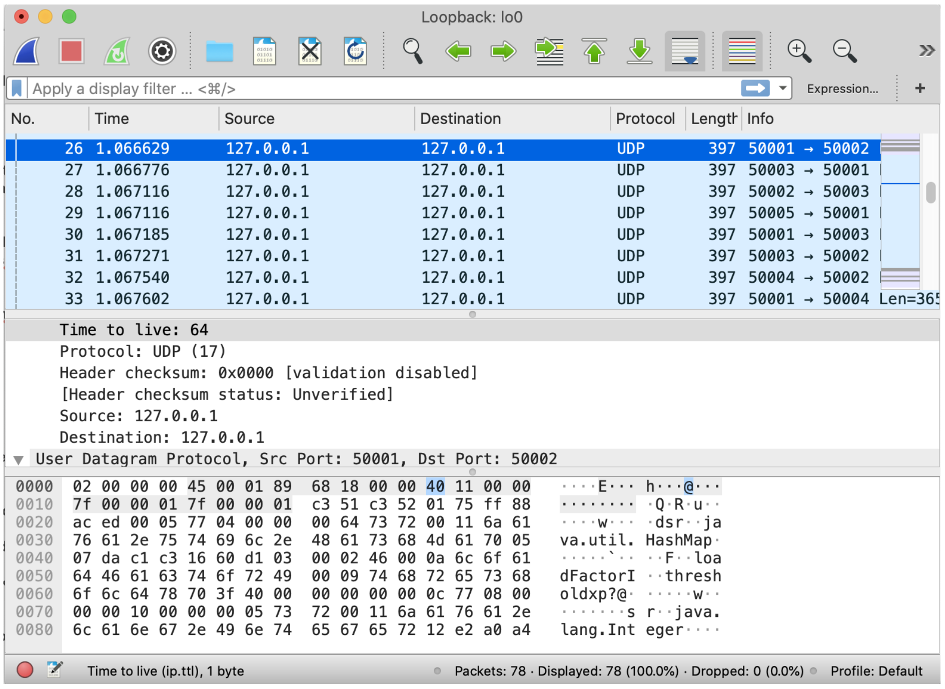
Listing 5: Information about each router in the network, and each router's neighbours is stored.

Information regarding each port about its neighbours and the distances to those neighbours is stored by each router in the variable 'portToNeighbours'. For now each router only has information regarding itself, which gets stored in 'portToNeighbours'.

Packets containing information each router has about itself and any other routers in the network are then sent to all other routers in the network; see listing 6 and figure 5

```
for (int port : portToNeighbours.keySet()) {
        if (port != SRC_PORT) { // don't want to send to self
                sendLinkState(portToNeighbours, port);
        }
}
```

Listing 6: Information router has about itself and other routers in the network are sent to all other routers in the network.



```
No.  Time      Source     Destination  Protocol Length Info
  26 1.066629  127.0.0.1  127.0.0.1    UDP        397  50001 → 50002
  27 1.066776  127.0.0.1  127.0.0.1    UDP        397  50003 → 50001
  28 1.067116  127.0.0.1  127.0.0.1    UDP        397  50002 → 50003
  29 1.067116  127.0.0.1  127.0.0.1    UDP        397  50005 → 50001
  30 1.067185  127.0.0.1  127.0.0.1    UDP        397  50001 → 50003
  31 1.067271  127.0.0.1  127.0.0.1    UDP        397  50003 → 50002
  32 1.067540  127.0.0.1  127.0.0.1    UDP        397  50004 → 50002
  33 1.067602  127.0.0.1  127.0.0.1    UDP        397  50001 → 50004 Len=365

    Time to live: 64
    Protocol: UDP (17)
    Header checksum: 0x0000 [validation disabled]
    [Header checksum status: Unverified]
    Source: 127.0.0.1
    Destination: 127.0.0.1
▼ User Datagram Protocol, Src Port: 50001, Dst Port: 50002

0000  02 00 00 00 45 00 01 89  68 18 00 00 40 11 00 00   ····E··· h···@···
0010  7f 00 00 01 7f 00 00 01  c3 51 c3 52 01 75 ff 88   ········ ·Q·R·u··
0020  ac ed 00 05 77 04 00 00  00 64 73 72 00 11 6a 61   ····w··· ·dsr··ja
0030  76 61 2e 75 74 69 6c 2e  48 61 73 68 4d 61 70 05   va.util. HashMap·
0040  07 da c1 c3 16 60 d1 03  00 02 46 00 0a 6c 6f 61   ·····`·· ··F··loa
0050  64 46 61 63 74 6f 72 49  00 09 74 68 72 65 73 68   dFactorI ··thresh
0060  6f 6c 64 78 70 3f 40 00  00 00 00 00 0c 77 08 00   oldxp?@· ·····w··
0070  00 00 10 00 00 00 05 73  72 00 11 6a 61 76 61 2e   ·······s r··java.
0080  6c 61 6e 67 2e 49 6e 74  65 67 65 72 12 e2 a0 a4   lang.Int eger····

Time to live (ip.ttl), 1 byte     Packets: 78 · Displayed: 78 (100.0%) · Dropped: 0 (0.0%)     Profile: Default
```

Figure 5: Shows various routers sending their information to other routers in the system, namely the router at port 50001 sending its information to the router at port 50002. This information consists of a Java HashMap as can be seen in the bottom right corner.

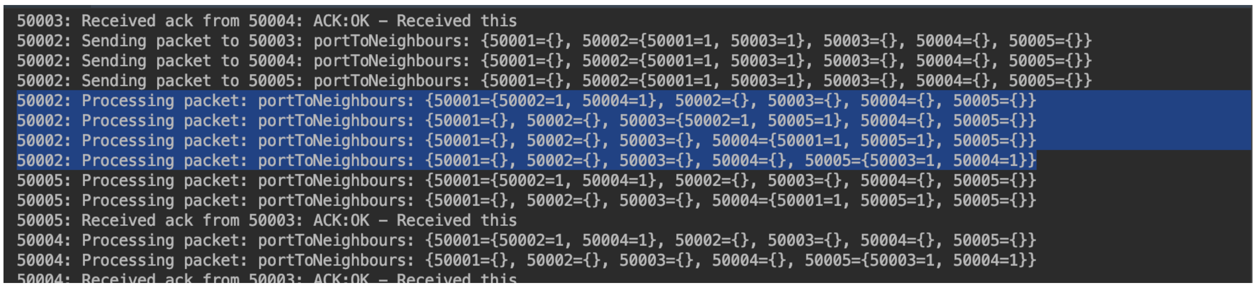
This information is received by each router and gets stored; see listing 7.

```java
public void processLinkState(DatagramPacket packet,LinkStateContent content) {

    // info received
    HashMap<Integer, HashMap<Integer, Integer>> newPortToNeighbours =
        content.getNeighbours();

    // if there is any info
    if (newPortToNeighbours != null) {

        // for each port info was received for
        for (Map.Entry<Integer, HashMap<Integer, Integer>> port :
                        newPortToNeighbours.entrySet()) {

            // if don't already have info and there is info for that port
            if (portToNeighbours.get(port.getKey()).isEmpty() &&
                        !port.getValue().isEmpty()) {

                // add info about neighbours for this port
                for (int newNeighbour : port.getValue().keySet()) {
                    portToNeighbours.get(port.getKey()).put(newNeighbour, 1);
                }
            }
        }
    }
}
```

Listing 7: Information is received and processed by router.

The router checks each port for any information that was received, ignoring any information that the router already has. This information is then added to portToNeighbours for the relevant port.

Take as an example the router at port 50002 receiving information about the network; see figure 6. The router receives packets from 50001 containing its neighbours 50002 and 50004 and the distances of one hop to them. It then receives information from 50003, 50004, and 50005 respectively about their neighbours. Note that all of these packets were sent before any routers had any information about routers other than their neighbours. However in that case, ports other than their neighbours would have information where here they are empty.



```
50003: Received ack from 50004: ACK:OK - Received this
50002: Sending packet to 50003: portToNeighbours: {50001={}, 50002={50001=1, 50003=1}, 50003={}, 50004={}, 50005={}}
50002: Sending packet to 50004: portToNeighbours: {50001={}, 50002={50001=1, 50003=1}, 50003={}, 50004={}, 50005={}}
50002: Sending packet to 50005: portToNeighbours: {50001={}, 50002={50001=1, 50003=1}, 50003={}, 50004={}, 50005={}}
50002: Processing packet: portToNeighbours: {50001={50002=1, 50004=1}, 50002={}, 50003={}, 50004={}, 50005={}}
50002: Processing packet: portToNeighbours: {50001={}, 50002={}, 50003={50002=1, 50005=1}, 50004={}, 50005={}}
50002: Processing packet: portToNeighbours: {50001={}, 50002={}, 50003={}, 50004={50001=1, 50005=1}, 50005={}}
50002: Processing packet: portToNeighbours: {50001={}, 50002={}, 50003={}, 50004={}, 50005={50003=1, 50004=1}}
50005: Processing packet: portToNeighbours: {50001={50002=1, 50004=1}, 50002={}, 50003={}, 50004={}, 50005={}}
50005: Processing packet: portToNeighbours: {50001={}, 50002={}, 50003={}, 50004={50001=1, 50005=1}, 50005={}}
50005: Received ack from 50003: ACK:OK - Received this
50004: Processing packet: portToNeighbours: {50001={50002=1, 50004=1}, 50002={}, 50003={}, 50004={}, 50005={}}
50004: Processing packet: portToNeighbours: {50001={}, 50002={}, 50003={}, 50004={}, 50005={50003=1, 50004=1}}
50004: Received ack from 50003: ACK:OK - Received this
```

Figure 6: Shows the router at port 50002 receiving information from the other routers in the network.

Once a router has information about all of the routers in the network, it can construct a routing table using Dijkstra's Algorithm; see listing 8

```java
ArrayList<Route> tentative = new ArrayList<>();
```

```
ArrayList<Route> permanent = new ArrayList<>();

// Step 1
Route n = new Route(SRC_PORT, 0, -1);

// Step 2
permanent.add(n);

// while not every node is permanent
// Step 6
while (permanent.size() < portToNeighbours.size()) {

    // format n's neighbours
    // Step 3
    for (Map.Entry<Integer, Integer> r :
        portToNeighbours.get(n.port).entrySet()) {

        // Step 4
        tentative.add(new Route(r.getKey(), r.getValue() + n.dist, n.port));
    }

    // Step 5a
    Routes.removeDups(tentative, permanent);

    // Step 5b
    n = Routes.getShortest(tentative);
    tentative.remove(n);
    permanent.add(n);
}

routingTable = Routes.toMap(permanent, SRC_PORT);
```

Listing 8: Implementation of Dijkstra's Algorithm, with steps commented.

Note that this implementation uses an object 'Route' for simplicity. This object simply has a port, a neighbour, and a distance to that neighbour. This implementation is the application of what was explained previously, with comments noting each step.

3.3 Changes in Topology

One of the purposes of using a dynamic routing system is so that changes can occur without breaking the system. These changes would be nodes being added and removed from the system, as well as links between nodes being broken. This wasn't completed by the deadline, however this section will still explain how it would have been implemented.

If a node was added to the network, it would broadcast its information to the rest of the network, and they would return the information they have. With this information, the new node could, using Dijkstra's algorithm, form its own routing table. The other nodes would also be able to update their routing tables to include the new node; by adding new routes with that node as the destination, as well as altering any existing routes if the new node allowed for shorter routes.

If a node was removed from the network, either that node would broadcast that it was leaving, allowing the other routers to change their routing tables accordingly, or nodes attempting to send packets to that node wouldn't receive any acknowledgements. These node would eventually realise that either the link between the two nodes has broken, or the node is no longer there. In each case, as each node's routing table contains not just the next node to which a packet should be sent, but the entire route the packet is to take, each node will be able to alter their tables in accordance with their knowledge of the network. If the node was in fact removed, it would be removed from the routing tables as a destination. Otherwise, new paths to that node would be found.

3.4 Sending Packets

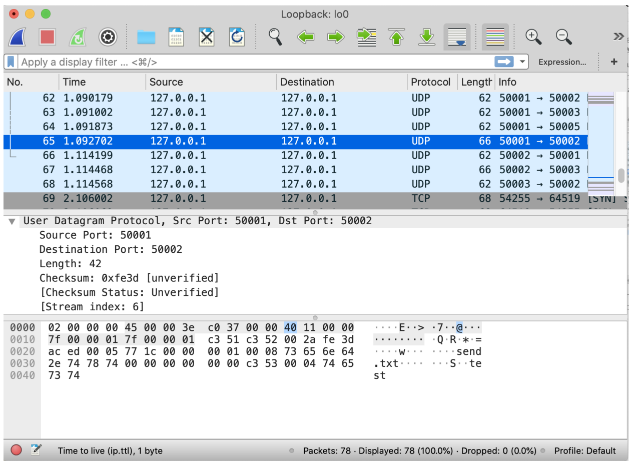
Sending packets is relatively simple once the routing tables are constructed. Each packet contains the desired destination port, and each node reads this port, and finds the next port to which it maps in that node's routing table; see listing 9.

```
public void processFileFunc(DatagramPacket packet, FileFuncContent content) {
        switch (content.getFunc()) {
            default:
                if (routingTable.isEmpty()) { //if routing table not yet made
                    System.out.println(SRC_PORT +
                        ": Adding packet " +
                        content.toString() +
                        " to queue");
                    queue.add(packet);
                } else { // if routing table is made
                    int dstPort = content.getDstPort();
                    if (dstPort == SRC_PORT) { // if this is dst
                        System.out.println(SRC_PORT + ": Received packet!");
                        /*
                        Process packet
                         */
                    } else { // else forward
                        int  nextPort = routingTable.get(dstPort)
                                .getValue() // get route
                                .get(0); // get next port in route
                        sendFileFunc(content, nextPort);
                    }
                }
                break;
        }
}
```

Listing 9: Node receives a packet, and does one of three things; adds it to the queue if the routing table is not yet made; processes the packet if that node is the desired destination; or forwards it the appropriate port as described by its routing table.

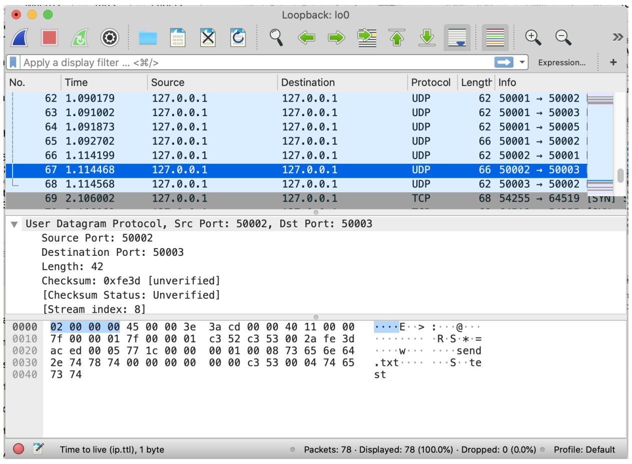
In this case, the map contains the entire route that packet should take to reach the desired destination, thus each node need only take the first element in this list; that being the next port in the route.
If a packet is received before the routing table is constructed, it is added to the queue and processed once the routing table is constructed.
If this destination port is the node's source port, the packet has reached its destination, and the node may now process that packet.
Otherwise, the packet is forwarded to the appropriate next port.

Take as an example the router at port 50001 wants to send a packet to the router at port 50003. To do this, it must either send the packet via 50002 or 50004 and 50005. As the route via 50002 is shorter, the router's routing table should map the destination port 50003 to the next port 50002; see figure 7a. This router's routing table should then map the destination port 50003 to the next port 50003; see figure 7b. The router at port 50003 then sends an acknowledgement to 50002 as it has received the packet; see figure 7c.



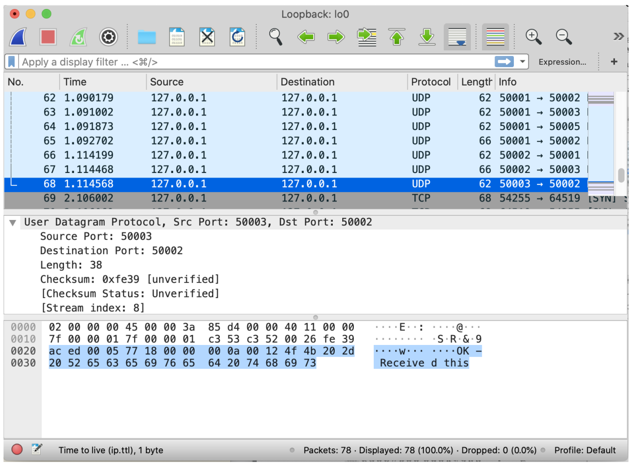
```
Loopback: lo0
No.   Time        Source      Destination  Protocol  Length  Info
  62  1.090179    127.0.0.1   127.0.0.1    UDP       62  50001 → 50002
  63  1.091002    127.0.0.1   127.0.0.1    UDP       62  50001 → 50003
  64  1.091873    127.0.0.1   127.0.0.1    UDP       62  50001 → 50005
  65  1.092702    127.0.0.1   127.0.0.1    UDP       66  50001 → 50002
  66  1.114199    127.0.0.1   127.0.0.1    UDP       62  50002 → 50001
  67  1.114468    127.0.0.1   127.0.0.1    UDP       66  50002 → 50003
  68  1.114568    127.0.0.1   127.0.0.1    UDP       62  50003 → 50002
  69  2.106002    127.0.0.1   127.0.0.1    TCP       68  54255 → 64519 [SYN]

▼ User Datagram Protocol, Src Port: 50001, Dst Port: 50002
    Source Port: 50001
    Destination Port: 50002
    Length: 42
    Checksum: 0xfe3d [unverified]
    [Checksum Status: Unverified]
    [Stream index: 6]

0000  02 00 00 00 45 00 00 3e  c0 37 00 00 40 11 00 00   ····E··> ·7··@···
0010  7f 00 00 01 7f 00 00 01  c3 51 c3 52 00 2a fe 3d   ········ ·Q·R·*·=
0020  ac ed 00 05 77 1c 00 00  00 01 00 08 73 65 6e 64   ····w··· ····send
0030  2e 74 78 74 00 00 00 00  00 00 c3 53 00 04 74 65   .txt···· ···S··te
0040  73 74                                              st

Time to live (ip.ttl), 1 byte    Packets: 78 · Displayed: 78 (100.0%) · Dropped: 0 (0.0%)    Profile: Default
```

(a) Shows the router at port 50001 sending a packet to the router at port 50002, which is the next step towards the destination router at port 50003.



```
Loopback: lo0
No.   Time        Source      Destination  Protocol  Length  Info
  62  1.090179    127.0.0.1   127.0.0.1    UDP       62  50001 → 50002
  63  1.091002    127.0.0.1   127.0.0.1    UDP       62  50001 → 50003
  64  1.091873    127.0.0.1   127.0.0.1    UDP       62  50001 → 50005
  65  1.092702    127.0.0.1   127.0.0.1    UDP       66  50001 → 50002
  66  1.114199    127.0.0.1   127.0.0.1    UDP       62  50002 → 50001
  67  1.114468    127.0.0.1   127.0.0.1    UDP       66  50002 → 50003
  68  1.114568    127.0.0.1   127.0.0.1    UDP       62  50003 → 50002
  69  2.106002    127.0.0.1   127.0.0.1    TCP       68  54255 → 64519 [SYN]

▼ User Datagram Protocol, Src Port: 50002, Dst Port: 50003
    Source Port: 50002
    Destination Port: 50003
    Length: 42
    Checksum: 0xfe3d [unverified]
    [Checksum Status: Unverified]
    [Stream index: 8]

0000  02 00 00 00 45 00 00 3e  3a cd 00 00 40 11 00 00   ····E··> :···@···
0010  7f 00 00 01 7f 00 00 01  c3 52 c3 53 00 2a fe 3d   ········ ·R·S·*·=
0020  ac ed 00 05 77 1c 00 00  00 01 00 08 73 65 6e 64   ····w··· ····send
0030  2e 74 78 74 00 00 00 00  00 00 c3 53 00 04 74 65   .txt···· ···S··te
0040  73 74                                              st

Time to live (ip.ttl), 1 byte    Packets: 78 · Displayed: 78 (100.0%) · Dropped: 0 (0.0%)    Profile: Default
```

(b) Shows the router at port 50002, having received the packet from 50001 with destination 50003, forwarding the packet to the router at port 50003.



```
Loopback: lo0
No.   Time        Source      Destination  Protocol  Length  Info
  62  1.090179    127.0.0.1   127.0.0.1    UDP       62  50001 → 50002
  63  1.091002    127.0.0.1   127.0.0.1    UDP       62  50001 → 50003
  64  1.091873    127.0.0.1   127.0.0.1    UDP       62  50001 → 50005
  65  1.092702    127.0.0.1   127.0.0.1    UDP       66  50001 → 50002
  66  1.114199    127.0.0.1   127.0.0.1    UDP       62  50002 → 50001
  67  1.114468    127.0.0.1   127.0.0.1    UDP       66  50002 → 50003
  68  1.114568    127.0.0.1   127.0.0.1    UDP       62  50003 → 50002
  69  2.106002    127.0.0.1   127.0.0.1    TCP       68  54255 → 64519 [SYN]

▼ User Datagram Protocol, Src Port: 50003, Dst Port: 50002
    Source Port: 50003
    Destination Port: 50002
    Length: 38
    Checksum: 0xfe39 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 8]

0000  02 00 00 00 45 00 00 3a  85 d4 00 00 40 11 00 00   ····E··: ····@···
0010  7f 00 00 01 7f 00 00 01  c3 53 c3 52 00 26 fe 39   ········ ·S·R·&·9
0020  ac ed 00 05 77 18 00 00  00 0a 00 12 4f 4b 20 2d   ····w··· ····OK -
0030  20 52 65 63 65 69 76 65  64 20 74 68 69 73          Receive d this

Time to live (ip.ttl), 1 byte    Packets: 78 · Displayed: 78 (100.0%) · Dropped: 0 (0.0%)    Profile: Default
```

(c) Shows the router at port 50003, having received the packet from 50002, sends an acknowledgement to the router at port 50002 to say that it has successfully received the packet.

4 Choices and Advantages & Disadvantages

4.1 Routing

A dynamic routing protocol rather than a hard-coded set of routes was chosen for this project.

Advantages:

- Allows for addition and removal of nodes: if a node is added or removed from the system, a dynamically constructed routing protocol can calculated the appropriate paths for this change, unlike a static system, where packets would simply get stuck, either not knowing where to go, or trying to go to a node that no longer exists.
- Allows for link breakage: if a link between two routers is broken, the next best routes can be calculated in a dynamic system, whereas in a static system, any packets attempting to be sent between these two routers would get stuck.
- Scalable: in a static system, each route has to be hard-coded by hand, which scales poorly to larger systems. Thus a dynamic system that, once the algorithm is implemented, calculates the routes

independently scales better.

Disadvantages:

- Complexity: dynamic routing protocols are more complicated to implement than hard-coded routes. Thus in a theoretical system where you knew that no routers would be added or removed, and no links would break, and there were relatively few nodes, static routing tables might be preferable. Of course this could never occur in reality, and so dynamic routing protocols are always recommended.

4.2 Dynamic Routing Protocol

The Link State routing protocol rather than the Distance Vector routing protocol was chosen for this project.

Advantages:

- No 'Count to Infinity' problem: as explained previously, Distance Vector routing can run into this problem, where Link State routing does not. While this can be remedied to some extent, no solution can deal with arbitrary topology cycles.
- No constant loops: Distance Vector routing has a constant loop so that each router can send updates after a period of time, whereas Link State routing only needs to send updates when a change occurs.

Disadvantages:

- More bandwidth required: the Link State packets are larger than Distance Vector packets, resulting in more bandwidth required. In this system this isn't a big issue, but it would be more so in reality.
- Flooding: Packets are sent to all other routers, resulting in a large number of large packets being received by each router, which can cause flooding.
- Greater memory requirements: each router has to know about the entire system, unlike in Distance Vector routing, which scales poorly with regards to memory.

5 Summary

This report has presented my attempt at a system which allows for OpenFlow communication between nodes. It highlights the essential components of my solution and demonstrates the execution of the solution.

6 Reflection

I enjoyed how this project was built on the back of the previous project, as it allowed me to focus more so on the implementation of OpenFlow specifically, with little to no set-up beyond patching up any loose ends from the previous project. As well as that, I found that having to choose and implement a routing protocol allowed me to fully understand how both Link State and Distance Vector Routing worked and differed from one another in much more depth than simply from reading about them. This assignment took me about 15 hours to complete.