CS2031 Telecommunications: Assignment 2

LEE CAMPBELL

Introduction:

The given task was to design a openflow-inspired routing system. In such a system, end-users are associated with routers and send out packets addressed to other end-users. Each router has a number of ports, which can communicate with a subset of other routers and end-user devices and route packets based on rules which associate destinations with incoming and outgoing ports. However, each router is considered 'dumb' and doesn't initially know these rules so in turn when the router receives a packet going to a destination it doesn't know how to get to it must contact a central controller which. pushes the routing information for the given destination to each router that needs it.

Adapted from assignment specification: TCD Blackboard/Telecommunications II

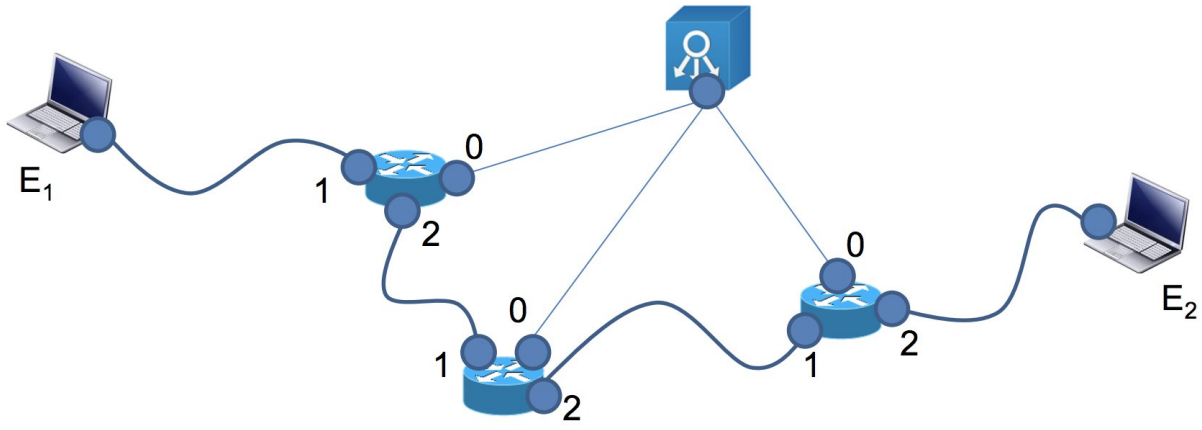


Figure 1 source: TCD Blackboard/Telecommunications II

Design and Implementation:

SETUP CLASS:

In order to simulate a network setup such as the one illustrated above in *Figure 1,* I designed a setup class which initialises a series of routers, clients (end-points) and a central controller.

- The controller at this point is simply initialised with a local address as each router and client when they are initialised contacts the controller to give it information about it's existence such as it's connected ports.
- Each client is initialised with a local port, a destination port (i.e. the client it wants to talk to) and the router to which it is connected to.
- Each router is initialised with a local port and and an array of remote ports to which it should connect to.

PACKET HEADER:

In assignment 1 , one of the most critical elements in the design of the protocol was the formation of the packet header which contains various snippets of information that allowed a packet to be transported correctly from A to B through a gateway. In this assignment, I have taken that structure which worked really well previously and generalised each of the 4 byte slots so that they can contain different snippets of information depending on who is sending them.

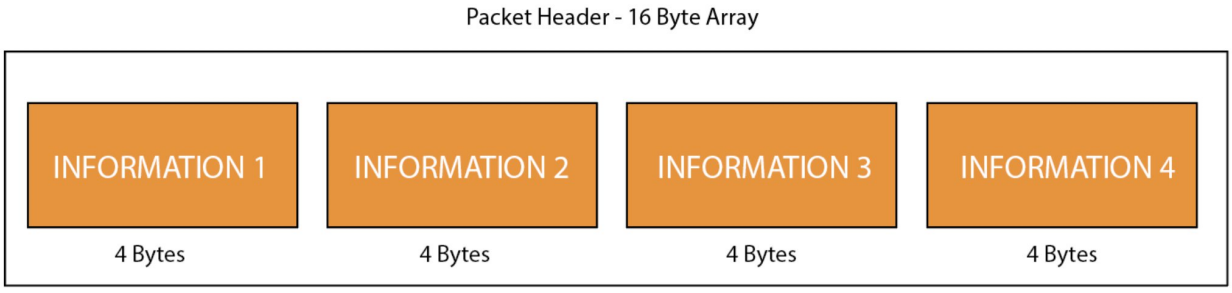


Figure 2

Examples:

Header - Client to Client

When a client is sending a packet to another client the header structure follows that of a standard header implementation.

- Information 1: This contains the sequence no of the packet being sent.
- Information 2: This contains the local port of client sending the packet.
- Information 3: This contains the destination port (i.e. receiving client port) of packet being sent.
- Information 4: Not used in this case.

Header - Router to Controller

When a client is sending a packet to a controller the header structure contains a slightly altered information as this packet serves the sole purpose of providing the controller with the

necessary information to have an overview of the entire network and build a routing table.

- Information 1: This contains the out port on the router i.e. the port which is connected to another router or client.
- Information 2: This contains the local port of router sending the packet.
- Information 3: This contains the receiving port of router or client which the local port is connected to.
- Information 4: Not used in this case.

Snippet: Header Class with header construction method.

```
public static byte[] construct(byte[] header, int info1, int info2,
                               int info3, int info4)
    {

        ByteBuffer tmpBuf = ByteBuffer.wrap(header);

        tmpBuf.putInt(INFO_1_INDEX, info1);
        tmpBuf.putInt(INFO_2_INDEX, info2);
        tmpBuf.putInt(INFO_3_INDEX, info3);
        tmpBuf.putInt(INFO_4_INDEX, info4);
        header = tmpBuf.array();

        return header;
    }
```

Advantage: Creating a header class with functionality to construct and de-construct a header is an efficient way to allow the client, router and controller classes to construct and re-construct headers that contain useful information, without code duplication.

CONTROLLER

The controller constructor creates a controller as follows:

1. Initialises a socket at the local port of the controller.
2. Initialises a listener on the local port of the controller and calls the .start(); function.

Routing Table - Data Structure

Figure 3

Dest	Src	Router	In	Out
E2	E1	R1	1	2
		R2	1	2
		R3	1	2

An essential part of the controller class is to maintain a routing table based on the controllers overview of the network, this allows the controller to provide each router with routing information for packets they receive. The routing table in figure 2 illustrates a routing table pertaining to the network

shown in figure 1. In the context on the controller class this table needed to be able to take a query for routing information based on a destination & source. In order to achieve this I decided to use a Hash-map data structure in which the key is of type String and the value is of type Routers.

```
HashMap<String, Routers> map = new HashMap<String, Routers>();

    private class Routers
    {
        int router1[] = new int[2];
        int router2[] = new int[2];
        int router3[] = new int[2];

        public Routers(int router1[], int router2[], int router3[])
        {
            this.router1 = router1;
            this.router2 = router2;
            this.router3 = router3;
        }

    }
```

The string key contains a string of the Source and Destination e.g. “40000, 50000” of two clients. The value element contains an instance of the internal class Routers. The instance of Routers contains three arrays router1, router2, and router3 each with two elements; an out port for the said router and the receive port of the router it is connected to. This in turn allows the Controller to send the routing information for a particular Source and Destination to each router for e.g. Router 1 would receive the router1 array.

Advantage: An efficient, asymptotically low-cost way to store and retrieve routing information.

Once the controller is constructed, onReceipt(); waits for a packet to be received:

1. When a packet is received, it's type is checked i.e. if it is from a:
    a. **Client:** If a packet is identified as a client packet, this means a client has been initialised and is contacting the controller to tell it, it exists. It's port address is stored in an array of client port numbers and the port of it's connected router is stored in an array of connectedRouters.
    b. **Router:** If a packet is identified as a router packet, this means a router has been initialised and is contacting the controller to tell it, it exists. It's out port and the receiving port of the router or client that it is connected to are both stored in an array. The buildPath() function is then called to build the path of routers a packet needs to take from either client1 to client 2 or from client 2 to client 1 depending on the direction set. Once this network is returned in the form of an instance of the Routers class this is then put into the hash-map as the value along with the destination & source string key.
    c. **Router requesting a routing table:** If a packet is identified as router request packet, the source client address and destination client address are unpacked from

the header and appended into a string for e.g. “40000, 50000” and then passed into a get key request from the hash-map which should return a value. In this case the value is an instance of the Router class. Using the instance of router class we can then forward to each router; the relevant out and receive ports needed to get the packet from client 1 to client 2.

```
Routers routingInfo = this.map.get(destinationSource);

//Send routing info for packet to Router 1
out = routingInfo.router1[0];
in = routingInfo.router1[1];
constructSendPacket(out, in, router1Port);

//Send routing info for packet to Router 2
out = routingInfo.router2[0];
in = routingInfo.router2[1];
constructSendPacket(out, in, router2Port);

//Send routing info for packet to Router 3
out = routingInfo.router3[0];
in = routingInfo.router3[1];
constructSendPacket(out, in, router3Port);
```

buildPath() function

This function finds the path that a packet needs to take in a particular direction e.g. Client 1 to Client 2 using the controllers overview of the network. It returns an instance of class Routers which forms the value part of a hash-map input in the program.

Disadvantage: Ideally, had I had more time I would have liked to implement a more elaborate and dynamic method of finding the shortest path such as integrating Dijkstra's algorithm.

It creates this instance of the class Routers by:

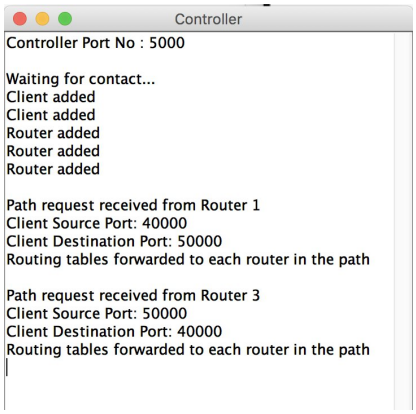
a. Finding the router in the array of routers connected to the controller, which matches the router that the client sending the packet is connected to. It then sets router1 within the Routers class to contain the Out Port and Next Receive Port that correspond with the matching array element.

b. Next it finds the router in the array of routers connected to the controller, which contains the port value of the Next Receive Port from router 1. Once this is found, it then sets router 2 within the Routers class to contain Out Port and Next Receive Port that correspond with the matching array element.

c. Finally it finds the router in the client routers connected to the controller, which contains the port value of the Next Receive Port from router 2. Once this is found, it then sets router 3 within the Routers class to contain Out Port and Next Receive Port that correspond with the matching array element.

d. The order in which the router values are passed into the Routers class constructor to create an instance of Routers changes depending on which Client a packet is coming from.

```
if (clientNo == client1ToClient2)
{
    clientToClient = new Routers(router1, router2, router3);
}
else if (client2ToClient1)
{
    clientToClient = new Routers(router3, router2, router1);
}
```



ROUTER:

Each router essentially acts as a gateway between two or more clients, however the routers are considered 'dumb' as in this open-flow inspired approach each router doesn't initially know where to send a packet it receives, it must contact a central controller to obtain a routing table for a given packet.

The router constructor creates a client as follows:

1. It initialises 3 sockets on the router with a local port address and connects each of those sockets/ports to a remote socket as specified by the array of remote ports passed in from the Setup class. It then creates a new listener at each of those sockets to listen for incoming packets from connected remote port sockets.

```
for(int i=0;i<socketCount;i++)
{
port[i] = new DatagramSocket(localPort+i);
port[i].connect(new InetSocketAddress("localhost", remotePorts[i]));
new Listener(port[i]).start();
}
```

2. Next the router sends two packets to the controller using the sendRouterInfo(); function, containing information regarding it's ports and the remote ports which receive in-

formation from those ports. This gives the controller an overview of the network and allows it to build routing paths between clients.

Advantage: Creating a series of sockets and connecting them to set of remote ports in a structured manner as well as creating a listener for each makes routing and setting up routing tables simpler and more efficient.

Once a client is constructed, onReceipt(); waits for a packet to be received:

1. When a packet is received, it's type is checked i.e. if it is from a:
    a. **Controller:** If a packet is identified as a controller packet, this means the controller is sending on routing information to the router so it can forward the packet to correct router/client. Much like in the Controller, a hash-map data structure is used to store routing information on each router once it is received from the controller.. The key in this case is simply a integer containing the destination client port of the packet the router is trying to forward and the value is of the type portParing a local class which essentially stores the local out port and receiving port of the router that the local out is socket is connected to.

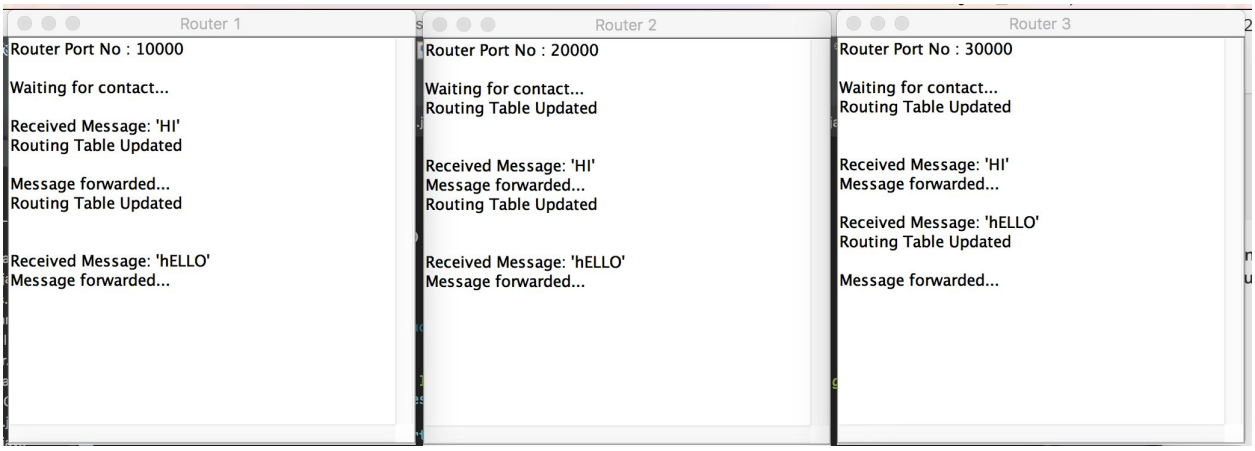
```
HashMap<Integer, portPairing> rules = new HashMap<Integer, portPairing>();

DatagramSocket port[];private class portPairing
{
int localOutPort;
int nextReceivePort;

public portPairing(int localOutPort, int nextRecievePort)
{
this.localOutPort = localOutPort;
this.nextReceivePort = nextRecievePort;
}

}

portPairing ports = new portPairing(out, in);
rules.put(clientDST, ports);
```

    b. **Router/Client:** If a packet is identified as a message packet coming from a client or another router, the router will then check it's rules for routing information regarding a packet travelling to a specific client destination port. If the hash-map of rules contains the key it calls the forward(); function to forward the packet onto either the next router or the destination client if it is connected to it directly. If the key is not contained within the hash-map the router sends a packet to the controller asking for rules on how to forward the message packet it received from a client to a destination client, as described above.

forward(); fucntion

The forward function extracts the value for local out port and the receive port of the con-

nected router/client from the hash-map using the key (i.e. the destination client port integer). It then sets the packet's InetSocket to receive port address and uses the .send(); method on the local out port to forward the message packet to next router or destination client on the network.



CLIENT

Each client initialised functions as an endpoint within the network illustrated in figure 1. A client can and send and receive messages over the network.

The client constructor creates a client as follows:

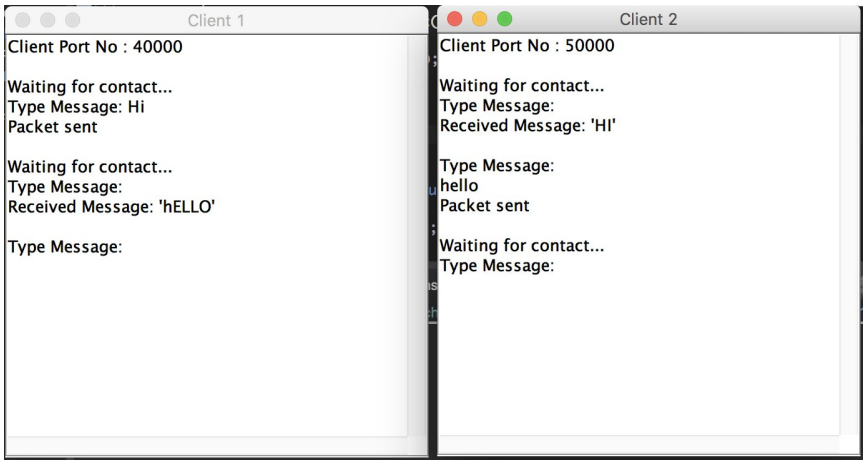
1. Initialises a socket at the local port of the client..
2. Initialises a listener on the local port of the client.
3. Initialises the client with a connecting router port and a destination client port.
4. Creates a new thread, which allows for two clients to run and take input concurrently.
5. Runs the sendClientToController(); function which creates and sends a packet to the controller containing information about itself and it's connected ports so that controller can add it to it's overview of the network.

Advantage: Using multi-threading it is possible to have both clients run simultaneously and take input.

1. This calls the constructSendPacket(); function which begins creates and puts together each of the elements needed for a valid packet. It first calls the Header.construct (); function as seen in the code snippet above which provides the header element. It then takes in the payload (i.e. the message/data) from the user and appends it to the header which in then used to create a new packet. The packet is then sent via the socket.send(); method.

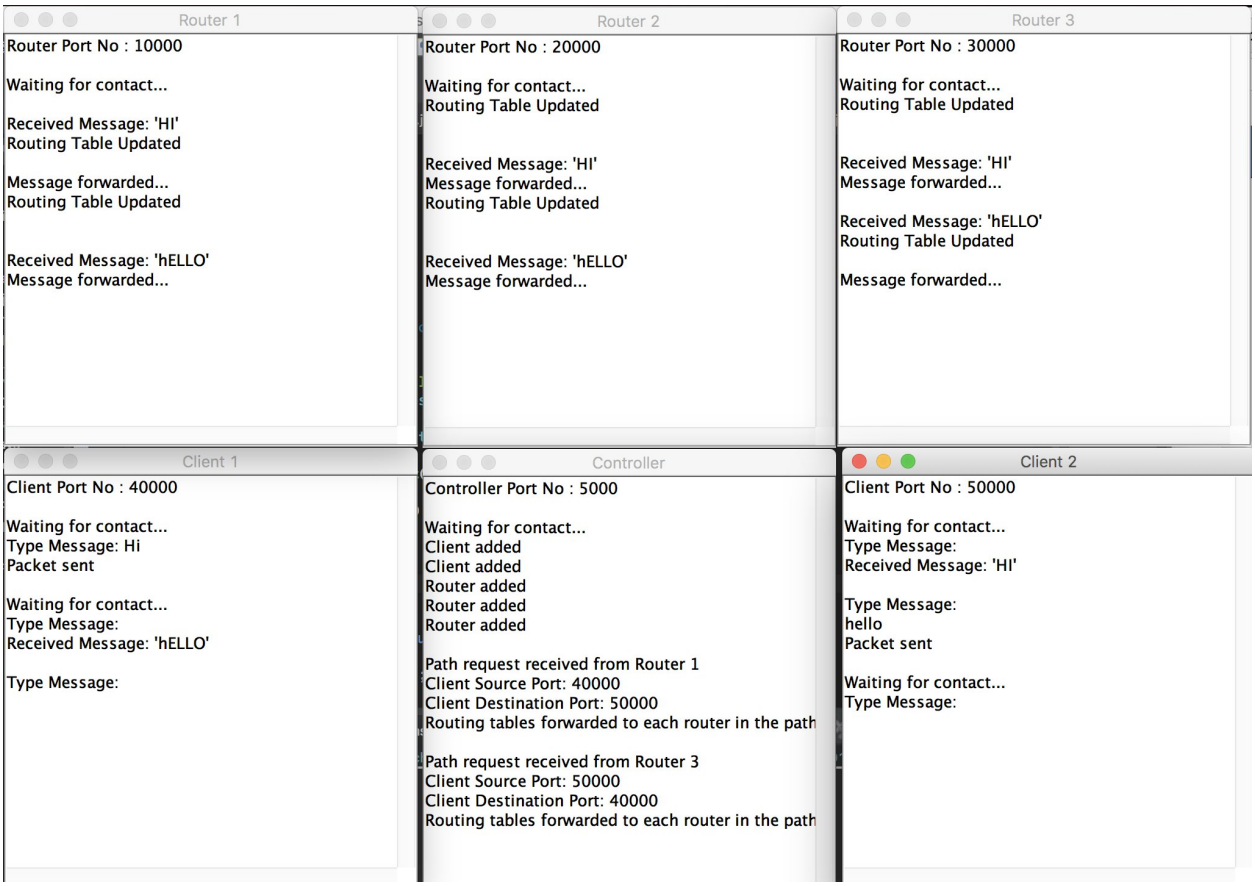
Finally the onReceipt(); function is called once a packet is received

1. Once a packet is received the onReceipt(); function unpacks the payload from the packet and prints it's content to a string in the said client terminal. See below.



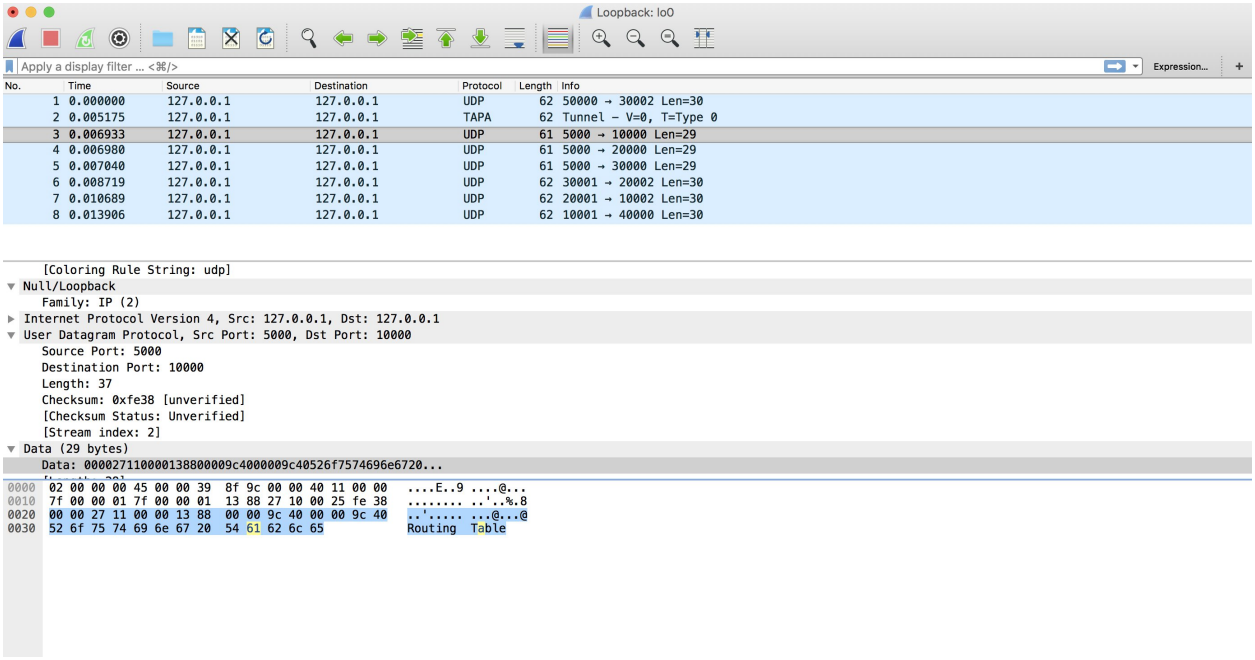
OVERVIEW

In the below figure we can see a sample overview of the network through a series of terminals initialised by the Setup class.



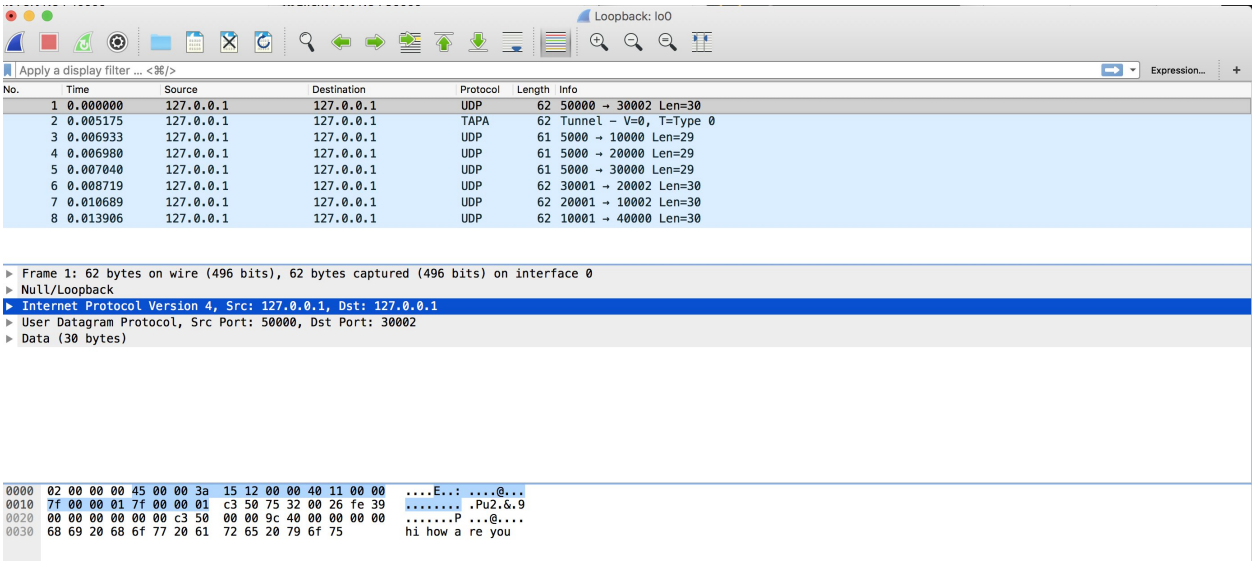
Controller sending routing information to router 1:

Using wire-shark I was able to capture the packets being sent over the simulated ports on the loopback adaptor. As you can see in the figure below, the source port was 5000 (i.e. The controller), the destination was 10000 (i.e. Router 1) and the payload contained the string “Routing table” this tells us that the controller was updating router one with routing information.



Client 2 sending a message to Client 1 via Router 3:

Again using wire-shark I was able identify a packet message “hi how are you” being sent from Client to at port 50000 to a receiving port on router 3, 30002 presumably being forwarded across the network to client 1.



Evaluation

Throughout this report, in explaining my implementation I outlined various **advantages** and **disadvantages** associated with it - if I were to do this project again I would certainly like to work on improving/alleviating the disadvantages which I didn't have time to do this time around.

On a broader level; I initially found elements of the assignment challenging, especially in terms of getting to grips with multi-threading and using certain data structures for the first time in order to implement the project. This undoubtedly thought me a lot but unfortunately took up a lot of my time for this project, meaning that I couldn't implement it more dynamically i.e. where you could add in additional routers or clients to the number specified. If I were to this project again this would be one of my main focuses along with implementing a more efficient and dynamic shortest path finding menthod such as Dijkstra's algorithm.

Finally, as I mentioned in assignment 1, if I were to do this assignment again I would certainly like to try implement it in Pyhton which from my research seems very useful and clean for such an application.

Estimated hours spent on the assignment: 34 Hours.

eee