CS2031 Telecommunications II
Assignment 2

Name: *Owen Burke*
ID: *15316452*

This report describes the design and implementation of the protocol summarised in the assignment specification provided. This implementation allows end-users (clients) to send packets to each other via a system of routers. A controller is also used in order to control the sending of packets and ensure they arrive to the appropriate end-user. If a router receives a packet from an end-user that it has not received before (i.e. addressed to a new client), it will not have a rule to follow regarding the forwarding of that packet, so it must contact the controller. The controller maintains a table that contains the information required along the router network to send packets correctly (the table contains the start and end-point client ports and has a corresponding table for each router, which contains the port to send the packet out of and the port on the next point that the packet should be sent to). Whenever a router receives a packet addressed to this client again, it will have a rule to follow and, therefore, will have no need to contact the controller. Naturally, each router must allow maintain a table, which contains the destination client port and two corresponding ports (1.The port to send the packet out of, 2. The port on the next point that the packet should be sent to).

Step One Implementation:

The first step of the implementation began by creating a client, router and controller class as these will be the foundation of the implementation of the protocol. A setup class was also made that would create all the relevant routers, clients and the controller with each class having certain parameters in their constructors, i.e. name and associated ports.

Each client object is created with a source port (this being the port that packets are sent and received from), a tied router port (the port on the router which is tied to this end-user) and the port on the destination client. In this implementation there are only two users, so the destination client ports are already known, however other clients and routers could be added. The implementation would have to be changed slightly (make new port arrays, etc.) as the system is static (dynamic behaviour was not a requirement) and the implementation for part 2 would have become drastically more complicated with the approach I took (as I explain later).

Each client object will run on its own thread in the run method, which allows the clients to easily send and receive packets (i.e. on receipt is run, even if the “String to send” prompt in run is showing).

When a string is entered, that string is added to the payload along with the client source port and the port on the destination client. This is needed by the routers in order to check if they have a rule and send the information to the controller, if necessary.

Each router object has a local port which is used in the constructor to allocate all the other ports on the router which, in turn, is used to create sockets at all ports on the router which are all listening concurrently, stored in `DatagramSocket[] sockets`. In order to create sockets that listen simultaneously, the Node abstract class had to be modified, so now, it essentially sets up a listener on each socket.

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

The routers also take in an array of ports, `int[] remotePorts`, which contains all the ports that are immediately associated/next to the router. This array will always contain the controller port, as all the routers are connected to the controller. Each router object also runs on its own thread, allowing them to act independently.

When the client receives a packet, the first thing the router checks is if it came from the controller

`if(packet.getSocketAddress().equals(controllerAddress))`.

If the packet did come from the controller, it extracts the relevant information from the payload (the destination client port, the port to send the packet out of, and the port on the next point to send the packet to). The router then stores this information in its table for future packets going to the same end-user. The routers table is composed of a hashmap, whose keys are the string: 'destination client port', and whose values are an instance of the private nested class 'portPairing'. The class portPairing simply contains the value of the port to send the packet out of `int sendOutPort`, and the port on the next point to which the packet should be sent `int toNextPort`.

If the router is holding a packet in its value `DatagramPacket tempPacket` (this will contain the packet that was sent to the router that didn't have a rule for that packet. It holds it here while it waits for a response from the controller), signified by `Boolean isHoldingPacket`, then the router sets the packet's address to the toNextPort and sends the tempPacket onto the next point from the sendOutPort. Otherwise, it doesn't send on anything.

If the packet doesn't come from the controller, it checks if the hashmap contains the key of the destination client port

`if(this.rules.containsKey(dstClientPort))`

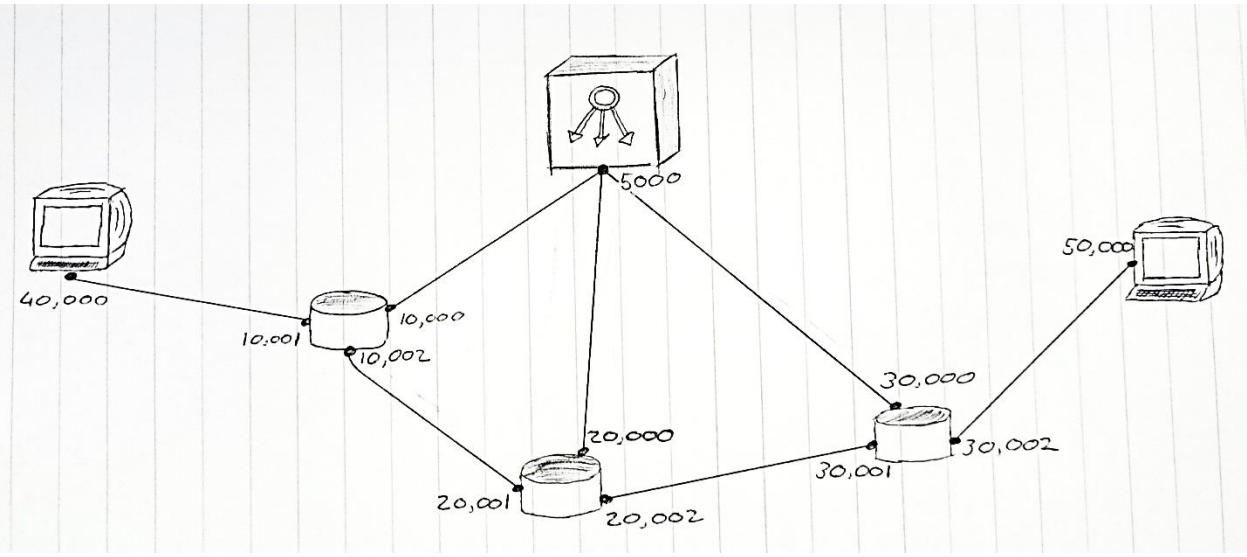
and if it does, it sets the socket address of the packet to the nextPort value from the portPairing instance. It then searches through its socket array to find the socket with the same port number as the sendOutPort value, and then sends the packet out of this port (same procedure as before for finding the correct socket).

```
for (int i = 0; i < this.sockets.length; i++)
{
	if (sockets[i].getLocalPort() == sendOutPort)
	{
		sockets[i].send(packet);
		i = this.sockets.length;
	}
}
```

If the router doesn't have a rule for the packet it just received, it stores the packet in the tempPacket value and sets the boolean isHoldingPacket to true. It then sends a packet to the controller containing the source client port, the destination client port and the port on the router that the packet will be sent from. The controller then uses this information to generate a packet containing the correct port pairings (as seen above) and sends it back to the router.

The controller class has a similar information structure as the router class. It uses a hashmap with the keys being the string: 'source client port' + ", "+ 'destination client port' and whose values are an instance of the private nested class RouterInfoInst. The RouterInfoInst contains three arrays (one for each router) and in each array, the element at index 0 will be the sendOutPort and the element at index 1 will be the toNextPort. The table for part one of the implementation is made in the constructor (line 45 approx.), with the port values used in the protocol diagram (see below).
On the reception of a packet, the controller splits apart the payload and uses the values in the payload (the source client port and the destination client port) as requests for the hashmap info. The controller then uses the information from the table to make packets that it will send to the routers, from the last router on the route back to the first router, so that they are all up to date when the packets start moving, because if the first router received info, it would send the packet it's storing to the second router before the second router had received the rules (see line 85 approx.).

Protocol Diagram:



Step One Implementation Packet Description:

```
> Frame 11: 114 bytes on wire (912 bits), 59 bytes captured (472 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 40000, Dst Port: 10001
> Data (27 bytes)
```

```
0000  02 00 00 00 45 00 00 37  70 26 00 00 80 11 00 00   ....E..7 p&......
0010  7f 00 00 01 7f 00 00 01  9c 40 27 11 00 23 f0 ac   ........ .@'..#..
0020  00 00 00 00 00 00 00 00  00 00 48 65 6c 6c 6f 0a   ........ ..Hello.
0030  34 30 30 30 30 0a 35 30  30 30 30                 40000.50 000
```

This is the packet from Client1 to Router1. The payload can be seen as having the source client port (40,000) and the destination client port (50,000). This is the first packet sent from this origin to this destination, so the next packet in the sequence will be going to the Controller. This packet is being sent to Router1 (10,001).

```
> Frame 12: 114 bytes on wire (912 bits), 59 bytes captured (472 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 10000, Dst Port: 5000
> Trapeze Access Point Access Protocol
```

```
0000  02 00 00 00 45 00 00 37  70 27 00 00 80 11 00 00   ....E..7 p'......
0010  7f 00 00 01 7f 00 00 01  27 10 13 88 00 23 0b d8   ........ '....#..
0020  00 00 00 00 00 00 00 00  00 00 34 30 30 30 30 0a   ........ ..40000.
0030  35 30 30 30 30 0a 31 30  30 30 30                 50000.10 000
```

As you can see, this packet is being sent to the Controller (5,000) from Router1 (10,000), with the source client port (40,000) and the destination client port (50,000) in the payload, along with the port the router is sending the packet out of (10,000).

```
> Frame 13: 114 bytes on wire (912 bits), 59 bytes captured (472 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 5000, Dst Port: 30000
> Trapeze Access Point Access Protocol
```

```
0000  02 00 00 00 45 00 00 37  70 28 00 00 80 11 00 00   ....E..7 p(......
0010  7f 00 00 01 7f 00 00 01  13 88 75 30 00 23 b8 b7   ........ ..u0.#..
0020  00 00 00 00 00 00 00 00  00 00 33 30 30 30 32 0a   ........ ..30002.
0030  35 30 30 30 30 0a 35 30  30 30 30                 50000.50 000
```

This packet is being sent from the Controller (5000) to Router3 (30000), with the required information for the router's table (sendOutPort: 30002, toNextPort: 50000, destination client port: 50000).

```
> Frame 14: 114 bytes on wire (912 bits), 59 bytes captured (472 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 5000, Dst Port: 20000
> Trapeze Access Point Access Protocol
```

```
0000  02 00 00 00 45 00 00 37  70 29 00 00 80 11 00 00   ....E..7 p).....
0010  7f 00 00 01 7f 00 00 01  13 88 4e 20 00 23 e1 c7   ........ ..N .#..
0020  00 00 00 00 00 00 00 00  00 00 32 30 30 30 32 0a   ........ ..20002.
0030  33 30 30 30 31 0a 35 30  30 30 30                  30001.50 000
```

This packet is being sent from the Controller (5000) to Router2 (20000), with the required information for the router's table (sendOutPort: 20002, toNextPort: 30001, destination client port: 50000).

```
> Frame 15: 114 bytes on wire (912 bits), 59 bytes captured (472 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 5000, Dst Port: 10000
> Trapeze Access Point Access Protocol
```

```
0000  02 00 00 00 45 00 00 37  70 2a 00 00 80 11 00 00   ....E..7 p*.....
0010  7f 00 00 01 7f 00 00 01  13 88 27 10 00 23 0a d8   ........ ..'..#..
0020  00 00 00 00 00 00 00 00  00 00 31 30 30 30 32 0a   ........ ..10002.
0030  32 30 30 30 31 0a 35 30  30 30 30                  20001.50 000
```

This packet is being sent from the Controller (5000) to Router1 (10000), with the required information for the router's table (sendOutPort: 10002, toNextPort: 20001, destination client port: 50000).

```
> Frame 16: 114 bytes on wire (912 bits), 59 bytes captured (472 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 10002, Dst Port: 20001
> Data (27 bytes)
```

```
0000  02 00 00 00 45 00 00 37  70 2b 00 00 80 11 00 00   ....E..7 p+.....
0010  7f 00 00 01 7f 00 00 01  27 12 4e 21 00 23 3e cb   ........ '.N!.#>.
0020  00 00 00 00 00 00 00 00  00 00 48 65 6c 6c 6f 0a   ........ ..Hello.
0030  34 30 30 30 30 0a 35 30  30 30 30                  40000.50 000
```

Having received the rule information from the Controller, Router1 (10,002) then sends the packet stored in the tempPacket to Router2 (20,001).

```
> Frame 17: 114 bytes on wire (912 bits), 59 bytes captured (472 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 20002, Dst Port: 30001
> Data (27 bytes)
```

```
0000  02 00 00 00 45 00 00 37  70 2c 00 00 80 11 00 00   ....E..7 p,......
0010  7f 00 00 01 7f 00 00 01  4e 22 75 31 00 23 f0 aa   ........ N"u1.#..
0020  00 00 00 00 00 00 00 00  00 00 48 65 6c 6c 6f 0a   ........ ..Hello.
0030  34 30 30 30 30 0a 35 30  30 30 30                  40000.50 000
```

Having received the rule information from the Controller, Router2 (20,002) then sends the packet it just received to Router3 (30,001).

```
> Frame 18: 114 bytes on wire (912 bits), 59 bytes captured (472 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 30002, Dst Port: 50000
> Data (27 bytes)
```

```
0000  02 00 00 00 45 00 00 37  70 2d 00 00 80 11 00 00   ....E..7 p-......
0010  7f 00 00 01 7f 00 00 01  75 32 c3 50 00 23 7b 7b   ........ u2.P.#{{
0020  00 00 00 00 00 00 00 00  00 00 48 65 6c 6c 6f 0a   ........ ..Hello.
0030  34 30 30 30 30 0a 35 30  30 30 30                  40000.50 000
```

Having received the rule information from the Controller, Router3 (30,002) then sends the packet it just received to Client2 (50,000).

```
20 20.751949    127.0.0.1       127.0.0.1       UDP     118 40000 → 10001 Len=29
21 20.757420    127.0.0.1       127.0.0.1       UDP     118 10002 → 20001 Len=29
22 20.760121    127.0.0.1       127.0.0.1       UDP     118 20002 → 30001 Len=29
23 20.762392    127.0.0.1       127.0.0.1       UDP     118 30002 → 50000 Len=29
```

This is the sequence of packets also sent from Client1 to Client2, but when the routers all contain the required information to forward the packets correctly. The packets are the same as those exchanged between the routers as seen above. However, as you can see from the sequence, the controller does not need to be contacted.

Step Two Implementation:

For the second part of the implementation for this protocol, we were tasked with altering the system so that the controller doesn't have any hardcoded information at start-up, but rather builds the table from information that the routers send to the controller at the initial launch. When the system is run, all the routers send their local port number and the port numbers of their surrounding points in the network (except the controller port) to the controller so that it can build the routing table for the system. This is done in the initialise function:

```
this.terminal.println("Sending initialisation info");
this.wait(100);
DatagramPacket routerInfo = (new StringContent("Initialise" + "\n" +
this.localPort + "\n" + this.remotePorts[1] + "\n" +
this.remotePorts[2])).toDatagramPacket();
this.sockets[0].send(routerInfo);
```

The router's initialise function contains a this.wait(100) function call, so that the controller has the time to insert the information into its table before receiving more packets from different routers. The system still functions without it for this network, however for a larger network it may be required as there would be more information to insert into the controller table, taking more time.

When the controller receives a packet, it now checks if it contains the string "Initialise" at the start of the payload, signifying that the packet contains information required to build the table. If the packet is an initialisation packet, the initialiseArrays function is called. The router also must keep count of how many initialisation packets it has received. If the count is equal to the number of routers, it is time to build the table using the makeTable() function. This is required as each router packet does not contain enough information in itself, to create a RouterInfoInst instance, so we must wait for more packets.

In the initialiseArrays function, the initRouter arrays are assigned port numbers from the appropriate packet payloads. For the first setup, these contain the route (port numbers) from Client1 to Client2, so they must be reversed in direction at some point, done by the reverseArraysDirection() function, which doesn't reverse each array, but rather reverses the direction of the route. This is called in the makeTable function, so the correct information is added for packets going in either direction. It is clear that this results in a very static implementation as the complexity would grow drastically with a large network (tens/hundreds of routers or clients) but it does mean that the information isn't directly hardcoded into the controller, but instead builds the table from information provided by the routers.

Step Two Implementation Packet Description:

```
> Frame 1: 136 bytes on wire (1088 bits), 70 bytes captured (560 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 10000, Dst Port: 5000
> Trapeze Access Point Access Protocol
```

```
0000  02 00 00 00 45 00 00 42  65 9b 00 00 80 11 00 00   ....E..B e.......
0010  7f 00 00 01 7f 00 00 01  27 10 13 88 00 2e 8b 2b   ........ '......+
0020  00 00 00 00 00 00 00 00  00 00 49 6e 69 74 69 61   ........ ..Initia
0030  6c 69 73 65 0a 31 30 30  30 30 0a 34 30 30 30 30   lise.100 00.40000
0040  0a 32 30 30 30 31                                 .20001
```

This is the packet sent from Router1 at start-up. As you can see, it contains the string "Initialise" at the start of the payload, along with the port numbers of its surrounding points (40000: Client1, 20001: Router2) and its local port (10000).

```
> Frame 2: 136 bytes on wire (1088 bits), 70 bytes captured (560 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 20000, Dst Port: 5000
> Trapeze Access Point Access Protocol
```

```
0000  02 00 00 00 45 00 00 42  65 9c 00 00 80 11 00 00   ....E..B e.......
0010  7f 00 00 01 7f 00 00 01  4e 20 13 88 00 2e 64 1a   ........ N ....d.
0020  00 00 00 00 00 00 00 00  00 00 49 6e 69 74 69 61   ........ ..Initia
0030  6c 69 73 65 0a 32 30 30  30 30 0a 31 30 30 30 32   lise.200 00.10002
0040  0a 33 30 30 30 31                                 .30001
```

This is the packet sent from Router2 at start-up. As you can see, it contains the string “Initialise” at the start of the payload, along with the port numbers of its surrounding points (10002: Router1, 30001: Router3) and its local port (20000).

```
> Frame 3: 136 bytes on wire (1088 bits), 70 bytes captured (560 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 30000, Dst Port: 5000
> Trapeze Access Point Access Protocol
```

```
0000  02 00 00 00 45 00 00 42  65 9d 00 00 80 11 00 00   ....E..B e.......
0010  7f 00 00 01 7f 00 00 01  75 30 13 88 00 2e 3d 07   ........ u0....=.
0020  00 00 00 00 00 00 00 00  00 00 49 6e 69 74 69 61   ........ ..Initia
0030  6c 69 73 65 0a 33 30 30  30 30 0a 32 30 30 30 32   lise.300 00.20002
0040  0a 35 30 30 30 30                                 .50000
```

This is the packet sent from Router3 at start-up. As you can see, it contains the string “Initialise” at the start of the payload, along with the port numbers of its surrounding points (20002: Router2, 50000: Client2) and its local port (30000).

Reflection:

As a whole, I found the assignment a good bit more difficult than the first. I feel that the first part of the implementation went better for me than the second. The second part, being the most difficult, had arduous moments and the implementation, as an entire system, would have been much better if it was entirely dynamic, even if it wasn’t a requirement. A non-static implementation would have had its benefits for the second part.

In terms of hours spent on the assignment, I would guess around the 30 hour range, although that is just a rough guess as I’m not entirely sure.