Assignment 2

CS2031 Telecommunication II

John Carbeck – 16309095

15/12/17

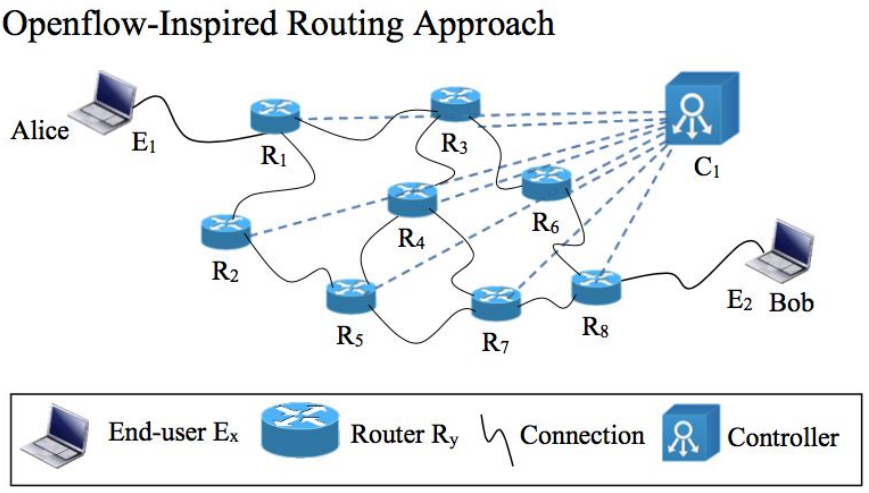


Figure 1: Sample Scenario

Protocol:

The implementation of this network is similar to a Openflow Routing Approach, where each router is connected to a controller. The controller contains a routing table that describes the topology of the entire network. Each Router only contains a flow table describing the adjacent routers and recent routes that it has requested from the controller.

When a packet is created and sent from a end node to a specified location, either a server or a client, it passes the packet to the router that it is connected to. Here the router looks to see if it contains the route, describing the next node to pass the packet to, based on the source and destination in the packet header. If Need the router will store the packet and request the route from the controller. Upon receiving the route from the controller it is added to the flow table of the router. The end node packet is then passed onto the next node that is stored in the route in the flow table.

Implementation:

To run the program the network layout must be predetermined. A simple 2 router network setup would be:

```
        java -cp "tcdlib.jar:." Client 50010
  java -cp "tcdlib.jar:." Controller 50012 50022
java -cp "tcdlib.jar:." Router 1 60010 50020 60001
java -cp "tcdlib.jar:." Router 2 50011 60011 60002
        java -cp "tcdlib.jar:." Server 50021
```

Note: ports of each node is stored in Constants.java

Routing tables:

All of the routers and the controller contain a data type RoutingTable which contains an array of the private data type Route. Each Route is made up of a source, destination, gateway and hops to final destination.

Example:

```
localhost/127.0.0.1:50012|/127.0.0.1:60011|localhost/127.0.0.1:50011|4
localhost/127.0.0.1:50011|/127.0.0.1:60011|localhost/127.0.0.1:50020|3
localhost/127.0.0.1:50020|/127.0.0.1:60011|localhost/127.0.0.1:50021|2
localhost/127.0.0.1:50021|/127.0.0.1:60011|localhost/127.0.0.1:60011|1
```

Construtor:

The RoutingTable is created using a specified size. This determines how many routes the RoutingTable can contain.

Methods:

- addRoute()

  Takes in a set of parameters or a string that have the information needed to create a new route to add to the graph. This method is used in two major instances in the network implementation. When the Router gets a route from the controller proceeding a request, and a new shortest path is found using BFS on the NetworkGraph in the Controller.

```
public boolean addRoute(InetSocketAddress sourceAddress, InetSocketAddress finalAddress,
        InetSocketAddress gateway, int hops)
{
    boolean routeAdded = false;
    Route routeToAdd = new Route(sourceAddress, finalAddress, gateway, hops);
    if(!isInTable(sourceAddress, finalAddress))
    {
        if(!isRoutingTableFull())
        {
            int i = 0;
            while((routes[i] != null) && (i < routes.length)) i++;
            if(i >= 0 && i < routes.length)
            {
                routes[i] = routeToAdd;
                routeAdded = true;
                routesInTable++;
            }
        }
    }
    return routeAdded;
}
```

  Add Route takes in various parameters for creating a route. The routeToAdd is a temporary route that is then checked to not be in the table and checked to make sure the table is not full. If it is a new instance of a route and the table is not full the array of routes will be indexed for the first empty element. When that is found the amount of routes in the table is added to adn the boolean route added is changed from false to true. Then the value of routeAdded is returned.

- addAllRoutes()

  Takes two routing tables and takes all of the routes in one table and adds it to another. This is used when the router gets a packet containing the routing information for all the sockets in a router based on a requested path.

```
public boolean addAllRoutes(RoutingTable tableTo, RoutingTable tableFrom)
{
    boolean valid = false;
    for(int i = 0; i < tableFrom.length; i++)
    {
        if(!tableTo.addRoute(tableFrom.getRoute(i))) return false;
    }
    return true;
}
```

  This method takes in two tables tableTo which is the destination of the routes that will be added and tableFrom which is the source of the routes to be added. All the routes in table From are indexed and added to the table to. If this for this is unable to be done false is returned instead of true.

- stringToTable()

The form in which the route is sent as is a string. This method is used to take a table as a string and turn it into a RoutingTable. The method getTable() returns the table as a string in such a way that it can be reconstructed with stringToTable().

```
public RoutingTable stringToTable(String tableAsString)
{
    String[] tableElements = tableAsString.split("\\$");
    RoutingTable newTable = new RoutingTable(tableElements.length);
    for(int i = 0; i < tableElements.length; i++) newTable.addRoute(tableElements[i]);
    return newTable;
}
```

stringToTable takes in a table is the form of a string. It then uses split to get the individual routes of the table. These are separated by "$". From there a new table is created from the length of elements/routes to be added. For all the routes to be added they are added to the newTable and newTable is returned.

- getRoute()

This method indexes the RoutingTable for a Route that contains the specified source and destination. The method returns the route as a string. This is used in various places and is vital for function of the data type.

```
public String getRoute(InetSocketAddress source, InetSocketAddress destination)
{
    String route = "";
    boolean finished = false;
    int i = 0;
    while(i < routesInTable && !finished)
    {
        InetSocketAddress routeSource = routes[i].getSourceAddress();
        InetSocketAddress routeFinal = routes[i].getFinalAddress();
        if(routeSource.equals(source) && routeFinal.equals(destination))
        {
            route = getRoute(i);
            finished = true;
        }
        i++;
    }
    return route;
}
```

```
public String getRoute(int index)
{
    return (routes[index].getSourceAddress()).toString() + "|" + (routes[index].getFinalAddress()).toString() +
    "|" + (routes[index].getGateway()).toString() + "|" + (routes[index].getHops());
}
```

getRoute() takes a source and destination for that route that is being looked for. The routes in the routing table are indexed then and the values of the source and destination of the route indexed are compared to the ones specified. If they are both equal then that index i is used to get the string using the getRoute() method that only takes a index. This gives a string constructed in such a way so that is can be used by addRoute().

- getTable()

This method takes a RoutingTable and constructs it into a string that is readable to stringToTable(). This method is used in the sending of routes in a Datagram Packet payload.

```
public String getTable()
{
    String routingTableString = "";
    for(int i = 0; i < routes.length; i++)
    {
        if(routes[i] != null)
        {
            routingTableString += getRoute(i);
            if(i < (routes.length - 1))
            {
                routingTableString += "$";
            }
        }
    }
    return routingTableString;
}
```

This method indexes all the routes in the routing table turning them in strings using the getRoute() method described above. When it .is the last element in the table the character "$" signaling that the end route and the start of the next one.

- isInTable()

This method looks for a route in a table based of a source and destination address. If the table contains the specified route the method returns true, else false.

```
public boolean isInTable(InetSocketAddress source, InetSocketAddress destination)
{
    boolean isInTable = false;
    for(int i = 0; i < routesInTable; i++)
    {
        InetSocketAddress routeSource = routes[i].getSourceAddress();
        InetSocketAddress routeFinal = routes[i].getFinalAddress();
        if(routeSource.equals(source) && routeFinal.equals(destination))
        {
            isInTable = true;
            i = routesInTable;
        }
    }
    return isInTable;
}
```

isInTable uses the specified source and destination addresses and indexes through all the routes in the table comparing the values of each elements to the ones that are parameters. If one of the routes have both the same source and destinations as the ones specified the value true is returned, else false.

- getNextRoute()

This method will take a source and destination and find the route in table. The method then returns the gateway that the packet needs to passed along to in the form of a InetSocketAddress. This is used by the routers when they want to pass a packet to the next destination in the route

```
public InetSocketAddress getNextRouter(InetSocketAddress source, InetSocketAddress destination)
{
    for(int i = 0; i < routesInTable; i++)
    {
        InetSocketAddress routeSource = routes[i].getSourceAddress();
        InetSocketAddress routeFinal = routes[i].getFinalAddress();
        if(routeSource.equals(source) && routeFinal.equals(destination)) return routes[i].getGateway();
    }
    return null;
}
```

This method takes indexes the routes in the table and until the specified one is found. The value of the the gateway is then returned.

- nextOpenRoute()

  This method return the index in the RoutingTable that is free. This is used in the adding of route to an existing table.

```
private int nextOpenRouteInTable()
{
    int i = 0;
    while((routes[i] != null) && (i < routes.length)) i++;
    if(i >= 0 && i < routes.length) return i;
    else return - 1;
}
```

  nextOpenRoute() goes through all the elements and returns the index i for the index that points to a empty route

- isRoutingTableFull()

  This method returns if the RoutingTable is full or not. Mainly used in adding a route to a existing table.

```
private boolean isRoutingTableFull()
{
    boolean isRoutingTableFull = true;
    for(int i = 0; i < routes.length; i++)
    {
        if(routes[i] == null) isRoutingTableFull = false;
    }
    return isRoutingTableFull;
}
```

  isRoutingTablefull goes through all the elements aif it finds a null Route/open route then the value false is returned else it is returned true.

Advantages:

RoutingTable allows for the easy sending of routing information from one node to the next. Structuring the routing information this way allows for the flexibility in how much routing information is sent at once. This allows for tables, single routes, or specific addresses to sent using a single data type.

Disadvantages:

The drawback to this data type is the static nature of arrays. The size of the routing table is predetermined thus making it expensive to update the size of the table. This also means that the router has to only remember the most recent routes allowing for more contact to the controller making for slower transmission speeds. This could have been improved by using a binary tree. This would allow for fast indexing and flexibility in size. Though this implementation works for a small static network it would not for a large dynamic one.

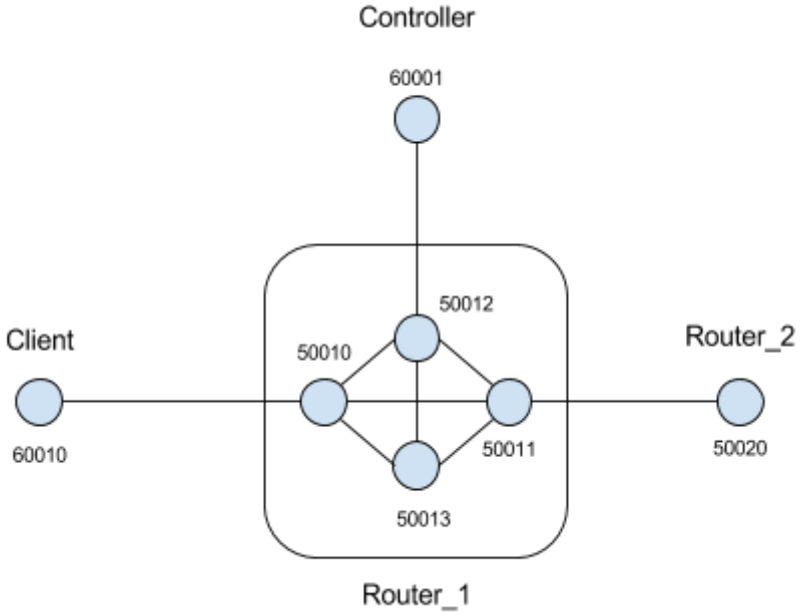
Router:

The main functions of the router are to contact the controller with its routing table of adjacent nodes on startup, request and store a route for a unknown path, and pass packets from a source to the appropriate gateway based on a packet's destination.

Setup:

Upon setup the router creates a routing table with its local connections. This is sent to the controller in the form of a string. This is to allow for the controller to create a complete routing table that contains all the connections in the network.

Example network as seen by router:



Initial Routing Table:

```
localhost/127.0.0.1:50010|localhost/127.0.0.1:50011|localhost/127.0.0.1:50011|0
localhost/127.0.0.1:50011|localhost/127.0.0.1:50012|localhost/127.0.0.1:50012|0
localhost/127.0.0.1:50012|localhost/127.0.0.1:50013|localhost/127.0.0.1:50013|0
localhost/127.0.0.1:50013|localhost/127.0.0.1:50010|localhost/127.0.0.1:50010|0
localhost/127.0.0.1:50010|localhost/127.0.0.1:50012|localhost/127.0.0.1:50012|0
localhost/127.0.0.1:50011|localhost/127.0.0.1:50013|localhost/127.0.0.1:50013|0
localhost/127.0.0.1:50010|localhost/127.0.0.1:60010|localhost/127.0.0.1:60010|1
localhost/127.0.0.1:50011|localhost/127.0.0.1:50020|localhost/127.0.0.1:50020|1
localhost/127.0.0.1:50012|localhost/127.0.0.1:60001|localhost/127.0.0.1:60001|1
```

Packet with routers local table:

2	0.000053	192.168.0.12	192.168.0.255	UDP	76	57621 → 57621 Len=44
3	7.042866	127.0.0.1	127.0.0.1	UDP	768	50012 → 60001 Len=736
4	8.963286	127.0.0.1	127.0.0.1	UDP	768	50022 → 60002 Len=736

Requesting Routes:

When the router receives a packet is will call the sendPacket() method. This will contact the controller if the there is no route in the table for the specified destination. The router will send a packet with the address of the router and the destination in the header. This format will make the controller get the relevant routes for the router and send the routes back to the router in the form of a packet with the routes as strings in the payload. The router will then add those routes to its local flow table and send the last packet received.

The following method, sendPacket(), is called from the method processPacket().

```
private synchronized void sendPacket(InetSocketAddress source, InetSocketAddress destination, DatagramPacket packet)
{
    try{
        if(!flowTable.isInTable(source, destination))
        {
            this.lastPacket = packet;
            this.lastDestination = destination;
            source = new InetSocketAddress(Constants.DEFAULT_NODE, socket[controllerIndex].getLocalPort());
            this.lastSource = source;
            byte[] buffer = new byte[Constants.HEADERLENGTH];
            byte[] srcAddress = addressToByte(source.getAddress());
            int srcPort = source.getPort();
            byte[] dstAddress = addressToByte(destination.getAddress());
            int dstPort = destination.getPort();

            ByteBuffer tempbuffer = ByteBuffer.wrap(buffer);
            for(int i = 0; i < srcAddress.length; i++)
                tempbuffer.put(Constants.SOURCE_INDEX + i, srcAddress[i]);
            for(int i = 0; i < dstAddress.length; i++)
                tempbuffer.put(Constants.DEST_INDEX + i, dstAddress[i]);

            tempbuffer.putInt(Constants.SOURCE_PORT_INDEX, srcPort);
            tempbuffer.putInt(Constants.DEST_PORT_INDEX, dstPort);

            buffer = tempbuffer.array();
            DatagramPacket newPacket = new DatagramPacket(buffer, buffer.length, controllerAddress);
            socket[controllerIndex].send(newPacket);
        }
        else
        {
            InetSocketAddress gateway = flowTable.getNextRouter(source,destination);
            int index = findSocketIndex(gateway);
            packet.setSocketAddress(gateway);
            socket[index].send(packet);
        }

    } catch(java.lang.Exception e) {e.printStackTrace();}
}
```

In sendPacket() the route is checked to be in the flowtable of the router. If it's not the router the current packet, source, and destination are all saved in the variables: lastPacket, lastDestination, and lastSource. These will be used when the packet receives a response from the controller. The router then creates a packet with no payload and the source and destination of the intended route, the source now being the address of the router socket. The request packet is then sent through the socket allocated for route request to the controller.

The following process takes place in the processPacket() which is called in the onRecipt() method of Router.

```
if(source.equals(controllerAddress))
{
    terminal.println("Here");
    InetSocketAddress newSource = lastSource;
    flowTable.addAllRoutes(flowTable, flowTable.stringToTable((new StringContent(packet)).toString()));

    //find route that connects outside the router

    while(((flowTable.getNextRouter(newSource,lastDestination)).getPort() >= localPorts[0]) &&
        ((flowTable.getNextRouter(newSource,lastDestination)).getPort() <= localPorts[localPorts.length-1]))
    {
        newSource = flowTable.getNextRouter(newSource, lastDestination);
    }

    sendPacket(newSource, lastDestination, lastPacket);
}
else
{
    sendPacket(source, destination, packet);
}
```

The source of the packet is checked to be the controller address if it is then the packet being sent is not something to be passed on rather it is a answer to a route request. The routes in the packet are parsed from the payload and added to the routing table of the router. Next the source is changed until it is a socket that connects to a socket that is outside the router that this is being processed it. This has to be done due to the NetworkGraph looking at all the sockets as nodes instead of router. Once the new source is figured out a call to sendPacket is made using the newSource and the packet information stored from before the route request.

Packets Exchanged In Request:

5	13.443093	127.0.0.1	127.0.0.1	UDP	50	60010 → 50010 Len=18
6	13.443947	127.0.0.1	127.0.0.1	UDP	48	50012 → 60001 Len=16
7	13.455987	127.0.0.1	127.0.0.1	UDP	154	60001 → 50012 Len=122
8	13.461753	127.0.0.1	127.0.0.1	UDP	50	50011 → 50020 Len=18

5. The client sending a packet to the final destination through the router that it is connected to.
6. The router sending a route request to the controller
7. The controller sending the routes back to the router
8. The router passing the packet onto the next node in the path to the final destination

Packet With Routing Information:

6	13.443947	127.0.0.1	127.0.0.1	UDP	48	50012 → 60001 Len=16
7	13.455987	127.0.0.1	127.0.0.1	UDP	154	60001 → 50012 Len=122
8	13.461753	127.0.0.1	127.0.0.1	UDP	50	50011 → 50020 Len=18
9	13.465693	127.0.0.1	127.0.0.1	UDP	48	50022 → 60002 Len=16

```
▶ Frame 7: 154 bytes on wire (1232 bits), 154 bytes captured (1232 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: 60001, Dst Port: 50012
▶ Data (122 bytes)
0000  02 00 00 00 45 00 00 96  5b ad 00 00 40 11 00 00   ....E... [...@...
0010  7f 00 00 01 7f 00 00 01  ea 61 c3 5c 00 82 fe 95   ........ .a.\....
0020  7f 00 00 01 7f 00 00 01  00 00 ea 61 00 00 c3 5c   ........ ...a...\
0030  2f 31 32 37 2e 30 2e 30  2e 31 3a 35 30 30 31 31   /127.0.0 .1:50011
0040  7c 2f 31 32 37 2e 30 2e  30 2e 31 3a 36 30 30 31   |/127.0. 0.1:6001
0050  31 7c 2f 31 32 37 2e 30  2e 30 2e 31 3a 35 30 30   1|/127.0 .0.1:500
0060  32 30 7c 33 24 2f 31 32  37 2e 30 2e 30 2e 31 3a   20|3$/12 7.0.0.1:
0070  35 30 30 31 32 7c 2f 31  32 37 2e 30 2e 30 2e 31   50012|/1 27.0.0.1
0080  3a 36 30 30 31 31 7c 2f  31 32 37 2e 30 2e 30 2e   :60011|/ 127.0.0.
0090  31 3a 35 30 30 31 31 7c  34 24                     1:50011| 4$
```

Contains the table:

```
localhost/127.0.0.1:50022|/127.0.0.1:60010|localhost/127.0.0.1:50020|4
localhost/127.0.0.1:50020|/127.0.0.1:60010|localhost/127.0.0.1:50011|3
```

Note: Due to the structure of the network, all sockets treated at individual nodes, the routes sent back are all the connections that are needed to get to a socket outside of the router.

Controller:

The controllers contains the routing table for the entire network. The way the controller builds a routing table is it receive the local table of all the routes. This local table contains all the connections within the router and the its connections to adjacent elements. In set one this was just simply hard coded in. This dynamic creation is similar to link state protocol.

The controller has to deal with two types of packets. Ones that are establishing a connection, a packet containing all the routes the router can see, the other is a request for route information. The way the controller is able to distinguish is if the packet has string contents. None being a route request and not empty being a route request.

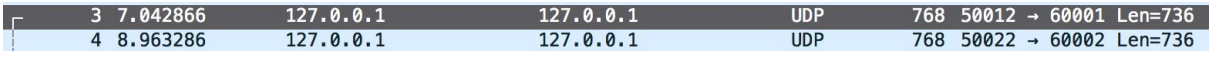
```
if(!((new StringContent(packet).toString()).equals("")))
{
    String tableString = (new StringContent(packet)).toString();
    routingTable.addAllRoutes(routingTable, routingTable.stringToTable(tableString));
    network.add(tableString);

    terminal.println("New Router Connection!");
}
else
{
    terminal.println("Gateway Request!");

    if(!(routingTable.isInTable(source, destination)))
    {
        RoutingTable tempTable = routingTable.stringToTable(network.findRoute(source, destination, network));
        routingTable.addAllRoutes(routingTable, tempTable);
    }
    try
    {
        int[] routerPorts = getRouterPorts(source.getPort());

        RoutingTable tempTable = new RoutingTable(Constants.ROUTER_SIZE);
        for (int i = 0; i < routerPorts.length; i++ )
        {
            InetSocketAddress tempAddress = new InetSocketAddress(Constants.DEFAULT_NODE, routerPorts[i]);
            tempTable.addRoute(routingTable.getRoute(tempAddress,destination));
        }
        String payloadString = tempTable.getTable();


        byte[] payload = payloadString.getBytes();
        byte[] header = new byte[Constants.HEADERLENGTH];

        InetSocketAddress newSource = new InetSocketAddress(Constants.DEFAULT_NODE, socket[findSocketIndex(source)].getLocalPort());
        InetSocketAddress newDestination = source;
        byte[] newSrcData = addressToByte(source.getAddress());
        byte[] newDestData = addressToByte(destination.getAddress());

        tempbuffer = ByteBuffer.wrap(header);
        for(int i = 0; i < newDestData.length; i++)
            tempbuffer.put(Constants.DEST_INDEX + i, newSrcData[i]);
        for(int i = 0; i < newSrcData.length; i++)
            tempbuffer.put(Constants.SOURCE_INDEX + i, newSrcData[i]);

        tempbuffer.putInt(Constants.SOURCE_PORT_INDEX, newSource.getPort());
        tempbuffer.putInt(Constants.DEST_PORT_INDEX, newDestination.getPort());
        header = tempbuffer.array();

        byte[] buffer = new byte[header.length + payload.length];
        System.arraycopy(header, 0, buffer, 0, header.length);
        System.arraycopy(payload, 0, buffer, header.length, payload.length);

        socket[findSocketIndex(source)].send(new DatagramPacket(buffer, buffer.length, source));

    } catch(java.lang.Exception e) {e.printStackTrace();}
}
```

This packet processing is done in the processPacket() method called in the onRecipt() of Controller.

You can see that if the packet is not empty the routing table adds all the routes that are contained in the packets payload. The network, which is of type NetworkGraph, adds the routes into the graph representing the topology of the network.

If the packet is empty that means that the controller will be processing a gateway/route request. To do this the controller first checks to see if the controller has the route already in its routing table. If this is the case the controller goes on to create a table of all the routes to send to router. This is turned into a string and stored in the payload of the packet. A new datagram packet is then constructed with the source being the controller and the destination being the socket of the router that requested the information. This packet is then sent through the socket that is allocated for the address of which the packet was received from.

Establish Packets being sent to controller:



Router 1 table:

```
localhost/127.0.0.1:50010|localhost/127.0.0.1:50011|localhost/127.0.0.1:50011|0
localhost/127.0.0.1:50011|localhost/127.0.0.1:50012|localhost/127.0.0.1:50012|0
localhost/127.0.0.1:50012|localhost/127.0.0.1:50013|localhost/127.0.0.1:50013|0
localhost/127.0.0.1:50013|localhost/127.0.0.1:50010|localhost/127.0.0.1:50010|0
localhost/127.0.0.1:50010|localhost/127.0.0.1:50012|localhost/127.0.0.1:50012|0
localhost/127.0.0.1:50011|localhost/127.0.0.1:50013|localhost/127.0.0.1:50013|0
localhost/127.0.0.1:50010|localhost/127.0.0.1:60010|localhost/127.0.0.1:60010|1
localhost/127.0.0.1:50011|localhost/127.0.0.1:50020|localhost/127.0.0.1:50020|1
localhost/127.0.0.1:50012|localhost/127.0.0.1:60001|localhost/127.0.0.1:60001|1
```

Router 2 table:

```
localhost/127.0.0.1:50020|localhost/127.0.0.1:50021|localhost/127.0.0.1:50021|0
localhost/127.0.0.1:50021|localhost/127.0.0.1:50022|localhost/127.0.0.1:50022|0
localhost/127.0.0.1:50022|localhost/127.0.0.1:50023|localhost/127.0.0.1:50023|0
localhost/127.0.0.1:50023|localhost/127.0.0.1:50020|localhost/127.0.0.1:50020|0
localhost/127.0.0.1:50020|localhost/127.0.0.1:50022|localhost/127.0.0.1:50022|0
localhost/127.0.0.1:50021|localhost/127.0.0.1:50023|localhost/127.0.0.1:50023|0
localhost/127.0.0.1:50020|localhost/127.0.0.1:50011|localhost/127.0.0.1:50011|1
localhost/127.0.0.1:50021|localhost/127.0.0.1:60011|localhost/127.0.0.1:60011|1
localhost/127.0.0.1:50022|localhost/127.0.0.1:60002|localhost/127.0.0.1:60002|1
```

Controller table:

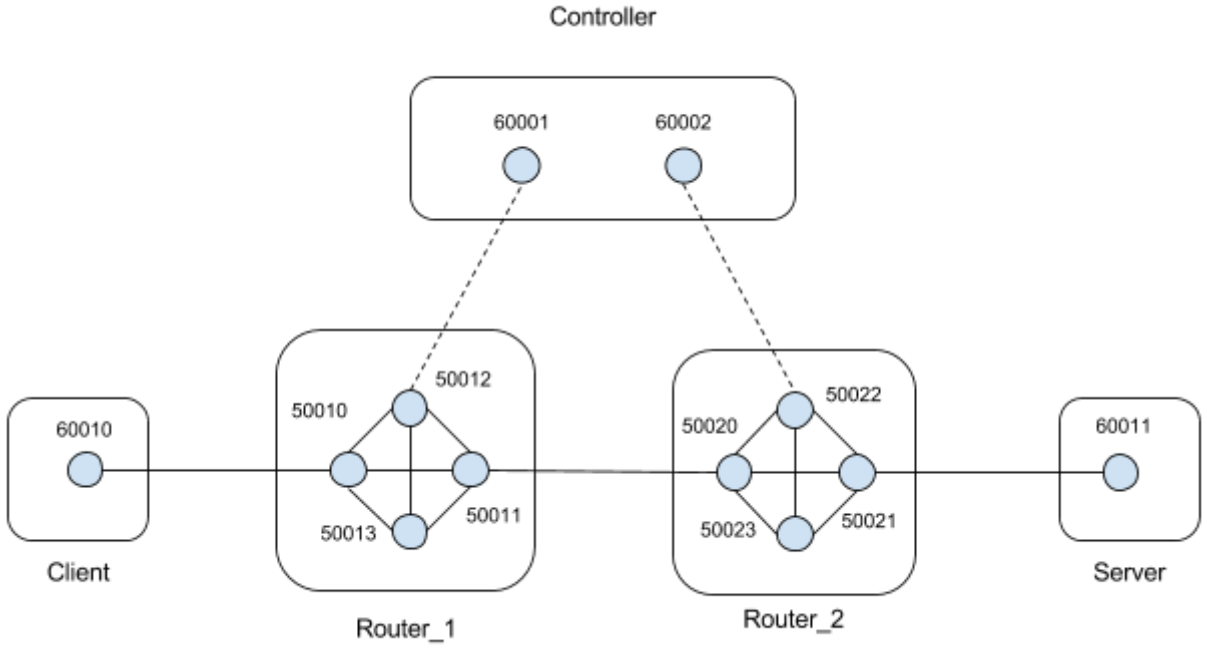
```
/127.0.0.1:50010|/127.0.0.1:50011|/127.0.0.1:50011|0
/127.0.0.1:50011|/127.0.0.1:50012|/127.0.0.1:50012|0
/127.0.0.1:50012|/127.0.0.1:50013|/127.0.0.1:50013|0
/127.0.0.1:50013|/127.0.0.1:50010|/127.0.0.1:50010|0
/127.0.0.1:50010|/127.0.0.1:50012|/127.0.0.1:50012|0
/127.0.0.1:50011|/127.0.0.1:50013|/127.0.0.1:50013|0
/127.0.0.1:50010|/127.0.0.1:60010|/127.0.0.1:60010|1
/127.0.0.1:50011|/127.0.0.1:50020|/127.0.0.1:50020|1
/127.0.0.1:50012|/127.0.0.1:60001|/127.0.0.1:60001|1
/127.0.0.1:50020|/127.0.0.1:50021|/127.0.0.1:50021|0
/127.0.0.1:50021|/127.0.0.1:50022|/127.0.0.1:50022|0
/127.0.0.1:50022|/127.0.0.1:50023|/127.0.0.1:50023|0
/127.0.0.1:50023|/127.0.0.1:50020|/127.0.0.1:50020|0
/127.0.0.1:50020|/127.0.0.1:50022|/127.0.0.1:50022|0
/127.0.0.1:50021|/127.0.0.1:50023|/127.0.0.1:50023|0
/127.0.0.1:50020|/127.0.0.1:50011|/127.0.0.1:50011|1
/127.0.0.1:50021|/127.0.0.1:60011|/127.0.0.1:60011|1
```

```
/127.0.0.1:50022|/127.0.0.1:60002|/127.0.0.1:60002|1
```

Note: Some connections are repeated. This doesn't matter because the network is represented as a undirected graph. But in a real network this might not be the case and each connection may only be one way which would be significant for routing.

From all the connections in the network the controller creates a graph. This undirected graph is stored in a data type NetworkGraph. It should be noted that the graph of this network doesn't have weighted and is undirected. In a large network it would be important to have weighted edges as in a real implementation the connections have different speeds and this would be important in finding the fastest route.

Graph from routing table above:



The use of breadth first search algorithm on the undirected graph allows for finding the shortest path from a source to a destination through the nodes that are adjacent in the graph. In a real implementation with weighted edges it would be better to use Dijkstra Algorithm, which includes the weight of edges in calculation of fastest path.

Path and routes from Router_1 request for a route to controller:

```
FOUND PATH![50012, 50011, 50020, 50021, 60011]
localhost/127.0.0.1:50012|/127.0.0.1:60011|localhost/127.0.0.1:50011|4
localhost/127.0.0.1:50011|/127.0.0.1:60011|localhost/127.0.0.1:50020|3
localhost/127.0.0.1:50020|/127.0.0.1:60011|localhost/127.0.0.1:50021|2
localhost/127.0.0.1:50021|/127.0.0.1:60011|localhost/127.0.0.1:60011|1
```

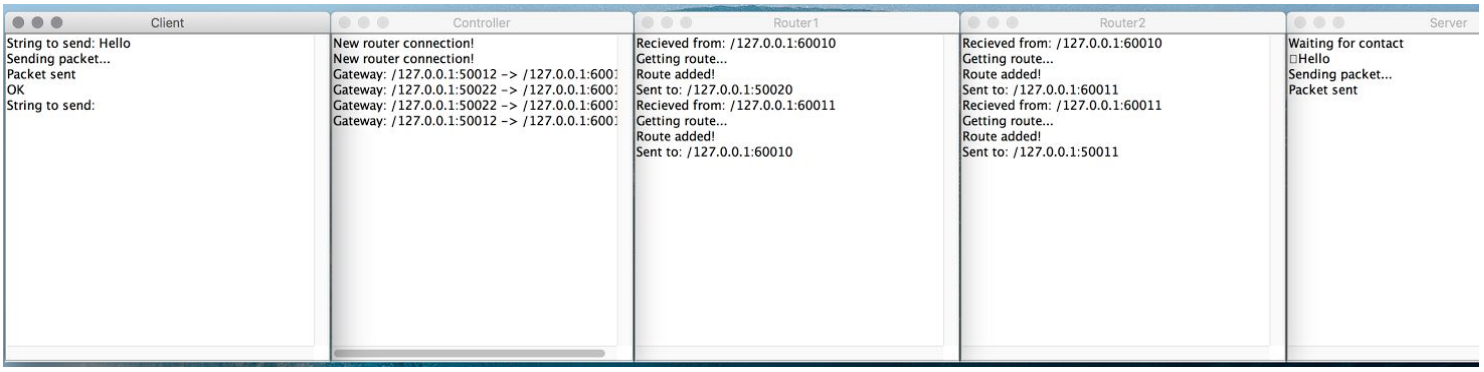
Each step in calculated path is turned into routes and added to the routing table for the controller. The controller is then able to send the relevant routes to each router due to the controller knowing all the ports that exist within the router. The routes sent to the routers are the ones that allow them to pass the packet to adjacent routers until the packet ends up in its final destination. Having the graph used the individual sockets at nodes would become very inefficient in a big net with routers with many sockets. In that case it would be better to just look at the router as a whole and ignore the routes between sockets in the router.

Routes sent to Router_1 on request:

```
localhost/127.0.0.1:50012|/127.0.0.1:60011|localhost/127.0.0.1:50011|4
localhost/127.0.0.1:50011|/127.0.0.1:60011|localhost/127.0.0.1:50020|3
```

Routes sent to Router_2 on request:

```
localhost/127.0.0.1:50020|/127.0.0.1:60011|localhost/127.0.0.1:50021|2
localhost/127.0.0.1:50021|/127.0.0.1:60011|localhost/127.0.0.1:60011|1
```



Reflection:

In this assignment I did well to create the RoutingTable class and the NetworkGraph class. These classes made allowed for a cleaner implementation of the protocols. It also allowed for easy sharing of information between the router and controller as they shared a data type with the same methods. I also believe that my implementation of the undirected graph for NetworkGraph was done very well. This allowed for dynamic routing and something close to link state protocol.

Things I thought I could have done better was optimise the code to run faster on large networks. Two things about my current implementation limit me. One is the storage or routes in a array. This means when looking for a certain route the runtime is n where if the routes were stored in a binary tree then the runtime would be ln(n). This is not a issue for a small network but in a network where the router has hundreds or thousands routes in the routing tables the current implementation would be very inefficient. The second thing is the NetworkGraph looking at each individual socket at nodes on the graph. This means more hops per path. Again this does not have a huge impact on performance for a small network but the amount nodes that need to be shown in the NetworkGraph should be as small as possible.

Time: 45hrs