

Assignment Two

Telecommunications II



Conor Clery

16320175

Introduction

From my interpretation of the assignment specification the goal of the assignment was to design and implement an OpenFlow-like network whereby a controller, connected to each router in the system, makes routing decisions for every packet sent from a specific client to another client. Such routing decisions would be made the first time a client transmits to another client. After the first transmission, if another packet is transmitted to the same destination the system of routers will remember what path the packet should take without help from the controller. The controller itself will need to have some representation of the network (which it attains at system initialization) which can then be transformed into an *undirected graph* which will help with routing decisions. An object oriented programming approach complemented by the use of threads will be necessary to simulate a realistic networking situation as well as greatly reduce the difficulty of implementation. In addition to all of this, the system should be designed in such a way that the transfer and routing of packets as well as the processes of the controller all work on any $m \times n$ grid of routers.

Specifications to Meet

Controller

Needs to be able to query each router generated by the system for information about the various connections it possesses and from the received data, generate a sort of *map* of the network. From this network map, an *undirected graph* is generated with each router and client being represented as a vertex on the graph and each connection between routers/clients being represented as edges in the graph. Upon receiving a query from a router on how to reach a destination, the controller will either send to the router from it's stored routes the next station to send a packet or, if a path does not exist, generate one by use of Dijkstra's Shortest Path algorithm.

Router

Each router shall have five sockets. Four of these sockets will be used for connection to clients and routers whereas the fifth socket will be used for communication with the controller. In addition to this each router will also maintain a *destination table* which contains information on where to send a certain packet to a specific destination for that specific router. Should a router not possess an entry in said table it shall send a DSTQUERY (destination query) to the controller. Upon receipt of a NEXTDST packet from

the controller, the router will then store the information received from the controller (so it does not have to query it again for that destination) and sends the packet on to the next station. On top of all this, each router shall also maintain a routing table detailing all of its connections. Each table entry contains 4 fields. The destination name, the port to use for the sending of the packet, the destination port of the packet and a randomly assigned latency (in seconds). This arbitrarily assigned time is what enables the functionality of Dijkstra's Algorithm. Should the controller poll the router for data, this table will be sent (and is used to draw the controllers network map).

Client

The clients will be less complicated than the other types of node as they simply require two sockets. One for sending transmissions to it's connected router and one for receiving them. Each socket shall be threaded to enable simultaneous sending and receiving of packets. The user should also be able to input a string to send in the payload of the packet as well as typing in the address of the destination.

Dijkstra's Shortest Route

This algorithm, as previously mentioned, will require the conversion of received router data, with particular importance based on the randomly assigned connection times, at the controller into an *undirected graph* and from this information implement Dijkstra's algorithm with an object oriented approach. This algorithm, along with the theory behind it and it's implementation in this assignment, will be discussed in detail later on within this documentation.

Demonstration of Implementation

Firstly, to compile the source code given in the *src* folder, the following command must be run from a terminal. Alternatively one can also compile the source code using their preferred IDE however the *tcdlib.jar* file must be added to the build path of the project.

```
$ javac -cp tcdlib.jar:. *.java
```

And to execute the code:

```
$ java -cp tcdlib.jar:. Main
```

The Setup Window

Upon startup of the program, a window to set various parameters of the system will be displayed. The options include whether or not to show terminal windows for all of the routers within the system (only practical with a small amount of routers but can be useful for debugging) and whether or not to be verbose with console output (displays routing tables, initialization details and transmission information) which is again useful for debugging. After these two options are decided upon, two inputs are then required for the generation of the $m \times n$ grid of routers. The width and the height of the grid. All of these inputs will then set various global variables within the system that enable its overall functionality. By default at startup, the only windows that are displayed are the client windows and the controller and the *verbose* type of console output is switched off.

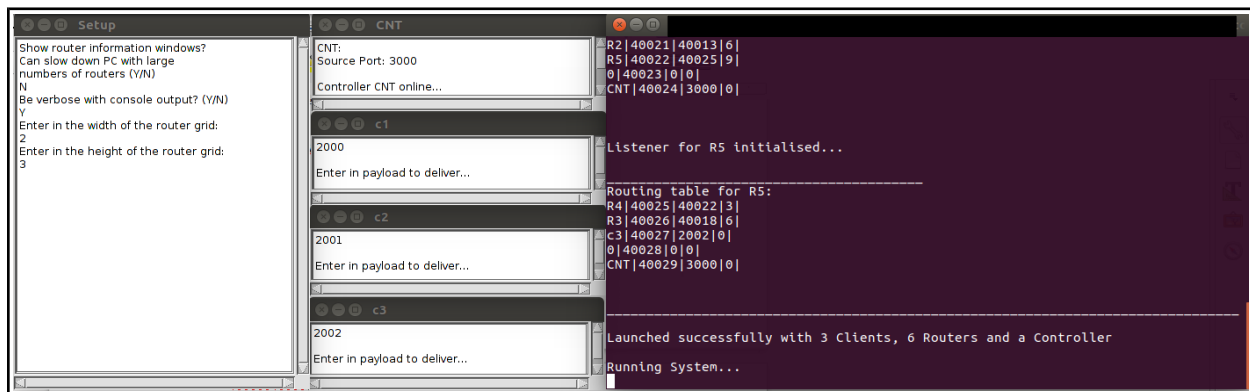
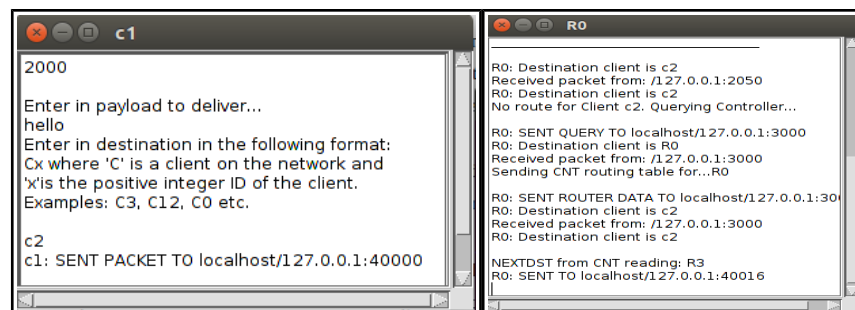


Image shows the startup of the program, which windows are displayed and various console output based on the Setup parameters.

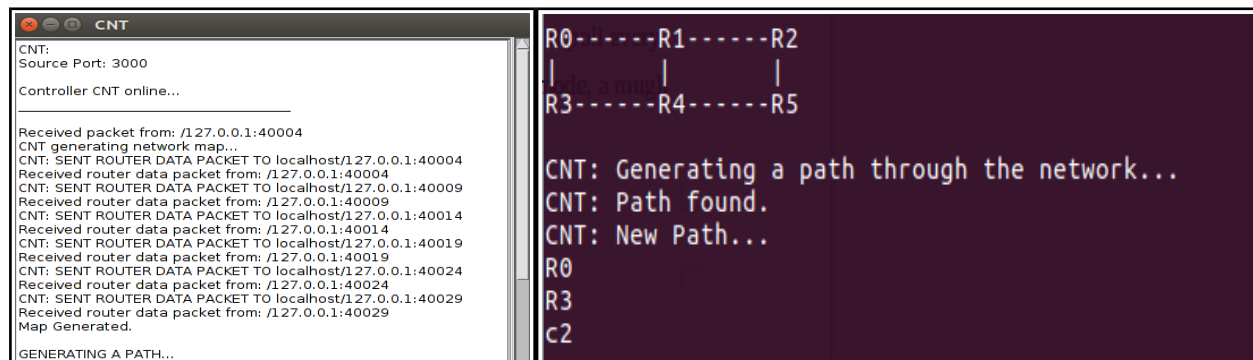
The First Transmission and its Effects on the System

Once the program has initialized correctly, a transmission can be made from one client to another. Firstly the user is prompted to input a payload to deliver. Secondly, the user is asked to input a destination. This destination *must be in lower case*. The destination is inputted as 'cX' where c stands for client and X is the number associated with the client. The names of each client can be found at the top of each client window. Note that transmissions directly from the client to the controller are not possible (due to the fact that the client side implementation should be reduced in complexity by the controller-router system).



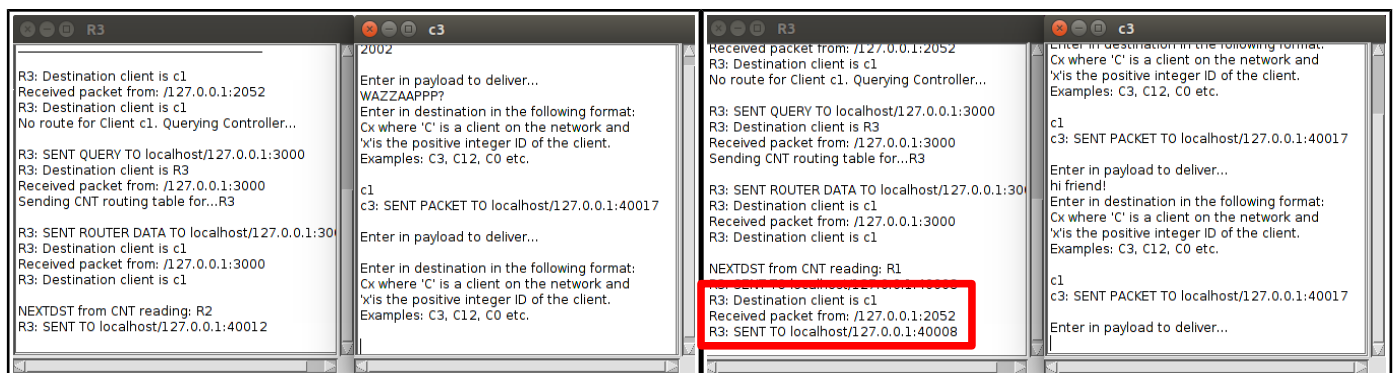
Images showing sample client input and router's first interactions with controller and client.

Upon sending of the first packet, the router will inspect its records for the next station which it shall transmit the packet to. If no record exists (which it won't for the first transmission), it will query the controller. The controller will then, upon receipt of the query check if it possesses a map of the network. Since it is the first transmission, the controller will not possess a network map and proceed to poll every router in the system for their routing tables and generate a graph from the data. In verbose mode, a rough ASCII picture of the created network can also be seen. After the map is generated, a path through the network is generated and the next station on the path is sent to the router that initially queried the controller.



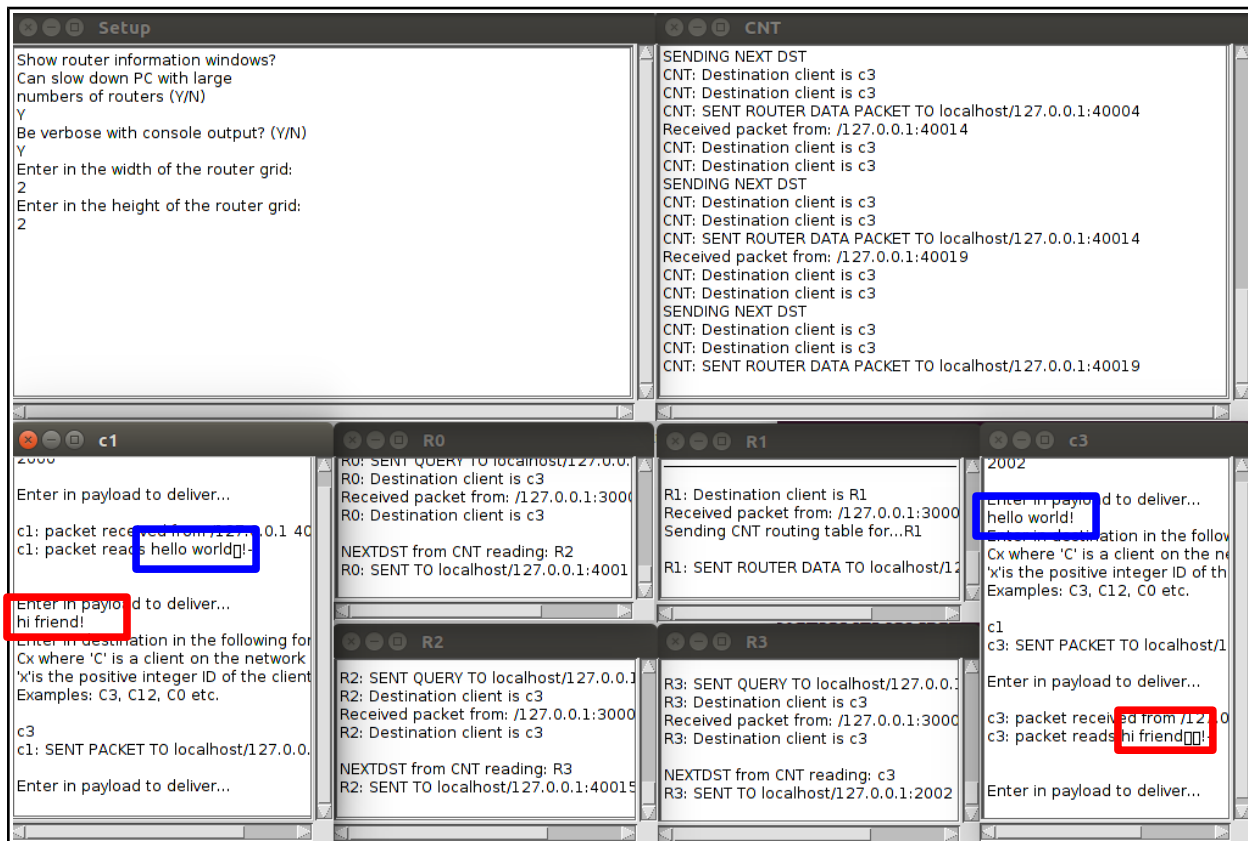
Controller generating a map of the network by polling all routers. When all data is received, a path through the network based on the original query is generated.

When the router receives a packet encoded with NEXTDST from the the controller, it stores the relevant information in it's database and forwards the packet on to the next station and the process repeats (however a network map and path are only generated once and stored for efficiency) until the destination is reached. It is important to note that interaction with the controller should only take place when a client first transmits to another client. All subsequent transmissions should successfully reach their destinations based solely on the router's records.



On left: c3 sends for the first time a packet to c1. Controller is queried, map is generated and a NEXTDST is sent to the router. *On right:* Router already has the next destination stored in its records and simply sends the packet on to the router in its records.

This process can then of course be repeated for all clients in the system. Below is an screen shot from the program with the routers laid out in the format in which they are stored within the controller demonstrating the programs functionality:



Screen shot demonstrating overall program functionality of the program as well as the ability of the system to handle more than one client at any given time. Route calculated for both connections uses R0, R2, and R3 to deliver packets between c1 and c2. R1 simply sent off its data to the controller when polled and performed no other action during the two transmissions.

Though the image may be complicated to understand, it is there simply to demonstrate the program working overall as an OpenFlow like system, being able to handle multiple clients and working as a grid. Its important to note that in this implementation the shortest route is calculated based on randomly assigned times for each connection between routers. This means that the route with the least amount of hops will not necessarily be the shortest route. That detail is particularly important when dealing with larger network grids.

Structure of the Implementation

The 'Main' Class

The Main class is responsible for handling system startup, the generation and instantiation of a grid of interconnected Routers, the instantiation of any Clients in the system and creation of the Controller. At startup, it creates the Setup window which was discussed in detail earlier. It was described earlier to set Global Variables necessary for the system to function the way the user wants and does just that by using the `readInt()` and `readString()` functions available in the Terminal class.

```
terminal.println("Enter in the width of the router grid: ");
GlobalVars.routerWidth = terminal.readInt();
terminal.println("Enter in the height of the router grid: ");
GlobalVars.routerHeight = terminal.readInt();
```

Example of code which sets variables located in the 'GlobalVars' class. This particular section of code handles the size of the grid.

When generating the grid of routers, the Main class follows the following protocols.

1. Routers possess 5 ports. These are represented as the left, the up, the right and the down ports with a port reserved purely for communications with the controller. These ports are stored, in that order, within an array. The value of each port number is calculated based on an offset calculated from the number of routers that have so far been placed in the grid.
2. Router ports located on the outer edge of the grid (for example, the up port on the top row of the grid) will be assigned the value of zero. For the purposes of this assignment, a port number of zero represents a lack of connection with another router or client.
3. Each time a router is instantiated, it is also passed in a routing table which details its surrounding connections.

Probably the most important factor here is the generation of a routing table which takes the form of an ArrayList of a custom made data type called *RoutingTableEntry*. There is a separate *RoutingTableEntry* for every connection surrounding the router. Each Routing Table Entry contains four important variables. The name of the station at the end point of the connection, the source port/socket to use when transmitting to the station, the destination port of the station and a randomized latency which is used by the controller for

calculating the shortest path through network. It's important to note that the system has been made in such a way that only one latency value exists between a single connection and not two different values. If this was not attended to, it would cause issues with the routing algorithm and would produce a different path for a given route going in the opposite direction. The code for assigning that is located in the Controller.

```
public static ArrayList<RoutingTableEntry> createRoutingTable(ArrayList<RoutingTableEntry> routingTableEntries,
                                                             int[] routerSrcPorts, int[] routerDstPorts, int routerCount,
                                                             int clientCount) {
    routingTableEntries = new ArrayList<RoutingTableEntry>();
    for (int i = 0; i < ROUTER_PORTS; i++) {
        if (routerDstPorts[i] != 0 && routerDstPorts[i] >= ROUTERS_START_PORT) {
            int routerNum = 0;
            if (i == 0) {
                routerNum = routerCount - 1;
            }
            else if (i == 1) {
                routerNum = routerCount - GlobalVars.routerWidth;
            }
            else if (i == 2) {
                routerNum = routerCount + 1;
            }
            else if (i == 3) {
                routerNum = routerCount + GlobalVars.routerWidth;
            }
            routingTableEntries.add(new RoutingTableEntry("R" + routerNum), routerSrcPorts[i], routerDstPorts[i]);
        }
        else if (routerDstPorts[i] >= CLIENTS_START_PORT && routerDstPorts[i] < CLIENTS_START_PORT + 5) {
            routingTableEntries.add(new RoutingTableEntry("C" + clientCount), routerSrcPorts[i], routerDstPorts[i]);
        }
        else if (routerDstPorts[i] == CONTROLLER_SRC_PORT) {
            routingTableEntries.add(new RoutingTableEntry("CNT"), routerSrcPorts[i], routerDstPorts[i]);
        }
        else if (routerDstPorts[i] == 0) {
            routingTableEntries.add(new RoutingTableEntry("0"), routerSrcPorts[i], routerDstPorts[i]);
        }
    }
    return routingTableEntries;
}
```

Code which produces routing tables for routers before they are instantiated. Located in Main.java

```
import java.util.Random;

public class RoutingTableEntry {

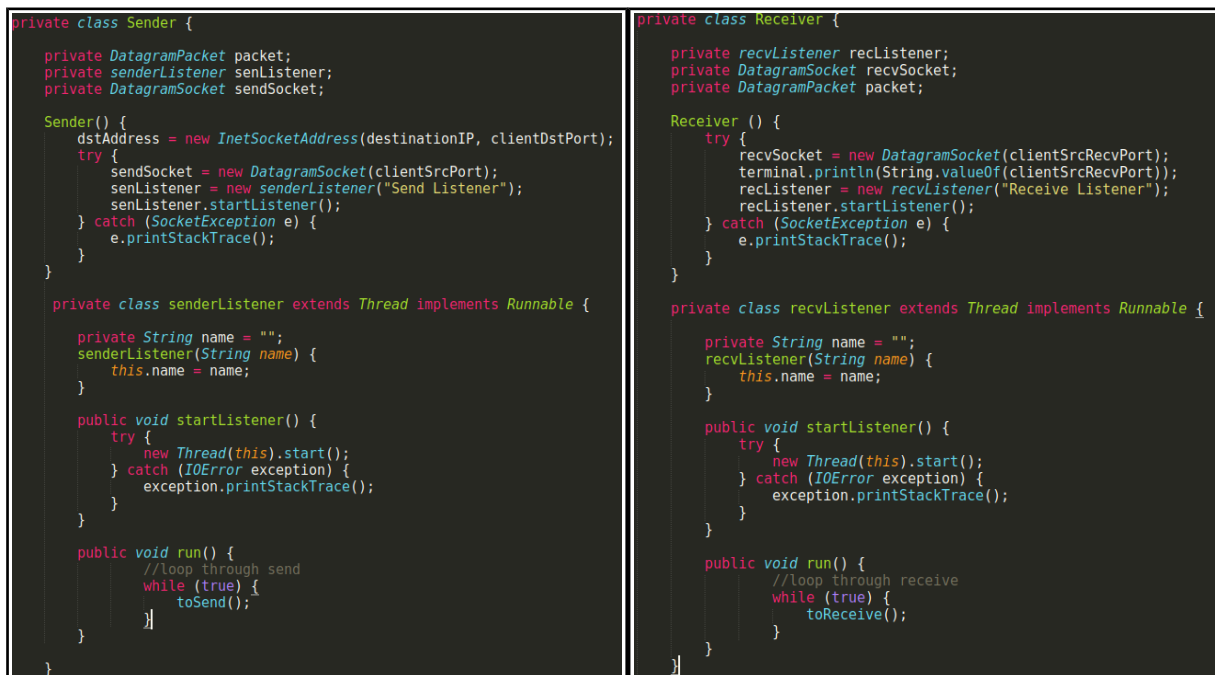
    private String entry;
    private int receivingPort;
    private int portToSendTo;
    private int linkLatency;
    private Random latencyGenerator;

    RoutingTableEntry (String entry, int receivingPort, int portToSendTo) {
        this.entry = entry;
        this.portToSendTo = portToSendTo;
        this.receivingPort = receivingPort;
        this.linkLatency = 0;
        if (entry.contains("R")) {
            latencyGenerator = new Random();
            this.linkLatency = (latencyGenerator.nextInt(10) + 1);
            this.linkLatency += 1;
        }
    }
}
```

RoutingTableEntry class. Seen here to generate random numbers between 2 and 10 to simulate latency between routers.

The ‘Client’ Class

The implementation of the client class is *to a degree* more trivial than the other classes in the program. It consists of two sockets. One for transmitting to it's connected router and one for receiving from the router. Each of these sockets are attached to their own thread and are separated into two sub classes of Client. Those are the Sender class and Receiver class. Both of these classes are instantiated in the Client constructor body and are attached to another thread. So, it can be said that the Client essentially runs on a single thread even though two new threads are created in Sender and Receiver. This structure allows the client to be constantly listening for transmissions on the receiving port as well as being able to send transmissions at any given time without the two processes interfering with one another. Beyond that, there isn't too much more to the Client class and works quite elegantly with the Sender and Receiver processes being able stream their outputs to the same terminal window despite belonging to different threads of execution.



```
private class Sender {
    private DatagramPacket packet;
    private senderListener senListener;
    private DatagramSocket sendSocket;

    Sender() {
        dstAddress = new InetSocketAddress(destinationIP, clientDstPort);
        try {
            sendSocket = new DatagramSocket(clientSrcPort);
            senListener = new senderListener("Send Listener");
            senListener.startListener();
        } catch (SocketException e) {
            e.printStackTrace();
        }
    }

    private class senderListener extends Thread implements Runnable {
        private String name = "";
        senderListener(String name) {
            this.name = name;
        }

        public void startListener() {
            try {
                new Thread(this).start();
            } catch (IOException exception) {
                exception.printStackTrace();
            }
        }

        public void run() {
            //loop through send
            while (true) {
                toSend();
            }
        }
    }
}

private class Receiver {
    private rcvListener rcvListener;
    private DatagramSocket rcvSocket;
    private DatagramPacket packet;

    Receiver () {
        try {
            rcvSocket = new DatagramSocket(clientSrcRecvPort);
            terminal.println(String.valueOf(clientSrcRecvPort));
            rcvListener = new rcvListener("Receive Listener");
            rcvListener.startListener();
        } catch (SocketException e) {
            e.printStackTrace();
        }
    }

    private class rcvListener extends Thread implements Runnable {
        private String name = "";
        rcvListener(String name) {
            this.name = name;
        }

        public void startListener() {
            try {
                new Thread(this).start();
            } catch (IOException exception) {
                exception.printStackTrace();
            }
        }

        public void run() {
            //loop through receive
            while (true) {
                toReceive();
            }
        }
    }
}
```

The subclasses of Client, Sender (on left) and Receiver (on right).

The ‘Router Class’

One of the *heftier* classes of the system, the router's main functionality is present in it's toSend() and toReceive() functions. Upon instantiation of a Router object in the Main class, the router creates a listener whose constructor can be found in the Abstract Class, Node (class Router inherits from Node). The object will then constantly loop through the two methods toSend() and toReceive().

Upon receipt of a packet, the router will output to the terminal where the packet has come from and then parse the packet's destination. The packets are encoded in the following fashion within their headers:

SourceName|DestinationName|-Payload

A typical Header within the system looks roughly like the following:

c1|c2|-hello

OR

R2|c2|-hello

The field source name constantly changes based on the current station that the packet is currently at whereas the destination name remains the same through the transmission. This allows the controller to keep track of the packet as it moves through the system on its journey to the destination. As said before, if the router has the destination stored in its *dstTableEntries* of type *dstTableEntry* (another custom made Data Type used only by the Router), it will simply forward the packet on to its next destination using the details stored in its destination database.

If however an entry for a certain destination is not stored, a boolean *needToSendQuery* is set to be true and the router will query the router for the next *hop*. This triggers the process shown in the Demonstration section of this report.

```

@Override
protected synchronized void toReceive() {
    packet = new DatagramPacket(new byte[PACKET_SIZE], PACKET_SIZE);
    for (int i = 0; i < sockets.length; i++) {
        trySocket(sockets[i]);
    }
}

private synchronized void trySocket(DatagramSocket sock) {
    String packetDestination = "";
    try {
        sock.receive(packet);
        packetDestination = getPacketDestination();
        if (GlobalVars.showRouterTerminals) {
            terminal.println("Received packet from: " + packet.getSocketAddress());
        }
        if (dstTableContainsDst(packetDestination)) {
            int index = 0;
            for (int i = 0; i < dstTableEntries.size(); i++) {
                if (dstTableEntries.get(i).getDstName().equals(packetDestination)) {
                    index = i;
                }
            }
            DstTableEntry entry = dstTableEntries.get(index);
            sendPacketOn(entry.getDstName(), getPacketPayload(), index, routingTable.get(entry.getSocketNum()).getPortToSendWith());
            //terminal.println("\nHave destination. Sending packet on to " + entry.getNextRouter() + "\n");
        }
        else if ((new String(packet.getData())).contains("DATAREQ")) {
            sendControllerRouterData();
        }
        else if ((new String(packet.getData())).contains("NEXTDST")) { //WHERE NEXTDST IS STORED INTO DST TABLE
            String nextDest = parseNextDest();
            String finalDest = getPacketDestination();
            if (GlobalVars.showRouterTerminals) {
                terminal.println("\nNEXTDST from CNT reading: " + nextDest);
            }
            for (int i = 0; i < routingTable.size(); i++) {
                if (routingTable.get(i).getEntry().equals(nextDest) && !dstTableContainsDst(nextDest)) {
                    dstTableEntries.add(new DstTableEntry(packetDestination, nextDest, findIndexInRoutingTable(nextDest)));
                    sendPacketOn(finalDest, payload, i, routingTable.get(i).getPortToSendWith());
                    packet = null;
                    i = routingTable.size();
                }
            }
        }
        else {
            needToSendQuery = true;
        }
    } catch (IOException e) {}
}

```

Code in Router.java which constantly iterates through an array of sockets and performs actions based on the contents of each packet.

The ‘Controller’ Class

The controller class is by far the most complex and its functionality centers on the jobs it performs in its toReceive() function. The Controller will only ever receive two types of packet in its life cycle. One packet being the DATAREQ packet and the other being the QRESP_ packet. The first of which being what a router sends to the controller when it does not have a stored destination for a certain packet. If the Controller possesses the route, it will immediately respond to the DSTQUERY with a NEXTDST packet. The QRESP_ packet is what a router sends to the Controller is the Controller has polled the router for its data (its routing table information).

As mentioned before if the Controller does not possess a network map, it will go on to generate one by polling all routers on the network for their routing tables and adds them to a 2D array of the Data Type RouterConnections (which is similar to the RoutingTableEntry data type except for the fact that the entire routing table of a router is stored in just one instance of a RouterConnections object). Whilst the map is being generated, the map is also converted into graph format with each router representing a vertex and

each link between routers being represented as an edge. This is all handled in the `generateNetworkMap()` function located in the Controller Class. Once the map has been generated the controller will go on to generate a route based on the source and destination nodes encoded within the originally received packet using Dijkstra's Algorithm.

```
protected synchronized void toReceive() {
    packet = new DatagramPacket(new byte[PACKET_SIZE], PACKET_SIZE);
    try {
        socket.receive(packet);
        terminal.println("Received packet from: " + packet.getSocketAddress());
    } catch (IOException e) {
        e.printStackTrace();
    }

    if (new String(packet.getData()).contains("DSTORY")) {
        if (!haveNetworkMap) { //GETS CALLED ON FIRST CLIENT TRANSMISSION TO CONTROLLER
            originalPacket = packet;
            terminal.println("CNT generating network map...");
            generateNetworkMap();
            packet = originalPacket;
            terminal.println("Map Generated.\n");
            haveNetworkMap = true;
        }
        if (!pathInList()) {
            terminal.println("GENERATING A PATH...");
            System.out.println(controllerName + ": Generating a path through the network...");
            ArrayList<String> newPath = generatePath(connectionsTables, getPacketSrc(), getPacketDestination());
            System.out.println(controllerName + ": Path found.");
            System.out.println(controllerName + ": New Path...");
            for (int i = 0; i < newPath.size(); i++)
                System.out.println(newPath.get(i));
            storedPaths.put(getPacketDestination(), newPath); //AAAAAAHHHHH
            System.out.println("\n");
        }
        if (pathInList()) {
            terminal.println("SENDING NEXT DST");
            sendResponse(packet.getPort());
        }
    }
}
```

`toReceive()` function which performs numerous roles such as generating routes if necessary, polling routers for data and sending on NEXTDST packets to routers.

Dijkstra's Algorithm

Dijkstra's algorithm is one which is certainly not too hard to conceptualize but is certainly not too easy to implement. For the purposes of this implementation the problem was dealt with as a *graph* problem as would commonly be encountered in the study of Algorithms and Data Structures. The main outline of the algorithm is that it searches for the shortest path from a single source to a certain destination. The algorithm has a list of settled nodes (nodes for which the shortest route has been found) and unsettled nodes (nodes for which the shortest route has *not* been found. For each step the algorithm takes through a graph of nodes, the next shortest *edge* or *link* is followed to a subsequent node until the destination is reached. Once the destination is reached, the node is added to the list of settled nodes and the process repeats until all nodes have been added to the list (or in our case, when the client has been added to the list of settled nodes. What becomes immediately obvious when implementing this in code is the need to

create a Data Type for the Graph which contains the lists of two other Data Types, Vertex (used for representing our nodes/routers/clients) and Edge (used for representing connections between nodes).

```

import java.util.List;

public class Graph {

    private List<Vertex> vertices;
    private List<Edge> edges;

    Graph (List<Vertex> vertices, List<Edge> edges) {
        this.vertices = vertices;
        this.edges = edges;
    }

    public List<Vertex> getVertices() {
        return vertices;
    }

    public List<Edge> getEdges() {
        return edges;
    }

}

public class Vertex {

    private String id;
    private String name;

    Vertex(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

}

```

Sample code from the Graph and Vertex classes. The Graph consists of two lists. One for the Vertices and one for the Edges.

The main loop of the Algorithm, located in DijkstraAlgo.java, is relatively simple. The first node encountered by the algorithm gets added to the list of unsettled nodes and next smallest hop is calculated in getMinimum(). The resulting node of getMinimum() is then added to the list of unsettled nodes while the originally used node is added to the settled nodes list. It is important to note that getMinimum() will return the node with the shortest link only if it is *not* already in the settled node list. It is this functionality that enables the algorithm to visit all possible nodes along a route to a certain destination and then compare them to find which was shortest in findMinimumDistance(). The algorithm will finish when there are no more unsettledNodes and all have been visited. When the path is found it can be returned to the Controller with generatePath() which pulls the path from the data generated in the execution of the algorithm.

```

DijkstraAlgo(Graph graph) {
    this.nodes = new ArrayList<Vertex>(graph.getVertices());
    this.edges = new ArrayList<Edge>(graph.getEdges());
}

public void executeAlgorithm(Vertex sourceNode, String dstClient) {
    this.dstClient = dstClient;
    settledNodes = new HashSet<Vertex>();
    unsettledNodes = new HashSet<Vertex>();
    distance = new HashMap<Vertex, Integer>();
    predecessors = new HashMap<Vertex, Vertex>();
    distance.put(sourceNode, 0); //Put source node in first and set its distance to zero
    unsettledNodes.add(sourceNode);
    while (unsettledNodes.size() > 0) {
        Vertex node = getMinimum(unsettledNodes); //Gets minimum node from unsettled nodes
        settledNodes.add(node); //adds node to settled nodes
        unsettledNodes.remove(node); //removes node from unsettled nodes
        findMinimumDistances(node); //Gets node with shortest distance
    } //Repeats until no unsettled nodes left
}

```

Main loop of DijkstraAlgo which executes the shortest path algorithm

The algorithm will also finish executing if it encounters the *dstClient* string while executing the *getMinimum()* function. The link between a router and a client (specifically our destination client) in this implementation is considered to be zero and therefore if the *dstClient* is encountered whilst executing the algorithm, the route will be stored for comparison at the end of the algorithm.

```
private void findMinimumDistances(Vertex node) {
    List<Vertex> adjacentNodes = getNeighbours(node);
    for (Vertex target: adjacentNodes) {
        if (target.getId().equals(dstClient)) {
            distance.put(target, getShortestDistance(node) + getDistance(node, target));
            predecessors.put(target, node);
            unsettledNodes.add(target);
        }
        else if (target.getId().contains("R") && getShortestDistance(target) > getShortestDistance(node) + getDistance(node, target)) {
            distance.put(target, getShortestDistance(node) + getDistance(node, target));
            predecessors.put(target, node);
            unsettledNodes.add(target);
        }
    }
}
```

If the target is encountered, the current 'tree' will end at the dstClient.

Intercepted WireShark Traffic

For practical reasons, traffic was intercepted on WireShark only for a 1-wide and 2-high grid of routers as following the traffic through any larger grids begins to take quite a long time. However the software is capable of running smoothly with any reasonably sized $m \times n$ grid. The results from the WireShark investigation were very rewarding and are a very nice example of how efficient the transfer of packets becomes when the controller only has to be queried once for a NEXTDST. On the next page there is the sample 1 x 2 network of routers with a transmission sent from c2 to c1 followed by the traffic intercepted by WireShark. We can immediately see that from the moment the first transmission is sent from c2 (with sender port at 2051), R1 sends a query to the Controller named CNT. The controller then polls every router in the system for data. Each poll and response can be seen in the traffic report from packets 3 to 6. Packet number 7 then, is a NEXTDST from the Controller to R1 after which the router clearly sends the packet onward to R0.

We then see R0 sending a query to the Controller for the data. However seeing as the route has just been generated the Controller only needs to inspect its stored routes and simply send on the next station based on the name of the router who sent the query. This can be seen from traffic entries 9 and 10. And finally in entry 11, the packet reaches its destination.



Sample input for capturing traffic on WireShark. Transmission from c2 to c1 of "hello".

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	55	2051 → 40005 Len=13
2	0.128537095	127.0.0.1	127.0.0.1	DIS	55	PDUType: 124 ...
3	0.130863880	127.0.0.1	127.0.0.1	DIS	57	PDUType: 84 ...
4	0.181116468	127.0.0.1	127.0.0.1	DIS	135	PDUType: 124 ...
5	0.186687320	127.0.0.1	127.0.0.1	DIS	57	PDUType: 84 ...
6	0.229850626	127.0.0.1	127.0.0.1	DIS	139	PDUType: 124 ...
7	0.268496497	127.0.0.1	127.0.0.1	DIS	61	PDUType: 84 ...
8	0.296785286	127.0.0.1	127.0.0.1	UDP	55	40006 → 40003 Len=13
9	0.341880437	127.0.0.1	127.0.0.1	DIS	55	PDUType: 124 ...
10	0.353528811	127.0.0.1	127.0.0.1	DIS	61	PDUType: 84 ...
11	0.392247384	127.0.0.1	127.0.0.1	UDP	55	40000 → 2000 Len=13

Intercepted traffic from the above sample input on WireShark. 11 transmissions in total.

What is very interesting and satisfying given the above results is that if another transmission is sent from c1 to c2 is that the traffic report is significantly smaller as the routers need only follow the order in which they sent the packet on beforehand. This proves that the designed system implements the original specification of routers only having to query the Controller if they have not sent a packet on to a destination before. The results along with sample input are shown below:

The image displays four terminal windows arranged in a 2x2 grid, showing the logs of a network simulation. The top-left window is for client 'c1', the top-right for router 'R0', the bottom-left for client 'c2', and the bottom-right for router 'R1'. The logs show the flow of packets and queries between these entities, including destination client identification and routing table lookups.

```

c1
2000
Enter in payload to deliver...

c1: packet received from /127.0.0.1 40000
c1: packet reads hello

Enter in payload to deliver...

c1: packet received from /127.0.0.1 40000
c1: packet reads you muppet

Enter in payload to deliver...

R0
Received packet from: /127.0.0.1:3000
Sending CNT routing table for...R0

R0: SENT ROUTER DATA TO localhost/127.0.0.1:3000
R0: Destination client is c1
Received packet from: /127.0.0.1:40006
R0: Destination client is c1
No route for Client c1. Querying Controller...

R0: SENT QUERY TO localhost/127.0.0.1:3000
R0: Destination client is c1
Received packet from: /127.0.0.1:3000
R0: Destination client is c1

NEXTDST from CNT reading: c1
R0: SENT TO localhost/127.0.0.1:2000
R0: Destination client is c1
Received packet from: /127.0.0.1:40005
R0: SENT TO localhost/127.0.0.1:2000

c2
Enter in destination in the following format:
Cx where 'C' is a client on the network and
'x' is the positive integer ID of the client.
Examples: C3, C12, C0 etc.

c1
c2: SENT PACKET TO localhost/127.0.0.1:40005

Enter in payload to deliver...
you muppet
Enter in destination in the following format:
Cx where 'C' is a client on the network and
'x' is the positive integer ID of the client.
Examples: C3, C12, C0 etc.

c1
c2: SENT PACKET TO localhost/127.0.0.1:40005

Enter in payload to deliver...

R1
Received packet from: /127.0.0.1:2051
R1: Destination client is c1
No route for Client c1. Querying Controller...

R1: SENT QUERY TO localhost/127.0.0.1:3000
R1: Destination client is R1
Received packet from: /127.0.0.1:3000
Sending CNT routing table for...R1

R1: SENT ROUTER DATA TO localhost/127.0.0.1:3000
R1: Destination client is c1
Received packet from: /127.0.0.1:3000
R1: Destination client is c1

NEXTDST from CNT reading: R0
R1: SENT TO localhost/127.0.0.1:40003
R1: Destination client is c1
Received packet from: /127.0.0.1:2051
R1: SENT TO localhost/127.0.0.1:40003
    
```

Sample input for second transmission.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	60	2051 → 40005 Len=18
2	0.011714663	127.0.0.1	127.0.0.1	UDP	60	40005 → 40003 Len=18
3	0.079198555	127.0.0.1	127.0.0.1	UDP	60	40000 → 2000 Len=18

Significantly smaller table of intercepted traffic on WireShark for the second transmission from c2 to c1.

Conclusion: Assessment of Strengths, Weaknesses and Implementation Complexity

A very challenging assignment this was indeed. For certain the most challenging part was the implementation of Dijkstra's Algorithm and required lots of research, trial and error, toiling through various series of lectures, documentation and perseverance. I think I completely redesigned the routing algorithm at least three times before I was happy with how it functioned. I chose to use Dijkstra's Algorithm for routing mostly because it has the greatest relevance in the real world as routing algorithms go. It is used in everything from artificial intelligence in video games to how OpenFlow system actually route their data. So it is definitely good to be more than familiar with the algorithm and I most definitely am after this assignment. The routing protocol I have designed is strong in that it will work and any $m \times n$ network of routers. I even tried it with 200×200 before (though it took quite a while to generate the network map). However a weakness lies in how the Controller stores the routes in its database. The routes are stored in a Hash Map of ArrayLists in Controller.java but the keys used to search for a route in the Hash Map are referred to only by the destination client as supposed to the source client *and* the destination client. This does not effect back and forth communication between two end users. In fact four end users can operate on the same network properly as long as each pair communicate only with themselves. However, the following situation will crash the controller:

1. c1, c2, c3 and c4 are all initialized.
2. c1 sends a transmission to c2, route saved as c2 in controller.
3. c2 sends a transmission to c1, route saved as c1 in controller.
4. c3 sends transmission to c1 and then c3 tries to send a transmission to c2, when the controller is queried it breaks as it looks at the routes for c1 or c2 and cannot find the next station to send to as it was generated in the context of communication between c1 and c2. The Controller stalls indefinitely and holds up transmission.

This flaw in the design could be fixed by either encoding the source node into the search key for the Hash Map or by having a number of possible routes being stored in the entries value field as supposed to a

single path through the network. But alas, I have run out of time to implement this and have in my folly constrained the system in such a way that it would take quite a while to change.

Other weaknesses of the implementation would involve the lack of flow control or error detection within the system. However, I feel that the intent of the assignment was more focused on learning about the architecture of larger networks and how routing decisions are made. Therefore I chose not to include this feature in the implementation. What the implementation loses in this, it makes up for in the modularity of its routers and the smoothness with which the routing algorithm operates with in the Controller (with the exception of course of the flaw in how the paths are stored in the Controller). Overall, this assignment has been incredibly challenging but not without reason as it has opened my eyes to the actual complexity of real world network routing and design and has in fact only driven the interest I have for this module to even greater heights.