Patrick Geoghegan - 1332 0590
Conor O'Flanagan - 1332 3109
Sofwat Islam - 1331 8593

CS2031 - Telecommunications
Assignment 2

Protocol Design for Distributed Applications
Routing Protocol for Disaster Zones

Distance Vector Routing:

Abstract Explanation of Distance Vector

"Distance" is the cost of reaching a destination. The value is usually based on the number of routers (laptops) the message passes through, or the total sum of all the metrics assigned to the links in the path (e.g. the amount of time it takes to travel between routers).

The "Vector" is the path of routers that messages travel through as they travel from one end node to another.

This dynamic routing protocol initiates when each node sends a message to all its immediate known neighbours. These nodes contain an empty routing table which will store the next node they should take to reach its final destination.

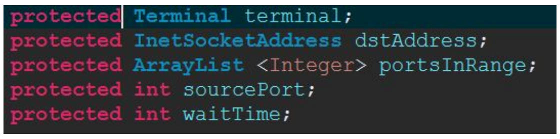
Each node will respond to the node it received the message from (the message source), storing the distance. The node then forwards a message to its neighbours excluding the neighbour node that have already received this particular message. This process continues until all the messages reach an endpoint and then return to their original senders so that the routing tables can be built.

When building the routing table, each node will add the information and the node that will result in the shortest distance. The time taken to complete these initial steps is know as convergence. Following this, if links fail or metrics change, periodic updates in the form of a partial or complete routing table are sent throughout the network so nodes can maintain up-to-date routing tables.

Description of the Implementation

Since this network was of a fixed topology, periodic routing messages are not sent after the initial setup. Our implementation began with the construction of classes to represent the necessary type of nodes (Phones or Laptops). These devices are children of Device class and inherit traits, such as being an extension of "Node" and having an "onReceipt" method.

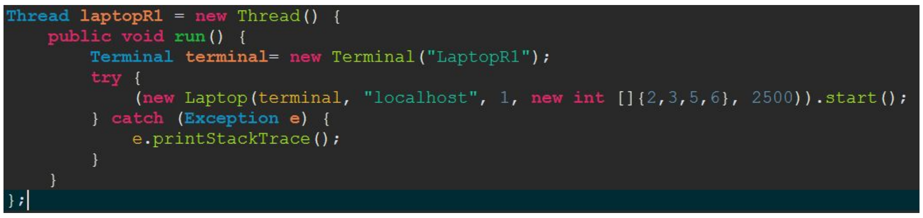
This implementation starts by creating 8 threads (4 phones and 4 laptops) and assigning ports 5001, 5002, 5003, 5004 to the Laptops and 5005, 5006, 5007, 5008 to the Phones.



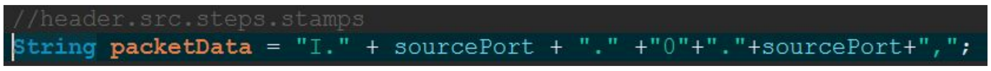
```
protected Terminal terminal;
protected InetSocketAddress dstAddress;
protected ArrayList <Integer> portsInRange;
protected int sourcePort;
protected int waitTime;
```

Each device contains an array of only its immediate neighbours (portsInRange in Device class). Our layout of nodes and neighbours are based off the system depicted in the assignment sheet. These threads are started in the Main class.

Here is an example of one of the devices instantiation. The last argument (in this case "2500") is a delay time each thread waits for to avoid concurrency related issues at the start of the program. Each thread is given a slightly different time. Once all the threads have begun the laptops will send to its neighbours a message, which will in turn cause a response message and forwarding messages depending where it is in the network and if the message hasn't already been to another node. Below is the format of the message.



```
Thread laptopR1 = new Thread() {
    public void run() {
        Terminal terminal= new Terminal("LaptopR1");
        try {
            (new Laptop(terminal, "localhost", 1, new int []{2,3,5,6}, 2500)).start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
};
```



```
//header.src.steps.stamps
String packetData = "I." + sourcePort + "." +"0"+"."+sourcePort+",";
```

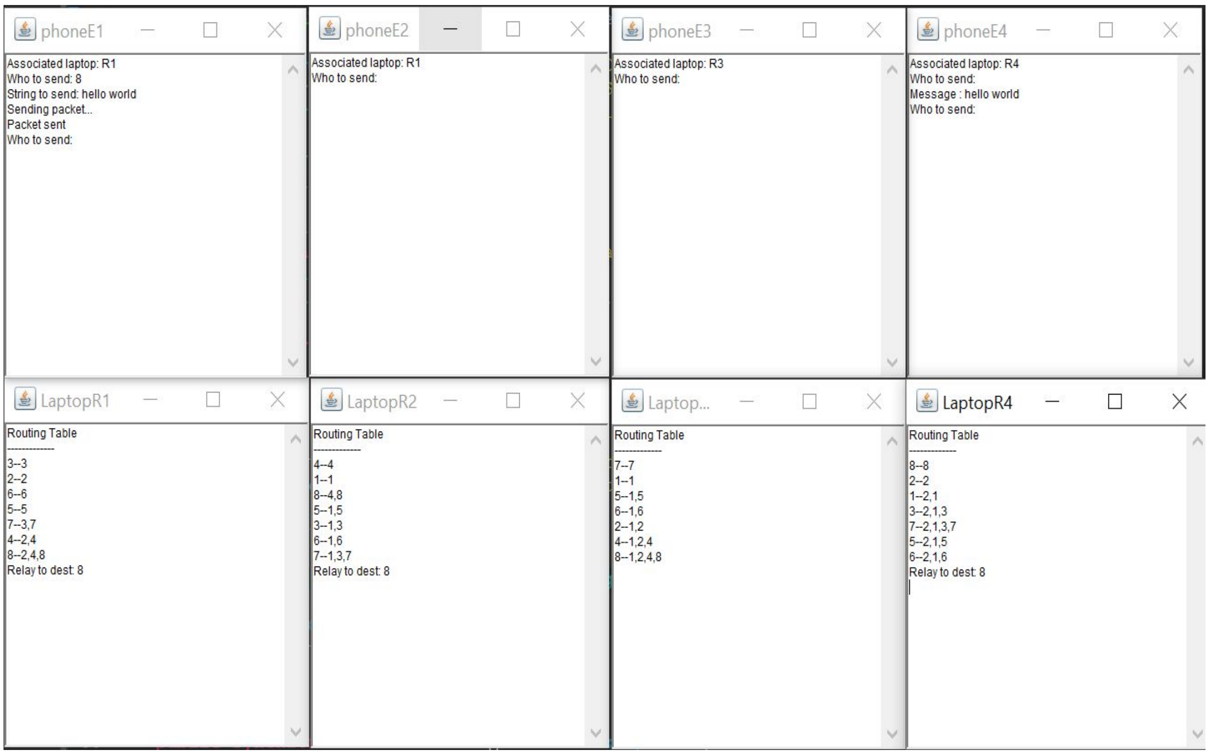
The header denotes whether it's going back to the source ("R") or continuing forward ("I"). This is followed by the source port of the message (where it came from), the steps the message has gone from the source port (the number of ports the message has passed) and the end which will have a sequence of ports or "stamps" of where the message has been (starting with the source port), so it can find its way back to the source and return the distance.

The "onReceipt" message does most of the work by breaking up the packet into its pieces and constructing a response to send back to the source by changing the header and stamping the message. (The original message is also stamped and a "hop" or step is added.) This is then sent to the other neighbour nodes.

The "onReciept" will also decide whether to process the message into its routing table. This happens when the header is "R" and the message "sourcePort" matches the node port (i.e. a message has returned to the port that it started at). If the given message has a shorter path than one originally defined in the table, it is added, replacing the information originally defined in the table. However, in this particular topology "shorter" paths are not a concern because there is only one way to reach each node.

Since the ports are numbered 1-8, we also used them as an index to an arraylist of strings representing the routing table. This is convenient in the construction of the routing tables. The "distance" to a node is essentially decided by the message that comes back with the least number of hops.

Below we can see the 4 laptops with their constructed routing tables. Each table has 7 paths. We can see an example of a message (“hello world”) being sent from E1 (Port 5) on the top left to E4 (Port 8) on the top right, through the appropriate laptops (R1 -> R2 -> R4). The “phoneRelay” method in the “Laptop” class checks the destination of messages and checks the routing table for the next node to forward to, while also printing the relay event to the terminal. This is consistent with the assignment’s given topology. In the implementation each phone is only linked to one laptop. This is printed to its terminal.



```
Associated laptop: R1
Who to send: 8
String to send: hello world
Sending packet...
Packet sent
Who to send:
```

```
Associated laptop: R1
Who to send:
```

```
Associated laptop: R3
Who to send:
```

```
Associated laptop: R4
Who to send:
Message : hello world
Who to send:
```

```
Routing Table
-------------
3--3
2--2
6--6
5--5
7--3,7
4--2,4
8--2,4,8
Relay to dest: 8
```

```
Routing Table
-------------
4--4
1--1
8--4,8
5--1,5
3--1,3
6--1,6
7--1,3,7
Relay to dest: 8
```

```
Routing Table
-------------
7--7
1--1
5--1,5
6--1,6
2--1,2
4--1,2,4
8--1,2,4,8
```

```
Routing Table
-------------
8--8
2--2
1--2,1
3--2,1,3
7--2,1,3,7
5--2,1,5
6--2,1,6
Relay to dest: 8
```

Advantages

The advantages of Distance Vector is that approach is simpler and easy to implement. Moreover, it does not demand high bandwidth level to send their periodic updates as the size of the packets are relatively small. This protocol will also dynamically choose routes better, given all the metrics are up to date. This is better suited for smaller networks.

Disadvantages

The main drawbacks of Distance Vector are limited scalability due to slow convergence time, bandwidth consumption and routing loops. Loops cause problems such as “the count to infinity problem”. Some methods are used for avoiding loops such as “split horizon” with or without “poisoned reverse”. Since a node only knows its neighbour it can only check the shortest distance at a given point but not the “better” route (e.g. less congested) unlike link state.

Link State Routing:

<u>Abstract Explanation of Link State Routing</u>

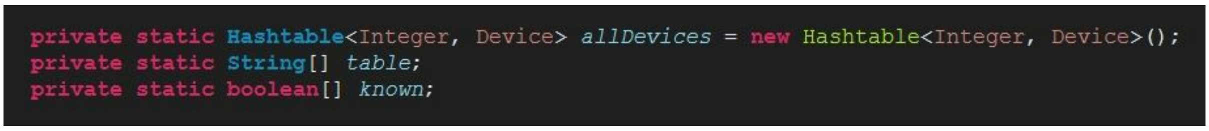
Link State Routing creates a topology of a network using an algorithm known as Dijkstra's algorithm (or Shortest Path First). The algorithm will find the shortest path between nodes based on the "cost" of travelling in between the source and destination. As such, the "shortest" path may not be the one with fewest nodes, but the one with the least "cost" to travel between all the nodes in the path.

Since Link State Routing shares some basic common traits with the Distance Vector protocol the Link State implementation is an extension of the Distance Vector implementation mentioned above.

The ports were renamed from 1-8 to 5001-5008 to avoid some UNIX compatibility errors. However, we still use these as indexes for the ArrayLists by subtracting 5000 from them. The variable named PORT_OFFSET does the conversion.

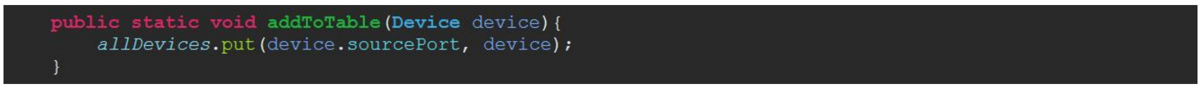
<u>Description of the Implementation</u>

Dijkstra's Algorithm is carried out in the Dijkstra class. The class contains a hashtable containing each device (with the device sourcePort being the key to the table), a String array which will represent the output table, and a boolean array to keep track of which nodes have been checked in the algorithm (or visited by the algorithm).



```
private static Hashtable<Integer, Device> allDevices = new Hashtable<Integer, Device>();
private static String[] table;
private static boolean[] known;
```

The "addToTable" method simply adds a device to the hashtable so it can be included in the algorithm. Each device calls this method in the Device constructor.

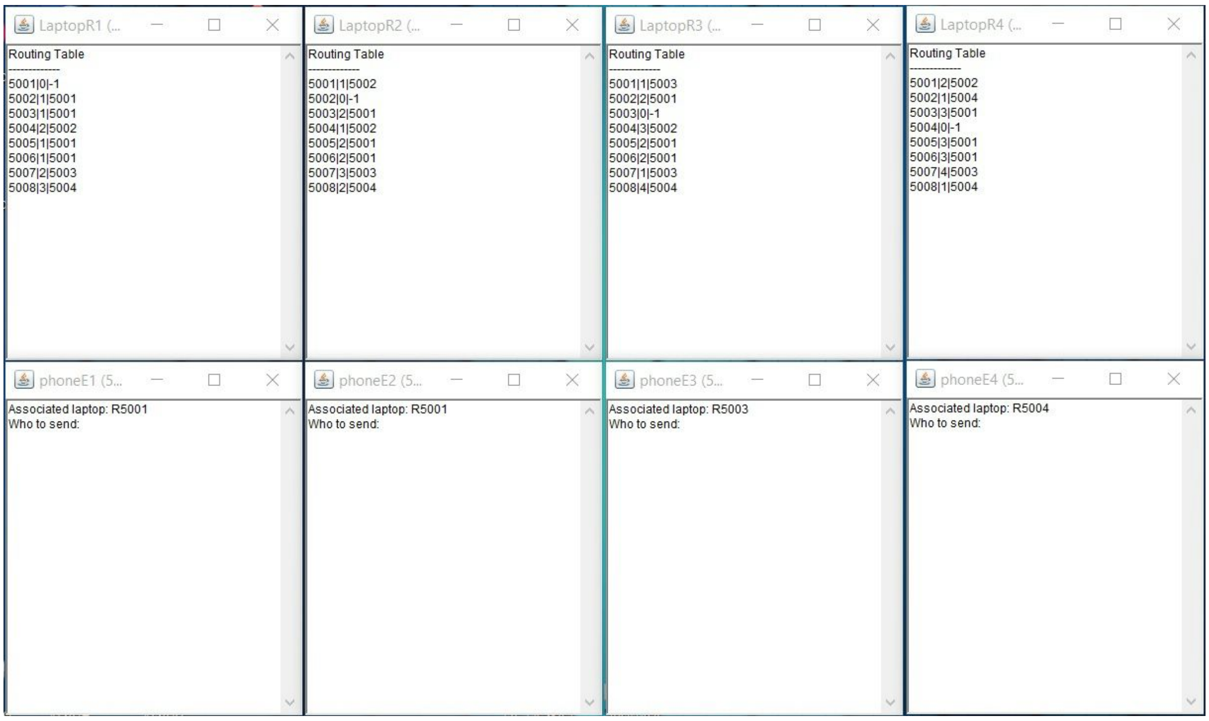


```
    public static void addToTable(Device device){
        allDevices.put(device.sourcePort, device);
    }
```

The constructTable function clears all previous 'table' Strings and 'known' booleans so it can build a Dijkstra table from scratch. After Dijkstra's algorithm has completed the table, this function returns the table in String form. This function takes in a device as a starting point for Dijkstra's algorithm so it can give the node its relevant table from the typology.

The applyDijkstra method takes in the current node and the count. All the neighbour nodes are checked, and the "costs" of travelling from the current node to each neighbour node are retrieved. These values are compared to the current values in the Dijkstra table. If a new value is less than a corresponding current value, it is placed into the table, overwriting the current value. If there is no current value for the neighbour node, the value is placed into the table as well. This process repeats with a neighbour node (as long as the neighbour node has not already gone through the process). This function uses the getCost function to extract the cost value from a String in the Dijkstra table.

The results can be seen below:



```
LaptopR1 (...
Routing Table
-------------
5001|0|-1
5002|1|5001
5003|1|5001
5004|2|5002
5005|1|5001
5006|1|5001
5007|2|5003
5008|3|5004
```

```
LaptopR2 (...
Routing Table
-------------
5001|1|5002
5002|0|-1
5003|2|5001
5004|1|5002
5005|2|5001
5006|2|5001
5007|3|5003
5008|2|5004
```

```
LaptopR3 (...
Routing Table
-------------
5001|1|5003
5002|2|5001
5003|0|-1
5004|3|5002
5005|2|5001
5006|2|5001
5007|1|5003
5008|4|5004
```

```
LaptopR4 (...
Routing Table
-------------
5001|2|5002
5002|1|5004
5003|3|5001
5004|0|-1
5005|3|5001
5006|3|5001
5007|4|5003
5008|1|5004
```

```
phoneE1 (5...
Associated laptop: R5001
Who to send:
```

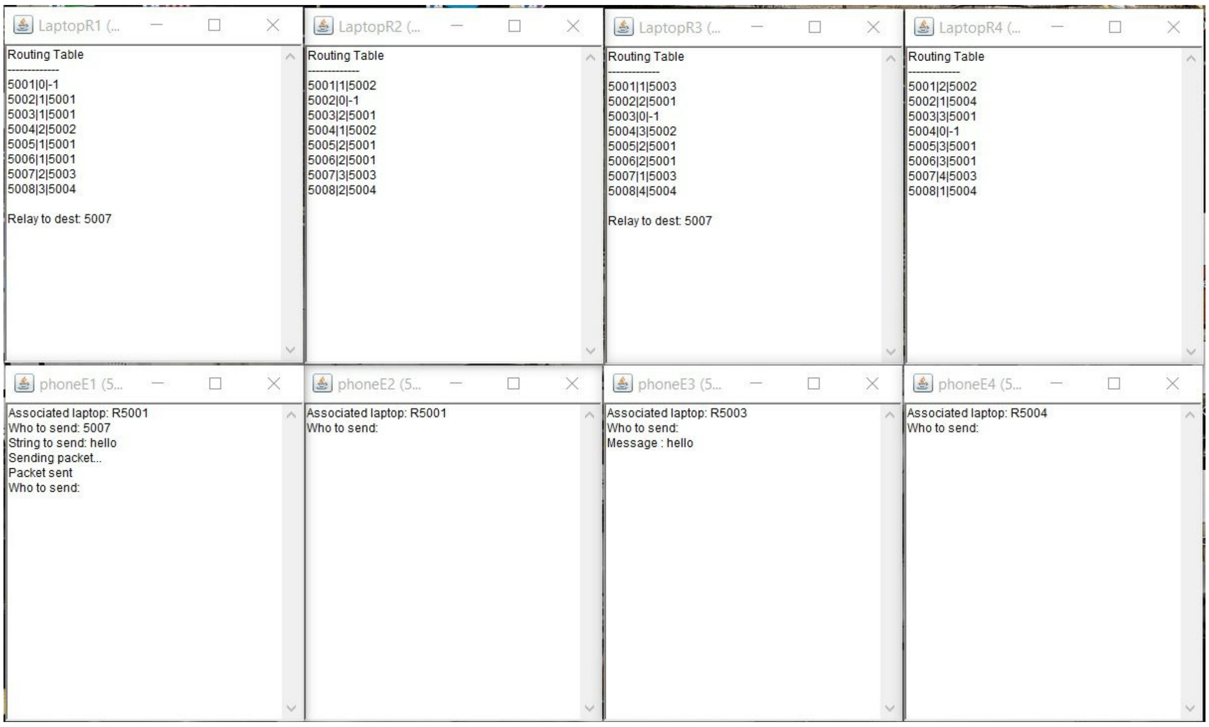
```
phoneE2 (5...
Associated laptop: R5001
Who to send:
```

```
phoneE3 (5...
Associated laptop: R5003
Who to send:
```

```
phoneE4 (5...
Associated laptop: R5004
Who to send:
```

The Dijkstra (Link State) tables for laptops numbered R1(port 5001) to R4(port 5004) can be seen in the top panels.

For example:
If we send a “hello” message from E1 (port 5005) to phone E3 (port 5007), using these constructed tables the following executes which is what we expect from the typology given.



```
LaptopR1 (...
Routing Table
-------------
5001|0|-1
5002|1|5001
5003|1|5001
5004|2|5002
5005|1|5001
5006|1|5001
5007|2|5003
5008|3|5004

Relay to dest: 5007
```

```
LaptopR2 (...
Routing Table
-------------
5001|1|5002
5002|0|-1
5003|2|5001
5004|1|5002
5005|2|5001
5006|2|5001
5007|3|5003
5008|2|5004
```

```
LaptopR3 (...
Routing Table
-------------
5001|1|5003
5002|2|5001
5003|0|-1
5004|3|5002
5005|2|5001
5006|2|5001
5007|1|5003
5008|4|5004

Relay to dest: 5007
```

```
LaptopR4 (...
Routing Table
-------------
5001|2|5002
5002|1|5004
5003|3|5001
5004|0|-1
5005|3|5001
5006|3|5001
5007|4|5003
5008|1|5004
```

```
phoneE1 (5...
Associated laptop: R5001
Who to send: 5007
String to send: hello
Sending packet...
Packet sent
Who to send:
```

```
phoneE2 (5...
Associated laptop: R5001
Who to send:
```

```
phoneE3 (5...
Associated laptop: R5003
Who to send:
Message : hello
```

```
phoneE4 (5...
Associated laptop: R5004
Who to send:
```

Advantages

One of the benefits of using the Link State protocol is the quick network convergence. Link State responds immediately for any networks changes by employing triggered flooded updates, which report the changes once they occur
This isn’t as important in this set typology scenario as convergence will occur once as we can assume there will be no errors. The complete and synchronized view for each protocol about the network makes it hard for routing loops to exist unlike Distance Vector. Additionally, link state protocols consume less bandwidth compared to Distance Vector, as it does not engage periodic updates while maintaining the network. Likewise, the bandwidth efficiency has an impact on network scalability, making Link State more advantageous for larger and complex networks.

Disadvantages

In contrast to Distance Vector, the extensive use of databases such as neighbour table (portsInRange) and Link State tables, in addition to the routing table require substantial amount of memory, moreover, it requires significant CPU power to run Dijkstra’s algorithm especially in large networks. Finally, it is quite complex to implement in comparison to Distance Vector a problems are harder to fix when they occur.