Telecommunications Assignment 2

REPORT

Niall Hunt

16319138
15/12/2017

Contents

Intro

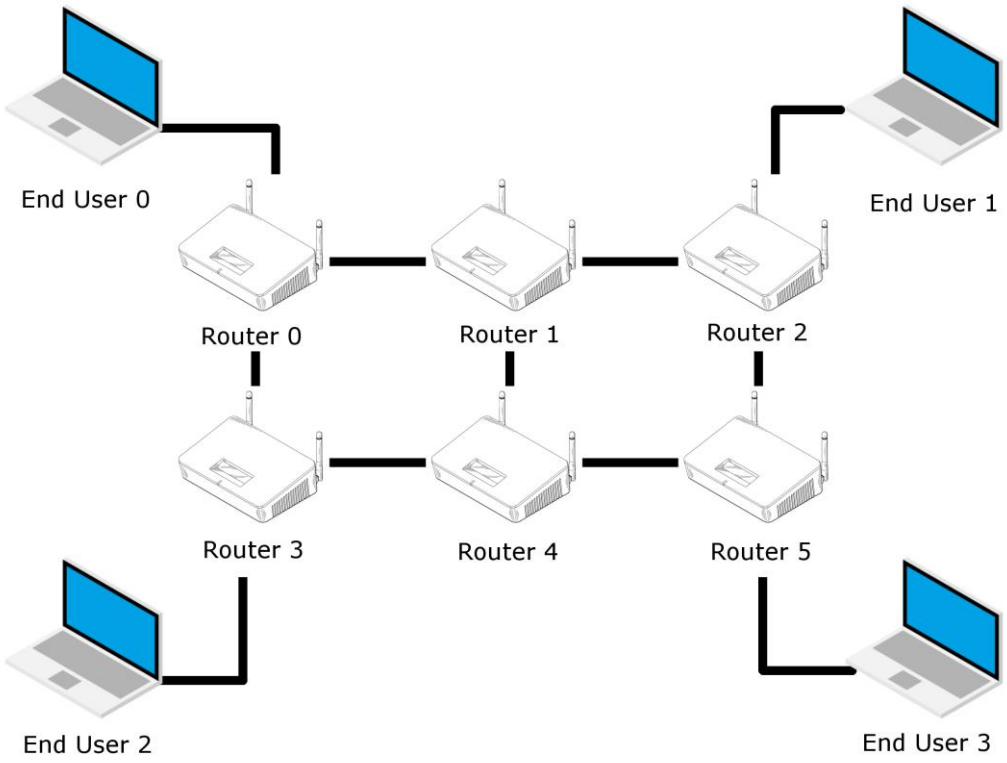
Introduction

Assignment

We were tasked with developing an open flow like controller. To start this assignment, we were to hard-code the controller with rules based on the known network. From there we were asked to implement link state routing or distance vector routing. The controller would be asked where to send a packet if a router wasn't sure where it was to go. The controller would then inform all routers along the path where the packet is to be sent. The program was to allow an end user to send a packet to another end user through the routers.

My Implementation

I created an Assignment2Main class that creates and runs the end users (called clients in my program) and the routers. These are all hard-coded with information provided as final ints. The controller is run separately first. This creates the network that can be seen below.



Introduction

Intro

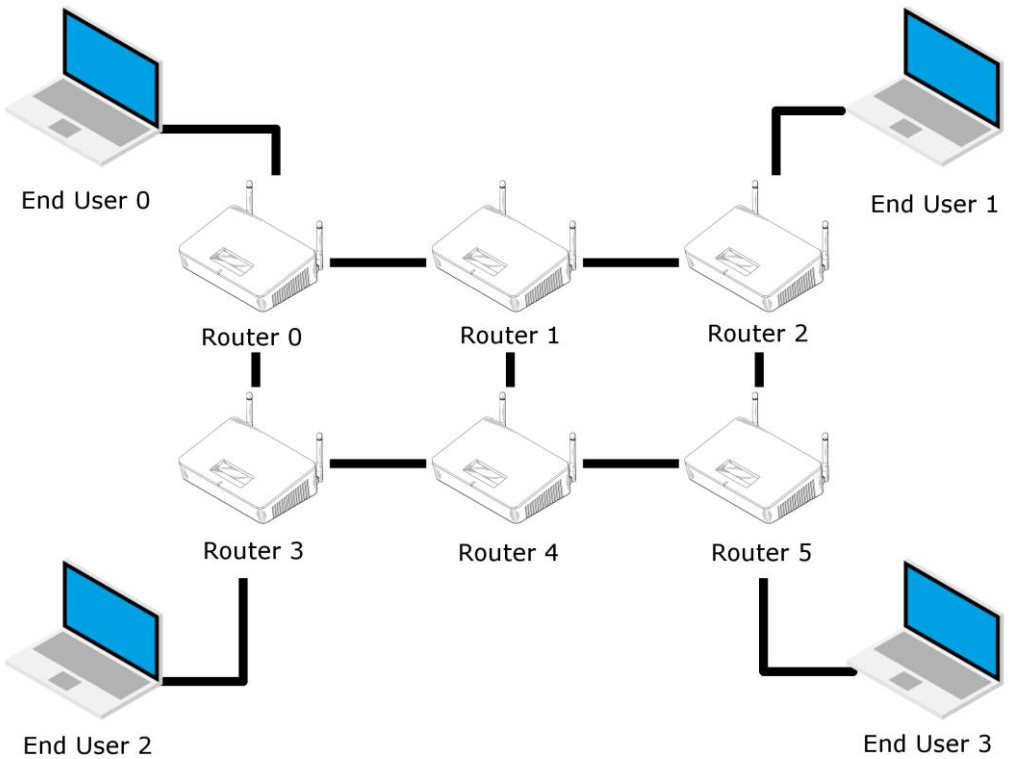
This simplistic network is used as an overview and example of a proper network. Each router in my implementation is connected to 3 other nodes. Each end user is connected to a specific router. So, for a packet to be sent to an end user it will have to go through the router it is connected to as part of its path.

Constants

Constants

Below I have some constants that are used through out my program and my implementation. I also provided a second picture of the network, so it can be used in conjunction with the table to understand the network.

Node	Port Number	Connection 1	Connection 2	Connection 3
End User 0	40000	Router 0	—	—
End User 1	40001	Router 2	—	—
End User 2	40002	Router 3	—	—
End User 3	40003	Router 5	—	—
Router 0	50000	End User 0	Router 1	Router 3
Router 1	50001	Router 0	Router 2	Router 4
Router 2	50002	Router 1	Router 5	End User 1
Router 3	50003	End User 2	Router 0	Router 4
Router 4	50004	Router 3	Router 1	Router 5
Router 5	50005	Router 4	Router 3	End User 3
Controller	60001	—	—	—



Part 1

Part 1: Hard-Coded Controller

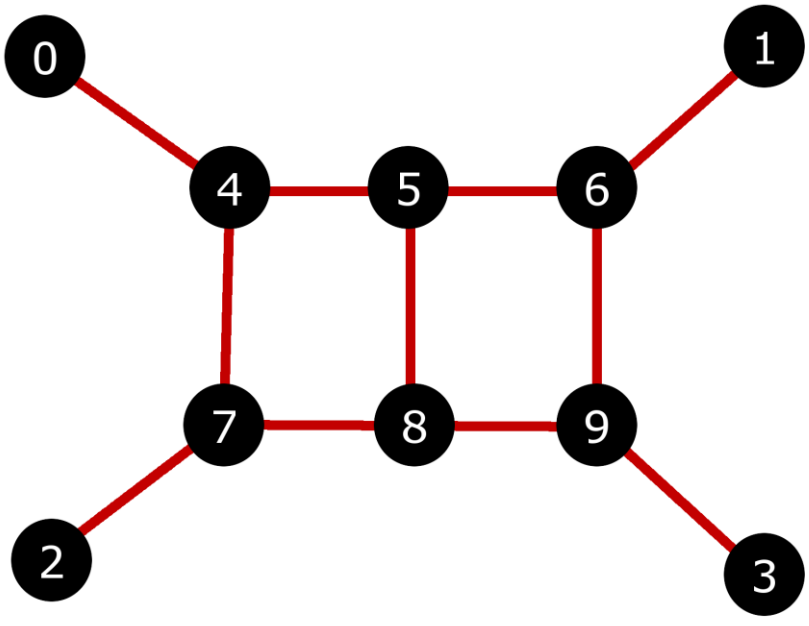
Explanation

The first part of this assignment was to implement a controller with preconfigured routing information. After a packet is sent to the controller there should be no need for the subsequent routers to ask the controller for the routing information. Doing it this way is only feasible for small networks such as this one. It's not possible for larger scale networks with different levels of routers this is where link state routing and distance vector routing protocols come in.

My implementation

In my implementation I use a graph to represent the network. The graph exists in the controller and is preconfigured for this exact network. Each vertex of the graph represents a node. Each edge represents a connection between the nodes. I used the graph class provided by Princeton University at https://algs4.cs.princeton.edu/41graph/Graph.java.html. Also provided is a BreadthFirstPath class which allows me to calculate the shortest path between 2 vertices. I use this to calculate the path for my packet.

When a router receives a packet, it checks the known path part of its header. This will equal zero if the path is unknown. In this case the packet is sent to the controller. The controller decapsulates the header and uses the destination port and source port to find the best path through the network. The ports of the network are kept in an array with each entry corresponding to a node and a vertex in the graph.



Part 1: Hard-Coded Controller

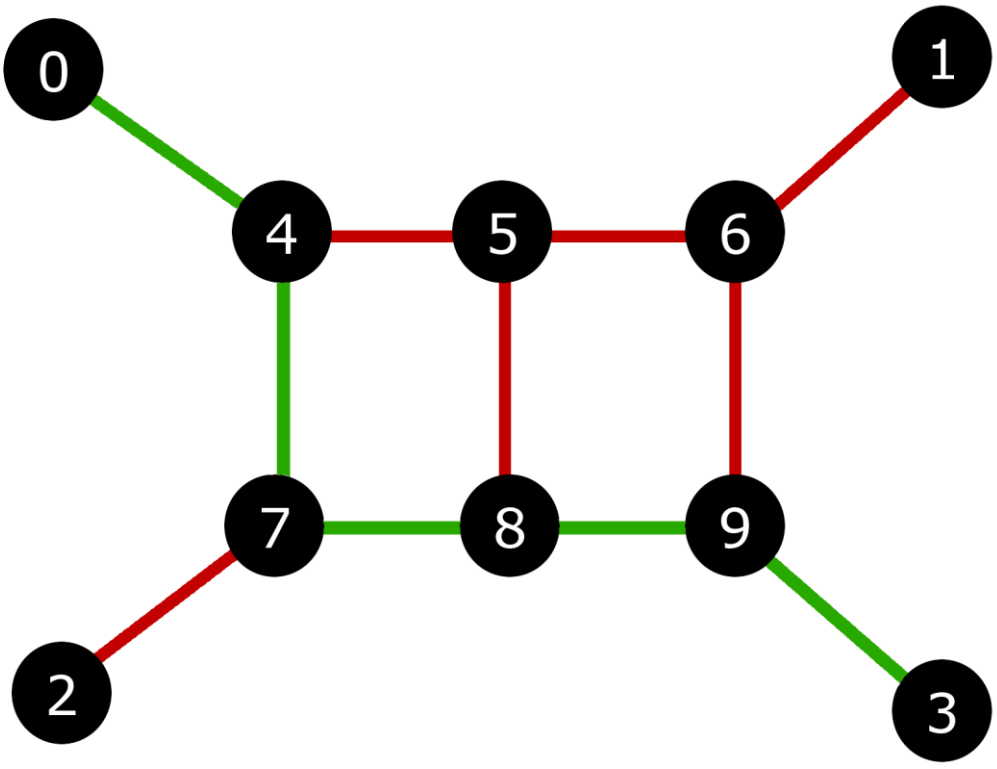
The port array = {40000, 40001, 40002, 40003, 50000, 50001, 50002, 50003, 50004, 50005}

Therefore, I can binary search my port array for the source port and the destination port and this will give me their index in the array. This index is how I refer to the corresponding vertices in the graph.

So, the path from 0 -> 3 (End User 0 -> End User 3) = 0 -> 4 -> 7 -> 8 -> 9 -> 3. This corresponds to End User 0 -> Router 0 -> Router 3 -> Router 4 -> Router 5 -> End User 3.

My shortest path function returns an Iterable<Integer> of the path. I use this to iterate through the path to create a new pathPort array. The pathPort array is then used to encapsulate a header onto the packet for each step in the path. The new header has a known port value of one meaning that the router will not have to send it to the controller to know where to send it on to.

The packet which is now encapsulated with a header for each step in its journey is sent back to the router from which it came. The router receives the packet and knows the route it needs to take. It decapsulates its header and sends it on to the next node.



Part 1: Hard-Coded Controller

Each of my routers has a terminal where they inform the person running the program that a packet has been received or sent.

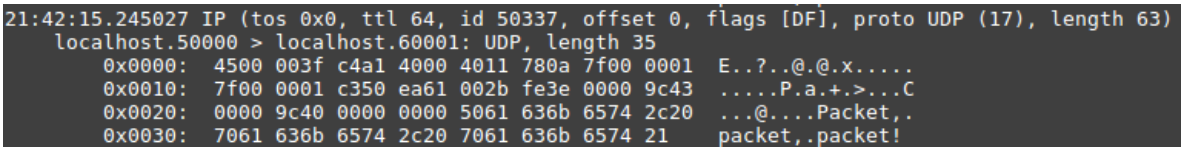
Captured Packets

```
niall@niall-laptop ~ $ sudo tcpdump -v -X -i lo
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
21:42:15.242182 IP (tos 0x0, ttl 64, id 50336, offset 0, flags [DF], proto UDP (17), length 63)
    localhost.40000 > localhost.50000: UDP, length 35
        0x0000:  4500 003f c4a0 4000 4011 780b 7f00 0001  E..?..@.@.x.....
        0x0010:  7f00 0001 9c40 c350 002b fe3e 0000 9c43  .....@.P.+.>...C
        0x0020:  0000 9c40 0000 0000 5061 636b 6574 2c20  ...@....Packet,.
        0x0030:  7061 636b 6574 2c20 7061 636b 6574 21    packet,.packet!
21:42:15.245027 IP (tos 0x0, ttl 64, id 50337, offset 0, flags [DF], proto UDP (17), length 63)
    localhost.50000 > localhost.60001: UDP, length 35
        0x0000:  4500 003f c4a1 4000 4011 780a 7f00 0001  E..?..@.@.x.....
        0x0010:  7f00 0001 c350 ea61 002b fe3e 0000 9c43  .....P.a.+.>...C
        0x0020:  0000 9c40 0000 0000 5061 636b 6574 2c20  ...@....Packet,.
        0x0030:  7061 636b 6574 2c20 7061 636b 6574 21    packet,.packet!
21:42:15.329010 IP (tos 0x0, ttl 64, id 50341, offset 0, flags [DF], proto UDP (17), length 123)
    localhost.60001 > localhost.50000: UDP, length 95
        0x0000:  4500 007b c4a5 4000 4011 77ca 7f00 0001  E..{..@.@.w.....
        0x0010:  7f00 0001 ea61 c350 0067 fe7a 0000 c350  .....a.P.g.z...P
        0x0020:  0000 9c40 0000 0001 0000 c353 0000 9c40  ...@.......S...@
        0x0030:  0000 0001 0000 c354 0000 9c40 0000 0001  .......T...@....
        0x0040:  0000 c355 0000 9c40 0000 0001 0000 9c43  ...U...@.......C
        0x0050:  0000 9c40 0000 0001 0000 9c43 0000 9c40  ...@.......C...@
        0x0060:  0000 0000 5061 636b 6574 2c20 7061 636b  ....Packet,.pack
        0x0070:  6574 2c20 7061 636b 6574 21              et,.packet!
21:42:15.330766 IP (tos 0x0, ttl 64, id 50342, offset 0, flags [DF], proto UDP (17), length 111)
    localhost.50000 > localhost.50000: UDP, length 83
        0x0000:  4500 006f c4a6 4000 4011 77d5 7f00 0001  E..o..@.@.w.....
        0x0010:  7f00 0001 c350 c350 005b fe6e 0000 c353  .....P.P.[.n...S
        0x0020:  0000 9c40 0000 0001 0000 c354 0000 9c40  ...@.......T...@
        0x0030:  0000 0001 0000 c355 0000 9c40 0000 0001  .......U...@....
        0x0040:  0000 9c43 0000 9c40 0000 0001 0000 9c43  ...C...@.......C
        0x0050:  0000 9c40 0000 0000 5061 636b 6574 2c20  ...@....Packet,.
        0x0060:  7061 636b 6574 2c20 7061 636b 6574 21    packet,.packet!
21:42:15.333473 IP (tos 0x0, ttl 64, id 50343, offset 0, flags [DF], proto UDP (17), length 99)
    localhost.50000 > localhost.50003: UDP, length 71
        0x0000:  4500 0063 c4a7 4000 4011 77e0 7f00 0001  E..c..@.@.w.....
        0x0010:  7f00 0001 c350 c353 004f fe62 0000 c354  .....P.S.O.b...T
        0x0020:  0000 9c40 0000 0001 0000 c355 0000 9c40  ...@.......U...@
        0x0030:  0000 0001 0000 9c43 0000 9c40 0000 0001  .......C...@....
        0x0040:  0000 9c43 0000 9c40 0000 0000 5061 636b  ...C...@....Pack
        0x0050:  6574 2c20 7061 636b 6574 2c20 7061 636b  et,.packet,.pack
        0x0060:  6574 21                                  et!
21:42:15.334157 IP (tos 0x0, ttl 64, id 50344, offset 0, flags [DF], proto UDP (17), length 87)
    localhost.50003 > localhost.50004: UDP, length 59
        0x0000:  4500 0057 c4a8 4000 4011 77eb 7f00 0001  E..W..@.@.w.....
        0x0010:  7f00 0001 c353 c354 0043 fe56 0000 c355  .....S.T.C.V...U
        0x0020:  0000 9c40 0000 0001 0000 9c43 0000 9c40  ...@.......C...@
        0x0030:  0000 0001 0000 9c43 0000 9c40 0000 0000  .......C...@....
        0x0040:  5061 636b 6574 2c20 7061 636b 6574 2c20  Packet,.packet,.
        0x0050:  7061 636b 6574 21                        packet!
21:42:15.334780 IP (tos 0x0, ttl 64, id 50345, offset 0, flags [DF], proto UDP (17), length 75)
    localhost.50004 > localhost.50005: UDP, length 47
        0x0000:  4500 004b c4a9 4000 4011 77f6 7f00 0001  E..K..@.@.w.....
        0x0010:  7f00 0001 c354 c355 0037 fe4a 0000 9c43  .....T.U.7.J...C
        0x0020:  0000 9c40 0000 0001 0000 9c43 0000 9c40  ...@.......C...@
        0x0030:  0000 0000 5061 636b 6574 2c20 7061 636b  ....Packet,.pack
        0x0040:  6574 2c20 7061 636b 6574 21              et,.packet!
21:42:15.335666 IP (tos 0x0, ttl 64, id 50346, offset 0, flags [DF], proto UDP (17), length 63)
    localhost.50005 > localhost.40003: UDP, length 35
        0x0000:  4500 003f c4aa 4000 4011 7801 7f00 0001  E..?..@.@.x.....
        0x0010:  7f00 0001 c355 9c43 002b fe3e 0000 9c43  .....U.C.+.>...C
        0x0020:  0000 9c40 0000 0000 5061 636b 6574 2c20  ...@....Packet,.
        0x0030:  7061 636b 6574 2c20 7061 636b 6574 21    packet,.packet!
^C
8 packets captured
16 packets received by filter
0 packets dropped by kernel
niall@niall-laptop ~ $
```

I used TCPDump to capture the packets being sent across the localhost. These can all be seen above. These are the packets sent for the path shown above (End User 0 -> End User 3).

Single Packet Example

I will focus on one packet to explain the different aspects of it.



This is the packet being sent from Router 0 to the Controller. We can see that the string contents of the packet is “Packet, Packet, Packet!”. Within the hex we can see the destination port, the source port and the knownPath.

The destination of this packet is End User 3, which has a port number of 40003. In hex it is 9C43.

The source port is 40000 (End User 0). In hex it is 9C40.

The known path is 0 as we have not passed it to the controller yet and we do not know the path. This is obviously 0000 in hex.

Near the end of the hex dump we can see the sequence **0000 9C43 0000 9C40 0000 0000**. This sequence is our header as we know the header is made up of our destination port, source port and our known path. Following this we have our string content. Which we can see in the ASCII representation of the packet to the right.

Part 2: Link State Routing

Explanation

While my preconfigured, hard-coded controller as shown above does work. It requires a maintainer to go through and manually update the system if there is a change or if a link goes down. This is where Link State routing and Distance vector routing come in. Both protocols are used today. They both have their own strengths and weaknesses. I will be focusing on Link State Routing.

The idea of Link State Routing is that each router tells all the routers on the network about its neighbors and its links. Each router creates a Link State Packet (LSP) with information regarding its neighbors and links. These LSP are sent to every router on the network (This is known as the LSP being flooded across the network). Each router has a Link State Database containing all the LSPs from across the network. The Link State Database (LSDB) is used to create a “picture” of the network from the perspective of that router.

From my understanding there are multiple ways of creating this “picture”. One such way is to create a graph much like I have in my preconfigured Controller above. They then use this graph to calculate a routing table. Which is a table of the shortest path to each other router from the current router. The shortest path can be calculated with Dijkstra's algorithm. This routing table is then used every time a packet is received to see the shortest path to the packet’s destination.

My Implementation

Unfortunately, I was unable to successfully implement a link state routing protocol into my program. However, I do believe I was on the right track. Each router was to send its LSP to every other router in the network. With an LSBD I would’ve been able to use my existing graph code to create a picture of the network from the perspective of each router. I would’ve also been able to use my existing code for the shortest path to create the routing table.

Conclusion

I realize now that my implementation would've greatly benefited from have multiple sockets for each router. One for receiving and one for sending. This would have been relatively simple to implement had I been doing it from the start but unfortunately, I only became aware when it was already too late.

Having multiple sockets wouldn't change my part 1 very much as I would just have to use a directional graph rather than an unidirectional one. It would have, however, made the second part a lot more doable. I think if I had been using multiple sockets from the beginning I would have been able to get the second part to work.

I also updated how I handled headers since the first assignment which made dealing with them a lot easier. Especially in the Controller, however it did take up some valuable time.

Overall, I have a much better understanding of the routing of packets now and a greater appreciation for the work that goes into routing them correctly. I am incredibly disappointed that I couldn't implement Link State Routing but learning how it works has definitely peaked my interest. I will be attempting to come back to this and finish the implementation when I have time.