Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU33031 Computer Networks
Assignment #2: Flow Forwarding Protocol

Claire Gregg, 20332015

September 11, 2023

Contents

1 Introduction **1**

2 Theory of Topic **2**
2.1 Flow Forwarding . . . 2
2.2 Topology . . . 3
2.3 Shortest Path Algorithms . . . 4
2.3.1 Dijkstra's Shortest Path algorithm . . . 4
2.3.2 Floyd Warshall Algorithm . . . 6

3 Implementation **7**
3.1 My Topology . . . 7
3.2 Headers . . . 9
3.2.1 Sending IP Addresses . . . 10
3.3 Employee & Server . . . 10
3.4 Forwarder . . . 13
3.5 Controller . . . 15
3.5.1 Graph . . . 15
3.5.2 Floyd Warshall . . . 16

4 Summary **18**

5 Discussion **20**

6 Reflection **21**

7 Bibliography **21**

1 Introduction

The focus of this assignment is designing a software defined forwarding system, controlled by a central controller which has information about the full system. This requires learning and reasoning about decisions to forward flows of packets. This includes looking into the information that is kept at network devices

that makes forwarding decisions, and how to use that information to reduce processing required by network elements, while also allowing for scalability and flexibility. The particular problem this assignmnent addresses is a protocol for Software Defined Wide-Area Networks - developing an overlay for network elements linking forwarding mechanisms to a cloud provider's forwarding service. The idea underlying is that of a company whose employees are working from home whose packets must be forwarded correctly to the company's servers in some cloud provider. All of this should be controlled by a central controller, which provides information to network elements about forwarding paths.
For my implementation, I decided to invisage a company which uses a ticketing system for some reason - perhaps to track bugs, or customer service issues. Therefore, employees must be able to:

1. Create a new ticket.
2. Get the first ticket in the queue, removing it from the queue as the employee is working on it.
3. Solve a ticket.

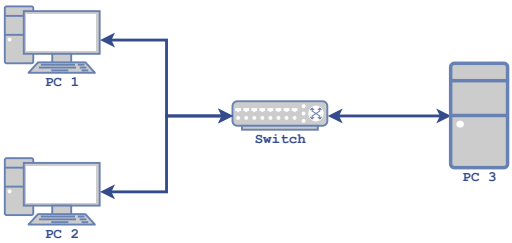
This system has different servers to deal with tickets from different products, and employees need to be able to access all of those servers in the above ways.

2 Theory of Topic

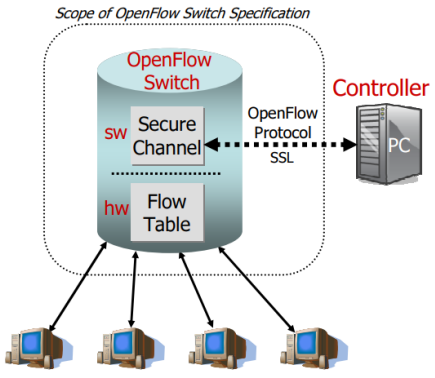
This section will describe the theory underlying the assignment, as well as the theory underlying my implementation. It is important to understand the basis on which my solution is built. I will discuss some general information regarding flow forwarding, the general topology as described in the assignment brief, and an overview of shortest path algorithms in computer networks.

2.1 Flow Forwarding

The internet is essentially a network of networks. Flow forwarding is a method of transporting messages across networks using forwarding tables. What we are implementing in this assignment is a flow forwarding protocol with a central controller. An example of this is the OpenFlow protocol, which uses network controllers to decide the path of network messages between routers. This is shown in Figure 1.



(a) A very simple network example with 3 PCs and a switch.



(b) An idealised OpenFlow Switch (taken from [1]). The switch is controlled by a remote controller, which updates the switch's flow table.

Figure 1: An example of a network and the idealised OpenFlow switch. The OpenFlow switch would replace the switch in the example network.

The theoretical protocol underlying this includes forwarders or switches declaring themselves to the controller when they start, so the controller understands the system. Then, when there is a change to the network, the controller should be informed. Finally, if a switch or network is reached and it does not know where the

destination of the packet is, it contacts the controller to find out the best path to the destination, before sending the packet on.

2.2 Topology

An example of the theoretical topology is given in Figure 2. The solution of this problem should work on this topology, as well as larger topologies.

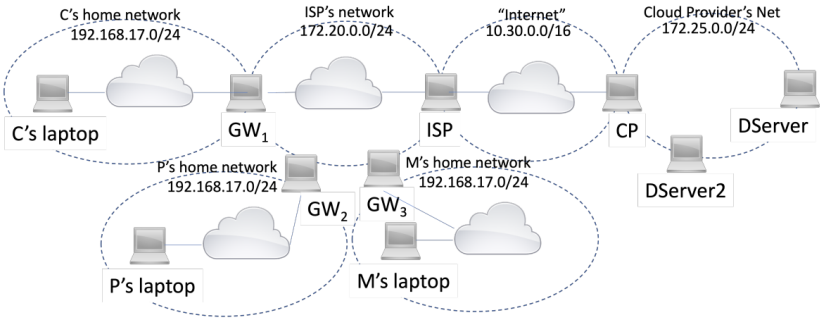


Figure 2: This is an example of a network topology this system should be implemented over given in the assignment brief. There are 6 sub networks in this example, 3 of which are employee's home networks. These employees' home networks are connected to their ISPs via gateways (or forwarders) GW_1, GW_2 and GW_3. The ISP's network is connected to the "Internet" network via the ISP gateway. Finally, the ISP is connected to the cloud provider's network at the CP gateway. Then, inside the cloud provider's network, there are 2 servers which should be accessible by any of the other endpoints. Not shown is the controller which is connected to at least all of the forwarders (GW_1, GW_2, GW_3, ISP, and CP).

However, this, topology only includes the IP addresses. The intention of this assignment is to have endpoint application IDs so that a client can just send a message to a certain ID, without needing to know the IP address of where that application is hosted. This allows flexibility if a server goes down and is replaced, for example. An example of these endpoint IDs being communicated and used is shown in Figure 3.

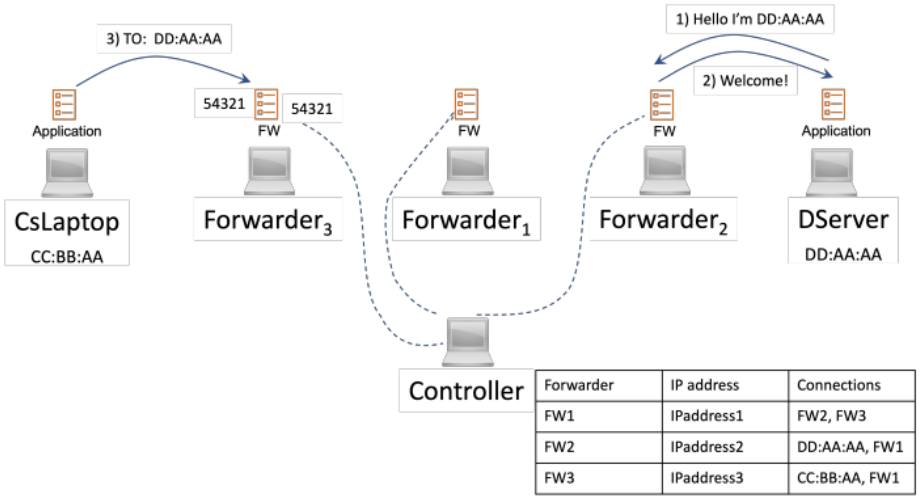
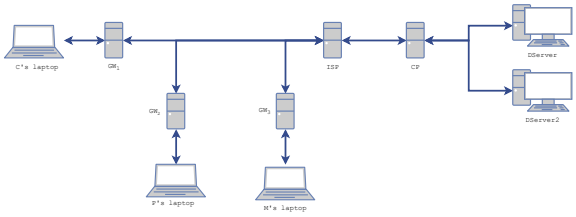


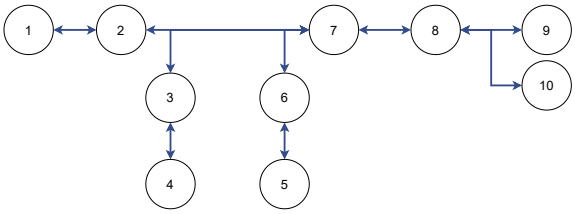
Figure 3: This figure shows the forwarding of traffic between two endpoints with IDs CC:BB:AA and DD:AA:AA. First of all, the endpoint at DServer declares itself to its nearest forwarder, which it acknowledges. Then, the controller knows about what each forwarder has access to. Finally, the endpoint at CsLaptop sends a message to its nearest forwarder with a destination ID. This forwarder contacts the controller to find out where to send the message so that it will reach the correct endpoint. The forwarder saves this information so that in future it does not need to request the forwarding table again. Note that network traffic is passed between ports 54321 for this forwarding service.

2.3 Shortest Path Algorithms

As the controller in this problem has a full overview of the network, this flow forwarding problem can be viewed as a shortest path problem in graph theory. This means that there is a good deal of resources available for finding the most efficient paths between network elements. The two options which I will discuss for finding shortest paths are Dijkstra's Shortest Path algorithm and the Floyd Warshall algorithm. First, an example of how a network topology may be turned into a graph is shown in Figure 4.



(a) A simplified version of Figure 2. All networks have been removed, leaving just network elements and their connections



(b) A further simplified version of Figure 4a. All network elements have been replaced by vertices, and network connections are now edges.

Figure 4: An example of how a network's topology may be changed into a graph for use in shortest path algorithms.

2.3.1 Dijkstra's Shortest Path algorithm

Dijkstra's algorithm is used in Link State routing. In Link State routing, the knowledge of the network topology must be shared between routers. However, as the controller in this assignment will have knowledge of the network topology, that is not necessary. So, instead, we can treat this as a typical shortest path problem. Dijkstra's algorithm follows these steps to find the shortest path to all nodes from one source node.

1. Start with the source node.
2. Assign a cost of 0 to this node and make it "permanent".
3. Examine each neighbour node of the last permanent node.
4. Assign a tentative cumulative cost to each node.
5. Find the node with the smallest cumulative cost in the list of tentative nodes, and make it "permanent". If the node can be reached in more than one direction, pick the direction with the smallest cumulative cost.
6. Repeat 3-5 until there are no more tentative nodes and every node is permanent.

We will take a short example using Figure 4b, starting at node 1. In this, we will assume all "edges" have the same weight, as we have no information to inform us otherwise. After running through Dijkstra's algorithm once, we get Tables 1, as node 1 can only access one node. After this, as shown in Tables 2, there are several nodes with the same distance, so we must make a selection of which node to use first. In this case, we will select that with the lowest number. The next iteration is shown in Tables 3, where a new node is added to the tentative node list but is node made permanent as its distance is longer. Finally, we skip ahead to after all of the iterations are done in Tables 4

Routing Table		
Source	Distance	Prev Node
1	0	-
2	1	1

(a) Initial Forwarding Table

Tentative Nodes		
Source	Distance	Prev Node
-	-	-

(b) Initial Tentative Node Table

Table 1: Forwarding table and tentative node table after one iteration of Dijkstra on Figure 4b.

Routing Table		
Source	Distance	Prev Node
1	0	-
2	1	1
3	2	2

(a) Forwarding Table after 2 iterations of Dijkstra on Figure 4b.

Tentative Nodes		
Source	Distance	Prev Node
6	2	2
7	2	2

(b) Tentative Node Table after 2 iterations of Dijkstra on Figure 4b.

Table 2: Forwarding table and tentative node table after two iterations of Dijkstra on Figure 4b.

Routing Table		
Source	Distance	Prev Node
1	0	-
2	1	1
3	2	2
6	2	2

(a) Forwarding Table after 3 iterations of Dijkstra on Figure 4b.

Tentative Nodes		
Source	Distance	Prev Node
7	2	2
4	3	3

(b) Tentative Node Table after 3 iterations of Dijkstra on Figure 4b.

Table 3: Forwarding table and tentative node table after three iterations of Dijkstra on Figure 4b.

Routing Table		
Source	Distance	Prev Node
1	0	-
2	1	1
3	2	2
6	2	2
7	2	2
4	3	3
5	3	6
8	4	7
9	5	8
10	5	8

(a) Final Forwarding Table after Dijkstra's Algorithm has run on Figure 4b.

Tentative Nodes		
Source	Distance	Prev Node
-	-	-

(b) Empty Tentative Node Table after Dijkstra's Algorithm has run on Figure 4b.

Table 4: Final forwarding table and tentative node table after Dijkstra's Algorithm has run on Figure 4b.

So at the end of Dijktra's algorithm, we get details on how to get from one node to every other node in the network.

2.3.2 Floyd Warshall Algorithm

The Floyd Warshall algorithm is different from Dijkstra's in that it is an all-pairs algorithm - that is, it find the shortest path from every node to every node. It incrementally improves upon the shortest path between every two points in $\Theta(N^3)$ time. For the version of Floyd Warshall with paths as well as distances, it uses two $N \times N$ matrices, where N is the number of nodes in the graph. One is the distance matrix, and the other is the next node matrix. The distance matrix is initialised to all infinities, and the next node matrix is initialised to all null. Then, the following algorithm is enacted (taken from [2]).

```
procedure FloydWarshall() is
    for each edge (u, v) do
        dist[u][v] <- w(u, v)  // The weight of the edge (u, v) (in our case
            ↪ this is always 1)
        next[u][v] <- v
    for each vertex v do
        dist[v][v] <- 0
        next[v][v] <- v
    for k from 1 to |V| do // standard Floyd-Warshall implementation
        for i from 1 to |V|
            for j from 1 to |V|
                if dist[i][j] > dist[i][k] + dist[k][j] then
                    dist[i][j] <- dist[i][k] + dist[k][j]
                    next[i][j] <- next[i][k]
```

Once this algorithm is enacted on Figure 4b, you get Tables 5.

	1	2	3	4	5	6	7	8	9	10
1	0	1	2	3	3	2	2	3	4	4
2	1	0	1	2	2	1	1	2	3	3
3	2	1	0	1	2	1	1	2	3	3
4	3	2	1	0	3	2	2	3	4	4
5	3	2	2	3	0	1	2	3	4	4
6	2	1	1	2	1	0	1	2	3	3
7	2	1	1	2	2	1	0	1	2	2
8	3	2	2	3	3	2	1	0	1	1
9	4	3	3	4	4	3	2	1	0	1
10	4	3	3	4	4	3	2	1	1	0

(a) *dist* matrix after running Floyd Warshall algorithm

	1	2	3	4	5	6	7	8	9	10
1	1	2	2	2	2	2	2	2	2	2
2	1	2	3	3	6	6	7	7	7	7
3	2	2	3	4	6	6	7	7	7	7
4	3	3	3	4	3	3	3	3	3	3
5	6	6	6	6	5	6	6	6	6	6
6	2	2	3	3	5	6	7	7	7	7
7	2	2	3	3	6	6	7	8	8	8
8	7	7	7	7	7	7	7	8	9	10
9	8	8	8	8	8	8	8	8	9	10
10	8	8	8	8	8	8	8	8	9	10

(b) *next* matrix after running Floyd Warshall algorithm

Table 5: Matrices after running the Floyd Warshall algorithm.

Having looked at these two shortest path algorithms I decided to use Floyd Warshall over Dijkstra as the algorithm only needs to be run once every time the network has been updated, which is more efficient than Dijkstra which would need to run for every node which requests an update.

3 Implementation

This section provides the overall design of my solution. I explain the topology I used to demonstrate my solution, including the infrastructure I used to implement this. Then, I discuss the headers of the protocol I have created. I discuss my implementations of the relevant network elements. This includes going into detail as to how I implemented the graphing and Floyd Warshall algorithm in the controller.

3.1 My Topology

The topology I created to implement and demonstrate my solution is shown in Figure 5.

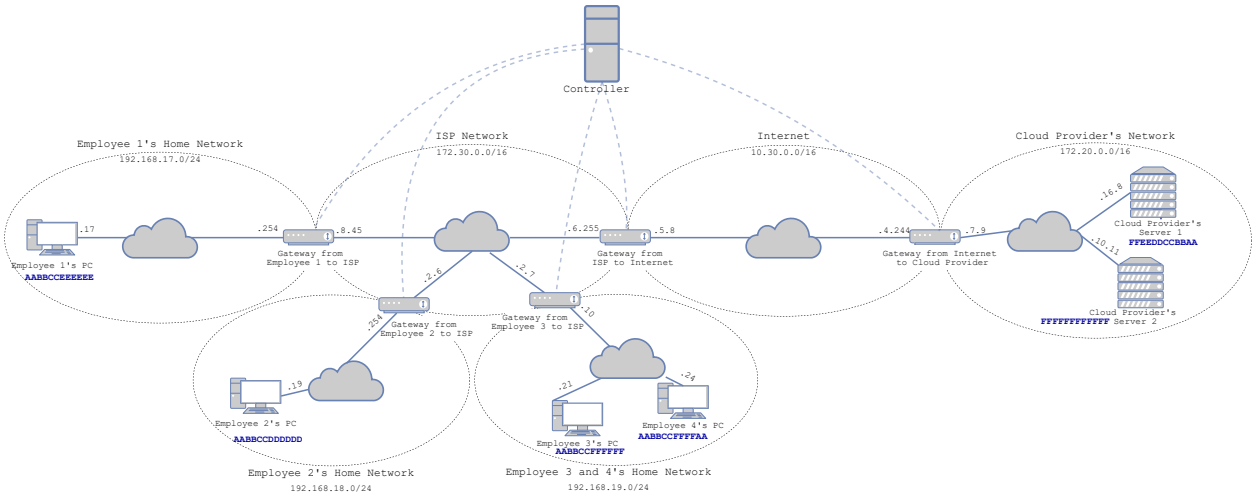


Figure 5: This is the topology I created for this assignment. IP addresses of all network elements are given. There are 6 networks, which are all connected by gateways, or forwarders. All of the gateways are connected to the controller. There are 3 employee home networks, 2 of which have just 1 employee computer in them, and the 3rd of which has 2 separate employee's computers. All of the employee networks connect to their ISP's network (via forwarders). The ISP network connects to the Internet's network via another forwarder. The Internet connects to the cloud provider's network via another forwarder. Within the cloud provider's network, there are 2 servers. Also note that all endpoints have IDs, which are 6 byte hexadecimal numbers.

I implemented this topology using Docker Compose, which is a tool for working with multi-container Docker applications. For this, each network element has a Dockerfile which defines information like how to run it, where to get its dependencies from, and where to put its output, if it needs to. To use Docker Compose, there is also a docker-compose.yml file. This file contains definitions for the networks and the services (network elements). The networks are defined by naming them and giving their subnet (e.g 192.168.17.0/24). The network elements are defined with the following information: the network element's name, where its Dockerfile is located, any command line arguments to give to the element, the network(s) the element is in, its IP addresses in those networks, and what other network elements it depends on. There is also a service declaration for a tcpdump service to dump the PCAP file, which is configured to only collect UDP packets. An example of an element's declaration is given in Listing 2. I wrote a python script to automate the writing of the docker-compose file as it got lengthy to type by hand.

```
gateway_e1_to_isp:
    build:
        dockerfile:
            forwarders/Dockerfile
    # Appends any IP addresses this element can access
    command: ["192.168.17.17","172.30.6.255","172.30.2.6","172.30.2.7"]
    networks:
        home1:
            ipv4_address: 192.168.17.254
        isp:
            ipv4_address: 172.30.8.45
    depends_on:
        - tcpdump
        - controller
```

Listing 2: Declaration of gateway from employee 1's home network to the ISP network. It takes the IP addresses it can access as command line arguments. It is in 2 networks and has different IP addresses in those 2 networks. Finally, it depends on the controller being up and on tcpdump running.

3.2 Headers

This protocol relies on headers which define details about why a message is being sent, where from, and where to. It uses a number of different headers, which I will discuss in tables 6, 7 and 8. All messages in the network will have one of these for headers.

Endpoint Sending Declaration	
Byte 1	1
Bytes 2-7	New endpoint ID

(a) This header is used in the case that an endpoint (an employee or a server) is declaring its endpoint ID to its nearest forwarder. It consists of a control byte, which is set to 1 (representing that a declaration is being sent), and the new endpoint ID. This packet does not have any payload.

Endpoint Sending Packets	
Byte 1	0
Bytes 2-7	Destination endpoint ID
Bytes 8-13	Source endpoint ID
Bytes 14+	Payload

(b) This header is used in the case that an endpoint is sending a packet. It consists of a control byte, which is set to 0 (identifying that a packet is being sent), the destination endpoint ID, and the source endpoint ID. This is followed by the payload.

Table 6: These are the headers for messages endpoints will send to forwarders.

Forwarder Requesting an Update	
Byte 1	4

(a) This header is used when the forwarder needs new information. This is used when it receives a packet whose destination is not in its forwarding table. It sends this message to the controller, with no payload. The controller will respond using header 8.

Forwarder Declaring a New Endpoint ID	
Byte 1	2
Bytes 2-5	Forwarder's IP address
Bytes 6-11	New Endpoint ID

(b) This header is used when the forwarder is declaring a new endpoint ID to the controller, after an endpoint has declared itself to the forwarder using header 6a. It consists of a control byte (set to 2 representing that a new endpoint ID is being sent), one of its IP addresses, and the new endpoint ID.

Forwarder Declaring itself to Controller	
Byte 1	1
Bytes 2-5	Forwarder's first IP address
Bytes 6-9	Forwarder's second IP address
Bytes 9+	All addresses accessible to forwarder

(c) This header is used when the forwarder is declaring itself to the controller. It consists of a control byte (set to 1, representing that a forwarder declaration is being sent), its 2 IP addresses, and a list of IP addresses the forwarder can access. See Section 3.2.1 to see how IP addresses are encoded.

Table 7: These are the headers for messages the forwarder will send to the controller.

Forwarder Requesting an Update	
Byte 1	4
Bytes 2-7	New endpoint ID
Bytes 8-11	IP address which is next step to new endpoint
Bytes 12+	New endpoint ID and next IP addresses repeated

Table 8: This is the header of a message the forwarder will receive after requesting new information from the controller. It indicates more information about the network will be included. This will take the form of a new endpoint ID followed by the next IP address toward that endpoint. This repeats for all endpoints in the network the controller knows about, excluding those for which the forwarder being contacted is the gateway.

3.2.1 Sending IP Addresses

IP addresses are generally stored and used as strings - for example "192.168.17.17". However, this is an inefficient way to transport this particular information. I took advantage of the structure of IP addresses to send messages containing addresses more efficiently. IPv4 addresses (which is what we are concerned with) consist of 4 one byte numbers separated by .s. So, I wrote two functions to translate a string IP address into bytes and translate bytes into a string IP address. These are shown in Listings 3 and 4.

```
def ip_address_to_bytes(ipAddress: str) -> bytes:
    numbers = ipAddress.split(".")
    integers = [int(number) for number in numbers]
    bytes = b''
    for integer in integers:
        # Only need one byte for each IP address segment as they have a max of
            ↪ 255
        bytes += integer.to_bytes(1, 'big')
    return bytes
```

Listing 3: A python function to translate an IP address in string form into a series of bytes. It splits the string using '.' as a delimiter, then parses each number into an integer, which are then concatenated together into a bytes object.

```
def bytes_to_ip_address(bytes: bytes) -> str:
    ip = ""
    for byte in bytes:
        ip += str(byte) + "."
    return ip[:-1]
```

Listing 4: A python function to translate a bytes object into an IP address in a string format. It loops through the bytes, and concatenates each byte's int value to a string using a '.' to split them.

3.3 Employee & Server

This section will concern the employee and server network elements. They are very similar as both are endpoints, but they have some differences. I will also go into more detail of the theoretical company I invisaged for this solution.
Both servers and clients take in two command line arguments. The first is the endpoint ID it should use, and the second is the endpoint's gateway IP address. These are used so that endpoints can share code. Initially, any of the endpoints will take in these two pieces of information, and will use the header shown in Table 6a to declare itself to the forwarder. All ports used in this solution are ports 54321, as defined in the assignment brief.

There are two versions of the employee in my implementation. One interacts with the system automatically, and is most useful for testing. The other is interactive and can be controlled using the command line. After sending the declaration, the interactive client will take command line input to decide what to do. The options are described in Listing 5, which shows how user input is taken. The automatic user instead selects a server randomly, and creates a new ticket, then gets a ticket from that server, then solves the ticket it got from that server. Any of these will use the header described in Table 6b, followed by an action byte (1 for new ticket, 2 for getting a ticket, and 4 for solving a ticket). This is followed by the ticket number if the user is solving a ticket.

```
while True:
    val = input("Do you want to create a new ticket (1), get a ticket from a
        ↪ server (2), or solve a ticket (3)? (Stop by typing quit)")
    if val == '1':
        val = input("Which server do you want to request from? Index into the
            ↪ following: {}".format(lib.destinations))
        destination = lib.destinations[int(val)]
        new_ticket(UDPClientSocket, gatewayAddress, elementId, destination)
        recv(UDPClientSocket)
    elif val == '2':
        val = input("Which server do you want to get the ticket from? Index
            ↪ into the following: {}".format(lib.destinations))
        destination = lib.destinations[int(val)]
        get_ticket(UDPClientSocket, gatewayAddress, elementId, destination)
        recv(UDPClientSocket)
    elif val == '3':
        val = input("Which server do you want to get the ticket from? Index
            ↪ into the following: {}".format(lib.destinations))
        destination = lib.destinations[int(val)]
        ticket = int(input("What ticket number do you want to solve? Make sure
            ↪ you have already claimed any ticket you are trying to solve!"))
        solve_ticket(UDPClientSocket, gatewayAddress, elementId, destination,
            ↪ ticket)
        recv(UDPClientSocket)
    elif val == 'quit':
        break
```

Listing 5: Employee code for taking user input at the command line. It allows the user to choose between creating a ticket, getting a ticket from the server (at which point the user should solve that ticket), or solving a ticket. Once the user has selected that, they must select which product's server they want to work with tickets from. In the case they are solving a ticket they must give a ticket number to solve. Depending on the option, the relevant function is called, and the client waits to receive a response.

After sending the declaration, the server declares two lists (for new tickets and tickets which are in progress) and an int variable to track the next ticket number. Then, the server waits to receive a packet. If the packet is a new ticket declaration, it adds the next ticket number to the new tickets list, sends that number to the client, and increments the next ticket number. As this is intended to be a simple demonstration of flow forwarding, once the ticket number gets to 256, it is reset to 0, so it can always be contained in a single byte. If the packet the server receives is a request to claim a ticket, it takes the first ticket in the new tickets list, adds it to the tickets in progress list, and sends it back to the client. Finally, if the packet is a ticket solved message, the server removes the ticket from the tickets in progress list, and sends a message saying the ticket has been solved back to the client.
There are two servers in my example topology, and they intentionally contain entirely separate ticket lists, as they are intended to represent separate products, and tickets regarding those products.
A simplified example of the flow of the endpoints is shown in Figure 6. It is simplified as it does not include any of the forwarder information which will be described in the next section.

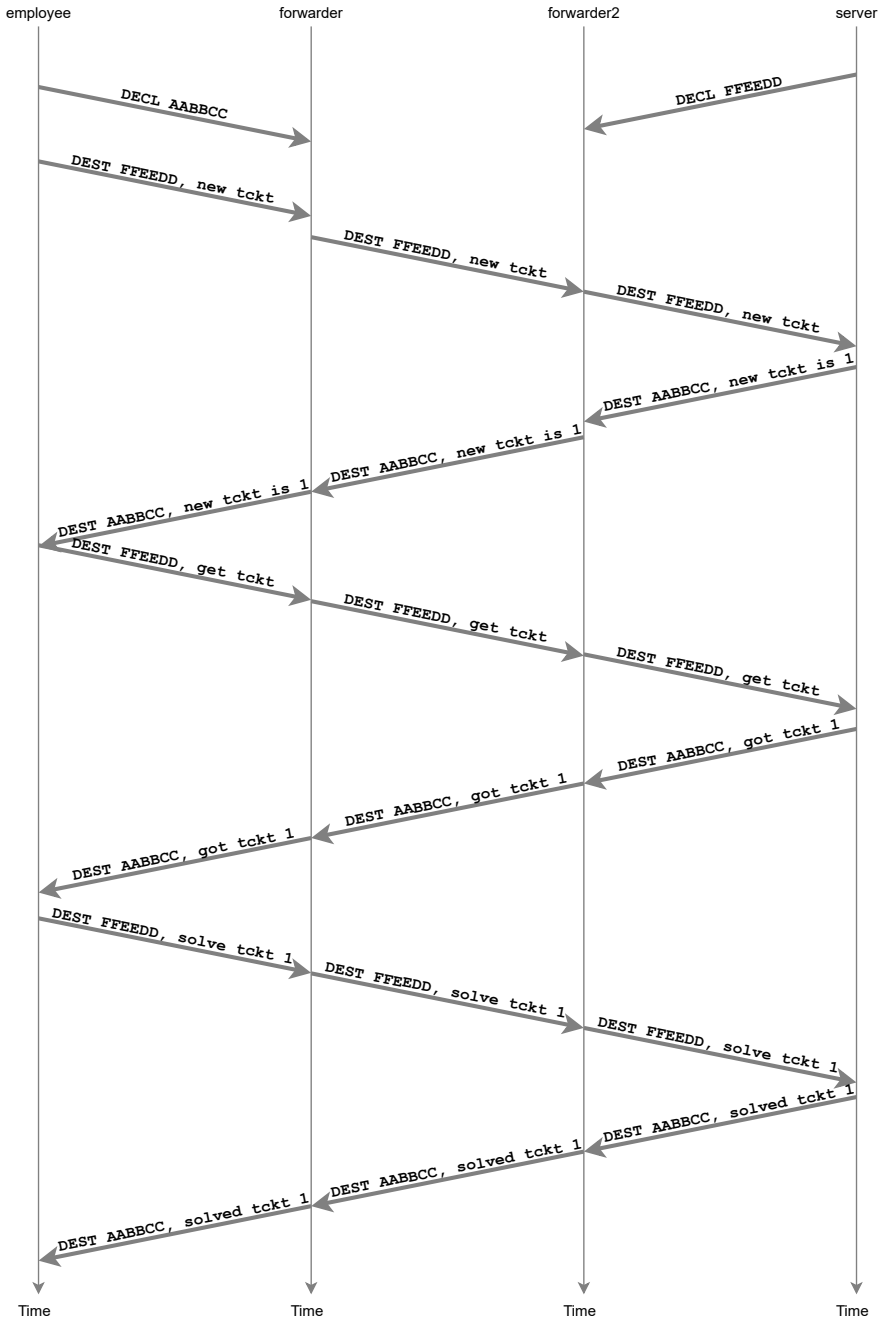


Figure 6: This flow diagram shows an example of a employee and server network element might go through. It is simplified to not include the forwarders actions in contacting the controller. Initially, both the employee and the server contact their nearest forwarders to declare their endpoint IDs. Then, the employee declares a new tciket to the server, and you can see that message being passed along. The reply to that new ticket, with the number assigned to it is shown being passed back. Then the employee claims the first ticket in the queue. This ends up being ticket 1 as this is the only employee in this theoretical network. Finally, the employee solves ticket 1, and gets a confirmation message that they successfully solved ticket 1.

3.4 Forwarder

The forwarder takes in packets from either endpoints or other forwarders, and forwards them on to other forwarders or endpoints, with the intention of the packets eventually getting to an endpoint. When a forwarder does not know where to send a packet to, it contacts the controller to get an update to its forwarding table. This section will discuss this network element and its interaction with the other network elements in more detail.
The forwarder takes in a list of IP addresses it can access as command line arguments. This means it can set up a basic forwarding table with just the IP addresses. It initially sets up a dictionary shared between processes for the routing table, and initialises it with the IP addresses passed in as command line arguments in this format ["192.168.17.17":"192.168.17.17"]. It also initialises a mutual exclusion lock for this dictionary. The forwarder finds one of its IP addresses by calling `socket.gethostbyname(socket.gethostname())`, and finds the other by comparing that IP address with a dictionary of forwarder IP addresses which maps matching IP addresses, as shown in Figure 7. Then, the forwarder creates two sockets in different processes bound to port 54321 on these IP addresses. It declares this forwarder from just one of these ports using the header shown in Table 7a.

```
gateways = {
    "192.168.17.254" : "172.30.8.45",
    "172.30.8.45" : "192.168.17.254",
    "192.168.18.254": "172.30.2.6",
    "172.30.2.6": "192.168.18.254",
    "172.30.6.255" : "10.30.5.8",
    "10.30.5.8" : "172.30.6.255" ,
    "10.30.4.244" : "172.20.7.9",
    "172.20.7.9" : "10.30.4.244",
    "192.168.19.10" : "172.30.2.7",
    "172.30.2.7": "192.168.19.10"
}
```

Figure 7: This figure shows the map of forwarder IP addresses to matching forwarder IP addresses. So, for example, the gateway from employee 1's home network to their ISP's network has two IP addresses, "192.168.17.254", and "172.30.8.45". So, this dictionary has a mapping in both directions, so the forwarder can set up the port in either direction first, and then find the matching IP address.

Once both ports are set up and the forwarder has been declared to the controller, the forwarder listens on both ports using multiprocessing. When a forwarder receives a packet typically, it must find out if the packet is a declaration from an endpoint, or if it is a packet to be passed along. It does this by identifying the control byte and matching it against the headers shown in Tables 6a and 6b. If it is a declaration of a new endpoint, the forwarder first updates its own forwarding table to map the new endpoint ID to the IP address the declaration was sent from. Then, the forwarder sends a message to the controller using the header shown in Table 7b.
If the forwarder has received a packet to send along, and it recognises the destination endpoint ID, it sends the packet on to the IP address which the endpoint ID maps to in the forwarding table. However, if the forwarder does not recognise the destination endpoint ID (it is not in the forwarding table) it sends a request for new information to the controller using the header shown in Table 7c. Now, while it waits for the response, the forwarder checks any messages it receives for being a declaration from a new endpoint, packet to be sent along, or new information from the controller. The new information from the controller can be identified by the control byte shown in Table 8. If the packet is either of the standard messages, the forwarder responds as normal. However, if the message is an update from the controller, the forwarder updates its forwarding table to map all of the endpoint IDs given in the packet to the next IP addresses in the packet. Then, if the endpoint ID in the original message is still not in the forwarding table, the message is dropped, as it is unlikely that destination will appear, and a message is printed to console. However, usually, the destination endpoint is generally now in the forwarding table, and the message can be sent on. An example of the forwarding table after its initial IP addresses, declaration from an endpoint, and new information from the

controller is shown in Figure 9.

Destination	Next IP Address Towards that Destination
192.168.18.19	192.168.18.19
172.30.6.255	172.30.6.255
172.30.8.45	172.30.8.45
172.30.2.7	172.30.2.7
AABBCCDDDDDD	192.168.18.19
FFEEDDCCBBAA	172.30.6.255
FFFFFFFFFFFF	172.30.6.255
AABBCCFFFFAA	172.30.2.7
AABBCCFFFFFF	172.30.2.7
AABBCCEEEEEE	192.168.17.254

Table 9: An example forwarding table, for what is labeled as Gateway from Employee 2 to ISP in Figure 5. Its initial forwarding table consists of the first 4 rows, which are just the IP addresses it can access. The 5th row is added when employee 2 declares itself to the forwarder. The remaining rows are added when the forwarder requests more information from the controller, and map the remaining endpoints to the next IP towards them. Importantly, the endpoint in row 5 is not remapped.

An overview of the flow a forwarder might go through is shown in Figure 8.

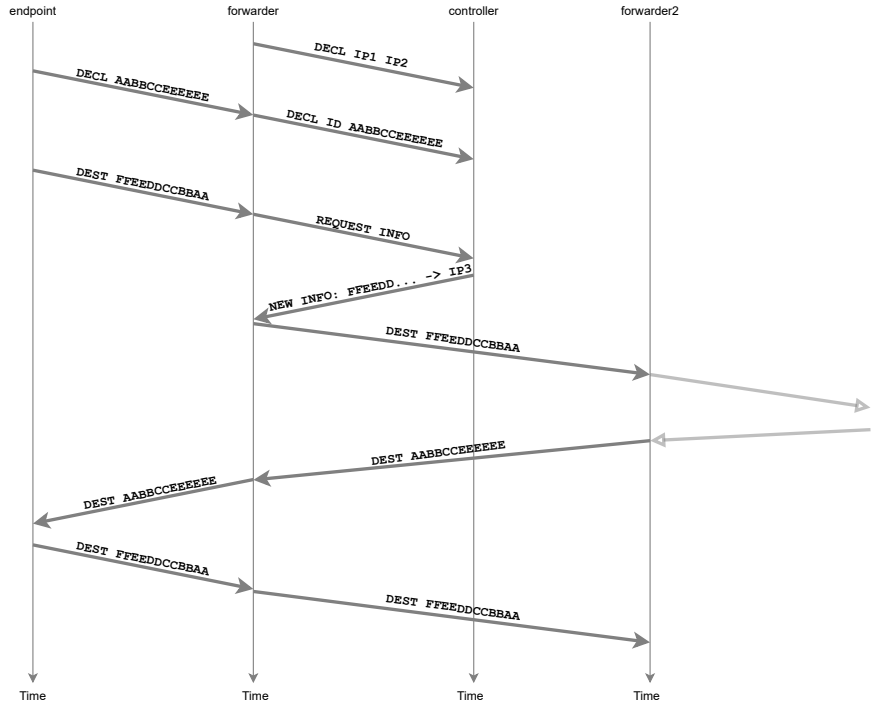


Figure 8: This flow diagram shows an example of the flow a forwarder might go through. First, it declares itself to the controller. Then, it receives a declaration of a new endpoint ID, which it declares to the controller. Then, it receives a packet with a destination it does not recognise, so it contacts the controller. The controller informs it of all of the endpoint mappings, including that which it previously did not recognise. Then it sends the message on to the next IP. Eventually, it receives a reply which it forwards to the original endpoint, as it knows where that is. When the original endpoint sends out a message to the same destination again, the forwarder does not need to find out where that destination is again, so it can send the message straight on to the next IP address.

3.5 Controller

The controller is the most complex network element. It takes in an IP address in every network which it assigns ports to using multiprocessing (all at port 54321, of course). However, so that the controller can work successfully across these many processes, there are several shared variables: a dictionary to map IP addresses to node IDs, a list of edges, an int to store the number of nodes, all of which will be discussed more in Section 3.5.1, as well as a next node matrix which will be discussed more in Section 3.5.2, and a variable which is used to store whether or not the shortest path has been calculated since the network has updated. When the controller receives a message, it identifies what to do by matching the control byte to those described in Table 7. If it is a declaration from a forwarder, the forwarder is added to the graph. If it is a declaration of a new endpoint ID, the endpoint ID is added to the graph on the forwarder's node. If it is a request for new information, if the shortest path has not been calculated since the last time the network updated, it calculates the shortest path again. Then, it sends back an update of all the endpoints which are not located at the requesting forwarder, along with the IP address of the next node.

3.5.1 Graph

The graph representation of the network is created automatically by the controller as it receives declarations. When the controller receives a declaration of a new forwarder, it creates a new node, assigning the current node number to it. Then, it adds the forwarder's IP addresses to the map of IP addresses to node IDs, mapping to the node number. Then it creates temporary nodes for all of the IP addresses the forwarder says

it can contact, or if those IP addresses already have nodes, uses those node IDs. Then, it adds edges from the new node associated with the forwarder to the new temporary node. The controller does this for every IP address the forwarder says it can access. When the forwarder can access an already existing node, that node ID is used instead.
When a new endpoint ID declaration is received, the controller takes the IP address it was received from, finds what node that is, and adds the endpoint ID to the map of IP addresses to node IDs, mapping to the same node ID as the IP address it came from.
Once all of the nodes have been fully declared with the topology shown in Figure 5, the graph might look like Figure 9. Every time either a new forwarder is declared or a new endpoint ID is declared, the variable tracking whether the shortest path has been calculated since the network updated is set to false.

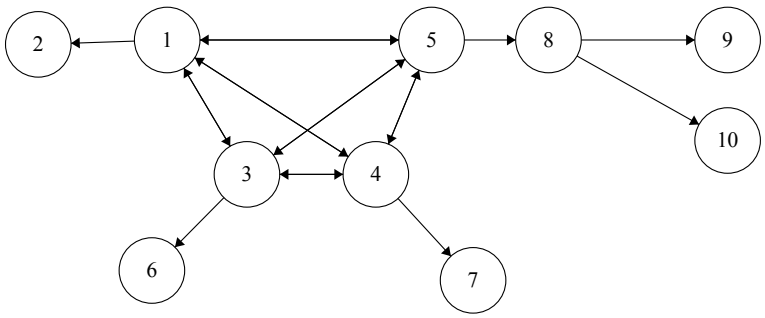


Figure 9: This directed graph shows an example of how the controller might store the network shown in Figure 5. The numbers may switch around depending on what order forwarders declare themselves. Note that the endpoint nodes are only nodes-to, as they do not interact with the controller. The IP address to node ID dictionary gives the IP addresses for each node.

3.5.2 Floyd Warshall

When the variable tracking whether the shortest path has been calculated since the network updated is set to false and a forwarder requests new information, the Floyd Warshall algorithm runs. The code I use for this is given in Listing

```
# Parameters:
#  numNodes: Number of nodes in network
#  edges: List of tuples of edges in network
#  nextNodeMatrix: Matrix containing the next node required to travel to a
    ↪ certain node index
def calculate_routes(numNodes: multiprocessing.Value, edges: list,
    ↪ nextNodeMatrix: list):
    matrixDimen = numNodes.value + 1
    distArray = [[math.inf for _ in range(0, matrixDimen)] for _ in range(0,
        ↪ matrixDimen)]
    localNextNodeMatrix = [[None for _ in range(0, matrixDimen)] for _ in
        ↪ range(0, matrixDimen)]
```

```
    # Loop through edges, adding them to the matrices
    for edge in edges:
        distArray[edge[0]][edge[1]] = 1 # Everything has a weight of 1 in this
            ↪ network
        localNextNodeMatrix[edge[0]][edge[1]] = edge[1]

    # Loop through numNodes
    for i in range(0, matrixDimen):
        distArray[i][i] = 0
        localNextNodeMatrix[i][i] = i

    # Floyd-Warshall
    for k in range(0, matrixDimen):
        for i in range(0, matrixDimen):
            for j in range(0, matrixDimen):
                if distArray[i][j] > distArray[i][k] + distArray[k][j]:
                    distArray[i][j] = distArray[i][k] + distArray[k][j]
                    localNextNodeMatrix[i][j] = localNextNodeMatrix[i][k]

    nextNodeMatrix[:] = localNextNodeMatrix
```

Listing 6: My code for calculating the shortest route from any node in the network to any node in the network. The next node from every node index is stored in nextNodeMatrix, a variable shared among all of the processes. This follows the pseudocode shown in Listing 1. Initially, the matrices are initialised to infinity (for the distance matrix) and null (for the next node matrix). Then, the edges are looped through, with each edge being added with a weight of 1. Then, the vertices are looped through using numNodes. The distance from any vertex to itself is 0, and the next node is itself. Then, the Floyd Warshall algorithm runs. Finally, the shared version of the next node matrix is set to the newly calculted version.

Once Floyd Warshall has run, or if the shortest path had already been calculated, there is a function to send the new endpoints and IP addresses to the forwarder. It takes the IP dictionary, the node which is requesting new information, and the next node matrix. It loops through all the endpoint IDs in the IP dictionary, and if they are not located at the node which is requesting the update, the node which is the next node towards that endpoint from the requesting node is found. Then, to find the IP for that next node, the starts of the IP addresses of the requesting node and the next node are matched. Finally, once all of this has been done for all of the endpoint IDs in the dictionary, all of the endpoint IDs and next IP addresses are sent back to the forwarder as shown in Table 8. One of these messages is shown in Figure 10.

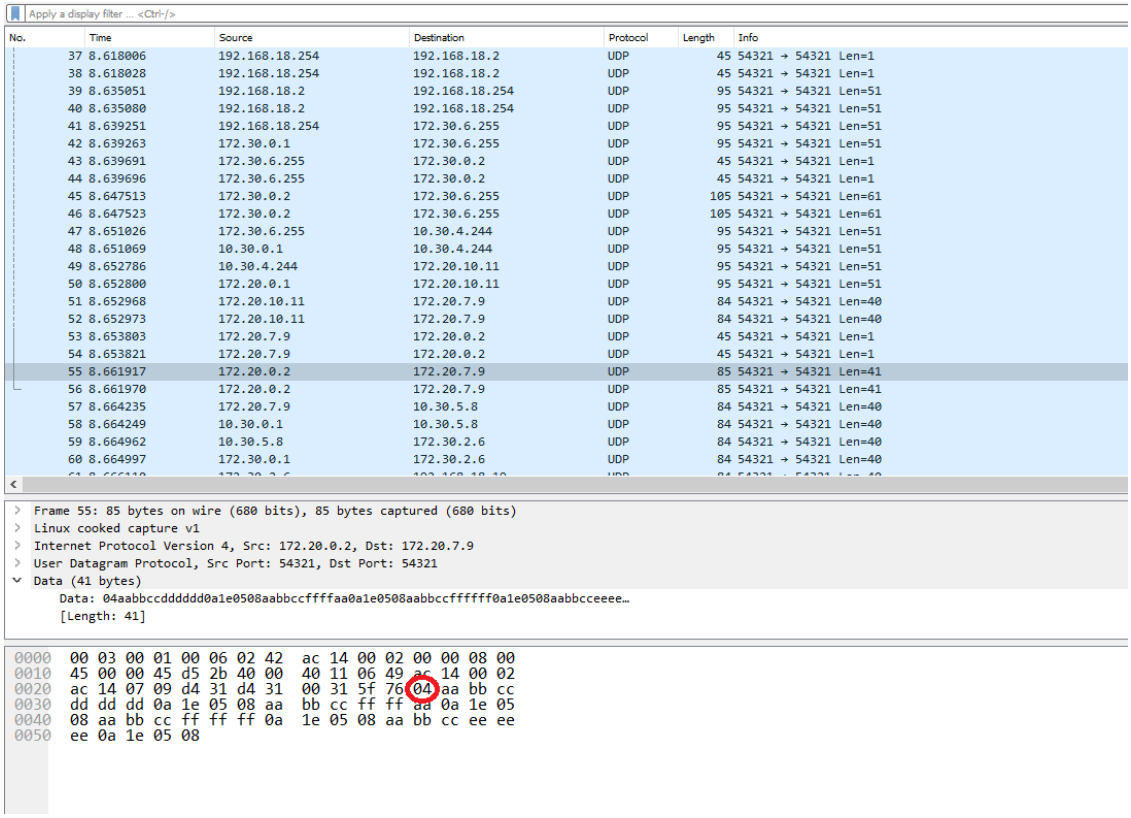


Figure 10: This WireShark screenshot shows a new information packet sent from the controller to a forwarder at 172.20.7.9. The start of the message is circled, with the control byte of 4 as the first byte. Following this, endpoint IDs are sent followed by IP addresses encoded in 4 bytes. 4 new endpoint IDs are sent in this message.

4 Summary

With all of the network elements I have discussed implemented, my solution gives a topology and a flow forwarding protocol which could be used for any company. I showed this using the example of a company that works with tickets. You can see the full process working in Figures 11 (without the interactive client), 12 and 13. The network is controlled by a central controller which forwarder contact whenever they need information on how to access a given endpoint. The controller uses the Floyd Warshall algorithm to calculate the all pairs shortest path in the network.

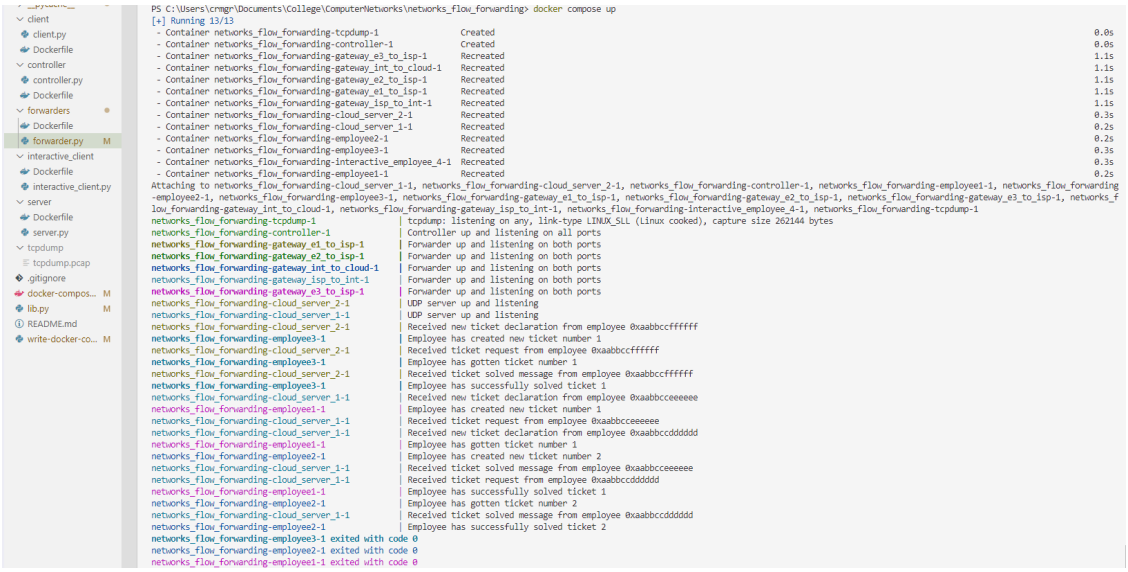


Figure 11: This shows my solution working using docker compose. The controller, forwarders, clients and servers are running, and the clients and server are able to send information about tickets back and forth. Note that because of how docker-compose works, the output from the containers is not always in order.

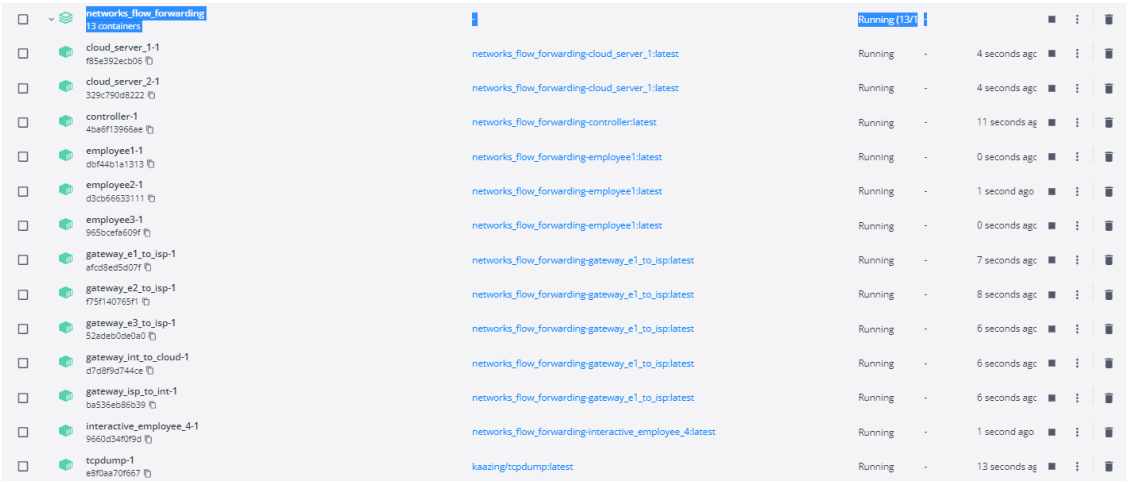


Figure 12: This shows my solution working in Docker. 13 containers are up and running, the 4 employees, 2 servers, 5 forwarders (gateways), the controller, and the tcpdump container.

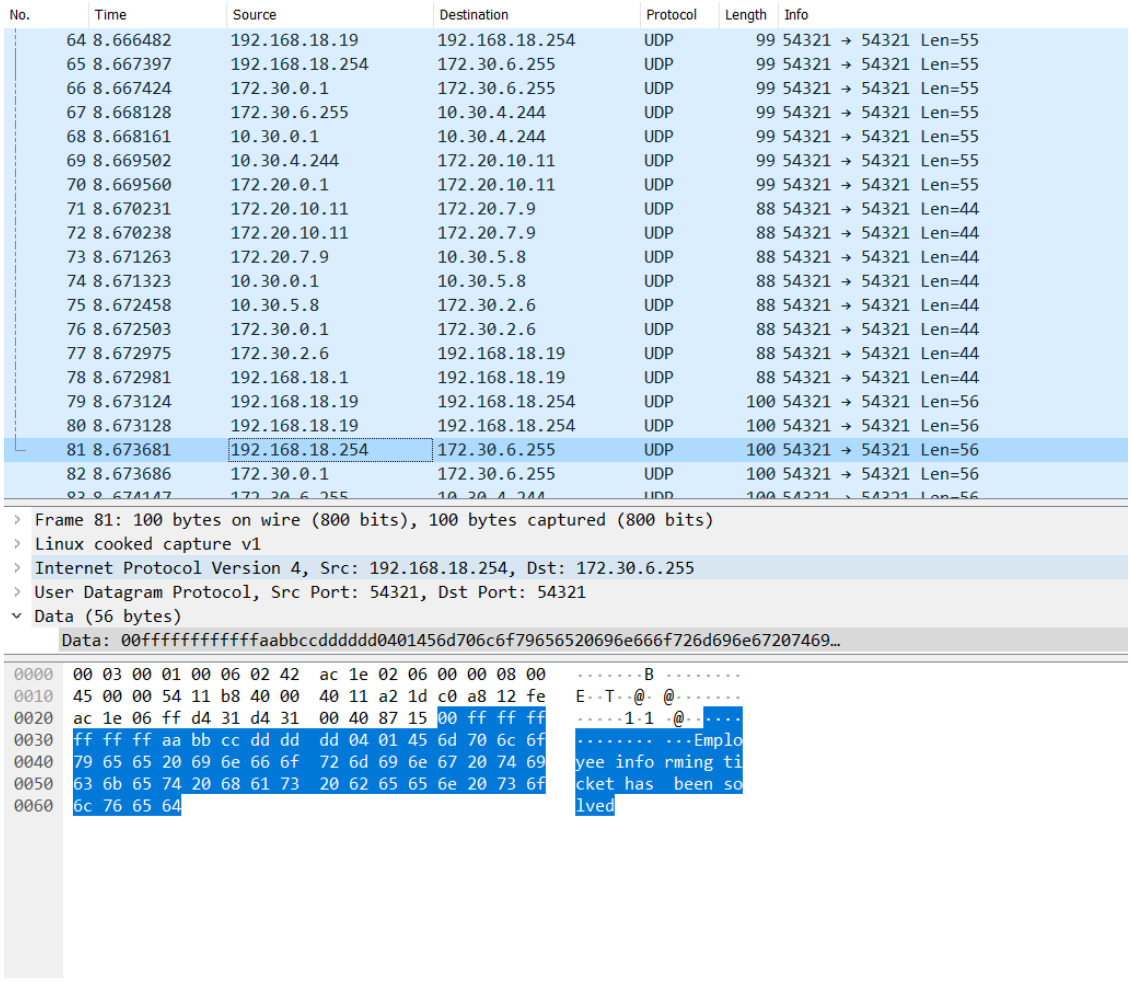


Figure 13: This shows a message being passed between forwarders in Wireshark, from the tcpdump's PCAP file. It starts with 0 to define that the packet is a packet and not a declaration, followed by the destination endpoint ID (FFFFFFFFFFFFF), the source endpoint ID (AABBCCDDDDDD), the byte identifying the packet type (ticket solution), a byte identifying the ticket to be solved, followed by an explanatory message only included in as explanation for any readers. This explanatory message would be removed before this system would be put into production.

5 Discussion

I am very happy with my solution to this assignment. I am happiest with the shortest path implementation, as this means that the solution can be very efficiently scaled up to larger topologies, ensuring that the most efficient paths are taken. This protocol can deal with $2^{8*6} = 2^{48}$ endpoint, which is certainly sufficient. However, there are some improvements I would make if continuing to work on this solution.
Implementing multiple ISPs would provide the opportunity to investigate using parts of the endpoint ID to identify where in the network the endpoint lies, with the remaining parts used for the specific identification. For example, all endpoints IDs based in one ISP could start with AABBCC, while those in another ISP could start with BBBBCC, so the forwarders could send the packets to those networks before using the end of the IDs to identify the specific endpoint.
It would be preferred to store the list of edges in a more efficient and organised way. As it is, the solution stores the edges as a list of tuples. It would have been preferred to do it more similarly to the way shown in the assignment brief, with a dictionary with the node ID as the key, and a list of nodes it can access as the value. On a similar note, it would be a good idea to store the specific IP addresses a node can access

rather than relying on matching the starts of IP addresses when the controller is sending back details of new endpoints.
Adding acknowledgement messages across the protocol would be a good addition. With this solution, declarations are assumed to arrive, and I never found this to fail, but it would increase the stability of the system. ACK messages could also be used to notice when forwarders go offline so the controller could reroute shortest paths.
However, overall, this is a good proof of concept for a flow-forwarding protocol using an efficient shortest path algorithm. The example topology can be very easily scaled up.

6 Reflection

I found this assignment very interesting to reason about, and fairly easy to understand the basics of. I appreciated the opportunity to create my own protocol from the ground up, and I believe it was a very good learning experience. I appreciated the opportunity the opportunity to use theory I have learned in previous years in the implementation of my solution. In future projects, I will refer back to the brief more often as when I read back over the brief when writing this report, I noticed some suggestions which would have made my solution more effective. Using python for this project worked very well, as it is a good language for rapid prototyping.
I would estimate this project took me about 30 hours, excluding writing the report.

7 Bibliography

[1] McKeown, N. et al. (2008) “OpenFlow,” ACM SIGCOMM Computer Communication Review, 38(2), pp. 69–74. Available at: https://doi.org/10.1145/1355734.1355746.
[2] Hasan, B. (no date) “Chapter 22: Floyd-Warshall Algorithm,” in Algorithms: Notes for professionals. Goalkicker.com.