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

CSU33031 Computer Networks

Assignment #2: Flow Forwarding Protocol

Abdelaziz Abushark, Std# 20332134

December 4th, 2022

Contents

1 Introduction

In this assignment we are tasked to design a protocol which implements our own version of the OpenFlow software defined networking standard. We are tasked to create a protocol consisting of End-Nodes, Switches, and a Controller in which the End-Nodes can send messages between each other. However, we must emulate the OpenFlow standard which means that we must send the payload packet to a switch which will then forward the packet to another Switch until the Switch forwards it to the destination End-Node.

I was able to set up the skeleton code for numerous classes that I thought I would need right away by using the knowledge I had gained from the previous assignment. I found design ideas for the packets from the various websites and lecture slides. That considerably aided my understanding of how the OpenFlow standard is put into practice. To better understand and appreciate the basic concept of software defined networks, I also conducted research on several websites regarding the Open-Flow standard. I tried to incorporate what I knew about the OpenFlow standard when I was building the protocol. After completing the fundamental implementation, I did some research on graphs, graph path finding methods, and particularly the Dijkstra algorithm.

The idea behind this assignment will first be described in this report. The implementation's specifics will next be discussed, followed by a look at the choices that were made when building this protocol. Finally, a summary and reflection on the entire task will be given.

2 Overall Design

This section details the features I used, how I interpret the Open-Flow standard, and the protocol. The fundamental idea that guides the protocol's design and execution will be covered in this part. The purpose of this section is to eliminate any requirement for the reader to infer anything about the concepts employed and to ensure that any reader can understand the material presented in this report without needing to consult further sources. The Internet Protocol, User Datagram Protocol, Software Defined Networking, and OpenFlow will all be covered in detail in this part. I'll describe my understanding of OpenFlow and SD-WAN in the first two sections. The explanation of how these two methods works will then be used to illustrate how I came up with the solution. Following an explanation of the design, a description of the packets that are exchanged between the various network nodes will be provided.

2.1 OpenFlow

A software defined wide-area network is a method of managing networks that enables programming of network settings and decouples the control plane from the data plane. This indicates that the data transferring requirements will continue to perform as intended, but that all network devices will be programmable and controllable from a single central control unit. Three fundamental elements make up the network's OpenFlow standard for software-defined networks. the End-Node, the Controller, and the Switch. The node that performs the bulk of the work is the Controller. All of the Switches are connected to the Controller. When a switch encounters a table miss and is unsure of where to route an incoming packet, it sends a packet in packet to the controller. The Switches can be programmed by the Controller. The Controller has access to a wide range of OpenFlow operations, including the ability to add and remove entries from the Switch's flow table. A flow mod packet is one that modifies the switch's flow table. Both the Controller and the Switch can send hello packets to one another to indicate that they are both online. The Switches' capabilities may also be requested by the Controller. To do this, the Controller sends a feature request packet, and the Switch responds with a feature outcome packet. There is not a lot for the Switches to do. To simply forward any incoming packets, they simply maintain a flow table of all the routes. If they don't have the entry in their flow table, they can ask the Controller for the next hop of the inbound packet. The Switches can convey information about their neighbors. The two primary tasks of the End-Nodes are to either print out the message that has been sent by another End-Node or send a message to a Switch, which will then pass it to the desired End-Node.

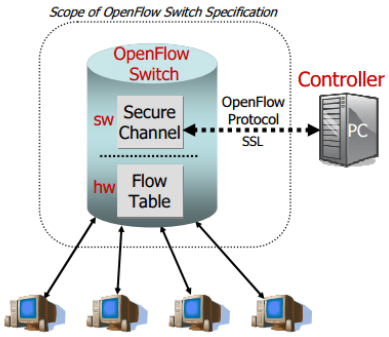

Figure 1: This diagram shows the general topology of an Open-flow network. A controller is linked to the switches in the network. When an incoming packet doesn't fit any rules in the flow table, the switch will get in touch with this controller.

2.2 Software Defined wide-area Networking

Software Moving away from conventional networking is Software Defined wide-area Networking (SD-WAN). In a network that is software-defined, a network engineer or administrator can control traffic from a central console without having to alter individual switches. Three "planes" make up the network: a data plane in the infrastructure layer, a controller plane in the application layer, and a controller plane in the control layer. The application layer includes common networking tools that a company or organization might utilize. SD-WAN control software is present at the control layer. This layer offers network services that control how much traffic is flowing through the system. The infrastructure layer is composed of actual switches, which act as network routers and direct traffic through the network along routes determined by the control layer. The application and control layers often have application programming interfaces (APIs) between them. Interfaces are required between the layers. A Control Data Plane interface, like OpenFlow, sits between the control layer and infrastructure layer.

a programmatic, automated method for controlling business network connectivity and circuit costs. By doing so, it expands software-defined Wide-Are Networks (SD-WAN), in which I explained earlier, into a tool that companies can use to swiftly build a smart hybrid WAN. With the help of SD-WAN, which combines wireless, broadband Internet, and IP VPN of the highest quality for business, you can manage applications more affordably, especially those in the cloud. Based on network conditions, requirements for the security and quality of service (QoS) of application traffic, and the cost of the circuit, traffic is automatically and dynamically forwarded across the most suitable and effective WAN path. The routing policies can be modified. Every public and private line service is monitored by SD-WAN software operating on CPE (customer premises equipment), which also decides how to route each type of application traffic. For instance, voice-over-IP (VoIP) traffic might be sent by default over an MPLS VPN service. The SD-WAN, however, may divert that traffic to a broadband Internet or 4G LTE wireless circuit if the MPLS connection becomes overloaded. For optimal performance and cost-effective routing, the SD-WAN offers automatic load balancing and network congestion control.

2.3 ADSL routers

Asymmetric Digital Subscriber Line, or ADSL, is a well-known, more traditional form of broadband. It is a type of broadband connection that utilizes the copper wires of existing phone lines and is primarily utilized for home and small business broadband. For these firms, ADSL is an excellent entry-level broadband option that makes budgeting simple and won't significantly affect finances or forecasts. It may be more than sufficient for your requirements with download and upload rates of up to 24 Mbps and 8 Mbps, respectively. When this is the case, saving money by not purchasing more than you require and investing it back into your success is a sensible business move.

2.4 Design of Network Elements

I have tried to follow the standard in my implementation of OpenFlow. However, there are a lot of adjustments or features that are lacking because it would be very challenging for me to implement OpenFlow completely on my own. In which I designed a controller, application, forwarding service and switches (routers) that I talked about more in my implementation section.

2.4.1 Design of the Protocol

The design of the protocol's nodes is comparable to how I previously presented my understanding of OpenFlow. To create the path to the provided destination, the protocol makes advantage of Hard coded routing table in which it uses the latency between nodes as the weights of the graph to determine the shortest path to the destination.in which Controllers updated the tables every time there's a new entry and send new updated tables to switches to inform them about their next destinations followed by acknowledgments which I explained in communication and packet description section.

2.5 Communication and Packet Description

I have developed a wide variety of packet types that are used to send data between network nodes. The request packet is used to request that the Switches send the Controller their specs. When constructing a graph of the network configuration, the switch result packet, which contains data about the Switches, is employed. The flow tables of the Switches can be changed using the flow mod packet. The purpose of the packet-in-packet is to query the Controller as to which hop the payload packet must make to reach its destination. The message, together with the switch's and destination node's names, are all contained in the payload packet. The packet that is forwarded among network nodes is this one. The switches are informed by the unknown destination packet that the destination does not exist or that no path could be found. The switches are initialized, and the version number is configured using the "sign" packet. The various packet types will now be thoroughly explained in this section. The network traffic between network nodes is visualized in the flow diagrams that follow. It displays a series of communications that took place between the components.

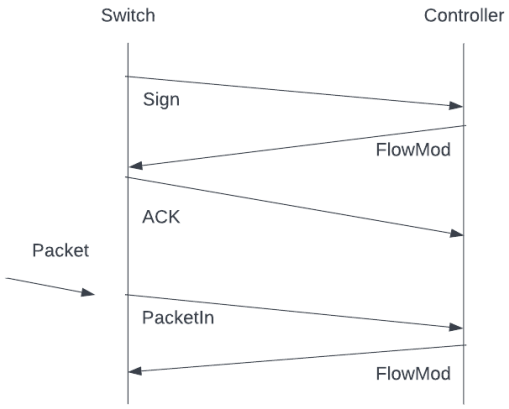

Figure 2: The diagram illustrates a switch introducing itself to the controller and receiving a sign to modify its flow table. It responds to this by acknowledging it. A switch will get in touch with the controller and get an additional flow table modification when it gets a packet with an unknown destination.

Figure 2 which is the Flow Diagram shows the communication between components of my protocol in which now I will go and talk about the different type of packets that was exchanged during my protocol.

2.5.1 Packet Classes

The method Packet is used to encode the packet classes. By doing so, we can generate a byte array that can later be utilized to create a datagram packet. A description of the packets transferred between the controller and switches will be given. This protocol implements five different message types: sign packets, flow-mod packets, employee packets, payload packets and error packet. These packet types are divided into two kinds in OpenFlow: symmetric, and controller-to-switch messages. Each of these is illustrated in detail in this procedure.

Sign (empty) Packet

This is a welcome or keepalive message sent from the switch to the controller, or vice versa. These packets are considered symmetric messages in OpenFlow because they are transmitted in both directions. For the sake of simplicity, this protocol only sends the information from the switch to the controller upon switch startup. This packet just contains the header, which is one byte with the value 0 in it.

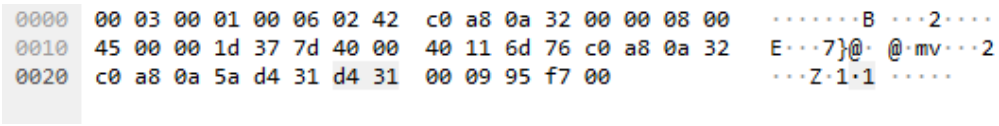

Figure 3: sign packet captured using Wireshark.

Flow-Mod Packet

The controller sends this message to the router to add, remove, or update the router's flow table. This is categorized as a controller-to-switch message in OpenFlow. In this protocol, the forwarding table data of the switch is used to identify packets delivered from the controller to the switch. The payload, which makes up the remaining portion of this packet, can be any length depending on how long the forwarding table is being transmitted. To conclude, A Switch's flow table can be modified using the flow mod packet. The destination and the following hop are two strings found in the flow mod packet. The Switch updates its flow table after receiving a flow mod packet, using the destination as the key.

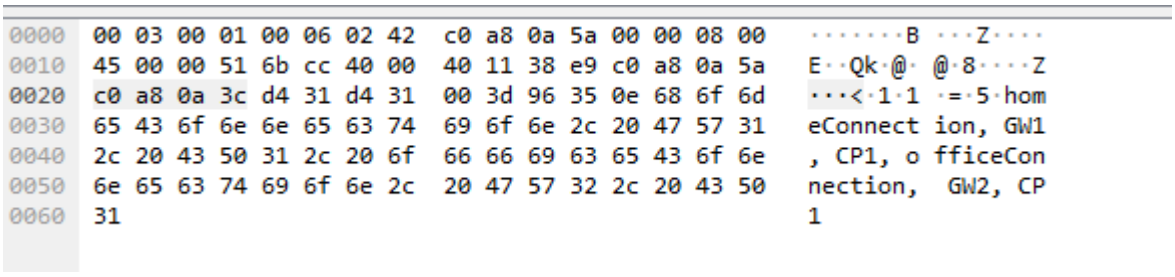

Figure 4: Flow-Mod packet captured using Wireshark

Employee Packet

The first byte encodes the packet type, the second byte encodes the value's length, and the subsequent bytes encode the value itself. The destination in this instance serves as the value. For these messages, only the GIVEN ID packet type, which is decoded as 1, is accepted. The payload, or message the user intends to send over the network, is included in the remaining packets of these datagrams.

```
0000  00 03 00 01 00 06 02 42  c0 a8 0a 46 00 00 08 00   ·· ·· ·· B ··· F ····
0010  45 00 00 3a db 54 40 00  40 11 c9 9f c0 a8 0a 46   E ·· : ·T@ · @ ····· F
0020  c0 a8 0a 28 d4 31 d4 31  00 26 95 f6 01 0e 68 6f   ··· ( ·1 ·1 · & ···· ho
0030  6d 65 43 6f 6e 6e 65 63  74 69 6f 6e 68 65 6c 6c   meConnec tionhell
0040  6f 20 44 53 65 72 76 65  72 31                    o DServe r1
```

```
0000  00 04 00 01 00 06 02 42  c0 a8 0a 14 00 00 08 00   ····· ·· B ····· ···
0010  45 00 00 3c c5 da 40 00  40 11 df 3f c0 a8 0a 14   E ·· < ·· @ · @ ·· ? ····
0020  c0 a8 0a 32 d4 31 d4 31  00 28 95 d0 01 10 6f 66   ··· 2 ·1 ·1 · ( ··· ·of
0030  66 69 63 65 43 6f 6e 6e  65 63 74 69 6f 6e 68 65   ficeConn ectionhe
0040  6c 6c 6f 20 44 53 65 72  76 65 72 32               llo DSer ver2
```

Figure 5: Employee Packet captured using Wireshark

Payload Packet

The payload, and packet's destination, is contained in the payload packet. Both the Switches and the End-Nodes which is application in my implementation send this packet. The packet that is transmitted from Switch to Switch and then to the target End-Node is this one.

```
0000  00 04 00 01 00 06 02 42  c0 a8 0a 5a 00 00 08 00   ····· ·· B ··· Z ····
0010  45 00 00 37 f1 98 40 00  40 11 b3 22 c0 a8 0a 5a   E ·· 7 ·· @ · @ ·· " ··· Z
0020  c0 a8 0a 50 d4 31 d4 31  00 23 96 2f 0e 6f 66 66   ··· P ·1 ·1 · # · / · off
0030  69 63 65 43 6f 6e 6e 65  63 74 69 6f 6e 2c 20 4d   iceConne ction, M
0040  4c 50 2c 20 49 53 50                              LP, ISP
```

```
0000  00 04 00 01 00 06 02 42  c0 a8 0a 5a 00 00 08 00   ····· ·· B ··· Z ····
0010  45 00 00 35 a5 99 40 00  40 11 ff 41 c0 a8 0a 5a   E ·· 5 ·· @ · @ ·· A ··· Z
0020  c0 a8 0a 32 d4 31 d4 31  00 21 96 0f 0e 68 6f 6d   ··· 2 ·1 ·1 · ! ··· hom
0030  65 43 6f 6e 6e 65 63 74  69 6f 6e 2c 20 43 4c 50   eConnect ion, CLP
0040  2c 20 49 53 50                                    , ISP
```

Figure 6: Payload Packet captured using Wireshark

Error Packet

When a switch(router) is unable to determine the destination, it sends this packet to the controller so that it can determine the next destination.

```
0000  00 04 00 01 00 06 02 42  c0 a8 0a 32 00 00 08 00   ·······B ···2····
0010  45 00 05 6c 88 b7 1f 13  40 11 37 da c0 a8 0a 32   E··l···· @·7····2
0020  c0 a8 0a 5a 00 00 00 00  00 00 00 00 00 00 00 00   ···Z···· ········
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
```

Figure 7: Error Packet captured using Wireshark

3 Theory of Topic

This section will cover Internet protocol, The User Datagram Protocol, Header design and information, Deployment and Topology that I used to implement in this assignment, and I'll now go into the theory that underlies the flow forwarding protocol I created. You'll be able to comprehend some background information from this that will help you comprehend how I've implemented the protocol.

Overall Topology

This assignment took a significant amount of time because I had to research the theory to implement my approach. I'll lay you the idea behind my answer to prevent any misunderstandings.

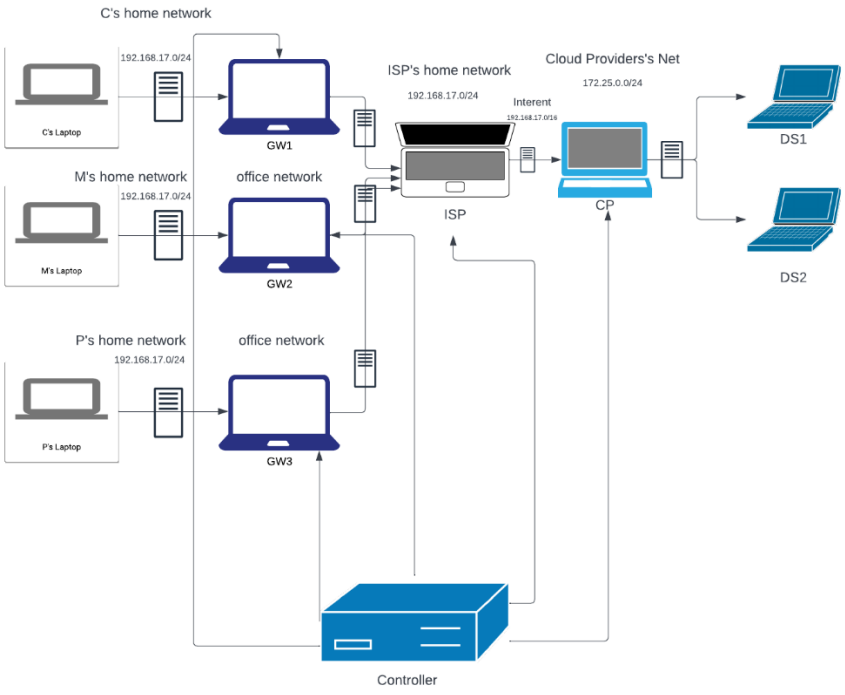

Figure 8: Topology used to implement my protocol

Figure 8 explains my Topolgy in which I call it the final complex topology as it includes a various number of employees with their home networks and number of applications servers in their own network. This topology was realized in docker by setting up several containers and networks. As you can see It has number of employees, network of a provider and a network for a cloud provider with an application server in the same network.

3.1 IP, Ports and Sockets

The network layer of the Open Systems Interconnection (OSI) paradigm is what we're talking to when we talk about the Internet Protocol (IP). IP is used to transport packets, which are identified by IP addresses, from a source host to a destination host. Whether IPv4 or IPv6 is utilized, IP addresses are a distinct and universal sequence of 32 or 128 bits. An endpoint for transmitting and receiving packets over a network is a socket. A communication endpoint is symbolized by a port. A communication endpoint is symbolized by a port. A port number is used to identify a process or service, whereas an IP is being used to distinguish a host in a network. In this method, my protocol and the forwarding service are identified by the port number, which is always 54321. An endpoint for transmitting and receiving packets over a network is a socket. Each end node, router, and controller instance have a separate socket in this implementation. Port 54321 is used for receiving inbound packets.

```
Src Port: 54321, Dst Port: 54321
```

Figure 9: shows the source and destination port as 54321

3.2 UDP, Datagrams and Forwarding services.

A protocol that operates on top of IP and is a component of the transport layer in the OSI model is the User Datagram Protocol (UDP). Due to its unreliability, it is connectionless and mostly used for low-latency and loss-tolerating connections. It is more appropriate for applications that can offer their own flow or error control because neither are built in. Datagrams are the term used to describe packets transmitted through UDP. A header and a data segment make up a UDP datagram which is shown in figure 10.

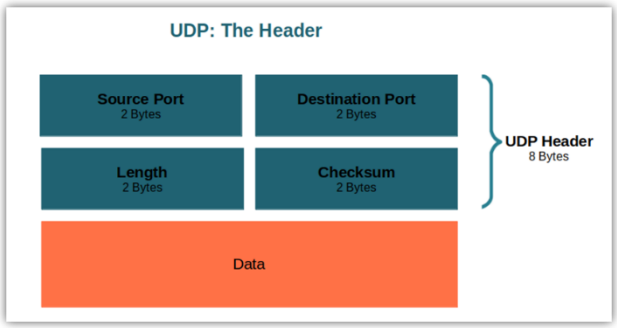

Figure 10: the fundamental make-up of a datagram or UDP packet. The source port number, destination port number, packet length, and an optional checksum field are the four components that make up the header. The payload to be transported makes up the remaining portion of the datagram.

In my implementation I designed an application shown in figure 11 that will transmit UDP datagrams to a local host forwarding service. The forwarding service will look for the destination in a table using the

header information from your protocol. then it passes it to another instance of the forwarding service on this IP address after the table has given it the IP address of the network element that is the next hop. My implementation of the forwarding services for the overlay will use a UDP socket bound to port 54321 which I mentioned in section 3.1 at every network element.

An application that wants to accept network traffic from the overlay will send a datagram to the nearest forwarding service, informing it that it wants to accept traffic for a specific ID. The port number and string that were used by the application must be recorded by the forwarding provider. It should deliver any datagrams that come in for the specified ID to the port that corresponds to the ID which is shown in figure 12.

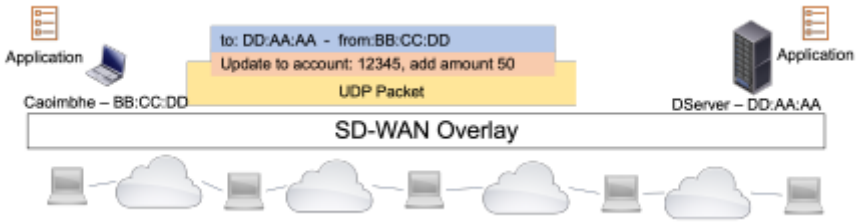

Figure 11: shows overview of the application that sends UDP datagrams

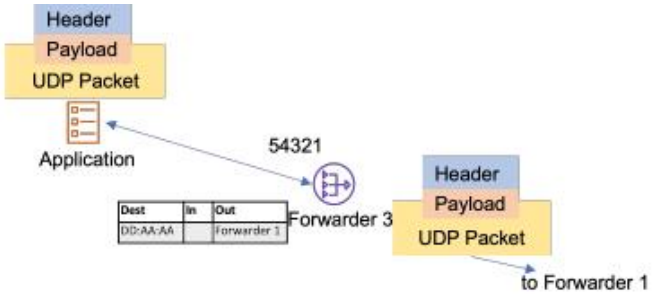

Figure 12: shows overview of the application when transmitting a header and payload to a forwarding service bound to port 54321.

Accepting incoming packets, checking the header information, checking the forwarding table, and sending the header and payload information to the destination are the fundamental functions that a forwarding service must offer. And if the destination is not known, forwarding service will drop an incoming packet and will contact the controller to enquire about a forwarding information to the unknown destination which is integrated in the forwarding table. Figure 13 helps the reader visualizes the main functions of forwarding service.

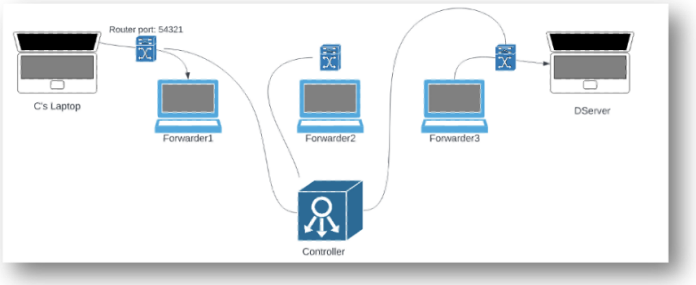

Figure 13: Forwarding Service

3.3 Header Design

The header design for this protocol was significantly simpler than it was for the prior assignment. The protocol required destinations of different lengths to be encoded in the header, therefore storing the length of the value was a sensible way to accomplish this. For packets exchanged between switches and the controller, the header design. I made the decision to keep them as straightforward as possible by only having the first byte serve as the packet type and the remaining data serve as the payload. I believe that this is sufficient for the purposes of this assignment and adding a more intricate header would merely add to the overhead with no discernible speed advantages.

3.4 Application

Figure 14: Overview of the Application

An application will transmit UDP datagrams to a local host forwarding service. The forwarding service will look for the destination in a table using the header information from your protocol. Your implementation should pass it to another instance of the forwarding service on this IP address after the table has given it the IP address of the network element that is the next hop.

3.5 Switch

Incoming packets are accepted by switches, also known as ADSL routers, which examine the header data, check the forwarding table, and then send the header and payload data to the destination. They are essential because they are the parts that let us forward the packet flow between end users. They speak with the controller to ask questions concerning information transmission.

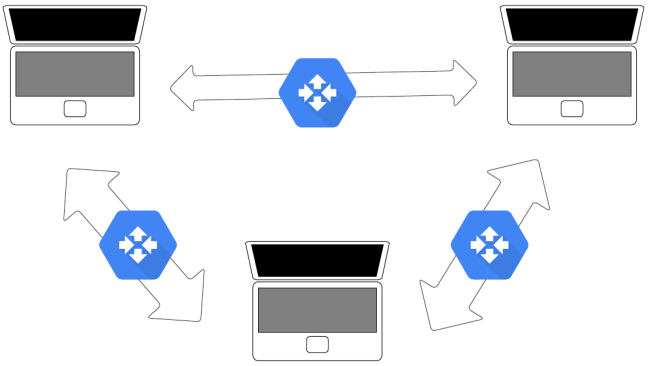

Figure 15: shows employees with switches allowing them to communicate amongst them.

3.6 Controller

By implementing the flow tables of the network and informing them of routes to destinations as the network evolves, a controller manages the network elements. The controller is also essential since it houses the table used by the switches to compile forwarding data and forward packets, enabling communication

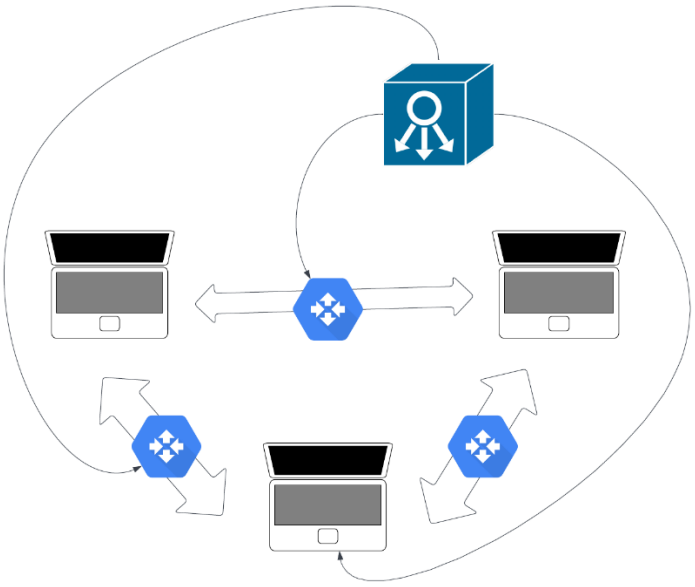

Figure 16: shows employees who are equipped with switches and receive route information from a controller.

3.7 Forwarding Service

The application must have access to a flow forwarding service in order to ask for packets to be forwarded from end-node to end-node and vice versa. The header information will be used by the forwarding service to look up the destination in a table and decide where to send the datagram. To complete this assignment, we must guarantee that the service is tied to port 54321.

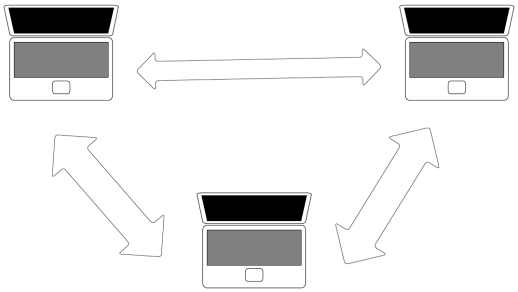

Figure 17: shows employees communication via forwarding service

4 Implementation

I will be talking about the protocol's implementation in this section in details Python was used to implement the protocol as I found that it has so many libraries that will help me finish this assignment faster and more efficiently and I took some code from my from my Assignment 1 (File Retrieval Protocol) while Docker, which I talked in detail about it in Discussion section, was used to stimulate the components and then I will explain Python classes that I have used.

The 5 classes were created by myself using Python as this was not difficult as I already done similar implementation from the previous assignment and Python already has in built libraries and modules such as socket module to enable program communication. There could only be 2 end-nodes and forwarding service for the first implementation. After completing this simple design solution, I made the decision to focus on implementing switches and a controller that control the network. Application code was modified such that it now has direct connection with switches in which directs the next destination for the next packet (message) which was connected directly by the controller who was controlling the network. My final implementation contains a complex Topolgy which Stefan gave us and I tried my best to follow it which explained in section 3 Theory of Topic which includes a number of employees with their home networks and a number of application servers in their own network.

4.1 Packet.py

These functions below specify just what would occur whenever the component gets a packet, has several implementations depending on the class. Additionally, they provide constants that are utilized by its three subclasses and includes several fundamental assistance functions. For instance, the protocol's port number and the variables used for various OpenFlow packet types are defined here.

```
def ByteLoad (self,packet):
      b = packet.countBytes()
      result = [self.header+ b.length]
      return result
    def countDes(self, packet):
      data: object = packet.getData()
      b= []
      des = str(b)
      return des
def receivePacket(self,data:list,
packet:socket):
     infoPacket = data
      b = [packet.getLength() - list]
      received = str(b)
      return received
```

Listing 1: Packet.py

4.2 Forwarders.py

This class in particular services as a forwarding service between the employees that uses the application within their home networks. Functions below carry the message between the employees and forward it between the end users. In addition, these functions accept incoming packets, inspect header information, consult forwarding table and forward the header and payload information to the destination and if the destination is not known, a forwarding service will contact the controller to enquire about a forwarding information to the unknown destination and if it receives forwarding information, it will integrate it into its forwarding table.

```
def forward(self , message):
     message.countDown()

def transfer(self, message):
if (not(des(packet))):
print("packet was dropped due to no existent
destination")

     try:
        self.message.wait()

        while True:
           sign = recevied([self.size])
           self.socket.receive(sign)
           forward(sign)

     except Exception as w:
        if (not (type(w)== countDes(self))):
           print(end=2)
```

Listing 2: Forwarders.py

4.3 Application.py

This class is like the backbone of my implementation as different employees interact with to send or receive messages across the network. The user in other words the employee will first be prompted when the application is launched as to whether it should send a message or receive a message. If the application is designated as the end node for a certain destination string and the user chooses to receive a message, it

will just sit and wait until a packet arrives to it. Otherwise, the user will be invited to input the address to which they want to give a message if they choose to do so. It should be noted that there is currently no error handling that can detect whether a destination exists or not. This is so that the switch can handle it, which I will talk about in Switch.py. The client will then be prompted for the message they wish to send. In dissection section I will explaining how the communication was via Application.

```
from socket import socket

class Application ():


   def userEndInteraction(self, packet):
      data = packet.getData()
      message = getMessage(data, packet)
      print("Receive message:  " + message)
   def userStartInteraction(self):
      d = input("Forwarder: Enter the
destination for the message : ")
      m = input(f"Forwarder: Enter the
message  + {d}  :")
      addreess = socket("GW1", port)
      packetA = format(data, data.length)
               socket.send(packetA)
      addreess = socket('GW2', port)
      packetB = format(data, data.length)
      socket.send(packetB)
      print(f"Message {m} +was sent to {d}")
   def applicationRun(self):
      while (not executed):
         print("Forwarder: Enter SEND or
RECEIVE : ")
         value = input()
         if value.equalsIgnoreCase('SEND'):
            self.sendMessage()
         elif
value.equalsIgnoreCase("RECEIVE"):
            print("Waiting to receive the
message")
            self.wait()
         else:
            print("Error !! ")
```

Listing 3: Application.py

4.4 Switch.py

The listing below has the functions that carry out the procedure I will explain in this paragraph. Receiving incoming packets, inspecting the header, and then forwarding the packet to the next switch or end node based on the location contained in the header constitute the switch's fundamental operation. Although these network nodes are known to as switches in OpenFlow. A forwarding table is unique to each router(switch). This forwarding table informs the router of the location of the packet's next hop. Three columns make up the structure. The first is the text that the user chose as the packet's destination. When the switch software is launched, it initially displays out its present forwarding table, which ought to be blank since the router has not yet established communication with the controller and as a result is unaware of the network. The router will then proceed to send the controller a "Hello" packet. Which in my implementation called “Sign” Packet then the controller and router are simply introduced to one another at this initial hello. The forwarding table for that router is sent out to the router in the form of a "Flow Mod" packet by the controller in response to receiving this greeting packet. At this point, the switch has knowledge of the network's paths and knows where to direct packet data. A switch will first attempt to determine which network element very next destination is when it gets a message packet, which is

recognized if the packet in order to accomplish this, it will extract the packet's destination After looking through its forwarding table it will verify that the element the packet originated from matches.

```
                def ACK(self):
                  data = bytes()
             data[TYPE] = self.header
                        data =
      InetSocketAddress("controller",
        printForwardingTable(self))
                    packetRec =
       property(socket.sendto(data))
            socket.send(packetRec)

       print("Sign was sent to controller")

def updateForwardingTable(self, packet):
             data = packet.getData()
           b = [packet.getLength() - 1]
              tableUpdate = str(b)
      switchTable = tableUpdate.split(", ")

     controllerTable: object = [switchTable
                  / 7][7]

            def nextDes(packet):

            print("Final destination")
             print("Packet source :")

                 for m in range(0,
            controllerTable.size):
                   if destination ==
             controllerTable[m]:
            if controllerTable[m] == bytes:
                return forwardingTable[n]

           return "it does not exist !"

  def controllerDevice(self, packetSource):
       finalDes = packetSource.getData()
                finalDes[e] = range

                  addressCont =
     InetSocketAddress("Controller",
                packetSource)
                    nextPacket =
         socket.send(packetSource)
             finalDes[e] = nextPacket

                  addressCont =
    socket("Controller",packetSource)

            socket.send(nextPacket)

      print("Error. Next destination cannot
     be found !!  forwarding packet to
                controller")
```

Listing 4: Switch.py

4.5 Controller.py

When it comes to SD-WAN, one of the crucial network elements is the controller. As implied by the name, the controller manages the network components, in this protocol simply referred to as the routers(switches). It accomplishes this by setting up each router's flow table from scratch and updating route destinations as the network evolves. The controller has a table that stores all the details on the routes in a network. It also has an overall view of the network. This table's six columns list a packet's source and its hops before it is routed to a particular destination. The controller will print out its most

recent view of the network when it first launches. After then, it will just wait till it gets a package. A "Hello" which is “Sign” packet in my implementation from one of the routers will be the first packet it receives. The controller will respond by returning the forwarding table for that router after receiving this. This conversation can be viewed as the router initializing and registering with the controller. As previously noted, the controller has one sizable database, but instead of transmitting the entire table to the router, it will just extract the data pertinent to that router. The simplest way to convey this is through an example. when a switch first activates, and the controller receives a "Hello" packet. The controller can then obtain the name of the router. The controller then does a loop around the array containing the data from the database, looking for any rows that contain the desired destination. The destination address, the router in, and the router out are added to a different table, which is shown as an Array List of strings, if there is a match. The router (switch) will receive this table and process it before updating its own flow table with the destination, router in, and router out information. This protocol also has a feature that causes packets to be dropped if the destination cannot be determined. For instance, the end user might specify that they want to send a message to "officeConnection"; this packet is subsequently routed to the network's first router. The router will then search its forwarding table for that destination string. When it discovers that there is no match, it will get in touch with the controller and inquire as to where the packet should be forwarded after that. To accomplish this, it takes the packet that was received, modifies it and then sends the packet to the controller. The controller recognizes the packet as a request from the router (switch) to attempt to establish the next hop when it is received. Next, the controller looks up the destination in its database; if it can't be found, the packet is dropped; otherwise, the controller sends the routers an updated flow table. Because in OpenFlow it is used to hand over control of a packet from the gateway to the controller if the gateway is unsure what to do with it. The listings below has the functions that carry out the procedure I have just explained in the above paragraph.

```
    controllerTable: list[Union[list[str], Any]] = [
      ['homeConnection - >','CLP - >','GW1 - >','CLP-
                >','ISP- >','CLP'],
       ['homeConnection - >','CLP - >','ISP - >','GW1-
               >','FWR- >','CLP'],
       ['homeConnection - >','CLP - >','FWR - >','ISP-
                >','DS1- >','CLP'],
       ['homeConnection - >','PLP - >','GW2 - >','PLP-
                >','ISP- >','PLP'],
       ['homeConnection - >','PLP - >','ISP - >','GW2-
               >','FWR- >','PLP'],
       ['homeConnection - >','PLP - >','FWR - >','ISP-
                >','DS2- >','PLP'],
      ['homeConnection - >','MLP - >','GW3 - >','CLP-
                >','ISP- >','MLP'],
    ['homeConnection - >','MLP - >','ISP - >','GW3-
               >','FWR- >','MLP'],
       ['homeConnection - >','MLP - >','FWR - >','ISP-
                >','DS1- >','MLP'],
      ['homeConnection - >','ALP - >','GW4 - >','PLP-
                >','ISP- >','ALP'],
       ['homeConnection - >','ALP - >','ISP - >','GW4-
               >','FWR- >','ALP'],
       ['homeConnection - >','ALP - >','FWR - >','ISP-
                >','DS2- >','ALP'],
       ['homeConnection - >','SLP - >','GW5 - >','CLP-
                >','ISP- >','SLP'],
       ['homeConnection - >','SLP - >','ISP - >','GW5-
                >','FWR- >','SLP'],
       ['homeConnection - >','SLP - >','FWR - >','ISP-
                >','DS1- >','SLP'],
      ['homeConnection - >','WLP - >','GW6 - >','PLP-
                >','ISP- >','WLP'],
       ['homeConnection - >','WLP - >','ISP - >','GW6-
               >','FWR- >','WLP'],
       ['homeConnection - >','WLP - >','FWR - >','ISP-
                >','DS2- >','WLP'],
```

Listing 5: shows the Controller table

```
  def controlTable(self, packet):

        info.self=  packet.getInfo()
        packetSource =  packet.findAdd()

        if info == signPacket:
           print(f"Sign was received from Switch
{packetSource}")

        elif data[TYPE]== emptyPacket
           print(f"packet was received with
unclear destination  {packetSource}")

           if searchDes(packet):
              return
           print(" packet was dropped")

        else:
           print(f"unexpected packet was received
{str(packet)}")

  def lookUpDestination(self,packet):

     print("Looking for the destination")

        if des ==  controllerTable[i][DEST_ADDR]:
           print("Creating new table")
           return True


     return False

  ]
```

Listing 6: Controller.py

4.6 User Interaction

I designed my program to have as many containers as I want but for this assignment I used 9 containers, some for switches, some for employees and one for controller. Once the user starts the programs, forwarding service will ask the user if they want to send or receive a message (packet) then after picking send , you will be asked to send a message then you will be asked the content of the message and the destination of the message and then your message will be sent but if you picked receive you will wait until another employee sends a message to be received by the end node of the other employee . During this procedure a controller will be controlling the network and a router will be consulting the controller for the next destination of the packet.

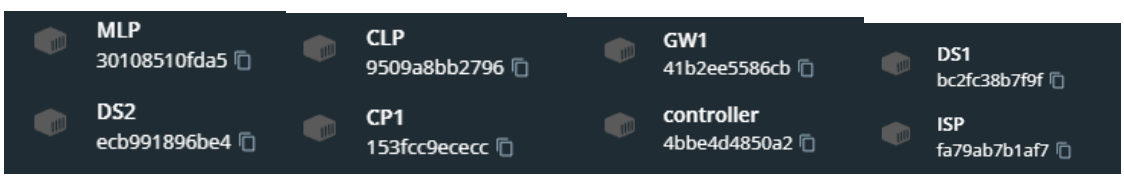

Figure 18: shows some of the containers used to develop this assignment

4.6.1 Terminal Usage (Full Topology)

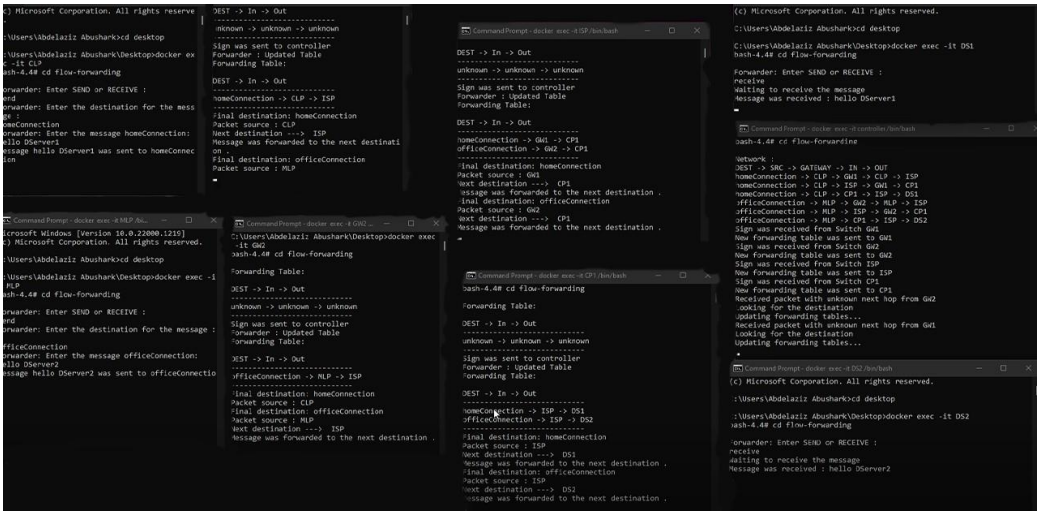

Figure 19: shows the whole Topolgy that includes number of employees with their home networks and a number of applications serves int their own network many using virtual infrastructure, as you can see an application asks the user to wither send or receive a message and then it proceeds with the procedure using OpenFlow protocol that I have explained earlier in my report.

4.6.2 Detailed Terminal Usage

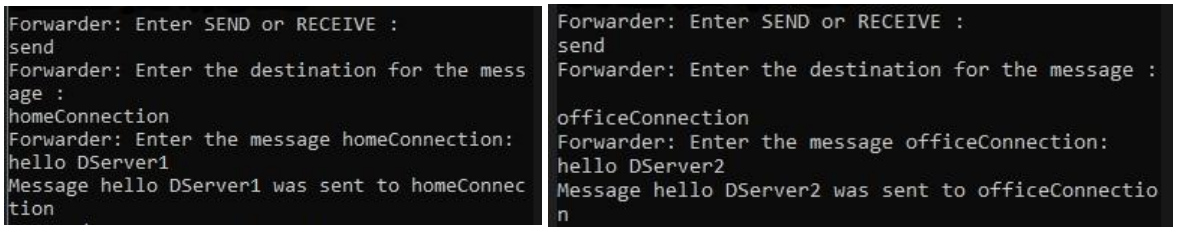

Figure 20: shows when the application asks the employee (user) to either send or receive a message, as the user picked send, it will asl him for the destination of the message and then the content of the message. Finally, it informs him that the message was sent to the chosen destination.

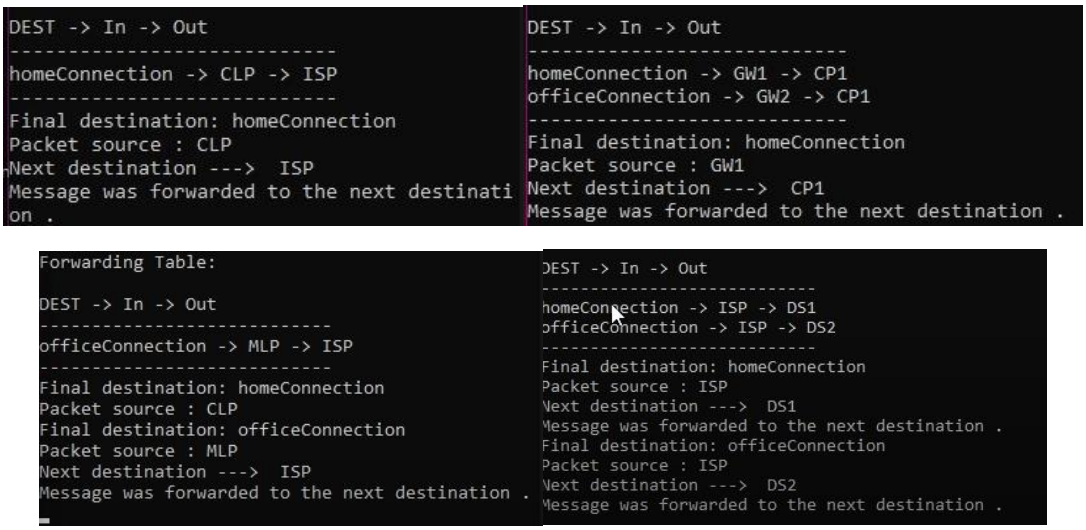

Figure 21: shows the switches (routers) receiving the message and forwarding it to the next destination and updating the routing table by the help of the controller.

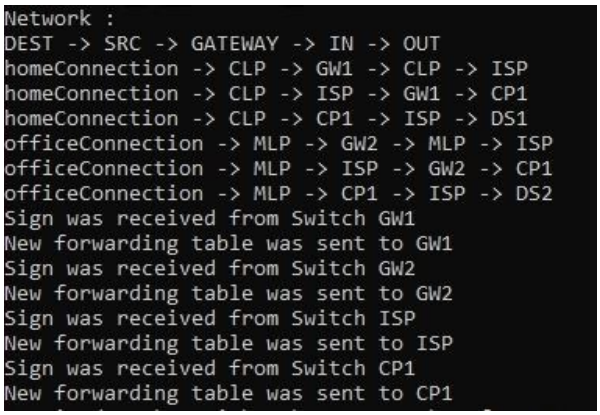

Figure 22: shows the Controller with the hardcoded table being consulted by the switches (routers) and sending new forwarding tables to switches.

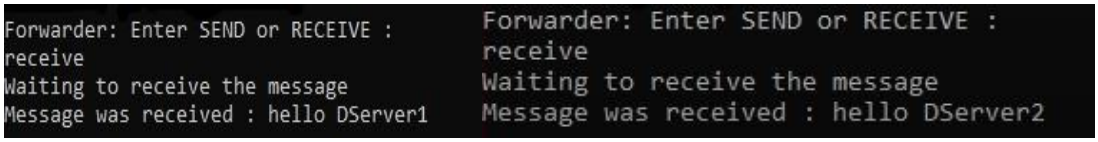

Figure 23: shows the application when the user picks the option (receive) in which will receive the message that was being forwarded from the first employee (user) to the end employee (user) by switches consulting the controller.

Figure 24: shows in case of an error or an unknown destination, packet will be sent back to the controller and if it does not find destination for it. It will be dropped immediately.

4.7 Deployment and Network capture

4.7.1 Docker and Wireshark

In my system, each component in my program such as switches, controller, end users (employees) run within a separate Docker container, which is powered by Docker. The key benefit of this is that any container will have a distinct IP address, enabling accurate simulation of the various network components. As well as it can run and support any number of switches and end nodes (employees) with their own distinct address. To examine the protocol's functionality using data packets, Wireshark was used to capture the traffic in which the communications between client, server and workers were visible to us.

Example of the network capture:

No.	Time	Source	Destination	Protocol	Length	Info
86	293.970794	192.168.10.50	192.168.10.90	UDP	45	54321 → 54321 Len=1
87	293.970806	192.168.10.50	192.168.10.90	UDP	45	54321 → 54321 Len=1
88	293.987997	192.168.10.90	192.168.10.50	UDP	69	54321 → 54321 Len=25
89	293.988014	192.168.10.90	192.168.10.50	UDP	69	54321 → 54321 Len=25
116	344.024012	192.168.10.80	192.168.10.90	UDP	45	54321 → 54321 Len=1
117	344.024022	192.168.10.80	192.168.10.90	UDP	45	54321 → 54321 Len=1
118	344.026192	192.168.10.90	192.168.10.80	UDP	71	54321 → 54321 Len=27
119	344.026204	192.168.10.90	192.168.10.80	UDP	71	54321 → 54321 Len=27
142	439.324983	192.168.10.60	192.168.10.90	UDP	45	54321 → 54321 Len=1
143	439.324996	192.168.10.60	192.168.10.90	UDP	45	54321 → 54321 Len=1
144	439.326816	192.168.10.90	192.168.10.60	UDP	97	54321 → 54321 Len=53
145	439.326824	192.168.10.90	192.168.10.60	UDP	97	54321 → 54321 Len=53
178	1088.056885	192.168.10.70	192.168.10.90	UDP	45	54321 → 54321 Len=1
179	1088.056897	192.168.10.70	192.168.10.90	UDP	45	54321 → 54321 Len=1
180	1088.058778	192.168.10.90	192.168.10.70	UDP	97	54321 → 54321 Len=53
181	1088.058789	192.168.10.90	192.168.10.70	UDP	97	54321 → 54321 Len=53
200	1138.567297	192.168.10.10	192.168.10.50	UDP	74	54321 → 54321 Len=30

Figure 25: shows the traffic that was captured using Wireshark.

5 Discussion

In this section I am going to discuss why I decided to use this approach, why did I choose docker as my virtual interface and why I decided to choose these specific functionalities to implement in my program. In addition to raising some thoughts and questions about Wireshark and traffic that was captured by it.

5.1 Why Docker

I used Docker instead of local host as I wanted to learn how to create containers and images via dockers as my I made the components of my protocol to connect with each other using Docker desktop once I start the command line writing "docker start – i component" it will start working and program will start executing waiting for user interaction. Communication between the components was happening via Docker Desktop, there is communication between end nodes, controller, and switches. As When the Application is operational, user can send and receive different messages and then controller sends the table to switches in which will transfer the message to the next destination.

5.2 Why I used these Functionalities

5.2.1 OpenFlow and SDWAN

In an established network, OpenFlow offers a simple method of communication between a controller and a switch. fits the criteria, Switches can use all their hardware resources in just forwarding data rather than computing routes thanks to SD-WAN's ability to separate the control and data planes. OpenFlow is supported by most modern devices, but it is not enabled by default. We can easily enable it and use it to switch to SD-WAN. A switch's configuration DOES NOT change as a result of OpenFlow. It merely updates the flow tables, which specify a packet's path.

5.2.2 Flow Table

The assignment is currently implemented using a flow table for each Switches based on the Controller's preconfiguration data. The flow tables might alter as the program goes on and Input from the user, for instance, could be used by the controller to define a new path between switches. Another illustration is the potential removal of a Switch from the network if it is not directly linked to an EndNote. The Controller would then need to create new paths in place of the network's existing paths that included the removed node.

5.3 Other Design decisions

The network routes were done using hard-coded table and linked with controller and switches. Initially, I intended make it hardcoded for the flow table, so that the controller could determine the shortest way between the end nodes once it had a view of the network elements.

5.4 Features

This section will briefly discuss some of the key aspects of this protocol that go beyond the essential specifications listed in the assignment description.

- support for numerous destinations and the use of various router paths according to the destination. Depending on who the user wishes to send a message to, each end node can perform both sending and waiting for a packet.
- When a router is unsure about the next destination for a packet, it first gets in touch with the controller to try and determine the destination. As a result, the router either drops the packet or updates its forwarding table.
- can dynamically add more switches to the network.
- allows only one Controller to be used.

6 Summary

This report details my attempt to develop a customized Python implementation of the OpenFlow Protocol. An Application Class, Forwarders class, a Controller class, a Switch class, and a Packet class made up most of my implementation.

Application Class: a class that asks the user about whether to send or receive a message and then the message will be forwarded using forwarders class.

Forwarders Class: acts as a forwarding service between several end nodes (employees) and UDP datagram will be sent through this class with a UDP socket bound to port 54321.

Controller Class: a network-agnostic class with a hard-coded flow table that switches can use to determine where to forward packets.

Switch Class: a class that transfers packets across endpoints by contacting the controller, utilizing its flow table to determine the packet's destination, and sending the packet to the following node.

Packet Class: a class that offers utilitarian features to other modules in my project that need to handle packet decoding and structuring. It enables other classes to transmit packets of a specific type and upon starting, packets are sent using a socket while continuously waiting for them..

7 Reflection

I've learned a lot about the OpenFlow software defined networking standard thanks to this assignment. I've learned more about Python concurrent programming and Python threads because of working on this assignment. Additionally, I gained a lot of knowledge on how to use implement complex topologies. In general, I am pleased with the protocol I used for this task. I learned more about how network components interact and how routers direct network traffic. In addition, I was able to learn more about SD-WAN. After finishing the last assignment, this implementation went much more quickly because I already knew a lot of the fundamental skills, such how to use Docker and transmit UDP packets.

8 References

[1] William Strunk, Jr. and E. B. White. *The Elements of Style*. Pearson Education, 4th edition, 1999.
[2] Docker. Docker project page. https://www.docker.com
[3] OpenFlow Protocol
[4] OpenFlow network architecture by Idris Zoher Bholebawa
[5] https://overlaid.net/2017/02/15/openflow-basic-concepts-and-theory
[6] Docker Walkthrough sample code by Dr. Stefan Weber
[7] https://abdesol.medium.com/udp-protocol-with-a-header-implementation-in-python-b3d8dae9a74b
[8] https://linuxhint.com/send_receive_udp_python/
[9] https://linuxhint.com/send_receive_udp_python/
[10] Trinity College. CSU33031 Lecture Slides and my previous report.
[11] Managed SD-WAN Services A Clear and Concise Reference