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

CSU33031 Computer Networks

Flow Forwarding

Prathamesh Sai
3rd year Integrated Computer Science
Student ID: 19314123

School of Computer Science and Statistics,
Trinity College Dublin

Contents

1 Introduction

This assignment is focused on learning about the decisions to forward flows of packets and the information that is kept at network devices that make those forwarding decisions. The aim for the protocol is to develop a replacement for IPv4 or IPv6 by designing a protocol that forwards payloads based on a collection of strings that identify the source and destination of traffic. The forwarding information will be kept in forwarding tables located at the individual network elements and the content of these tables will be controlled by a central controller. We have some requirements for this assignment:

1. Be able to send messages between two different end-users, with a number of switches/routers in-between, that communicate with a controller before forwarding the flow of packets to the next component(s).
2. Use a UDP service bound to port 51510 at every network, so an application using your protocol will create the header, attach the payload to be transmitted, and send it as payload to the service bound to port 51510 on the local host element.
3. The header information should be encoded in a type-length-value (TLV) format.
4. The forwarding service has to accept incoming packets, inspect the header information, consult the forwarding table and forward the header and payload information to the destination.

The OpenFlow protocol aligned well with the the functionalities that I wanted to include in my protocol, hence my network operates similarly to an OpenFlow network in terms of it's features. I have written my code in Java and it works with a number of Docker containers, meaning my protocol can connect components located at a number of hosts.

2 Theory of Topic

I will now discuss the theory behind the flow forwarding protocol I have developed. This will allow you to understand some background information that will be useful in understanding my implementation of the protocol.

2.1 End-users

End-users will act as endpoints which can communicate with each other. In my network, I have designed it with 2 end-users in mind. This means two machines on two different IP addresses can communicate. For example, end-user 1 on port 49600 of a particular IP address, can interact with a service bound to port 51510 on the same IP address, to forward packets to another end-user 2 on port 53521 on a completely different IP address with the help of it's own service on port 51510. There are other components such as routers between these two that can make this happen.

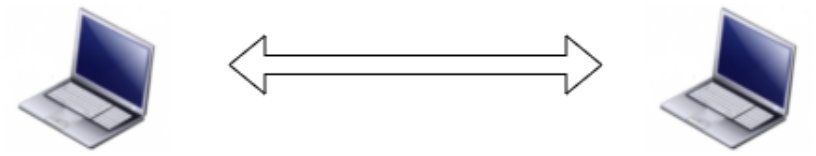


Figure 1: Two end-users with different IP addresses can communicate by sending messages to each other.

2.2 Switch

Routers, or more commonly referred to as switches, are used to accept incoming packets, inspect the header information, consult the forwarding table and forward the header and payload information to the destination. They are vital, as they are the components that allow us to forward the flow of packets between end-users. They communicate with the controller to enquire about forwarding information.

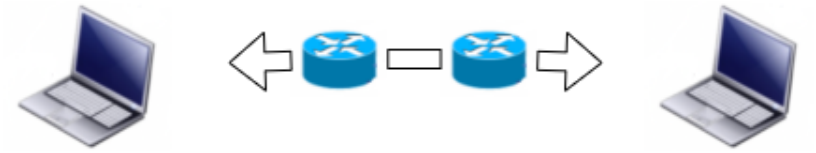


Figure 2: Two end-users with switches in-between them that let them communicate.

2.3 Controller

A controller controls the network elements by initializing the flow tables of the network elements and informs them about routes to destinations as the network changes. The controller is also vital, as it contains the table that the switches use to gather forwarding information to be able to forward the packets, therefore allowing communication between two endpoints.

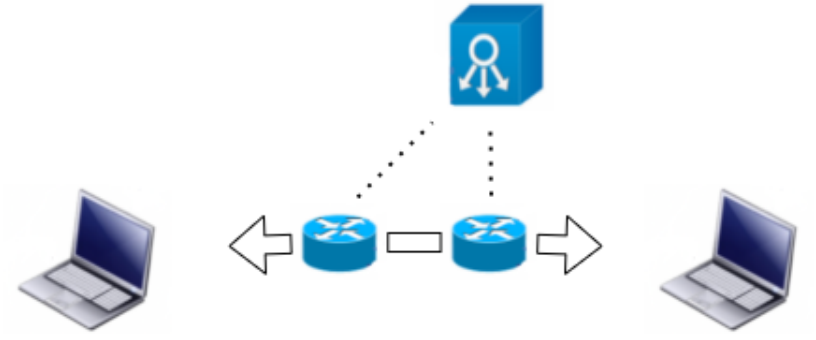


Figure 3: Two end-users with switches and a controller that informs them about routes to destinations.

2.4 Application

For this assignment, a user on a specific IP address can communicate via an application on a specific port. In my network, I have designed it such that the application for end-user 1 is on port 49600, and the application for end-user 2 is on port 53521. The application will be able to send UDP datagrams to a forwarding service on the local host. Visually, this will simply be a terminal opened by a user trying to send information to a service via user input.

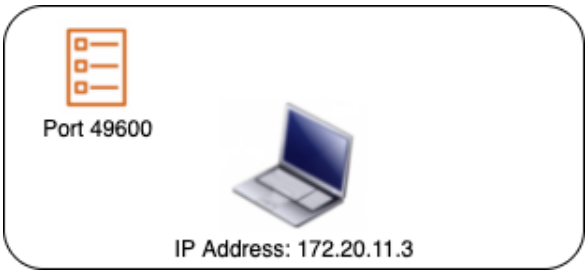


Figure 4: An application on port 49600 in a container for end-user 1.

2.5 Service

A flow forwarding service must be accessible to the application so that it can request to forward packets from end-user 1 to end-user 2 and vice-versa. The forwarding service will consult a table to determine where to forward the datagram to from the header information. For this assignment, we are required to ensure that the service is bound to port 51510.

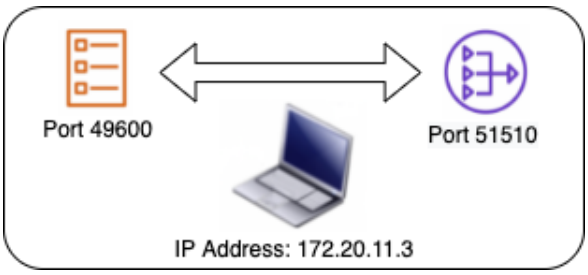


Figure 5: An application on port 49600 communicating with a service on port 51510 to send packets from a container for end-user 1 to end-user 2.

2.6 OpenFlow Protocol

OpenFlow[1] is a communications protocol that gives access to the forwarding plane of a network switch or router over the network. OpenFlow enables network controllers to determine the path of network packets across a network of switches. My network incorporates many features from an OpenFlow network.

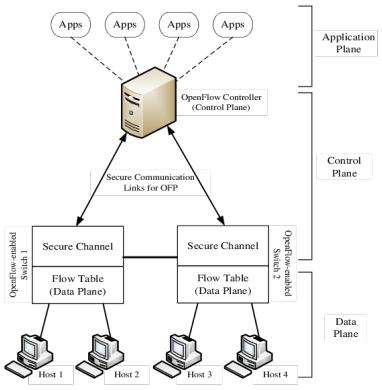


Figure 6: An OpenFlow network architecture[2]

3 Implementation

I will now discuss how my flow forwarding implementation works with the theory stated above. I have written my implementation in Java. I used OpenJDK-8 as the Terminal functionality in tcdlib.jar on blackboard was not compatible with OpenJDK-16. I noticed that the service and switches can use the same port 51510, meaning I did not need to make a separate service class; The switches could be adapted to also work as a service. I ended up making two other switches to forward information between the end-users, and a controller for the switches to communicate with.

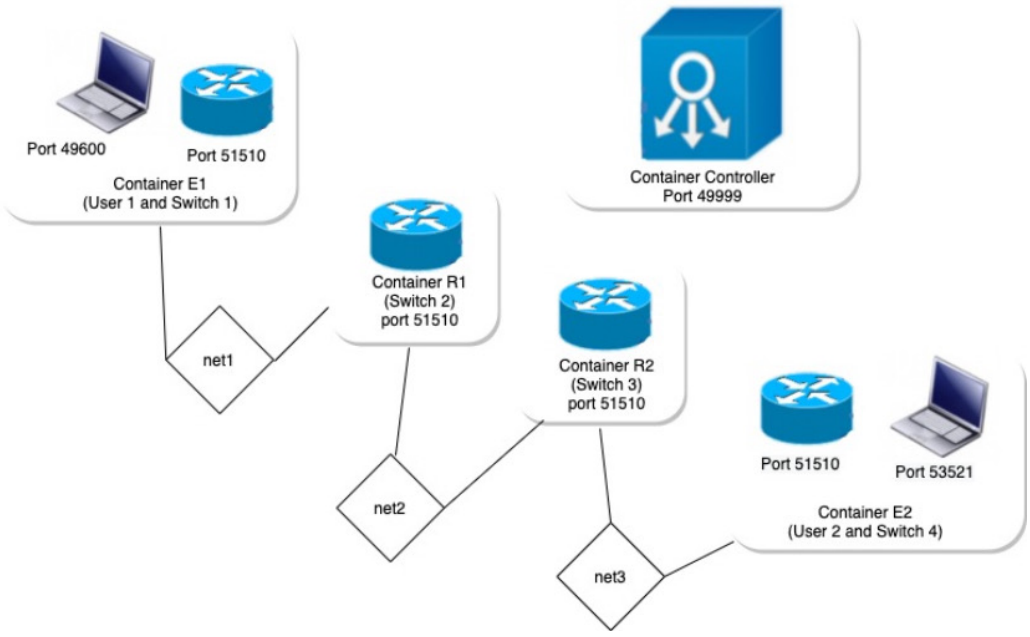


Figure 7: A diagram illustrating the implementation of my project; net1 has an address of 172.20.11.0/24, net 2 has an address of 172.20.33.0/24 and net 3 has an address of 172.20.66.0/24.

3.1 Node.java

The Node class is an abstract class that is extended by the controller, end-users, and switches. We were provided a generic Node.java file on Blackboard for the Docker Walk-through[3], which I used for my implementation. This class contains a nested Listener class which constantly listens for any incoming packets, calling the onReceipt function on arrival.

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;
import java.util.concurrent.CountDownLatch;

public abstract class Node {
  static final int PACKETSIZE = 65536;
  DatagramSocket socket;
  Listener listener;
  CountDownLatch latch;

  Node() {
    latch = new CountDownLatch(1);
    listener = new Listener();
    listener.setDaemon(true);
    listener.start();
  }

  protected void sendPacket(PacketUtility packetUtility) {
    try {
      socket.send(packetUtility.getPacket());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /*
   * This function is Abstract so it can be adjusted to different classes
   such as Switches,
   * End-users or a Controller.Hence, different classes can behave
   differently when they receive packets.
   */
  public abstract void onReceipt(DatagramPacket packet);
  /**
   * Listens for incoming packets on a datagram socket and informs
   registered
   * receivers about incoming packets.
   */
  class Listener extends Thread {
    //Stating to the listener that the socket has been initialized.
    public void go() {
      latch.countDown();
    }

    //Listen for incoming packets and inform receivers.
```

```
    public void run() {
      try {
        latch.await();
        // Infinite loop that attempts to receive packets, and notify
        receivers.
        while (true) {
          DatagramPacket packet = new DatagramPacket(new byte[PACKETSIZE],
          PACKETSIZE);
          socket.receive(packet);
          onReceipt(packet);
        }
      } catch (Exception e) {
        if (!(e instanceof SocketException))
          e.printStackTrace();
      }
    }
  }
}
```

Listing 1: This onReceipt function is abstract, so any class that extends the Node class can deal with the incoming packets in a unique way.

3.2 INFO.java

The INFO class contains important constant values that are used throughout the network. This reduces repeated arbitrary values and improves the overall readability of my code.

```
public class INFO {
  //Types of OpenFlow packets.
  static final int HELLO = 1;
  static final int PACKET_IN = 2;
  static final int PACKET_OUT = 3;
  static final int FLOW_MOD = 4;

  //End-users in the network and it's associated number.
  static final String ENDUSER1 = "1";
  static final String ENDUSER2 = "2";

  //Host names involved in the network.
  static final String CONTROLLER_HOST = "Controller";
  static final String LOCALHOST = "localhost";

  //Ports of components on the network
  static final int PORT_OF_SWITCH = 51510;
  static final int PORT_NUMBER_OF_CONTROLLER = 49999;
  static final int PORT_NUMBER_OF_USER_1 = 49600;
  static final int PORT_NUMBER_OF_USER_2 = 53521;

  //Addresses of involved in the network.
  static final String LOCALHOST_IP = "127.0.0.1";

  //Network 1 (net1):
  static final String SWITCH_1_ADDRESS = "172.20.11.3";
  static final String USER_1_ADDRESS = "172.20.11.3";
```

```
  static final String SWITCH_2_ADDRESS_NET1 = "172.20.11.4";

  //Network 2 (net2):
  static final String SWITCH_2_ADDRESS_NET2 = "172.20.33.3";
  static final String SWITCH_3_ADDRESS_NET2 = "172.20.33.4";

  //Network 3 (net3):
  static final String SWITCH_3_ADDRESS_NET3 = "172.20.66.4";
  static final String SWITCH_4_ADDRESS = "172.20.66.3";
  static final String USER_2_ADDRESS = "172.20.66.3";
}
```

Listing 2: Important constant values such as hosts, ports, and IP addresses used throughout my code.

3.3 EndUser1.java

The end-user 1 class is ran in a terminal (acting as the application on port 49600) to allow user 1 to interact with the service on port 51510 with the user input.

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;

import tcdIO.Terminal;

public class EndUser1 extends Node {
  private Terminal terminal;

  EndUser1(Terminal terminal) {
    this.terminal = terminal;
    try {
      socket = new DatagramSocket(INFO.PORT_NUMBER_OF_USER_1);
      listener.go();
    } catch (java.lang.Exception e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    try {
      Terminal terminal = new Terminal("End-User-1");
      (new EndUser1(terminal)).start();
    } catch (java.lang.Exception e) {
      e.printStackTrace();
    }
  }

  public synchronized void start() throws Exception {
    while (true) {
      String command = terminal.readString("As end-user 1, you can either
      :\n " + "1. Send a message to end-user 2.\n" + "2. Wait for messages
      from end-user 2.\n");
      if (command.contains("1")) {
        sendMessageToEndUser2();
```

```
      }
      else if (command.contains("2")) {
        terminal.println("Waiting for a message from end-user 2...");
        this.wait();
      }
    }
  }

  public synchronized void onReceipt(DatagramPacket incomingPacket) {
    this.notify();
    byte[] incomingPacketAsByteArray = incomingPacket.getData();
    if(incomingPacketAsByteArray[0] == INFO.PACKET_OUT) {
      //Convert the incoming packet to a string, ignoring the first 3
      bytes which is the header information.
      byte[] messageInBytes = new byte[incomingPacketAsByteArray.length];
      for(int i = 0; i+3 < incomingPacketAsByteArray.length; i++) {
        messageInBytes[i] = incomingPacketAsByteArray[i+3];
      }
      //Trim the message to get rid of any leading or trailing blank
      spaces.
      String message = new String(messageInBytes).trim();
      terminal.println("I just got a message from end-user 2: " + message
      + "\n");
    }
  }

  public void sendMessageToEndUser2() {
    String messageToSend = terminal.readString("What message would you
    like to send?: \n");
    PacketUtility messageToSendAsPacket = new PacketUtility();
    messageToSendAsPacket.packetOut(INFO.LOCALHOST, INFO.PORT_OF_SWITCH,
    2, messageToSend);
    sendPacket(messageToSendAsPacket);
  }
}
```

Listing 3: Depending on the user input, the application on port 49600 will either make a request to the flow forwarding service on port 51510 to send a message to end-user 2, or wait for a message from end-user 2.

3.4 EndUser2.java

The end-user 2 is almost identical to the end-user 1 class. It allows user 2 to interact with the running class in a terminal (acting as the application on port 53521) and interact with the service on port 51510 with the user input.

```
  EndUser2(Terminal terminal) {
    this.terminal = terminal;
    try {
      socket = new DatagramSocket(INFO.PORT_NUMBER_OF_USER_2);
      listener.go();
    } catch (java.lang.Exception e) {
      e.printStackTrace();
```

```
    }
  }

  public static void main(String[] args) {
    try {
      Terminal terminal = new Terminal("End-User-2");
      (new EndUser2(terminal)).start();
    } catch (java.lang.Exception e) {
      e.printStackTrace();
    }
  }
```

Listing 4: Apart from minor differences with input/output, the code above is the only difference between EndUser1.java and EndUser2.java. Similarly to EndUser1.java, the application on port 53521 will either make a request to the flow forwarding service on port 51510 to send a message to end-user 1, or wait for a message from end-user 1.

3.5 Switch.java

The switch class gets its switch number as input from the user at the beginning to set up the initialization of the switches in the network. After it is initialized, the switch sends a 'hello' packet to the controller. The controller sends back a 'hello' packet. When a switch gets a 'flowMod' packet, it will modify the flow table from the incoming packets and will forward the packets in the switch's buffer.

```
public synchronized void onReceipt(DatagramPacket incomingPacket) {
    byte[] incomingPacketAsByteArray = incomingPacket.getData();
    //The hello message we sent has been acknowledged.
    if(incomingPacketAsByteArray[0] == INFO.HELLO) {
      terminal.println("I just got a 'hello' packet.");
    }
    //If we must send a packet out.
    else if (incomingPacketAsByteArray[0] == INFO.PACKET_OUT) {
      terminal.println("I just got a 'packet_out' packet.");
      if(switchBuffer == null) {
        switchBuffer = incomingPacket;
      }
      //Create a 'packet_in' packet and send it to the controller.
      PacketUtility packetIn = new PacketUtility();
      packetIn.packetIn(INFO.CONTROLLER_HOST,
      incomingPacketAsByteArray[1]);
      sendPacket(packetIn);
      terminal.println("I just sent a 'packet_in' packet to the
      Controller");
    }
    else if (incomingPacketAsByteArray[0] == INFO.FLOW_MOD) {
      terminal.println("I just got a 'flowMod' packet from the
      Controller");
      flowTableModification(incomingPacketAsByteArray);
      try {
        forwardPacketInSwitchBuffer();
```

```
      } catch (UnknownHostException e) {
        e.printStackTrace();
      }
    }
    else {
      terminal.println("An unknown packet has been sent.");
    }
  }
```

Listing 5: The onReceipt function in the Switch class checks the type of packet being sent, and responds appropriately by either printing out the fact that it got a 'hello' packet, or modifying the flow table and forwarding the packets in the buffer.

When the flowTableModification method is called, the header information is removed from the array of bytes, and we remove any unused space. We convert this back to a string using the UTF-8 character set and insert it into the flow table in the switch class using modulo arithmetic inside of a for-loop.

```
  public void flowTableModification(byte[] packetData) {
    byte[] incomingPacketWithoutHeader = new byte[packetData.length-3];
    for(int i = 0; i < incomingPacketWithoutHeader.length; i++) {
      incomingPacketWithoutHeader[i] = packetData[i+3];
    }
    String messageFromIncomingPacket = new
    String(incomingPacketWithoutHeader).replaceAll("\0", "");
    byte[] messageFromPacket = messageFromIncomingPacket.getBytes();
    String tableAsString = "";
    try {
      tableAsString = new String(messageFromPacket, "UTF-8");
    } catch (UnsupportedEncodingException e) {
      e.printStackTrace();
    }
    String[] tableRowsAsStringArray = tableAsString.split(",");
    flowTable = new String[tableRowsAsStringArray.length/3][3];
    int currentRow = 0;
    for(int i = 0; i < tableRowsAsStringArray.length; i++) {
      if(i != 0) {
        if(i%3 == 0) {
          currentRow++;
        }
      }
      if(flowTable == null || flowTable.length == 0) {
        break;
      }
      flowTable[currentRow][i%3] = tableRowsAsStringArray[i];
    }

  }
```

Listing 6: This function allows the switch to modify the local flow table before forwarding the packet.

When a switch is sent a 'packet_out' packet, it then caches this packet to the Datagram-Packet called switchBuffer and will then send a 'packet_in' packet to the controller. This lets the controller choose where to forward the packet. After this, the controller sends a 'flowMod' packet meaning the switch modifies the flow table and forwards the packet using this updated flow table.

3.6 PacketUtility.java

The PacketUtility class provides utility functions such as adding header information to an array of bytes, adding the message to an array of bytes, or providing OpenFlow functionality such as in the functions packetIn, packetOut, and helloPacket and flowMod.

```
//Create 'flowMod' packet by adding appropriate header and message
information after getting neccessary data.
public void flowMod(String address, int portNumber, String
controllerFlowTable[][]) {
  //Now we must process the flow table from the controller before it is
  sent to the switch.
  String data = "";
  for(int i = 0; i < controllerFlowTable.length; i++) {
    if(controllerFlowTable[i][2].equals(address)) {
      data = data + controllerFlowTable[i][0] + "," +
      controllerFlowTable[i][3] + "," + controllerFlowTable[i][4] + ",";
    }
  }
  byte[] packetAsByteArray = new byte[data.length() + 3];
  includeHeaderInformation(packetAsByteArray, INFO.FLOW_MOD,
  INFO.PORT_NUMBER_OF_CONTROLLER, packetAsByteArray.length);
  includeMessageInformation(packetAsByteArray, data);
  InetSocketAddress dstAddress = new InetSocketAddress(address,
  portNumber);
  DatagramPacket packet = new DatagramPacket(packetAsByteArray,
  packetAsByteArray.length, dstAddress);
  this.packet = packet;
}

//Create 'packet_in' packet by adding appropriate header information.
public void packetIn(String address, int finalDestination) {
  byte[] headerInformation = new byte[3];
  includeHeaderInformation(headerInformation, INFO.PACKET_IN,
  finalDestination, headerInformation.length);
  InetSocketAddress dstAddress = new InetSocketAddress(address,
  INFO.PORT_NUMBER_OF_CONTROLLER);
  DatagramPacket packet = new DatagramPacket(headerInformation,
  headerInformation.length, dstAddress);
  this.packet = packet;
}

//Create 'hello' packet by adding appropriate header information.
public void helloPacket(String address, int finalDestination) {
  byte[] headerInformation = new byte[3];
  includeHeaderInformation(headerInformation, INFO.HELLO,
  finalDestination, headerInformation.length);
```

```
    InetSocketAddress dstAddress = new InetSocketAddress(address,
    INFO.PORT_NUMBER_OF_CONTROLLER);
    DatagramPacket finalPacket = new DatagramPacket(headerInformation,
    headerInformation.length, dstAddress);
    this.packet = finalPacket;
  }

  //Create 'packet_out' packet by adding appropriate header information.
  public void packetOut(String address, int port, int finalDestination,
  String message) {
    byte[] packetAsByteArray = new byte[message.length()+3];
    includeHeaderInformation(packetAsByteArray, INFO.PACKET_OUT,
    finalDestination, packetAsByteArray.length);
    includeMessageInformation(packetAsByteArray, message);
    InetSocketAddress dstAddress = new InetSocketAddress(address, port);
    DatagramPacket packet = new DatagramPacket(packetAsByteArray,
    packetAsByteArray.length, dstAddress);
    this.packet = packet;
  }
```

Listing 7: The flowMod function converts the rows of the flow table that are relevant to the switch to a string, and converts it to a string that is sent to the switch. The packetIn function is used for sending a packet to a particular address, and the packetOut function is used when a node is sending a packet to the next component in the network and the packet is stored in the local DatagramPacket variable.

In the assignment specification, we were told to encode the header of the packet in a type-length-value (TLV) format. I decided to incorporate this type of format, but also add my own unique data to the header by also including the final destination of the packet. I did this so that the switches in the network could differentiate the flow of data by comparing this value in the header with the final destination in the corresponding value in the flow table. This ended up being really helpful as it resulted in a smooth flow of packets between the switches and minimizing extra logic to determine the flow of data. I believed that the extra 1 byte of header information would be worth it to minimize on extra unnecessary logic that might over-complicate my project and increase complexity.

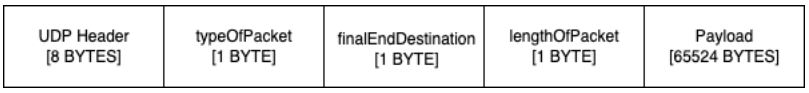


Figure 8: The design of my packet including the UDP header in all Datagram Packets containing the source port/address and destination port/address.

3.7 E1.java

Since end-user 1 (i.e. the application) and switch 1 (i.e. it's service) are inside of the same container E1, I wanted to run them simultaneously in the same container. To do this, I had to make two threads in the E1 class, that runs an instantiation of EndUser1.java, and Switch.java.

```
public class E1 {
  public static void main(String[] args) {
    try {
      //Create two threads to setup the Docker container E1, containing
      User 1 and it's local switch (i.e. service).
      Thread one = new Thread(() -> Switch.main(args));
      Thread two = new Thread(() -> EndUser1.main(args));
      one.start();
      two.start();
      one.join();
      two.join();
    } catch (java.lang.Exception e) {
      e.printStackTrace();
    }
  }
}
```

Listing 8: The code above makes two threads, end-user 1 and switch 1, to be ran inside of the E1 container on Docker.

3.8 E2.java

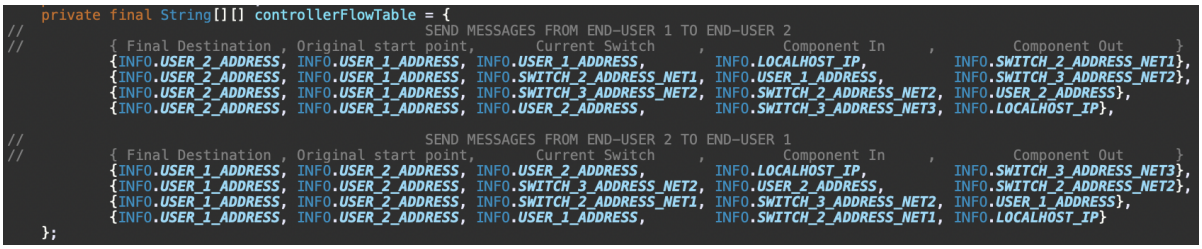
The E2 class is very similar to the E1 class, except this time since end-user 2 (i.e. the application) and switch 4 (i.e. it's service) are inside of the same container E2, I wanted to run them simultaneously in the same container too. To do this, I had to make two threads in the E2 class, that runs an instance of EndUser2.java, and Switch.java.

```
public class E2 {
  public static void main(String[] args) {
    try {
      //Create two threads to setup the Docker container E2, containing
      User 2 and it's local switch (i.e. service).
      Thread one = new Thread(() -> Switch.main(args));
      Thread two = new Thread(() -> EndUser2.main(args));
      one.start();
      two.start();
      one.join();
      two.join();
    } catch (java.lang.Exception e) {
      e.printStackTrace();
    }
  }
}
```

Listing 9: The code above makes two threads, end-user 2 and switch 4, to be ran inside of the E2 container on Docker.

3.9 Controller.java

The controller class contains the main flow table for this system, which is stored in the form of a 2D array of strings which represent the IP addresses of different components.



```
private final String[][] controllerFlowTable = {
//                                    SEND MESSAGES FROM END-USER 1 TO END-USER 2
//        { Final Destination , Original start point,      Current Switch     ,        Component In     ,        Component Out     }
        {INFO.USER_2_ADDRESS, INFO.USER_1_ADDRESS, INFO.USER_1_ADDRESS,      INFO.LOCALHOST_IP,        INFO.SWITCH_2_ADDRESS_NET1},
        {INFO.USER_2_ADDRESS, INFO.USER_1_ADDRESS, INFO.SWITCH_2_ADDRESS_NET1, INFO.USER_1_ADDRESS,        INFO.SWITCH_3_ADDRESS_NET2},
        {INFO.USER_2_ADDRESS, INFO.USER_1_ADDRESS, INFO.SWITCH_3_ADDRESS_NET2, INFO.SWITCH_2_ADDRESS_NET2, INFO.USER_2_ADDRESS},
        {INFO.USER_2_ADDRESS, INFO.USER_1_ADDRESS, INFO.USER_2_ADDRESS,        INFO.SWITCH_3_ADDRESS_NET3, INFO.LOCALHOST_IP},

//                                    SEND MESSAGES FROM END-USER 2 TO END-USER 1
//        { Final Destination , Original start point,      Current Switch     ,        Component In     ,        Component Out     }
        {INFO.USER_1_ADDRESS, INFO.USER_2_ADDRESS, INFO.USER_2_ADDRESS,      INFO.LOCALHOST_IP,        INFO.SWITCH_3_ADDRESS_NET3},
        {INFO.USER_1_ADDRESS, INFO.USER_2_ADDRESS, INFO.SWITCH_3_ADDRESS_NET2, INFO.USER_2_ADDRESS,        INFO.SWITCH_2_ADDRESS_NET2},
        {INFO.USER_1_ADDRESS, INFO.USER_2_ADDRESS, INFO.SWITCH_2_ADDRESS_NET1, INFO.SWITCH_3_ADDRESS_NET2, INFO.USER_1_ADDRESS},
        {INFO.USER_1_ADDRESS, INFO.USER_2_ADDRESS, INFO.USER_1_ADDRESS,        INFO.SWITCH_2_ADDRESS_NET1, INFO.LOCALHOST_IP}
};
```

Figure 9: The hard-coded flow table that allows other nodes in the system to forward information to the correct destination.

```
//If the controller gets a "hello" packet from any of the switches, then
it must respond back with a "hello" packet.
  //If the controller gets a "packet_in" packet, then it must respond back
  by updating the flow table of thw switch.
  public synchronized void onReceipt(DatagramPacket incomingPacket) {
    byte[] incomingPacketAsByteArray = incomingPacket.getData();
    //Get the switch number that this packet is coming from.
    String switchNumber = "";
    if(incomingPacket.getAddress().getHostAddress()
       .equals(INFO.SWITCH_1_ADDRESS)) {
      switchNumber = "1";
    }
    else if(incomingPacket.getAddress().getHostAddress()
            .equals(INFO.SWITCH_2_ADDRESS_NET1) ||
            incomingPacket.getAddress().getHostAddress()
            .equals(INFO.SWITCH_2_ADDRESS_NET2)) {
      switchNumber = "2";
    } else if(incomingPacket.getAddress().getHostAddress()
              .equals(INFO.SWITCH_3_ADDRESS_NET2) ||
              incomingPacket.getAddress().getHostAddress()
              .equals(INFO.SWITCH_3_ADDRESS_NET3)) {
      switchNumber = "3";
    } else if(incomingPacket.getAddress().getHostAddress()
              .equals(INFO.SWITCH_4_ADDRESS)) {
      switchNumber = "4";
    }

    if(incomingPacketAsByteArray[0] == INFO.HELLO) {
      terminal.println("I just got a 'hello' packet from switch number: "
      + switchNumber);
      //Send back a 'hello' packet!
      PacketUtility respondBackWithHello = new PacketUtility();
      respondBackWithHello.helloPacket(incomingPacket.getAddress()
      .getHostAddress(), incomingPacketAsByteArray[1]);
      sendPacket(respondBackWithHello);
    }
    else if (incomingPacketAsByteArray[0] == INFO.PACKET_IN) {
      terminal.println("I just got a 'packet_in' packet from switch
      number: " + switchNumber);
```

```
        //Update the table and send it.
        PacketUtility updatedTable = new PacketUtility();
        updatedTable.flowMod(incomingPacket.getAddress().getHostAddress(),
        incomingPacket.getPort(),  this.controllerFlowTable);
        sendPacket(updatedTable);
        terminal.println("I just sent a 'flowMod' packet to switch number: "
        + switchNumber);
    }
}
```

Listing 10: The onReceipt function figures out which type of packet was sent to the controller, and acts corresponding to the type of packet that was sent.

In the onReceipt method, if there is an incoming 'hello' packet, then we send back a 'hello' packet as an acknowledgement. If there is an incoming 'packet_in" packet, it responds back by updating the flow table of the switch it came from. This type of packet occurs when a switch is unsure where to send a packet so it is asking the controller for the best place to send it or it doesn't have a entry in the flow table to forward this packet to. To respond to this, the controller sends back a 'flowMod' packet. Due to time constraints, I was unable to implement link state routing, hence I hard-coded the values in the flow table and it is immutable, hence it is a 'final' variable in Java.

4 Discussion

4.1 Docker

My protocol can connect components located at a number of hosts. This can be demonstrated with Docker, where I can run the E1, R1, E2, R2, and Controller containers, with each of them having their own IP address and still be able to send messages from end-user 1 to end-user 2 and from end-user 2 to end-user 1. I had to use Ubuntu containers instead of Java containers as the java image for Docker containers is only available for Intel processors, and I was fortunate that I knew this after the first assignment. I originally started off my implementation with the components simply having different ports, and running them locally. It was definitely a big jump to get my implementation working on Docker, as it meant that I had to do a lot to set up the code correctly, such as setting socket addresses instead of simply setting the ports, using threads to set up containers E1 and E2, modifying the hard-coded flow table to support addresses instead of port numbers and much more.

```
//Network 1 (net1):
static final String SWITCH_1_ADDRESS = "172.20.11.3";
static final String USER_1_ADDRESS = "172.20.11.3";
static final String SWITCH_2_ADDRESS_NET1 = "172.20.11.4";

//Network 2 (net2):
static final String SWITCH_2_ADDRESS_NET2 = "172.20.33.3";
static final String SWITCH_3_ADDRESS_NET2 = "172.20.33.4";

//Network 3 (net3):
static final String SWITCH_3_ADDRESS_NET3 = "172.20.66.4";
static final String SWITCH_4_ADDRESS = "172.20.66.3";
static final String USER_2_ADDRESS = "172.20.66.3";
```

Figure 10: The range of addresses the different components had in different networks.

4.2 Communication between Nodes

The switches, end-users, and controller have to interact in a specific order for the communication between the end-users to be smooth. For example, all switches need to send a 'hello' packet to the controller before an end-user can request to send information to another end-user with a service.

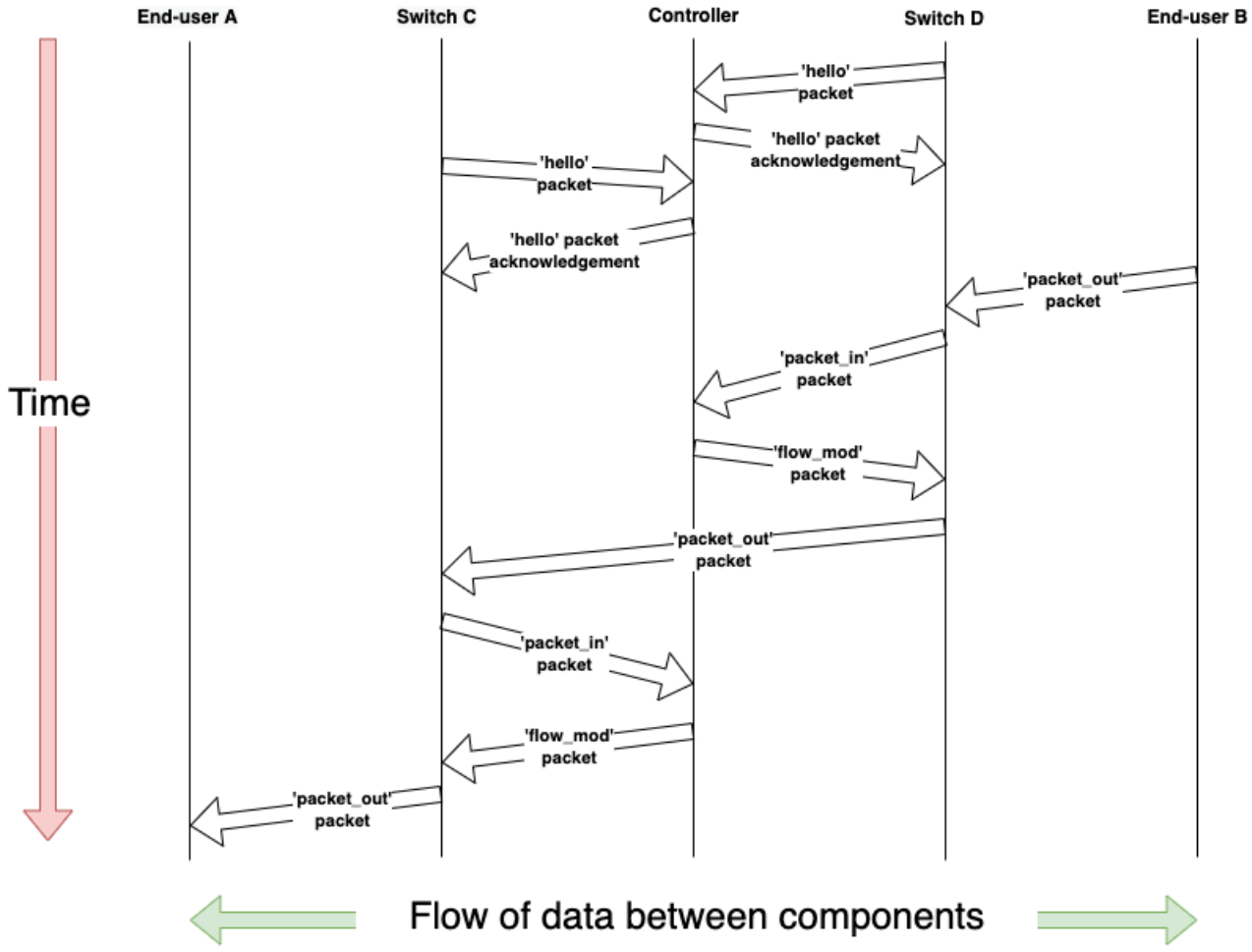


Figure 11: An example of how 2 switches, and 2 end-end users would communicate with the controller with my protocol.

4.3 Example of implementation

I will now show you an example of my code working with Docker containers, and being able to send messages from end-user 1 inside of container E1, to end-user 2 inside of container E2, and vice-versa.

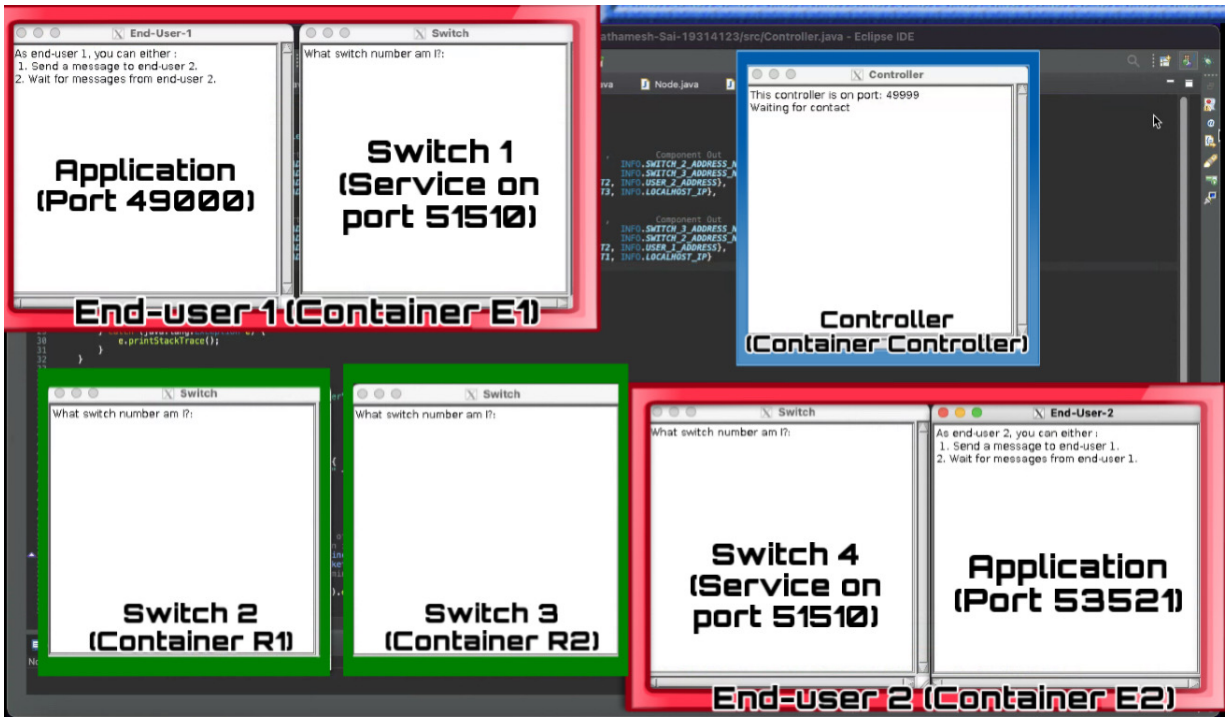


Figure 12: The layout of my flow forwarding network.

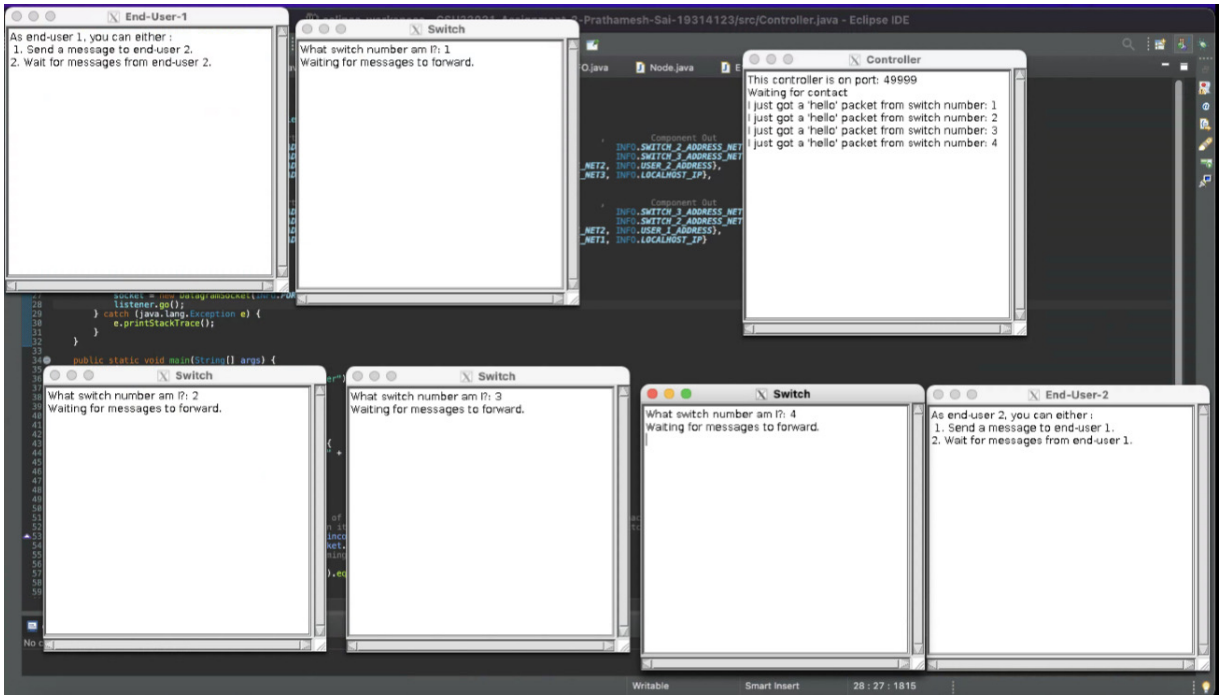


Figure 13: Connecting all the switches to the controller by sending a 'hello' packet.

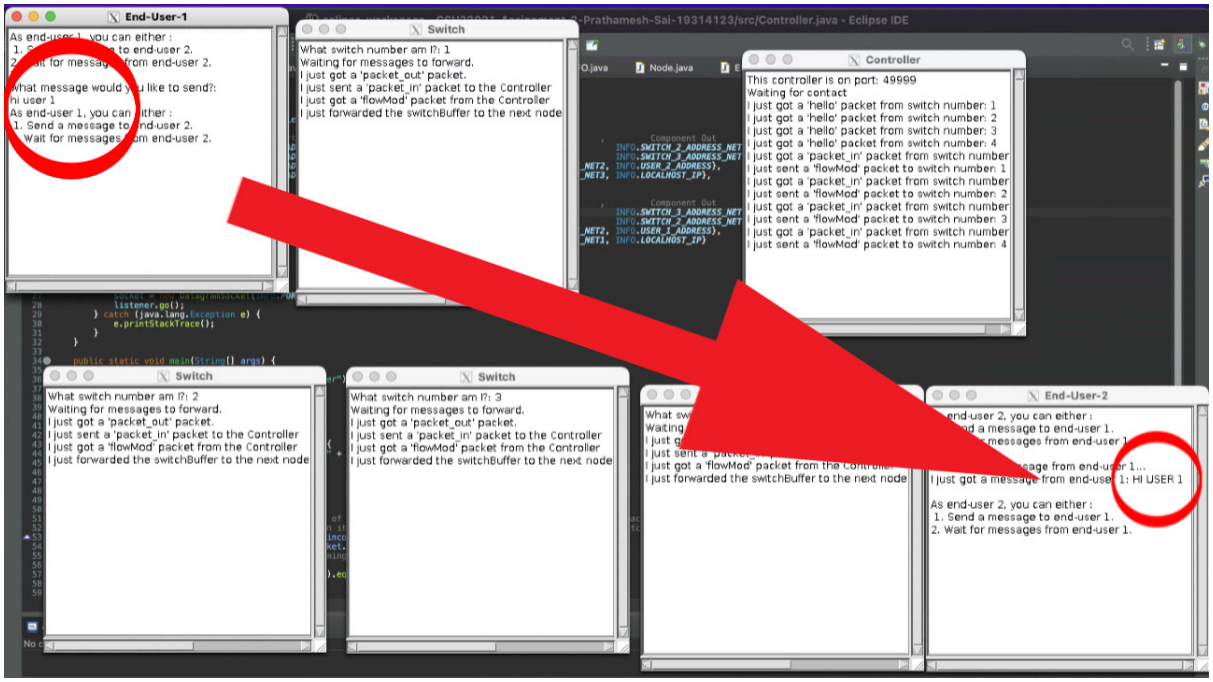


Figure 14: Sending a message from end-user 1 to end-user 2.

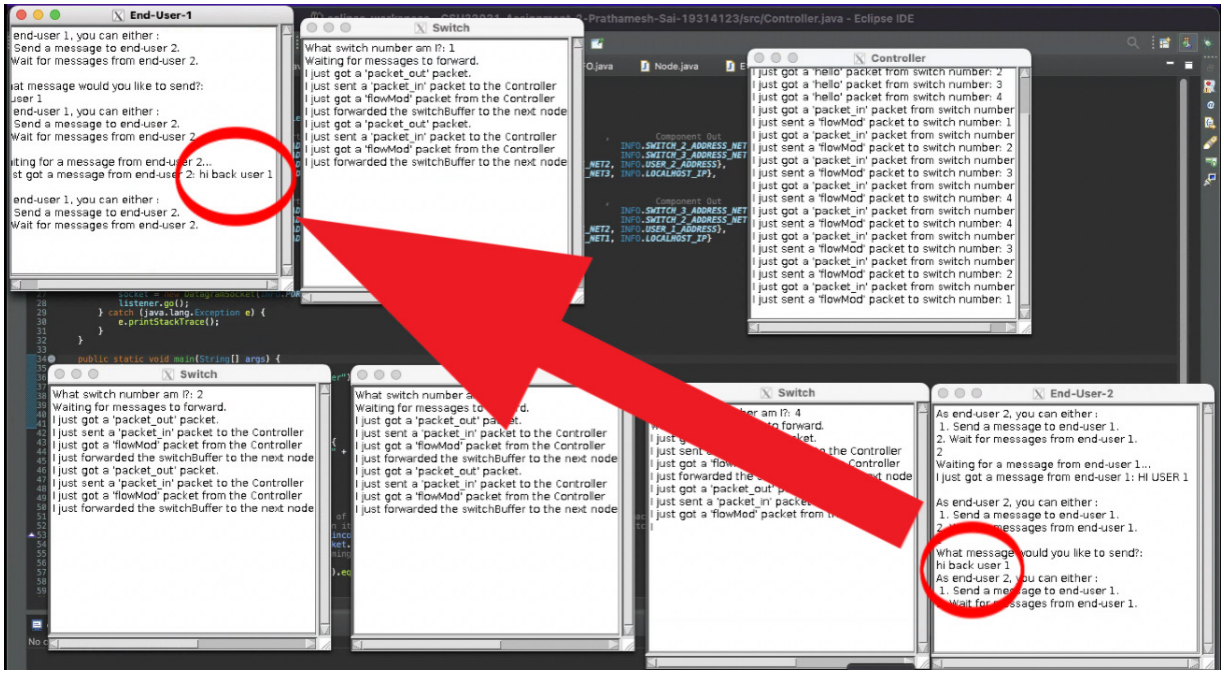


Figure 15: Sending a message from end-user 2 to end-user 1.

5 Summary

In this report I have described my implementation of a protocol that forwards payloads based on a collection of strings that identify the source and destination of traffic. My implementation used 9 classes, which were an abstract Node class, PacketUtility class, Switch class, EndUser1 class, EndUser2 class, Controller class, INFO class, E1 class, and an E2 class. The switches, end-users, and controllers are all extensions of the Node class, with unique functionality in the onReceipt function to react differently to incoming packets.

Node class: an abstract class continuously waiting for packets with the initialization of a listener thread and sends packets using socket.send().

INFO class: a class that contains important final values such as the IP addresses of Docker containers, the port numbers of components in my solution and more. This reduces arbitrary values in other code which increases readability.

EndUser1 class: a class that acts as an endpoint to transfer information from, and to transfer information to. This is on port 49600, and interacts with it's service on port 51510 which is switch 1 to send packets to end-user 2.

EndUser2 class: a class that acts as an another endpoint to transfer information from, and to transfer information to. This is on port 53521, and interacts with it's service on port 51510 which is switch 4 to send packets to end-user 1.

Switch class: a class that forwards the flow of packets between endpoints by contacting the controller and then using it's flow table to find out the destination of the packet, and sending it to the next node.

PacketUtility class: A utility class providing utility functions for other classes in my project that wants to perform packet encoding and formatting. It allows other classes to send packets with of a particular 'type'.

E1 class: A helper class that is ran inside of the E1 container, that makes two threads. One thread is for EndUser1.java, and another for Switch.java that acts as it's service.

E2 class: A helper class that is ran inside of the E2 container, that makes two threads. One thread is for EndUser2.java, and another for Switch.java that acts as it's service.

Controller class: A class that is connected to all networks and contains a hard-coded flow table for the switches to use to be able to know where to forward packets to.

6 Reflection

As I reflect on the process of implementing this protocol, I enjoyed working on this assignment. I learnt how to forward packets between end-users, how to format flow tables, and also how to process them and the complexity that is associated with more complicated routing algorithms. Due to time constraints, I was not able to implement link state routing or any distance based routing algorithms which I would have loved to do. However I am still very happy with my project overall as it was a huge challenge for me who had never even used Docker before this semester. I learnt a lot during this module, and I was even able to use my experiences from the first assignment to help me in this one, even with small things like not knowing how to use 'docker cp' to copy files into containers. I have no doubt that the learning experiences I've gathered during this assignment and module overall will help me throughout my career.

7 References

[1]: OpenFlow Protocol (accessed December, 2021).
[2]: OpenFlow network architecture by Idris Zoher Bholebawa (accessed December, 2021).
[3]: Docker Walkthrough sample code by Dr. Stefan Weber (accessed December, 2021).