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

CSU33031 Computer Networks I
Assignment #2: Flow Forwarding

Thomas Moroney, 20332993

December 4, 2022

Contents

1 Introduction

The problem description for this assignment is to develop an overlay that links implementations of forwarding mechanisms in ADSL routers to network elements of the provider to forwarding services at a cloud provider. It defined that the solution should have a central controller that controls the flow of packets between network elements. A header should be attached to each packet that each forwarding service will examine and decide where to forward the packet to, based on the header information and the forwarding table. A system like this would be used to allow employees working from home to access services only available on the work servers.

In my solution, I have multiple clients (users) on different networks that are each connected a forwarding service on their local network. When a client sends a packet, a header is attached to the packet containing the destination address and return address. The client will send each packet to it's local forwarding service (GW) which will then ask the controller where to forward the packet next. The Gateway (GW) will forward the packet to the ISP, which will then forward it to the Cloud Provider (CP). The Cloud Provider will forward the incoming data to the requested service.

In the following, I will first discuss the mechanism that was fundamental to my solution. This will be followed by an explanation of the overall design of the network elements of my solution and a description of the data being exchanged between the network elements.

2 Theory of Topic

This section will cover the concepts and protocols that I used to come up with my solution. For my solution I followed the topology diagram featured in the Assignment Description document (as seen below). I have two clients on different networks, each connected to a forwarding service on their local network. Each client's local forwarding service (Gateway) is connected to the same network as the ISP, which shares a network with the Cloud Provider (CP). The Cloud Provider shares a network with all of the destination servers (DServer) which means that it can direct incoming packets to the server containing the service requested by the client. Despite only having two clients, the network can be easily expanded by adding the paths to the controller. In my example, each DServer is a bank account that the user is trying to access.

2.1 Design of Solution

Firstly, I used docker containers to store each of the actors. I created five docker networks to connect all of the network elements. Of these networks, two of them are for connecting each client to it's gateway (local forwarding service). A client will send a packet to it's gateway over their shared network. When the gateway receives the packet from the client, it will ask the controller where it needs to forward the packet in order for it to get the data to the desired destination. I have a fourth network connecting all the gateways (GW) to the Internet Service Provider (ISP), and a fifth network connecting the ISP to the Cloud Provider (CP). The sixth network connects the Cloud Provider to all of the destination servers (DServer).

I decided to make three different types of packets for use in my design. The main packet used is the text packet (TEXTPACKET), which simply contains a text string. The clients, forwarders, and servers, all only send text packets between each other. When forwarding service receives a Text packet, it will extract the destination from the header of the packet. If that destination is in the forwarding table, then the packet will be sent to the next node for that destination in the forwarding table. If the destination is not in the forwarding table, then the packet will be sent to the Controller. The controller has all of the different paths to all the network nodes, so when the controller receives a packet check the path to that destination for the next node to send the packet to. It will then return the port of the next node to the forwarding service in the form of a Jump packet (NEXTNODE). When the Jump packet is received, the forwarding service will update it's forwarding table with the next node for that destination, then it will send back an ACK packet to let the controller know that the Jump packet was received and that the forwarding table was updated. The the forwarding service will match the port number in the Jump packet to the name of the node (using a hashmap) so that a Socket address can be created for the packet to be sent to. If the controller does not receive an ACK packet within 5 seconds of sending a Jump packet, then it will send the Jump packet again.

When a DServer receives a Text packet it will extract the return address from the header and make a new packet with that as the destination address. This new packet will contain the information requested by the client and will be sent back to the Cloud Provider, which will forward it in the right direction to get to it's destination (back to the client).

2.2 Header Design

Before each packet is sent off from the client, a custom header is attached to the packet. A three digit destination address and a three digit return address, both in binary (eg. 001 = DServer2), are attached to the start of the string inside of the every Text packet that is sent by the client. The destination address is necessary in order for the forwarders to know where to send it, and the return address is there so that the DServer knows where to send the requested data back to.

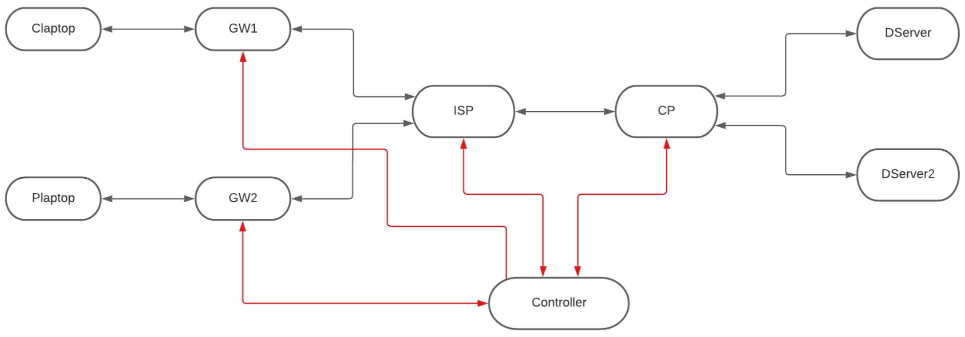


Figure 1: The figure above is topology of my network. The network is using UDP and the Simplest Flow Control protocol. The client sends a request packet to the Ingress, which then forwards this packet on to a worker. The chosen worker returns the requested file to the Ingress, which forwards this packet on to the Client. First the worker sends an info packet, and then a content packet.

3 Implementation

In my implementation, each DServer is a bank account that the Clients want to access. The user can choose which account they want to access, and they can choose to make a deposit or just check the current balance in the account.

3.1 Client Implementation

```
public synchronized void onReceipt(DatagramPacket packet) {
   try {
      PacketContent content = PacketContent.fromDatagramPacket(packet);

       if (content.getType() == PacketContent.TEXTPACKET) {
         TextPacket inPacket = ((TextPacket)content);
         String s = inPacket.text;
         String output = s.substring(3); // remove destination header
         System.out.println(output + "\n");
         notify();
      }
   }
   catch(Exception e) {e.printStackTrace();}
}
```

Listing 1: The onReceipt() method handles incoming packets for the clients. This method is the same for every client (Claptop, Plaptop). When a packet is received, the text string is extracted from the packet and the first 3 characters containing the destination address are removed. Then the string is printed to console so that the user can see the response from the Bank account (DServer) they where trying to access.

```
public synchronized void start() throws Exception {
      Scanner scanner = new Scanner(System.in);
      while (true) {
         System.out.print("Would you like to check your balance (1) or make a deposit (2)? ");
         String input = scanner.next();
         if (input.equals("exit")) {
            scanner.close();
            return;
         }
         else if (input.equals("2")) {
            System.out.print("Enter an amount to deposit: ");
            input = scanner.next();
         }
         else {
            input = "balance";
         }
```

Listing 2: The start() method of every client runs when the client is started or when notify() is called. This method handles the user interaction and sending of packets. First it asks if the user if they would like to make a deposit in the bank account account, or just check the current balance of the account. The user can enter ”exit” here to quit the program. If they choose to make a deposit, it will ask them how much money they would like to deposit.

```
        int serverNum = 1;
         boolean hasInt = false;
         while(!hasInt) {
            System.out.print("Which bank account would you like to access (1 or 2)?: ");
            if (scanner.hasNextInt()) {
               serverNum = scanner.nextInt();
               hasInt = true;
            }
            else {
               System.out.println("Please enter a number!");
            }
         }

         String dest;
         switch (serverNum) {
            case 1:
               dest = "000"; // DServer
               break;
            case 2:
               dest = "001"; // DServer2
               break;
            default:
               dest = "000";
         }

         input = dest + CLAPTOP_ID + input; // add return address

         DatagramPacket request;
         request = new TextPacket(input).toDatagramPacket();
         request.setSocketAddress(dstAddress);
         socket.send(request);
         this.wait();
```

Listing 3: The start() method then asks the user which bank account (DServer) they would like to access. I am using binary for the destination addresses. Depending on the number entered, the corresponding DServer address will be attached to the beginning of the string being sent to the DServer, along with the return address of the client (in this case Claptop). This string will then be put into a Text packet that is sent to the client's local forwarder (GW1 for Claptop).

3.2 Forwarding Service Implementation

```
public void onReceipt(DatagramPacket packet) {
     try {
        System.out.println("Received packet");
        PacketContent content= PacketContent.fromDatagramPacket(packet);

        if (content.getType()==PacketContent.TEXTPACKET) {
           TextPacket inPacket = ((TextPacket)content);
            dest = (inPacket.text).substring(0, 3); // isolate destination of packet

           if (nextJump.containsKey(dest)) {
              int destPort = nextJump.get(dest);
              System.out.println("Found next jump in forwarding table.");
              System.out.println("Forwarded packet to " + nodeList.get(destPort));
              dstAddress = new InetSocketAddress(nodeList.get(destPort), destPort);
              packet.setSocketAddress(dstAddress);
              socket.send(packet);
           }
           else {
                      currentPacket = packet;
              System.out.println("Next jump not stored. Requesting next jump from Controller...");
              dstAddress = new InetSocketAddress(CONTROLLER_NODE, CONTROLLER_PORT);
              packet.setSocketAddress(dstAddress);
              socket.send(packet);
           }
        }
```

Listing 4: The onReceipt in every forwarding service handles two different types of incoming packets. When a Text packet is received, the string inside will be extracted and the first 3 characters will be isolated, as this is the destination addess of the packet. It will then check if the forwarding table contains the next jump for that destination. The forwarding table is a hashmap that contains a corresponding node for each destination address so that the forwarder can quickly find out where it needs to send the packet next in order to get it to it's destination. If the destination is not included in the forwarding table, then the packet will be sent to the Controller which should return where to send the packet to. The variable "currentPacket" of type DatagramPacket holds the packet for when a packet is returned from the controller, so that it is not overwritten and can be sent on.

```
else if (content.getType()==PacketContent.NEXTNODE) {
   Random random = new Random();
   int num = random.nextInt(6);
   if (num != 0) {
      JumpPacket inPacket = ((JumpPacket)content);
      int port = inPacket.getPort();

      DatagramPacket ackPacket;
      ackPacket= new AckPacketContent("Updated GW1 forwarding table").toDatagramPacket();
      dstAddress = new InetSocketAddress(CONTROLLER_NODE, CONTROLLER_PORT);
      ackPacket.setSocketAddress(dstAddress);
      socket.send(ackPacket);

      System.out.println("Controller returned next jump - updated forwarding table.");
      System.out.println("Forwarded packet to " + nodeList.get(port));

      nextJump.put(dest, port); // adds next node to hashmap for later use
      dstAddress = new InetSocketAddress(nodeList.get(port), port);
      currentPacket.setSocketAddress(dstAddress);
      socket.send(currentPacket);
   }
   else {
      System.out.println("There was disturbance in the network - next node was not
          received!!!");
   }
 }
```

Listing 5: If the forwarding service receives a Jump packet (NEXTNODE) then the onReceipt() method knows this is from the controller. This packet contains the port of the next node to send the packet to. The port contained in the packet is extracted from the packet and added to the forwarding table for that address. Then the socket address is created using the port from the packet and the corresponding node name (retrieved from a separate hashmap of node names). An acknowledgment packet is sent to the controller to let it know that the forwarding table was successfully updated, and then the text packet is sent on to the next node.
In the real world there would be interference in the network and there is a possibility that the packet would not be received by the controller. So to simulate this, there is a chance that the forwarder does nothing when it receives the Jump packet, and the controller is forced to send it again.

3.3 Controller Implementation

```
Controller(int port) {
     try {
        ArrayList<int[]> dest1 = new ArrayList<int[]>();
        dest1.add(new int[]{GW1_PORT, ISP_PORT, CP_PORT, DSERVER_PORT}); // path to DServer (from
            GW1)
        dest1.add(new int[]{GW2_PORT, ISP_PORT, CP_PORT, DSERVER_PORT}); // path to DServer (from
            GW2)
        ArrayList<int[]> dest2 = new ArrayList<int[]>();
        dest2.add(new int[]{GW1_PORT, ISP_PORT, CP_PORT, DSERVER2_PORT}); // path to DServer2
            (from GW1)
        dest2.add(new int[]{GW2_PORT, ISP_PORT, CP_PORT, DSERVER2_PORT}); // path to DServer2
            (from GW2)
        ArrayList<int[]> dest3 = new ArrayList<int[]>();
        dest3.add(new int[]{CP_PORT, ISP_PORT, GW1_PORT, CLAPTOP_PORT}); // path to CLaptop
        ArrayList<int[]> dest4 = new ArrayList<int[]>();
        dest4.add(new int[]{CP_PORT, ISP_PORT, GW2_PORT, PLAPTOP_PORT}); // path to PLaptop

        map.put("000", dest1); // DServer
        map.put("001", dest2); // DServer2
        map.put("010", dest3); // Claptop
        map.put("011", dest4); // Plaptop

        socket= new DatagramSocket(port);
        listener.go();
     }
     catch(java.lang.Exception e) {e.printStackTrace();}
  }
```

Listing 6: This shows the Controller's constructor that populates the hashmap with the paths to the DServers the clients. This is so the controller can return the next node in the path to each destination.

```
public void onReceipt(DatagramPacket packet) {
     try {
        PacketContent content= PacketContent.fromDatagramPacket(packet);

        if (content.getType()==PacketContent.TEXTPACKET) {
           System.out.println("Received packet");
           TextPacket inPacket = ((TextPacket)content);
           String dest = (inPacket.text).substring(0, 3); // isolate destination in header
           System.out.println("Destination: " + dest);
           ArrayList<int[]> paths = map.get(dest); // gets path to destination
           boolean foundJump = false;

           for (int i=0; i<paths.size() && !foundJump; i++)
           {
              int[] array = paths.get(i);
              for (int j=0; j<array.length && !foundJump; j++) // find current position
              {
                 if (packet.getPort() == array[j])
                 {
                    JumpPacket nextJump = new JumpPacket(array[j+1]); // get next node
                    System.out.println("Next jump is to port: " + nextJump);
                    returnPacket = nextJump.toDatagramPacket();
                    returnPacket.setSocketAddress(packet.getSocketAddress());
                    socket.send(returnPacket);
                    foundJump = true;
```

```
                }
            }
        }
        t = new Timer();
        t.schedule(new rt(), seconds*1000); //schedule the timer
      }
      else if (content.getType()==PacketContent.ACKPACKET) {
        t.cancel(); //stop the thread of timer
        System.out.println("Received acknowledgment: " +
            ((AckPacketContent)content).getPacketInfo());
      }
    }
    catch(Exception e) {e.printStackTrace();}
  }
```

Listing 7: The controller's onReceipt() method handles incoming Text packets and ACK packets. If a Text packet is received then the destination is taken from the header and get the list of paths to that destination (from the paths hashmap). The outer for-loop will find the path that includes the current node, so that the inner for-loop can get the next node after it. Once the next node has been found, node's port will be returned to the forwarding service in a Jump packet. Then a 1 second timer will be set, so that if an ACK packet is not received within 1 seconds of sending the Jump packet, the controller will assume the packet was not received and send it again. If an ACK packet is received, it will disable the timer.

```
   class rt extends TimerTask {
    public void run() {
      try{
        socket.send(returnPacket); // Send Packet again if it wasn't received
        t.cancel(); //stop the thread of timer
        t = new Timer();
        t.schedule(new rt(), seconds*1000); //schedule the timer
      }
      catch(Exception e) {e.printStackTrace();}
    }
  }
```

Listing 8: Shows the function that is called when the controller's timer is completed. When 5 seconds have passed, the controller will send the same packet again and then reset the timer.

3.4 DServer Implementation

```
public void onReceipt(DatagramPacket packet) {
      try {
         PacketContent content= PacketContent.fromDatagramPacket(packet);

         if (content.getType()==PacketContent.TEXTPACKET) {
            System.out.println("Received request packet");

             TextPacket returnPacket;
            String returnString;

             TextPacket inPacket = ((TextPacket)content);
            String dest = (inPacket.text).substring(3, 6); // isolate return destination
            String s = (inPacket.text).substring(6); // remove header
            System.out.println("Packet content: "+ s);
            System.out.println("Return Destination: " + dest);

            dstAddress = new InetSocketAddress(CP_NODE, CP_PORT);

            if (s.contains("balance")) { // SEND DATA REQUESTED
               System.out.println("Checking Account Balance...");
               System.out.println("Balance is: $" + balance);

               returnString = dest + "Your current balance is $" + balance + " (Bank Account 1)";
               returnPacket = new TextPacket(returnString);
               sendReturnPacket(returnPacket);
            }
            else if (Character.isDigit(s.charAt(0))) {
               double deposit = Double.parseDouble(s.trim());
               balance = balance + deposit;
               returnString = dest + "Deposit was successful - new balance is $" + balance + "
                    (Bank Account 1)";
               returnPacket = new TextPacket(returnString);
               sendReturnPacket(returnPacket);
            }
         }
      }
      catch(Exception e) {e.printStackTrace();}
   }
```

Listing 9: Shows the DServers' onReceipt() method that handles all incoming text packets. When a text packet is received, the string inside is extracted and the return address in the header is isolated. The header is removed from the string and if the text says "balance" then it will make a new text packet with the current bank account balance and the return address in the header. If the text contains a number, then it adds this number to the balance as a deposit, and then returns the new balance.

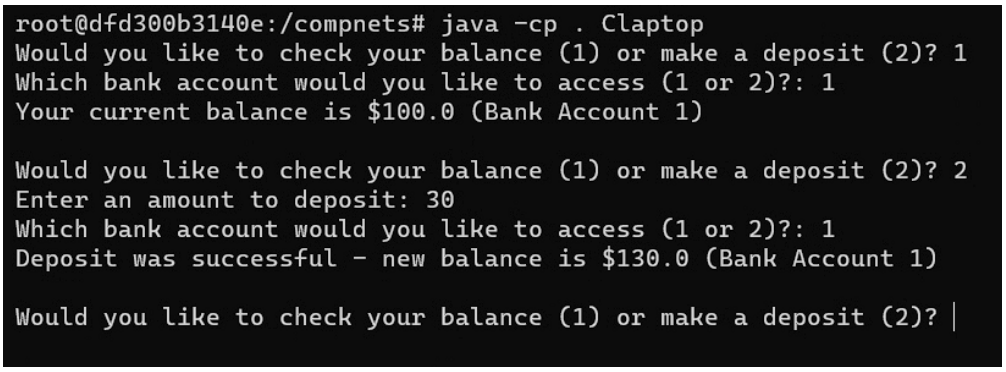
3.5 User Interaction

The user can choose which account they want to access, and they can choose to make a deposit or just check the current balance in the account.

3.6 Packet Encoding

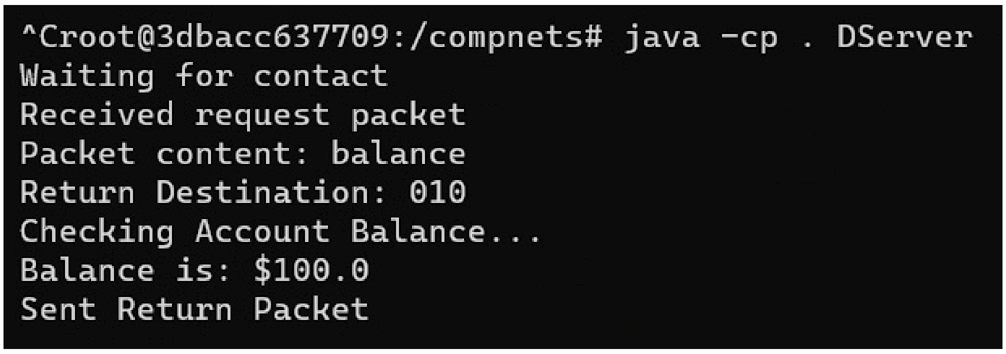
My program uses UDP datagram packets. The Text packet contains a String, Jump packet contains an integer value, and ACK packet contains a String. These are all written to the ObjectOutputStream.

4 Demonstration



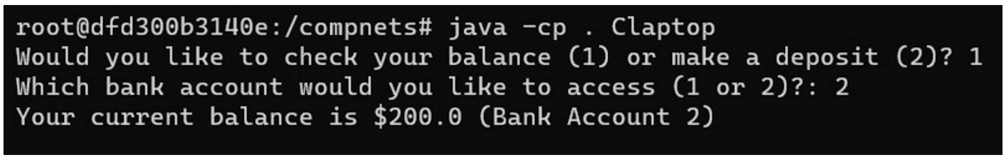
```
root@dfd300b3140e:/compnets# java -cp . Claptop
Would you like to check your balance (1) or make a deposit (2)? 1
Which bank account would you like to access (1 or 2)?: 1
Your current balance is $100.0 (Bank Account 1)

Would you like to check your balance (1) or make a deposit (2)? 2
Enter an amount to deposit: 30
Which bank account would you like to access (1 or 2)?: 1
Deposit was successful - new balance is $130.0 (Bank Account 1)

Would you like to check your balance (1) or make a deposit (2)? |
```

Figure 2: We can see here that the user first requested the balance of Bank account 1, and the current balance was returned from the bank. Then they requested to make a deposit of 30 dollars into Bank account 1. The bank account returned that the deposit was successful and that the new balance is 130 dollars.



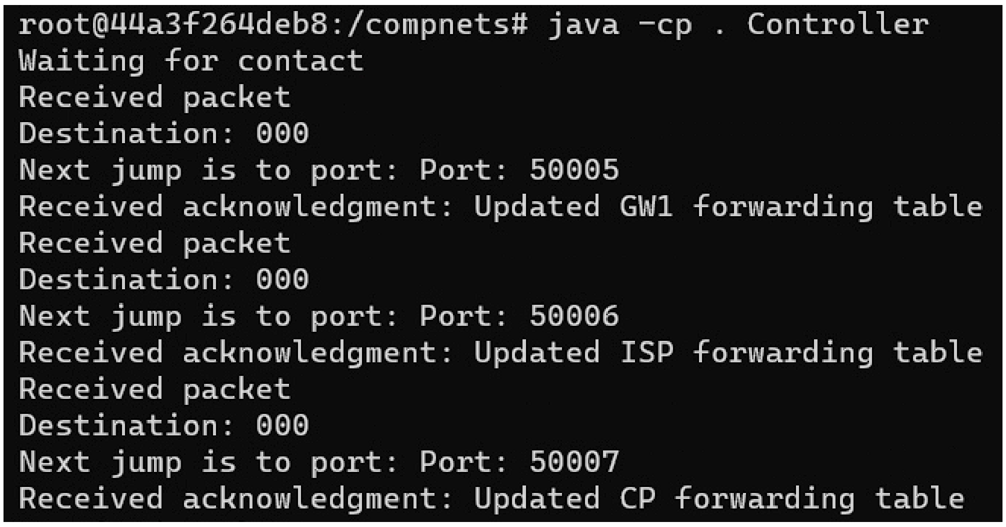
```
^Croot@3dbacc637709:/compnets# java -cp . DServer
Waiting for contact
Received request packet
Packet content: balance
Return Destination: 010
Checking Account Balance...
Balance is: $100.0
Sent Return Packet
```

Figure 3: When the Bank account (DServer) receives the packet it finds the return address of Claptop in the header (010) and sends a packet containing the balance back to the Claptop.



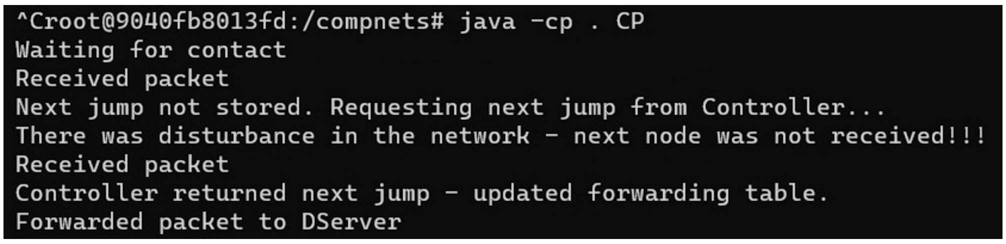
```
root@dfd300b3140e:/compnets# java -cp . Claptop
Would you like to check your balance (1) or make a deposit (2)? 1
Which bank account would you like to access (1 or 2)?: 2
Your current balance is $200.0 (Bank Account 2)
```

Figure 4: Here the user is requesting the balance from the second bank account (DServer2).



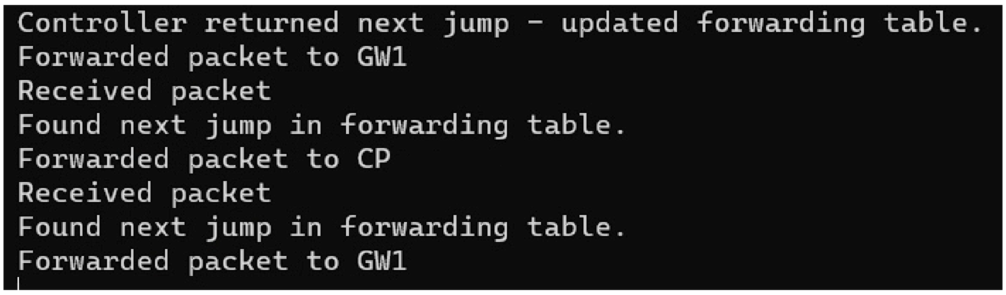
```
root@44a3f264deb8:/compnets# java -cp . Controller
Waiting for contact
Received packet
Destination: 000
Next jump is to port: Port: 50005
Received acknowledgment: Updated GW1 forwarding table
Received packet
Destination: 000
Next jump is to port: Port: 50006
Received acknowledgment: Updated ISP forwarding table
Received packet
Destination: 000
Next jump is to port: Port: 50007
Received acknowledgment: Updated CP forwarding table
```

Figure 5: We can see here that when the controller receives a packet, it checks the path to the destination 000 (DServer) and then returns the port of the next node.



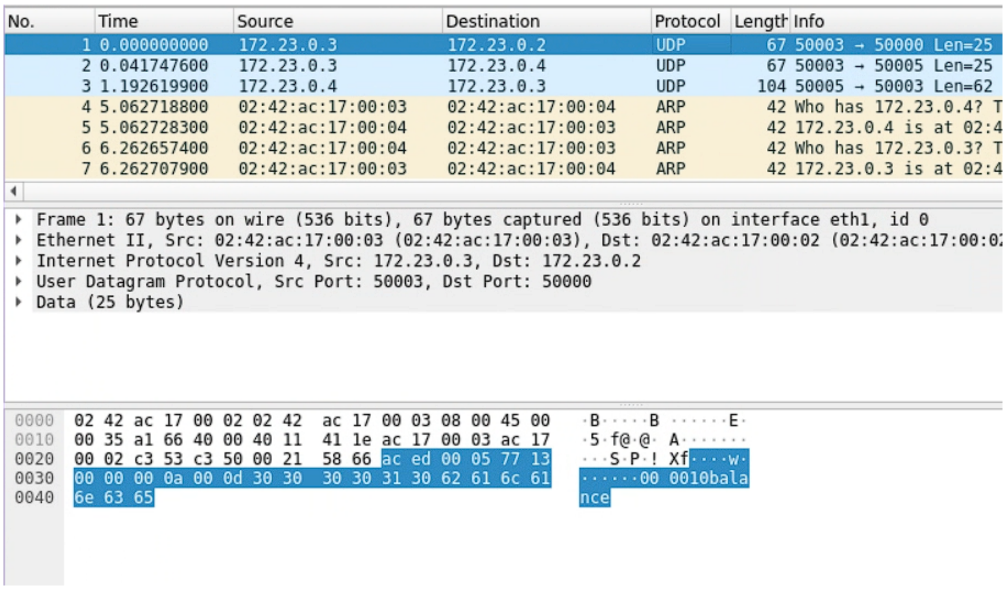
```
^Croot@9040fb8013fd:/compnets# java -cp . CP
Waiting for contact
Received packet
Next jump not stored. Requesting next jump from Controller...
There was disturbance in the network - next node was not received!!!
Received packet
Controller returned next jump - updated forwarding table.
Forwarded packet to DServer
```

Figure 6: We can see here that there was some interference in the network and the forwarding service (ISP in this case) did not receive the Jump packet from the controller on the first try, so the controller was forced to send it again. It was successfully received the second time and the forwarding table was updated with the next node.



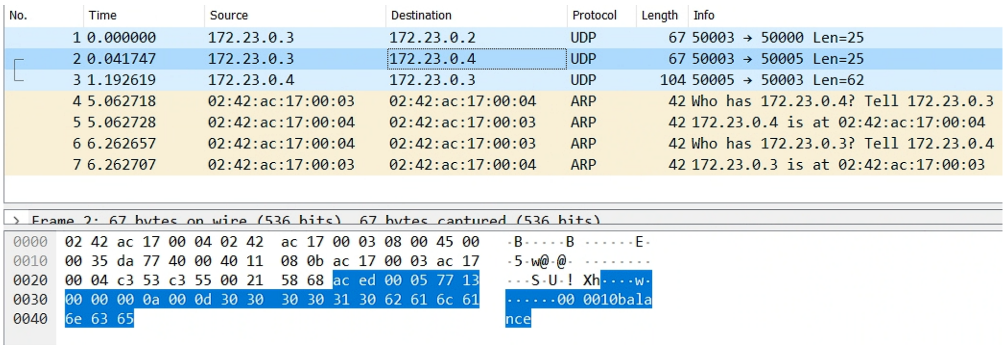
```
Controller returned next jump - updated forwarding table.
Forwarded packet to GW1
Received packet
Found next jump in forwarding table.
Forwarded packet to CP
Received packet
Found next jump in forwarding table.
Forwarded packet to GW1
```

Figure 7: Once the forwarding service has updated it's forwarding table with the next node to get to a destination, then it no longer needs to ask the controller as it can just consult the forwarding table far more efficient.



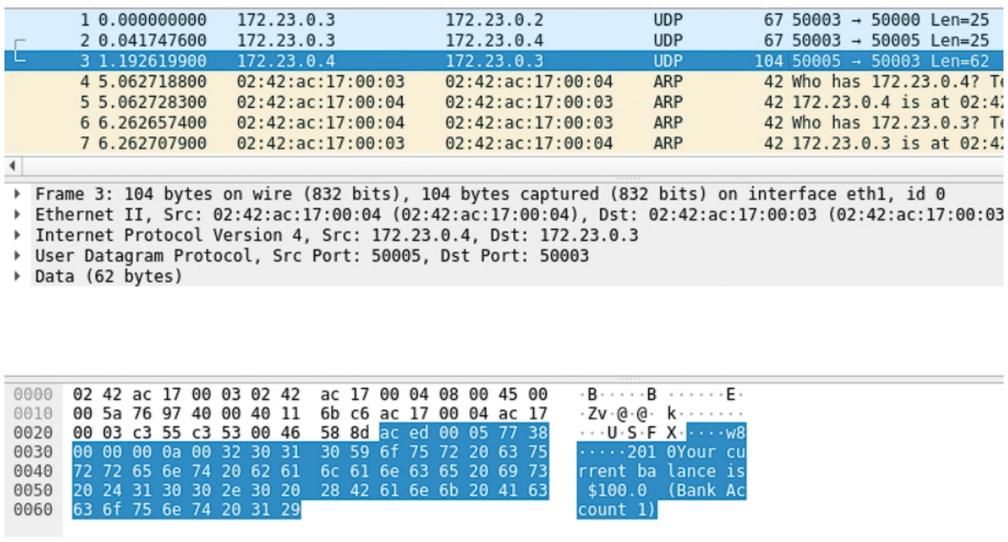
```
No.  Time          Source              Destination         Protocol Length Info
   1 0.000000000   172.23.0.3          172.23.0.2          UDP          67 50003 → 50000 Len=25
   2 0.041747600   172.23.0.3          172.23.0.4          UDP          67 50003 → 50005 Len=25
   3 1.192619900   172.23.0.4          172.23.0.3          UDP         104 50005 → 50003 Len=62
   4 5.062718800   02:42:ac:17:00:03   02:42:ac:17:00:04   ARP          42 Who has 172.23.0.4? T
   5 5.062728300   02:42:ac:17:00:04   02:42:ac:17:00:03   ARP          42 172.23.0.4 is at 02:4
   6 6.262657400   02:42:ac:17:00:04   02:42:ac:17:00:03   ARP          42 Who has 172.23.0.3? T
   7 6.262707900   02:42:ac:17:00:03   02:42:ac:17:00:04   ARP          42 172.23.0.3 is at 02:4

▸ Frame 1: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface eth1, id 0
▸ Ethernet II, Src: 02:42:ac:17:00:03 (02:42:ac:17:00:03), Dst: 02:42:ac:17:00:02 (02:42:ac:17:00:0
▸ Internet Protocol Version 4, Src: 172.23.0.3, Dst: 172.23.0.2
▸ User Datagram Protocol, Src Port: 50003, Dst Port: 50000
▸ Data (25 bytes)

0000  02 42 ac 17 00 02 02 42  ac 17 00 03 08 00 45 00   ·B·····B ······E·
0010  00 35 a1 66 40 00 40 11  41 1e ac 17 00 03 ac 17   ·5·f@·@· A·······
0020  00 02 c3 53 c3 50 00 21  58 66 ac ed 00 05 77 13   ···S·P·! Xf····w·
0030  00 00 00 0a 00 0d 30 30  30 30 31 30 62 61 6c 61   ······00 0010bala
0040  6e 63 65                                           nce
```

Figure 8: We can see from the contents of this packet ”000010balance”, that the destination address is 000 (DServer) and the return address is 010 (Claptop), and that the user is requesting the balance. This packet is being sent from the ISP (port 50003) to the controller (port 50000) because the destination is not in the forwarding table.



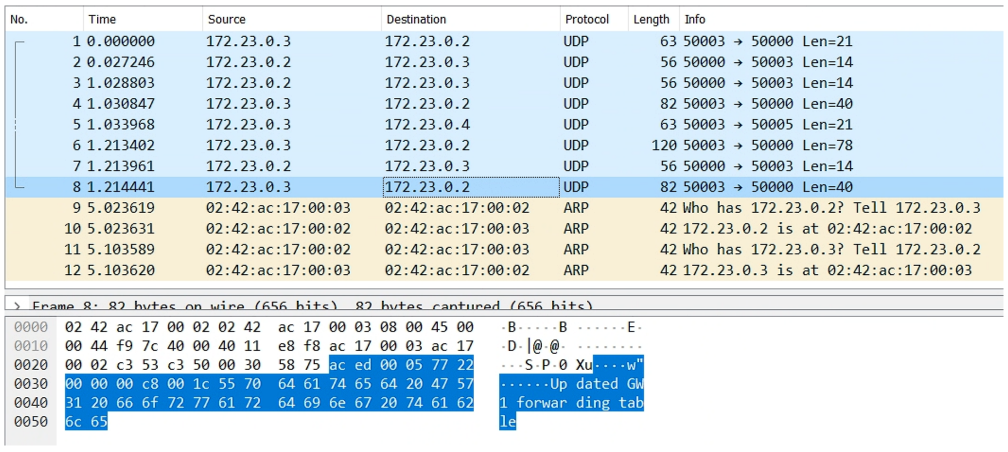
```
No.  Time      Source              Destination         Protocol Length Info
   1 0.000000  172.23.0.3          172.23.0.2          UDP          67 50003 → 50000 Len=25
   2 0.041747  172.23.0.3          172.23.0.4          UDP          67 50003 → 50005 Len=25
   3 1.192619  172.23.0.4          172.23.0.3          UDP         104 50005 → 50003 Len=62
   4 5.062718  02:42:ac:17:00:03   02:42:ac:17:00:04   ARP          42 Who has 172.23.0.4? Tell 172.23.0.3
   5 5.062728  02:42:ac:17:00:04   02:42:ac:17:00:03   ARP          42 172.23.0.4 is at 02:42:ac:17:00:04
   6 6.262657  02:42:ac:17:00:04   02:42:ac:17:00:03   ARP          42 Who has 172.23.0.3? Tell 172.23.0.4
   7 6.262707  02:42:ac:17:00:03   02:42:ac:17:00:04   ARP          42 172.23.0.3 is at 02:42:ac:17:00:03

> Frame 2: 67 bytes on wire (536 bits), 67 bytes captured (536 bits)

0000  02 42 ac 17 00 04 02 42  ac 17 00 03 08 00 45 00   ·B·····B ······E·
0010  00 35 da 77 40 00 40 11  08 0b ac 17 00 03 ac 17   ·5·w@·@· ········
0020  00 04 c3 53 c3 55 00 21  58 68 ac ed 00 05 77 13   ···S·U·! Xh····w·
0030  00 00 00 0a 00 0d 30 30  30 30 31 30 62 61 6c 61   ······00 0010bala
0040  6e 63 65                                           nce
```

Figure 9: The ISP has received the next node from the Controller and is forwarding the packet to the Cloud Provider.



```
   1 0.000000000   172.23.0.3          172.23.0.2          UDP         67 50003 → 50000 Len=25
   2 0.041747600   172.23.0.3          172.23.0.4          UDP         67 50003 → 50005 Len=25
   3 1.192619900   172.23.0.4          172.23.0.3          UDP        104 50005 → 50003 Len=62
   4 5.062718800   02:42:ac:17:00:03   02:42:ac:17:00:04   ARP         42 Who has 172.23.0.4? Te
   5 5.062728300   02:42:ac:17:00:04   02:42:ac:17:00:03   ARP         42 172.23.0.4 is at 02:42
   6 6.262657400   02:42:ac:17:00:04   02:42:ac:17:00:03   ARP         42 Who has 172.23.0.3? Te
   7 6.262707900   02:42:ac:17:00:03   02:42:ac:17:00:04   ARP         42 172.23.0.3 is at 02:42

▸ Frame 3: 104 bytes on wire (832 bits), 104 bytes captured (832 bits) on interface eth1, id 0
▸ Ethernet II, Src: 02:42:ac:17:00:04 (02:42:ac:17:00:04), Dst: 02:42:ac:17:00:03 (02:42:ac:17:00:03
▸ Internet Protocol Version 4, Src: 172.23.0.4, Dst: 172.23.0.3
▸ User Datagram Protocol, Src Port: 50005, Dst Port: 50003
▸ Data (62 bytes)

0000  02 42 ac 17 00 03 02 42  ac 17 00 04 08 00 45 00   ·B·····B ······E·
0010  00 5a 76 97 40 00 40 11  6b c6 ac 17 00 04 ac 17   ·Zv·@·@· k·······
0020  00 03 c3 55 c3 53 00 46  58 8d ac ed 00 05 77 38   ···U·S·F X·····w8
0030  00 00 00 0a 00 32 30 31  30 59 6f 75 72 20 63 75   ·····201 0Your cu
0040  72 72 65 6e 74 20 62 61  6c 61 6e 63 65 20 69 73   rrent ba lance is
0050  20 24 31 30 30 2e 30 20  28 42 61 6e 6b 20 41 63    $100.0  (Bank Ac
0060  63 6f 75 6e 74 20 31 29                            count 1)
```

Figure 10: This packet has been forwarded from the Cloud Provider to the ISP. It is the DServer's response to a request for the balance. Contained in the packet is the message "010Your current balance is 100.0 (Bank Account 1)". We can see that the destination address in the header is 010 (Claptop).



```
No.   Time       Source              Destination         Protocol  Length Info
   1  0.000000   172.23.0.3          172.23.0.2          UDP           63 50003 → 50000 Len=21
   2  0.027246   172.23.0.2          172.23.0.3          UDP           56 50000 → 50003 Len=14
   3  1.028803   172.23.0.2          172.23.0.3          UDP           56 50000 → 50003 Len=14
   4  1.030847   172.23.0.3          172.23.0.2          UDP           82 50003 → 50000 Len=40
   5  1.033968   172.23.0.3          172.23.0.4          UDP           63 50003 → 50005 Len=21
   6  1.213402   172.23.0.3          172.23.0.2          UDP          120 50003 → 50000 Len=78
   7  1.213961   172.23.0.2          172.23.0.3          UDP           56 50000 → 50003 Len=14
   8  1.214441   172.23.0.3          172.23.0.2          UDP           82 50003 → 50000 Len=40
   9  5.023619   02:42:ac:17:00:03   02:42:ac:17:00:02   ARP           42 Who has 172.23.0.2? Tell 172.23.0.3
  10  5.023631   02:42:ac:17:00:02   02:42:ac:17:00:03   ARP           42 172.23.0.2 is at 02:42:ac:17:00:02
  11  5.103589   02:42:ac:17:00:02   02:42:ac:17:00:03   ARP           42 Who has 172.23.0.3? Tell 172.23.0.2
  12  5.103620   02:42:ac:17:00:03   02:42:ac:17:00:02   ARP           42 172.23.0.3 is at 02:42:ac:17:00:03

> Frame 8: 82 bytes on wire (656 bits), 82 bytes captured (656 bits)
0000  02 42 ac 17 00 02 02 42  ac 17 00 03 08 00 45 00   ·B·····B ······E·
0010  00 44 f9 7c 40 00 40 11  e8 f8 ac 17 00 03 ac 17   ·D·|@·@· ········
0020  00 02 c3 53 c3 50 00 30  58 75 ac ed 00 05 77 22   ···S·P·0 Xu····w"
0030  00 00 00 c8 00 1c 55 70  64 61 74 65 64 20 47 57   ······Up dated GW
0040  31 20 66 6f 72 77 61 72  64 69 6e 67 20 74 61 62   1 forwar ding tab
0050  6c 65                                              le
```

Figure 11: Here we can see the controller receiving an ACK packet from the forwarder GW1 stating that it's forwarding table was successfully updated with the modification from the Controller.

5 Summary

This report has described my attempt at a solution to address the flow of packets between network elements using forwarding services. I described the concepts and mechanisms that I implemented into my design. I showed some snippets of my code and explained it. Lastly, I demonstrated my working program, with screenshots of the terminal and Wireshark as it executed.

6 Reflection

Over the course of completing this assignment, I learned a lot about how packets are directed across the internet. I discovered how forwarding services with controllers are used to direct the flow of traffic across networks. I learned how forwarding services are made more efficient by utilising forwarding tables to store past routes so they don't have to the query the controller every time a packet is received.

I now feel I know how a system like this would work in the real world and the changes I would make if I where implementing on a larger scale. Firstly, I would have added a shortest path algorithm to the controller (e.g Dijkstra), so that instead of the paths being hard coded, it would find the shortest path to a destination itself through all the forwarders. This would be more scalable than hard coding every path. different flow control protocol other than Simplest that uses acknowledgements. If I had more time, I would have added more DServers and more clients. Despite this, I am happy with my completed solution in the end.
All in all, excluding the time taken to write the report, the assignment took me about 60 hours to complete.