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

CSU33031 Computer Networks Assignment #2: Flow Forwarding

Yi Ren, 20304391

December 4, 2022

Contents

1 Introduction

This assignment focuses on learning about decisions to forward flows of packets and the information that is kept at network devices that make forwarding decisions. The aim of our protocol is to reduce the processing required by network elements that forward traffic while providing scalability and flexibility. For my protocol, it involves four types of actors: one controller, one deployment server, several employee applications, and each has one forwarder connected with it. It allows the applications under different networks to communicate with each other through the forwarding service.

2 Theory of Topic

In this section, I will start with discussing the basic functionality. The additional functionality for the sake of efficiency or flow control will be discussed afterwards.

2.1 Basic Functionality

The basic functionality for this protocol is to accept incoming packets, inspect the header information, consult the forwarding table and forward the header and payload information to the destination. The forwarding table was hardcoded first so that I can focus on the communication between nodes. After the communication succeeded, I started to implement the function of the controller. Once a forwarder starts running, it sends hello message to the controller and gets its initial forwarding table including only the neighbours around. One of the applications issues request to send message to the server, the forwarder connected to it receives the request and looks up its forwarding table to find the server. If server exists, then forwards the request packet to the next hop according to the forwarding table. If server does not exist, then sends a consultation to the controller. The controller thus look for the shortest path from the forwarder to the server, and sends back information of the next hop. The forwarder hence updates its forwarding table and forwards the packet to the next hop. This process is repeated in every forwarders the packet passes through until it reaches the server. Acknowledgement messages(ACK) are sent to the previous packet sender after each packet-transfer so that we are able to monitor the transfer from the console.

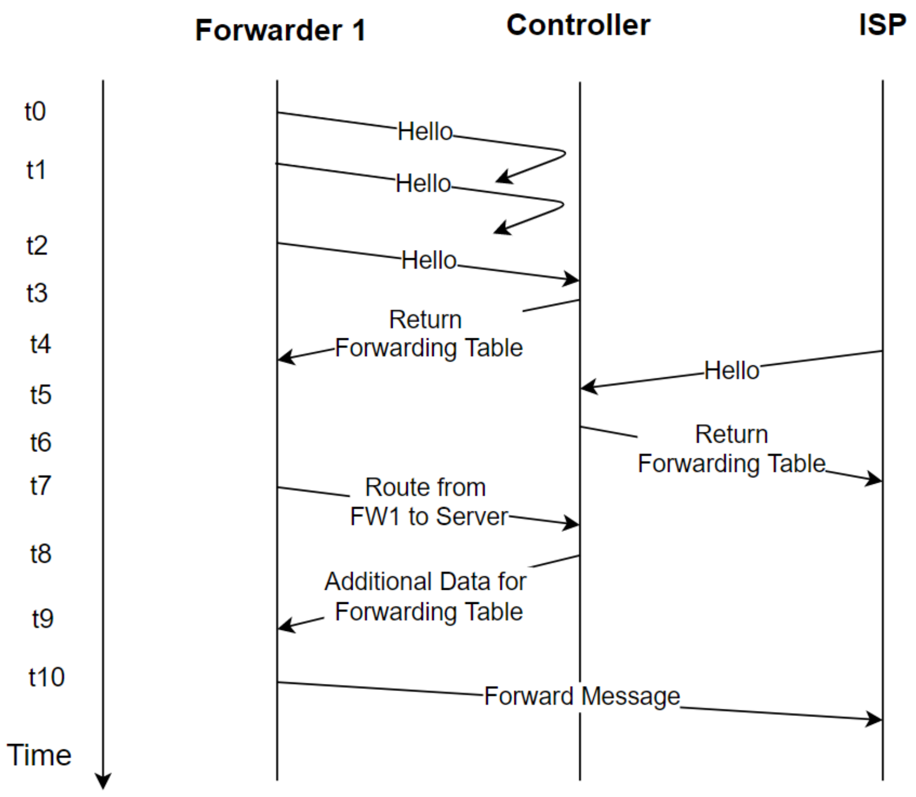


Figure 1: Flow diagram for the process of Forwarder 1 forwards a packet to an unknown address. The forwarder keeps sending hello message to the controller until the controller is implemented.

2.1.1 Packets

There are six types of packets that will be sent among actors:

- A Request Packet which contains the message sent by the client to the destination.

- An Acknowledgement(ACK) Packet which includes the acknowledgement message of the packet receipt.
- A Hello Packet which contains a string the hello message sent from the forwarder to the controller.
- A Consult Packet which contains the consultation sent from the forwarder to the controller when an unknown destination occurs. The string sent as the payload includes both its own ID and the destination ID, separated by a comma.
- A TableMod Packet which contains the initial forwarding table sent back from the controller to the forwarder.
- An Update Packet which contains a reply to the consultation that sent from the forwarder. It contains the destination ID in the packet header with the ID of the next hop as a payload.

2.1.2 Nodes

I have two employee applications, two forwarders, one Internet service provider(ISP), one cloud provider(CP) and one deployment server as nodes in my topology.

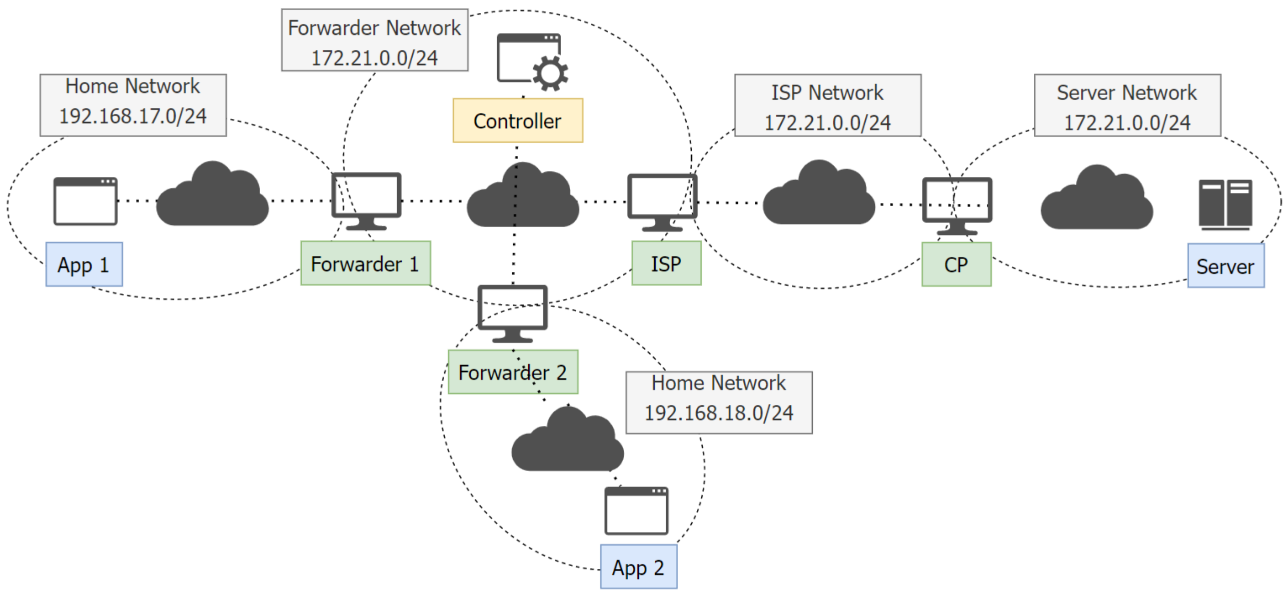


Figure 2: This topology is kept to a minimum by showing a network of two employee, the network of a provider and a network for a cloud provider with an application server in the same network.

- Client: A class designed for the employee applications, named app1 and app2.
- Forwarder: A class designed for the forwarders, named fw1, fw2, isp and cp.
- Controller: A class designed for the controller.

2.1.3 User Input Handling

The program only allows user input in Client class and Forwarder class. Both classes take in user input as ID code when it starts, so that the hostname and the corresponding forwarder/application name can be generated by the program instead of being hardcoded.

```
System.out.print("Welcome␣to␣the␣forwarding␣service␣system!\nEnter␣ID␣code:␣");
String code = input.next();
String hostname = "app" + code;
String gateway = "fw" + code;
int programStatus = 1;
Client clt = new Client(gateway, hostname, DEFAULT_DST_PORT, DEFAULT_SRC_PORT);
```

Listing 1: Client Class Pseudo-code

```
System.out.print("Enter ID code: ");
String code = input.next();
if(code.equalsIgnoreCase("cp") || code.equalsIgnoreCase("isp")){
hostname = code;
clientname = code.equalsIgnoreCase("cp") ? "dserver" : "";
}
else{
hostname = "fw" + code;
clientname = "app" + code;
}
(new Forwarder(hostname, clientname, DEFAULT_PORT)).start();
```

Listing 2: Forwarder Class Pseudo-code

2.2 Flow Control

2.2.1 Stop-and-Wait ARQ

In our basic functionality, we have completed a system using a Stop-and-Wait protocol that waits for and acknowledgement(ACK) after each packet transfer. However, we cannot solve the problem if the packet is dropped. Since the receiver will never get the packet, the ACK will never be sent. Since it never receives the acknowledgement it is expecting, the sender will never know and never take actions to send the packet again. Thus, I applied an automatic repeat-request(ARQ) protocol. For every packet that is sent, a timer will be created. When the timer expires, the corresponding packet will be sent again and a new timer started. The timer is destroyed when the ACK is received. A variable *count* is set to control the resend times. When it reaches 30 times, the sender will stop re-sending packet and ask the user to resend the message manually.

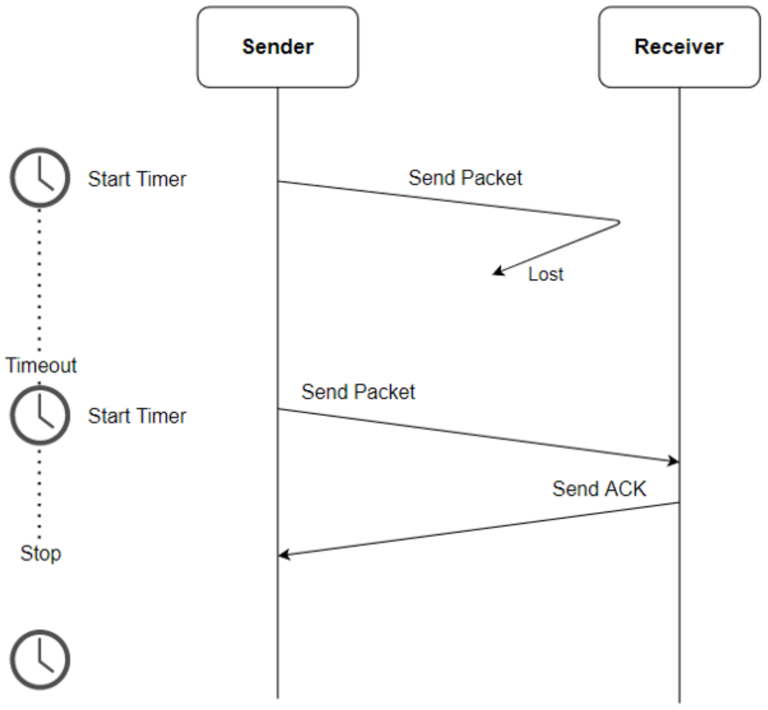
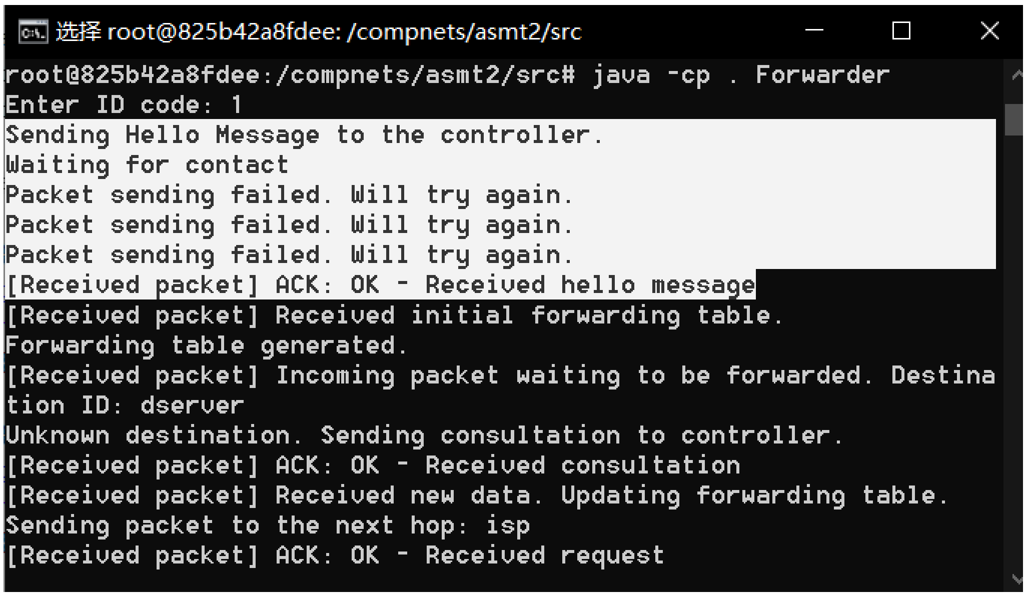


Figure 3: Flow diagram for ARQ protocol



```
选择 root@825b42a8fdee: /compnets/asmt2/src
root@825b42a8fdee:/compnets/asmt2/src# java -cp . Forwarder
Enter ID code: 1
Sending Hello Message to the controller.
Waiting for contact
Packet sending failed. Will try again.
Packet sending failed. Will try again.
Packet sending failed. Will try again.
[Received packet] ACK: OK - Received hello message
[Received packet] Received initial forwarding table.
Forwarding table generated.
[Received packet] Incoming packet waiting to be forwarded. Destina
tion ID: dserver
Unknown destination. Sending consultation to controller.
[Received packet] ACK: OK - Received consultation
[Received packet] Received new data. Updating forwarding table.
Sending packet to the next hop: isp
[Received packet] ACK: OK - Received request
```

Figure 4: This is the console screenshot for the ARQ protocol. The highlighted part shows the forwarder 1 sending out hello message to the controller while the controller was not yet opened. It kept sending hello packet until it received Acknowledgement back from the controller.

3 Implementation

An overview and a description of the implementation are provided in this section. I employed my last assignment on protocol design as a starting point, including the Node class, Client, Server and Sender classes and the AckPacketContent and RequestContent classes which extend the PacketContent class. They were modified in order to meet the requirements.

3.1 Header

I applied three bytes as the header of a packet.

- Packet Type
  The first byte of the header plays the role of specifying the packet type. I have added two more types beside ACKPacket and FileInfoPacket to make the sorting of the incoming packets easier.
- Origin ID
- Destination ID
  The second and third byte of the header is specially designed for the Request packet. Thus, for all other types of packets, their second byte of header should all be 0 as default. For the Request packet, the origin and destination ID are written to the header when the client is packing the request packet that includes messages, specifying where the packet is from, and where it should arrive. When dealing with the message forwarding, the forwarder will collect the third byte of the header, which is the destination ID, to decide the next hop that the packet should be forwarded to. When the packet is sent to the destination, the origin ID tells the destination application where this message comes from.

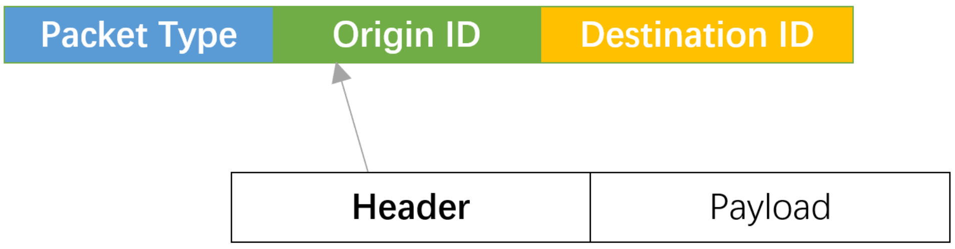
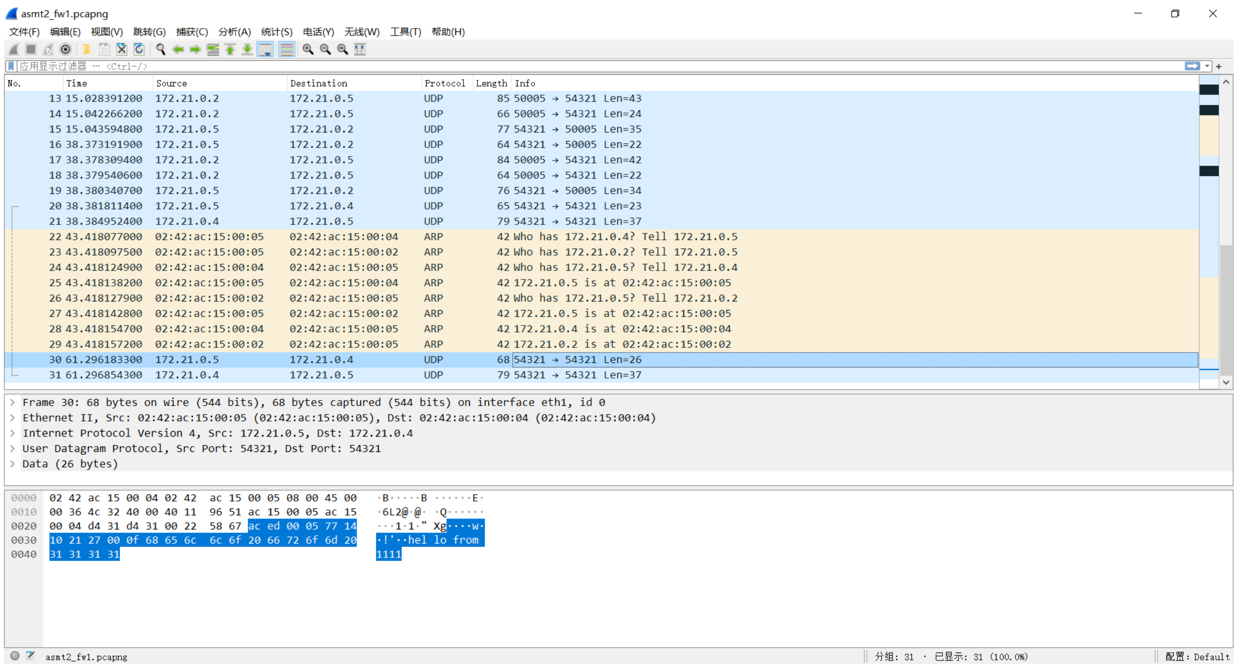


Figure 5: Visualisation of the header

```
REQUEST= 0b00010000; //0x10
HELLO= 0b00010001; //0x11
CONSULT= 0b00010010; //0x12
TABLEMOD= 0b00010011; //0x13
ACK= 0b00010100; //0x14
UPDATE= 0b00010101; //0x15
CONTROLLER= 0b00100000; //0x20
APP1= 0b00100001; //0x21
FW1= 0b00100010; //0x22
APP2= 0b00100011; //0x23
FW2= 0b00100100; //0x24
CP= 0b00100101; //0x25
ISP= 0b00100110; //0x26
SERVER= 0b00100111; //0x27
```

Listing 3: Headers


asmt2_fw1.pcapng
文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)
应用显示过滤器 … <Ctrl-/>
分组: 31 · 已显示: 31 (100.0%)
配置: Default

```
No.   Time            Source              Destination         Protocol Length Info
   13 15.028391200    172.21.0.2          172.21.0.5          UDP      85 50005 → 54321 Len=43
   14 15.042266200    172.21.0.2          172.21.0.5          UDP      66 50005 → 54321 Len=24
   15 15.043594800    172.21.0.5          172.21.0.2          UDP      77 54321 → 50005 Len=35
   16 38.373191900    172.21.0.5          172.21.0.2          UDP      64 54321 → 50005 Len=22
   17 38.378309400    172.21.0.2          172.21.0.5          UDP      84 50005 → 54321 Len=42
   18 38.379540600    172.21.0.2          172.21.0.5          UDP      64 50005 → 54321 Len=22
   19 38.380340700    172.21.0.5          172.21.0.2          UDP      76 54321 → 50005 Len=34
   20 38.381811400    172.21.0.5          172.21.0.4          UDP      65 54321 → 54321 Len=23
   21 38.384952400    172.21.0.4          172.21.0.5          UDP      79 54321 → 54321 Len=37
   22 43.418077000    02:42:ac:15:00:05   02:42:ac:15:00:04   ARP      42 Who has 172.21.0.4? Tell 172.21.0.5
   23 43.418097500    02:42:ac:15:00:05   02:42:ac:15:00:02   ARP      42 Who has 172.21.0.2? Tell 172.21.0.5
   24 43.418124900    02:42:ac:15:00:04   02:42:ac:15:00:05   ARP      42 Who has 172.21.0.5? Tell 172.21.0.4
   25 43.418138200    02:42:ac:15:00:05   02:42:ac:15:00:04   ARP      42 172.21.0.5 is at 02:42:ac:15:00:05
   26 43.418127900    02:42:ac:15:00:02   02:42:ac:15:00:05   ARP      42 Who has 172.21.0.5? Tell 172.21.0.2
   27 43.418142800    02:42:ac:15:00:05   02:42:ac:15:00:02   ARP      42 172.21.0.5 is at 02:42:ac:15:00:05
   28 43.418154700    02:42:ac:15:00:04   02:42:ac:15:00:05   ARP      42 172.21.0.4 is at 02:42:ac:15:00:04
   29 43.418157200    02:42:ac:15:00:02   02:42:ac:15:00:05   ARP      42 172.21.0.2 is at 02:42:ac:15:00:02
   30 61.296183300    172.21.0.5          172.21.0.4          UDP      68 54321 → 54321 Len=26
   31 61.296854300    172.21.0.4          172.21.0.5          UDP      79 54321 → 54321 Len=37

> Frame 30: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface eth1, id 0
> Ethernet II, Src: 02:42:ac:15:00:05 (02:42:ac:15:00:05), Dst: 02:42:ac:15:00:04 (02:42:ac:15:00:04)
> Internet Protocol Version 4, Src: 172.21.0.5, Dst: 172.21.0.4
> User Datagram Protocol, Src Port: 54321, Dst Port: 54321
> Data (26 bytes)

0000  02 42 ac 15 00 04 02 42  ac 15 00 05 08 00 45 00   ·B·····B ······E·
0010  00 36 4c 32 40 00 40 11  96 51 ac 15 00 05 ac 15   ·6L2@·@· ·Q······
0020  00 04 d4 31 d4 31 00 22  58 67 ac ed 00 05 77 14   ···1·1·" Xg····w·
0030  10 21 27 00 0f 68 65 6c  6c 6f 20 66 72 6f 6d 20   ·!'··hel lo from 
0040  31 31 31 31                                        1111

asmt2_fw1.pcapng
```

Figure 6: This is wireshark capture of one of the forwarders. The 30th row shows a Request packet, sending from app1 to the server.

3.2 ID Map

The ID mentioned in this program refers to the string form of hostnames. An ID_MAP is hard coded and stored in Node class. The ID_MAP is a Hashmap which plays a role as a dictionary to transfer the string of hostnames into the final bytes defined in the PacketContent class that act as header bytes. ID_MAP is used when writing the second and third byte of the header into a packet.

```
ID_MAP.put("app1", PacketContent.APP1);
ID_MAP.put("app2", PacketContent.APP2);
ID_MAP.put("dserver", PacketContent.SERVER);
```

Listing 4: ID$_MAPPseudo-code$

3.3 Actor Classes

The existing Client, Server classes and a new Controller class all extend the abstract class Node. The port numbers for all applications including employee applications and the deployment server are set to 50000, and the port number for all forwarders are set to 54321. The port number of the controller is 50005.

3.4 Forwarding Table

3.4.1 Initial Tables

When a forwarder is generated, it will first ask the controller for its initial forwarding table from the controller. The initial table includes only neighbours that are directly connected with this node.

Destination	Next Hop
app1	app1
fw2	fw2
isp	isp

Figure 7: This is the initial table for the forwarder 1, implemented using Hashmap in my Java code.

The initial connection among nodes is hardcoded in the controller according to my topology(see Figure 2). To collect initial forwarding table for a certain node, the controller looks up the overall forwarding table(see Figure?), locates to the row of the corresponding ID and collects all columns whose value is not -1.

	app1	fw1	app2	fw2	dserver	cp	isp
app1	0	1	MAX	MAX	MAX	MAX	MAX
fw1	1	0	MAX	1	MAX	MAX	1
app2	MAX	MAX	0	1	MAX	MAX	MAX
fw2	MAX	1	1	0	MAX	MAX	1
dserver	MAX	MAX	MAX	MAX	0	1	MAX
cp	MAX	MAX	MAX	MAX	1	0	1
isp	MAX	1	MAX	1	MAX	1	0

Figure 8: This is the initial distance table for all nodes. Since the actual time cost difference among different pairs of adjacent nodes is not important in this program, I set the distance between all pairs of adjacent nodes to 1 as default. The distance to the node itself is 0, and the distance between unconnected nodes is set to the MAX_VALUE.

	app1 (index=0)	fw1 (index=1)	app2 (index=2)	fw2 (index=3)	dserver (index=4)	cp (index=5)	isp (index=6)
app1	-1	1	-1	-1	-1	-1	-1
fw1	0	-1	-1	3	-1	-1	6
app2	-1	-1	-1	3	-1	-1	-1
fw2	-1	1	2	-1	-1	-1	6
dserver	-1	-1	-1	-1	-1	5	-1
cp	-1	-1	-1	-1	4	-1	6
isp	-1	1	-1	3	-1	5	-1

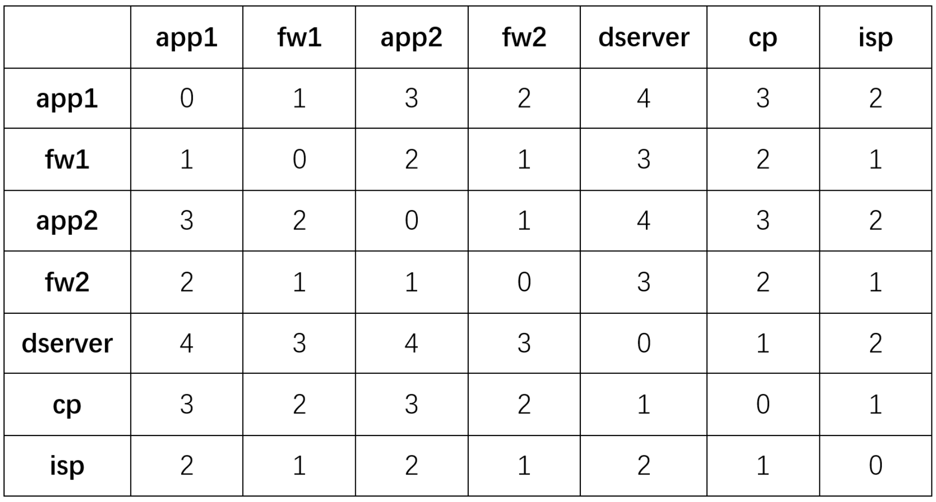
Figure 9: This is the initial forwarding table for all nodes, implemented using two-dimensional array in my Java code. The integer value refers to the index of the corresponding next hop. If not connected, the value should be -1 as default. Take [fw1][fw2]=3 as an example, this means in order to travel from fw1 to a destination of fw2, index 3(which is fw2 in this table) should be the next hop. To collect the initial table for fw1, the controller locate to the second row and collects the columns with index 0,3,6.

3.4.2 Updating Forwarding Tables

When the application is sending message to an unknown destination, the forwarder asks the controller. The controller takes in Consult packet including the forwarder ID and destination ID, and searches the shortest path between them using Dijkstra algorithm. The dijkstraAlgorithm() function takes in the index of the destination ID and updates the corresponding column in the initial overall forwarding table(see Figure 9). The controller then collects the grid of the forwarder ID from that column, which indicates the next hop.

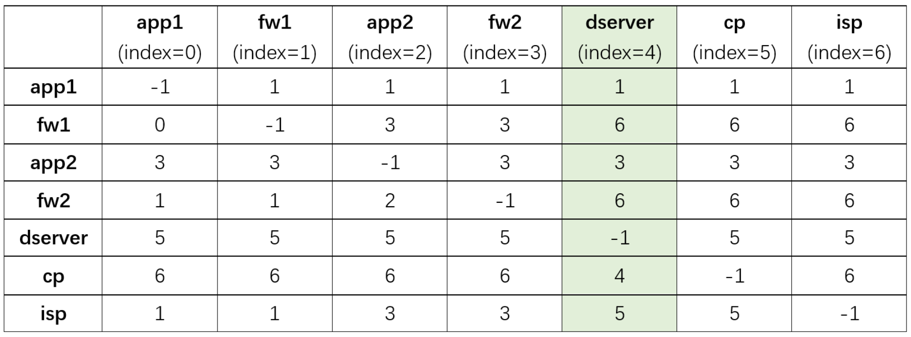
```
private void dijkstraAlgorithm(int des) {
    boolean[] shortest = new boolean[TABLE.length];
    shortest[des] = true;
    boolean exit = false;
    while(!exit) {
        int vertex = -1;
        for(int i = 0; i < DIST.length; i++) {
            if((!shortest[i]) && (DIST[i][des] != Integer.MAX_VALUE)){
                vertex = i;
                shortest[vertex] = true;
                for (int j = 0; j < DIST.length; j++) {
                    if (DIST[j][vertex] + DIST[vertex][des] < DIST[j][des]) {
                        DIST[j][des] = DIST[j][vertex] + DIST[vertex][des];
                        shortest[j] = false;
                        TABLE[j][des] = vertex;
                    }
                }
            }
        }
        if(vertex == -1) {
            exit = true;
        }
    }
}
```

Listing 5: Dijkstra algorithm Pseudo-code



	app1	fw1	app2	fw2	dserver	cp	isp
app1	0	1	3	2	4	3	2
fw1	1	0	2	1	3	2	1
app2	3	2	0	1	4	3	2
fw2	2	1	1	0	3	2	1
dserver	4	3	4	3	0	1	2
cp	3	2	3	2	1	0	1
isp	2	1	2	1	2	1	0

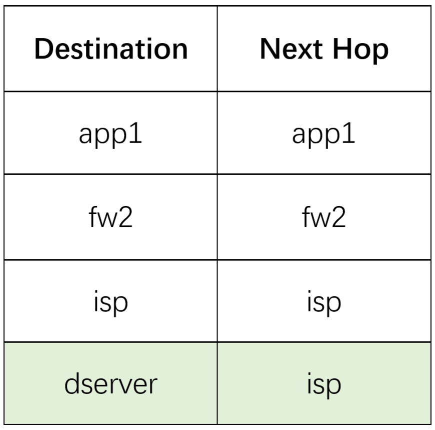
Figure 10: This is the Updated distance table for all nodes after running the dijkstraAlgorithm() function for each destination ID (Ideally, in fact the row with application IDs will never be updated in this program because they can never be the origin ID when consulting, it should always be the forwarders that come to consult).



	app1 (index=0)	fw1 (index=1)	app2 (index=2)	fw2 (index=3)	dserver (index=4)	cp (index=5)	isp (index=6)
app1	-1	1	1	1	1	1	1
fw1	0	-1	3	3	6	6	6
app2	3	3	-1	3	3	3	3
fw2	1	1	2	-1	6	6	6
dserver	5	5	5	5	-1	5	5
cp	6	6	6	6	4	-1	6
isp	1	1	3	3	5	5	-1

Figure 11: This is the updated forwarding table for all nodes after running the dijkstraAlgorithm() function for each destination ID (Ideally, in fact the row with application IDs will never be updated in this program because they can never be the origin ID when consulting, it should always be the forwarders that come to consult). Each column here refers to the next hop for every other node to reach it. Assume we have forwarder1 consulting for path from fw1 to dserver, in order to collect the updated table for a destination of dserver, the controller locate to the fifth column and collects the row with index 1. Hence, index 6(which is isp) should be the next hop for app1 to reach dserver.

The controller sends an update packet with the next-hop as a string message back to the forwarder. The forwarder then updates its own forwarding table with the new destination and corresponding next hop. Therefore, the Request packet can be successfully forwarded.



Destination	Next Hop
app1	app1
fw2	fw2
isp	isp
dserver	isp

Figure 12: This is the updated forwarding table for the forwarder 1, implemented using Hashmap in my Java code.

3.5 Sender Class

I have this Sender class to implement the Stop-and-Wait ARQ protocol using concurrency mechanisms. Since wait() method does not return anything, I cannot differentiate whether the wait() exits for being notified or reaching a timeout. Thus, I added a boolean variable that is set to True only when being notified. Below is my pseudo-code for Sender.

```
synchronized run(){
    int count = 0;
    while(count<=30) {
        try {
            sendPacket(packet);
            wait(WAIT_PERIOD);
            if(sent){
                return;
            }else{
                print("Packet␣sending␣failed.␣Will␣try␣again.");
                count++
            }
        } catch (Exception e) {e.printStackTrace();}
    }
    System.out.println("Time␣out,␣please␣try␣again.");
}

synchronized ackReceipt(){
        sent = true;
        notify();
    }
```

Listing 6: Sender Class Pseudo-code

4 Discussion

I have first implemented the forwarders using different class, such as Forwarder1, Forwarder2, ISP and CP. Then I realized that all forwarders have the same function. Thus, I combined all forwarder classes and took user inputs to define individual hostnames and client names. I could have more pairs of forwarders and employee applications here, but I kept to a minimum of two pairs considering the convenience.

5 Summary

This report outlines my attempt to design a forwarding flow decision system among nodes. The description of the implementation in this report highlights the key elements of my solution and shows how the theory is put into practice.

6 Reflection

At the beginning of this assignment, I found that my code from the last assignment was quite helpful. It served as a good starting point. I had practically finished the basic functionalities during the first phase, so I took my time with the controller part. I implemented the Dijkstra algorithm with my Algorithms assignment in the second year and modified according to my requirement. While debugging, I continued running into issues, but I eventually found a solution that I am pleased with. Since I have almost achieved the requirements of the whole assignment, I was allowed to spend more time and dig into some higher-level functionalities. Although some of the functionalities failed to be implemented at last, I learned plenty of programming skills from my research. The three phases of the assignment have given me an opportunity to achieve progresses step by step during each phase. I therefore think it is crucial to divide tasks into parts and deal with a small portion each time.
I spent roughly 27 hours total on the assignment, not including the time it took to produce the report.