Telecommunications II Assignment 2

Open Flow Protocol

DOCUMENTATION

Koh Li Hang | 18313798 | Computer Science Year 2

Table Of Content

Introduction

The aim of the second assignment in this module is to implement an Open Flow Protocol. The task is to build up this protocol on the controller-side and the router-side to send and receive the messages transfer among the clients. While doing this assignment the students will have a clearer image on the sockets, packets and also java net features.

Besides that, to implement this protocol, students will also need to have a degree of understanding on the find shortest path algorithm (For instance, I'm using Dijkstra Algorithm) to generate a path for the router map to pass over the data packet. This is also a good opportunity to understand how the network works in real life. After complete this assignment, it will help the students to enhance their impression on the data-link layer in telecommunications.

Open Flow Protocol

This protocol basically provided a way that helps the nodes forwarding the data packet between them. There will be a controller that will tell those switches or routers where should they send the packet to. And the controller is distinct from the routers, since they are having different function in this network, separating them will allows for more complex traffic management.

In my case of designing this protocol, my controller is using the Dijkstra Algorithm to generate paths for the clients and routers to transmit the data. Although there are still some problems on my work but at least it works for some cases.

Definition

OpenFlow is a type of protocol that enable a controller/a server to inform the network switches the next destination for them to send the data packet. For each switch, there will be a fully implemented function to parse the controller's command to tell them what to do. By using the OpenFlow protocol, the path decision for the packet will be centralized, which able the network to be programmed independently of the individual switches and the data center gear.

In a real-life network, packet forwarding decision (for the paths) and high-level routing (the control path) occur in the same device. OpenFlow protocol will separate them, which the packet forwarding decision will keep in the switches and the controlling part will be taken by a separate controller. And this method is also as known as a software-defined networking (SDN), which allows for a more efficient way of using the network resources compare with a traditional network.

Benefits

- Contains centralized controller
- More efficient compare with traditional protocol design
- Allows automated configuration of the network, less manual configuration
- Create a clearer image on the implemented network elements
- Provide end-to-end visibility path decision
- Increase the independence of the network by decreasing the use on network appliances like load balancers.

Limitations

- With a limited traffic engineering
- Might be difficult to use a centralized controller with fast convergence techniques like BFD (Bidirectional Forwarding Detection)
- Expensive to reinvent all the wheels when implementing a new OpenFlow protocol

Clients

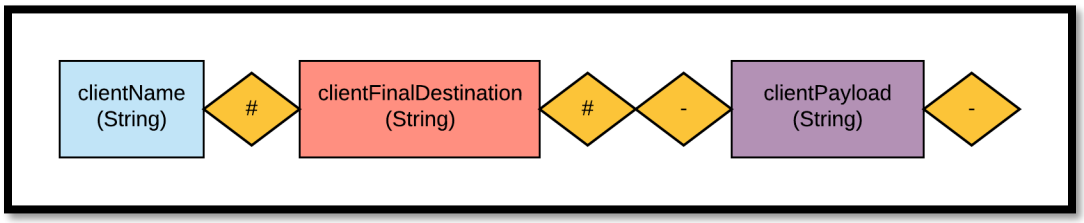
Basically, the Clients act as an interface for the users. There are 2 parts in this class which are the Sender part and the Receiver part.

The Sender part allows the user to enter whatever they want to send. After getting the input for the user, the Sender will format it into a data packet and send to the router which connected with itself. The Receiver part in Clients will parse the messages from the received data packet and print it out on the terminal.

By looking forward on the codes provided, there are also a senderListener class and a receiverListener class that allows both the Sender and the Receiver to keep running on threads. Therefore, the Clients will keep trying to receive a data transmission all the time.

Payload Data Packet

Since there's not much things that need to be contains in the client's data packet, a simple structure that can store the client's name, the destination, and the user's payload is enough for it. Therefore, I create a simple data packet structure for storing those details. Here is the image of the data packet structure.



Those symbols appeared in this figure are act as the delimiter character to help the receiver parse the messages out from the packet.

Pseudo Code & Explanation

toSend function Pseudo Code

```
toSend(){
    message = requestForInput();
    dst = requestForInput();
    packet = generatePacket(senderName,dst,message);
    socket.send(packet);
    print("Packet send to:"+packet.getSocketAddress());
}
```

toSend function Explanation

This function will be called constantly whenever the system starts to run. The senderListener will keep calling this function when the code is still running, so the users are allowing to send multiple times of the message to another client.

toReceive function Pseudo Code

```
toReceive(){
    recvSocket.receive(recvPacket);
    print("Received packet from:"+recvPacket.getAddress());
    print("Received payload:"+getPacketPayload());
}
```

toReceive function Explanation

Similar to the toSend function mentioned above, this function will keep running unless the program has been terminated. This mechanism avoids the Receiver to

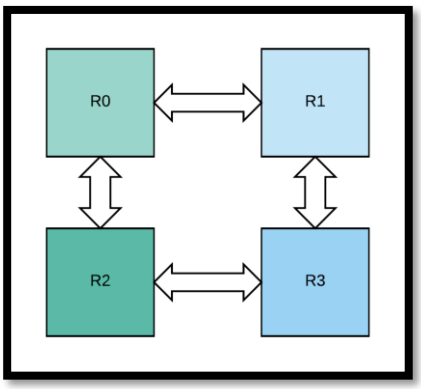
miss any transmission of the data packet. This is the function that will print the payload from the received packet.

Router

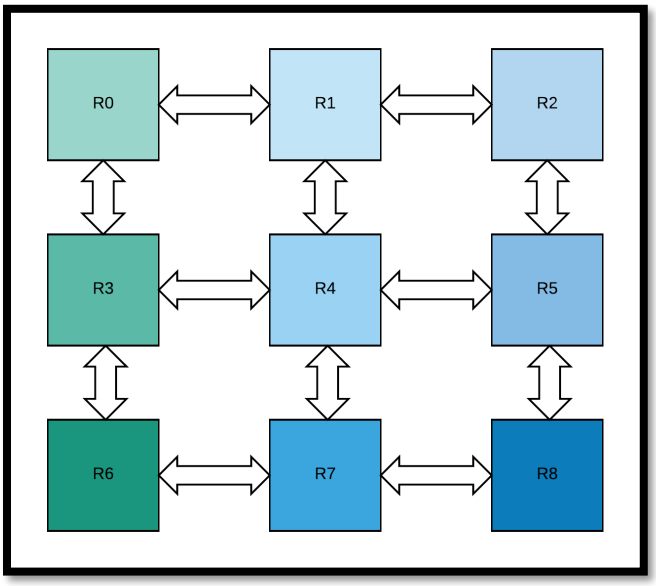
Routers are act as the switches in the Open Flow protocol, they are responsible to forward the data packets around the network map. But they will need to request a flow table from the controller to know where should the packet goes, since the decision for paths function is separated from the router.

In order to request the flow table, the router will need to forward its details to the controller for generating the flow table for each router. Besides that, the router will also need to handle the packets forwarding between the routers and the clients. In total, the router will need to require 3 different data packet structure to handle those transmissions. They are the Router-to-Controller [Details], the Router-to-Controller [Request for new paths], and the Router-to-Router/Client [Forwarding Packet].

Besides that, to understand how the router transmit the packet in the map, we will need to know the structure of the network in this protocol. In my design, the routers are connected as the same in these following diagrams.



2x2 Router map



3x3 Router map

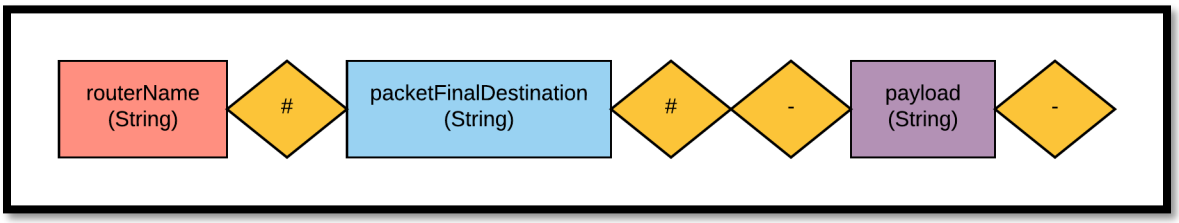
All routers are set to have 5 ports, they are the left connection, right connection, top connection, bottom connection and a port to contact the controller. From the map above, you there will be no connection for the routers are on the edge. Therefore, those ports are set to zero or it can be connected with

the client. In my design of the clients (3 clients) location will always be the top left, bottom left and the bottom right. To deal with those ports, each router will also contain the same amount of the sockets. So, there's a socket list with a size of 5 to deal with different ports.

From my codes, after the router has received the path from controller, it will forward to the next router in his path. But my controller won't send out the path to every router in the same time, so after the second router received the packet, it will still request for the path. Since the path will stored in the controller, the controller won't need to do the calculation again, it will just sending out the next destination for the routers until the packet has reached it's destination.

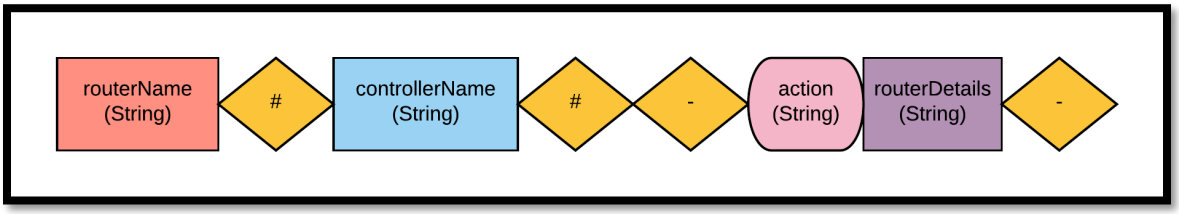
Router-to-Router/Client [Payload] Data Packet

The forwarding packet among the routers and clients is the same with the one with the client. It contains the router name, the final destination of the packet and the also the payload.



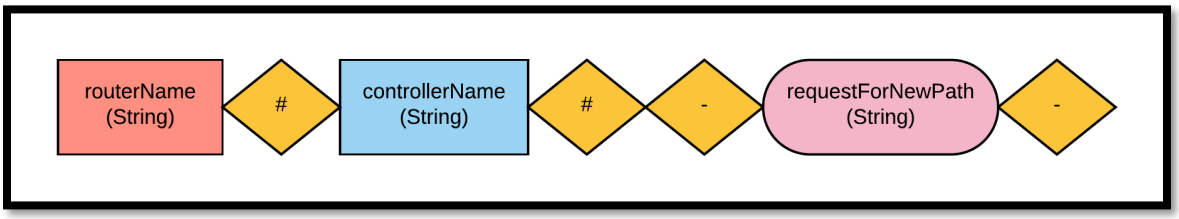
Router-to-Controller [Router Details] Data Packet

For the router's details packet, there are the router name, the controller name, the action and the router details. The router details are basically the pre-set flow table about the router which contains the matched destination for each port, the receiving port and the port to sent for that destination, and the time latency for the sockets. These information will help the controller to generate the path.



Router-to-Controller [Request for Paths] Data Packet

The path request data packet will be smaller than the packets mentioned above, since it only contains the router name, the controller name and a request for the new path.



Pseudo Code & Explanation

toSend function Pseudo Code

```
toSend(){
    if(needToRequestForNewPath)
    {
        payload = getPayload();
        packetDst = getPacketDst();
        CNTPacket = constructResquestPacket(packetDst);
        socketForController.send(CNTPacket);
        needToRequestForNewPath = false;
    }
}
```

toSend function Explanation

This function will only be called when the router has received a packet that is not stored on its destination table. It will get the payload and the packet destination from the received packet and construct the request packet and send it to the controller. After the packet is sent, disable this function.

toReceive function Pseudo Code

```
toReceive(){
    for(DatagramSocket socket : socketList)
    {
        socket.receive(packet);
        packetDst = getPacketDst();
        if(dstTableContainsDst(packetDst))
        {
            index = getIndexInDstTableRecords();
            sendPacketToNextDst(packetDst,payload,index,getPortToSendTo);
        }
        else if(isRequestPacket)
        {
            sendControllerRouterData();
        }
        else if(isNextDstPacket)
        {
            nextDst = parseNextDst();
            if(flowTableContains(nextDst)&& !dstTableContainsDst(packetDst))
            {
                index = getIndexInFlowTable();
                createDstTableRecord(nextDst);
                sendPacketToNextDst(packetDst,payload,index,getPortToSendTo);
            }
        }
        else{
            needToRequestForNewPath = true;
        }
    }
}
```

toReceive function Explanation

The toReceive function is slightly different with the others. Since there are a total of 5 sockets in the router, there will need a for-loop for this mechanism to run through every socket in the router.

If the socket received a packet, it will run into several statement to decide which action should be taken. If the destination table of the router has contained the packet destination, it will directly forward the packet to the next destination.

Else if the received packet is a request packet from the controller, it will call the sendControllerRouterData function to set up the packet of router details and send it to the controller.

Else if the received packet is the reply of the request path packet that the router sent before. It will be the packet that contains the next destination for the router. After checking the flow table contains the next destination, and it is not in the destination table of the router, create a new destination entry and store into the table. Finally, forward the packet to the next destination.

For the packet didn't agree with any statement, set the Boolean value needToRequestForNewPath to true and call the toSend function.

Controller

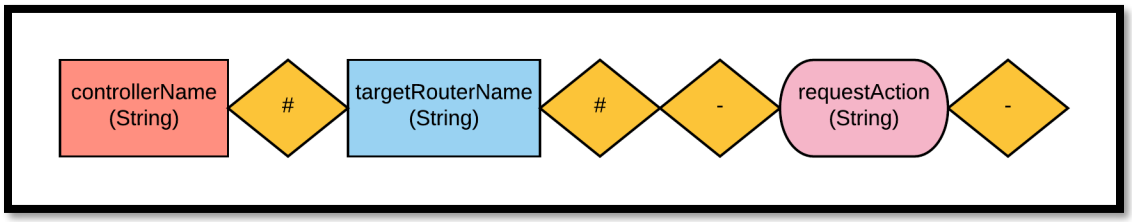
This centralized controller is separated from the routers and the clients, which there's no direct connections between the clients and the controller. But for every router, there's a port and a socket reserved for the controller.

Basically, the controller won't give any action before it has received a data packet from the routers. After received a data packet, the controller will try to generate a connection map of the whole network in the beginning. So, it will send a request for all routers in the network for their details. It will set up a flag when the connection map has been built up. After that, it will move to the next process which is check the path needed is available in the list or not. If it is not, generate a new path with the Dijkstra's Algorithm that I implement for this assignment. If the path is available in the list, then send back the respond (next destination) for the router.

The sophisticated part in this class is to set up all the needed elements for the Dijkstra's Algorithm to find the shortest path between the nodes. In order to do that, I will need to create a vertex, edge and a graph class. (And the edges will need to be weighted) While generating the connection map, the new vertices will be added into a list. After all of the vertices has been added, it will start another process which is to generate the edges with those added vertices. After those elements have been set up, then the Dijkstra's Algorithm will be ready to generate the path for the routers. And finally, send back the respond to the routers.

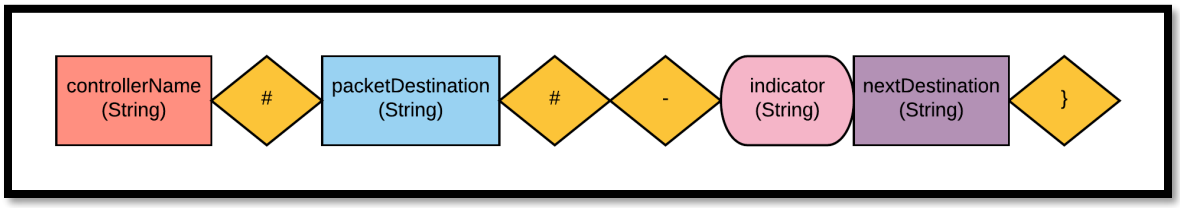
Router Details Request Data Packet

This data packet contains the controller name, the target router name, and the request action. As long as the router received this request data packet, it won't be too hard to distinguish what type of packet it has received by using the java contains() built-in method.



Router Next Destination Data Packet

This is the data packet that will inform the router the next destination to forward the data packet. It contains the controller name, the packet destination, an indicator for the next destination, and finally the next destination. There's a slightly change on the last delimiter symbol, it helps the router to extract the next destination in a more efficient way.



Pseudo Code & Explanation

toSend function is not available in this class

As my knowledge on the controller, there won't be a need for the toSend method. Since the controller is not keep asking data to transmit, it will only transmit the data while it received some request.

toReceive function Pseudo Code

```
toReceive(){
    socket.receive(packet);
    if(isPacketRequestForNewPath())
    {
        if(!connectionMapAvailable)
        {
            generateConnectionMap();
            connectionMapAvailable = true;
        }
        if(!pathAvailable())
        {
            newPath = generatePath(routerConnectionTable, packetSrc, packetDst);
            sortedPaths.put(packetDst, newPath);
        }
        if(pathAvailable())
        {
            sendNextDstToRouter();
        }
    }
}
```

toReceive function Explanation

This function is the main core of the Controller class, which help the controller to decide what to do and what to send. All of this began after the socket has received a packet from router. The only reason that the controller is receiving the packet is the router is trying to forward its data packet to the next destination. Therefore, after verified it is a request for a new path, the process run into a new statement.

First of all, it will check is the existence of connection map about this network, if there's none, start to generate one. While generating the connection map, the router will send the request for each router for there details to complete the map. There's where the controller sends the request packet.

Since the controller will stored every path generated in the process. Therefore, it will check the existence of the path for every specific route, if not,

then it will generate a new path for it. And this is where the Dijkstra's Algorithm is being called. After the path has generated, it is being stored in the sorted path hash map. The hash map helps the program to get the matched path for the router and destination easily.

When the path is available, the controller will simply construct the next destination packet and send it back to the router for further process.

Dijkstra's Algorithm

The aim of Dijkstra's Algorithm is to find the shortest path between 2 nodes in a graph. To start with, it will pick the unvisited adjacent nodes and calculate the distance though it to another unvisited adjacent node. It constantly updates the neighbor's distance if the distance is smaller. For those visited nodes will be store in another set to avoid the algorithm run through the same node again.

For my codes, after set up the vertices and edges, the calculation will take the source node and start to sort the unvisited nodes into visited nodes. The process won't end unless there's no unvisited nodes left in the graph. The nodes that comes from the unvisited nodes set will go through another method call seacrhCheapestPath. This is the method will constantly update the unvisited nodes list and the sortedPath hash map.

After the calculation has been done, the controller can simply call the generatePath function in Dijkstra Algorithm class to get the new path for the routers. Since the way the path sorted is in reverse order, the Collection.reverse() function will take care of it. Therefore, the path will be return in a form of ArrayList which might be easier for the controller to access.

Here are the Graph class, which followed by the Vertex class and the Edge class.

Graph Class

```
public class Graph {

    private ArrayList<Vertex> vertices;
    private ArrayList<Edge> edges;

    Graph (ArrayList<Vertex> vertices, ArrayList<Edge> edges) {
        this.vertices = vertices;
        this.edges = edges;
    }

    public ArrayList<Vertex> getVertices() {
        return vertices;
    }

    public ArrayList<Edge> getEdges() {
        return edges;
    }
}
```

Vertex Class

```java
public class Vertex {

    private String id;
    private String name;

    Vertex(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

}
```

Edge Class

```java
public class Edge {

    private String id;
    private Vertex sourceVertex;
    private Vertex destVertex;
    private int weight;

    Edge(String id, Vertex sourceVertex, Vertex destVertex, int weight) {
        this.id = id;
        this.sourceVertex = sourceVertex;
        this.destVertex = destVertex;
        this.weight = weight;
    }

    public String getId() {
        return id;
    }

    public Vertex getStartPoint() {
        return sourceVertex;
    }

    public Vertex getEndPoint() {
        return destVertex;
    }

    public int getWeight() {
        return weight;
    }
```

Pseudo Code & Explanation

runCalculation function Pseudo Code

```
runCalculation(srcNode, dstNode){
    setFinalDstNode(dstNode);
    unVisitedNodes.add(srcNode);
    distances.put(srcNode,0);
    while(!unVisitedNodes.isEmpty())
    {
        nextNode = getCheapest(unVisitedNodes);
        visitedNodes.add(nextNode);
        unVisitedNodes.remove(nextNode);
        searchCheapestPath(nextNode);
    }
}
```

runCalculation function Explanation

This function will take 2 vertices as parameters, after set up the final destination for the path, it will initialize the unvisited node list by putting the source node into the list and also set the distance from the source node to source node to zero.

It will keep sorting the path while the unvisited node list is not empty, there's a searchCheapestPath function inside the while-loop which is responsible to sort the shortest path.

searchCheapestpath function Pseudo Code

```
searchCheapestPath(aNode){
    neighbourNodeList = getNeighbours(aNode);
    for(tempNode : neighbourNodeList)
    {
        cmpD = getShortestDistance(aNode)+getDistance(aNode,tempNode);
        if(isRouter(tempNode))
        {
            if(getShortestDistance(tempNode) > cmp)
            {
                distances.put(tempNode,cmpD);
                unVisitedNodes.add(tempNode);
                unsortedPath.put(tempNode,aNode);
            }
        }
        else if(isDst(tempNode))
        {
            distances.put(tempNode,cmpD);
            unVisitedNodes.add(tempNode);
            unsortedPath.put(tempNode,aNode);
        }
    }
}
```

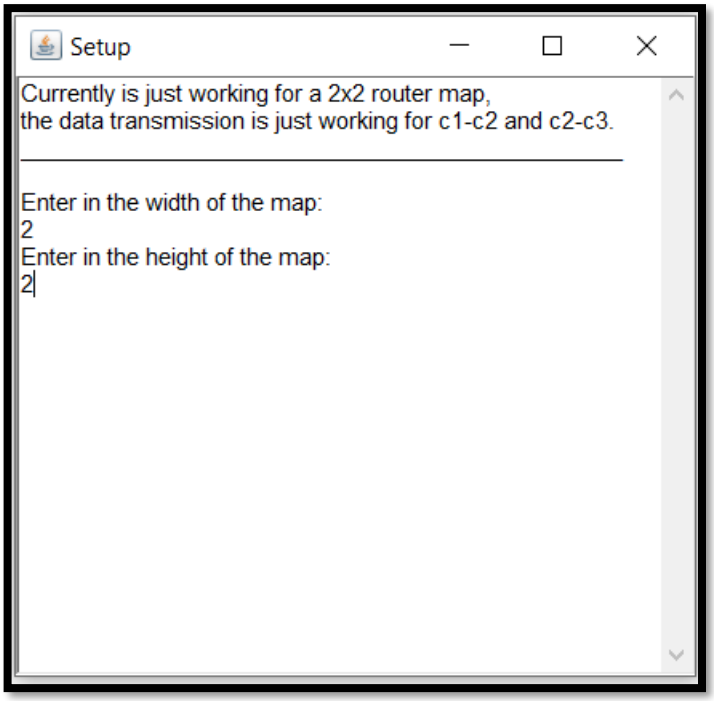
searchCheapestpath function Explanation

This function is the main core of the Dijkstra's Algorithm, which it generates the path here. In the beginning, this function will ask for the neighbor node list and pass it into a for-loop to get every neighbor node of the current node. Only 2 situations that will update the distances hash map and the unsorted path hash map, which are the neighbor node is a router or it is the destination client.

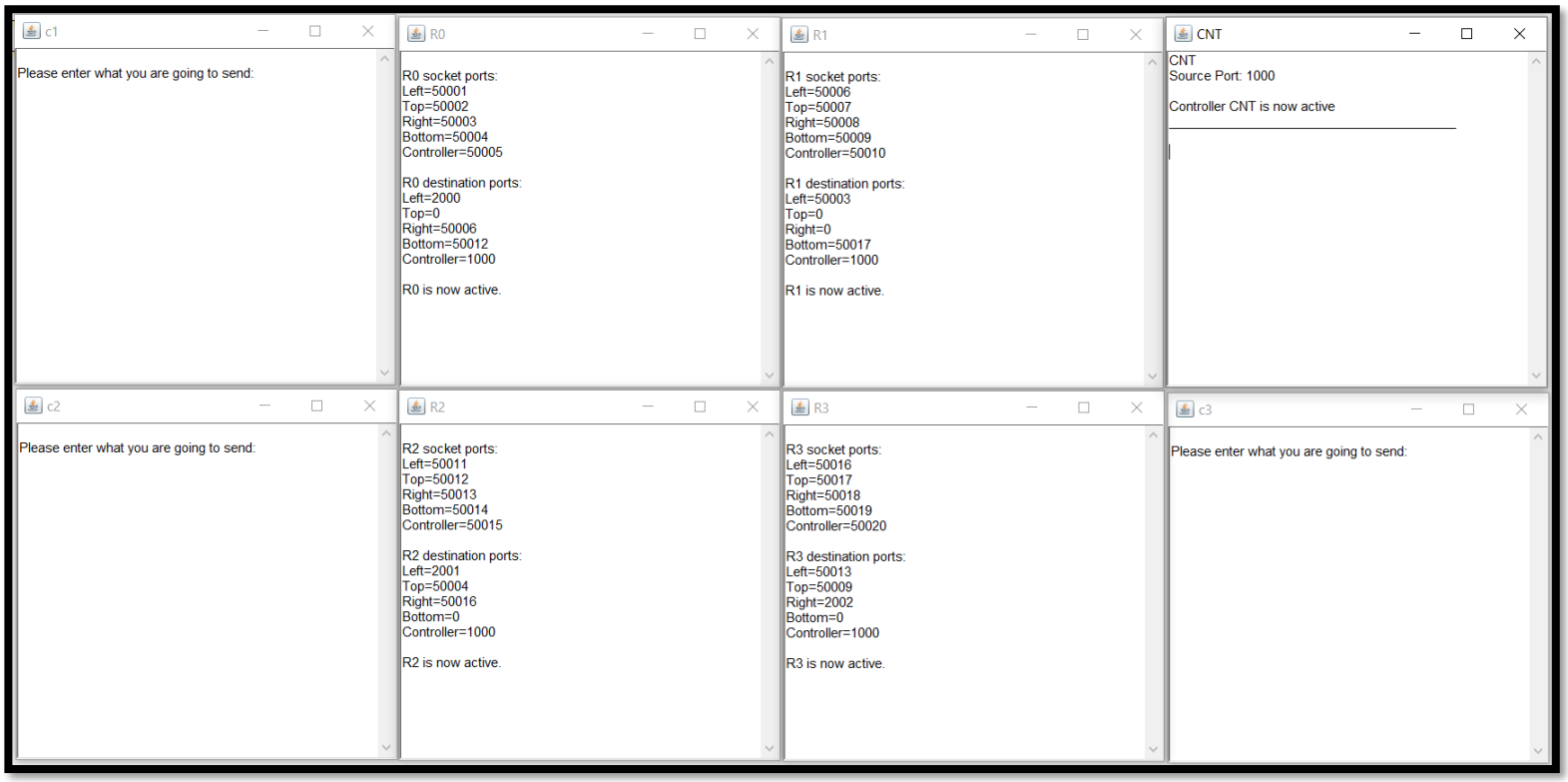
When it matches the statement, the program will update those hash map. But for the unsorted path, it will to reverse the order of the path to get the right order. Since it is storing in the reverse way.

Codes Operation Process Display

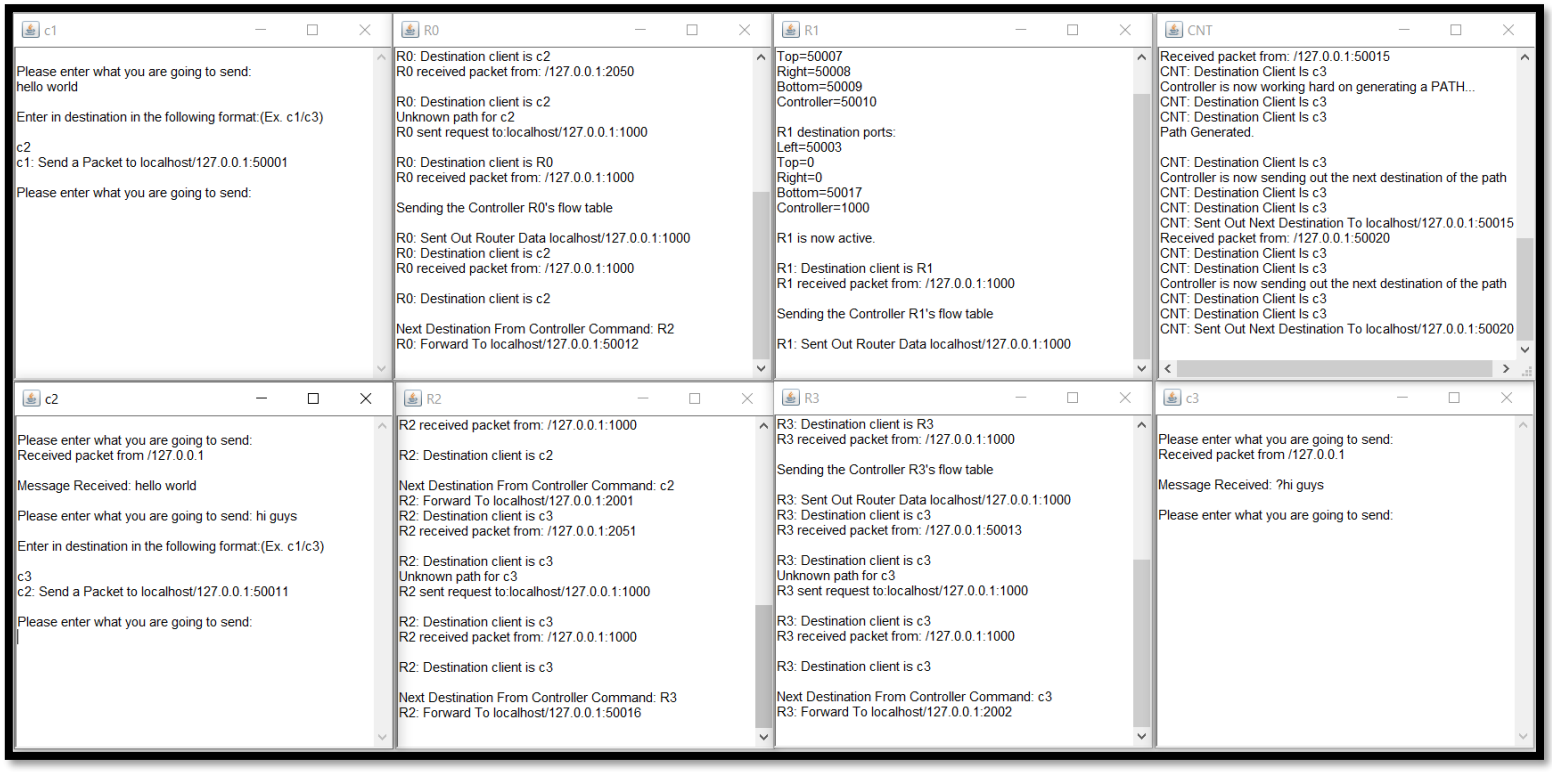
1) This is the setup terminal. As mentioned, currently my codes are just working for 2x2 router map. And must follow the order starting from c3-c2, then c2-c1. You can also do in c2-c3 or c1-c2 but the data packet wouldn't work for c1-c3 or c3-c1.



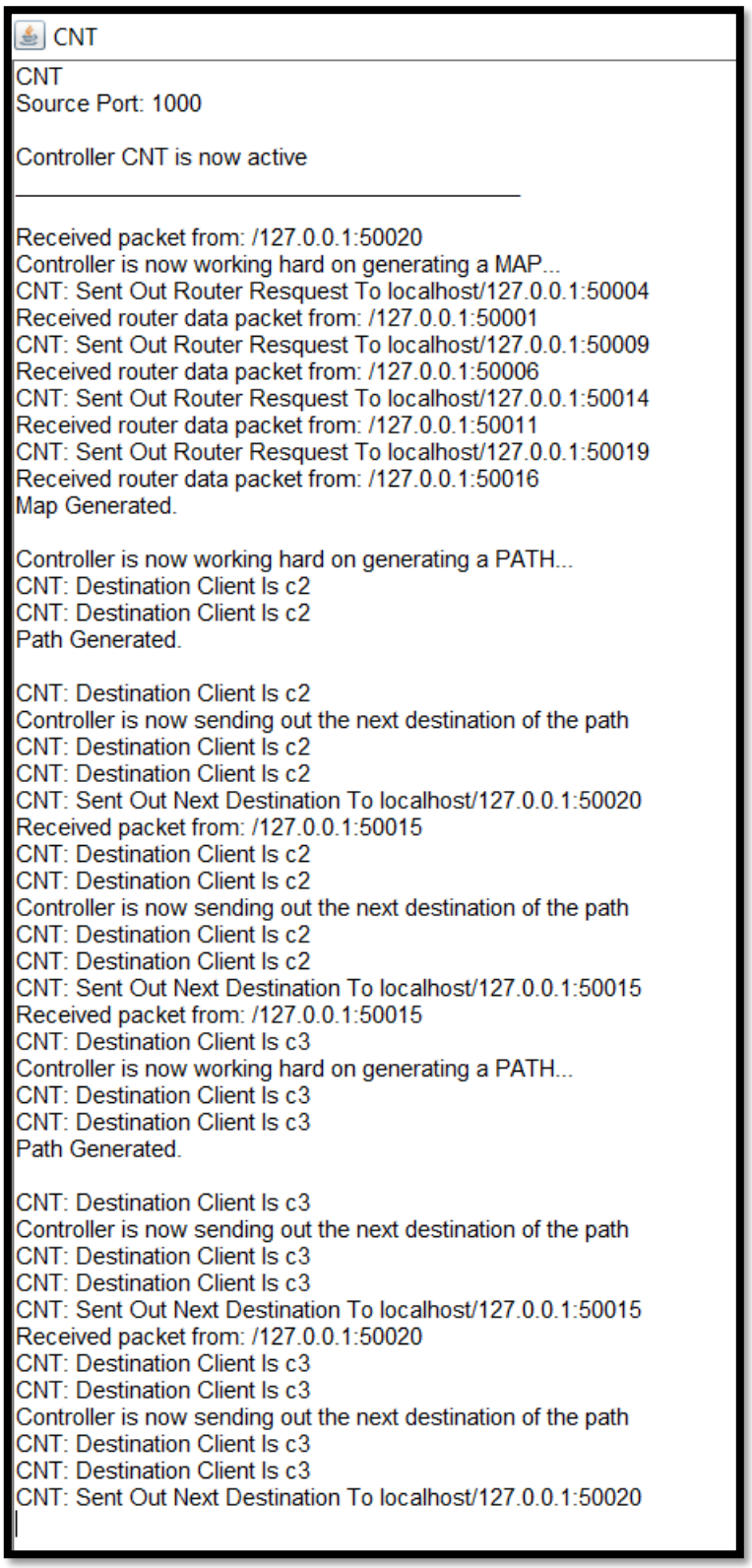
2) This is the starting interface for the routers and the clients. They are connected in the way shows in the screenshot. Except for the Controller.



3) You can see there are processes happening in the routers, and the clients have received their message. But somehow the message will add in a “?” symbol after several transmission.



5) This is what actually happening in the controller when the same clients are keep sending messages to each other.



Reflection

The most challenging part in this assignment is to implement a short path finding algorithm for the controller. Since we have enhanced our knowledge about the sockets, packets, and net features in java, therefore using those elements to implement a simple protocol is not a big problem. But implementing a path generating algorithm which fits your system? This is really tough.

I have spent a lot of time researching about the path sorting algorithms, and finally I pick Dijkstra's Algorithm for my algorithm part. Probably a wrong choice, since I will need to implement more on the vertices, edges and graph class. Although it took me a really long time to figure it out and tons of time to debug it, but I would say I have learned a lot from that.

To be honest, my program isn't working perfectly. There are still a lot of bugs while running the program. I tried to fixed most of them, so at least it is working is some specific condition. Optimistically, at least I got it working at last.

Estimation of Time Spent

I would say I have spent over 100 hours on it, it really took a long to understand, to research, to implement, and to debug.