Name: Haochen, Li NYU ID: hl4151 Date: 05/15/2022

Section: 001

Assignment #8 – Final Project

_____Part 3 functionality [Max 70 points]
_____Part 4 functionality [Max 25 points]
_____Style [Max 5 points]
_____Part 3 extra credit questions [Max 20 points (10 each)]

_____Total [Max 100 points + Extra Credit Points]

Total in points: _____

Total in extra credit points: _____

Professor's Comments:

Affirmation of my Independent Effort: Haochen Li

1. Introduction

Network layer is a layer both involved in OSI 7-layer model and TCP/IP model. This layer is in charge of provide point to point, best-effort data transmission service [1]. Network layer can be divided into 2 planes according to functions, which are control plane and data plane [1]. Control plane determines the best paths for packet transmission using routing tables generated by routing algorithms, while data plane forwards packets according to the path information in its routing table generated by control plane.

Traditionally, the duty of control plane is separated into every router in the entire network. This decentralized architecture makes routing path calculation expensive due to high time and communication complexities. Besides, traditional routing tables only contain network-layer forwarding information and are unprogrammable. Software Defined Networking (SDN) [2], an emerging network architecture where network control is decoupled from forwarding and is directly programmable, has emerged recently to address these challenges above. SDN centralized the functionality of control plane into a controller, which can use customized algorithm to realize complex business logics and generate flow tables corresponding to each device.

Flow tables can not only realize Network layer forwarding logics, but it can also be used to implement logics in other layers, e.g., switching, load balancing, firewall, etc. [3] Flow table uses “match + action” pattern to match header fields in an incoming packet, and then do some actions according to the corresponding header field values [3]. This makes network devices highly programmable, and a device can be configured with functionalities in multiple layers.

This work mainly focuses on realizing different functionalities in flow tables. In this project, SDN is used to realize both layer-3 (Network layer) routing and load balance routing. Layer-3 routing takes the network topology as input and generate forwarding rules for each switch in the network. Load balance routing listens to TCP SYN messages and insert appropriate address rewriting rules into a switch to realize address resolution.

The rest of this work is organized as follows. Section 2 gives the solutions to our 2 tasks. Section 3 gives the implementation of our solution, and section 4 shows the corresponding running results of our implementation. Finally, section 5 concludes this work.

2. Solutions & Algorithms

Below the solutions to layer-3 routing and load balance routing are given. The code implementation of the solutions will be shown in section 3 below.

2.1 Layer-3 Routing Solution

For layer-3 routing, this project first generates the topology of the network, and then implements a shortest path algorithm to calculate optimal routes between each pair of switches. With the shortest paths calculated, we can finally configure each switch with corresponding forward rules into its flow table. Algorithm 1 shows the entire work flow of layer-3 routing when network topology changes, e.g., new switch added, switch removed, new link added, etc.

Algorithm 1 L3Routing

Require: The link set in the network $Links = \{l_1, l_2, \dots, l_{2m}\}$;
The host set in the network $Hosts = \{h_1, h_2, \dots, h_n\}$
Ensure: Configure each switch in the network with its flow table;
1: $adjTable \leftarrow$ topoGeneration$(Links)$;
2: $ShortestPaths \leftarrow$ DijkstraAll$(adjTable)$;
3: **for** host h_1 **in** $Hosts$:
4: **for** host h_2 **in** $Hosts$:
5: ConfigureSwitches$(ShortestPaths, h_1, h_2)$;
6: **end for**
7: **end for**

Algorithm 2 below shows the generation of network topology topoGeneration() in this project. It is based on adjacency list [4] to store the directed graph topology of the network, which is an array of linked list head nodes storing adjacency relations.

Algorithm 2 topoGeneration

Input: The link set in the network $Links = \{l_1, l_2, \dots, l_n\}$;
Output: The adjacency graph of network topology $adjTable$;
1: Initialize $adjTable$;
2: **for** link l **in** $Links$:
3: Current source vertex $curKey \leftarrow l.src$;
4: Node $n \leftarrow adjTable[curKey]$;
5: **while** $n \neq null$:
6: $n \leftarrow n.next$;
7: **end while**
8: $n.next \leftarrow l.dst$;
9: **end for**
10: **return** $adjTable$;

After running Algorithm 2, the topology will be generated as an adjacency list. Figure 1 below shows the correspondence between a graph and its adjacency list.

Figure 1: A graph and its adjacency list.

This project calculates shortest path based on Dijkstra algorithm [5]. Algorithm 3 below shows the process of DijkstraAll(), which uses Dijkstra algorithm to calculate shortest paths between each pair of nodes in a graph.

Algorithm 3 DijkstraAll

Input: The adjacency table of the network $adjTable$;
Output: The shortest paths set $ShortestPaths$;

Initialize $ShortestPaths$;
$tableSize \leftarrow adjTable.keys().size()$;
for vertex v **in** $adjTable.keys()$:
    Initialize $nodeSet \leftarrow \{v\}$;
    Initialize distance $D[tableSize - 1][tableSize]$;
    Initialize previous nodes $P[tableSize - 1]$;
    for $nodes\ v' \neq v$ **in** $adjTable.keys()$:
        if v' **in** linked list $adjTable[v]$:
            $D[v'][0] \leftarrow 1$;
            $P[v'] \leftarrow v$;
        else
            $D[v'][0] \leftarrow \infty$;
            $P[v'] \leftarrow none$;
        end if
    end for
    $i \leftarrow 0$;
    while $nodeSet \neq adjTable.keys()$:
        Find v'that minimizes $D[v'][i]$;
        Add v' to $nodeSet$;
        for node v_{adj} **in** $adjTable[v']$:
            $D[v_{adj}][i + 1] \leftarrow \min(D[v_{adj}][i], D[v'][i] + 1)$;
            if $D[v_{adj}][i + 1] = D[v'][i] + 1$:

```
                    P[v_adj] ← v';
                end if
            end for
        end while
        for nodes v' ≠ v in adjTable.keys():
            Initialize stack PathStack and temp ← v';
            while P[temp] ≠ v:
                PathStack.push(P[temp]);
                temp ← P[temp];
            end while
            Pop elements in PathStack into ShortestPaths[v][v'];
        end for
    end for
    return ShortestPaths;
```

After that, ConfigureSwitches() is used for each pair of hosts to configure switches in the network with the calculated flow table. Algorithm 4 shows the details of this function.

Algorithm 4 ConfigureSwitches

Require: The shortest paths set $ShortestPaths$; 2 hosts h_1 *and* h_2;
Ensure: Configure switches between shortest path from h_1 *to* h_2 *with* corresponding flow table;

```
1:  PathSwitch ← h1.getSwitch();
2:  EndSwitch ← h2.getSwitch();
3:  for switch sw in ShortestPaths[PathSwitch][EndSwitch]:
4:      NextPort ← sw's port connecting the next switch in the path;
5:      Add rule ([h2.IP match] → [NextPort, Forward]) to sw;
6:  end for
```

Specially, when network topology is unchanged, e.g., adding / removing / moving hosts, only ConfigureSwitches() is called to update the flow tables. The topology and shortest paths will not be recalculated in order to avoid unnecessary costs.

When the workflow above is finished, a flow table is built in a switch in the network. This flow table contains the rules to match Network layer IP addresses and do forwarding actions to a specific port on that switch.

2.2 Load Balance Routing Solution

For load balance routing, things are a little different. Load balance routing should process both ARP requests for virtual MAC addresses in the Network layer and TCP SYN requests in the Transport layer, different rules should be added into another table from that shortest path table discussed in 2.1.

Several load balancers in this project correspond to a pre-assigned virtual IP address and virtual MAC addresses. This load balancer should implement rules for switches to forward TCP and ARP packets with virtual IP addresses as their network layer destination addresses to the SDN controller. This requires to implement rules to match ARP and ARP.TPA (Target iP Address) for ARP packets, and rules to match IPv4 and IP.Destination for TCP packets using IP as its network layer protocol; Other matching conditions indicates that the packet should be transmitted to other hosts rather than load balancer. Thus, another rule should also be added to look up the shortest path flow table discussed in 2.1. Algorithm 5 describes the switch configuration process.

Algorithm 5 ConfigureNewSwitch

Require: The virtual IP set $VirtualIPs$; a new switch sw;
Ensure: Configure flow table for sw in load balance routing;
for vIP **in** $VirtualIPs$:
    Add rule ($[dst = vIP, l3proto = ARP]$, $[toCtrl]$) into sw;
    Add rule ($[dst = vIP, l3proto = IP]$, $[toCtrl]$) into sw;
end for
Add rule ($[other\ match]$ → [to L3RoutingTable]) into sw;

After that, the controller can wait for the incoming ARP / TCP packets. For ARP packets, an ARP reply should be directly sent back to the requesting client; for TCP SYN, new rules to rewrite the virtual IP and virtual MAC for inbound and outbound TCP packets should be installed into the packet receiving switch; for other TCP packets, a TCP RESET packet should be sent back to the requesting client; other packets will just be ignored. Algorithm 6 describes the ARP / TCP handling process.

Algorithm 6 PacketProcessing

Require: An incoming Ethernet packet p in a switch sw;
Ensure: Different actions for different type of p;
if $p.type =$ ARP:
    Send an ARP reply from $(p.dstIP, p.dstMAC)$ to $(p.srcIP, p.srcMAC)$;
else if $p.type =$ TCP:
    if $p.flag =$ SYN:

5: Add rule([src: $(p.srcIP, p.srcMAC)$; dst: $(p.dstIP, p.dstMAC)$; TCP + IP] → [$(vIP, vMAC)$ → $(realIP, realMAC)$, to L3RoutingTable]) to sw;

6: Add rule([src: $(realIP, realMAC)$; dst: $(p.srcIP, p.srcMAC)$; TCP + IP] → [$(realIP, realMAC)$ → $(p.dstIP, p.dstMAC)$, to L3RoutingTable]) to sw;

7: **else**:

8: Send a TCP RESET from $(p.dstIP, p.dstMAC)$to $(p.srcIP, p.srcMAC)$;

9: **end if**

10: **end if**

Specially, the rules installed during TCP SYN processing are specially for this TCP connection, and have an idle timeout time of 20 seconds.

3. Implementation

In this section, the implementation of algorithms mentioned in 2 will be given. The implementation is based on Java, and some packages provided by FloodLight and OpenFlow are used to realize some logics.

3.1 Implementation of Layer-3 Routing

The implementation for topoGeneration is shown in Figure 2 below. The type Node defined here is consists of a switch number and a port number. The switch number indicates the switch corresponding to this node, and the port number is the port on the source switch (index of the adjacency list) to connect to this switch.

```
public void topoGeneration(Collection<Link> topoLinks) {
    adjacencyTable.clear();
    for(Link l : topoLinks) {
        long curKey = l.getSrc();
        if(!adjacencyTable.containsKey(curKey)) {
            adjacencyTable.put(curKey, new Node(l.getDst(), l.getSrcPort()));
        }
        else {
            Node temp = adjacencyTable.get(curKey);
            while(temp.next != null) {
                temp = temp.next;
            }
            temp.next = new Node(l.getDst(), l.getSrcPort());
        }
    }
}
```

Figure 2: The implementation of topoGeneration.

The implementation of DijkstraAll() are shown in Figures 3 to 5. In figure 3, the initialization of the algorithm is shown; in figure 4, the Dijkstra algorithm's logic is presented; in figure 5, the shortest path generation is given.

```
// First create network topology and initialize shortest path table;
topoGeneration(topoLinks);
shortestPaths.clear();
int tableSize = adjacencyTable.size();
for(long curSwitchID : adjacencyTable.keySet()) {

    // Initialize node set;
    nodeSet.clear();
    nodeSet.add(curSwitchID);

    // Initialize Dijkstra calculation table;
    distTable.clear();
    pathTable.clear(); // Node used as Pair<Long, int> here;
    for(long key : adjacencyTable.keySet()) {
        if(key != curSwitchID) {
            distTable.put(key, new int[tableSize]);
            distTable.get(key)[0] = 10000;
            pathTable.put(key, new Node(-1L, -1));
        }
    }
    Node temp = adjacencyTable.get(curSwitchID);
    while(temp != null) {
        if(distTable.keySet().contains(temp.dstID)) {
            distTable.get(temp.dstID)[0] = 1;
            pathTable.put(temp.dstID, new Node(curSwitchID, temp.srcPort));
        }
        temp = temp.next;
    }
```

Figure 3: Initialization in DijkstraAll();

```
// Dijkstra calculation below;
for(int i = 0; i < tableSize - 1; i ++) {
    int minDist = Integer.MAX_VALUE;
    long curAdded = -1L;
    // Find node that is not in noseSet and distance is minimum;
    for(long key : distTable.keySet()) {
        if(!nodeSet.contains(key)) {
            if(distTable.get(key)[i] < minDist) {
                minDist = distTable.get(key)[i];
                curAdded = key;
            }
        }
    }
    // Add to nodeSet;
    nodeSet.add(curAdded);

    // Set the next dists to prev dist;
    Node adjNode = adjacencyTable.get(curAdded);
    for(long key : distTable.keySet()) {
        if(!nodeSet.contains(key) && distTable.keySet().contains(key)) {
            distTable.get(key)[i + 1] = distTable.get(key)[i];
        }
    }
    // For each node adjacent to this newly added node, update their dist to source;
    while(adjNode != null) {
        if(!nodeSet.contains(adjNode.dstID)) {
            int newDist = distTable.get(curAdded)[i] + 1;
            if(distTable.get(adjNode.dstID) != null && newDist < distTable.get(adjNode.dstID)[i + 1]) {
                distTable.get(adjNode.dstID)[i + 1] = newDist;
                pathTable.put(adjNode.dstID, new Node(curAdded, adjNode.srcPort));
            }
        }
        adjNode = adjNode.next;
    }
}
```

Figure 4: Implementation of Dijkstra algorithm in DijkstraAll();

```
// Extract paths and add to the shortest path map;
shortestPaths.put(curSwitchID, new HashMap<Long, Node>());
pathStack.clear();
Map<Long, Node> curPaths = shortestPaths.get(curSwitchID);
for(long key : pathTable.keySet()) {
    Node curPos = pathTable.get(key);
    while(curPos != null && curPos.dstID != curSwitchID) {
        pathStack.push(curPos);
        curPos = pathTable.get(curPos.dstID);
    }
    if(curPos == null) {
        continue;
    }
    curPaths.put(key, new Node(curPos.dstID, curPos.srcPort));
    Node pathNode = curPaths.get(key);
    if(pathNode != null) {
        while(!pathStack.empty()) {
            pathNode.next = new Node(pathStack.peek().dstID, pathStack.peek().srcPort);
            pathStack.pop();
            pathNode = pathNode.next;
        }
    }
}
```

Figure 5: The shortest path generation in DijkstraAll();

The process in Figures 3 to 5 shows one single loop in DijkstraAll(). This whole process can generate the shortest paths from a switch *sw* to all the other switches. Looping for *sw* will generate shortest paths for any pair of switches in the network.

Besides, the Node structure is also used in the representation of shortest paths. The switch ID still indicates the current switch, but the port number in a shortest path node represents the port on current switch to the next node on the path.

Figure 6 shows the installation of forwarding rules for a specific host as destination. It obtains the shortest paths from the routing manager that run the DijkstrAll(). For each switch on the shortest path, its port can be directly obtained through its Node structure, just like the descriptions above. The matching conditions and actions of rules are realized using classes OFMatch and OFInstruction provided by OpenFlow.

The function shown in Figure 6 should be looped for all hosts to realize forwarding rule update for all hosts in the network. This is equal to running ConfigureSwitches() for any given pair of hosts.

For adding / removing / moving hosts, only the function in Figure 6 is looped for all existing hosts to update forwarding flow tables. For adding / removing switches and discovering new link, the topology computation & DijkstraAll() process is executed, and after that, the host forwarding table update will be executed. Specially, when network topology is unchanged, The topology computation & DijkstraAll() process will not be executed, but instead returned immediately to avoid unnecessary costs.

```
private void configureFlowTable(Host host) {
    if(host.getIPv4Address() == null) return;
    Map<Long, Map<Long, Node>> curMap = routingManager.getShortestPaths();
    Map<Long, IOFSwitch> switches = getSwitches();

    // Matching criteria for each switch;
    OFMatch newMatch = new OFMatch();
    newMatch.setDataLayerType(OFMatch.ETH_TYPE_IPV4);
    newMatch.setNetworkDestination(host.getIPv4Address());

    // Install rules for the directly-connected router;
    OFInstructionApplyActions newInstruction = new OFInstructionApplyActions();
    newInstruction.setActions(Arrays.asList((OFAction) new OFActionOutput(host.getPort())));
    IOFSwitch directSwitch = host.getSwitch();

    // Sometimes this switch can be null, indicating singular device;
    if(directSwitch != null) {
        SwitchCommands.removeRules(directSwitch, table, newMatch);
        SwitchCommands.installRule(directSwitch, table, SwitchCommands.DEFAULT_PRIORITY, newMatch, Arrays.asList((OFInstruction) newInstruction));

        // Update rules for switches on the forwarding path;
        for(long swId : curMap.keySet()) {
            if(swId == directSwitch.getId()) continue;
            Node curNode = curMap.get(swId).get(directSwitch.getId());
            while(curNode != null) {
                IOFSwitch curSw = switches.get(curNode.dstID);
                if(curSw != null) {
                    OFInstructionApplyActions curInstruction = new OFInstructionApplyActions();
                    curInstruction.setActions(Arrays.asList((OFAction) new OFActionOutput(curNode.srcPort)));
                    SwitchCommands.removeRules(curSw, table, newMatch);
                    SwitchCommands.installRule(curSw, table, SwitchCommands.DEFAULT_PRIORITY, newMatch, Arrays.asList((OFInstruction) curInstruction));
                }
                curNode = curNode.next;
            }
        }
    }
}
```

Figure 6: Implementation of adding forwarding rules to a switch;

3.2 Implementation of Load Balance Routing

The implementation of ConfigureNewSwitch() is shown in Figure 7 below. For both matching ARP packets and TCP packets, it uses data layer type (indicating the upper layer protocol) to match the protocol used in Network Layer, and uses current virtual Ips to match the destination address. A default rule with default priority is added for using layer-3 routing table to process other data packets. The priority of ARP and TCP matching rules are slightly higher than default. Thus, when a packet arrives, ARP and TCP matching will be done first. If unmatched, this packet will be handed to layer 3 routing to another host instead of handed to the controller.

The implementation of processing ARP requests in a switch is shown in Figure 8 below. We can observe that the reply sent back to the client has type ARP.OP_REPLY, and its destination and source addresses are reversed compared with the initial ARP request received.

Besides, the reply is constructed in-place. This means that the packet sent in is simply refactored into a reply packet, as well as the Ethernet frame received. This saves the time and storage overhead of processing incoming packets.

```
for(Integer virtualIP : instances.keySet()) {
    OFInstructionApplyActions newInst = new OFInstructionApplyActions();
    newInst.setActions(Arrays.asList((OFAction) new OFActionOutput().setPort(OFPort.OFPP_CONTROLLER.getValue())));
    List<OFInstruction> instList = Arrays.asList((OFInstruction) newInst);

    // ARP matching rule;
    OFMatch arpMatch = new OFMatch();
    arpMatch.setDataLayerType(OFMatch.ETH_TYPE_ARP);
    arpMatch.setField(OFOXMFieldType.ARP_TPA, virtualIP);

    // Install ARP rule to the new switch;
    SwitchCommands.removeRules(sw, table, arpMatch);
    SwitchCommands.installRule(sw, table, (short) (SwitchCommands.DEFAULT_PRIORITY + 1), arpMatch, instList);

    // TCP matching rule;
    OFMatch tcpMatch = new OFMatch();
    tcpMatch.setDataLayerType(OFMatch.ETH_TYPE_IPV4);
    tcpMatch.setNetworkDestination(virtualIP);

    // Install TCP rule to the new switch;
    SwitchCommands.removeRules(sw, table, tcpMatch);
    SwitchCommands.installRule(sw, table, (short) (SwitchCommands.DEFAULT_PRIORITY + 1), tcpMatch, instList);
}

// Installing rules for any other packets that needs to go to the next table;
OFInstructionGotoTable changeTableInst = new OFInstructionGotoTable(ShortestPathSwitching.table);
SwitchCommands.installRule(sw, table, SwitchCommands.DEFAULT_PRIORITY, new OFMatch(), Arrays.asList((OFInstruction) changeTableInst));
```

Figure 7: The implementation of ConfigureNewSwitch();

```
if(curPacketType == OFMatch.ETH_TYPE_ARP) {
    // Get the payload of current ethPkt as an ARP packet, and obtain its destination IP (virtual IP value);
    ARP arpContent = (ARP) ethPkt.getPayload();
    int destVirtualIP = IPv4.toIPv4Address(arpContent.getTargetProtocolAddress());
    if(arpContent.getOpCode() != ARP.OP_REQUEST || !(instances.keySet().contains(destVirtualIP))) {
        return Command.CONTINUE;
    }
    // log.info("Destination IP address: " + destVirtualIP);
    // Construct ARP reply;
    arpContent.setOpCode(ARP.OP_REPLY);
    // L2 address;
    arpContent.setTargetHardwareAddress(arpContent.getSenderHardwareAddress());
    arpContent.setSenderHardwareAddress(instances.get(destVirtualIP).getVirtualMAC());
    // L3 address;
    arpContent.setTargetProtocolAddress(arpContent.getSenderProtocolAddress());
    arpContent.setSenderProtocolAddress(destVirtualIP);

    // Encapsule this packet into a ethernet frame;
    ethPkt.setDestinationMACAddress(ethPkt.getSourceMACAddress());
    ethPkt.setSourceMACAddress(instances.get(destVirtualIP).getVirtualMAC());

    // Send this packet using SwitchCommands.sendPacket();
    SwitchCommands.sendPacket(sw, (short) pktIn.getInPort(), ethPkt);
}
```

Figure 8: The implementation of processing ARP requests;

Figures 9 shows the processing of TCP SYN packets. For TCP SYN packets, a temporary rule only for this TCP connection is generated, with an idle timeout time of 20 s. When this rule is idle (unused) for 20 seconds, then it will be automatically removed.

```
if(tcpContent.getFlags() == TCP_FLAG_SYN) {
    // Addresses;
    int clientIP = ipContent.getSourceAddress();
    short clientPort = tcpContent.getSourcePort();
    int curLoadBalancerHostIP = instances.get(destVirtualIP).getNextHostIP();
    short curLoadBalancerHostPort = tcpContent.getDestinationPort();
    byte[] curLoadBalancerHostMAC = getHostMACAddress(curLoadBalancerHostIP);
    byte[] destVirtualMAC = instances.get(destVirtualIP).getVirtualMAC();

    // Create match for inbound / outbound rules of a host;
    // Inbound: src is the client, and dst is the virtual IP + true port;
    OFMatch inMatch = new OFMatch();
    inMatch.setDataLayerType(OFMatch.ETH_TYPE_IPV4);
    inMatch.setNetworkProtocol(IPv4.PROTOCOL_TCP);
    inMatch.setNetworkSource(clientIP);
    inMatch.setTransportSource(clientPort);
    inMatch.setNetworkDestination(destVirtualIP);
    inMatch.setTransportDestination(curLoadBalancerHostPort);
    // Install inbound rule using idle timeout;
    // Inbound action is to rewrite destination IP and MAC from virtual to actual;
    OFInstructionApplyActions newInst = new OFInstructionApplyActions();
    OFInstructionGotoTable changeTableInst = new OFInstructionGotoTable(ShortestPathSwitching.table);
    OFActionSetField setMACAction = new OFActionSetField();
    OFActionSetField setIPAction = new OFActionSetField();
    setMACAction.setField(new OFOXMField(OFOXMFieldType.ETH_DST, curLoadBalancerHostMAC));
    setIPAction.setField(new OFOXMField(OFOXMFieldType.IPV4_DST, curLoadBalancerHostIP));
    newInst.setActions(Arrays.asList((OFAction) setMACAction, (OFAction) setIPAction));
    List<OFInstruction> instList = Arrays.asList(newInst, changeTableInst);
    SwitchCommands.installRule(sw, table, SwitchCommands.MAX_PRIORITY, inMatch, instList, HARD_TIMEOUT, IDLE_TIMEOUT);
```

```
// Outbound: src is the real host IP + true port, dst is the client;
// Outbound action is to rewrite source IP and MAC from actual to virtual;
OFMatch outMatch = new OFMatch();
outMatch.setDataLayerType(OFMatch.ETH_TYPE_IPV4);
outMatch.setNetworkProtocol(IPv4.PROTOCOL_TCP);
outMatch.setNetworkSource(curLoadBalancerHostIP);
outMatch.setTransportSource(curLoadBalancerHostPort);
outMatch.setNetworkDestination(clientIP);
outMatch.setTransportDestination(clientPort);
// Install outbound rule using idle timeout;
newInst = new OFInstructionApplyActions();
changeTableInst = new OFInstructionGotoTable(ShortestPathSwitching.table);
setMACAction = new OFActionSetField();
setIPAction = new OFActionSetField();
setMACAction.setField(new OFOXMField(OFOXMFieldType.ETH_SRC, destVirtualMAC));
setIPAction.setField(new OFOXMField(OFOXMFieldType.IPV4_SRC, destVirtualIP));
newInst.setActions(Arrays.asList((OFAction) setMACAction, (OFAction) setIPAction));
instList = Arrays.asList(newInst, changeTableInst);
SwitchCommands.installRule(sw, table, SwitchCommands.MAX_PRIORITY, outMatch, instList, HARD_TIMEOUT, IDLE_TIMEOUT);
```

Figure 9: The implementation of processing TCP SYN;
Upper: address determination & inbound rule generation; Lower: outbound rule generation;

Both inbound and outbound rules use full information for matching this requires TCP protocol on Transport layer, IP protocol on Network layer, and the source and destination IP & port should all match. The action is to rewrite between virtual and real IP / MAC addresses. For inbound packets going into this switch, its destination IP and MAC addresses (originally virtual) will be rewritten; for outbound packets going back to clients, its source IP and MAC addresses (originally real) will be turned into virtual.

Figure 10 shows the implementation of processing non-SYN TCP packets. A TCP RESET packet is simply sent back to the client.

```
// Otherwise, send TCP RESET back to client;
else {
    // Addresses;
    int clientIP = ipContent.getSourceAddress();
    short clientPort = tcpContent.getSourcePort();
    short curLoadBalancerHostPort = tcpContent.getDestinationPort();

    // Construct TCP RESET reply;
    TCP tcpReplyContent = new TCP();
    tcpReplyContent.setFlags(TCP_FLAG_RST);
    tcpReplyContent.setSourcePort(curLoadBalancerHostPort);

    // Construct IPv4 packet;
    IPv4 ipReplyContent = new IPv4();
    ipReplyContent.setProtocol(IPv4.PROTOCOL_TCP);
    ipReplyContent.setSourceAddress(destVirtualIP);
    ipReplyContent.setDestinationAddress(clientIP);
    ipReplyContent.setPayload(tcpReplyContent);

    // Construct Ethernet packet to encapsule IPv4;
    Ethernet replyCarrier = new Ethernet();
    replyCarrier.setEtherType(Ethernet.TYPE_IPv4);
    replyCarrier.setSourceMACAddress(instances.get(destVirtualIP).getVirtualMAC());
    replyCarrier.setDestinationMACAddress(ethPkt.getSourceMACAddress());
    replyCarrier.setPayload(ipReplyContent);

    // Send this packet using SwitchCommands.sendPacket();
    SwitchCommands.sendPacket(sw, (short) pktIn.getInPort(), replyCarrier);

}
```

Figure 10: The implementation of processing non-SYN TCP packets.

4. Results

4.1 Experimental Environment

The code implementation in this project is all compiled and run on ubuntu 14.04 LTS Linux system lying on Oracle VirtualBox virtual machine. The machine uses 4 GB main storage and 1 CPU. The JDK on this machine is OpenJDK Java 7, and the Python version on this machine is 2.7.

4.2 Results of Layer-3 Routing

The implementation for layer 3 is compiled and run with FloodLight, a Java-based SDN controller. After running the compiled FloodLight with applications JAR file, I started mininet, a simple network simulator, with network topology Assign1. The topology of Assign1 is shown in Figure 11 below, and we can observe that it has 6 switches and 10 hosts.

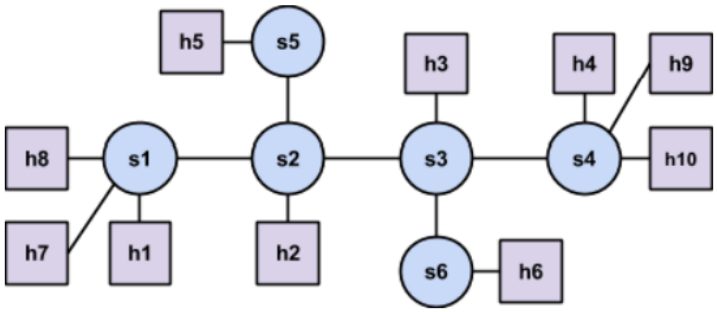

Figure 11: The network topology of Assign1.

After running Assign1 on FloodLight, the console running FloodLight will hint that switches, hosts, and links are all added, shown in Figure 12 below.

```
0:00:00:00:00:02 outPort=3, dst=00:00:00:00:00:00:00:03, inPort=2]
06:34:21.633 WARN [n.f.l.i.L.s.notification:New I/O server worker #2-1] Link updated: Link [src=00:00:00:00:00:00:00:02
outPort=3, dst=00:00:00:00:00:00:00:03, inPort=2]
06:34:21.634 WARN [n.f.l.i.L.s.notification:New I/O server worker #2-2] Link added: Link [src=00:00:00:00:00:00:00:04 ou
tPort=4, dst=00:00:00:00:00:00:00:03, inPort=3]
06:34:21.639 INFO [ShortestPathSwitching:Topology Updates] Link s3:3 -> 4:4 updated
06:34:21.734 INFO [ShortestPathSwitching:Topology Updates] Link s3:2 -> 2:3 updated
06:34:21.738 INFO [ShortestPathSwitching:Topology Updates] Link s2:3 -> 3:2 updated
06:34:21.738 INFO [ShortestPathSwitching:Topology Updates] Link s2:3 -> 3:2 updated
06:34:21.738 INFO [ShortestPathSwitching:Topology Updates] Link s4:4 -> 3:3 updated
06:34:22.060 INFO [n.f.d.i.Device:Scheduled-0] updateAttachmentPoint: ap [AttachmentPoint [sw=3, port=4, activeSince=165
2697261558, lastSeen=1652697261558]]  newmap null
06:34:22.063 INFO [n.f.d.i.Device:Scheduled-0] updateAttachmentPoint: ap [AttachmentPoint [sw=2, port=3, activeSince=165
2697260702, lastSeen=1652697261039]]  newmap null
06:34:22.071 INFO [n.f.d.i.Device:Scheduled-0] updateAttachmentPoint: ap [AttachmentPoint [sw=6, port=2, activeSince=165
2697260846, lastSeen=1652697261776]]  newmap null
06:34:22.373 INFO [ShortestPathSwitching:New I/O server worker #2-2] Host h5 added
06:34:23.438 INFO [ShortestPathSwitching:New I/O server worker #2-1] Host h6 added
06:34:24.595 INFO [ShortestPathSwitching:New I/O server worker #2-2] Host h7 added
06:34:25.587 INFO [ShortestPathSwitching:New I/O server worker #2-2] Host h8 added
06:34:26.626 INFO [ShortestPathSwitching:New I/O server worker #2-2] Host h9 added
06:34:27.655 INFO [ShortestPathSwitching:New I/O server worker #2-2] Host h10 added
```

Figure 12: The console running FloodLight after starting mininet with Assign1 topology;

Then, I tried to run command pingall in mininet. This command let each host in the network ping all the other hosts to test connectivity between any host pair. The result of pingall is shown in Figure 13 below, indicating that the forwarding rules in L3Routing's flow tables are successfully installed into all switches. Similar results can also be observed with network configurations of single, tree, linear, mesh, and someloops.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Results: 0% dropped (90/90 received)
mininet>
```

Figure 13: The result of running pingall in mininet with Assign1;

4.3 Results of Load Balance Routing

The load balance routing is also compiled and run with FloodLight. This time the network topology is simply single, 3, which is a single switch with 3 host connected. First, I started FloodLight with applications JAR file and mininet with “single, 3” configuration, and then I executed curl command on host h1. The result returned by h1 is shown in Figure 14, and the result in FloodLight console is shown in Figure 15.

```
mininet> h1 curl 10.0.100.1
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    23  100    23    0     0     22      0  0:00:01  0:00:01 --:--:--    22
WEB PAGE SERVED BY: h2
mininet>
```

Figure 14: Curl result in mininet;

```
06:23:43.026 INFO [ArpServer:New I/O server worker #2-2] Received ARP request for 10.0.100.1 from 00:00:00:00:00:01
06:23:43.046 INFO [ShortestPathSwitching:New I/O server worker #2-2] Host h1 added
06:23:44.018 INFO [ArpServer:New I/O server worker #2-2] Received ARP request for 10.0.0.1 from 00:00:00:00:00:02
06:23:44.018 INFO [ArpServer:New I/O server worker #2-2] Sending ARP reply 10.0.0.1->00:00:00:00:00:01
```

Figure 15: Curl result in FloodLight;

The results above shows that the rules and workflows to forward ARP request to the controller and send back response are correctly functioning.

5. Conclusion

In this project, Software Defined Network (SDN) is used to realize both Network layer routing and load balance routing, which has shown the programmability of SDN to configure a device into any layers.

The Network layer routing uses Dijkstra algorithm on the adjacency list of the network topology to generate shortest paths between arbitrary host pairs, and rules for forwarding packets with destination host’s IP address are installed in switches within the corresponding shortest paths. The load balance routing forwards ARP and TCP packets with virtual IPs as destination addresses to the controller, and then process ARP request, TCP SYN and non-SYN TCP packets differently. The experiment tests the implementation of the 2 tasks with a simulated network. The results of the experiment can show the correctness of the work flow of this project.

References

[1] J. Kurose and K. Ross, Computer Networking: A Top-down Approach. Pearson, 2020. [Online]. Available: https://books.google.com.hk/books?id=ms3fygEACA

[2] Palo Alto, Software-defined networking: The new norm for networks, CA, USA, White Paper, Apr. 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/white-papers/wp-sdnnewnorm.pdf

[3] Xia W, Wen Y, Foh C H, et al. A survey on software-defined networking[J]. IEEE Communications Surveys & Tutorials, 2014, 17(1): 27-51.

[4] Singh H, Sharma R. Role of adjacency matrix & adjacency list in graph theory[J]. International Journal of Computers & Technology, 2012, 3(1): 179-183.

[5] M. Noto and H. Sato, A method for the shortest path search by extended Dijkstra algorithm, Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' no.0, 2000, pp. 2316-2320 vol.3