Computer Networks COL334/672

Assignment 3

An Introduction to the World of SDN

Submitted by

Name	Entry Number
Arindam Sal	2024JCS2041
Ayan Shil	2024JCS2048

Part 1: Report for Controller Hub and Learning Switch

Introduction

This experiment compares the performance of a Hub Controller and a Learning Switch using Ryu with Mininet. The Hub Controller redirects all traffic to itself and forwards it to all switch ports except the incoming one, while the Learning Switch installs flow rules on switches based on MAC to port mappings learned from incoming traffic.

Experimental Setup

Environment

- **Controller:** Ryu running on Windows
- **Network Simulator:** Mininet running on Ubuntu within VirtualBox
- **Python Version:** Python 3.8 on Windows (Ryu), Python 3.12 on Ubuntu (Mininet)
- **Topology Used:** P1 Topology with two switches and five hosts.

P1 Topology

- Two switches (s1 and s2).
- Five hosts: h1, h2, h3 connected to s1; h4, h5 connected to s2.
- A link between the two switches (s1 to s2).

Controllers

- **Learning Switch:** Implements flow rules dynamically based on incoming traffic.
- **Hub Controller:** Redirects al traffic on a switch to itself and forwards it to all other ports except the incoming port.

Commands Executed

Starting the Virtual Environment:

Windows (Ryu Controller)

ryu-env\Scripts\activate

ryu-manager learning_switch.py

Linux (Mininet Environment)

source ~/Desktop/a3/mininet/mininet-venv/bin/activate # Start Mininet virtual environment

Running the Topology

Command to Start P1 Topology in Mininet

sudo ~/Desktop/a3/mininet/mininet-venv/bin/python3 p1_topo.py # Run the P1 topology script

Ping Test to Check Connectivity

Command

mininet> pingall # Test connectivity among all hosts

```
(mininet-venv) arindam-sal@arindam-sal-VirtualBox:~/Desktop/a3/mininet/topologies$ sudo ~/D
esktop/a3/mininet/mininet-venv/bin/python3 p1_topo.py
[sudo] password for arindam-sal:
*** Creating network
*** Adding hosts:
h1 h2 h3 h4 h5
*** Adding switches:
s1 s2
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s2) (h5, s2) (s1, s2)
*** Configuring hosts
h1 h2 h3 h4 h5
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ...
*** Running CLI
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5
h2 -> h1 h3 h4 h5
h3 -> h1 h2 h4 h5
h4 -> h1 h2 h3 h5
h5 -> h1 h2 h3 h4
*** Results: 0% dropped (20/20 received)
mininet>
```

1.1. Result for Learning Switch

```
(mininet-venv) arindam-sal@arindam-sal-VirtualBox:~/Desktop/a3/mininet/topologies$ sudo ~/Desktop/a3/mininet/mininet-venv/bin/python3 p1_topo.py
*** Creating network
*** Adding hosts:
h1 h2 h3 h4 h5
*** Adding switches:
s1 s2
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s2) (h5, s2) (s1, s2)
*** Configuring hosts
h1 h2 h3 h4 h5
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ...
*** Running CLI
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5
h2 -> h1 h3 h4 h5
h3 -> h1 h2 h4 h5
h4 -> h1 h2 h3 h5
h5 -> h1 h2 h3 h4
*** Results: 0% dropped (20/20 received)
```

1.2. Result for Hub Controller

Bandwidth Test Using Iperef

Commands Used

mininet> h5 iperf -s & # Start iperf server on h5

mininet> h1 iperf -c h5 # Run iperf client on h1 to check throughput to h5

```
mininet> h5 iperf -s &
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
mininet> h1 iperf -c h5
------------------------------------------------------------
Client connecting to 10.0.0.5, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 34044 connected with 10.0.0.5 port 5001 (icwnd/mss/irtt=14/1448/319)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0052 sec  35.2 GBytes  30.2 Gbits/sec
mininet>
```

1.3. Result for Learning Switch

```
mininet> h5 iperf -s &
mininet> h1 iperf -c h5
------------------------------------------------------------
Client connecting to 10.0.0.5, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 46014 connected with 10.0.0.5 port 5001 (icwnd/mss/irtt=14/1448/294)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0022 sec  29.5 GBytes  25.3 Gbits/sec
```

1.4. Result for Hub Controller

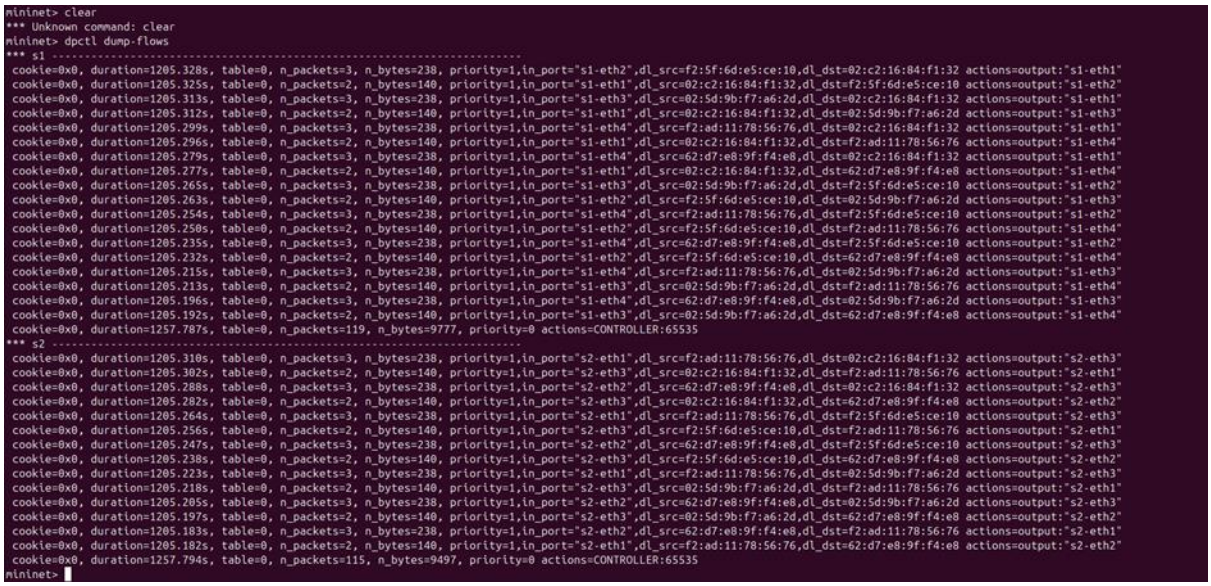
Checking Flow Rules Installed on Switches:

Commands Used:

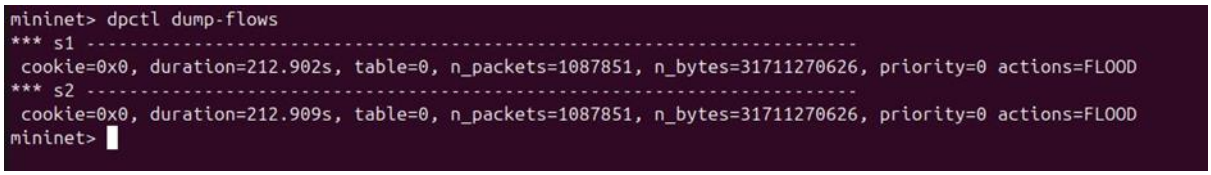
mininet> dpctl dump-flows # Check flow rules installed on all switches

mininet> sh ovs-ofctl dump-flows s1 # View specific flow rules on switch s1

mininet> sh ovs-ofctl dump-flows s2 # View specific flow rules on switch s2



1.5. Result for Learning Switch



1.6. Result for Hub Controller

Results and Observations

Ping Test (Connectivity Check)

Learning Switch

- Command: pingall
- Result: 0% packet loss (20/20 packets received), indicating full connectivity among all hosts.
- **Screenshot Evidence**: Shows that all hosts (h1 to h5) can communicate with each other without packet drops.

Hub Controller

- Command: pingall
- Result: 0% packet loss (20/20 packets received), showing similar full connectivity.
- **Comparison**: Both controllers demonstrate reliable connectivity, but the Hub Controller may lead to more network congestion due to its flooding nature.

Flow Rules Analysis

Learning Switch:

- Flow rules are dynamically created based on source and destination MAC addresses, directing traffic specifically rather than broadcasting.
- Observations: Flow rules are specific, reducing unnecessary traffic and optimizing network performance.
- Extracted Results: Flow rules show match conditions on source and destination addresses with actions specifying particular output ports.

Hub Controller:

- Flow rules are simplistic with actions set to FLOOD, meaning traffic is broadcasted across all ports except the incoming port.
- Extracted Results: Broadcasting causes a high packet count (n_packets), which can increase unnecessary traffic and load on the network.

Bandwidth Test (Throughput Measurement)

Learning Switch

- Command: h1 iperf -c h5
- Result: Transfer rate of 35.2 GBytes at 30.2 Gbits/sec.

Hub Controller

- Command: h1 iperf -c h5
- Result: Transfer rate of 29.5 GBytes at 25.3 Gbits/sec.
- **Comparison:** The Learning Switch shows higher throughput due to efficient flow rules that minimize network congestion.

Analysis

- The Learning Switch is more efficient in managing traffic due to its specific flow rules based on observed traffic patterns. This approach minimizes congestion and maximizes throughput.
- The Hub Controller, while simple, leads to a higher volume of unnecessary traffic due to its flooding nature, which can degrade overall performance.
- The Learning Switch is better suited for environments where performance and resource optimization are critical.

Conclusion

The Learning Switch demonstrates superior performance in both connectivity and throughput compared to the Hub Controller. This performance gain is due to the dynamic and targeted flow rule installation, which reduces broadcast traffic and optimizes data flow. The results reinforce the importance of intelligent flow management in SDN environments, highlighting the inefficiencies of basic hub-like behaviour in more complex network scenarios.

Part 2: Report on Spanning Tree Implementation

Introduction and Objective

The controller must:

- Construct a spanning tree given a network topology.
- Forward broadcast packets only to the ports in the spanning tree and to directly attached hosts.

The network consists of four switches (S1, S2, S3, S4) and four hosts (H1, H2, H3, H4), each connected to one switch. The switches are interconnected to form a cycle, which creates potential broadcast storms if packets are continuously forwarded in loops.

The controller's spanning tree protocol (STP) ensures that the loops are broken and a loop-free topology is created dynamically.

```
class CustomTopo(Topo):
    """Custom topology with 4 switches forming a cycle and 1 host per switch."""
    def build(self):
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')
        s4 = self.addSwitch('s4')


        # Create 4 hosts
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        h3 = self.addHost('h3')
        h4 = self.addHost('h4')

        # Connect each host to its respective switch
        self.addLink(h1, s1)
        self.addLink(h2, s2)
        self.addLink(h3, s3)
        self.addLink(h4, s4)

        # Connect switches to form a cycle
        self.addLink(s1, s2)
        self.addLink(s2, s3)
        self.addLink(s3, s4)
        self.addLink(s4, s1)
```

2.1 The topology used(Although it'll work with any topology)

Controller Logic to Handle Spanning Tree

The Ryu controller provided implements the logic of STP (Spanning Tree Protocol) using the ryu.lib.stplib library. The STP prevents broadcast loops by designating specific ports as blocked and others as forwarding based on the spanning tree algorithm.

```
class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'stplib': stplib.Stp}

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.topo_structure = TopoStructure()
        self.stp = kwargs['stplib']
        config = {dpid_lib.str_to_dpid('0000000000000001'):
                      {'bridge': {'priority': 0x8000}},
                  dpid_lib.str_to_dpid('0000000000000002'):
                      {'bridge': {'priority': 0x9000}},
                  dpid_lib.str_to_dpid('0000000000000003'):
                      {'bridge': {'priority': 0xa000}}}
        self.stp.set_config(config)
```

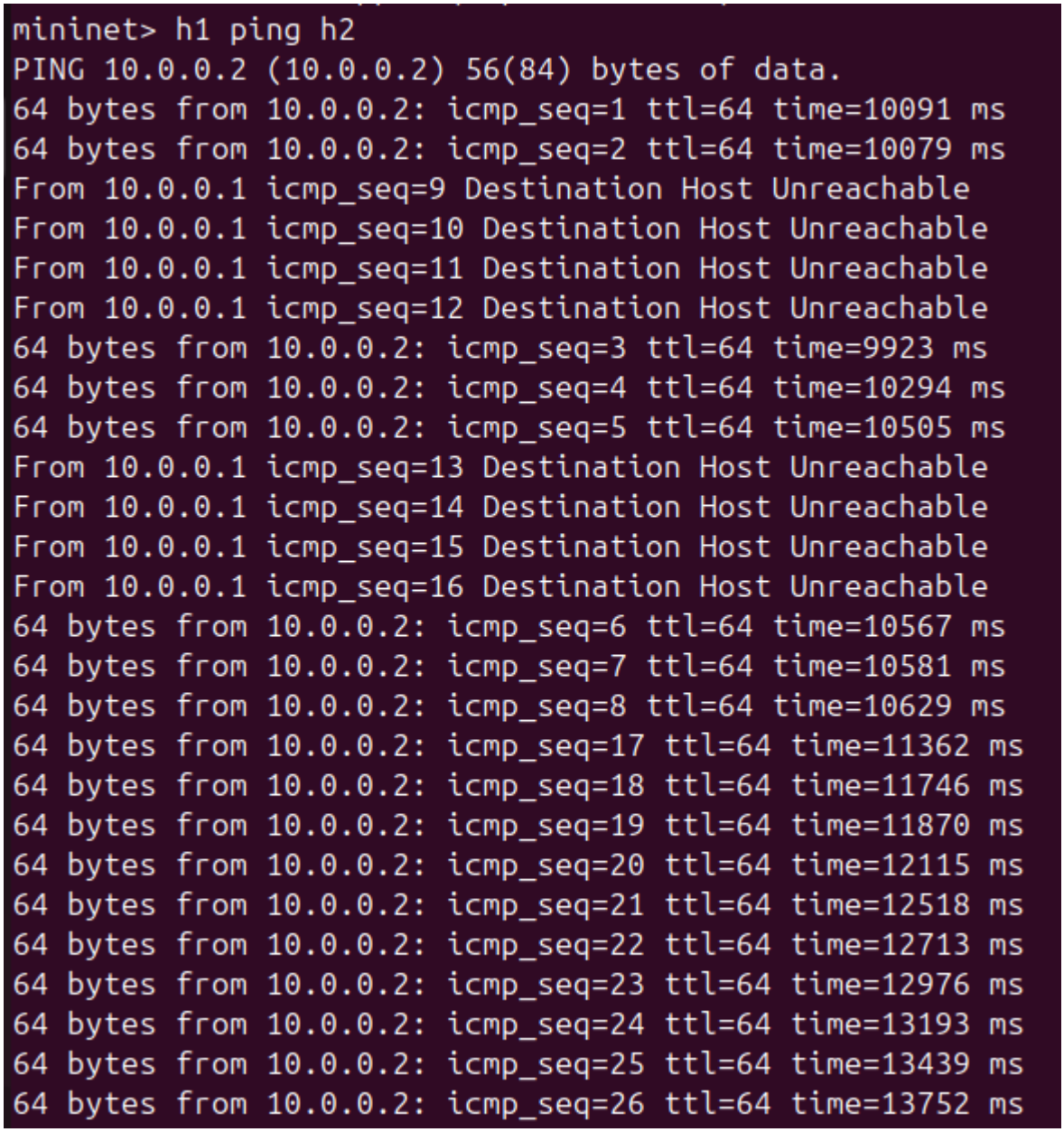
2.2 Key portions of the controller

The stplib.Stp context in the controller ensures that the STP is appropriately set up across the switches. The logic behind the spanning tree is based on setting bridge priorities and ensuring that one root switch is selected while the other switches block some ports to break cycles.

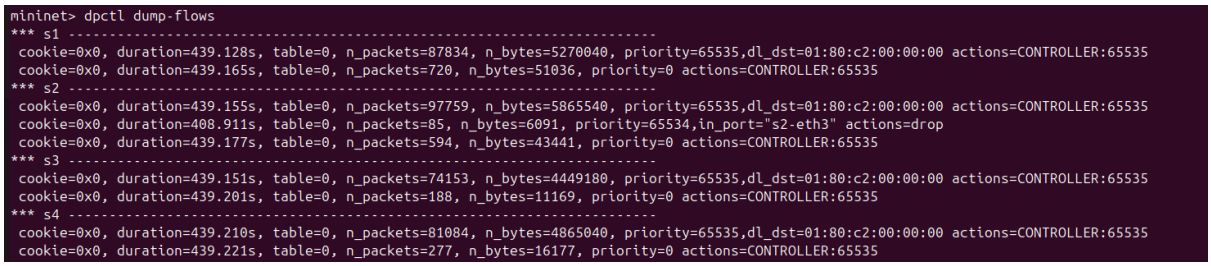
Results on Mininet

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X X
h4 -> X X X
*** Results: 100% dropped (0/12 received)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 X X
h2 -> h1 X X
h3 -> X X X
h4 -> X X h3
*** Results: 75% dropped (3/12 received)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet>
```

2.3 ‘pingall’ Result on Mininet



2.4 “p1 ping h2” result on mininet



2.5 “dpctl dump-flows” result on Mininet

Why Three Pingall Calls?

As seen in the test outputs, it takes three pingall commands for the network to stabilize and fully converge to a spanning tree. The first time pingall is executed, all packets are dropped due to the initial setup phase of the spanning tree algorithm. The second run shows partial packet success, as the STP has begun blocking specific ports. By the third run, the STP had fully converged, and all 12 packets were successfully transmitted, as seen in the 0% packet loss.

This delay is expected because:

- The spanning tree algorithm takes a few iterations to stabilize the network and designate root bridges and forwarding/blocking ports.
- During the initial phases, the switches still process broadcast packets, leading to packet loss.

Key Code Segments in Spanning Tree Logic

STP Configuration

Each switch in the network has a priority, and the STP designates the switch with the lowest priority as the root bridge. The root bridge forwards packets, while other switches may block specific ports to prevent loops.

```
config = {dpid_lib.str_to_dpid('0000000000000001'):
              {'bridge': {'priority': 0x8000}},
          dpid_lib.str_to_dpid('0000000000000002'):
              {'bridge': {'priority': 0x9000}},
          dpid_lib.str_to_dpid('0000000000000003'):
              {'bridge': {'priority': 0xa000}}}
self.stp.set_config(config)
```

2.6 STP Configuration

Learning MAC Addresses

The SimpleSwitch13 class also learns MAC addresses and installs flow entries to avoid flooding packets unnecessarily.

```
self.mac_to_port[dpid][src] = in_port
if dst in self.mac_to_port[dpid]:
    out_port = self.mac_to_port[dpid][dst]
else:
    out_port = datapath.ofproto.OFPP_FLOOD
```

2.7 Learning MAC Addresses

Approach and Assumptions

- **Spanning Tree Protocol**: We assume that STP is enabled across all switches and automatically selects a root bridge. The Ryu controller dynamically builds the spanning tree based on bridge priorities and network events (such, port up/down).
- **MAC Address Learning**: The controller learns MAC addresses and installs flow, which helps reduce unnecessary broadcast traffic after the initial packets.
- **Handling Loops**: The STP algorithm ensures that loops are blocked. Only after the STP has converged can packets be successfully forwarded across the network without any loops.

Conclusion

The network setup required three iterations of pingall for the spanning tree to converge fully. This behavior is typical when STP is in use because it takes time for switches to communicate, agree on a root bridge, and block the necessary ports. Once stabilized, the network operates as expected, with no packet loss and proper packet forwarding. The controller successfully avoids loops using STP and dynamically updates flow entries based on learned MAC addresses.

Part 3: Shortest Path Routing

Introduction:

In this assignment, we implemented a controller app that applies **shortest path routing** while maintaining L2 switching functionalities. The aim was to configure link weights based on link delay to optimize the routing of packets across the network topology. The task was built on a network topology provided in the assignment, with different delays added to each link.

Solution Overview

We divided the implementation into two essential parts:

- **Controller Logic:** The Ryu-based controller app uses the NetworkX library to compute the shortest path and manage ARP requests.
- **Mininet Topology:** The custom topology implemented in Mininet defines four switches connected in a cycle with different link delays.

Controller Logic

The controller is split into two major modules:

- **ShortestPath.py**: Responsible for calculating the shortest path and handling packet forwarding.
- **ArpHandler.py**: Responsible for managing ARP requests and tracking host locations.

ShortestPath.py:

This file handles the core routing logic, leveraging the Dijkstra algorithm provided by the NetworkX library to compute the shortest path. It installs forwarding rules in the switches based on link delays. A few key methods:

switch_features_handler:

- This function initializes switches and adds the table-miss flow entry, ensuring packets are sent to the controller if no matching flow is found.

```python
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    msg = ev.msg
    dpid = datapath.id
    self.datapaths[dpid] = datapath
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                      ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)
```

3.1 Switch_feature_handler

arp_forwarding:

- Handles ARP requests by sending them to the destination host if the host is in the access table, otherwise flooding the request.

```python
def arp_forwarding(self, msg, src_ip, dst_ip):
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    result = self.arp_handler.get_host_location(dst_ip)
    if result:  # host record in access table.
        datapath_dst, out_port = result[0], result[1]
        datapath = self.datapaths[datapath_dst]
        out = self._build_packet_out(datapath, ofproto.OFP_NO_BUFFER,
                                     ofproto.OFPP_CONTROLLER,
                                     out_port, msg.data)
        datapath.send_msg(out)
    else:
        self.flood(msg)
```

3.2 arp_forwarding

shortest_forwarding:

- The function that handles the calculation of the shortest path between switches. It uses NetworkX to compute the shortest path based on the link delay and installs flow entries accordingly.

```python
def shortest_forwarding(self, msg, eth_type, ip_src, ip_dst):
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    result = self.get_sw(datapath.id, in_port, ip_src, ip_dst)
    if result:
        src_sw, dst_sw, to_dst_port = result[0], result[1], result[2]
        if dst_sw:
            to_dst_match = parser.OFPMatch(
                eth_type = eth_type, ipv4_dst = ip_dst)
            port_no = self.arp_handler.set_shortest_path(ip_src, ip_dst, src_sw, dst_sw, to_dst_port, to_dst_match)
            self.send_packet_out(datapath, msg.buffer_id, in_port, port_no, msg.data)
    return
```

3.3 shortest_forwarding

ArpHandler.py:

get_topology:

- Discovers the network topology, including switches and links, and calculates link delays. This information is essential for computing the shortest path.

```python
def get_topology(self, ev):
    switch_list = get_all_switch(self)
    self.create_port_map(switch_list)
    self.switches = self.switch_port_table.keys()
    links = get_link(self.topology_api_app, None)
    self.create_interior_links(links)
    self.create_access_ports()
    self.get_graph()
```

3.4 get_topology

Mininet Topology

In the provided Mininet topology, we created four switches and hosts, connected cyclically, with different bandwidth and delay configurations. The following code snippet shows the essential part of the topology creation:

```
class CustomTopo(Topo):
    def build(self):
        # Create four switches
        s1 = self.addSwitch('s1', protocols='OpenFlow13')
        s2 = self.addSwitch('s2', protocols='OpenFlow13')
        s3 = self.addSwitch('s3', protocols='OpenFlow13')
        s4 = self.addSwitch('s4', protocols='OpenFlow13')

        # Create four hosts, one for each switch
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        h3 = self.addHost('h3')
        h4 = self.addHost('h4')

        # Add links between hosts and their respective switches
        self.addLink(h1, s1)
        self.addLink(h2, s2)
        self.addLink(h3, s3)
        self.addLink(h4, s4)

        # Add links between switches in a cycle with 20 Mbps bandwidth
        self.addLink(s1, s2, bw=2, delay='20ms',cls=TCLink)
        self.addLink(s2, s3, bw=2, delay='10ms', cls=TCLink)
        self.addLink(s3, s4, bw=1, delay='20ms', cls=TCLink)
        self.addLink(s4, s1, bw=1, delay='15ms', cls=TCLink)
```

3.5 Mininet Topology

Testing and Results:

As demonstrated in the Mininet output (pingall command result), after running the topology and the controller code, the hosts successfully communicate with each other via the shortest path, calculated based on the link delays.

Ping Output:

- The result shows that after three attempts (pingall), all packets were successfully delivered between hosts, verifying that the shortest path was computed and applied correctly.

```
arindam-sal@arindam-sal-VirtualBox:~/Desktop/a3/mininet$ source ~/mininet-venv/bin/activate
(mininet-venv) arindam-sal@arindam-sal-VirtualBox:~/Desktop/a3/mininet$ sudo python3 p3_topo.py
[sudo] password for arindam-sal:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

3.6 “pingall” Result

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=41.1 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=41.5 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=59.7 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.8 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=64.3 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=61.3 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=52.0 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=49.3 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=78.2 ms
```

3.7 “ping” Result

Assumptions Made

- **Static Topology**: The network topology is static, meaning the link delays remain constant throughout the simulation.
- **Equal Bandwidth Consideration**: While the links have different delays, bandwidth was not a critical factor in calculating the shortest path (as the focus was on delay-based shortest path routing).
- **Switch-to-Switch Communication**: All switches are assumed to have stable connections without any packet loss at the control layer.

Conclusion

In this assignment, we successfully implemented shortest-path routing using the NetworkX library. The controller calculates the shortest path based on link delays and installs flow rules accordingly in the switches. The Mininet topology demonstrated proper routing as all hosts could communicate through the shortest available path.

The modular structure of the code allows it to be scalable and adaptable to more complex topologies with varying delay or other metrics for shortest path calculation.

Part 4: Report on Congestion-aware Shortest Path Routing

Introduction

This report presents the implementation of a congestion-aware shortest path routing algorithm in a Software-Defined Networking (SDN) environment using Ryu as the controller and Mininet to simulate the network topology. The primary objective is to dynamically route traffic based on real-time link utilization, as opposed to static link weights, allowing for better traffic management and congestion avoidance in the network. This functionality was developed as an extension of a simple shortest path algorithm, adapted to monitor link loads and modify routing decisions accordingly.

Assumptions and Routing Level

In the implementation, several assumptions were made to simplify the complexity of the congestion-aware routing:

- **Routing Level**: The routing is performed at **flow-level** rather than packet-level. Each flow between two hosts is dynamically assigned the least congested path based on the current link utilization.
- **Link Bandwidth**: It is assumed that each link in the network has an initial bandwidth capacity of **10 Gbps** (or 10000000 Kbps). Bandwidth consumption on each link is continuously monitored to account for traffic congestion.
- **Link Congestion Metric**: Congestion on a link is defined by the link's utilization, calculated by monitoring the difference in tx_bytes over time. This value is updated every second using OpenFlow port statistics.

Implementation Logic

Topology Setup

The network topology was created using Mininet, consisting of several switches and hosts. The key Mininet topology script sets up a controller and defines the links between switches. Each link is initialized with a default bandwidth capacity. Here’s a snippet of the topology setup:

```
# Hosts
h1 = net.addHost('h1', ip='10.0.0.1/24', position='10,10,0')
h2 = net.addHost('h2', ip='10.0.0.2/24', position='20,10,0')

# Switches
switch1 = net.addSwitch('switch1', protocols=protocolName, position='12,10,0')
switch2 = net.addSwitch('switch2', protocols=protocolName, position='15,20,0')
switch3 = net.addSwitch('switch3', protocols=protocolName, position='18,10,0')
switch4 = net.addSwitch('switch4', protocols=protocolName, position='14,10,0')
switch5 = net.addSwitch('switch5', protocols=protocolName, position='16,10,0')
switch6 = net.addSwitch('switch6', protocols=protocolName, position='14,0,0')
switch7 = net.addSwitch('switch7', protocols=protocolName, position='16,0,0')
switch8 = net.addSwitch('switch8', protocols=protocolName, position='16,0,2')
switch9 = net.addSwitch('switch9', protocols=protocolName, position='16,0,3')
```

4.1 Topology

```
info("*** Adding the Link\n")
net.addLink(h1, switch1)
net.addLink(switch1, switch2)
net.addLink(switch1, switch4)
net.addLink(switch1, switch6)
net.addLink(switch2, switch3)
net.addLink(switch4, switch5)
net.addLink(switch5, switch3)
net.addLink(switch6, switch7)
net.addLink(switch7, switch3)
net.addLink(switch7, switch9)
net.addLink(switch8, switch5)
net.addLink(switch3, h2)
```

4.2 Topology

Dynamic Path Calculation

To ensure that traffic follows the least congested route, a Breadth-First Search (BFS) algorithm is used to explore all possible paths between a source and a destination. For each path, the cumulative bandwidth cost is calculated. The path with the lowest cost (i.e., least utilization) is selected for routing.

Key code for path calculation:

```python
def find_paths_and_costs(self, src, dst):
    if src == dst:
        return [Paths([src], 0)]
    queue = [(src, [src])]
    possible_paths = []
    while queue:
        (edge, path) = queue.pop(0)  # BFS uses a queue (FIFO)
        for vertex in set(self.neigh[edge]) - set(path):
            if vertex == dst:
                path_to_dst = path + [vertex]
                cost_of_path = self.find_path_cost(path_to_dst)
                possible_paths.append(Paths(path_to_dst, cost_of_path))
            else:
                queue.append((vertex, path + [vertex]))
    return possible_paths
```

4.3 Key Code

Link Utilization Monitoring

The current bandwidth utilization for each link is determined using OpenFlow's **PortStatsRequest** and **PortStatsReply** messages. The controller queries the switches at regular intervals (1 second) to gather statistics about the transmitted bytes on each port, which is then used to compute the bandwidth utilization.

Key code for gathering link utilization:

```python
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def _port_stats_reply_handler(self, ev):
    ''' Reply to the OFPPortStatsRequest '''
    switch_dpid = ev.msg.datapath.id
    for p in ev.msg.body:
        self.bw[switch_dpid][p.port_no] = (p.tx_bytes - self.prev_bytes[switch_dpid][p.port_no]) * 8.0 / 1000000
        self.prev_bytes[switch_dpid][p.port_no] = p.tx_bytes
```

4.4 Key Code

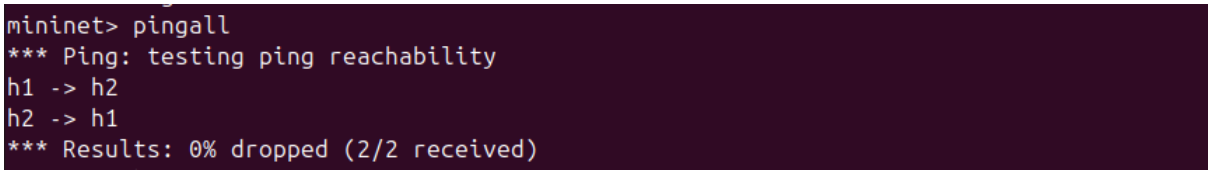
Path Installation

Once the least congested path is determined, flow entries are installed on the switches to direct traffic along the chosen path. The controller uses OpenFlow to install flow entries that match the source and destination IP addresses and forward packets through the optimal ports.

Observed Throughput and Ping Results

Ping Results

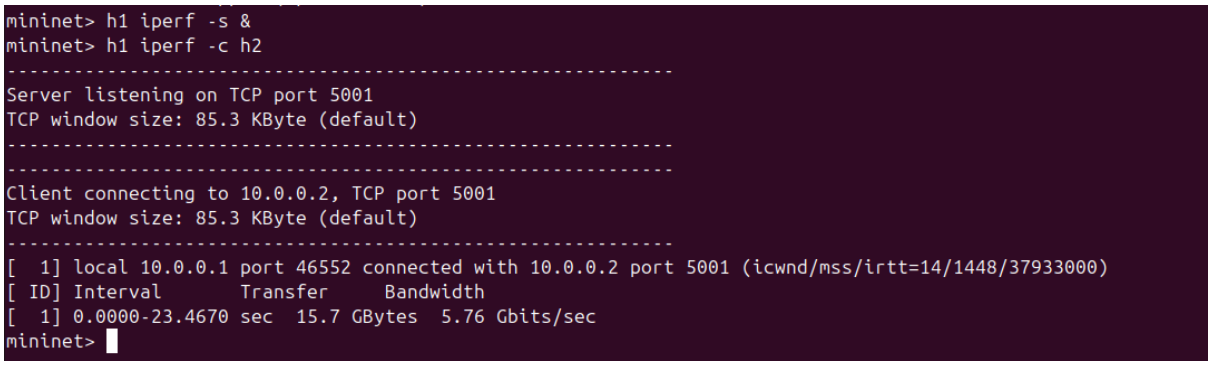
The network topology was tested using Mininet's pingall command to verify reachability between hosts. The following ping results were observed, confirming 100% connectivity between h1 and h2:



4.5 pingall Result

Throughput Results using iperf

The iperf tool was used to measure the bandwidth between h1 and h2. The server was initiated on h2, and the client on h1. The observed bandwidth between the two hosts was approximately **5.76 Gbps**, which is consistent with the expected bandwidth given the simulated traffic load and the link utilization.



4.6 iperf(Throughput Result)

Conclusion

This project successfully implemented a congestion-aware shortest path routing solution in a software-defined network using Ryu and Mininet. The routing algorithm dynamically selects paths based on real-time link utilization to avoid congestion and improve throughput. The results demonstrate that the solution works effectively, achieving optimal path selection and stable throughput under test conditions.

The design could be further optimized by implementing routing at a packet-level, allowing for finer granularity in traffic management. Additionally, the bandwidth monitoring interval could be adjusted dynamically based on traffic patterns to reduce controller overhead.