Introduction to Software Defined Networks(SDN)

Entry No.: 2022CS51136
Anubhav Pandey

Entry No.: 2022CS51139
Abhiram Dharme

Contents

Introduction

This report explores key aspects of Software-Defined Networking (SDN) by implementing fundamental network policies using OpenFlow-like APIs. Through the use of the Ryu controller framework, we have developed a Hub Controller, Learning Switch, Spanning Tree Protocol, and Shortest Path Routing algorithms. Our goal was to gain practical experience in SDN's centralized control paradigm, comparing different approaches for handling network traffic and improving network management.

1 Controller Hub and Learning Switch

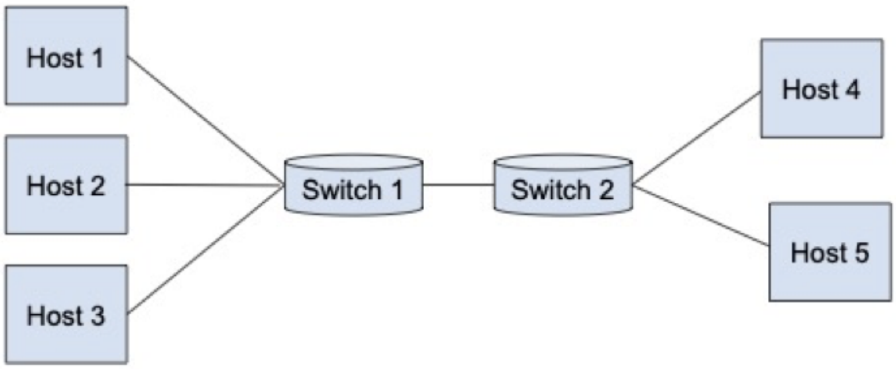
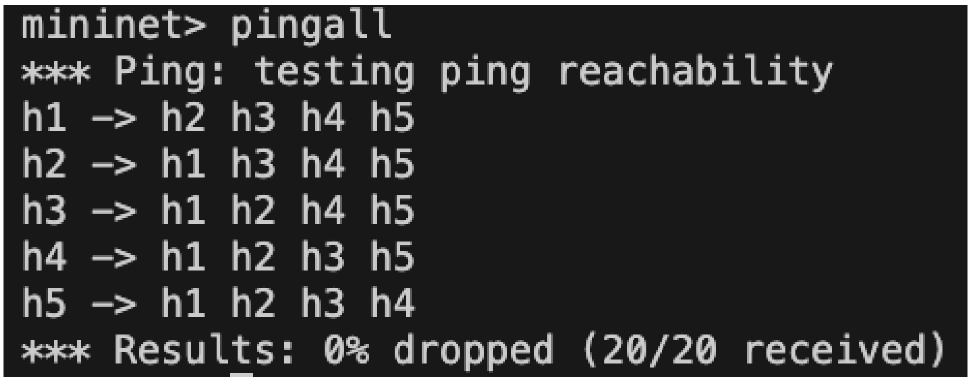


Figure 1: This is the given topology for the network.

1.1 Self-Learning Switch:

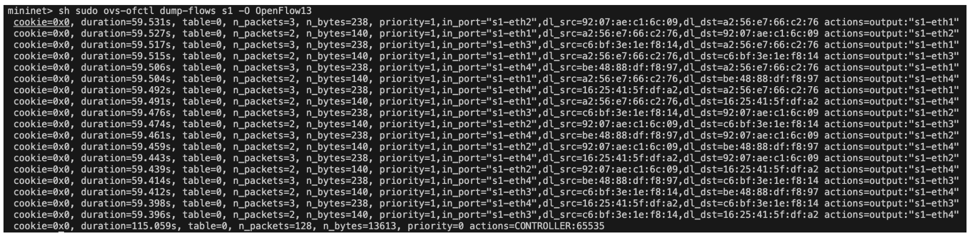
The self-learning switch mimics the behavior of traditional switches that learn MAC addresses dynamically. When a packet enters through a port, the switch learns the source MAC address and stores the corresponding port in its flow table. Future packets for that address are sent directly to the correct port, avoiding the need to flood the packet across the entire network.



```
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)
```

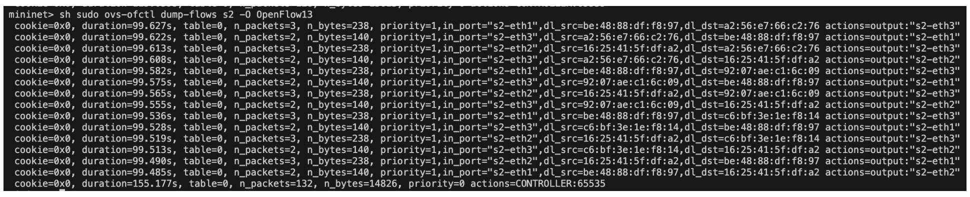
Figure 2: **pingall** for self-learning switch.

The test showed successful connectivity between hosts using the self-learning switch. After some initial flooding, the switch learns the network's MAC addresses and avoids unnecessary flooding.



```
mininet> sh sudo ovs-ofctl dump-flows s1 -O OpenFlow13
 cookie=0x0, duration=59.531s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth2",dl_src=92:07:ae:c1:6c:09,dl_dst=a2:56:e7:66:c2:76 actions=output:"s1-eth1"
 cookie=0x0, duration=59.527s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth1",dl_src=a2:56:e7:66:c2:76,dl_dst=92:07:ae:c1:6c:09 actions=output:"s1-eth2"
 cookie=0x0, duration=59.517s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth3",dl_src=c6:bf:3e:1e:f8:14,dl_dst=a2:56:e7:66:c2:76 actions=output:"s1-eth1"
 cookie=0x0, duration=59.515s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth1",dl_src=a2:56:e7:66:c2:76,dl_dst=c6:bf:3e:1e:f8:14 actions=output:"s1-eth3"
 cookie=0x0, duration=59.506s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth4",dl_src=be:48:88:df:f8:97,dl_dst=a2:56:e7:66:c2:76 actions=output:"s1-eth1"
 cookie=0x0, duration=59.504s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth1",dl_src=a2:56:e7:66:c2:76,dl_dst=be:48:88:df:f8:97 actions=output:"s1-eth4"
 cookie=0x0, duration=59.492s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth4",dl_src=16:25:41:5f:df:a2,dl_dst=a2:56:e7:66:c2:76 actions=output:"s1-eth1"
 cookie=0x0, duration=59.491s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth1",dl_src=a2:56:e7:66:c2:76,dl_dst=16:25:41:5f:df:a2 actions=output:"s1-eth4"
 cookie=0x0, duration=59.476s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth3",dl_src=c6:bf:3e:1e:f8:14,dl_dst=92:07:ae:c1:6c:09 actions=output:"s1-eth2"
 cookie=0x0, duration=59.474s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth2",dl_src=92:07:ae:c1:6c:09,dl_dst=c6:bf:3e:1e:f8:14 actions=output:"s1-eth3"
 cookie=0x0, duration=59.461s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth4",dl_src=be:48:88:df:f8:97,dl_dst=92:07:ae:c1:6c:09 actions=output:"s1-eth2"
 cookie=0x0, duration=59.459s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth2",dl_src=92:07:ae:c1:6c:09,dl_dst=be:48:88:df:f8:97 actions=output:"s1-eth4"
 cookie=0x0, duration=59.443s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth4",dl_src=16:25:41:5f:df:a2,dl_dst=92:07:ae:c1:6c:09 actions=output:"s1-eth2"
 cookie=0x0, duration=59.439s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth2",dl_src=92:07:ae:c1:6c:09,dl_dst=16:25:41:5f:df:a2 actions=output:"s1-eth4"
 cookie=0x0, duration=59.414s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth4",dl_src=be:48:88:df:f8:97,dl_dst=c6:bf:3e:1e:f8:14 actions=output:"s1-eth3"
 cookie=0x0, duration=59.412s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth3",dl_src=c6:bf:3e:1e:f8:14,dl_dst=be:48:88:df:f8:97 actions=output:"s1-eth4"
 cookie=0x0, duration=59.398s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth4",dl_src=16:25:41:5f:df:a2,dl_dst=c6:bf:3e:1e:f8:14 actions=output:"s1-eth3"
 cookie=0x0, duration=59.396s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth3",dl_src=c6:bf:3e:1e:f8:14,dl_dst=16:25:41:5f:df:a2 actions=output:"s1-eth4"
 cookie=0x0, duration=115.059s, table=0, n_packets=128, n_bytes=13613, priority=0 actions=CONTROLLER:65535
```

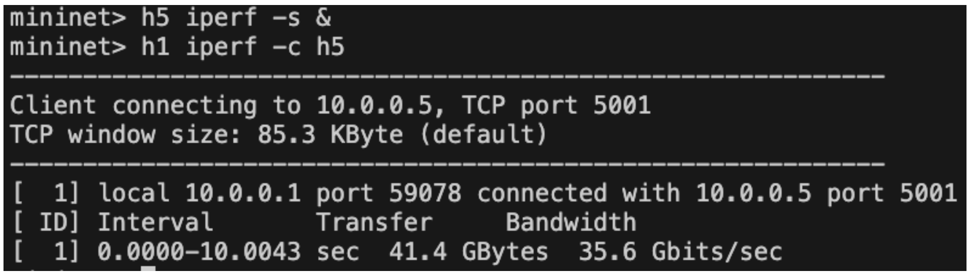
Figure 3: Flow table for switch 1 in self-learning switch.



```
mininet> sh sudo ovs-ofctl dump-flows s2 -O OpenFlow13
 cookie=0x0, duration=99.627s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s2-eth1",dl_src=be:48:88:df:f8:97,dl_dst=a2:56:e7:66:c2:76 actions=output:"s2-eth3"
 cookie=0x0, duration=99.622s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s2-eth3",dl_src=a2:56:e7:66:c2:76,dl_dst=be:48:88:df:f8:97 actions=output:"s2-eth1"
 cookie=0x0, duration=99.613s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s2-eth2",dl_src=16:25:41:5f:df:a2,dl_dst=a2:56:e7:66:c2:76 actions=output:"s2-eth3"
 cookie=0x0, duration=99.608s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s2-eth3",dl_src=a2:56:e7:66:c2:76,dl_dst=16:25:41:5f:df:a2 actions=output:"s2-eth2"
 cookie=0x0, duration=99.582s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s2-eth1",dl_src=be:48:88:df:f8:97,dl_dst=92:07:ae:c1:6c:09 actions=output:"s2-eth3"
 cookie=0x0, duration=99.575s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s2-eth3",dl_src=92:07:ae:c1:6c:09,dl_dst=be:48:88:df:f8:97 actions=output:"s2-eth1"
 cookie=0x0, duration=99.565s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s2-eth2",dl_src=16:25:41:5f:df:a2,dl_dst=92:07:ae:c1:6c:09 actions=output:"s2-eth3"
 cookie=0x0, duration=99.555s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s2-eth3",dl_src=92:07:ae:c1:6c:09,dl_dst=16:25:41:5f:df:a2 actions=output:"s2-eth2"
 cookie=0x0, duration=99.536s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s2-eth1",dl_src=be:48:88:df:f8:97,dl_dst=c6:bf:3e:1e:f8:14 actions=output:"s2-eth3"
 cookie=0x0, duration=99.528s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s2-eth3",dl_src=c6:bf:3e:1e:f8:14,dl_dst=be:48:88:df:f8:97 actions=output:"s2-eth1"
 cookie=0x0, duration=99.519s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s2-eth2",dl_src=16:25:41:5f:df:a2,dl_dst=c6:bf:3e:1e:f8:14 actions=output:"s2-eth3"
 cookie=0x0, duration=99.513s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s2-eth3",dl_src=c6:bf:3e:1e:f8:14,dl_dst=16:25:41:5f:df:a2 actions=output:"s2-eth2"
 cookie=0x0, duration=99.490s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s2-eth2",dl_src=16:25:41:5f:df:a2,dl_dst=be:48:88:df:f8:97 actions=output:"s2-eth1"
 cookie=0x0, duration=99.485s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s2-eth1",dl_src=be:48:88:df:f8:97,dl_dst=16:25:41:5f:df:a2 actions=output:"s2-eth2"
 cookie=0x0, duration=155.177s, table=0, n_packets=132, n_bytes=14826, priority=0 actions=CONTROLLER:65535
```

Figure 4: Flow table for switch 2 in self-learning switch.

In the flow tables, we can see that the switch stores MAC addresses as **dl_src** and **dl_dst**, allowing it to push packets to the appropriate **output_port** based on the learned information.



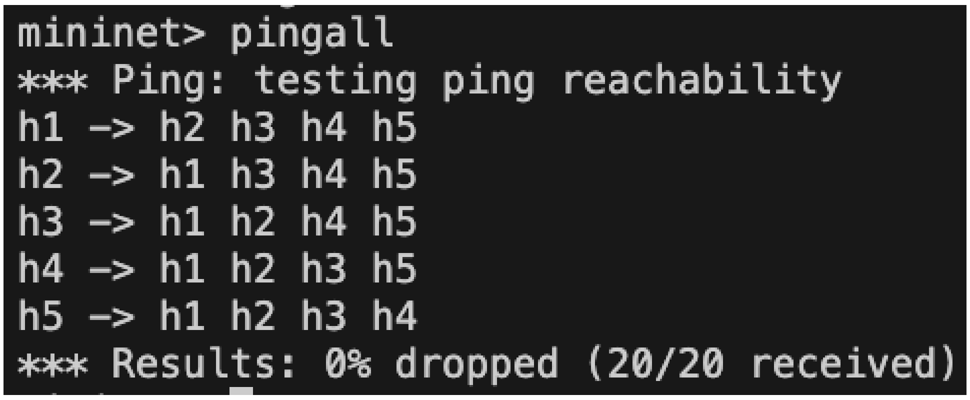
```
mininet> h5 iperf -s &
mininet> h1 iperf -c h5
------------------------------------------------------------
Client connecting to 10.0.0.5, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 59078 connected with 10.0.0.5 port 5001
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0043 sec  41.4 GBytes  35.6 Gbits/sec
```

Figure 5: Throughput for self-learning controller type.

The throughput for the self-learning switch was remarkably high, measured at around 35.6Gbps. This high throughput is due to the minimal processing involved once the MAC addresses are learned.

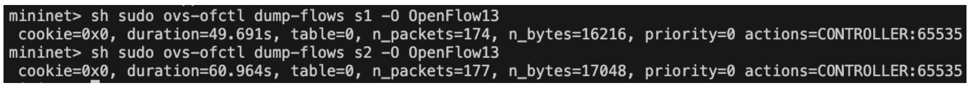
1.2 Hub Controller:

Unlike the self-learning switch, the Hub Controller is a simpler device that floods all incoming packets to all ports except the port it received them on. This approach mimics the behavior of an Ethernet hub.



```
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)
```

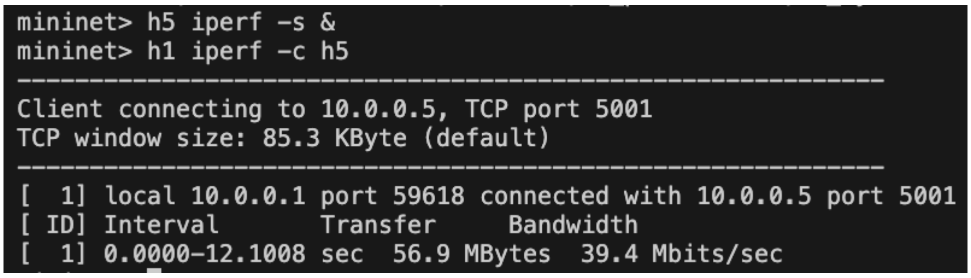
Figure 6: **pingall** for Hub.



```
mininet> sh sudo ovs-ofctl dump-flows s1 -O OpenFlow13
 cookie=0x0, duration=49.691s, table=0, n_packets=174, n_bytes=16216, priority=0 actions=CONTROLLER:65535
mininet> sh sudo ovs-ofctl dump-flows s2 -O OpenFlow13
 cookie=0x0, duration=60.964s, table=0, n_packets=177, n_bytes=17048, priority=0 actions=CONTROLLER:65535
```

Figure 7: s1 and s2 flow table for Hub.

Here, the controller doesn't maintain a flow table based on MAC addresses; instead, all traffic is broadcast. This reduces complexity but significantly affects performance.



```
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 59618 connected with 10.0.0.5 port 5001
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-12.1008 sec  56.9 MBytes  39.4 Mbits/sec
```

Figure 8: Throughput for hub controller type.

The throughput for the hub controller was much lower than the self-learning switch, coming in at just 39.4Mbps. This is expected, given that every packet must be flooded to all ports.

2 Spanning Tree Protocol (STP)

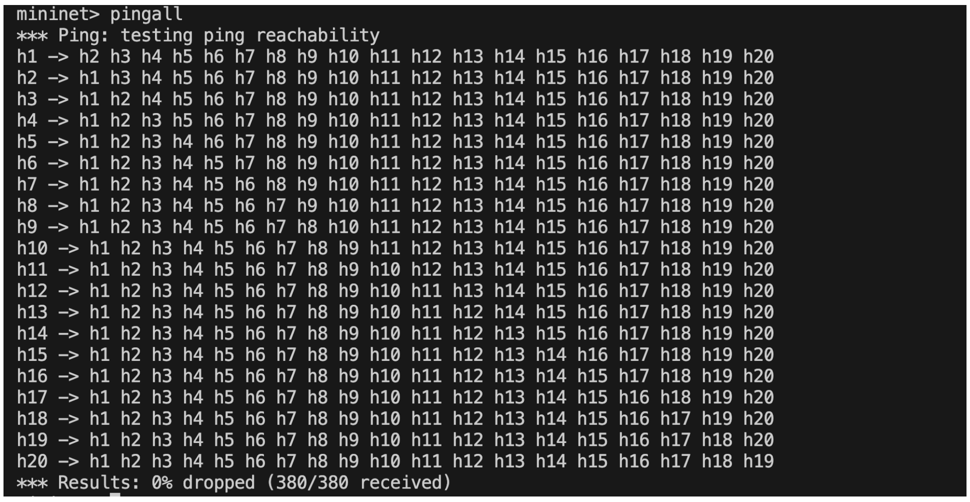
Spanning Tree Protocol (STP) is a critical protocol that ensures a loop-free topology by selectively blocking redundant paths between switches. In our implementation, we used the Ryu controller to construct a spanning tree to prevent network loops and ensure optimal packet routing.

2.1 Approach

To implement STP, we adopted the following approach:

- **Topology Discovery:** The controller monitors the network for topology changes. We use Ryu's topology API to detect switches joining or leaving the network and track updates to links and ports. The controller continuously updates its view of the network using `EventSwitchEnter` and `EventLinkAdd` events.
- **Adjacency Matrix:** The network topology is represented using an adjacency matrix, which tracks the connections between switches and the ports that connect them.
- **Spanning Tree Construction:** Using a Breadth-First Search (BFS) algorithm, the controller constructs a spanning tree. The BFS starts from the switch with the lowest DPID (Data Path Identifier), ensuring that all switches are visited without creating loops.
- **Port Configuration:** After the spanning tree is computed, the controller configures each switch's ports. Ports connecting switches that are part of the spanning tree are enabled, while redundant ports that would cause loops are disabled using the `OFPPC_NO_FWD` flag in OpenFlow.
- **Packet Forwarding:** When a packet is received, the controller checks if the destination MAC address is already known. If the address is known, the packet is

forwarded to the appropriate port. If the address is unknown, the controller floods the packet only through ports that are part of the spanning tree.



```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h12 h13 h14 h15 h16 h17 h18 h19 h20
h12 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h13 h14 h15 h16 h17 h18 h19 h20
h13 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h14 h15 h16 h17 h18 h19 h20
h14 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h15 h16 h17 h18 h19 h20
h15 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h16 h17 h18 h19 h20
h16 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h17 h18 h19 h20
h17 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h18 h19 h20
h18 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h19 h20
h19 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h20
h20 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19
*** Results: 0% dropped (380/380 received)
```

Figure 9: Pingall: moderate test

2.2 Assumptions

To simplify the implementation of STP, we made several assumptions:

- The switch with the lowest DPID is always selected as the root for the spanning tree.
- We assume the network topology remains stable during the computation of the spanning tree. No new switches or links are added until the tree is fully computed. Also, in case of addition or deletion of a link manually via terminal, the tree in computed again
- Hosts are directly connected to switches, and no loops exist between hosts themselves.
- Topology discovery is reliable, meaning that the controller is fully aware of all inter-switch links.
- Link failure recovery is not handled dynamically. If the topology changes, the spanning tree is recomputed from scratch.

The spanning tree protocol ensures that no loops are present in the network, allowing for efficient and safe packet forwarding.

Note : If there are x direct host connections to a switch, and there are a total of n hosts in the complete network, then there would be

$$2 \times \left(\frac{x(x-1)}{2}\right) + 2(x)(n-x) + 2$$

entries in the flow table. The multiplication factor of 2 accounts for the bidirectional transfer of packets. And additional 2 correspond to one the lowest priority entry which sends the packet to controller if not matched to anything else in the flow table, and the other the highest priority entry of lldp packets. (The action for packets that match this flow rule is to send them to the controller (e.g., the SDN controller managing the switch) with a buffer size of 65535. This means the switch is forwarding LLDP packets to the controller for further processing.)

3 Shortest Path Routing

In this part addition to STP, we implemented a Shortest Path Routing (SPR) controller that dynamically calculates the shortest path between switches using delay measurements as link weights. The controller is capable of managing the network and routing traffic efficiently, optimizing packet forwarding algorithm to optimize packet forwarding. SPR is designed to reduce the total number of hops packets need to traverse by selecting the shortest path between switches in the network.

3.1 Approach

The Shortest Path Routing controller follows these key steps:

- **Topology Discovery:** The controller monitors the network for changes in topology, detecting switches and ports using the `OFPStateChange` event. It updates the network topology dynamically by maintaining adjacency and port mappings for all switches.

- **Delay Measurement:** To determine the optimal path, the controller measures link delays by sending custom probe packets across all active links in the network. This provides real-time delay values that are used as weightalgorithm leverages Dijkstra's algorithm to compute the optimal paths between switches:

  - **Topology Discovery:** Similar to STP, the controller continuously monitors the network for topology changes and builds an adjacency matrix of the switches and their connections.
  - **Delay Measurement:** We periodically send probe packets between switches to measure link delays. These delays serve as the weights for the links when computing the shortest paths.
  - **Graph Construction:** The controller builds a network graph, whereith switches ares nodes and the links (edges) between them are weighted based on the measured delay. If no measurement is available, a default delay is assumed.
  - **Shortest Path Calculation:** Usinglink delays as edge weights, updating it based on delay measurements.
  - **Dijkstra's Algorithm:** Once the topology is known, the controller applies Dijkstra's algorithm, the controllero computes the shortest paths between all pairs of switches in the network. The shortest path is selected based on the minimal total link delay.
  - **Flow Rule Installation:** Once the shortest path is computed, flow rules are installed in each switch along the path. These rules ensure that packets are forwarded to the next switch in the path. The rules are installed bidirectionally for boswitches.

- **Flow Rule Installation:** Based on the computed paths, the controller installs flow rules in each switch to forward packets along the optimal path. Each switch stores the source and destination to allow for bidirectional communicationhortest path information in its flow table, ensuring efficient forwarding.
- **ARP Handling and Packet Flooorwarding:** ARPWhen a packets are handled separately by maintaining an ARP cache, allowing the controller to respond to ARP requests efficiently. If the destination is unknown, the packet is flooded throughout the network, but mechanisms are inrives at a switch, the flow table is checked for the destination MAC address. If a rule exists, the packet is forwarded to the correct port. Otherwise, the place to avoid redundant floodket is flooded or sent to the controller for further processing.

3.2 Assumptions

To ensure a smooth and efficientWe made the following assumptions during the implementation, of the following assumptions were madeshortest path routing algorithm:

- The network topology remains stable while delay measurements are taken and paths are computed.
- The controller has full knowledge of all inter-switch links and ports, and topology discovery is reliable.
- All switches in the network are SDN-compatible and promptly respond to flow rule installation requests.
- ARP requests and replies are correctly handled, and the controller's ARP cache is reliable.
- Packet loss does not occur during the delay measurement process, ensuring accurate computation of shortest paths.

3.3 Results and Performance

The controller successfully computes and installs flow rules based on the shortest path, ensuring that network traffic is forwarded optimally. Measured throughput and delay values were used to compute paths, and the bidirectional flow rules enabled efficient communication between hosts. The ARP handling mechanism also ensured that hosts could discover each other seamlessly without unnecessary flooding. is static during the execution of Dijkstra's algorithm.

- The controller has full knowledge of the network topology and all inter-switch link costs.
- Probe packets are assumed to be successfully received and processed for accurate delay measurement.

4 Part 4: Congestion-aware Shortest Path Routing

In this part, we modify the existing shortest path routing application to take into account the current congestion in the network in order to dynamically adjust the routing decisions and to recalculate the delays after every **PATH_COMPUTE_INTERVAL** seconds to get the most recent delays from the network topology then perform Shortest Path Routing. This implementation is based on the Ryu controller framework and uses the OpenFlow protocol to interact with the switches in the network.

4.1 Link Delay Measurement using LLDP

To measure the link delay between switches, the controller processes the received Link Layer Discovery Protocol (LLDP) packets. Each switch periodically broadcasts LLDP packets, which include the source DPID (Datapath ID) and the port number from which the packet was sent. Upon receiving an LLDP packet, the controller calculates the link delay by extracting a timestamp embedded in the packet and subtracting it from the current time. The delay is then updated in the controller's internal data structures, which are used later for routing decisions.

4.2 Link Congestion Estimation

To estimate the congestion on each link, the controller periodically requests port statistics from each switch. The statistics include the number of transmitted packets and the number of dropped packets for each port. By calculating the difference between the current and previous statistics, the controller estimates the congestion level. Specifically, the number of dropped packets is used as an indicator of congestion. The controller maintains a table of congestion metrics for all links, which is used to influence the path selection.

4.3 Computation of Link Costs

The controller maintains a graph representing the network topology, with switches as nodes and links as edges. The cost of each link is computed based on both the measured delay and the congestion level. The link cost is defined as:

$$Cost_{link} = Delay_{link} + \alpha \cdot Congestion_{link} \tag{1}$$

Here, α is a tunable parameter that controls the weight given to the congestion in the overall link cost. A higher value of α places more emphasis on avoiding congested links, while a lower value prioritizes shorter delays.

4.4 Shortest Path Calculation using Dijkstra's Algorithm

Once the link costs are updated, the controller computes the shortest paths between all switches using Dijkstra's algorithm. The algorithm finds the shortest path from each switch to every other switch by minimizing the cumulative link cost. The shortest paths are stored in a table, which is later used to install flow entries for specific traffic flows. Note that Dijktra's algorithm is called periodically, each time when we are recalculating the network due to congestion.

4.5 Path Installation

For each computed path, the controller installs flow entries in the switches along the path. These flow entries ensure that packets destined for a particular MAC address follow the shortest path from the source to the destination. Additionally, reverse paths are installed to allow for bidirectional communication. If a link becomes congested or a delay increases significantly, the controller recomputes the paths and updates the flow entries to reflect the new network conditions.

4.6 Handling ARP Requests and Replies

The controller handles Address Resolution Protocol (ARP) requests and replies by maintaining an ARP cache that maps IP addresses to MAC addresses. If the controller knows the destination MAC address for an ARP request, it generates an ARP reply directly. Otherwise, it floods the ARP request to all switches. The ARP cache is updated upon receiving ARP replies, allowing the controller to respond to future ARP requests more efficiently.

4.7 Flooding and Packet Tracking

To avoid flooding the same packet multiple times, the controller keeps track of flooded packets using a unique identifier. This prevents the network from being overwhelmed by repeated flooding, especially in cases where the destination MAC address is unknown. The controller ensures that packets are only flooded once by maintaining a set of flooded packet identifiers.

4.8 Conclusion

This modified shortest path routing application dynamically adjusts its routing decisions based on real-time link utilization metrics. By incorporating both delay and congestion into the link cost computation, the controller is able to route traffic along paths that minimize both latency and network congestion. This approach improves overall network performance, especially in scenarios with high traffic volumes or varying link conditions.