Assignment 3: An Introduction to the World of SDN

COL334/672, Diwali'25

October 17, 2025

Students:
Rishika Garg (2023CS10820)
Apurva Samota (2023CS10585)

Goal

The goal of this assignment was to gain hands-on experience with Software Defined Networking (SDN) by implementing network policies using OpenFlow-like APIs. We used **Mininet** along with the **Ryu Controller** for implementation and experimentation.

1 Part 1: Hub Controller and Learning Switch (20%)

In this part, we implemented two types of controllers and compared their performance: a **Hub Controller** and a **Learning Switch**.

Hub Controller

A Hub Controller forces the switch to act like a simple hub. All packets are sent to the controller, which then decides to either forward to a known port or flood the packet. Crucially, no forwarding rules are installed on the switch itself; it remains dependent on the controller for every packet.

Learning Switch

A Learning Switch makes the switch intelligent. It learns MAC-to-port mappings from incoming traffic and installs specific flow rules on the switch. Subsequent packets matching these rules are handled directly by the switch hardware, without needing to contact the controller.

Experiments and Results

Flow Rules Comparison

```
mininet> sh ovs-ofctl dump-flows s1
 cookie=0x0, duration=21.643s, table=0, n_packets=76, n_bytes=5752, priority=0 actions=CONTROLLER:65535
mininet> sh ovs-ofctl dump-flows s2
 cookie=0x0, duration=45.468s, table=0, n_packets=81, n_bytes=6102, priority=0 actions=CONTROLLER:65535
```

Figure 1: Flow table for Hub Controller. Only a default rule is present.

Figure 2: Flow table for Learning Switch Controller. Specific rules are learned and installed.

Throughput Comparison

Figure 3: Throughput results for Hub Controller.

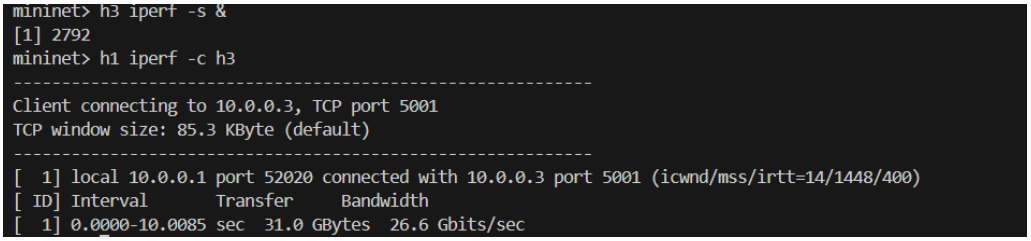

Figure 4: Throughput results for Learning Switch Controller.

Assumptions and Observations

Our primary assumption is that the Mininet virtual environment provides a high-capacity data plane, so any performance differences will be due to the controller logic and the interaction between the switch (data plane) and the controller (control plane).

1. **Flow Rules:** The results perfectly illustrate the difference between the two controllers.
    - With the **Hub Controller** (Figure 1), the switches only have a single, low-priority "table-miss" rule. This rule's only action is 'CONTROLLER', meaning every single packet is sent to the controller because no other rule will ever match. The switch hardware is not used for forwarding decisions.

- With the **Learning Switch** (Figure 2), the switch has learned specific MAC addresses. We can see multiple higher-priority rules that match on the incoming port and a destination MAC address (e.g., `in_port="s1-eth1",dl_dst=00:...:02`) and forward to a specific output port (`actions=output:"s1-eth2"`). This offloads the forwarding task from the controller to the switch.

2. **Throughput:** The performance difference is stark.

   - The **Hub Controller** (Figure 3) achieves a very low throughput of **15.9 Mbits/sec**. This is because the controller is a significant bottleneck. Every packet from the iperf test must travel from the switch to the controller and back, incurring latency and processing overhead for each packet.
   - The **Learning Switch** (Figure 4) achieves an extremely high throughput of **26.6 Gbits/sec**. This is because after the first few packets, flow rules are installed on the switch. All subsequent data packets are processed directly by the switch's data plane at line rate, bypassing the controller entirely. This demonstrates the core benefit of SDN: centralizing control for initial setup while allowing the data plane to operate at maximum speed.

2 Part 2: Layer2-like Shortest Path Routing (30%)

We implemented a custom Ryu controller (`p2_l2spf.py`) that performs L2-like shortest-path routing using Dijkstra's algorithm. The controller discovers the topology, computes shortest paths between hosts, and installs proactive flow rules to enable efficient forwarding.

Topology

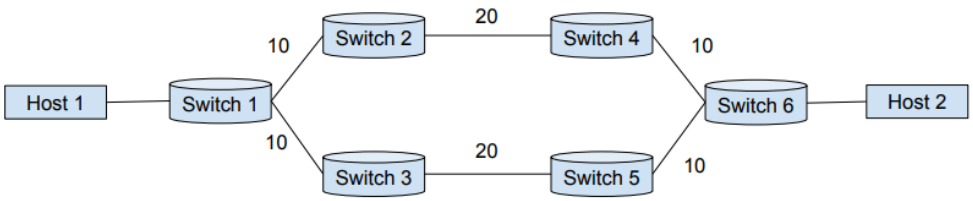

Figure 5: Network topology for Layer-2 Shortest Path Routing.

The topology was designed to have two equal-cost paths between h1 and h2:

- Path 1: `s1 -- s2 -- s3 -- s6`
- Path 2: `s1 -- s4 -- s5 -- s6`

All links were configured with a bandwidth of 10 Mbit/s.

Experiments and Results

We ran an `iperf` test between h1 and h2 for 10 seconds with 2 parallel TCP connections. The experiment was conducted twice: once with the ECMP flag set to `false` and once with it set to `true`.

Case 1: ECMP Disabled

```
mininet> h2 iperf -s &
mininet> h1 iperf -c h2 -t 10 -P 2
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 55956 connected with 10.0.0.2 port 5001 (icwnd/mss/irtt=14/1448/1000)
[  2] local 10.0.0.1 port 55954 connected with 10.0.0.2 port 5001 (icwnd/mss/irtt=14/1448/1000)
[ ID] Interval       Transfer     Bandwidth
[  2] 0.0000-12.9019 sec  7.25 MBytes  4.71 Mbits/sec
[  1] 0.0000-13.2758 sec  7.63 MBytes  4.82 Mbits/sec
[SUM] 0.0000-10.7219 sec  14.9 MBytes  11.6 Mbits/sec
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=24.044s, table=0, n_packets=50, n_bytes=3000, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=
0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=17.237s, table=0, n_packets=145, n_bytes=8005038, priority=1,tcp,nw_src=10.0.0.1,nw_dst=10.0.0
.2,tp_src=55956,tp_dst=5001 actions=output:"s1-eth2"
 cookie=0x0, duration=17.237s, table=0, n_packets=143, n_bytes=9474, priority=1,tcp,nw_src=10.0.0.2,nw_dst=10.0.0.1,
tp_src=5001,tp_dst=55956 actions=output:"s1-eth1"
 cookie=0x0, duration=17.237s, table=0, n_packets=135, n_bytes=7611162, priority=1,tcp,nw_src=10.0.0.1,nw_dst=10.0.0
.2,tp_src=55954,tp_dst=5001 actions=output:"s1-eth2"
 cookie=0x0, duration=17.237s, table=0, n_packets=132, n_bytes=8748, priority=1,tcp,nw_src=10.0.0.2,nw_dst=10.0.0.1,
tp_src=5001,tp_dst=55954 actions=output:"s1-eth1"
 cookie=0x0, duration=24.047s, table=0, n_packets=20047, n_bytes=1604382, priority=0 actions=CONTROLLER:65535
*** s2 ------------------------------------------------------------------------
 cookie=0x0, duration=24.047s, table=0, n_packets=46, n_bytes=2760, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=
0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=17.241s, table=0, n_packets=144, n_bytes=8004964, priority=1,tcp,nw_src=10.0.0.1,nw_dst=10.0.0
.2,tp_src=55956,tp_dst=5001 actions=output:"s2-eth2"
 cookie=0x0, duration=17.241s, table=0, n_packets=143, n_bytes=9474, priority=1,tcp,nw_src=10.0.0.2,nw_dst=10.0.0.1,
tp_src=5001,tp_dst=55956 actions=output:"s2-eth1"
 cookie=0x0, duration=17.241s, table=0, n_packets=134, n_bytes=7611088, priority=1,tcp,nw_src=10.0.0.1,nw_dst=10.0.0
.2,tp_src=55954,tp_dst=5001 actions=output:"s2-eth2"
 cookie=0x0, duration=17.241s, table=0, n_packets=132, n_bytes=8748, priority=1,tcp,nw_src=10.0.0.2,nw_dst=10.0.0.1,
tp_src=5001,tp_dst=55954 actions=output:"s2-eth1"
 cookie=0x0, duration=24.051s, table=0, n_packets=20102, n_bytes=1608892, priority=0 actions=CONTROLLER:65535
```

Figure 6: Throughput with ECMP disabled.

```
 cookie=0x0, duration=17.241s, table=0, n_packets=132, n_bytes=8748, priority=1,tcp,nw_src=10.0.0.2,nw_dst=10.0.0.1,
tp_src=5001,tp_dst=55954 actions=output:"s2-eth1"
 cookie=0x0, duration=24.051s, table=0, n_packets=20102, n_bytes=1608892, priority=0 actions=CONTROLLER:65535
*** s3 ------------------------------------------------------------------------
 cookie=0x0, duration=24.055s, table=0, n_packets=48, n_bytes=2880, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=
0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=24.058s, table=0, n_packets=20057, n_bytes=1605270, priority=0 actions=CONTROLLER:65535
*** s4 ------------------------------------------------------------------------
 cookie=0x0, duration=24.062s, table=0, n_packets=49, n_bytes=2940, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=
0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=17.254s, table=0, n_packets=144, n_bytes=8004964, priority=1,tcp,nw_src=10.0.0.1,nw_dst=10.0.0
.2,tp_src=55956,tp_dst=5001 actions=output:"s4-eth2"
 cookie=0x0, duration=17.254s, table=0, n_packets=143, n_bytes=9474, priority=1,tcp,nw_src=10.0.0.2,nw_dst=10.0.0.1,
tp_src=5001,tp_dst=55956 actions=output:"s4-eth1"
 cookie=0x0, duration=17.254s, table=0, n_packets=134, n_bytes=7611088, priority=1,tcp,nw_src=10.0.0.1,nw_dst=10.0.0
.2,tp_src=55954,tp_dst=5001 actions=output:"s4-eth2"
 cookie=0x0, duration=17.254s, table=0, n_packets=132, n_bytes=8748, priority=1,tcp,nw_src=10.0.0.2,nw_dst=10.0.0.1,
tp_src=5001,tp_dst=55954 actions=output:"s4-eth1"
 cookie=0x0, duration=24.065s, table=0, n_packets=20085, n_bytes=1605920, priority=0 actions=CONTROLLER:65535
*** s5 ------------------------------------------------------------------------
 cookie=0x0, duration=24.065s, table=0, n_packets=46, n_bytes=2760, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=
0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=24.069s, table=0, n_packets=20079, n_bytes=1606142, priority=0 actions=CONTROLLER:65535
*** s6 ------------------------------------------------------------------------
 cookie=0x0, duration=24.074s, table=0, n_packets=49, n_bytes=2940, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=
0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=17.266s, table=0, n_packets=144, n_bytes=8004964, priority=1,tcp,nw_src=10.0.0.1,nw_dst=10.0.0
.2,tp_src=55956,tp_dst=5001 actions=output:"s6-eth1"
 cookie=0x0, duration=17.266s, table=0, n_packets=143, n_bytes=9474, priority=1,tcp,nw_src=10.0.0.2,nw_dst=10.0.0.1,
tp_src=5001,tp_dst=55956 actions=output:"s6-eth2"
 cookie=0x0, duration=17.266s, table=0, n_packets=134, n_bytes=7611088, priority=1,tcp,nw_src=10.0.0.1,nw_dst=10.0.0
.2,tp_src=55954,tp_dst=5001 actions=output:"s6-eth1"
 cookie=0x0, duration=17.266s, table=0, n_packets=132, n_bytes=8748, priority=1,tcp,nw_src=10.0.0.2,nw_dst=10.0.0.1,
tp_src=5001,tp_dst=55954 actions=output:"s6-eth2"
 cookie=0x0, duration=24.077s, table=0, n_packets=20086, n_bytes=1606120, priority=0 actions=CONTROLLER:65535
mininet>
```

Figure 7: Flow rules with ECMP disabled.

Case 2: ECMP Enabled

```
(base) apporv@LAPTOP-LPGCIC9N:/mnt/d/Networks/Assignment_3/part2$ sudo python3 p2_topo.py
*** Creating network
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2 s3 s4 s5 s6
*** Adding links:
(h1, s1) (h2, s6) (10.00Mbit) (10.00Mbit) (s1, s2) (10.00Mbit) (10.00Mbit) (s1, s3) (10.00Mbit) (10.00Mbit) (s2, s4)
 (10.00Mbit) (10.00Mbit) (s3, s5) (10.00Mbit) (10.00Mbit) (s4, s6) (10.00Mbit) (10.00Mbit) (s5, s6)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 6 switches
s1 s2 s3 s4 s5 s6 ...(10.00Mbit) (10.00Mbit) (10.00Mbit) (10.00Mbit) (10.00Mbit) (10.00Mbit) (10.00Mbit)
 (10.00Mbit) (10.00Mbit) (10.00Mbit) (10.00Mbit)
*** Running CLI
*** Starting CLI:
mininet> h2 iperf -s &
mininet> h1 iperf -c h2 -t 10 -P 2
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  2] local 10.0.0.1 port 55184 connected with 10.0.0.2 port 5001 (icwnd/mss/irtt=14/1448/500659)
[  1] local 10.0.0.1 port 55182 connected with 10.0.0.2 port 5001 (icwnd/mss/irtt=14/1448/1000)
[ ID] Interval       Transfer     Bandwidth
[  2] 0.0000-12.0453 sec  13.4 MBytes  9.31 Mbits/sec
[  1] 0.0000-12.2726 sec  13.6 MBytes  9.31 Mbits/sec
[SUM] 0.0000-10.6357 sec  27.0 MBytes  21.3 Mbits/sec
```

Figure 8: Throughput with ECMP enabled.

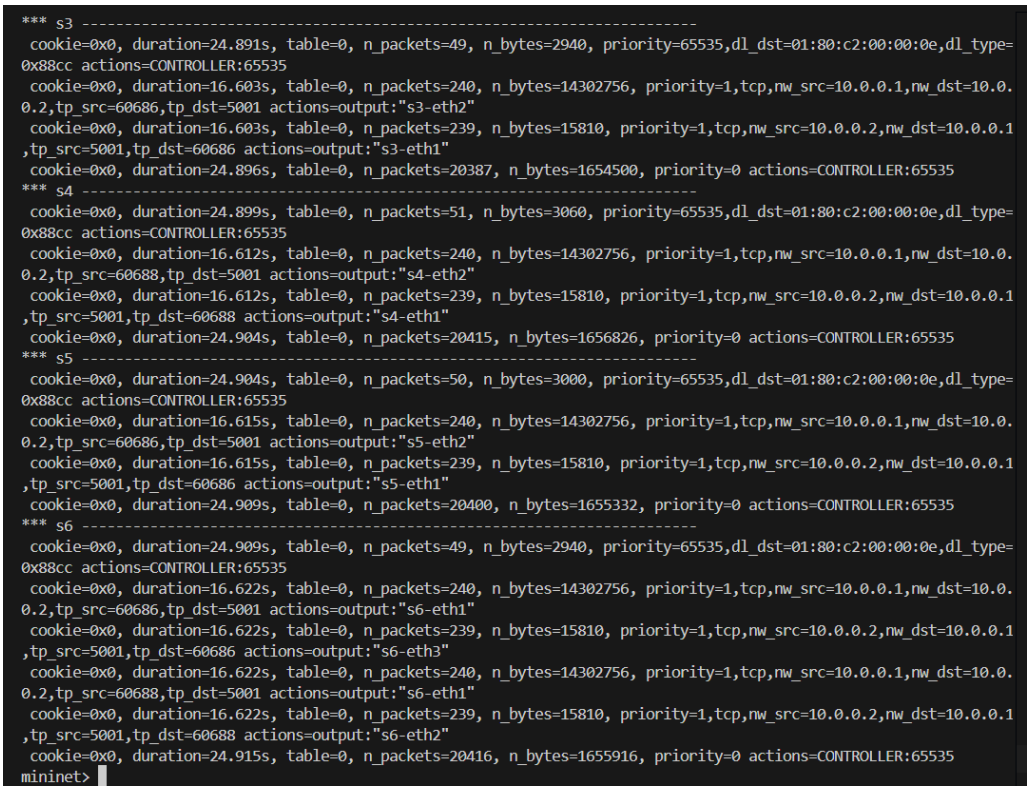

Figure 9: Flow rules on switches with ECMP enabled.

Assumptions and Observations

Our primary assumption is that the switches perform load balancing based on a **5-tuple hash** (Source IP, Destination IP, Protocol, Source Port, Destination Port) when multiple paths are available. When `iperf` runs with the `-P 2` flag, it creates two separate TCP connections, each with a unique source port. This difference in the 5-tuple allows the switch to potentially route them along different paths.

1. **ECMP Disabled:** As seen in Figure 6, the total throughput was approximately **11.6 Mbits/sec**. The controller calculated the two shortest paths but only chose one to install flow rules. Both parallel TCP connections were forced down this single path. Since the link bandwidth is 10 Mbits/sec, the two flows had to share this capacity, resulting in a lower combined throughput that slightly exceeds the link capacity due to burstiness and protocol overhead. The flow rules in Figure 7 confirm that only one path is active.

2. **ECMP Enabled:** As seen in Figure 8, the total throughput was significantly higher at approximately **21.3 Mbits/sec**. Each parallel flow achieved around 9.3 Mbits/sec, which is close to the maximum capacity of a single 10 Mbits/sec link. This demonstrates that the controller successfully installed rules for both paths, and the switch (`s1`) distributed the two flows across the two different paths. The different source ports of the TCP connections resulted in different hash values, leading to effective load balancing. The total throughput is effectively the sum of the bandwidth of the two paths ($10 + 10 = 20$ Mbits/sec). The flow rules in Figure 9 show entries on switches across both paths (e.g., s3 (for 1-3-5-6 path) and s4 (for 1-2-4-6 path)), confirming that both routes were actively used.

Bonus: Weighted Load Balancing

Mechanism: We modified the controller to be state-aware. It periodically polls switches for port statistics (byte counts) to calculate the current utilization of each link in Mbps. When a new flow needs a path, the controller calculates the "bottleneck load" for each available equal-cost path—defined as the maximum utilization of any single link along that path. It then installs flow rules for the path with the **minimum bottleneck load**.

Validation: We conducted a two-stage experiment. First, we initiated a high-traffic UDP flow from h1 to h2, designed to saturate one path. While it was running, we initiated a second, distinct UDP flow.

```
--- New Flow 10.0.0.1->10.0.0.2 (Port 45759) ---
    Path [1, 3, 5, 6] chosen with bottleneck load: 0.24 Mbps
```

Figure 10: The first flow (port 45759) is assigned to Path [1, 3, 5, 6], which is currently idle with a bottleneck load of 0.24 Mbps.

```
Load on link (3 -> 5): 8.55 Mbps
Load on link (5 -> 3): 0.20 Mbps
Load on link (5 -> 6): 8.45 Mbps
Load on link (6 -> 4): 0.21 Mbps
Load on link (6 -> 5): 0.21 Mbps
Load on link (4 -> 2): 0.20 Mbps
Load on link (4 -> 6): 0.20 Mbps
Load on link (2 -> 1): 0.20 Mbps
Load on link (2 -> 4): 0.21 Mbps
--- New Flow 10.0.0.1->10.0.0.2 (Port 46854) ---
    Path [1, 2, 4, 6] chosen with bottleneck load: 0.21 Mbps
```

Figure 11: After the first flow creates high load, the second flow (port 46854) is intelligently assigned to the alternate, less-congested Path [1, 2, 4, 6] with a bottleneck load of 0.21 Mbps.

```
*** s2 ----------------------------------------------------------------------
 cookie=0x0, duration=94.135s, table=0, n_packets=175, n_bytes=10500, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0
x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=21.034s, table=0, n_packets=1692, n_bytes=2558304, priority=1,udp,nw_src=10.0.0.1,nw_dst=10.0.0.2
,tp_src=47348,tp_dst=5002 actions=output:"s2-eth2"
 cookie=0x0, duration=21.034s, table=0, n_packets=1, n_bytes=170, priority=1,udp,nw_src=10.0.0.2,nw_dst=10.0.0.1,tp_src
=5002,tp_dst=47348 actions=output:"s2-eth1"
 cookie=0x0, duration=94.141s, table=0, n_packets=62815, n_bytes=5067252, priority=0 actions=CONTROLLER:65535
*** s3 ----------------------------------------------------------------------
 cookie=0x0, duration=94.140s, table=0, n_packets=175, n_bytes=10500, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0
x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=57.016s, table=0, n_packets=40326, n_bytes=60972912, priority=1,udp,nw_src=10.0.0.1,nw_dst=10.0.0
.2,tp_src=43903,tp_dst=5001 actions=output:"s3-eth2"
 cookie=0x0, duration=57.016s, table=0, n_packets=0, n_bytes=0, priority=1,udp,nw_src=10.0.0.2,nw_dst=10.0.0.1,tp_src=5
001,tp_dst=43903 actions=output:"s3-eth1"
 cookie=0x0, duration=94.146s, table=0, n_packets=63159, n_bytes=5092832, priority=0 actions=CONTROLLER:65535
```

Figure 12: Flow rules on switches s2 and s3 confirm that UDP flows (matched by destination ports) have been installed along the both paths.

Observations and Comparison: Our primary assumption was that the controller could accurately measure and react to network load changes between the arrivals of new flows. The results strongly validate this approach.

- The experiment began with an idle network. As shown in Figure 10, the controller correctly identified that both paths were clear and placed the first UDP flow on Path [1, 3, 5, 6].
- This first flow saturated its path. When the second UDP flow arrived, the controller's statistics showed high utilization on the first path. Consequently, it made an intelligent decision to route the new flow via the alternate, uncongested Path [1, 2, 4, 6], as shown in Figure 11.
- The flow rules installed on switches `s2` and `s3` (Figure 12) provide concrete evidence of this process. We can see rules specifically matching the UDP flows by their destination ports, directing them along the chosen path towards the next hop. Similar rules would be present on `s4` and `s5` for the second flow.
- This state-aware mechanism is a significant improvement over hash-based ECMP. While standard ECMP is effective at distributing a large number of small flows, it is stateless and could randomly assign two high-bandwidth flows to the same path, causing congestion while an alternate path remains idle. Our weighted, utilization-based approach actively avoids this "hash collision" problem for heavy flows, leading to more optimal and predictable network performance.

3 Part 3: Layer3-like Shortest Path Routing (25%)

In this part, we implemented a Layer-3 shortest path controller (`p3_l3spf.py`) that enables routing across different subnets. The controller uses Dijkstra's algorithm to find the shortest path and installs flow rules that perform necessary L2 header modifications and L3 TTL management.

Experiments and Results

We tested the controller by pinging from h1 (subnet 10.0.12.0/24) to h2 (subnet 10.0.67.0/24).

Ping Results and Explanation

```
mininet> h1 ping -c 5 h2
PING 10.0.67.2 (10.0.67.2) 56(84) bytes of data.
64 bytes from 10.0.67.2: icmp_seq=3 ttl=60 time=0.882 ms
64 bytes from 10.0.67.2: icmp_seq=4 ttl=60 time=0.081 ms
64 bytes from 10.0.67.2: icmp_seq=5 ttl=60 time=0.069 ms

--- 10.0.67.2 ping statistics ---
5 packets transmitted, 3 received, 40% packet loss, time 4091ms
rtt min/avg/max/mdev = 0.069/0.344/0.882/0.380 ms
mininet> h1 ping -c 5 h2
PING 10.0.67.2 (10.0.67.2) 56(84) bytes of data.
64 bytes from 10.0.67.2: icmp_seq=1 ttl=60 time=0.056 ms
64 bytes from 10.0.67.2: icmp_seq=2 ttl=60 time=0.085 ms
64 bytes from 10.0.67.2: icmp_seq=3 ttl=60 time=0.086 ms
64 bytes from 10.0.67.2: icmp_seq=4 ttl=60 time=0.077 ms
64 bytes from 10.0.67.2: icmp_seq=5 ttl=60 time=0.084 ms

--- 10.0.67.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4238ms
rtt min/avg/max/mdev = 0.056/0.077/0.086/0.011 ms
```

Figure 13: Ping results from h1 to h2. The first attempt shows packet loss as the controller learns the path, while the second attempt is successful.

The ping results clearly demonstrate the reactive nature of our SDN controller.

- **Initial Packet Loss:** The first ping attempt shows 40% packet loss. This is expected. The first ICMP packet from h1 to h2 did not match any existing flow rule on switch s1. This triggered a 'PacketIn' event, sending the packet to the Ryu controller. The controller then performed several actions: resolving ARP for the gateway, calculating the shortest path to h2's subnet (s1 $\rightarrow$ s2 $\rightarrow$ s3 $\rightarrow$ s6), and proactively installing the necessary L3 forwarding rules on all switches along this path. This entire process caused the initial packets to be dropped, resulting in the observed loss.

- **Subsequent Success:** The second ping attempt shows 0% packet loss with extremely low latency (avg RTT of 0.077 ms). This is because the flow rules were now installed in the switch hardware. All subsequent packets matched these rules and were processed entirely in the data plane at high speed, without any need to contact the controller.

- **TTL Verification:** The received packets have a TTL of 60. Since the default host TTL is 64, this confirms that the packet traversed 4 router hops. Our path (s1 $\rightarrow$ s2 $\rightarrow$ s3 $\rightarrow$ s6) consists of 4 switches, which validates that each switch correctly acted as a router and decremented the TTL.

Flow Rules and Explanation

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=103.875s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=94.093s, table=0, n_packets=9, n_bytes=882, priority=1,ip,nw_dst=10.0.67.2 actions=dec_ttl,mod_dl
_src:00:00:00:00:01:02,mod_dl_dst:00:00:00:00:02:01,output:"s1-eth2"
 cookie=0x0, duration=93.072s, table=0, n_packets=8, n_bytes=784, priority=1,ip,nw_dst=10.0.12.2 actions=dec_ttl,mod_dl
_src:00:00:00:00:01:01,mod_dl_dst:00:00:00:00:01:02,output:"s1-eth1"
 cookie=0x0, duration=103.875s, table=0, n_packets=28, n_bytes=2284, priority=0 actions=CONTROLLER:65535
*** s2 ------------------------------------------------------------------------
 cookie=0x0, duration=103.856s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=94.098s, table=0, n_packets=9, n_bytes=882, priority=1,ip,nw_dst=10.0.67.2 actions=dec_ttl,mod_dl
_src:00:00:00:00:02:02,mod_dl_dst:00:00:00:00:03:01,output:"s2-eth2"
 cookie=0x0, duration=93.077s, table=0, n_packets=8, n_bytes=784, priority=1,ip,nw_dst=10.0.12.2 actions=dec_ttl,mod_dl
_src:00:00:00:00:02:01,mod_dl_dst:00:00:00:00:01:02,output:"s2-eth1"
 cookie=0x0, duration=103.856s, table=0, n_packets=20, n_bytes=1592, priority=0 actions=CONTROLLER:65535
*** s3 ------------------------------------------------------------------------
 cookie=0x0, duration=103.842s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=94.104s, table=0, n_packets=9, n_bytes=882, priority=1,ip,nw_dst=10.0.67.2 actions=dec_ttl,mod_dl
_src:00:00:00:00:03:02,mod_dl_dst:00:00:00:00:06:01,output:"s3-eth2"
 cookie=0x0, duration=93.083s, table=0, n_packets=8, n_bytes=784, priority=1,ip,nw_dst=10.0.12.2 actions=dec_ttl,mod_dl
_src:00:00:00:00:03:01,mod_dl_dst:00:00:00:00:02:02,output:"s3-eth1"
 cookie=0x0, duration=103.842s, table=0, n_packets=23, n_bytes=1858, priority=0 actions=CONTROLLER:65535
*** s4 ------------------------------------------------------------------------
 cookie=0x0, duration=103.826s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=103.826s, table=0, n_packets=20, n_bytes=1592, priority=0 actions=CONTROLLER:65535
*** s5 ------------------------------------------------------------------------
 cookie=0x0, duration=103.803s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=103.803s, table=0, n_packets=20, n_bytes=1592, priority=0 actions=CONTROLLER:65535
*** s6 ------------------------------------------------------------------------
 cookie=0x0, duration=103.788s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=94.120s, table=0, n_packets=9, n_bytes=882, priority=1,ip,nw_dst=10.0.67.2 actions=dec_ttl,mod_dl
_src:00:00:00:00:06:03,mod_dl_dst:00:00:00:00:06:02,output:"s6-eth3"
 cookie=0x0, duration=93.099s, table=0, n_packets=8, n_bytes=784, priority=1,ip,nw_dst=10.0.12.2 actions=dec_ttl,mod_dl
_src:00:00:00:00:06:01,mod_dl_dst:00:00:00:00:03:02,output:"s6-eth1"
 cookie=0x0, duration=103.788s, table=0, n_packets=25, n_bytes=2022, priority=0 actions=CONTROLLER:65535
```

Figure 14: Flow rules installed by the L3 controller on all switches.

The flow rules installed by the controller confirm the L3 routing logic across the computed shortest path (s1 → s2 → s3 → s6). No rules were installed on s4 and s5 as they are not on this path.

- **Switch s1 (First Hop):** s1 has two primary rules. One rule forwards traffic destined for h2's IP (10.0.67.2) to s2 via port 's1-eth2'. The other rule handles the return traffic for h1's IP (10.0.12.2) by forwarding it to h1 via port 's1-eth1'. Both rules include actions to decrement TTL and rewrite the MAC headers.
- **Switches s2 & s3 (Intermediate Hops):** As intermediate switches, s2 and s3 act as pure routers. They have bidirectional rules to forward traffic based on destination IP. For instance, s2 forwards traffic for h2's subnet towards s3 ('output:"s2-eth2"') and traffic for h1's subnet back towards s1 ('output:"s2-eth1"'). Critically, they rewrite the source and destination MAC addresses at each hop and decrement the TTL.
- **Switch s6 (Final Hop):** s6 is the final hop before h2. It has a rule to forward traffic for h2's IP out of port 's6-eth3', which is directly connected to h2. It also has a rule to forward the return traffic from h2 back towards s3 ('output:"s6-eth1"') to reach h1's subnet.

Assumptions

The implementation of our L3 controller was based on several key assumptions:

- **Reliable Control Plane:** The controller is assumed to have a stable and uninterrupted connection to all switches.
- **Static Topology:** The network topology is considered static for the duration of a flow. The controller in this part does not handle dynamic link failures or recoveries.
- **Correct Host Configuration:** All hosts are assumed to be correctly configured with a default gateway IP, enabling them to initiate traffic to external subnets.
- **Symmetrical Link Costs:** Link costs are non-negative and symmetrical (cost A→B is the same as B→A), a requirement for Dijkstra's algorithm to function correctly.
- **IPv4/ARP Focus:** The controller is designed to handle only ARP and IPv4 traffic. Any other protocol type would trigger a table-miss and be handled by the default controller rule, but not explicitly routed.

4 Part 4: Comparison with Traditional Routing (OSPF)

In this final part, we compared the performance of our custom L3 SDN controller against a traditional routing protocol, OSPF, focusing on convergence time during a link failure event.

Warm-up Experiment

First, we ran a baseline `iperf` test between h1 and h2 under normal conditions for both setups.

- **SDN Controller:** Achieved a steady throughput near the link capacity.
- **OSPF Network:** After waiting for OSPF to converge, it also achieved a similar steady throughput.

This confirmed that both systems correctly identified and utilized the optimal path when the network was stable. The forwarding rules on the SDN switches and the routing tables on the OSPF routers both pointed to the same shortest path.

Comparison Under Link Failure

The core of the experiment involved running a 30-second `iperf` test from h1 to h2 while inducing a link failure. The primary link (s1-s2) was brought down 2 seconds into the test and brought back up 5 seconds later (at the 7-second mark).

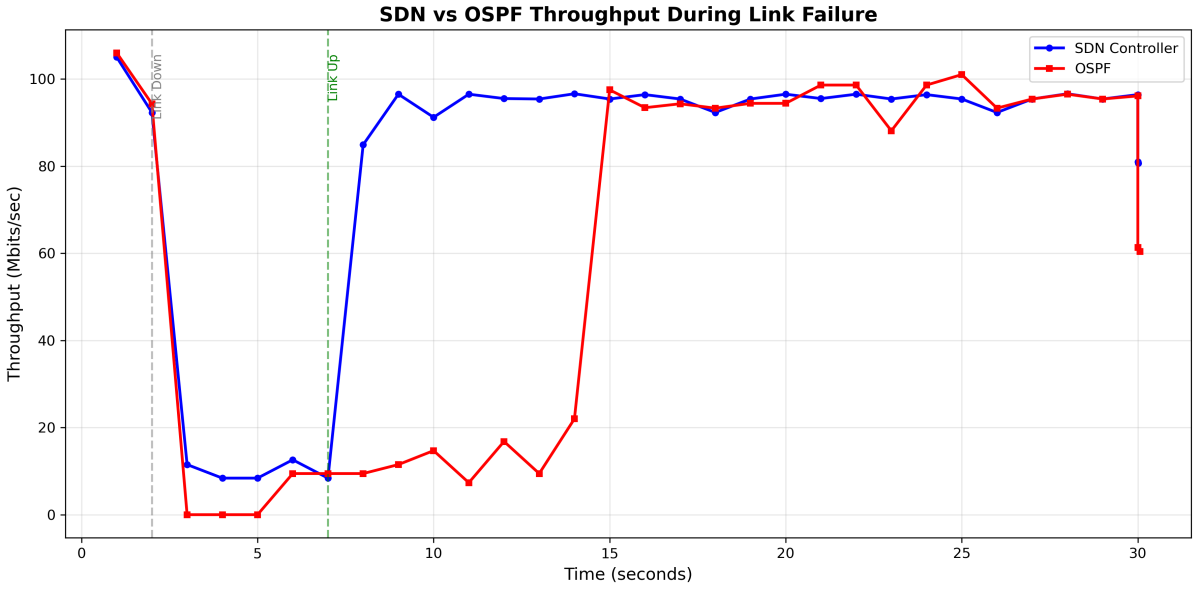

Figure 15: Throughput vs. Time for SDN and OSPF during a link failure event.

Analysis of Results

The graph in Figure 15 clearly illustrates the performance differences. We distinguish between two key metrics:

- **Control Plane Convergence:** The time required for the network's routing logic to stabilize after a topology change. For OSPF, this involves exchanging Link State Advertisements (LSAs).
- **Data Plane Convergence:** The user-perceived time until application traffic is successfully flowing along the new path. This is the most critical real-world metric.

Our analysis of the experiment yielded the following results:

- **SDN Controller:**
  - **Link Down (t=2s):** The SDN controller exhibits extremely fast convergence. The Ryu controller's link discovery service immediately detected the link failure. It took only a fraction of a second for the controller to re-calculate the new shortest path and push the updated flow rules to all affected switches. As seen in the graph, throughput drops to near zero but recovers to the backup path's capacity in about **1 second**.
  - **Link Up (t=7s):** When the original link was restored, the controller again detected the topology change. It recalculated the path (preferring the now-available, higher-bandwidth primary path) and updated the flow rules. The graph shows that the SDN setup seamlessly switched back to the high-capacity path, with throughput returning to 100 Mbps almost instantly.
- **OSPF:**
  - **Link Down (t=2s):** The OSPF network shows significantly slower convergence. After the link failed, the routers first had to detect the failure (based on

dead timers), then flood LSAs to inform their neighbors, and finally re-run the SPF algorithm to compute a new route. This distributed process took several seconds. The graph shows that OSPF traffic flatlined for approximately **4-5 seconds** before a new path was established and traffic began to flow again, albeit at the lower capacity of the backup path.

– **Link Up (t=7s):** The recovery was even slower. When the link came back up, OSPF routers again had to exchange LSAs to advertise the restored link. There is a noticeable delay in the graph from t=7s until approximately t=13s before the throughput returns to the high-capacity path. This delay is due to the inherent timers and convergence mechanisms of a distributed protocol.

Conclusion

This experiment highlights the superior agility of the SDN architecture. The SDN controller's centralized view and ability to deterministically install new paths allows for an order-of-magnitude faster recovery from link failures compared to OSPF. Our measured data plane convergence for SDN was around 1 second, whereas OSPF took several seconds to recover. The distributed, timer-based nature of OSPF introduces significant latency in both control plane stabilization and subsequent data plane updates, leading to longer periods of network disruption and suboptimal performance during topology changes.