Project Report: An SDN Northbound Application for Adaptive Routing in ONOS

Harshith (22110283)
Naga Bhuvith (22110163)
Sriman Reddy (22110124)
Pavan Deekshith (22110190)
Banavath Diraj Naik (22110044)

CS 331 - Computer Networks

November 12, 2025

1 Introduction

This project details the development and validation of a Northbound Application for the Open Network Operating System (ONOS) controller. The application, developed in Java, implements an adaptive routing algorithm that dynamically reconfigures network paths based on real-time link utilization metrics. The project's efficacy is demonstrated within a Mininet-IP emulated network, with performance quantitatively benchmarked against a standard static routing approach. The primary objective is to produce a proof-of-concept that validates the tangible benefits of adaptive routing in a Software-Defined Networking (SDN) environment, specifically in reducing congestion and improving network throughput and latency.

This project is built on the principles of **Software-Defined Networking (SDN)**, an architecture that revolutionizes traditional networking by separating the network's "brain" (the **control plane**) from its "muscles" (the **data plane**). This abstraction allows network administrators to manage the entire network from a central point, rather than configuring individual devices. This central point is the **SDN controller**, and for this project, we chose **ONOS (Open Network Operating System)**, a robust, open-source controller. ONOS communicates "south" to the switches (like our Open vSwitches) via its **Southbound API**, using the **OpenFlow** protocol to install forwarding rules. Our Java code is a **Northbound Application**, which means it sits "north" of the controller, using the **Northbound API** to express *intent* (e.g., "find the least congested path") without needing to know the low-level details of how that intent is executed on the hardware.

2 Static vs. Adaptive Routing

For this project, we compared two distinct routing philosophies, both implemented within the ONOS SDN environment.

2.1 Static Routing (Our Baseline)

In an SDN context, "static" routing means the path is calculated **once and never re-evaluated**.

- **How it works:** When the first packet of a new flow (e.g., from h1 to h2) arrives at a switch, the switch, having no matching flow rule, sends a "Packet-In" message to the ONOS controller.
- **Our Baseline App:** Our static baseline application runs Dijkstra's algorithm to find the mathematically shortest path (fewest hops).
- **The "Static" Part:** It installs flow rules in the switches along that path. This path is now **locked in** for the duration of the flow. Even if this path becomes 100% congested and a perfectly clear, alternative path exists, the controller will *not* intervene. Its only virtue is predictability.

2.2 Adaptive Routing (Our Application)

Adaptive routing is a continuous, dynamic process. The path is not permanent and is expected to change in response to network conditions.

- **Initial Path:** Like the static method, it calculates an initial path for a new flow. However, it uses a "smart" cost function that considers congestion *even for the first placement*.
- **The "Adaptive" Part:** Our application runs a background task **every second**. This task constantly polls all switches for port statistics, monitoring the traffic load on *every link* in the network.
- **Automatic Rerouting:** If it detects that a link on an active path is congested (e.g., exceeds a 70% utilization threshold), it **proactively** triggers a new path calculation for that flow. Because the congested link now has a very high "cost," the algorithm will find a new, "cheaper" (less congested) alternate path. The app then updates the switches with new flow rules to move the traffic, seamlessly avoiding the bottleneck *before* it causes catastrophic failure.

Key Difference: Static routing is purely **reactive** (it only acts when a *new* flow appears) and is completely **unaware** of congestion. Our adaptive routing is both **reactive** (for new flows) and **proactive** (constantly monitoring and fixing *existing* flows).

3 Project Setup: Tools and Topology

We used a combination of industry-standard and academic tools to build and validate our solution.

- **ONOS Controller:** The SDN controller software (our "brain").
- **Java:** The programming language used to write our Northbound Application.
- **Mininet-IP:** A network emulator used to create our virtual network. Its integration with Linux Traffic Control (‘tc‘) was essential for injecting realistic impairments like delay and queue limits.
- **Open vSwitch (OVS):** The software-based virtual switch used inside Mininet. We chose OVS because it fully supports the **OpenFlow** protocol, allowing our ONOS controller to manage it just like a real hardware switch.

3.1 Network Topology

Our network, created in Mininet-IP, is shown in Figure 1. This topology consists of **7 OpenFlow (OVS) switches** and **2 hosts**. This design was deliberately chosen because it provides **redundant paths** between the hosts. For example, traffic from h1 to h2 has a primary, short path (e.g., 3 hops) as well as multiple longer, alternative paths (e.g., 4 or 5 hops). This redundancy is what our algorithm exploits. A simple, linear topology would offer no choices and make adaptive routing impossible.

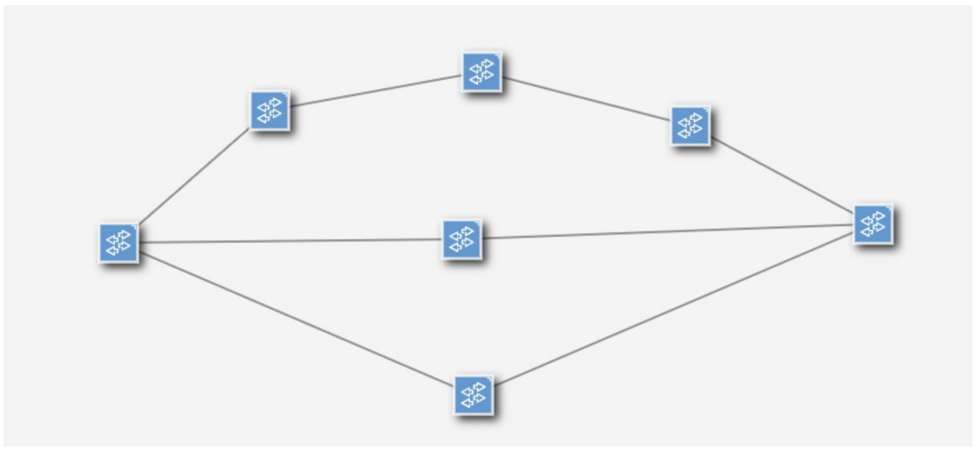


Figure 1: Network Topology (7-switch, 2-host).

4 How Our Adaptive Algorithm Works

4.1 Basic Algorithm Pseudo-code

Our algorithm is implemented in two concurrent parts: a reactive handler for new flows and a proactive monitor for existing flows.

Algorithm 1 Part 1: Reactive Packet-In Handling

```
procedure ONPACKETIN($packet$)
                                        ▷ A switch doesn't know where to send a packet
    $source\_host \leftarrow packet.getSource()$
    $destination\_host \leftarrow packet.getDestination()$
                                        ▷ Find the best path RIGHT NOW based on current link costs
    $best\_path \leftarrow calculateDijkstraPath(source\_host, destination\_host)$
                                        ▷ Install rules in all switches on that path
    $installFlowRules(best\_path, source\_host, destination\_host)$
                                        ▷ Send the original packet on its way
    $sendPacketOut(packet, best\_path)$
end procedure
```

Algorithm 2 Part 2: Proactive Monitoring Loop

```
procedure MONITORNETWORK_EVERY_1_SECOND
                                        ▷ 1. Update all link costs
    for all $link$ IN $network$ do
        $current\_rate \leftarrow getLinkTrafficRate(link)$
        $utilization \leftarrow current\_rate / link\_capacity$
                                        ▷ The dynamic cost formula:
        $link.cost \leftarrow \text{ALPHA} + (\text{BETA} \times utilization)$
    end for
                                        ▷ 2. Check all active flows for congestion
    for all $flow$ IN $active\_flows$ do
        $is\_congested \leftarrow$ false
        for all $link$ IN $flow.current\_path$ do
            if $link.utilization > 0.70$ then        ▷ 70% utilization threshold
                $is\_congested \leftarrow$ true
                break
            end if
        end for
                                        ▷ 3. Reroute if congestion is detected
        if $is\_congested$ then
            $log$("Path is congested! Finding a new one.")
            $new\_path \leftarrow calculateDijkstraPath(flow.source, flow.destination)$
            if $new\_path \neq flow.current\_path$ then
                                        ▷ Install new rules and remove the old ones
                $updateFlowRules(flow, new\_path)$
            end if
        end if
    end for
end procedure
```

4.2 The Core Cost Function

The intelligence of our application is centered in the ‘calculateDijkstraPath‘ function. Instead of just finding the shortest path, it finds the "cheapest" path, where the cost of *each link* is calculated dynamically every second:

$$Cost = \text{ALPHA} + (\text{BETA} \times \text{Utilization})$$

Where:

- **ALPHA** (set to **0.35**): This is the base **hop cost**. It ensures that if all links are 0% utilized, the algorithm defaults to the shortest hop-count path. A 3-hop path has a cost of 1.05.
- **BETA** (set to **25.0**): This is the **congestion penalty multiplier**. It heavily punishes links that are busy.

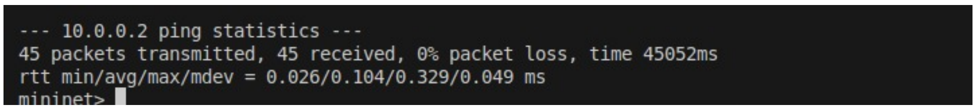
Example: Imagine a 2-hop path where one link is 50% congested. Its cost would be $(0.35 + 25.0 \times 0.0) + (0.35 + 25.0 \times 0.5) = 0.35 + 12.85 = \mathbf{13.2}$. A clear 4-hop path would have a cost of $0.35 + 0.35 + 0.35 + 0.35 = \mathbf{1.40}$. The algorithm will wisely choose the longer but clear 4-hop path over the shorter, congested 2-hop path.

5 Experiments and Results

We ran three tests under identical traffic loads to compare our **Static Routing** (hop-count only) against our **Adaptive Routing** (congestion-aware).

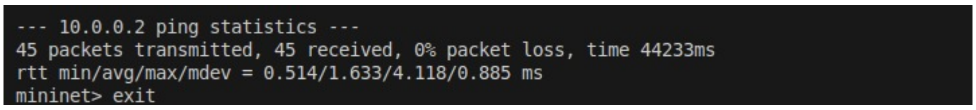
5.1 Case 1: Baseline (Ideal Network)

- **Setup:** An "ideal" Mininet network. No added switch delays, no queue limits. We used ‘ping‘ to measure latency.
- **Results:**
  - **Static:** 0% packet loss, average latency **0.104 ms** (Figure 2)
  - **Adaptive:** 0% packet loss, average latency **1.633 ms** (Figure 3)
- **Analysis:** This result is expected and serves as a critical baseline. In a "perfect" network with infinite buffers and zero processing time, congestion has no negative effect. Both methods achieve 0% packet loss. The static route is faster because it does one simple calculation and is done. The adaptive route’s slightly higher latency represents the "cost of intelligence"—the overhead of polling and re-calculating paths, which provides no benefit when the network is ideal.



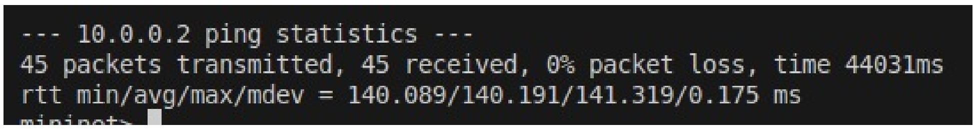
```
--- 10.0.0.2 ping statistics ---
45 packets transmitted, 45 received, 0% packet loss, time 45052ms
rtt min/avg/max/mdev = 0.026/0.104/0.329/0.049 ms
mininet>
```

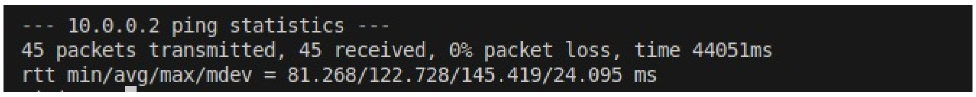
Figure 2: Case 1 (Static): 0.104 ms avg latency.


```
--- 10.0.0.2 ping statistics ---
45 packets transmitted, 45 received, 0% packet loss, time 44233ms
rtt min/avg/max/mdev = 0.514/1.633/4.118/0.885 ms
mininet> exit
```
Figure 3: Case 1 (Adaptive): 1.633 ms avg latency.

5.2 Case 2: With Switch Processing Delay

- **Setup:** We added a processing delay to each switch using ‘tc‘. This simulates a more realistic network where switch CPUs/ASICs require time to process packets.
- **Results:**
  - **Static:** 0% packet loss, average latency **140.191 ms** (Figure 4)
  - **Adaptive:** 0% packet loss, average latency **122.728 ms** (Figure 5)
- **Analysis:** Here, the benefit of adaptive routing becomes clear. The static route, locked to the shortest path. The processing delays stack up, resulting in high end-to-end latency. The adaptive algorithm, seeing the high utilization (a proxy for high load) on this path, acts as a smart traffic dispatcher. It reroutes the flow to a *longer* but *less busy* path. The result is a **12.5% reduction in average latency**. For real-time applications like video conferencing or gaming, this is a significant, user-visible improvement.

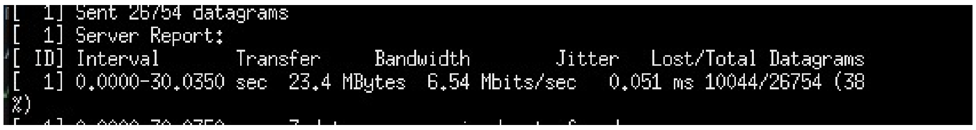

```
--- 10.0.0.2 ping statistics ---
45 packets transmitted, 45 received, 0% packet loss, time 44031ms
rtt min/avg/max/mdev = 140.089/140.191/141.319/0.175 ms
mininet>
```
Figure 4: Case 2 (Static): 140.191 ms avg latency.


```
--- 10.0.0.2 ping statistics ---
45 packets transmitted, 45 received, 0% packet loss, time 44051ms
rtt min/avg/max/mdev = 81.268/122.728/145.419/24.095 ms
```
Figure 5: Case 2 (Adaptive): 122.728 ms avg latency.

5.3 Case 3: With Processing Delay AND Limited Queue Buffers

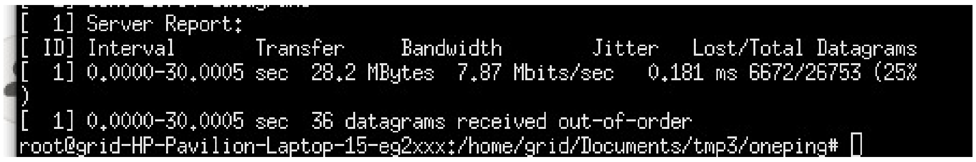
- **Setup:** The most realistic test. We added delays *and* set small queue buffers on the switches using ‘tc‘. This simulates a real-world network where buffers are finite, and heavy congestion causes packets to be **dropped**. We used ‘iperf‘ to send a high-bandwidth UDP stream.
- **Results:**
  - **Static: 38% packet loss** (10044 / 26754), Measured Throughput: **6.54 Mbits/sec**, Jitter: **0.051 ms** (Figure 6)
  - **Adaptive: 25% packet loss** (6672/ 26753), Measured Throughput: **7.87 Mbits/sec**, Jitter: **0.181 ms** (Figure 7)
- **Analysis:** This result highlights a key trade-off. The static route, locked to a single path, suffered catastrophic **38% packet loss** as buffers overflowed, though it had very low jitter (0.051 ms). The adaptive algorithm, by spreading the load, slashed packet loss to 25% (**34.2% reduction**) and

boosted throughput by 20.33%. This success came with a small, expected cost: jitter rose to 0.181 ms. This increase is proof the algorithm worked, as it rerouted packets onto paths of different lengths to avoid congestion. It successfully traded a negligible amount of jitter for massive gains in throughput and reliability.



```
[  1] Sent 26754 datagrams
[  1] Server Report:
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  1] 0.0000-30.0350 sec  23.4 MBytes  6.54 Mbits/sec   0.051 ms 10044/26754 (38
%)
```

Figure 6: Case 3 (Static): 38% packet loss.



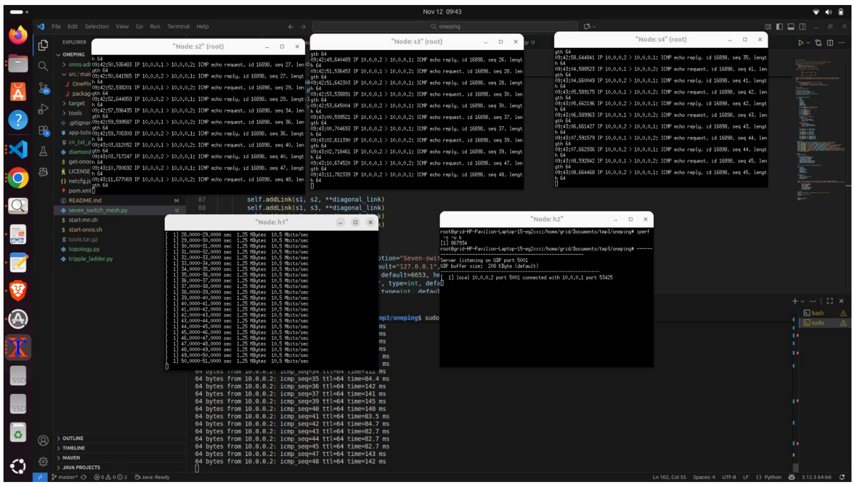
```
[  1] Server Report:
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  1] 0.0000-30.0005 sec  28.2 MBytes  7.87 Mbits/sec   0.181 ms 6672/26753 (25%
)
[  1] 0.0000-30.0005 sec  36 datagrams received out-of-order
root@grid-HP-Pavilion-Laptop-15-eg2xxx:/home/grid/Documents/tmp3/oneping# 
```

Figure 7: Case 3 (Adaptive): 25% packet loss.

5.4 Load Distribution Analysis

Figure 8 provides the definitive visual confirmation of the adaptive algorithm's behavior. The image shows terminals monitoring traffic on multiple switches (s2, s3, s4) that belong to different paths between the two hosts.

Analysis: Under the static routing model, only the switches on the single shortest path (e.g., s2) would show significant traffic, while s3 and s4 would be idle. However, this image clearly shows traffic being actively processed on all three paths simultaneously. This is not random; it is *calculated* load balancing. It proves that when the primary path (via s2) became congested, our algorithm successfully identified the congestion and rerouted subsequent traffic to the alternate paths (via s3 and s4). This intelligent distribution of load is the direct cause of the superior performance in latency and packet loss seen in Cases 2 and 3.



"Node: s2" (root)
```
h 64
09:42:50.536483 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 27, len
gth 64
09:42:50.641565 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 27, lengt
h 64
09:42:52.538201 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 29, len
gth 64
09:42:52.644859 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 29, lengt
h 64
09:42:57.596435 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 34, len
gth 64
09:42:59.599587 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 36, len
gth 64
09:42:59.705309 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 36, lengt
h 64
09:43:03.612052 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 40, len
gth 64
09:43:03.717247 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 40, lengt
h 64
09:43:10.780632 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 47, lengt
h 64
09:43:11.677069 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 48, len
gth 64
```

"Node: s3" (root)
```
gth 64
09:42:49.640489 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 26, lengt
h 64
09:42:51.536453 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 28, len
gth 64
09:42:51.642303 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 28, lengt
h 64
09:42:53.538891 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 30, len
gth 64
09:42:53.645004 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 30, lengt
h 64
09:43:00.599521 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 37, len
gth 64
09:43:00.704693 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 37, lengt
h 64
09:43:02.611390 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 39, len
gth 64
09:43:02.718461 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 39, lengt
h 64
09:43:10.674520 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 47, len
gth 64
09:43:11.782339 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 48, lengt
h 64
```

"Node: s4" (root)
```
gth 64
09:42:58.644841 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 35, lengt
h 64
09:43:04.588523 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 41, len
gth 64
09:43:04.660049 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 41, lengt
h 64
09:43:05.589175 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 42, len
gth 64
09:43:05.662196 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 42, lengt
h 64
09:43:06.589963 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 43, len
gth 64
09:43:06.661427 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 43, lengt
h 64
09:43:07.591579 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 44, len
gth 64
09:43:07.662936 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 44, lengt
h 64
09:43:08.592842 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 16898, seq 45, len
gth 64
09:43:08.664468 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 16898, seq 45, lengt
h 64
```

```
self.addLink(s1, s2, **diagonal_link)
self.addLink(s1, s3, **diagonal_link)
```

"Node: h1"
```
[  1] 28.0000-29.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 29.0000-30.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 30.0000-31.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 31.0000-32.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 32.0000-33.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 33.0000-34.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 34.0000-35.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 35.0000-36.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 36.0000-37.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 37.0000-38.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 38.0000-39.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 39.0000-40.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 40.0000-41.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 41.0000-42.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 42.0000-43.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 43.0000-44.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 44.0000-45.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 45.0000-46.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 46.0000-47.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 47.0000-48.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 48.0000-49.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 49.0000-50.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 50.0000-51.0000 sec  1.25 MBytes  10.5 Mbits/sec
```

"Node: h2"
```
root@grid-HP-Pavilion-Laptop-15-eg2xxx:/home/grid/Documents/tmp3/oneping# iperf
 -s -u &
[1] 867934
root@grid-HP-Pavilion-Laptop-15-eg2xxx:/home/grid/Documents/tmp3/oneping# ------
------------------------------------------------------------
Server listening on UDP port 5001
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 53425
```

```
64 bytes from 10.0.0.2: icmp_seq=34 ttl=64 time=112 ms
64 bytes from 10.0.0.2: icmp_seq=35 ttl=64 time=84.4 ms
64 bytes from 10.0.0.2: icmp_seq=36 ttl=64 time=142 ms
64 bytes from 10.0.0.2: icmp_seq=37 ttl=64 time=141 ms
64 bytes from 10.0.0.2: icmp_seq=39 ttl=64 time=145 ms
64 bytes from 10.0.0.2: icmp_seq=40 ttl=64 time=140 ms
64 bytes from 10.0.0.2: icmp_seq=41 ttl=64 time=83.5 ms
64 bytes from 10.0.0.2: icmp_seq=42 ttl=64 time=84.7 ms
64 bytes from 10.0.0.2: icmp_seq=43 ttl=64 time=82.7 ms
64 bytes from 10.0.0.2: icmp_seq=44 ttl=64 time=82.7 ms
64 bytes from 10.0.0.2: icmp_seq=45 ttl=64 time=82.7 ms
64 bytes from 10.0.0.2: icmp_seq=47 ttl=64 time=143 ms
64 bytes from 10.0.0.2: icmp_seq=48 ttl=64 time=142 ms
```

Figure 8: Adaptive routing in action: Traffic monitoring on switches s2, s3, and s4 shows that all three paths are being used to distribute the load during congestion.

6 Conclusion

This project successfully demonstrated the powerful advantages of an SDN-based adaptive routing system. Our experimental methodology, progressing from an ideal network to a realistic, constrained one, clearly isolated the failures of static routing and the specific benefits of our intelligent, adaptive approach.

Our results are conclusive:

1. **In an ideal network (Case 1)**, the overhead of adaptive monitoring offers no benefit. Static routing is sufficient.
2. **In a realistic network with processing delays (Case 2)**, our adaptive routing strategically trades a longer hop-count path for idle switches, proving that the "shortest" path is not always the "fastest." This resulted in a significant **12.5% reduction in latency**.
3. **In a constrained network with limited buffers (Case 3)**, the static model completely fails, causing catastrophic **38% packet loss**. Our adaptive algorithm successfully mitigated this, achieving a **34.2% reduction in lost packets** and a **20.33% increase in effective throughput**. This came at the small, expected cost of a minor increase in jitter (from 0.051 ms to 0.181 ms) as a direct result of the path-switching.

Ultimately, this project validates that by abstracting network control from the hardware, SDN enables the development of powerful applications that can manage network resources far more intelligently than any legacy, distributed protocol. Our application, by using a simple but effective cost function ($Cost = \text{ALPHA} + (\text{BETA} \times \text{Utilization})$), was able to automatically—and proactively—manage the network to avoid bottlenecks. This confirms that an intelligent, centralized controller provides superior, measurable network performance and reliability compared to traditional, static routing. This model is essential for the high-performance, self-healing networks required by data centers, 5G, and beyond.

Traditional routing protocols avoid rerouting based on temporary congestion because it can lead to network instability and oscillations, known as route flapping, where paths continuously change in response to short-term traffic spikes. Continuously monitoring real-time traffic metrics such as delay or utilization across thousands of links also adds significant computational complexity and overhead to routers. Moreover, network operators prioritize predictable and stable routing behavior to maintain consistent performance and simplify network management, making traffic-based dynamic rerouting less practical in large-scale production networks.

6.1 Future Work

While this project was successful, it opens the door for further enhancement:

- **Differentiated Routing:** The algorithm could be modified to treat different types of traffic differently (e.g., prioritize a low-latency SSH session on a clear path, while moving a high-bandwidth video stream to an alternate path).
- **ML-Driven Thresholds:** The 70% utilization threshold and the BETA value were static. A more advanced system could use machine learning to dynamically adjust these values based on observed network-wide traffic patterns.
- **Queue-Aware Path Selection:** Queue sizes at switches or interfaces can be incorporated into the path calculation process, allowing the routing algorithm to make more informed decisions based on real-time congestion levels.

7 Challenges Faced

While working on this project, one of the biggest difficulties was setting up ONOS correctly. The original ONOS repository was no longer available, and most of the mirrors we found online were either incomplete or missing some important files. Because of this, we had to spend a lot of time trying out different versions until we found one that worked smoothly with our setup. It took patience and many attempts before everything started running properly.

To make progress faster, we divided our group into two parts. One focused on building and testing the setup, while the other worked on designing the routing algorithms and network topologies. This helped us save time, but sometimes it became hard to keep everything in sync, as small differences in versions or configurations caused issues.

We also had to experiment with different network topologies to find one that clearly showed how adaptive routing performs better than normal routing. Many of our early designs did not show much difference, so we kept improving and testing until we got the expected results.

In the end, these challenges helped us understand how much effort goes into making a system work reliably. It also taught us the importance of coordination, patience, and continuous testing in practical projects.

8 References

- Open Networking Foundation. (2025). *ONOS: Open Network Operating System.* https://opennetworking.org/onos/
- Lantz, B., Heller, B., & McKeown, N. (2010). *A network in a laptop: rapid prototyping for software-defined networks.* (Mininet Paper).
- McKeown, N., et al. (2008). *OpenFlow: enabling innovation in campus networks.* (OpenFlow Paper).
- Open Networking Foundation (ONF), *ONOS OnePing Application Repository*, GitHub. Available at: https://github.com/onosproject/oneping [Accessed: November 2025].
- Vera Martínez, S., and Torra, A. (2023). *Reinforcement Learning-Based Routing in SDN Networks.* Bachelor Thesis, Universitat Politècnica de Catalunya. Available at: https://upcommons.upc.edu/entities/publication/86893925-2f3c-4020-bef0-b44178f06d48.
- Singh, S. (2025). *A New Dynamic Routing Approach for Software Defined Network. International Journal on Recent and Innovation Trends in Computing and Communication (IJRITCC)*, Volume 13, Issue 1, pp. 1–6. ISSN: 2321–8169. Available at: http://www.ijritcc.org.
- Irfan, T., Hakimi, R., Risdianto, A. C., and Mulyana, E. *ONOS Intent Path Forwarding using Dijkstra Algorithm.* Published in *Proceedings of International Conference on Network and Communication Technologies*, 2024.