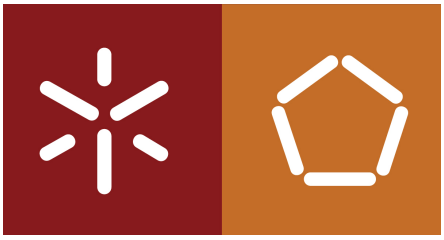

Redes Definidas Por Software

Data Plane Programming

Universidade do Minho

June 2025

Authors:

Carlos Silva	PG57518
Augusto Campos	PG57510
Amneh Al-Issa	E12305

Group 6

1 Introduction

Modern days require architectures that are software based, and thats where the need for this Software Defined Networks course came from. In this recent architecture, the **data plane** and **control plane** are disconnected, apart from eachother, to enhance flexibility and custom-ability . This assignment shows and teaches us the principles by having us implement a network using **P4**, a language where we can program ourselves the forwarding behavior of packets, and a custom **Label Switching Protocol (MSLP)** to emulate tunneling and **stateful** packet filtering.

Data Plane

The **data plane** is responsible for handling and processing live network traffic. It executes the logic defined in P4 programs to parse packets, apply rules, modify headers, etc... and decides where and how to forward each of the arriving packets. No intelligence is attributed to this modules. In this project, the data plane is implemented using P4, where each of the different "groups" of devices will be appointed different P4 files, depending on their role, like label-based forwarding, or firewall capabilities.

Control Plane

The **control plane** manages the behavior of the data plane. It makes decisions about routing and rule installation, and instructs the data plane via a protocol such as *P4Runtime*, using gRPC connections. In this project, the control plane is implemented using a Python-based controller that dynamically populates the tables across all network devices. It handles:

- MAC address learning in the L2 switch
- Forwarding rule installation for label-switched tunnels
- Firewall configuration and Bloom filter direction logic
- Tunnel load balancing using a dedicated thread

P4

As said before, is a language for programming packet processors. It lets us define custom packet parsing logic, header formats and match-action table. In our own project, four main P4 programs are used, each for different "kinds" of Routers/Switches

- `l2switch.p4`, for implementing MAC learning
- `ingress.p4`, for tunnel ingressing and egressing and label stack insertion at router from the left to the right
- `label_forwarder.p4`, for label-based forwarding at the intermediate routers
- `egress_firewall.p4`, for stateful firewalling and both ingress and egress at the router, from right to left

Label Switching Protocols

Label switching is a forwarding method where packets are routed based on labels instead of the usual IP addresses. This is most commonly used in *Multiprotocol Label Switching (MSLP)* as it has way better forwarding. In this project, a custom **MSLP** (My Sequence Label Protocol) is implemented.

- Pushes a label stack at one of the Edges Routers (R1 or R4) to encode a tunnel path
- Pops one label at each intermediate router (R2–R3, R6–R5)
- Guides forwarding decisions based on the current top label

This label-based routing enables multiple tunnel paths to be defined and chosen as we, the admin, would like

2 Implementation

Topology

We used **Mininet** to set up the network topology, which includes six routers (R1 to R6), one switch (S1), four hosts (h1 to h4). There are **two tunnels** (Tunnel #1 and Tunnel #2) connecting the two subnets: 10.0.1.0/24 – h1,h2,h3 – and 10.0.2.0/24 – h4 –.

My Sequence Label Protocol (MSLP)

Header Format, Fields and FieldSize The MSLP header carries up to **three 16-bit labels** stacked together. These labels are used to tunnel the packets inside the network. The protocol will use **(EtherType: 0x88B5)** to identify MSLP packets. The `bos` (Bottom of Stack) bit indicates whether the label is the last in the stack (1 = last label, 0 = more labels follow). The padding ensures proper alignment.

```
header label_t {
    bit<16> label;
    bit<1>  bos;
    bit<7>  padding; }
```

```
struct headers {
    ethernet_t     ethernet;
    ipv4_t         ipv4;
    label_t[3]     mslp_labels; }
```

Tunnel Selection Tunnel selection is handled by the **controller**, which dynamically assigns packets to one of the two tunnels using a round-robin mechanism. This is implemented with a thread that periodically updates the default action in the `tunnel_label_selector` table of the edge routers (`r1` and `r4`). This process enables **load balancing** and traffic distribution across the network.

```
// Toggle between tunnel1 and tunnel2
    if current_action == "set_labels_tunnel1":
        current_action = "set_labels_tunnel2"
    else:
        current_action = "set_labels_tunnel1"
```

Details on the controller's logic will be further discussed in the next sections.

P4 Actions and Tables We implemented **three primary P4 programs**, as the 4th one was given to us in one of the tasks. And we will now describe each of these in detail, including all the P4 actions and tables they have, one by one.

1. ingress.p4: This program handles tunnel ingress, MSLP label stack insertion, and basic label-based forwarding. The ingress logic identifies IPv4 packets, rewrites their EtherType to MSLP, inserts a three-label stack depending on the selected tunnel, and updates MAC headers to forward the packet toward the correct egress port. The processing logic is organized through a sequence of actions and tables, as outlined below:

- `set_labels_tunnel1()` and `set_labels_tunnel2()` are the core actions responsible for pushing three MSLP labels onto the packet. The labels are written from the bottom of the stack (BoS = 1) to the top (BoS = 0), as shown below:

```
// Tunnel 1 example
hdr.mslp_labels[2] = {0x4010, 1, 0}; // BoS label
hdr.mslp_labels[1] = {0x3020, 0, 0};
hdr.mslp_labels[0] = {0x2020, 0, 0};

hdr.mslp_labels[x].setValid();
```

  This process effectively encodes the tunnel path through the network. Tunnel 2 uses a different sequence of labels: `0x6020`, `0x5020`, and `0x4010`.

- `tunnel_label_selector` is a table that determines which label-setting action to apply. Currently, the selection is made using the IPv4 destination address as a fake key/ dummy key, and defaults to Tunnel 1.

- `rewriteMacsForTunnel()` updates the Ethernet source and destination MAC addresses according to the output port and next hop, enabling proper forwarding at the L2 level.

```
action rewriteMacsForTunnel(macAddr_t srcAddr, macAddr_t dstAddr) {
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ethernet.srcAddr = srcAddr;
    }
```

- `forTunnelMacrewrite` is a supporting table that applies the MAC rewrite logic based on the `egress_spec` field.

- For **incoming MSLP packets**, the ingress pipeline detects the EtherType and pops the top label using the 'pop_front(1)' operation. Once labels are processed and removed, the EtherType is reset to IPv4 and the packet is forwarded using a standard `ipv4Lpm` table. If a match occurs, MAC rewriting is performed again using `internalMacLookup`.

In the **apply** block of the `MyIngress` control, packet processing follows the logic below:

1. **IPv4 Packets (Tunnel Ingress):**
   - EtherType is set to MSLP.
   - `tunnel_label_selector` applies a label stack (currently defaults to Tunnel 1).
   - `forTunnelMacrewrite` rewrites MACs for outgoing forwarding.
2. **MSLP Packets (Tunnel Transit/Egress):**
   - The top label is removed using `pop_front(1)`.
   - EtherType is reset to IPv4.
   - The packet is matched in the LPM table and MAC headers are updated before delivery.
3. **Other Packets:** All other types are dropped.

The packet processing logic can be summarized in the following flowchart:

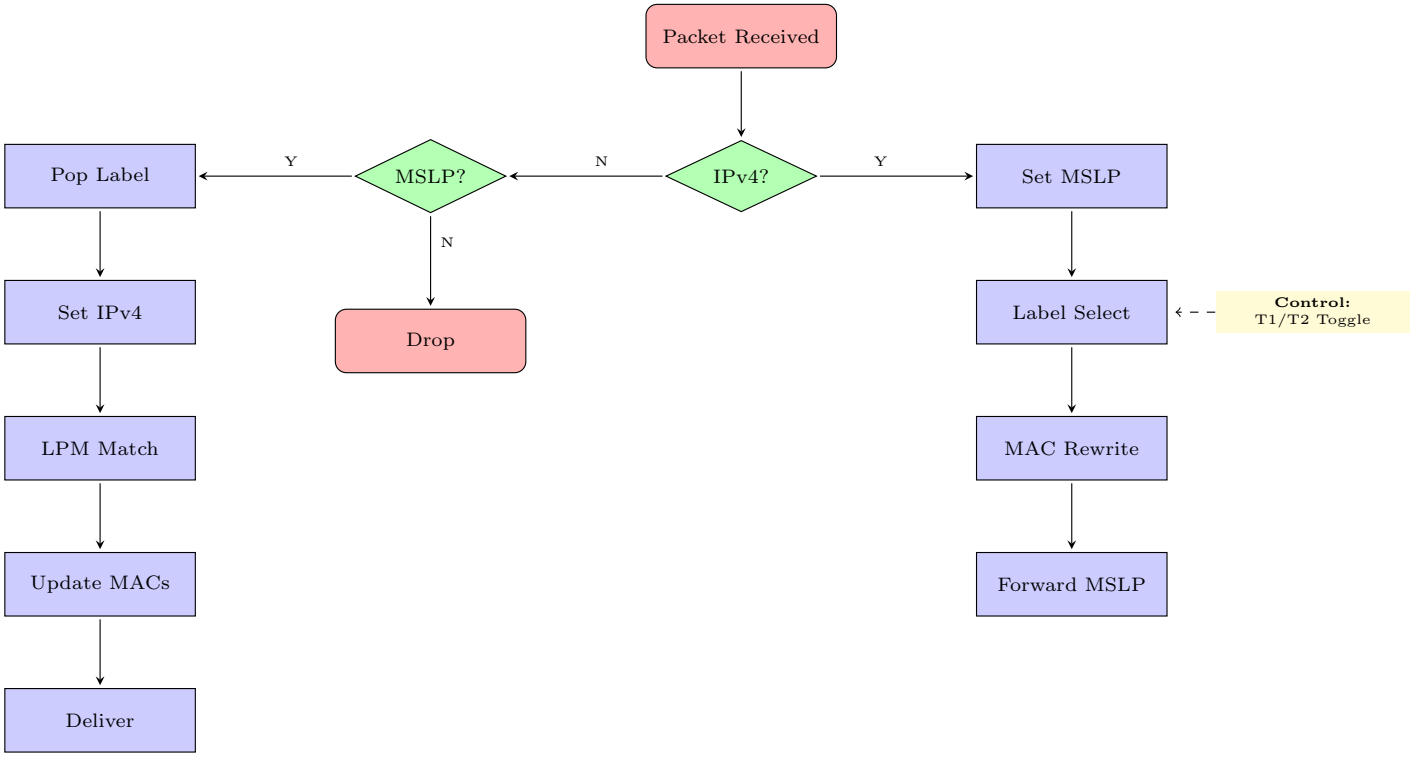

Figure 1: Packet processing logic.

2. label_forwarder.p4: This P4 program is used by the **intermediate routers: R2, R3, R5, R6** in the MSLP tunnels. These routers read the top MSLP label, pop it, and forward the packet to the next hop based on that label. The action `pop_and_forward` is responsible for popping the labels, updating MAC addresses, and setting the egress port.

```
action pop_and_forward(bit<9> port, macAddr_t dst_mac, macAddr_t src_mac) {
    hdr.mslp_labels.pop_front(1); // Pop top label
    hdr.ethernet.dstAddr = dst_mac;
    hdr.ethernet.srcAddr = src_mac;
    standard_metadata.egress_spec = port; } // Set egress port
```

The forwarding is done using the `label_forwarding` table, which matches on the top label and applies the appropriate action. In the `MyIngress` control block, it checks if the top label is valid. If it is, the `label_forwarding` table is applied. If not, the packet is dropped by default.

3. egress.p4: This program is deployed at the **egress router (R4)**, where it performs the reverse of what the ingress router does, since the tunnel is bidirectional.

- It checks if a packet is coming from the MSLP tunnel, and if so, it pops all labels and restores the original IPv4 packet before forwarding it to the internal network.
- The table `tunnel_label_selector` and actions like `set_labels_tunnel1`, `set_labels_tunnel2`, and `rewriteMacsForTunnel` are reused for setting and forwarding the appropriate MSLP label stack for outbound traffic.
- Incoming MSLP packets follow the same decapsulation and delivery logic as described for the ingress router, including label removal and MAC rewriting.

Additionally, this program includes **stateful firewall** logic, which will be described separately in the next section.

Stateful Firewall

We then implemented a stateful firewall at router R4 to control UDP traffic between networks. The implementation addresses **two key functionalities:**

- **Connection State Tracking:** Using a **Bloom filter** implemented with P4 registers, the firewall records outgoing UDP flows originating from the 10.0.2.0/24 network. This enables validation of incoming reply packets without maintaining full connection state.
- **Port-Based Allow Rules:** The firewall permits traffic to specific UDP ports (e.g., DNS on port 53) regardless of connection state.

For the **Bloom filter**, we use two parallel bloom filters (CRC16 and CRC32 hashed) to record outgoing UDP flows. We generate two independent hashes per flow (to reduce collisions), using the action `compute_hashes`. For each outbound UDP packet from the internal subnet `10.0.2.0/24`, the following **5-tuple** is used to compute the Bloom filter hash: `src_ip`, `dst_ip`, `src_port`, `dst_port`, and `protocol` (always UDP = 17). And when a packet is sent or received, the following happens:

- Outbound packets update both bloom filters:

```
bloom_filter_1.write(hash1, 1); // Set bit
bloom_filter_2.write(hash2, 1);
```

- While inbound packets perform verification:

```
bloom_filter_1.read(bit1, hash1);
bloom_filter_2.read(bit2, hash2);
if (!(bit1 && bit2)) { drop(); }
```

For the **Specific Allowed UDP Ports**, they can bypass state tracking via a whitelist:

```
table allowed_udp_ports {
key = { hdr.udp.dstPort: exact; }
actions = { NoAction; drop; }
size = 1024;
default_action = drop; } // Default deny policy
```

Below is a flowchart that shows the logic behind the stateful firewall:

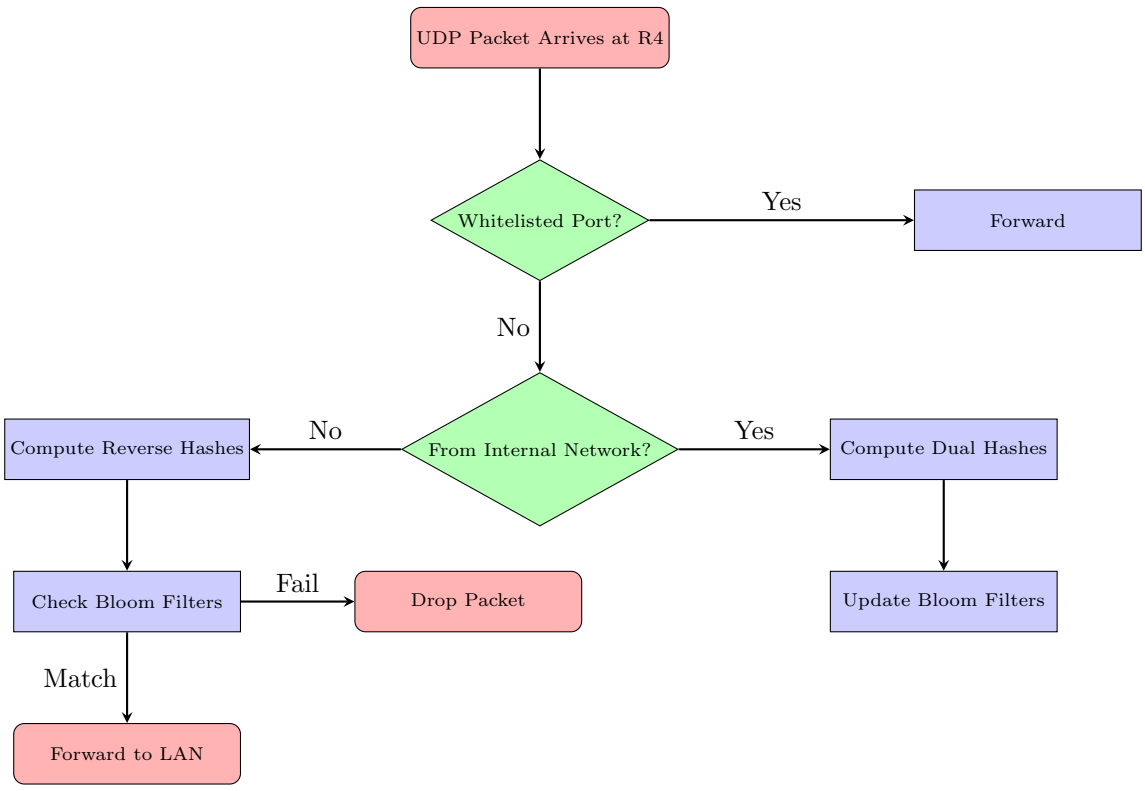

Figure 2: Firewall logic at Egress Router R4 showing separate paths for internal (record) and external (verify) packets

Controller

Architectural Design

Our Controller implements a multi-threaded architecture design to manage our network topology consisting of one L2 switch, 2 ingress/egress routers, and label switching routers inside our mslp core. The controller separates static configuration from dynamic operations through a carefully designed threading model. We would have a main thread and then two more threads beeing launched.
The controller architecture is built around three key operational components — ideas that were in our minds from the start as the foundation for tackling our problem:

1. Static Configuration Management (Main Thread)

The main thread handles all static table configurations that remain constant throughout the application's lifetime. This includes:

- **Forwarding Rules:** IP destination-based routing entries and routing depending on the choosen tunnel for ingress and egress routers.
- **Label-Based Forwarding:** tunnel forwarding rules for intermediate routers., depending on the seen labels.
- **Firewall Policies:** Predefined allowed UDP ports and directional traffic control.
- **Default Table Actions:** Fallback actions in case of no matches at all.

2. L2 Learning Switch Logic (Background Thread)

Dynamic MAC learning functionality through packet-in message processing. This thread handles the control plane logic for the L2 switch, learning MAC addresses dynamically and installing appropriate forwarding rules. This was given to us by the professor in the Task5 on the SDN course.

3. Dynamic Tunnel Selection (Background Thread)

A dedicated background thread implements round-robin tunnel selection for load balancing. This thread operates independently of the main configuration logic, switching between two predefined tunnel paths every 10 seconds

```
def tunnel_selector_thread(switches):
"""Thread that toggles tunnel selector every 10 seconds"""
current_action = "set_labels_tunnel1"

while True:
    try:
        # Toggle between tunnel1 and tunnel2 for load balancing
        if current_action == "set_labels_tunnel1":
            current_action = "set_labels_tunnel2"
        else:
            current_action = "set_labels_tunnel1"

        # Update tunnel selection on ingress (r1) and egress (r4) routers
        for switch_name in ['r1', 'r4']:
            if switch_name in switches:
                sw_data = switches[switch_name]
                setTunnelSelectorDefault(sw_data['helper'], sw_data['switch'],
                    current_action)

        print(f"Switched to {current_action}")
        time.sleep(10)  # 10-second switching interval
```

This tunnel selection works by changing the default action of one table especially dedicated to beeing changed every 10 seconds, where every try for matching a key will be missed, so it can always skip to the default action that is beeing manipulated.

```
table tunnel_label_selector {
    key = {hdr.ipv4.dstAddr: exact;} //FAKE KEY VAI FALHAR SEMPRE
    actions = {
        set_labels_tunnel1;
        set_labels_tunnel2;
    }
    size = 8;
    default_action = set_labels_tunnel1;
}
```

That same table will be called/checked just before the labeling printing on the packet on the apply block of both R1 and R4 routers:

```
apply {
    if (hdr.ethernet.etherType == TYPE_IPV4) {
        hdr.ethernet.etherType = TYPE_MSLP;
        tunnel_label_selector.apply();
        forTunnelMacrewrite.apply();
    }
```

3 Tests and Results

In this Tests and Results section we will show you the correct packet flow depending on the present choosen tunnel and the UDP connections being allowed or disallowed depending on the port.

Packet Flow tests and results

Packet Flowing through Tunnel1

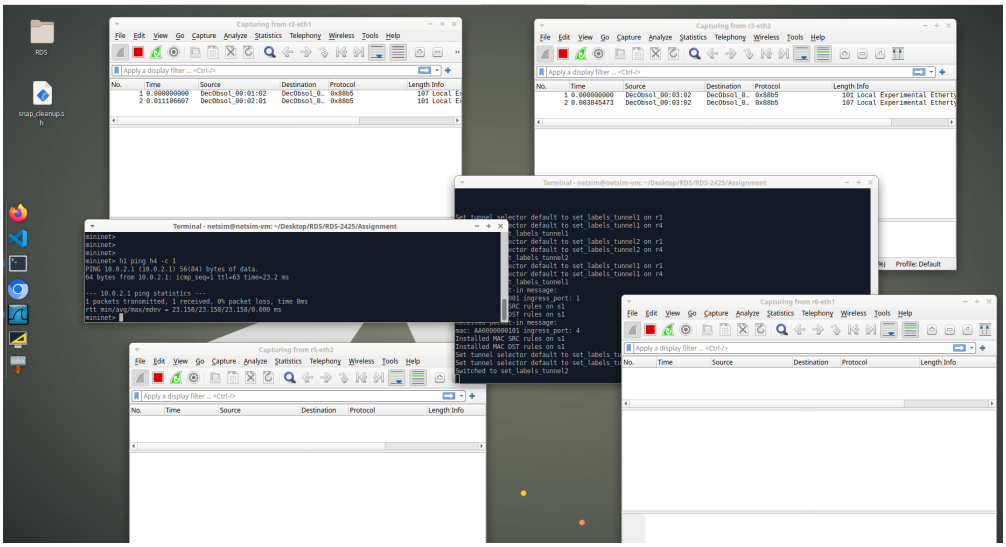

Figure 3: Packet Flowing through Tunnel1

Packet Flowing through Tunnel2

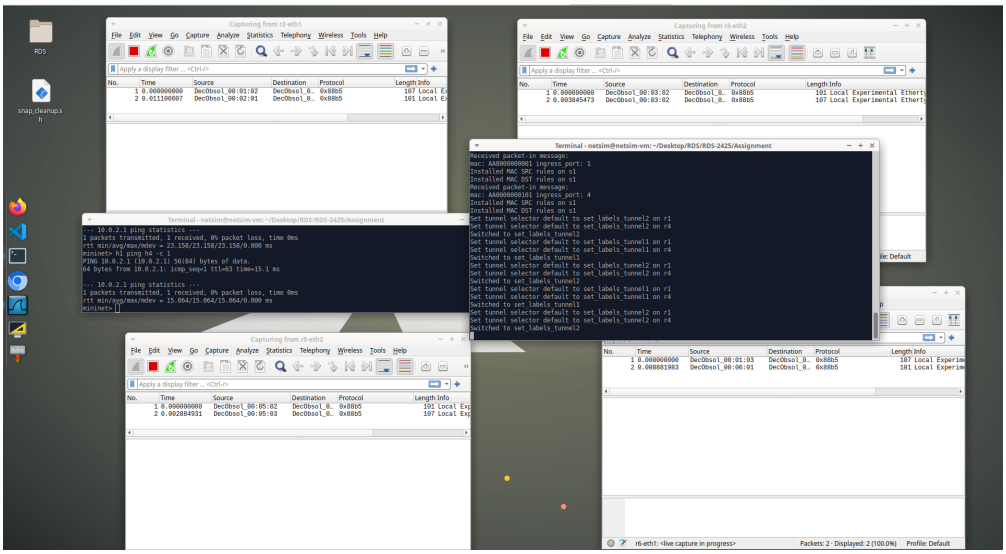

Figure 4: Packet Flowing through Tunnel2

Firewall tests and results

Allowed Traffic

The allowed ports are set in the controller through table population For now we have: 53,80,443,123

```
mininet> xterm h1 h4
```

On Node:h4

```
iperf3 -s -p 80 #test for allowed ports
```

On Node:h1

```
iperf3 -c 10.0.2.1 -u -p 80
```

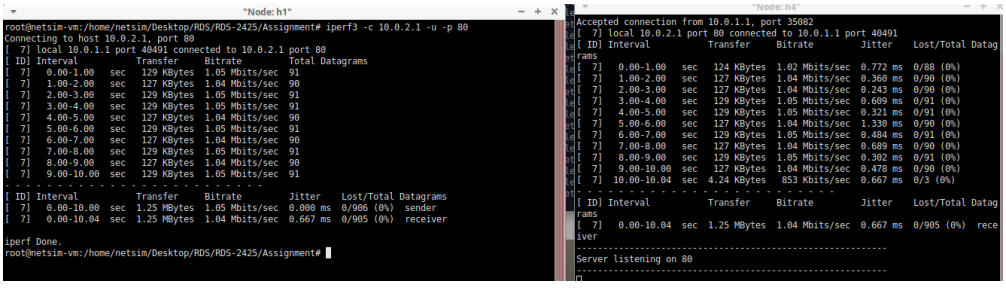

Figure 5: Allowed Traffic

Disallowed Traffic

```
mininet> xterm h1 h4
```

On Node:h4

```
iperf3 -s -p 90 #test for not allowed ports
```

On Node:h1

```
iperf3 -c 10.0.2.1 -u -p 90
```

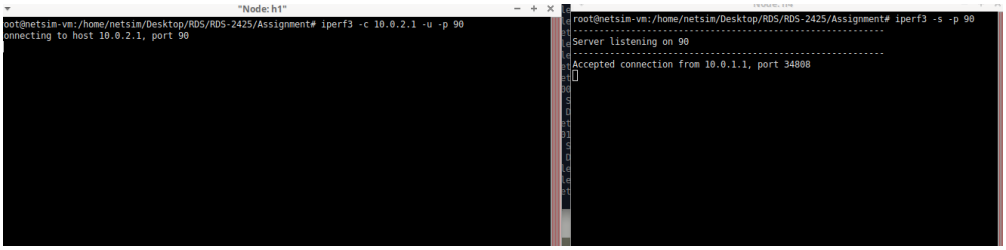

Figure 6: Not allowed Traffic

Every Traffic from H4 is allowed

On Node:h1

```
iperf3 -s -p 90
```

On Node:h4

```
iperf3 -c 10.0.2.1 -u -p 90
```

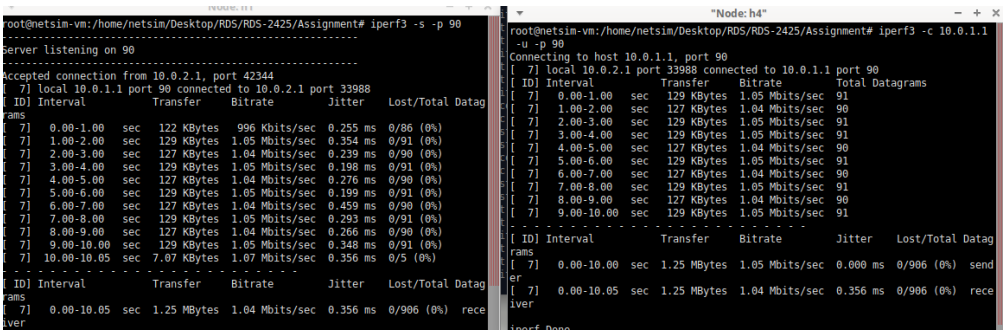

Figure 7: H4 as client

How you can test yourself!

We have prepared some scripts for both of us (you and us) so that it can be easier to test our implementation!

We have three mains scripts:

- **compilation_script.sh:** Used for compiling and generating all the P4 files.
- **topology_script.sh:** Used for the initialization of the topology with the correct JSONs and ports.
- **controller_script.sh:** Used for starting the controller.

Terminal 1

```
sudo mn -c #to clear mininet leftovers
./compilation_script.sh
./topology_script.sh
```

Terminal 2

```
./controller_script.sh
```

4 Conclusion

This project gave us hands-on experience with **programmable networks** by implementing a complete system using P4, from custom tunneling to stateful firewalls. Here's what we achieved:

This project demonstrated the power of programmable networks by building a complete SDN system using P4 — from custom tunneling protocols to stateful firewalls and a centralized controller. We designed and implemented four specialized P4 programs for distinct router roles, built a custom label-switching protocol (MSLP) with redundant tunnels, and used a Python-based controller to balance traffic dynamically.

One of the key insights was how programmable data planes and smart control logic can work together to achieve some more complex behaviors like dynamic tunneling and flow-level control using our controller architecture, with multithreaded design, coordinated table populating, MAC learning, firewall configurations and tunnel switching.

Looking ahead, we aimed to implement a smarter load balancer using real-time traffic metrics. By tracking counters on outgoing ports, the controller could make more intelligent forwarding decisions, dynamically adjusting tunnel usage based on actual load.

In short, we didn't just study SDN concepts in this course — we built and integrated them.