CESE4045

High Performance Data Networking

2025

Assignment 2

Programmable Data-plane Programming with P4

Teaching Team

Instructors

Dr. Nitinder Mohan Prof. Dr. Fernando Kuipers

Teaching Assistants

Chenxing Ji Adrian Zapletal Shatha Abbas Lorenzo Theunissen

Contents

Introduction

This assignment examines how programmable data-plane policies influence end-to-end performance by routing traffic in a controlled environment using P4 switches with the **BMv2** software switch. Using **Mininet** and **BMv2**, you will implement and examine the following three different data-plane pipelines:

1. **Basic forwarding**
2. **DiffServ with Expedited Forwarding**
3. **Round-robin Load Balancer**

You will design the topology, generate traffic, and analyze results to quantify how queuing and scheduling choices affect Flow Completion Time (FCT). In Part 3, you will implement a simple load balancer and analyze its performance by checking the amount of traffic received on different receivers.

Hint

- Make sure that you are always using the P4 `venv` created during the VM setup via: `source ~/p4-guide/bin/p4dev-python-venv/bin/activate`
- You can download the base code using the following link: `https://github.com/spear-lab/HPDN-Assignments.git`. The base code for this assignment resides in the `Assignment2` folder.

Prerequisites

- Completed VM setup with P4 installation.
- Basic programming experience.

Assignment Deliverables

For all parts of the assignment, you are required to submit a report (**max. 1 page**). For each part of the assignment, you are expected to submit the following:

Part 1: Topology

1. A topology file in the format of JSON named `topology.json` in the folder `MyTopo`

Part 2: Basic Forwarding

1. P4 program implementing simple L2/L3 forwarding behavior
2. Runtime configuration files (control-plane rules) for each switch, which must be named in the format of `sX-runtime.txt`
3. Experiment report:
   - FCT results
   - A paragraph explaining the results

Part 3: DiffServ

1. P4 program that implements setting the DSCP code point for traffic from different sources
2. P4 program that implements DSCP parsing and strict priority queuing for Expedited Forwarding
3. Runtime configuration files (control-plane rules)

4. Experiment report:
   - FCT results for Expedited Forwarding vs Best Effort
   - A paragraph explaining the difference in FCT between EF vs BE

Part 4: Load Balancer

1. A P4 program that implements a simple round-robin in-network load-balancer
2. Runtime configuration files (control-plane rules)
3. Experiment report:
   - Results of how much data is sent in total from the sender and how much is received at each receiver

Exercise 2.1: Topology [4 points]

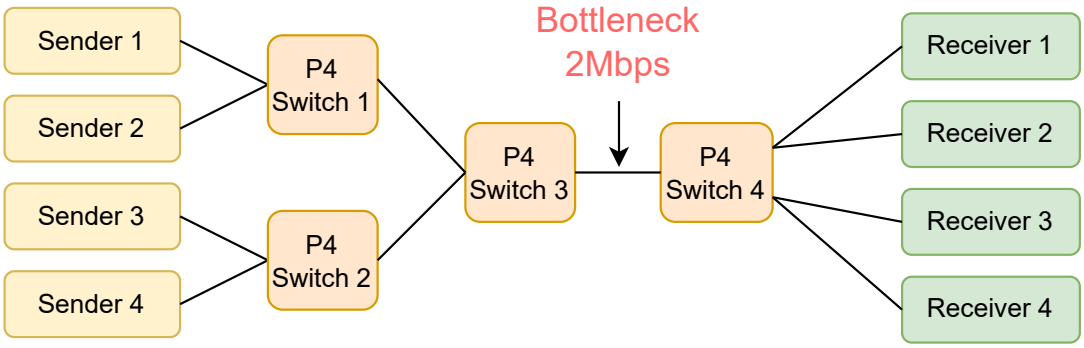

Figure 1: Topology for P4 exercises. Bandwidth needs to be set only for the bottleneck link. All links need not have additional latency.

You will start the exercise by building your Mininet topology with P4 switches as shown in Figure 1. All the following exercises will use this topology. Note that for grading purposes, each switch is associated with a number that you need to implement the topology in the same way.

Our P4 program will be written for the v1model architecture implemented on the BMv2 software switch, which is a simulated P4 software switch made for developing and testing P4 programs. The architecture file for the v1model can be found at:

1. Locally: `/usr/local/share/p4c/p4include/v1model.p4`.
2. Online: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4.

This file describes:

- The interfaces of P4 programmable elements in the architecture
- The supported externs
- The architecture's standard metadata fields

Hint

We encourage you to examine the v1model.p4 file to understand the available features.

In the `utils` folder of the code base, there is already a module that helps you to construct your topology and your P4 switch configurations based on an input file in JSON format. An example of a topology and the runtime configuration of the switches can be found in `P4Template/MyTopo`. The example, named `topology.json`, implements a simple topology with one host and one switch as shown in Figure 2.

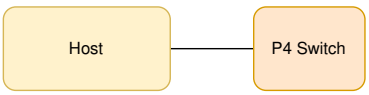

Figure 2: Topology in the provided `topology.json`

Note that with the topology JSON file you do create a Mininet network, but it follows a different format to define how the topology looks than the mininet code you wrote in the first assignment. In the format, the following elements are defined:

1. **Hosts**. Defines the hosts instantiated by Mininet. Parameters such as IP, MAC address, and startup commands are also defined in this section. In the example, the commands used are:
   (a) `route add`, which adds a static route to the default gateway (the switch).
   (b) `arp`, which manipulates the system's ARP cache (adds an entry for the switch's MAC address).
2. **Switches**. Defines switch behavior. The parameters that can be used here are:
   (a) `program`. Defines the program inserted into the switch (data plane). Note: If this parameter is not set, then the execution assumes the default P4 file (passed on startup).
   (b) `cli_input`. Defines the path for a command file to be executed in the switch's CLI. This file is directed at actions that are only supported by the `switch_cli` interface (such as setting up mirroring or setting queue rates and depths).
3. **Links**. Defines the links between network nodes. The following list format is used: `[Node1, Node2, Latency, Bandwidth]`, where nodes can be defined as `<Hostname>`, for hosts, and `<SwitchName>-<SwitchPort>` for switches. Both latency and bandwidth are optional, where latency is an integer defined in milliseconds (ms), and bandwidth is a float defined in megabits per second (Mbit/s).

Requirements

- Implement the topology from Figure 1 using the provided framework.

Submission

- A topology file in the format of JSON named `topology.json` in the folder `topo` that implements the topology in Figure 1.

Exercise 2.2: Basic Forwarding [10 points]

Objective

The objective of this exercise is to write a P4 program that implements basic Layer 3 IPv4 forwarding. With Layer 3 IPv4 forwarding, the switch must perform the following actions for every packet:

1. Forward the packet out on the appropriate port.
2. Updates the Ethernet destination address with the address of the next hop.
3. Updates the Ethernet source address with the address of the switch.
4. Decrement the time-to-live (TTL) in the IP header.

Both of your switches will have a single table, in which the control plane will be populated with static rules. Each rule will map an IP address or IP address range to the corresponding

MAC address and output port for the next hop.

P4 Program

In this first exercise, we will guide you step by step to complete the first P4 program. However, in the next exercises, you are expected to complete the project with a more general description and goal.

Header Definitions

The first step in writing a P4 program is to describe the headers that the switch will extract from an incoming packet. For Layer 3 forwarding, Ethernet and IPv4 packets need to be parsed. Therefore, the following header types are defined:

```
header ethernet_t {
    bit<48>   dstAddr;
    bit<48>   srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    bit<32>   srcAddr;
    bit<32>   dstAddr;
}
```

Additionally, populate the `headers` struct with instances of each header:

```
struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}
```

Metadata

Metadata contains local information about the packet that is not stored in packet headers (e.g., ingress port, egress port, timestamp). For this exercise, we don't need custom metadata:

```
struct metadata {
    /* empty */
}
```

However, we will use the `standard_metadata` supplied by the switch architecture:

```
struct standard_metadata_t {
```

```
    bit<9>      ingress_port;
    bit<9>      egress_spec;
    bit<9>      egress_port;
    bit<32>     instance_type;
    bit<32>     packet_length;
    ...
}
```

Note

The `standard_metadata` structure is implicitly defined by v1model and does not need to be explicitly defined in your P4 program.

Parser

The parser defines the order in which headers appear in the packet. First, parse the Ethernet header. Based on the `etherType`, either exit the parser or continue to parse IPv4:

```
parser MyParser(packet_in packet, out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {
    state start {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            /* TODO: add logic for state transition */
            default: accept;
        }
    }
    state parse_ipv4 {
        /* TODO: add logic for extracting IP header */
    }
}
```

Tasks

Fill in the missing parts:

1. Add state transition logic for IPv4
2. Implement the `parse_ipv4` state

Verify Checksum

The VerifyChecksum block executes after the parser. You can verify the IPv4 checksum:

```
control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
    apply {
        verify_checksum(
            hdr.ipv4.isValid(),
            { hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,
              hdr.ipv4.totalLen, hdr.ipv4.identification,
              hdr.ipv4.flags, hdr.ipv4.fragOffset, hdr.ipv4.ttl,
              hdr.ipv4.protocol, hdr.ipv4.srcAddr, hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
}
```

For simplicity, you can leave this block empty for this exercise.

Ingress Control Flow

The ingress block specifies all tables applied after the switch receives a packet:

```
control MyIngress(inout headers hdr, inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    apply {
        /* TODO: apply tables here */
    }
}
```

Match-Action Tables

P4 match-action tables associate lookup keys with actions. For example, a longest prefix-matching table based on destination IP looks like the following:

```
table ipv4_lpm {
    key = {
        hdr.ipv4.dstAddr: lpm;
    }
    actions = {
        ipv4_forward;
        drop;
        NoAction;
    }
    size = 1024;
    default_action = NoAction();
}
```

Actions

Define what operations each action performs:

```
action drop() {
    mark_to_drop(standard_metadata);
}

action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
    /* TODO: complete this action */
}
```

Tasks

Complete the `ipv4_forward` action to:

1. Set the egress port (`standard_metadata.egress_spec`)
2. Update the destination MAC address
3. Update the source MAC address
4. Decrement the TTL

Now, complete the Ingress control block's apply section:

Tasks

Apply the `ipv4_lpm` table only when the IPv4 header is valid.

Egress Control Flow

The egress block specifies processing at the output port. For this simple program, leave it empty:

```
control MyEgress(inout headers hdr, inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply { }
}
```

Update Checksum

Update the IPv4 checksum if any header fields were modified:

```
control MyComputeChecksum(inout headers hdr, inout metadata meta) {
    apply {
        update_checksum(
            hdr.ipv4.isValid(),
            { hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,
              hdr.ipv4.totalLen, hdr.ipv4.identification,
              hdr.ipv4.flags, hdr.ipv4.fragOffset, hdr.ipv4.ttl,
              hdr.ipv4.protocol, hdr.ipv4.srcAddr, hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
}
```

Deparser

The deparser defines the order of headers in outgoing packets:

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        /* TODO: emit IPv4 header */
    }
}
```

Task

Complete the deparser by emitting the IPv4 header.

Control-Plane Rules

Similar to the concept of Software-Defined Networking (SDN), the P4 program defines the data-plane processing capability, while the control-plane rules also need to be installed from the control plane. The BMv2 software switch that runs your compiled P4 program is `simple_switch`. It provides the data plane, but all tables start empty, so the switch needs control-plane rules to actually forward packets.

The API that simple_switch exposes for the control plane to add or remove table entries is Thrift. Tools like `simple_switch_CLI` use this Thrift interface to install forwarding rules during runtime.

You can find a small example of runtime rules in: **`P4Template/MyTopo/s1-runtime.txt`**.

Hint

Note: Each parameter should match its respective data-plane counterpart, meaning that for an entry to be inserted, a table must exist, and for an action to be called, it must be defined in the data-plane.

In the example, since there is only one table in the data-plane counterpart, all entries are defined for said table. Its behavior can be described as: for any given IP (key), the switch performs the `MyIngress.ipv4_forward` action, which sets both the `dstAddress` and `Port`.

Task

Create control-plane rules for all the switches, including **`s1`**, **`s2`**, **`s3`**, and **`s4`**, to enable IPv4 forwarding between all hosts in the above topology.

Running the Solutions

You have now completed most of Exercise 2.1, covering both the data-plane and control-plane parts. Upon completion of this exercise, any host in the submitted network is expected to be reachable in Layer 3 from any other host within the network. This means that the `pingall` command in Mininet works without packet drops.

Now, you need to measure how network congestion affects Flow Completion Time (FCT). More precisely, you will see the effect of bufferbloat and how the FCTs of short flows are affected when they co-exist with long flows.

We have provided a tool called **`fctm`** in **`Assignment2/scripts`**, a simple tool for sending a flow of a given size from one machine to another, and measuring the FCT. For more information, run the following command for the usage of the tool:

```
fctm -h
```

If you are running your VM on a Mac with ARM chip, you can use **`fctm_arm`**.

Hint

If you have trouble running the executable, refer to: https://gitlab.tudelft.nl/lois/fctm. We have also included the source code of fctm (same as in the repository linked above) in **`Assignment2/scripts`**.

Now run the following to obtain the FCT of the short flows using the provided tools and include the results in the report:

1. Start **`xterm`** inside the mininet for hosts **`h1, h2, h5, h6`**.

```
xterm h1 h2 h5 h6
```

2. Start an **`fctm`** receiver on **`h5`** and an **`iperf`** UDP server on **`h6`**.

```
h5> /path/to/fctm -s 300 -i ip_address_of_h5
h6> iperf -s -u -i 1
```

3. Start a long UDP flow from `h2` towards `h6` for 60 seconds.

```
iperf -c ip_address_of_h6 -u -t 60 -b 5M
```

4. Right after, start a short flow using the following command:

```
./fctm -c 10 -i ip_address_of_h5
```

5. Record the FCT value obtained using fctm in the report.

Of course, you can use `iperf` to generate a long TCP flow (the example above uses UDP) and you can play around with flow sizes in `fctm` (the example above uses 10 packets).

If you suspect that your VM has few compute resources (i.e., weak CPU), try a lower rate for the UDP flow (e.g., 2m) and/or try assigning more CPU to your VM.

Additionally, try to run `fctm` without the coexisting `iperf` traffic. What do you see?

Submission

- Complete P4 program implementing basic forwarding.
- Control-plane rules txt files, which must be in the format of `sX-runtime.txt`.
- Report the FCT collected with fctm with and without the coexisting long flow, and provide a short explanation (one paragraph in your report) on why the values differ.

Exercise 2.3: Expedited Forwarding [6 points]

`DiffServ` (Differentiated Services) is a scalable `QoS` architecture where each packet carries a Differentiated Services Code Point (DSCP) in the IP header. Instead of maintaining per-flow state on every router, packets are aggregated into a small number of `service classes`. Edge routers classify and mark traffic with appropriate DSCP values, while interior routers apply `Per-Hop Behaviors (PHBs)` that determine how packets of each class are queued and forwarded.

Common PHBs include Best Effort (BE, DSCP 0), Assured Forwarding (AF, four classes with three drop precedences each), and Expedited Forwarding (EF, DSCP 46). EF is typically implemented using strict-priority queues and aims to provide very low delay, jitter, and loss for high-priority traffic.

Objective

In this exercise, you will implement DSCP marking and an EF Per-Hop Behavior in bmv2 switches, and you will demonstrate the benefit of EF under congestion compared to BE forwarding. For this exercise, you will again use the topology created with Figure 1, and use the P4 Switch 1 and P4 Switch 2 as edge (exterior) routers to classify and mark traffic. The P4 Switch 3 will be implemented as the interior router that determines how traffic of each class is queued and forwarded. The P4 Switch 4 will be used for basic forwarding in this exercise.

For the simplicity of the exercise, you will treat traffic from `h1, h3` as expedited forwarding traffic, and apply best-effort forwarding for `h2, h4` traffic.

Edge Router

An edge router is a router that sits at the boundary between two networks, typically between an access network (e.g., campus network) and a larger provider or core network. Its main role is to connect end domains to the wider internet or backbone. Due to its unique location of being either the first or the last hop at the boundary, it is natural that the edge router classifies and marks traffic using DSCP for QoS purposes.

In this exercise, you will implement the behavior of an edge router for marking DSCP using both Switch 1 and Switch 2 using the topology in Figure 1. The following tasks describe the data-plane packet processing logic of an edge router:

Tasks

1. **Each Edge Router needs to implement DSCP marking.**
   (a) **Parse DSCP from the IPv4 header.**
      i. Modify or verify the IPv4 header definition so that the 8-bit *Type of Service / DiffServ+ECN* field is explicitly modeled, and the upper 6 bits are interpreted as the DSCP value.
      ii. Make sure the checksum computation block matches your header definition.
   (b) **DSCP-based classification.**
      i. Classify packets into the following two traffic classes based on the description above:
         - Traffic from h2 or h4 gets DSCP = 0 (Best Effort),
         - Traffic from h1 or h3 gets DSCP = 46 (Expedited Forwarding).
      ii. Update the packet headers accordingly such that interior routers can identify the service class.
   (c) **L3 forwarding.**
      i. Implement (or reuse) an IPv4 L3 forwarding table that matches on the destination IP address and selects the egress port and next-hop MAC address.
      ii. Ensure that both EF and Best Effort traffic are forwarded correctly using this table.

Interior Router

An interior router operates inside the DiffServ-enabled domain, typically within the core or backbone, and assumes that packets have already been classified and marked by edge routers. Instead of performing complex per-flow inspection, the interior router simply reads the DSCP of each packet and maps it to a small set of traffic classes, each associated with a separate per-hop behavior. For example, packets marked as Expedited Forwarding are placed into a strict-priority queue, while Best Effort packets are placed into a default queue.

In this exercise, you will implement the behavior of an interior router for packet scheduling using the DSCP value set by the edge routers. We will use the strict-priority queuing mechanism provided by bmv2, exposed via the `v1model.p4`: the highest-number priority queue is granted strict priority at all times.

Tasks

The Interior Router needs to implement queue selection and priority scheduling in the Ingress Processing.

1. In your P4 program, assign the correct priority to packets (via `standard_metadata.priority`) based on the DSCP values set by the edge routers.

Control-Plane Rules

Control-plane rules need to be configured for each match-action table on each switch.

Hint

You are running a different P4 program per switch in this exercise. Make sure you use the correct program for each switch in your topology file.

Task

Complete the runtime rules in each switch using the runtime txt file as you did in the previous exercise.

Running the Solution

We have provided you with a command to start your P4 switches with 8 priority queues per port.

```
make
```

For more information, refer to `utils/Makefile`

Important

The bmv2 implementation has a buffer at the egress port of a bmv2 switch that is a different buffer than the buffer used by the virtual interface that connects to the mininet link. Since the bmv2 switches are connected via mininet links, throttling the mininet link speed leads to queue buildup at the mininet link. However, the queue must be built inside the bmv2 switch for bmv2 scheduling to work correctly. Therefore, it is necessary to throttle the bmv2 egress port speed to ensure the queue builds inside the bmv2 switch. You can do so by starting `simple_switch_CLI` with the correct thrift port number for `s3` and then using the following command:

```
set_queue_rate <number_pps> egress_port
```

For example:

```
set_queue_rate 80 3
```

Instead of using `simple_switch_CLI` every time you start your mininet, you can also use your control plane rules to set the queue rate.

Hint

The above command uses packets per second (pps) for the bandwidth. You need to calculate the correct pps-to-Mbps conversion using a packet size of 1500 bytes. You should use a lower pps rate than the bottleneck bandwidth you specified in the topology JSON (to ensure that the queue builds up inside the bmv2 switch). Our example command uses 80 pps, which is slightly less than 1 Mbps.

Tasks

1. Repeat the experiment steps from the previous exercise and make sure the short flow generated by `fctm` is sent from an expedited host (e.g. `h1` or `h3`), while the `iperf` long flow is sent from a best-effort host (e.g. `h2` or `h4`).
2. **Measurement and FCT analysis.**
   (a) Record the flow completion time (FCT) for short EF flows when they coexist with a long flow.
   (b) Check your results and compare the FCT between baseline BE (from your previous exercise) and EF.
   (c) In your report, discuss why the resulting FCTs differ when using EF vs when using BE for the short flow.

Submission

1. P4 program implementing setting the DSCP for traffic from different sources
2. P4 program implementing DSCP parsing and strict priority queuing for Expedited Forwarding
3. Runtime configuration files (control-plane rules)
4. Experiment report:
   - FCT results of Expedited Forwarding vs Best Effort
   - A short paragraph comparing the FCTs and explaining the difference

---

Exercise 2.4: Round-robin Load Balancer [8 points]

`Load balancing` is a common technique to distribute traffic across a pool of backend servers. Instead of sending all packets directly to a single destination, clients connect to a *virtual IP address (VIP)* that represents the service as a whole. The load balancer receives traffic for the VIP and selects a backend server (with its own IP and port), forwarding the packet to that server. A key challenge is to implement this traffic steering at high speed while keeping the control-plane configuration simple and scalable.

In programmable data planes, we can implement load-balancing logic directly in P4. Stateless schemes such as simple hash-based ECMP only rely on packet header fields, whereas stateful schemes can maintain additional state across packets (e.g., counters or per-flow mappings) to implement policies such as round-robin, weighted round-robin, or flow pinning. In this exercise, you will implement a simple round-robin load balancer using registers.

Objective

In this exercise, you will implement a round-robin load balancer in P4 (bmv2) that forwards packets sent to a VIP toward a set of backend servers (e.g., `Receiver 1 - 4`). Your design should use a register to maintain state in the data plane and implement a simple round-robin scheme among the backend servers.

You will use a topology where a single P4 switch (`Switch 4`) acts as the front-end load balancer between a set of clients and multiple backend servers (see Figure 1). Clients send traffic to the VIP; the P4 switch rewrites the headers and forwards the packets to one of the backend servers. For simplicity, you may assume that the return traffic from servers back to clients is routed back to the same source (you do not need to implement any logic on the reverse path for this exercise – you only need to implement the route from the clients to the servers).

Load-Balancing Switch

The load-balancing switch is responsible for intercepting packets destined to the VIP and steering them to backend servers. In this exercise, you will implement the behavior of such a switch as follows.

Hint

Registers are a predefined `extern` object provided by the architecture model. Refer to `v1model.p4` for the detailed usage. Metadata can be used to store temporary results.

Tasks

1. **VIP identification.**
   (a) Identify packets destined to the VIP (e.g., a fixed IP address of the receiver subnet) using a match-action table in the ingress pipeline.
   (b) For packets that are not addressed to the VIP, your switch should behave as a normal router and forward them using regular L3 forwarding.
2. **In-data-plane state using a register.**
   (a) Declare a `Register` in your P4 program that holds a single integer counter (e.g., 8 bits wide) used as a global round-robin pointer.
   (b) On each packet destined to the VIP, your P4 code should:
      i. Read the current counter value from the register.
      ii. Map the counter value to one of the backend server ports (Hint: use `bit-wise AND`).
      iii. Compute the next counter value and write it back into the register.
3. **Backend selection and forwarding.**
   (a) Implement a match-action table with a corresponding action that forwards to the corresponding server correctly.

Control-Plane Rules

As in the previous exercises, you must configure the control-plane rules.

Task

Populate the control plane rules for the load-balancing switch so that:

- the VIP match table recognizes packets destined to the VIP address and applies your load-balancing action,
- the backend mapping table contains one entry per backend server, mapping backend indices to the corresponding IP addresses, MAC addresses, and egress ports.

In the `scripts` folder, we have provided two simple scripts:

1. `client.py`: send UDP packets to a designated IP address.
2. `server.py`: receives UDP packets from a designated IP address and reports the number of UDP packets received every 5 seconds.

Hint

If you were able to see packets captured on the link, but the server is not receiving packets, check the UDP checksum of the packets. What is wrong?

Running the Solution

Upon completing this exercise, your load-balancer switch should distribute traffic equally among the receivers. Therefore, running the solution correctly allows you to see roughly the same amount of traffic received at each host.

Tasks

1. Start receivers using using the provided `server.py` at each receiver.
2. Generate some traffic from `h1` toward the VIP using the provided `client.py`.
3. Measurement and load distribution analysis: record how many bytes or packets are received by each backend server.

Submission

1. A P4 program that implements a simple round-robin in-network load-balancer
2. Runtime configuration files (control-plane rules)
3. Experiment report:
   - Results of how many packets or bytes are sent in total from the sender and how many are received at each receiver.

End of Assignment 2