COMP536 | Homework Two: Programmable Networks

Course: COMP536 Secure and Cloud Computing
Name: Hua-Yu Cheng (hc105)
Email: hc105@rice.edu
Date: 10/10/2025

Overview

P4 Data Plane Program

`multi_lb.p4` implements the data plane logic for all the milestones in this assignment, which includes three different load balancing methods in S1:

- ECMP load balancing (`mode=1`)
- Per-packet load balancing (`mode=2`)
- Flowlet switching (`mode=3`)

S1 uses the custom headers below to check which load balancing method it is using and strips the `first_hop` header after processing the packet:

```
// First hop header tag for load balancing
// S1 removes this header and forwards the packet as usual
header first_hop_t {
    // 1: first hop (S1), 0: not first hop (S2, S3, S4)
    bit<8>    tag;
}

// Header for load balancing metadata
header lb_meta_t {
    bit<8>  mode;       // 1=per-flow ECMP, 2=per-packet, 3=flowlet
    bit<32> flow_id;    // Flow identifier (from sender)
    bit<32> seq;        // Per-packet sequence (from sender)
}
```

Scapy Program

Two Scapy scripts are used to test and verify the P4 data plane.

random_flows.py generates traffic from H1 to H2 with custom load-balancing headers and supports all three modes (`--mode 1` for ECMP, `--mode 2` for per-packet, and `--mode 3` for flowlet switching). It also queries S1 before and after sending to report how bytes are split across ports.

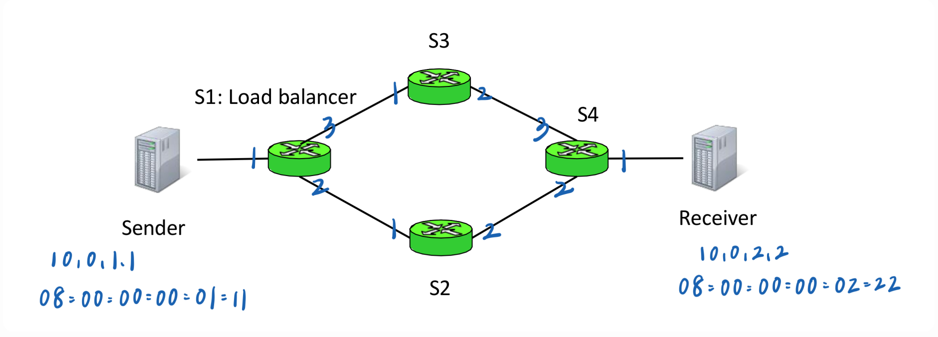
More options including `--dst`, `--flows`, `--proto` can be configured to customize the traffic. Use `python random_flows.py -h` to check more available options.

receive_ooo.py runs on H2 to record the sequence numbers of incoming packets and calculate how many are received out of order. Use `python receive_ooo.py -h` to check more available options.

Milestone 1: Build a Topology for Multipath Routing

Task 1

`/topo/topology.json` describes the topology, with the following IP address, MAC address and port assigned.



Task 2

`multi_lb.p4` implements the ECMP load balancing logic. The packet will have a `mode=1` header field in the `lb_meta` header so that S1 knows it is using ECMP.

S1 uses the `compute_ecmp_hash` action to compute which route (ECMP bucket) to send the traffic to by hashing the TCP/IP 5 tuple. It sets the `meta.ecmp_bucket` metadata so that when applying `table ecmp_select` we can choose the next hop by the exact bucket number.

Task 3

A new EtherType `TYPE_QUERY=0x1236`, a custom query header `query`, and two registers are used to keep track of the number of bytes sent to each of S1's egress ports:

```
// Query header
header query_t {
    bit<64>  port_2_bytes;  // The bytes sent to port 2 in S1
    bit<64>  port_3_bytes;  // The bytes sent to port 3 in S1
}

// Register to store byte counters for S1 egress ports
const bit<32> PORTS_MAX = 64;
register<bit<64>>(PORTS_MAX) port_bytes;
```

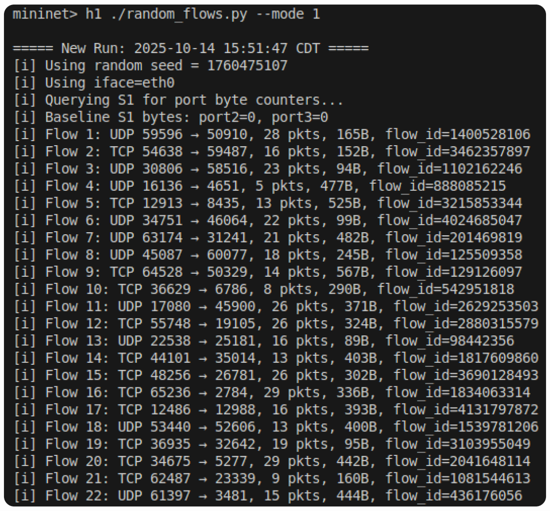
Inside `MyEgress` control block, if the `query` header is not valid (i.e., it is a normal packet), the `count_bytes` action will record and increment the corresponding register's byte count. If the `query` header is valid, the `answer_query` action will read the byte count in the register and fill in the answer to the `query` header.

To send the answer back, the `set_query` action in `MyIngress` sets the address and the port so that the reply goes back to whoever asked. The sender can simply look at the `query` header of the replied packet to get the statistics.

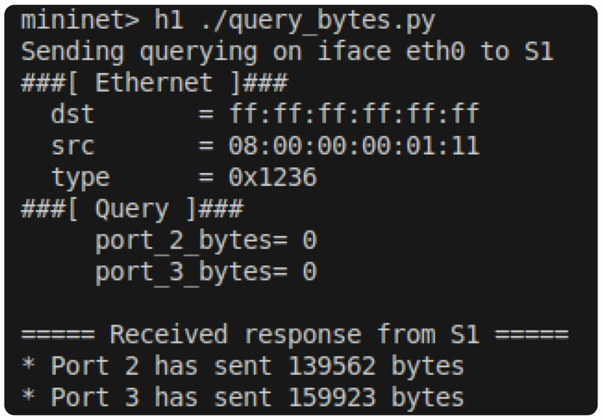
To test out the query function, run the following commands:

```
make clean && make

# Inside mininet CLI
h1 ./random_flows.py --mode 1
h1 ./query_bytes.py
```

Sample result:



```
mininet> h1 ./random_flows.py --mode 1

===== New Run: 2025-10-14 15:51:47 CDT =====
[i] Using random seed = 1760475107
[i] Using iface=eth0
[i] Querying S1 for port byte counters...
[i] Baseline S1 bytes: port2=0, port3=0
[i] Flow 1: UDP 59596 → 50910, 28 pkts, 165B, flow_id=1400528106
[i] Flow 2: TCP 54638 → 59487, 16 pkts, 152B, flow_id=3462357897
[i] Flow 3: UDP 30806 → 58516, 23 pkts, 94B, flow_id=1102162246
[i] Flow 4: UDP 16136 → 4651, 5 pkts, 477B, flow_id=888085215
[i] Flow 5: TCP 12913 → 8435, 13 pkts, 525B, flow_id=3215853344
[i] Flow 6: UDP 34751 → 46064, 22 pkts, 99B, flow_id=4024685047
[i] Flow 7: UDP 63174 → 31241, 21 pkts, 482B, flow_id=201469819
[i] Flow 8: UDP 45087 → 60077, 18 pkts, 245B, flow_id=125509358
[i] Flow 9: TCP 64528 → 50329, 14 pkts, 567B, flow_id=129126097
[i] Flow 10: TCP 36629 → 6786, 8 pkts, 290B, flow_id=542951818
[i] Flow 11: UDP 17080 → 45900, 26 pkts, 371B, flow_id=2629253503
[i] Flow 12: TCP 55748 → 19105, 26 pkts, 324B, flow_id=2880315579
[i] Flow 13: UDP 22538 → 25181, 16 pkts, 89B, flow_id=98442356
[i] Flow 14: TCP 44101 → 35014, 13 pkts, 403B, flow_id=1817609860
[i] Flow 15: TCP 48256 → 26781, 26 pkts, 302B, flow_id=3690128493
[i] Flow 16: TCP 65236 → 2784, 29 pkts, 336B, flow_id=1834063314
[i] Flow 17: TCP 12486 → 12988, 16 pkts, 393B, flow_id=4131797872
[i] Flow 18: UDP 53440 → 52606, 13 pkts, 400B, flow_id=1539781206
[i] Flow 19: TCP 36935 → 32642, 19 pkts, 95B, flow_id=3103955049
[i] Flow 20: TCP 34675 → 5277, 29 pkts, 442B, flow_id=2041648114
[i] Flow 21: TCP 62487 → 23339, 9 pkts, 160B, flow_id=1081544613
[i] Flow 22: UDP 61397 → 3481, 15 pkts, 444B, flow_id=436176056
```



```
mininet> h1 ./query_bytes.py
Sending querying on iface eth0 to S1
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 08:00:00:00:01:11
  type      = 0x1236
###[ Query ]###
     port_2_bytes= 0
     port_3_bytes= 0

===== Received response from S1 =====
* Port 2 has sent 139562 bytes
* Port 3 has sent 159923 bytes
```

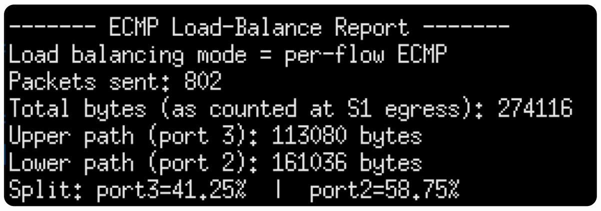
Milestone 2: Understanding the Limitations of ECMP

Task 1

Details about the scapy script `random_flows.py` can be found in the Overview section. To generate random flows and test how well ECMP load balances, run:

```
make clean && make

# Inside mininet CLI
h1 ./random_flows.py --mode 1
```

The log can be found in both terminal output and `random_flows.log`, which shows how many flows, packets per flow, and bytes were sent.



```
------- ECMP Load-Balance Report -------
Load balancing mode = per-flow ECMP
Packets sent: 802
Total bytes (as counted at S1 egress): 274116
Upper path (port 3): 113080 bytes
Lower path (port 2): 161036 bytes
Split: port3=41.25%  |  port2=58.75%
```

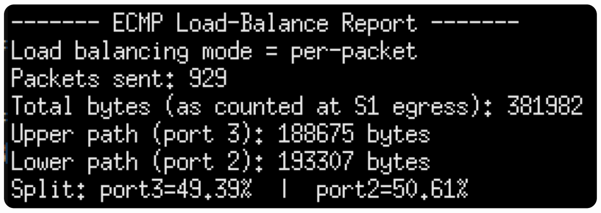
For this test run, 274,116 bytes were sent by the source. 41.25% (113.080 bytes) were sent via S1 port 3, while 58.75% (161,036 bytes) were sent via S1 port 2.

Task 2

Per-packet load balancing is implemented in the `compute_per_packet_hash` action in `multi_lb.p4`. It computes which path the packet should go using the TCP/IP 5 tuple and the **sequence number** of the packet, hence the route of each packet is reevaluated.

To generate a random flow that uses per-packet load balancing, run:

```
make clean && make

# Inside mininet CLI
h1 ./random_flows.py --mode 2
```



```
------- ECMP Load-Balance Report -------
Load balancing mode = per-packet
Packets sent: 929
Total bytes (as counted at S1 egress): 381982
Upper path (port 3): 188675 bytes
Lower path (port 2): 193307 bytes
Split: port3=49.39%  |  port2=50.61%
```

For this test run, 381,982 bytes were sent by the source. 49.39% (188,675 bytes) were sent via S1 port 3, while 50.61% (192,307 bytes) were sent via S1 port 2.

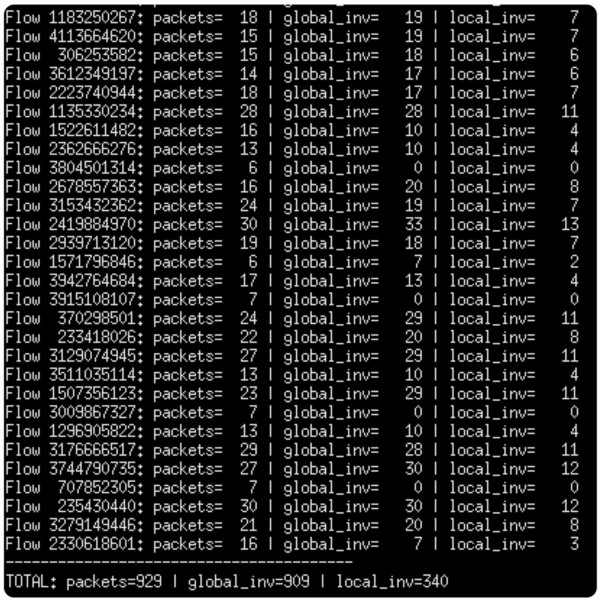
Task 3

To better observe out-of-order packet delivery, a link latency `2ms` is set for S1-S2, and `60ms` between S1-S3.

The receiver records the `seq` sequence number in the `lb_meta` header and count both global and local inversions to check how significant the out-of-order is.

To run a test run:

```
make clean && make

# Inside mininet CLI
xterm h1 h2

# Inside h2 terminal. Must do this first before h1
./receive_ooo.py

# Inside h2 terminal
./random_flows.py --mode 2

# After h1 sends all the packet,
# press Ctrl-C in h2 terminal to check the result
```



```
Flow 1183250267: packets=  18 | global_inv=   19 | local_inv=    7
Flow 4113664620: packets=  15 | global_inv=   19 | local_inv=    7
Flow  306253582: packets=  15 | global_inv=   18 | local_inv=    6
Flow 3612349197: packets=  14 | global_inv=   17 | local_inv=    6
Flow 2223740944: packets=  18 | global_inv=   17 | local_inv=    7
Flow 1135330234: packets=  28 | global_inv=   28 | local_inv=   11
Flow 1522611482: packets=  16 | global_inv=   10 | local_inv=    4
Flow 2362666276: packets=  13 | global_inv=   10 | local_inv=    4
Flow 3804501314: packets=   6 | global_inv=    0 | local_inv=    0
Flow 2678557363: packets=  16 | global_inv=   20 | local_inv=    8
Flow 3153432362: packets=  24 | global_inv=   19 | local_inv=    7
Flow 2419884970: packets=  30 | global_inv=   33 | local_inv=   13
Flow 2939713120: packets=  19 | global_inv=   18 | local_inv=    7
Flow 1571796846: packets=   6 | global_inv=    7 | local_inv=    2
Flow 3942764684: packets=  17 | global_inv=   13 | local_inv=    4
Flow 3915108107: packets=   7 | global_inv=    0 | local_inv=    0
Flow  370298501: packets=  24 | global_inv=   29 | local_inv=   11
Flow  233418026: packets=  22 | global_inv=   20 | local_inv=    8
Flow 3129074945: packets=  27 | global_inv=   29 | local_inv=   11
Flow 3511035114: packets=  13 | global_inv=   10 | local_inv=    4
Flow 1507356123: packets=  23 | global_inv=   29 | local_inv=   11
Flow 3009867327: packets=   7 | global_inv=    0 | local_inv=    0
Flow 1296905822: packets=  13 | global_inv=   10 | local_inv=    4
Flow 3176666517: packets=  29 | global_inv=   28 | local_inv=   11
Flow 3744790735: packets=  27 | global_inv=   30 | local_inv=   12
Flow  707852305: packets=   7 | global_inv=    0 | local_inv=    0
Flow  235430440: packets=  30 | global_inv=   30 | local_inv=   12
Flow 3279149446: packets=  21 | global_inv=   20 | local_inv=    8
Flow 2330618601: packets=  16 | global_inv=    7 | local_inv=    3
----------------------------------------
TOTAL: packets=929 | global_inv=909 | local_inv=340
```

Above is a corresponding result of a per-packet load balancing run. We can see the out-of-order packets per flow and total inversions across flows. There is 909 global inversions and 340 local inversions among 929 packets sent. Detail log can be found in `reorder_report.log`.

Milestone 3: Implement Flowlet Switching

Task 1

To implement flowlet load balancing, the following constant and registers are used:

```
// Hash table size for flowlets
const bit<32> FLOWLET_SLOTS = 1024;

// 30 ms idle gap between flowlet packets
const bit<32> FLOWLET_GAP_US = 30000;

// Last packet timestamp per flowlet
register<bit<48>>(FLOWLET_SLOTS) last_timestamp;

// Last selected ECMP bucket per flowlet
register<bit<8>>(FLOWLET_SLOTS) last_bucket;
```

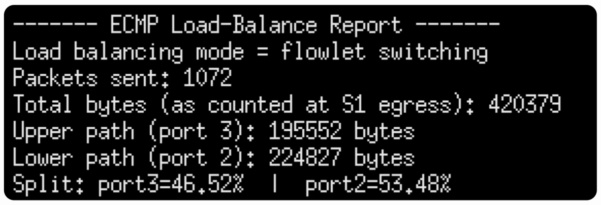
Since the link latency is `2ms` and `60ms`, the delta is `58ms`. Here we chose `30ms` as the delta to separate packets into flowlets, and the time between each bursts from the same flow is randomly selected from `(30, 58)` so to allow flowlets to switch paths while still see some reordering.

The `last_timestamp` register is to record the time of the last packet seen in a flowlet, if current time - `last_timestamp` of a flowlet is greater than `FLOWLET_GAP_US`, a flow can change its path.

The detail logic is implemented in `compute_flowlet_index` and `pick_new_bucket` action.

To generate a random flow that uses flowlet switching, run:

```
make clean && make

# Inside mininet CLI
h1 ./random_flows.py --mode 3
```



```
------- ECMP Load-Balance Report -------
Load balancing mode = flowlet switching
Packets sent: 1072
Total bytes (as counted at S1 egress): 420379
Upper path (port 3): 195552 bytes
Lower path (port 2): 224827 bytes
Split: port3=46.52%  |  port2=53.48%
```

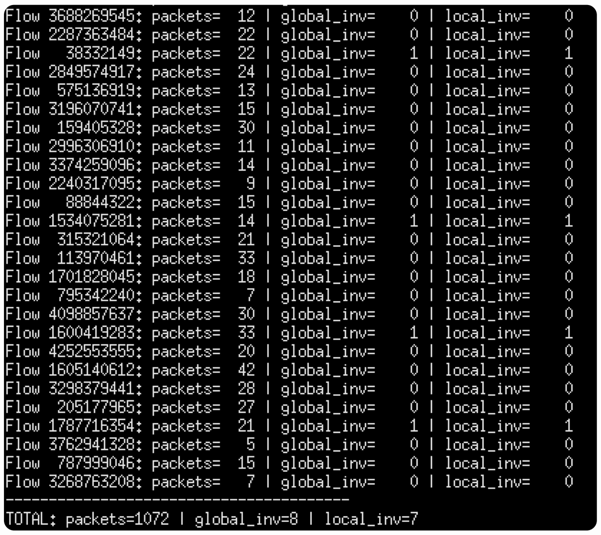
For this test run, 420,379 bytes were sent by the source. 46.52% (195,552 bytes) were sent via S1 port 3, while 53.48% (224,827 bytes) were sent via S1 port 2.

Task 2

To check the out-of-order amount of flowlet switching, run:

```
make clean && make

# Inside mininet CLI
xterm h1 h2

# Inside h2 terminal. Must do this first before h1
./receive_ooo.py

# Inside h2 terminal
./random_flows.py --mode 3

# After h1 sends all the packet,
# press Ctrl-C in h2 terminal to check the result
```

Below is the corresponding result. We can see the out-of-order packets per flow and total inversions across flows. There is 8 global inversions and 7 local inversions among 1072 packets sent. Detail log can be found in `reorder_report.log`.



```
Flow 3688269545: packets=  12 | global_inv=    0 | local_inv=    0
Flow 2287363484: packets=  22 | global_inv=    0 | local_inv=    0
Flow   38332149: packets=  22 | global_inv=    1 | local_inv=    1
Flow 2849574917: packets=  24 | global_inv=    0 | local_inv=    0
Flow  575136919: packets=  13 | global_inv=    0 | local_inv=    0
Flow 3196070741: packets=  15 | global_inv=    0 | local_inv=    0
Flow  159405328: packets=  30 | global_inv=    0 | local_inv=    0
Flow 2996306910: packets=  11 | global_inv=    0 | local_inv=    0
Flow 3374259096: packets=  14 | global_inv=    0 | local_inv=    0
Flow 2240317095: packets=   9 | global_inv=    0 | local_inv=    0
Flow   88844322: packets=  15 | global_inv=    0 | local_inv=    0
Flow 1534075281: packets=  14 | global_inv=    1 | local_inv=    1
Flow  315321064: packets=  21 | global_inv=    0 | local_inv=    0
Flow  113970461: packets=  33 | global_inv=    0 | local_inv=    0
Flow 1701828045: packets=  18 | global_inv=    0 | local_inv=    0
Flow  795342240: packets=   7 | global_inv=    0 | local_inv=    0
Flow 4098857637: packets=  30 | global_inv=    0 | local_inv=    0
Flow 1600419283: packets=  33 | global_inv=    1 | local_inv=    1
Flow 4252553555: packets=  20 | global_inv=    0 | local_inv=    0
Flow 1605140612: packets=  42 | global_inv=    0 | local_inv=    0
Flow 3298379441: packets=  28 | global_inv=    0 | local_inv=    0
Flow  205177965: packets=  27 | global_inv=    0 | local_inv=    0
Flow 1787716354: packets=  21 | global_inv=    1 | local_inv=    1
Flow 3762941328: packets=   5 | global_inv=    0 | local_inv=    0
Flow  787999046: packets=  15 | global_inv=    0 | local_inv=    0
Flow 3268763208: packets=   7 | global_inv=    0 | local_inv=    0
----------------------------------------
TOTAL: packets=1072 | global_inv=8 | local_inv=7
```

The inversion when using flowlet switching is less than that of the per-packet load balancing, detailed comparison can be found in the summary section.

Summary

Load Balance Ratio

LB Method	Upper Path (Port 3)	Lower Path (Port 2)
ECMP	41.25%	58.75%

Per-Packet	49.39%	50.61%
Flowlet	46.52%	53.48%

Out-of-Order Packet

LB Method	Global Inversion	Local Inversion
ECMP	0	0
Per-Packet (out of 929 packets)	909	340
Flowlet (out of 1072 packets)	8	7

All three load balancing methods successfully distribute traffic between the two available paths (S1–S2 and S1–S3).

However, the degree of balance and the ordering of packets vary based on how frequently the switch re-evaluates the forwarding path:

- **ECMP** provides stable but coarse-grained load distribution. Every packet in the same flow always follows the same path. This eliminates reordering but can easily lead to imbalance when a few large flows dominate one path.
- **Per-packet load balancing** achieves nearly perfect balance by independently hashing each packet's 5-tuple plus sequence number, allowing each packet to take any available path. While this maximizes link utilization, it causes significant packet reordering, which can harm TCP throughput or delay-sensitive applications.
- **Flowlet switching** strikes a middle ground. By detecting flowlet gaps and only switching paths at those points, it preserves ordering within bursts but still redistributes traffic across paths over time. The results show that flowlet switching achieves near-even utilization with far fewer reordering events.

The results highlight the trade-off between load balance and packet ordering.

ECMP is simple but uneven, per-packet is balanced but has a lot of reordering, and flowlet switching offers the best middle ground, which keeps the network well utilized with minimal packet disorder.

In short, flowlet switching achieves near-even load distribution with only slight reordering, making it the most practical approach among the three.