University of Crete
Computer Science Department
Lecturer: Prof. X. Dimitropoulos
TAs: E. Papadogiannaki, A. Shevtsov
M. Lakiotakis, A. Hatzivasileiou
Exercise Authors: Dr. V. Kotronis

Autumn Semester 2022
HY436 - Software Defined Networks
Tasks of **Assignment 3**
Assigned: 18 November 2022
Due: 07 December 2022, 23:59

Assignment 3

1 Before starting with this exercise

1. Attend the exercise session given on 18 Nov. 2022 by Antonis Hatzivasileiou. During the session, P4, and everything you will need to solve this exercise, will be presented/explained; also relevant questions will be answered on the spot.

2. Check the related guest lecture given on 15 Nov. 2022 by Vasileios Kotronis. The lecture material, while not a strict pre-requisite for starting with the exercise, will help you understand more about programmable data planes, and the main concepts behind P4.

3. Read the related material on the P4 official website [12]. We highly recommend that you read through the basic P4 tutorial [22], to get an idea of how P4 works. The material you will see in the current exercise is heavily adapted from that tutorial. The related GitHub repository can be found at [13] (this includes PDFs and other material). We highly recommend that you play with the tutorials there, and take a look at the sample solutions they provide. Moreover, we highly recommend that you check out the P4 seminal paper by Bosshart *et al.* [10]. For further reading, and to delve more into the current –state-of-the-art– $P4_{16}$ standard (which is also used to implement this exercise), we recommend checking [11]. Additional useful –optional– reads are [21] and [20].

4. Setup the exercise's Virtual Machine (VM), following the next steps:

   (a) Download and install VirtualBox [5].

   (b) Download the VM image from here [2].

   (c) Import the downloaded Virtual Machine (VM) file (`.ova`) into VirtualBox: open VirtualBox, select "File → Import Appliance", and navigate to the downloaded file. Then simply press next to create the image.

   (d) Go to "Settings → Network → Adapter 2" and add a host-only network adapter, in case you need to communicate with the VM via `ssh/scp`, etc. Make sure you enable both the option "Enable Network Adapter" and "Cable Connected".

   (e) Boot the VM: Select "P4 Tutorial 2019-04-25", and click "Start".

   (f) Connect using credentials "p4"/"p4".

   (g) Fire up a terminal and start!

Note that the VM has the following specifications:

- Name: `P4 Tutorial 2019-04-25_1`
- CPU cores: 2
- RAM: 2GB
- HDD: 16GB (expandable to 50GB dynamically)
- OS : Ubuntu Server 16.04 (64-bit)
- Network adapters: 1 NAT (communication with Internet)
- Installed packages: P4 dependencies (models, architectures), `Mininet`, `Wireshark`

You will thus need a host machine (Windows, Mac OS X or Linux) with at least 2 CPU cores, 4 GB RAM and 20 GB HDD to host this VM. You can use the host-virtualization software of your choice (e.g., VirtualBox [5]) to host the VM, the same way you did for the previous SDN exercises. The VM contains everything you will need in relation to P4 and the completion of this exercise pre-installed. Also, it contains `Mininet` [3, 4], which will be used to set up the test switch-host topology.

5. Optional: In case you are using the VM with an additional host-only interface (for host-guest communication), please make sure that you have the necessary tools to connect to the VM via `ssh` (e.g., `Putty` on Windows, native terminal on Mac OS X and Linux), as well as copy files between your host machine and the VM (e.g., Filezilla on any platform, or `scp` on Mac OS X or Linux). Besides these tools, you will not need anything extra in order to develop, compile, run and test this exercise in your environment (except for the editor of your choice; however the VM also provides a GUI and editor).

2 Introduction

In this exercise, you will learn about the open-source P4 (which stands for: "***P****rogramming* ***P****rotocol-independent* ***P****acket* ***P****rocessors*") architecture for programmable data planes, which is based on the following 3 pillars:

- Protocol-Independent Packet Processing (e.g., IP, OpenFlow, etc.)
- Language/Architecture separation
- "If you can interface with it, it can be used"

You will learn how to write the software of a programmable switch, emulating the data plane with `Mininet` (with which you are already familiar from previous assignments). The final objective of the exercise is the same as in Assigment 1; to write yourself a simple load balancer application which uses a programmable switch to balance load stemming from a set of clients towards a set of servers (4 clients-4 servers in the test topology). However, in the current assignment the entire logic will be implemented on the switch itself, without using a controller; the necessary software will be compiled and will be running directly on the switch. In fact, the control plane (P4 runtime) will be dynamically installed on the switch. Note that while P4 data planes can be programmed to talk to SDN controllers via OpenFlow, or even on-switch control planes using the device CPU, here we focus on the construction of an application-specific data plane that is configured at run-time. Therefore, and due to additional practical reasons, some features/requirements needed in Assignment 1 have been properly adjusted. More details on creating and submitting the code will be provided later on in the instructions. So, make sure that you follow each step carefully.
Note: you can skip this section and start directly with the assignment at Section 3, if you feel confident and are already familiar with P4 and its basic features.

2.1 The (Great) Planes

Traditional network architectures consist of the three well-known planes:

- Data (packet forwarding)
- Control (BGP, ARP, LLDP, OSPF, IS-IS, etc.)
- Management (configuration CLI/GUI, SNMP, etc.)

Each plane is essentially a group of algorithms. These algorithms:

- Process different kinds of traffic
- Have different performance requirements
- Are designed using different methodologies
- Are implemented using different programming languages
- Run on different hardware

While the control and management place have started becoming part of Network Operating Systems (NOS), the same does not hold for the data plane. This results in critical restrictions for programmability, since the vendor dictates how packets should be processed (with changes happening in cycles of several months or years).

2.2 P4 and Software Defined Networking

The most prominent enabler of Software Defined Networking (SDN) is OpenFlow, which is a standardized protocol to interact with a switch, leveraging a match/action abstraction. It enables a logically centralized entity (the "SDN controller") to insert, update and download flow table entries, query statistics, etc. Control applications run on top of the controller, interacting with it via a northbound interface; the controller interacts with the switch via a southbound interface (such as OpenFlow). This architecture drastically simplifies the control plane; for example, the controller may compute optimal paths at one location at the behest of a routing application and push the corresponding flow rules directly on the controlled switches, instead of waiting for distributed routing algorithms to converge. However, there are three major issues with the SDN data plane:

- Data plane evolution requires changes to standards (e.g., transitioning from 12 to 40 OpenFlow match types over the years).
- The interoperability between vendors is quite limited, resulting in the differences between different southbound interfaces (e.g., OpenFlow, `netconf`, `JSON/XML` variants, etc.) to be handled on the controller's side, introducing heavy complexity.
- Switches are typically equipped with fixed-function ASICs, which essentially "make the rules" with respect to what the controller can do with the switch.

To address the aforementioned issues and make the data plane truly programmable, P4 was proposed. Data plane programmability brings the following important benefits:

- Control and customization; make the device behave exactly as you want
- Introduction of new features (e.g., by adding new protocols)
- Reduction in complexity (e.g., by removing unused protocols); this also increases reliability and reduces risk.

- Efficient use of resources (e.g., flexible use of flow tables)
- Greater visibility (e.g., new diagnostic techniques, telemetry, etc.)
- Software-style development of hardware (e.g., rapid design cycle, fast innovation, fixes of data plane bugs in the field, etc.)
- Exclusivity and differentiation; there is no need to share your intellectual property with the chip vendor.

Therefore, while P4 can be used to host the data plane of a classic controller-OpenFlow switch setup, it is much more powerful and general than that in terms of enabling programmable data planes that can be (re-)configured at run-time. To demonstrate this, a brief list of publications that have used P4 to implement network applications is the following:

- Layer 4 Load Balancer – SilkRoad [18]
- Low Latency Congestion Control – NDP [15]
- In-band Network Telemetry – INT [17]
- Fast In-Network cache for key-value stores – NetCache [16]
- Consensus at network speed – NetPaxos [14]
- Aggregation for MapReduce Applications [19]
- ... and many more [6].

2.3 P4 Basics

2.3.1 Features

The P4 architecture for programmable data planes is:

- **Protocol Independent**: P4 programs specify how a switch processes packets and are not limited by existing protocols.
- **Target Independent**: P4 is suitable for describing everything from high-performance forwarding ASICs to software switches. Thus, several target architectures are supported.
- **Field Reconfigurable**: P4 allows network engineers to change the way their switches process packets after they are deployed (e.g., at run-time while a switch is running).

The current standard, called $P4_{16}$ [11], has the following specific features:

- **Clearly defined semantics**: the programmer can describe what the data plane program is doing.
- **Expressive**: supports a wide range of architectures through standard methodology.
- **High-level, Target-independent**: uses conventional constructs; the compiler manages the resources and deals with the hardware.
- **Type-safe**: enforces good software design practices and eliminates “stupid” bugs.
- **Agility**: high-speed networking devices become as flexible as any software.
- **Insight**: freely mixing packet headers and intermediate results.

The following related terms are important in the context of P4:

- **P4 Target**: An embodiment of a specific hardware implementation.
- **P4 Architecture**: A specific set of P4-programmable components, externs, fixed components and their interfaces available to the P4 programmer.
- **P4 Platform**: P4 Architecture implemented on a given P4 Target.

2.3.2 Workflow

To write P4 applications, the programmer does the following steps[1]:

1. Declares the headers that should be recognized and their order in the packet, using a programmable parser. This is used to generate a parsed representation of the packet.
2. Defines the tables and the exact processing algorithm, using a programmable match-action pipeline. This uses the parsed header information for matching and acting on the packet. Headers can be modified, added or removed.
3. Declares how the output packet will look on the wire, using a programmable "deparser". This serializes the packet for further forwarding.

The P4 program (including the parsers/deparsers, processing logic, etc.), written by the programmer, is combined with a P4 architecture model (e.g., the simple switch with the behavioral model `bmv2` [7] that we will also use in the rest of the exercise) and is compiled to a target-specific (the target being an embodiment of a specific hardware implementation) configuration binary, that is then loaded on the switch at run-time.

2.3.3 Components

- **Parsers**: state machine, bitfield extraction, etc.
- **Controls**: tables, actions, control flow statements, etc.
- **Expressions**: basic operations and operators.
- **Data types**: bitstrings, headers, structures, arrays.
- **Architecture description**: programmable blocks and their interfaces.
- **External libraries (or "externs")**: support for specialized components (e.g., vendor-specific features).

2.3.4 Basic Types

- `bit<n>`: Unsigned integer (bitstring) of size n
- `bit`: the same as `bit<1>`
- `int<n>`: Signed integer of size n (>=2)
- `varbit<n>`: Variable-length bitstring

[1] These steps comply to the PISA (Protocol-Independent Switch Architecture) model.

2.3.5 Header Types

These are ordered collection of members with the following features:

- Can contain `bit<n>`, `int<n>`, and `varbit<n>`
- Are byte-aligned
- Can be valid or invalid
- Provide several operations to test and set validity bit: `isValid()`, `setValid()`, and `setInvalid()`

An example is the following:

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

2.3.6 Other Types

- `struct`: Unordered collection of members (with no alignment restrictions), e.g.,:

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

- `header stack`: array of headers
- `header union`: one of several headers

Note that `typedef` can be used for alternative names of a given type. E.g.,:

```
typedef bit<48> macAddr_t;
```

2.3.7 Metadata

In P4, there are the standard metadata that accompany each packet, i.e.,

```
struct standard_metadata_t {}
    bit<9> ingress_port;    // the port on which the packet arrived
    bit<9> egress_spec;     // the port to which the packet should be sent to
    bit<9> egress_port;     // the port on which the packet is departing from
                            // (read only in egress pipeline)
    bit<32> clone_spec;     // ... (not relevant for the exercise)
    bit<32> instance_type;
    bit<1> drop;
    bit<16> recirculate_port;
    bit<32> packet_length;
    bit<32> enq_timestamp;
    bit<19> enq_qdepth;
    bit<32> deq_timedelta;
    bit<19> deq_qdepth;
    bit<48> ingress_global_timestamp;
```

```
    bit<32> lf_field_list;
    bit<16> mcast_grp;
    bit<1> resubmit_flag;
    bit<16> egress_rid;
    bit<1> checksum_error;
}
```

The programmer can also defined custom metadata, e.g., in the load-balancer exercise we have:

```
struct metadata {
    macAddr_t   dstMAC;     // the dst MAC to which the packet should be directed on L2
    bit<1>      isClient;   // whether the src IP of the packet belongs to a client
    bit<1>      isServer;   // whether the src IP of the packet belongs to a server
    bit<8>      srcGroup;   // the group of the src host
    bit<8>      dstGroup;   // the group of the dst host
}
```

2.3.8 Parsers

Parsers are functions that map packets into headers and metadata, written in a state machine style. Every parser has three predefined states: (i) start, (ii) accept, (iii) reject. Other states may be defined by the programmer. In each state, the parser executes zero or more statements, and then transitions to another state (loops are permitted). Note that the implementation of what happens when the "reject" state is reached is architecture/vendor-specific.

2.3.9 Selecting Header Fields

P4 has a select statement that can be used to branch in a parser. This is similar to case statements in C or Java, but without "fall-through behavior", i.e., break statements are not needed. In parsers, it is often necessary to branch based on some of the bits that were just parsed. For example, `etherType` determines the format of the rest of the packet. Match patterns can either be literals or simple computations such as masks.

2.3.10 Controls

P4 controls are similar to C functions (without loops). They can declare variables, create tables, instantiate externs, etc. The functionality is specified by code in apply statements. They can represent all kinds of processing that are expressible as a Directed Acyclic Graph (DAG):

- Match-Action pipelines
- Deparsers
- Additional forms of packet processing (e.g., updating checksums)

The interfaces with other blocks are governed by user- and architecture-specified types (typically headers and metadata).

2.3.11 Actions

Actions are very similar to C functions. They can be declared inside a control or globally. They are either directional (from the Data Plane) or directionless (from the Control Plane). Their parameters also have type and direction. Variables can be instantiated inside the actions. Many standard arithmetic and logical operations are supported:

```
+, -, *, ~, &, |, ^, >>, <<, ==, !=, >, >=, <, <=
```

, except for division and modulo. Non-standard operations that are supported are: bit-slicing ([m:l]; works as l-value too) and bit concatenation (++).

Table 1: IPv4_LPM table example.

Key (hdr.ipv4.dstAddr)	Action	Action Data (parameters)
10.0.1.1./32	`ipv4_forward`	{dstMAC=00:00:00:00:00:01, port=1}
10.0.1.2./32	`ipv4_forward`	{dstMAC=00:00:00:00:00:02, port=2}
*	NoAction	

2.3.12 Tables

Tables are the fundamental units of a Match-Action Pipeline. They:

- Specify what (packet) data to match on and match kind
- Specify (a list of) possible actions to apply on the packet
- Optionally specify a number of table properties: size, default action, static entries, etc.

Each table contains one or more entries (rules). Each entry contains:

- A specific key to match on
- A single action that is executed when a packet matches the entry
- Action data (possibly empty)

A data plane P4 program (1) defines the format of the table(s) (key fields, actions, action data), (2) performs the key lookup, and (3) executes the chosen action. The control plane (e.g., IP stack, routing protocols, OpenFlow controllers) populate the table entries with specific information (based on the configuration, automatic discovery, protocol calculations, etc.).
For an example, consider Table 1 (IPv4 Longest Prefix Match - LPM), which contains the information needed from a P4 switch to forward packets between two hosts (h1 with IP 10.0.1.1./32, MAC 00:00:00:00:00:01 on port 1 and h2 with IP 10.0.1.2/32, MAC 00:00:00:00:00:02 on port 2), matching the destination IP address of a packet. This table is populated by the control plane. The corresponding data plane P4 program which uses this table to actually forward the packet(s), is:

```
table ipv4_lpm {
    key = {
        hdr.ipv4.dstAddr: lpm;      \\ use the dst IP as the table match key
    }
    actions = {                     \\ list the possible table actions
        ipv4_forward;
        drop;
        NoAction;
    }
    size = 1024;                    \\ the maximum number of table entries
    default_action = NoAction();    \\ action upon Table ‘‘miss’’
}
```

Note that the underlying mechanism for the key match is based on the `match_kind` primitive, which is implemented in the architecture and is special in P4. The standard library (core.p4) defines three standard match kinds

- exact match
- ternary match
- LPM (Longest Prefix Matching) match

These are the three kinds that you will deal with in the exercise.
The architecture (`v1model.p4`) defines two additional match kinds:

- range
- selector

Other architectures may define (and provide implementations for) additional match kinds.
You can apply tables in control blocks as follows:

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    table ipv4_lpm {
        ...
    }
    apply {
        ...
        ipv4_lpm.apply();
    ...
    }
}
```

Note that you can also directly apply actions (without using tables) in control groups:

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action my_action(my_params) {
        ...
    }
    apply {
        ...
        my_action(my_params);
        ...
    }
}
```

2.3.13 Registers

Besides Tables, which are stateful objects keeping state between packets, there are other externs which serve as state keepers: counters, meters, registers, etc. For simplicity we will focus here only on registers, to explain the basic semantics and syntax. Consider the following example:

```
register<bit<48>>(16384) r;                      // an array of 16384
                                                 // 48-bit registers values
action ipg(out bit<48> ival, bit<32> x) {
    bit<48> last;
    bit<48> now;
    r.read(last, x);                             // use index x to read the
                                                 // reg-value into last
    now = std_meta.ingress_global_timestamp;
    ival = now - last;
    r.write(x, now);                             // write the new now value
                                                 // into x
}
```

2.3.14 Deparsing

A P4 deparser is expressed as a control function (no other special construct is needed) and its task is to assemble the headers back into a well-formed packet. This makes deparsing explicit and decouples the process from parsing. The function used in deparsers is the `emit(hdr)` which serializes the header if it is valid. Example:

```
control DeparserImpl(packet_out packet,
                     in headers hdr) {
    apply {
        ...
        packet.emit(hdr.ethernet);
        ...
    }
}
```

2.3.15 Debugging

`Bmv2` maintains logs that keep track of how packets are processed in detail. For example, in the exercise, you can check the file

```
/home/p4/tutorials/exercises/simple_load_balancer/logs/s1.log
```

after having built your .p4 application. Optionally, one can manually add information to the logs by using a dummy debug table that reads headers and metadata of interest, e.g.,:

```
control MyIngress(...) {
    table debug {
        key = {
            std_meta.egress_spec : exact;
        }
        actions = { }
    }
    apply {
        ...
        debug.apply();
    }
}
```

which results in additional information in logs, such as:

```
[15:16:48.145] [bmv2] [D]
[thread 4090] [96.0] [cxt 0]
Looking up key:
* std_meta.egress_spec : 2
```

2.3.16 Additional Information

For additional information and more advanced topics on the `P4Runtime`, `Makefile` and compilation process, API overview, `gRPC`, workflows and corresponding examples, please check slides 51-end from the official P4 tutorial [22].

2.3.17 FAQ

- Q: Can I apply a table multiple times in my P4 Program?
  A: No (except via resubmit / recirculate).
- Q: Can I modify table entries from my P4 Program?
  A: No (except for direct counters).
- Q:What happens upon reaching the reject state of the parser?
  A: This is architecture-dependent.
- Q: How much of the packet can I parse?
  A: This is architecture-dependent.

2.4 Hello World: Developing your first P4 application

A developer needs to make sure that the following parts are written and work correctly in a P4 program: library imports (Section 2.4.1), packet header descriptions (Section 2.4.2), packet header parsers (Section 2.4.3), packet checksum verification upon ingress (Section 2.4.4), ingress processing logic (Section 2.4.5), egress processing logic (Section 2.4.6), packet checksum (re-)computation upon egress (Section 2.4.7), and packet deparsing/serialization (Section 2.4.8). Finally, the workflow `parse` → `verify checksum` → `process ingress` → `process egress` → `(re-)compute checksum` → `deparse` is applied on the switch (Section 2.4.9). Note that it is also important to properly populate the defined tables via the control plane (e.g., runtime configuration; Section 2.4.10); this will instruct the data plane how to process packets.
In the following section we will write a hello world application that serves as a very basic router that does the following things:

- Parses Ethernet and IPv4 headers from packet
- Finds destination in IPv4 routing table
- Updates source / destination MAC addresses
- Decrements time-to-live (TTL) field
- Sets the egress port
- Deparses headers back into a packet

2.4.1 Imports

```
#include <core.p4>
#include <v1model.p4>

/* Optional: constants that will be used in the rest of the program */
const bit<16> TYPE_IPV4 = 0x800;
```

2.4.2 Headers

```
typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t 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;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

struct metadata {
    /* empty */
}

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

2.4.3 Parser

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

}
```

2.4.4 Checksum Verification

```
control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
    apply {  }
}
```

2.4.5 Ingress Processing

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

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

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}
```

2.4.6 Egress Processing

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

2.4.7 Checksum Computation

```
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
        );
    }
}
```

2.4.8 Deparser

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

2.4.9 Switch

```
V1Switch(
    MyParser(),
    MyVerifyChecksum(),
    MyIngress(),
    MyEgress(),
    MyComputeChecksum(),
    MyDeparser()
) main;
```

2.4.10 Runtime Configuration

Assuming a switch with 2 connected hosts (h1 on port 1 and h2 on port 2), the runtime (in `.json` format) of the example `.p4` program we wrote would look like the following:

```
{
  "target": "bmv2",
  "p4info": "build/basic.p4.p4info.txt",
  "bmv2_json": "build/basic.json",
  "table_entries": [
    {
      "table": "MyIngress.ipv4_lpm",
      "default_action": true,
      "action_name": "MyIngress.drop",          // drop if no match
      "action_params": { }
    },
    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.1.1", 32]
      },
      "action_name": "MyIngress.ipv4_forward",  // egress packet to h1 to port 1
      "action_params": {
        "dstAddr": "00:00:00:00:00:01",
        "port": 1
      }
    },
    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.1.2", 32]
      },
      "action_name": "MyIngress.ipv4_forward",  // egress packet to h2 to port 2
      "action_params": {
        "dstAddr": "00:00:00:00:00:02",
        "port": 2
      }
    }
  ]
}
```

Note that as you will also see in the exercise instructions, this runtime configuration can be changed dynamically (on the control plane's side) and be re-loaded on the P4 switch without any need to shutdown the processing pipeline(s).

3 Assignment: Build a simple load balancer

3.1 Motivation

Load balancing is a classic network-side application that is useful for data centers, ISPs and other enterprise networks. In this exercise, we will focus on a simple load-balancer implementation that balances the load stemming from different end-users, to a server farm, while taking into account features such as transparency and traffic isolation.

3.2 Overview of the Setup

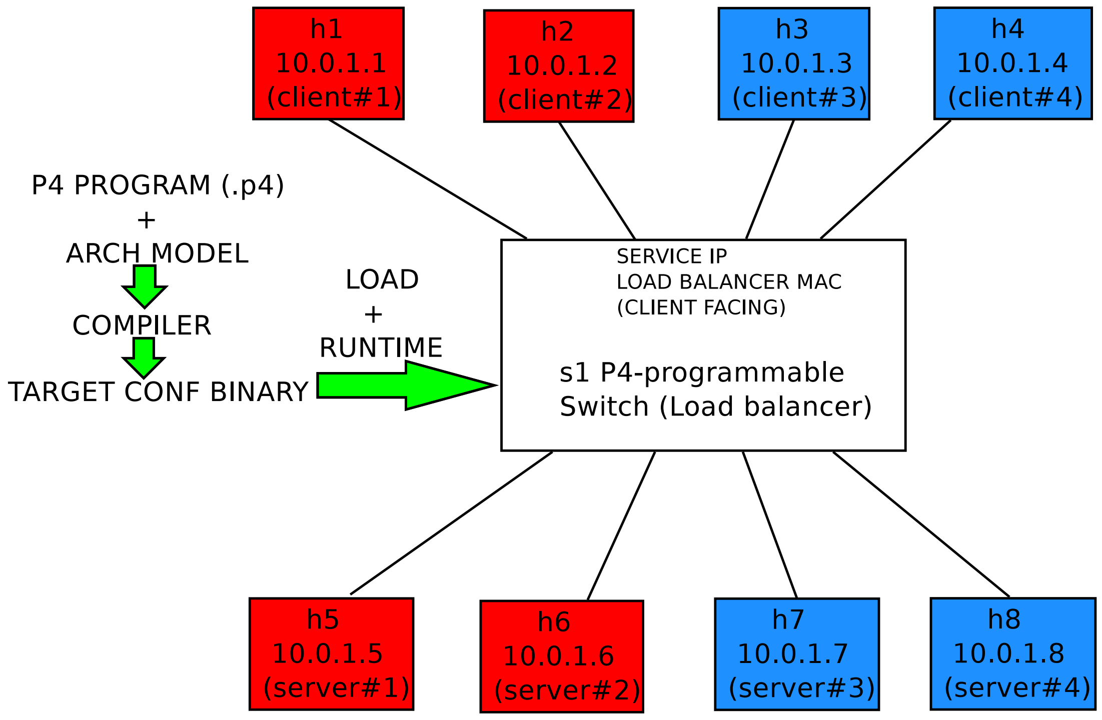

Figure 1: Load Balancer Setup.

The network you'll use in this exercise includes 8 hosts and a P4-enabled switch. The first 4 hosts (h1 to h4) act as clients for a service offered by the hosts h5 to h8, which act as servers. Each host belongs to a specific service group (e.g., "red/1" or "blue/2"); red clients can be served only by red servers, and blue clients by blue servers. The switch acts as a load balancer, and balances the flows from the clients towards the servers, taking into account their respective group membership. The clients are addressing the public IP of the service, and the switch acts as a transparent proxy between the clients and the actual server. Therefore, it also rewrites the destination IP targeted by the clients (public service IP) to the chosen server IP, and vice versa for the reverse communication. The setup is depicted in Fig. 1.
Note that for reasons of simplicity, the features of the load balancer are different from the first exercise (POX); please read carefully the following section!
Note that to implement this setup, we will use the following software components (pre-installed on the tutorial VM):

- `Bmv2`: a P4 software switch [7]
- `p4c`: the reference P4 compiler [8]
- `PI`: an implementation framework for a P4Runtime server [1]
- `Mininet`: a lightweight network emulation environment [3]

3.3 What the load balancer should do

- Answer to ARP requests both from the clients searching the MAC of the service IP address, and the servers searching the MAC of a client IP address. The switch should transform these requests into corresponding ARP replies which are sent back to the hosts (clients or servers) with a fake MAC that is associated with the load balancer (you can use "0A:00:00:00:00:01" for simplicity). Note that the ARP map that is required by the load balancer is provided by the runtime configuration for simplicity (i.e., the switch knows for each IP, its corresponding MAC address and physical port upon boot; this is different from the corresponding POX-based exercise).

- Redirect flows from the clients towards the servers using the following load balancing mechanism: **for each new IP flow from a client, select an –allowed– server at random.** After selecting a server belonging to the respective user service group, the application should direct the flow to this server. Of course, the server should see packets with their source MAC address changed to the MAC of the load balancer, but with the source client IP intact. The destination MAC and IP addresses should also be rewritten to the ones of the destination server. Be careful: the redirection should only happen for flows that stem from client IPs (i.e., non-server IPs) and which are directed to the service IP; their group membership should be taken into account when selecting the appropriate server.

- Direct flows from the servers to the clients. This should occur after rewriting the source IP address to the one of the service and the source MAC address to the load balancer fake MAC. In this way, the clients do not see any redirection happening, and they believe that all their communication takes place between their machines and the service IP (the load balancing mechanism is transparent).

- There is no need to handle forwarding between the servers themselves or between the clients themselves; in this exercise we are interested in the load-balancing behaviour and the traffic that flows between clients and servers. All the rest of the traffic (e.g., traffic from a client IP directly to a server IP or vice versa) does not need to be handled in-code for simplicity.

For the exercise you are required to implement a simple load balancing mechanism, without storing flow state on the switch (in contrast to the POX-based exercise). This means that a client talking to a chosen server replica, may use a changed replica even if its client-server flow is still ongoing; everything happens on a packet level (instead of a flow level). The load is balanced both on server basis from a single client, as well as on a client basis towards a single server. In practice, this statelessness creates an asymmetry which may cause problems, but is not considered in this exercise (*bonus question: What kind of problems? How could they be remedied in theory?*).

3.4 Writing your Application

First, download and unzip the hy436as3 source code (as delivered to you by the exercise TAs) on your local machine. You can choose your favorite editor and corresponding plugins for editing the P4 code.
Before proceeding, note that the following files:

- `hy436as3/Makefile`: makefile for compiling the P4 application.

- `hy436as3/README.md`: README for explaining how to set up.

- `hy436as3/receive.py`: script to receive IP packets.

- `hy436as3/send.py`: script to send IP packets.
- `hy436as3/topology.json`: configuration of Mininet topology.
- `hy436as3/slb-runtime.json`: the runtime configuration of the simple load balancer.
- `hy436as3/reconf_lb_groups_runtime.py`: script to reconfigure the switch at run-time with new rules (defined in `slb-runtime.json`).

are auxiliary and are provided as-is to you (no changes required).
You will need to implement the `hy436as3/simple_load_balancer.p4` script, which implements the P4 program that will run on the P4-enabled switch (emulated on Mininet).
For your convenience, a code skeleton for this file, with some initial functionality, is provided with the exercise, as well as all the other auxiliary files. **You will have to complete the parts named "WRITE YOUR CODE HERE".** In particular, you will need to write the following parts:

1. The `SLBParser parse_ethernet` logic for ethernet frame parsing.
2. The `SLBIngress arp_request_to_reply` action for transforming ARP requests into corresponding ARP replies.
3. Part of the `SLBIngress set_client_metadata` action for selecting a new server replica IP as the destination IP address of a client-to-server packet.
4. The description of the `SLBIngress arpmap` table that stores the mapping between a destination IP address and the corresponding destination MAC addesss and switch egress port.
5. The description of the `SLBIngress ipv4_servers` table that stores the information that a certain (source) IP address belongs to a server.
6. The description of the `SLBIngress src_group_membership` and `dst_group_membership` tables that store the information about which source and destination IP address is a member of which group(s), respectively.
7. The `SLBIngress` workflow that is applied when an incoming ARP request is received.
8. The `SLBIngress` workflow that is applied when an incoming IPv4 packet is received.
9. The `SLBEgress rewrite_client_to_server` action that rewrites the header of client-to-server packets.
10. The `SLBEgress rewrite_server_to_client` action that rewrites the header of server-to-client packets.
11. The `SLBEgress` workflow that is applied on outgoing IPv4 packets before they exit the switch.

Please check carefully the in-code comments that are designed to give you hints and help you complete the assignment. Also, please study the `slb-runtime.json` which is pre-populated by the control plane (i.e., us :)), to understand what kind of information (match keys, actions, action data/parameters) the different tables contain. The explanation of every table and action used is contained in-comment within the .p4 program skeleton.
Note 1: when you are programming your P4 application, the following function might come in handy w.r.t. randomized IP selection (IPs are given in integer format instead of dot-notation):
`extern void random<T>(out T result, in T lo, in T hi);`

To select servers, use the information contained in slb-runtime.json entries such as:

```
{
  "table": "SLBIngress.ipv4_clients",
  "match": {
    "hdr.ipv4.srcAddr": [
      "10.0.1.2",
      32
    ]
  },
  "action_name": "SLBIngress.set_client_metadata",
  "action_params": {
      "firstAllowedReplica": 167772421,
      "lastAllowedReplica": 167772422
  }
},
```

Observe that the replica ranges essentially determine which servers are allowed to be used by the client (in this case, 10.0.1.5 to 10.0.1.6 in integer format).
Note 2: the skeleton is indicative; any working solution that satisfies the constraints of the assignment is acceptable. Note that all necessary imports are already present in the provided code; if you use different than recommended functionality please take care of the corresponding imports.

3.5 Running your Application

1. In the VM location `/home/p4/tutorials/exercises`,
   create a new folder named `simple_load_balancer`.

2. Copy all files from `hy436as3` folder to the following VM location:

   ```
   /home/p4/tutorials/exercises/simple_load_balancer
   ```

3.
   ```
   cd /home/p4/tutorials/exercises/simple_load_balancer
   make run
   ```

   If the `make` process fails, check the error(s) on the terminal, fix and try again. *Note: The initial skeleton does not contain e.g., the implementation of missing tables, resulting in syntax errors; therefore you need first to think what to add and where and then start testing (and fix the errors).*

   If everything compiles correctly, you will see output like the following:

```
mkdir -p build pcaps logs
p4c-bm2-ss --p4v 16 --p4runtime-files build/simple_load_balancer.p4.p4info.txt
-o build/simple_load_balancer.json simple_load_balancer.p4
sudo python ../../utils/run_exercise.py -t topology.json -b simple_switch_grpc
Reading topology file.
Building mininet topology.
Switch port mapping:
s1:  1:h1 2:h2 3:h3 4:h4 5:h5 6:h6 7:h7 8:h8
Configuring switch s1 using P4Runtime with file slb-runtime.json
 - Using P4Info file build/simple_load_balancer.p4.p4info.txt...
 - Connecting to P4Runtime server on 127.0.0.1:50051 (bmv2)...
 - Setting pipeline config (build/simple_load_balancer.json)...
 - Inserting 34 table entries...
 - SLBIngress.ipv4_clients: (default action) => SLBIngress.unset_client_metadata()
 - SLBIngress.ipv4_clients: hdr.ipv4.srcAddr=['10.0.1.1', 32] =>
 SLBIngress.set_client_metadata(
 firstAllowedReplica=167772421,
 lastAllowedReplica=167772422)
 ...
s1 -> gRPC port: 50051
**********
h1
default interface: h1-eth0 10.0.1.1 00:00:00:00:01:01
**********
...
Starting mininet CLI

======================================================================
Welcome to the BMV2 Mininet CLI!
======================================================================
Your P4 program is installed into the BMV2 software switch
and your initial runtime configuration is loaded. You can interact
with the network using the mininet CLI below.

To view a switch log, run this command from your host OS:
  tail -f /home/p4/Desktop/tutorials/exercises/simple_load_balancer/
  logs/<switchname>.log

To view the switch output pcap, check the pcap files in
/home/p4/Desktop/tutorials/exercises/simple_load_balancer/pcaps:
 for example run:  sudo tcpdump -xxx -r s1-eth1.pcap

To view the P4Runtime requests sent to the switch, check the
corresponding txt file in
/home/p4/Desktop/tutorials/exercises/simple_load_balancer/logs:
 for example run:
 cat /home/p4/Desktop/tutorials/exercises/simple_load_balancer/
 logs/s1-p4runtime-requests.txt

mininet>
```

You now have a mininet terminal that you can use for testing (see Section 3.6).

3.6 Testing your application

1. On the mininet CLI, trigger an xterm for two different clients (let's say host h1 and h3, which belong to different service groups), and all servers (h5, h6, h7, h8):

   ```
   mininet> xterm h1 h3 h5 h6 h7 h8
   ```

   This will provide xterms with direct access to the different hosts.

2. Set up monitoring of ICMP packets (e.g., ping echo request/replies) on the servers' side:

   ```
   host_xterm# sudo tcpdump -ni <server_hostname>-eth0 icmp
   ```

   where <server_hostname>$\in$ {h5,$\cdots$,h8}.

3. Do the following tests:

   (a) From the h1 xterm, ping the service IP (10.1.2.3). You should see the ICMP echo requests being load-balanced to the different servers, which in turn respond to h3 with ICMP echo replies. Which servers accept this traffic and why?

   (b) From the h3 xterm, ping the service IP (10.1.2.3). You should see the ICMP echo requests being load-balanced to the different servers, which in turn respond to h1 with ICMP echo replies. Which servers accept this traffic and why?

   Optional: Note that, in addition to ping, you can also use any tool you like to generate IP traffic, such as iperf, related to both UDP and TCP communication. However, due to the stateless nature of the simple P4-based load balancer that you have implemented, you may encounter issues with establishing stable TCP sessions between the client(s) and the server(s) (see also bonus question). However, you can use instead the scapy-based packet sender and receiver that are available in SEND.PY and RECEIVE.PY auxiliary scripts. You can use them as follows:

   ```
   sender_xterm#     python send.py <destination> "<message>"
   receiver_xterm#   python receive.py
   ```

   Try experimenting with different configurations w.r.t. the `slb-runtime.json` (i.e., the run-time of the simple load balancer P4 app). You can apply your changes to the file at run-time by running:

   ```
   sudo python reconf_lb_groups_runtime.py
   ```

   on another terminal (and within the same folder) after editing (without shutting Mininet down!). For example, try changing the assigned groups of the clients. Does everything work as expected?

   Example change: "Change the group of a client to point to different server group". To do this, you need to actually make 3 changes:

   (a) Change the allowed ranges of the server IPs (note that we are using integer IP representations), so that they can be properly selected by the P4 switch, e.g., **from**:

```
...
{
  "table": "SLBIngress.ipv4_clients",
  "match": {
    "hdr.ipv4.srcAddr": [
      "10.0.1.1",
      32
    ]
  },
  "action_name": "SLBIngress.set_client_metadata",
  "action_params": {
      "firstAllowedReplica": 167772421,
      "lastAllowedReplica": 167772422
  }
},
...
```

to:

```
...
{
  "table": "SLBIngress.ipv4_clients",
  "match": {
    "hdr.ipv4.srcAddr": [
      "10.0.1.1",
      32
    ]
  },
  "action_name": "SLBIngress.set_client_metadata",
  "action_params": {
      "firstAllowedReplica": 167772423,
      "lastAllowedReplica": 167772424
  }
},
...
```

(b) Change the source group related to the client IP seen as traffic source, e.g., **from:**

```
...
{
  "table": "SLBIngress.src_group_membership",
  "match": {
    "hdr.ipv4.srcAddr": [
      "10.0.1.1",
      32
    ]
  },
  "action_name": "SLBIngress.set_src_membership",
  "action_params": {
      "group": 1
  }
},
...
```

to:

```
...
{
  "table": "SLBIngress.src_group_membership",
  "match": {
    "hdr.ipv4.srcAddr": [
      "10.0.1.1",
      32
    ]
  },
  "action_name": "SLBIngress.set_src_membership",
  "action_params": {
      "group": 2
  }
},
...
```

(c) Change the destination group related to the client IP seen as traffic destination, e.g., **from:**

```
...
{
  "table": "SLBIngress.dst_group_membership",
  "match": {
    "hdr.ipv4.dstAddr": [
      "10.0.1.1",
      32
    ]
  },
  "action_name": "SLBIngress.set_dst_membership",
  "action_params": {
      "group": 1
  }
},
...
```

to:

```
...
{
  "table": "SLBIngress.dst_group_membership",
  "match": {
    "hdr.ipv4.dstAddr": [
      "10.0.1.1",
      32
    ]
  },
  "action_name": "SLBIngress.set_dst_membership",
  "action_params": {
      "group": 2
  }
},
...
```

3.7 Debugging your application

Check Section 2.3.15. In general, you can check the following files after you build your application with make (*note that* `make clean` *will delete these files, so examine them either while Mininet is running or before cleaning everything up)*:

- `/home/p4/tutorials/exercises/simple_load_balancer/logs/s1.log`

  This file contains detailed logs of how the switch has processed received packets.
  Example log-lines:

```
...
[08:36:23.994] [bmv2] [D] [thread 3494] [0.0] [cxt 0] Processing packet
received on port 1
[08:36:23.997] [bmv2] [D] [thread 3494] [0.0] [cxt 0] Parser 'parser': start
[08:36:23.997] [bmv2] [D] [thread 3494] [0.0] [cxt 0] Parser 'parser' entering state
'start'
[08:36:23.997] [bmv2] [D] [thread 3494] [0.0] [cxt 0] Extracting header
'ethernet'
[08:36:23.997] [bmv2] [D] [thread 3494] [0.0] [cxt 0] Parser state 'start':
key is 86dd
[08:36:23.997] [bmv2] [T] [thread 3494] [0.0] [cxt 0] Bytes parsed: 14
[08:36:23.997] [bmv2] [D] [thread 3494] [0.0] [cxt 0] Parser 'parser': end
[08:36:23.997] [bmv2] [D] [thread 3494] [0.0] [cxt 0] Pipeline 'ingress':
start
[08:36:23.997] [bmv2] [T] [thread 3494] [0.0] [cxt 0] simple_load_balancer.p4(236)
Condition "!(hdr.arp.isValid() || hdr.ipv4.isValid())" (node_2)
is true
[08:36:23.997] [bmv2] [T] [thread 3494] [0.0] [cxt 0] Applying table 'tbl_drop'
...
```

- `/home/p4/tutorials/exercises/simple_load_balancer/logs/s1-p4runtime-requests.txt`

  This file contains requests from the control plane to update switch tables, etc.
  Example log-lines:

```
[2019-07-15 08:36:24.011] /p4.v1.P4Runtime/Write
---
election_id {
  low: 1
}
updates {
  type: INSERT
  entity {
    table_entry {
      table_id: 33610165
      match {
        field_id: 1
        lpm {
          value: "\n\000\001\001"
          prefix_len: 32
        }
      }
      action {
        action {
          action_id: 16782623
          params {
            param_id: 1
            value: "\n\000\001\005"
          }
          params {
            param_id: 2
            value: "\n\000\001\006"
          }
        }
      }
    }
  }
}
```

- /home/p4/tutorials/exercises/simple_load_balancer/logs/pcaps/*.pcap

  These files contain the captured packets on the input or output directions on the switch interfaces, and can be opened with Wireshark which is installed on the VM.

Note that you can also examine the following build files to check how your program has been compiled under the hood:

- /home/p4/tutorials/exercises/simple_load_balancer/build/ simple_load_balancer.p4.p4info.txt

  This file contains the captured P4 run-time attributes (table/action/param IDs, table structure, action params, etc.), e.g.,:

```
tables {
  preamble {
    id: 33561626
    name: "SLBIngress.arpmap"
    alias: "arpmap"
  }
  match_fields {
    id: 1
    name: "hdr.ipv4.dstAddr"
    bitwidth: 32
    match_type: LPM
  }
  action_refs {
    id: 16789210
  }
  action_refs {
    id: 16800567
    annotations: "@defaultonly"
    scope: DEFAULT_ONLY
  }
  size: 1024
}
```

- /home/p4/tutorials/exercises/simple_load_balancer/build/simple_load_balancer.json

  This file contains the compiled .p4 program in .json format, understood by the device.

3.8 Shutting everything down for re-testing

In case you need to shut down the P4 program and Mininet to re-test, please perform the following steps:

1. On the Mininet CLI, run:

```
mininet> exit
```

2. After Mininet exits, run:

```
vm_terminal$ make stop && make clean
```

Now you can re-edit your .p4 code locally, doing any desired changes. If needed, you can restart from Section 3.5, followed by all the steps of Section 3.6 and 3.7. Note that `make clean` deletes the temporary `logs`, `pcaps` and `build` directories, so run this only for cleaning up everything. There is no practical need to reboot the VM across re-tests.

3.9 How to submit your code

Please submit your code using the TURNIN submission program [9]:

- Log-in to one of the CS department's systems.
- Create a folder named ask3.
- ask3 folder should contain the following:
  1. `simple_load_balancer.p4`
  2. any additional version of `slb-runtime.json` you used to test your code (besides the given runtime), named e.g., `slb-runtime-vX.json`, where X$\in \{2, 3, \cdots\}$
- Use cd to make the directory one level above ask3 your current working directory.
- Issue the following command:

```
turnin assignment3@hy436 ask3
```

Regarding the code, comments are more than welcome to aid the examiner understand your code. The deadline for submission is: **07 December 2022, 23:59**. **This script should work exactly as expected to get full grade. Any working solution, under the constraints described in the assignment text, is acceptable. Points will be subtracted for missing or incorrect functionality. The penalty for late submission is 10% per day. The submitted code will be tested for plagiarism using plagiarism-detection software. Any attempt to plagiarize will be accordingly punished with 0 grade. The exercise weight is 25% of the overall lab grade.**

3.10 Debriefing

All the students who have submitted their code are requested to attend the debriefing session, where a sample solution will be presented by the assistants and a discussion of the exercise will follow. The date of the debriefing session will be announced via announcement in Moodle forum.

3.11 Oral Exam

All the students who have submitted their code are requested to attend the oral exam session, in order to present their solutions to the teaching assistants. A timeslot during the oral exam session will be assigned to each student using Doodle.
Attention:

- **Each student will only be examined during the timeslot assigned.**
- **During this session both the Assignments 3 and 4 will be examined.**
- **Both the timely submission and the oral exam session will contribute to the grading of the assignment.**

3.12 Asking for help

For any issues you may encounter regarding the exercise, please ask the teaching assistants during the exercise sessions (14:00-16:00) on 18/11, 24/11 and 01/12, or post your question on the Moodle forum. Please do not post raw code snippets on Moodle!!! **Before contact, please make sure that you have formulated your question clearly and that you have already studied [22, 11]. Good luck!**

Acknowledgments

This assignment is based on material from [22, 11, 13]. The exercise was written and tested by Dr. Vasileios Kotronis from ICS-FORTH.

References

[1] An implementation framework for a P4Runtime server. https://github.com/p4lang/PI.

[2] Link for downloading official tutorial P4 VM. https://tinyurl.com/hy436P4vm.

[3] Mininet official website. http://mininet.org/.

[4] Mininet Walkthrough. http://mininet.org/walkthrough/.

[5] Oracle VirtualBox Downloads. https://www.virtualbox.org/wiki/Downloads.

[6] P4 publications. https://p4.org/publications/.

[7] P4lang behavioral model (bmv2). https://github.com/p4lang/behavioral-model.

[8] P4lang P4_16 prototype compiler . https://github.com/p4lang/p4c.

[9] Turnin User Guide. http://www.csd.uoc.gr/services/useful-info/use-the-turnin.html.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[11] Mihai Budiu and Chris Dodd. The P416 Programming Language. *Operating Systems Review*, 51(1):5–14, 2017.

[12] P4 Language Consortium. Official P4 Website. https://p4.org/code/.

[13] P4 Language Consortium. P4 language tutorials: GitHub repository. https://github.com/p4lang/tutorials.

[14] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 5. ACM, 2015.

[15] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42. ACM, 2017.

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2017.

[17] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[18] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28. ACM, 2017.

[19] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 150–156. ACM, 2017.

[20] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28. ACM, 2016.

[21] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. Dc. p4: Programming the forwarding plane of a data-center switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 2. ACM, 2015.

[22] Noa Zilberman, Nate Foster, Theo Jepsen, Milad Sharif, Robert Soule, and Pietro Bressane. P4 Tutorial. https://github.com/p4lang/tutorials/blob/master/P4_tutorial.pdf.