Chapter 5

Lecture 5: P4

Download the files for these exercises here: `https://surfdrive.surf.nl/files/index.php/s/ftTe75WFx7n2JYk` Extract this archive in `~/HPDN_Exercises/`. This should create a new directory `week_5` in which you can do the exercises.

5.1 Acknowledgement

This exercise is derived from the official P4 tutorial Github repository.

5.2 Intro to P4 Programming Language

Programming Packet-independent Packet Processing (P4) is a domain-specific language used for network devices to specify the data plane functionalities inside networks.

This chapter provides a brief introduction on how to use the P4 programming language to achieve simple network functionalities. It also gives you several exercises to practice the use of the language and to get familiar with the wide variety of functionalities you can implement. The following exercises are included in this chapter:

1. Basic Forwarding and Tunneling
2. In-band Network Telemetry
3. Stateful Network Function

5.2.1 Dependencies and exercise folder layout

The provided VM has all the dependencies installed to be able to run these exercises. If you are not using the VM we provided, you must install the following:

- P4 software switch (`https://github.com/p4lang/behavioral-model`) and all its dependencies
- P4 compiler (p4c, `https://github.com/p4lang/p4c`) and all its dependencies

For installation of all the dependencies, we recommend using the following GitHub repository that provides many helper scripts for installation `https://github.com/jafingerhut/p4-guide.git`.

The exercise for this week contains two folders, one called `utils` with helper functions for configuring the network and the programmable switches; and one called `exercises` that contains three exercise folders that correspond to each subsection of this exercise, namely, `basic_forwarding_and_tunneling`, `in_band_network_telemetry`, and `firewall`.

5.2.2 Architecture and Switch Binary

In this exercise, we will use the v1model architecture included inside the p4c compiler and the BMv2 software switch. More information on this architecture can be found at v1model and simple_switch. To be more precise, the `simple_switch_grpc` will be used, and is developed upon `simple_switch` such that it supports the P4Runtime API.

Generally, to write a P4 program with the v1model, you will need to implement the following six programmable blocks:

1. Parser
2. VerifyChecksum

3. Ingress
4. Egress
5. ComputeChecksum
6. Deparser

Furthermore, you will need to define all the metadata and header fields that you will use throughout your program at the beginning. Thus, the general outline of your program should look like this, and more details of what should be inside each code block will be given in the latter part of this document:

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

//*********************** H E A D E R S ***************************************************************
//header definitions and metadata

struct metadata {
}

struct headers {
}

//************************ P A R S E R ***************************************************************
parser MyParser(packet_in packet, out headers hdr, inout metadata meta,
                inout standard_metadata_t standard_metadata) {
}

//************ C H E C K S U M V E R I F I C A T I O N *************************************************
control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
    apply {}
}

//************** I N G R E S S P R O C E S S I N G *****************************************************
control MyIngress(inout headers hdr, inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    apply {}
}

//*************** E G R E S S P R O C E S S I N G *****************************************************
control MyEgress(inout headers hdr, inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {}
}

//************* C H E C K S U M C O M P U T A T I O N *************************************************
control MyComputeChecksum(inout headers hdr, inout metadata meta) {
     apply {}
}

//************************ D E P A R S E R ***********************************************************
control MyDeparser(packet_out packet, in headers hdr) {
    apply {}
}


V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
```

```
MyDeparser()
) main;
```

5.3 Exercise 1. - P4 Basic Forwarding and Tunneling

In this exercise, you will first implement basic layer-3 forwarding and then add tunneling functionality between two hosts. Since this is your first P4 program, we will guide you step by step. The file `basic.p4` contains the base code file that we provide for you to complete this exercise.

5.3.1 Basic Forwarding

Topology

This section explains a very simple topology used for this exercise that contains three different hosts and one switch. Figure 5.1 depicts the topology. Your task is to enable basic forwarding between $h1$, $h2$, and $h3$, as well as tunneling between $h1$ and $h3$ such that messages sent with the tunneling identifier are tunneled to $h3$.

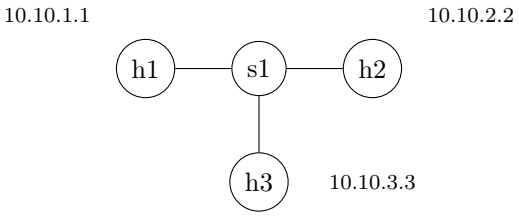

Figure 5.1: Topology - Exercise 1

Basic Forwarding

For this part of the exercise, you will be required to implement basic forwarding among the hosts, and we will guide you through the entire program for creating your very first P4 program.

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 our basic forwarding program, we only need to define Ethernet and IP as follows:

```
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;
}
```

Each line contains a field name preceded by the field length in bits.

Additionally, to be able to use the previously defined headers, we must populate the struct “headers” with an instance of each previously defined header:

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

Metadata

Metadata contains local information about the packet that is not normally stored by the packet headers (e.g., ingress port, egress port, or a timestamp). Metadata is defined in a similar way as headers by creating a struct:

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

However, in this exercise, we will not need any custom metadata. Consequently, you can leave this struct empty. However, we will use the metadata supplied by the switch architecture (called `standard_metadata`). **Note that we don't need to explicitly define the standard metadata, since it is already defined inside the v1model**.

```
struct standard_metadata_t {

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

The `standard_metadata` structure does not need to be defined in the P4 program, as it is implicitly defined and created at each packet reception. Out of all of these, while defining our custom processing blocks (e.g., actions, tables), we will only use the `egress_spec` field, defining which output port a packet will go to. More information on each of these fields can be found inside the v1model file located at `https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4`, and in the simple switch documentation `https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md`.

Parser

The next step in writing a P4 program is to define a parser. A parser defines the order in which the previously defined headers can be found in the packet. For example, consider the code provided below. First, each packet's Ethernet header is parsed (start state). As an output, the switch populates the Ethernet fields according to the bits specified in the header: `dstAddr`, `srcAddr`, and `etherType`. Next, depending on the value of `etherType`, the packet either exits the parser block (default accept) or continues to the next state (`parse_ipv4`). For this exercise, there is no transition after `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 parsing ip header */
        }
    }
}
```

Exercise: Fill in the missing parts yourself (the state transition and the `parse_ipv4`).

Verifying the Checksum

The VerifyChecksum block is executed after the previously defined parser and just before the Ingress control begins. In this block, you can choose to check the received packets' Checksums or leave the block empty (ignoring any potential problems). For example, using the code below, we can verify the checksum of the IPv4 packets.

```
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);
    }
}
```

However, in the v1model architecture, the switch will not automatically drop a packet having an incorrect checksum. Instead, it will update the value of the `checksum_error` field (in the standard metadata) to 1. Thus, you as the developer can specify the way the switch will handle these packets. However, since we do not intend to handle these packets differently, you can leave this block empty.

Ingress Control Flow

The next step to write a P4 program is to define the ingress control flow. The `ingress` block specifies all tables applied on a packet after the network node receives it (e.g. filter, determine the output port). Here is an overview of the function block that takes in a *header*, a *metadata* field and a *standard_metadata* field.

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

    }
}
```

In the ingress control flow, we can also define match-action tables and control actions needed for the correct operation. These will be further explained in the next two subsections.

Match-Action tables

Similarly to OpenFlow flow entries, P4 tables associate lookup keys and actions. Lookup keys are any combination of header fields and metadata to be examined to find a match. Each match's outcome is an action that specifies operations to be performed on the packet. For example, the next code is a longest prefix-matching table that is based on the destination IP address parsed in the header. There are three match kinds defined by the P4 core library, namely, `exact`, `ternary`, and `lpm`.

```
    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
```

```
        }
        size = 1024;
        default_action = NoAction();
    }
```

Actions

We defined two actions that can be the outcome of the lookup process: `drop_packet` and `ipv4_forward`. As the next step, we need to decide what exactly each of these actions will do. This can be done as follows:

```
    action drop() {
        mark_to_drop(standard_metadata);
    }
    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port){
        // TODO: fill this part for basic forwarding
    }
```

In P4, actions can only contain other actions (including some default primitive actions defined by the architecture). For more information on what statements you can have inside your actions, look at the P4 specification (`https://p4.org/p4-spec/docs/P4-16-v1.2.1.pdf`) and the v1model architecture: `https://raw.githubusercontent.com/p4lang/p4c/master/p4include/v1model.p4`

Exercise: Complete the `ipv4_forward` action yourself. You need to complete the following four steps:

1. Set the egress port for the next hop. Hint: the `egress_spec` field of the `standard_metadata` controls the egress port of the packet. **Notte:** You might want to refer to the `standard_metadata` struct.
2. Update the Ethernet destination address with the address of the next hop.
3. Update the Ethernet source address with the address of the switch.
4. Decrement the TTL.

Now that you have completed the actions and match-action table part of the Ingress control block, you can proceed to finish the apply block inside the ingress control block. You would need to determine when to apply the match action table that you've created earlier.

Exercise: Complete the Ingress control block yourself. **Hint:** You only need to apply the match-action table when the IP header is valid and you can use the library function provided by the P4 core library, `isValid()`.

Egress control flow

In contrast to the ingress block, the `egress` block can be used to specify all tables applied at the output port before the packet is sent out (e.g., rewrite the MAC address). However, since this is a really simple program and no further actions are required to be applied on outgoing packets, this block can remain empty.

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

Updating the Checksum

Similarly to the verify checksum block, you can choose to update any of the checksums at the end of the packet processing (before the deparser). Thus, if your program modified any of the IPv4 header fields, you can update the checksum so that the following switch in the path or the end host would not potentially drop it, using the code below:

```
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);
    }
}
```

Since our firewall will not update any IP header fields, the packet's checksum field will contain the correct value. Consequently, this block should be left empty to simplify the program run at the switch.

Deparser

The deparser is the last part of our P4 program, we will add the code of the block that performs the exact opposite role of the parser. This block defines the order of the headers in the packet after it is sent. This block is restricted to emit methods only and a sequence of calls to the this method.

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        //Fill in this part yourself
    }
}
```

Exercise: Fill in the missing parts yourself. **Hint:** Since we only process the Ethernet and IP headers, we only need to emit those.

Running and testing your code

Congratulations! You have just completed your very first P4 program that enabled basic IP-layer forwarding. Now you can compile your program and test it in the Mininet environment. We've provided you with the `util` code and a `Makefile` for testing to be easier. You can compile your P4 code and start Mininet with the topology specified within the `pod-topo` folder with the following command:

```
make run
```

This will compile your p4 code and start the Mininet prompt, and you should now be able to test using a command such as:

```
mininet> h1 ping h2
```

You should be able to see ping messages being able to traverse among the three different hosts.

5.3.2 Basic Tunneling

This basic tunneling exercise develops based on the basic forwarding program that you have just completed earlier, and utilizes the same topology as the previous part. It allows a tunneling connection to $h3$, such that all the packets sent with the tunneling identified will be tunneled to $h3$.

Below we will guide you toward completing the tunneling functionality, but with less hints than the previous part.

Header modification

For implementing the tunneling functionality, the first step is to modify the header declaration and add tunneling headers to it such that there exists a way of identifying which packets need to be tunneled within the network. You can add the following header declarations to your header part of the program, and also modify the header struct such that tunnel header can be parsed correctly. We also recommend you to define a constant number for the type for tunneling messages.

Moreover, we use a 16-bit protocol id to indicate the type of packets being encapsulated within the tunnel packet. In the scope of this exercise, we only support ipv4 packets so you only need to handle ipv4 packets being encapsulated while implementing the parser.

```
const bit<16> myTunnel_id = 0x1212

header myTunnel_t {
    bit<16> proto_id;
```

```
        bit<16> dst_id;
    }
```

Since tunneling packets and regular IP packets are different from each other, we can utilize the `etherType` field within the Ethernet header. Therefore, the size of the tunnel identifier can also be 16 bits.

Also modify the previously defined header struct to be like the following, such that the header has 3 different members:

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

Updating the Parser

The current parser does not support parsing the tunnel header. In order to incorporate with the new header type, the parser needs to be changed accordingly. You need to do the following:

1. Add transition in `parse_ethernet` based on the `etherType`. In this exercise, we'll use a special identifier `0x1212` (16 bits) for identifying tunneling packet.
2. Add a new parser state to extract the `myTunnel` header and transition based on the protocol id of the tunnel header to either `parse_ipv4` state or `accept`.

Exercise: Update the parser to be able to parse Tunnel header.

Updating the Ingress

In the basic forwarding program, the packets will be forwarded based on the longest prefix-match rule to different ports. Now we need to take a different action, such that when the destination id matches correctly, the tunneling packets need to be all forwarded to a specific port. The following needs to be implemented:

1. Add an action called `myTunnel_forward` with parameter of `egressSpec_t` type called `port` to set the egress port.
2. Add a tunneling match-action table called `myTunnel_exact` and apply action called `myTunnel_forward` to forward the packet when the `dst_id` matches exactly.
3. Apply the tunneling table for tunneling packets and make sure normal IP packets are processed only by the IPv4 match-action table.

Update Deparser

When sending out the packet don't forget to emit the `myTunnel` header back to the packet.

Exercise: Update the Ingress block based on the previous description.

Running and testing your solution

1. Run the following to compile and start a Mininet instance with the same topology as the previous `basic_forwarding` exercise.

   ```
   make tunneling
   ```

2. You should see a Mininet prompt. Open three xTerm for `h1`, `h2` and `h3`:

   ```
   mininet> xterm h1 h2 h3
   ```

3. On both `h2` and `h3`'s terminals start the listening server by:

   ```
   ./receive.py
   ```

4. First try testing the data without tunneling by sending a regular message from `h1` to `h2`, you should observe at `h2` that it did receive the message:

   ```
   ./send.py 10.0.2.2 "I_love_HPDN"
   ```

5. Then trying sending another message from `h1` to `h2`. However, this time we also include the tunneling functionality. You should be able to observe that even when the packet is sending for `h2`, the packet will actually received by `h3`.

```
./send.py 10.0.2.2 "I_love_HPDN_;D" --dst_id 3
```

5.4 Exercise 2. - P4 In-band Network Telemetry

Traditional SDN monitoring tools and protocols usually depend on a remote server to operate, which makes it hard to provide lower-than-millisecond accuracy and per-packet operation in the network. However, with the emergence of P4, this becomes feasible. In-band Network Telemetry has been a key application for programmable switches. In this exercise, you will implement the following functionality for `Explicit Congestion Notification`.

For this exercise, we are using the topology you can see below. **Note:** We have limited the bandwidth of the link between s1 and s2 to 512kbps in `topology.json`:

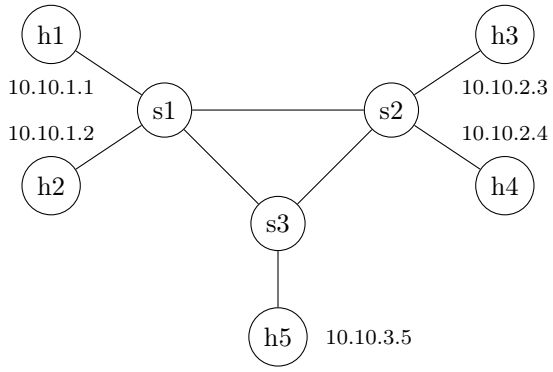

Figure 5.2: Topology - INT exercise.

There are basic templates for this exercise in the folder `inband_network_telemetry`

5.4.1 Explicit Congestion Notification

The objective of the explicit congestion notification is to extend the basic forwarding and allow end-to-end notification of network congestion without packets being dropped. As in the previous exercise, we have already handled and provided you with the control plane routing rules such that you only need to implement the data plane rules.

In order to achieve explicit congestion notification, the network utilizes the outdated `type of service (TOS)` field inside the IPv4 header and splits it into `DiffServ` and `ECN` fields. If the end host supports explicit congestion notification, the host set the value of 1 or 2 in the `ipv4.ecn` field to allow the network device to identify the data. Once congestion is detected, the network device can modify this value in the packet to notify the receiver of the packets. The receiver then copies and sends the same value to the original sender for it to lower the transmitting rate.

You will be developing the functionality described above using the `int.p4` base code that we provided to you which only supports L3-level forwarding at the moment.

Updating the headers

The very first step of completing this exercise is to update the IPv4 header definition and split the `tos` field into `diffServ` and `ecn` fields that are 6 bits and 2 bits correspondingly.

For simplicity of the program, we have defined a constant number at the beginning of the program as the threshold of the ECN that will be used later when comparing the queue length. The size of the number is chosen based on the size of the queue that is specified in the `standard_metadata` provided by the `v1model`.

```
const bit<19> ECN_threshold = 10; // NOTE: 10 being a temporary place holder
```

Exercise: Update the following IPv4 header

```
header ipv4_t {
    bit<4> version;
    bit<4> ihl;
    bit<8> tos; // TODO: split into 6 bits field of diffServ and 2 bits ecn
    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;
}
```

Parser, Checksum, and Ingress

Since this application does not require any operation in these blocks, no changes are necessary within the `parser`, `verifyChecksum`, and `MyIngress`.

Updating Egress control block

In the previous explanation, we've described that we use three for indicating the network is congested and 1 or 2 for indicating that the host supports `ecn`. To correctly perform the operations needed for the egress control block, you need to implement the following:

1. Add an action to change the value of `ipv4.ecn` to 3, indicating the network is congested.
2. Apply the logic to first determine if the `ecn` value in the ipv4 header is 1 or 2, then check whether the queue length is above the threshold.
3. If the queue length is greater than the threshold, then take the action in step one.

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {
        /*
         * TODO:
         * - if ecn is 1 or 2
         * - compare standard_metadata.enq_qdepth with threshold
         * and set hdr.ipv4.ecn to 3 if larger
         */
    }
}
```

Updating the Checksums

Since the header of the IPv4 packet is modified, the way of computing `checksum` needs to be updated correspondingly to split the `tos` field of the IPv4 packets into `diffServ` and `ecn`.

Exercise: Update the Checksum computation.

Running and testing your solution

This section guides you on how you are able to run and test the P4 program that you've just created. For testing the `ecn`, we want to send a high rate of traffic from h2 to h3. In the meantime, we also use low-rate traffic from `h1` to `h4`. Since the `s1` to `s2` link is the bottleneck, we should be able to observe the correct `ecn` value at `h4`.

1. By running the following command, a Mininet instance with the topology specified above will be started.

   ```
   make
   ```

2. You should be seeing a Mininet prompt and you can run the following commands to start terminals for individual hosts, for example:

   ```
   mininet> xterm h1 h2 h3 h4
   ```

3. In `h4`'s terminal, start server that capture packets, and redirect the output to `h4_capture.log`. We've provide a helper file called `receive.py`:

```
./receive.py > h4_catpure.log
```

4. In `h3`'s terminal, start a iperf UDP server by running the following command

```
iperf −s −u
```

5. In `h1`'s terminal, use the script that we provided, `send.py` (The script will set the `ecn` value to 1) to send one packet per second for 1 min:

```
./send.py 10.0.2.4 "I_love_HPDN" 60
```

6. In `h2`'s terminal, start a iperf UDP sender service by running the following command. This is to populate the network link between `s1` and `s2`.

```
iperf −c 10.0.2.3 −u
```

7. Verify the change of `tos` value in the log file in a separate window by running:

```
grep tos h4_catpure.log
```

Exercise: Run your solution with the instructions above and verify that your explicit congestion notification is working.

5.5 Exercise 3. - Simple P4 Firewall and Stateful P4 Firewall

In this exercise, you will need to use the knowledge you've learned so far about programmable switches and P4 to first create a simple firewall application, and secondly extend the application to become a more advanced stateful firewall application.

5.5.1 Simple P4 Firewall

In the first part of this exercise, you will write a simple P4 program that acts as a firewall that filters packets based on the IP source address and destination ports. For your implementation of the simple firewall, access should be granted only to:

- IP hosts under the 10.10.10.0/24 subnet establishing a TCP connection to the server with the IP address 10.0.2.3 using ports 80 (HTTP) or 443 (HTTPS).
- UDP packets with the destination port 53 (DNS)

There are basic templates for this exercise in the folder `firewall` of the P4 exercise folder.

Updating the headers, parser, and deparser once more

Since the firewall application is not only limited to permitting TCP connections but also UDP packets to support DNS queries, the parser and the deparser need to be updated to handle UDP data.

Exercise: Update parser and deparser to support both TCP and UDP traffic.

The Ingress Control Flow

For the ingress control block, you need to define two match action tables. One will be used for TCP connections and the other for UDP packets. Then, apply the two tables you defined earlier when the header type is valid. This part of the exercise will come after you define the match-action tables.

Match-Action tables

In our firewall application, we want to filter TCP and UDP packets.

For these protocols, you need to operate on specific header fields (e.g., source IP address to allow hosts from the 10.10.10.0/24 subnet, destination TCP port to allow ports 80 and 443). These fields form a lookup key inside the match-action table. Furthermore, based on these values, the switch should either drop or forward the packet. Thus, the switch should implement two actions `drop_packet` and a `forward_packet` function. **Note:** The actions `drop_packet` and `forward_packet` have already been defined for you.

For UDP packets, we only want the UDP packets that reach port 53 for DNS queries. You need to follow the exercise instructions to complete these parts. Below is the part of the skeleton code that we provided.

```
    table tcp_firewall {
        key = {
                hdr.ipv4.srcAddr: ternary;
            //Fill in this part yourself
        }
        actions = {
            drop_packet;
            forward_packet;
        }
        size = 1024;
        default_action = drop_packet();
    }

    table udp_firewall {
        //Fill in this part yourself
    }
```

Exercise: Fill in the missing parts according to the description below.

1. Update the lookup key of the `tcp_firewall` table. Keep in mind that TCP is a connection-oriented protocol, i.e., make sure that hosts are able to establish a connection.

2. Complete the table `udp_firewall` that will match the correct header fields and apply the correct actions for UDP traffic.

Applying the match action tables

Now that the tables are defined, these tables need to be properly applied in the ingress processing block.
Exercise: Apply the two match-action tables when the associated headers are valid.

```
if(hdr.ipv4.isValid(){
    // TODO: Check if header type is valid.
        // TODO: Apply your TCP connection table here
    // TODO: Check if header type is valid.
        // TODO: Apply your UDP table here.
}
```

Egress Control Flow and Checksum

Since our firewall application does not need to apply any actions on outgoing packets or check the checksum of the packets, these blocks can be left as they are at the moment.

Control-plane rules

For SDN, the flow entries are installed by a centralized controller. Similarly, for programmable switches, P4 runtime service can be used to push control-plane rules (the table entries of match-action tables) onto the switches. If you would like to learn more about the P4 runtime services, you can read the specification at the following location: `https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html`.

In the previous parts of the P4 exercises, these rules have been provided by us, and for this part of the exercise, you need to complete these rules yourselves. Inside the `firewall` folder, the `topology.json` specifies the topology that Mininet will run, and `s1-runtime.json` specifies the runtime control plane rule for `s1` switch.
So far, we have defined the way our switch will process the incoming packets inside the P4 program. However, we have not yet associated how packets should be matched to specific actions. More specifically, what data to input into the action when a key is matched. Our next step will be to fill the match-action tables with rules that filter out any unwanted data, as specified at the start of this assignment.

Match-action table entries are filled at run-time. This can be done locally or by a centralized remote controller, depending on your use case. In this exercise, we will use statically defined rules to fill the table entries locally. You will need to complete specifying the control plane rules inside the file `s1-runtime.json`

The runtime control plane file uses JSON format for specifying the rules, you can refer to the runtime configuration file of previous P4 exercises for an example of the table entry. For example, the json format below is what you need for constructing a ternary match-kind table entry.

```
{
    "table": "table_name",
    "match":{
        "field_name": ["value", "ternary_mask"]
    },
    "action_name": "name_of_your_action",
    "action_params": {
        "name_of_the_parameter": value
    },
    "priority": non_zero_value
}
```

Note: Priority is required for the ternary match-kind. "Don't care" matches are omitted in the runtime json file.

Exercise: Finish adding all rules in `s1-runtime.json` to allow TCP traffic for IP hosts under 10.10.10.0/24 subnet establishing TCP connection to server at 10.0.2.3 using with port 80 and 443. Make sure to add all entries for both TCP, as well as UDP traffic for port 53.

Testing your firewall Application

Same as previous exercises, you can run your firewall code and Mininet start with the topology specified in `topology.json`, by running the following command, and it will do the following:

```
make
```

1. Re-compile the P4 program
2. This will start a topology with three Mininet hosts (with IP addresses 10.10.10.1, 10.10.10.2, and 10.0.2.3) connected to one P4 switch that is running the previously implemented P4 firewall program (specified with the JSON flag) shown in Fig. 5.3.

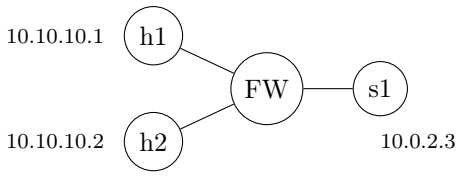

Figure 5.3: Topology - Exercise Simple firewall

3. Deploys control-plane rules specified in `s1-runtime.json` on the switches, populating the match-action tables.

Test your app as follows:

1. Start a new window for h3: `xterm h3`
2. On h3 start the netcat listener on port 80: `nc -l -p 80`
3. Start a new window for h1: `xterm h1`
4. Start a netcat client on h1: `nc 10.0.2.3 80`
5. Repeat the previous steps for different ports and IP addresses.
6. Send 5 UDP packets from h1 using the script UDPsend.py : `python UDPsend.py --dst-port 53`
7. Repeat step 6 by for different ports. Observe the output in h3 using `tcpdump -i eth0`

If the packets from steps 4 and 6 are the only packets received at `h3`, the app works as it is supposed to and you have finished your P4 application. If you also receive packets for other ports (other than 443, 80 and 53), or receive no traffic at all, the app is not functioning as it is supposed to.

`https://p4.org/p4-spec/docs/P4-16-v1.2.2.pdf` should provide you with all the information you need on writing more complex P4 applications. Additionally, the following two links will provide you with information you need to write control-plane rules using runtime_CLI and the software P4 switch you are using: `https://github.com/p4lang/behavioral-model/blob/master/docs/runtime_CLI.md` and `https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md`

Debugging options

You can use wireshark or tcpdump on host interfaces to examine the packets.

If both the switch is running correctly and the rules were added successfully, use the scripts: `TCPsend.py` and `UDPsend.py` to send custom TCP/UDP packets to the switch and observe the outcomes using tcpdump.

The Final Touch

If you are not using one single header for both TCP and UDP in the previous implementation, there is a way of combining the two headers into one. Rewrite the previous app by defining a single transport header (for both UDP and TCP)!

Hint: Analyze the header formats for TCP and UDP and create a general transport header. What UDP/TCP header fields are used by the firewall app? Where are they located? Remember that you can specify your custom header in P4.

5.5.2 Stateful Firewall

Extend your previous firewall program to allow packets to go through if they either match a known active connection or have the SYN flag set.

To complete this exercise, you will need to use two other P4 language elements, called **registers** and **hashes**.

Registers

Registers represent stateful memory. A register definition includes the length of the register cells in bits and the total number of cells. For example, a register table with 65535 rows of 4 bits can be defined as:

```
register<bit<4>>(65535) my_register;
```

Registers can be read from and written to by two specific actions:

```
my_register.read(meta.register_value, meta.index);
my_register.write(meta.index, meta.register_value)
```

Thus, updating a register always implies three actions:

1. Read the current register value into a metadata field
2. Modify the metadata value
3. Write the metadata value back in the register.

Hashes

Since this is the first time you encounter the hash function in P4, this portion of the document will demonstrate how to use a hash of some packet fields to calculate an index for a register array. The necessary steps are:

- Determining a field list you want to use for the calculation
- Defining the computation to be performed on it (the hash algorithm)
- Copying it into a user-defined metadata

For example, the code below calculates the CRC32 hash using the flow 5-tuple. Furthermore, to have the correct select value, it uses the modulo operation on the generated 32-bit hash and stores it in `meta.index` (metadata field):

```
    hash(meta.index,
         HashAlgorithm.crc32, (bit<16>)0,
         {hdr.ipv4.srcAddr,
          hdr.ipv4.dstAddr,
          hdr.tcp.srcPort,
          hdr.tcp.dstPort,
          hdr.ipv4.protocol},
          (bit<32>)TABLE_WIDTH);
```

Note that the metadata header field needs to be added and the value of `ecmp_base` and `ecmp_count` are specified in the switch runtime configuration file.

Exercise

Implement a stateful firewall based on your previous simple firewall application. Note that the control plane rules also need to be changed if you modify the match-action table.

Hint: The basic idea is to use a hash of the 5-tuple and a register array to track existing TCP connections.

Test your app using the same procedure as in the simple firewall exercise. Additionally, use the script `TCPsynsend.py`. This script will send 10 packets and you can change the SYN flags options to try different scenarios.

5.6 Useful links:

You can find many other examples on how to use P4 on `https://github.com/p4lang/tutorials`