**P4 EXERCISES WITH**

**DOCUMENTATION:**

Contents

[1)Basic.p4 4](#_heading=h.gjdgxs)

[About program 4](#_heading=h.30j0zll)

[Code: 6](#_heading=h.1fob9te)

[2)Basic tunnel.p4 9](#_heading=h.3znysh7)

[About program 9](#_heading=h.2et92p0)

[Code: 10](#_heading=h.tyjcwt)

[3)Calculator.p4 14](#_heading=h.3dy6vkm)

[About program 14](#_heading=h.1t3h5sf)

[Code: 15](#_heading=h.4d34og8)

[4)Explicit congestion notification.p4 19](#_heading=h.2s8eyo1)

[About program 19](#_heading=h.17dp8vu)

[Code: 20](#_heading=h.3rdcrjn)

[5)Firewall.p4 24](#_heading=h.26in1rg)

[About program 24](#_heading=h.lnxbz9)

[Code: 26](#_heading=h.35nkun2)

[6)Link monitor.p4 31](#_heading=h.1ksv4uv)

[About program 31](#_heading=h.44sinio)

[Code: 33](#_heading=h.2jxsxqh)

[7)Load balancing.p4 38](#_heading=h.z337ya)

[About program 38](#_heading=h.3j2qqm3)

[Code: 39](#_heading=h.1y810tw)

[8)Multi-Hop Route Inspection.p4 44](#_heading=h.4i7ojhp)

[About program 44](#_heading=h.2xcytpi)

[Code: 45](#_heading=h.1ci93xb)

[9)Multicast.p4 50](#_heading=h.3whwml4)

[About program 50](#_heading=h.2bn6wsx)

[Code: 51](#_heading=h.qsh70q)

[10)Quality of Service.p4 54](#_heading=h.3as4poj)

[About program 54](#_heading=h.1pxezwc)

[Code: 55](#_heading=h.49x2ik5)

[11)Reflector.p4 60](#_heading=h.2p2csry)

[About program 60](#_heading=h.147n2zr)

[Code: 62](#_heading=h.3o7alnk)

[12)Repeater.p4 64](#_heading=h.23ckvvd)

[Code: 65](#_heading=h.ihv636)

[13)Verifying checksum.p4 67](#_heading=h.32hioqz)

[Code: 68](#_heading=h.1hmsyys)

[14)Meter 72](#_heading=h.41mghml)

[(Direct meter.p4 & Indirect meter.p4) 72](#_heading=h.2grqrue)

[Code: 73](#_heading=h.vx1227)

[Direct meter.p4 73](#_heading=h.3fwokq0)

[Indirect meter.p4 75](#_heading=h.1v1yuxt)

[15)Counters 78](#_heading=h.4f1mdlm)

[(Direct counter.p4 & Indirect counter.p4) 78](#_heading=h.2u6wntf)

[Code: 79](#_heading=h.19c6y18)

[(Direct counter.p4) 79](#_heading=h.3tbugp1)

[(Indirect counter.p4) 80](#_heading=h.28h4qwu)

[16)Source routing.p4 83](#_heading=h.nmf14n)

[About program 83](#_heading=h.37m2jsg)

[Code: 84](#_heading=h.1mrcu09)

[17)IP Forwarding.P4 87](#_heading=h.46r0co2)

[Code: 88](#_heading=h.2lwamvv)

[18)Digest Messages.p4 91](#_heading=h.111kx3o)

[Code: 92](#_heading=h.206ipza)

[19)Heavy Hitter 95](#_heading=h.4k668n3)

[Code: 96](#_heading=h.3ygebqi)

[20)IP Forwarding two tables.p4 100](#_heading=h.sqyw64)

[Code: 101](#_heading=h.4bvk7pj)

[21)Stateful Firewall.p4 104](#_heading=h.2r0uhxc)

[Code: 105](#_heading=h.25b2l0r)

[22)Ternary Match.p4 110](#_heading=h.kgcv8k)

[Code: 111](#_heading=h.1jlao46)

[23)IP forwarding 2 examples.p4 114](#_heading=h.xvir7l)

[Code: 115](#_heading=h.1x0gk37)

[EXTRA EXAMPLES 118](#_heading=h.4h042r0)

[TO KNOW MORE ABOUT P4: 118](#_heading=h.2w5ecyt)

[\*L2 Basic Forwarding.p4 118](#_heading=h.1baon6m)

[Code: 119](#_heading=h.2afmg28)

[\*L2 learning.p4 121](#_heading=h.39kk8xu)

[Code: 123](#_heading=h.48pi1tg)

[\*L2 flooding.p4 126](#_heading=h.2nusc19)

[Code: 127](#_heading=h.2250f4o)

[\*Multiprotocol label switching.p4 (MPLS) 129](#_heading=h.319y80a)

[Code: 130](#_heading=h.40ew0vw)

[\*Recirculate.p4 135](#_heading=h.279ka65)

[Code: 136](#_heading=h.1ljsd9k)

[\*Recirculate and add headers.p4 139](#_heading=h.zu0gcz)

[Code: 140](#_heading=h.4iylrwe)

[\*Flowlet Switching.p4 142](#_heading=h.2y3w247)

[About program 142](#_heading=h.3x8tuzt)

[Code: 144](#_heading=h.2ce457m)

[\*Traceroutable.p4 147](#_heading=h.rjefff)

[Code: 149](#_heading=h.1qoc8b1)

[\*Simple routing (ECMP.P4) 153](#_heading=h.4anzqyu)

[Code: 155](#_heading=h.3oy7u29)

# 1)Basic.p4

**Implementing Basic Forwarding**

**Introduction**

The objective of this exercise is to write a P4 program that implements basic forwarding. To keep things simple, we will just implement forwarding for IPv4.

With IPv4 forwarding, the switch must perform the following actions for every packet: (i) update the source and destination MAC addresses, (ii) decrement the time-to-live (TTL) in the IP header, and (iii) forward the packet out the appropriate port.

Your switch will have a single table, which the control plane will populate with static rules. Each rule will map an IP address to the MAC address and output port for the next hop. We have already defined the control plane rules, so you only need to implement the data plane logic of your P4 program.

## About program

The basic.p4 file contains a skeleton P4 program with key pieces of logic replaced by TODO comments. Your implementation should follow the structure given in this file--- replace each TODO with logic implementing the missing piece.

A complete basic.p4 will contain the following components:

1. Header type definitions for Ethernet (ethernet\_t) and IPv4 (ipv4\_t). 2. **TODO:** Parsers for Ethernet and IPv4 that populate ethernet\_t and ipv4\_t fields. 3. An action to drop a packet, using mark\_to\_drop().

4. **TODO:** An action (called ipv4\_forward) that:

i. Sets the egress port for the next hop.

ii. Updates the ethernet destination address with the address of the next hop. iii. Updates the ethernet source address with the address of the switch. iv. Decrements the TTL.

5. **TODO:** A control that:

i. Defines a table that will read an IPv4 destination address, and invoke either drop or ipv4\_forward.

ii. An apply block that applies the table.

6. **TODO:** A deparser that selects the order in which fields inserted into the outgoing packet.

7. A package instantiation supplied with the parser, control, and deparser.

**To compile the basic.p4 and bring up a switch in Mininet to test its behavior.**

1. In your shell, run:

make run

This will:

o compile basic.p4, and

o start the pod-topo in Mininet and configure all switches with the appropriate P4 program + table entries, and

o configure all hosts with the commands listed in pod-topo/topology.json

2. You should now see a Mininet command prompt. Try to ping between hosts in the topology:

mininet> h1 ping h2

mininet> pingall

3. Type exit to leave each xterm and the Mininet command line. Then, to stop mininet: make stop

And to delete all pcaps, build files, and logs:

make clean

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  } }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  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(); }  } }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  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);  } }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  } }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 2)Basic tunnel.p4

**Implementing Basic Tunneling**

**Introduction**

In this exercise, we will add support for a basic tunneling protocol to the IP router that you completed in the previous assignment. The basic switch forwards based on the destination IP address. Your jobs is to define a new header type to encapsulate the IP packet and modify the switch code, so that it instead decides the destination port using a new tunnel header.

The new header type will contain a protocol ID, which indicates the type of packet being encapsulated, along with a destination ID to be used for routing.

## About program

Your job will be to do the following:

1. **NOTE:** A new header type has been added called myTunnel\_t that contains two 16-bit fields: proto\_id and dst\_id.

2. **NOTE:** The myTunnel\_t header has been added to the headers struct. 3. **TODO:** Update the parser to extract either the myTunnel header or ipv4 header based on the etherType field in the Ethernet header. The etherType corresponding to the myTunnel header is 0x1212. The parser should also extract the ipv4 header after the myTunnel header if proto\_id == TYPE\_IPV4 (i.e. 0x0800).

4. **TODO:** Define a new action called myTunnel\_forward that simply sets the egress port (i.e. egress\_spec field of the standard\_metadata bus) to the port number provided by the control plane.

5. **TODO:** Define a new table called myTunnel\_exact that perfoms an exact match on the dst\_id field of the myTunnel header. This table should invoke either the myTunnel\_forward action if the there is a match in the table and it should invoke the drop action otherwise.

6. **TODO:** Update the apply statement in the MyIngress control block to apply your newly defined myTunnel\_exact table if the myTunnel header is valid. Otherwise, invoke the ipv4\_lpm table if the ipv4 header is valid.

7. **TODO:** Update the deparser to emit the ethernet, then myTunnel, then ipv4 headers. Remember that the deparser will only emit a header if it is valid. A header's implicit validity bit is set by the parser upon extraction. So there is no need to check header validity here.

8. **TODO:** Add static rules for your newly defined table so that the switches will forward correctly for each possible value of dst\_id. See the diagram below for the topology's port configuration as well as how we will assign IDs to hosts. For this step you will need to add your forwarding rules to the sX-runtime.json files.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_MYTUNNEL = 0x1212;  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 myTunnel\_t {  bit<16> proto\_id;  bit<16> dst\_id;  }  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;  myTunnel\_t myTunnel;  ipv4\_t ipv4;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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\_MYTUNNEL: parse\_myTunnel;  TYPE\_IPV4: parse\_ipv4;  default: accept; }  }  state parse\_myTunnel {  packet.extract(hdr.myTunnel);  transition select(hdr.myTunnel.proto\_id) {  TYPE\_IPV4: parse\_ipv4;  default: accept; }  }  state parse\_ipv4 {  packet.extract(hdr.ipv4);  transition accept; }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  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();  }  action myTunnel\_forward(egressSpec\_t port) {  standard\_metadata.egress\_spec = port;  }  table myTunnel\_exact {  key = {  hdr.myTunnel.dst\_id: exact;  }  actions = {  myTunnel\_forward;  drop;  }  size = 1024;  default\_action = drop();  }  apply {  if (hdr.ipv4.isValid() && !hdr.myTunnel.isValid()) {  // Process only non-tunneled IPv4 packets  ipv4\_lpm.apply();  }  if (hdr.myTunnel.isValid()) {  // process tunneled packets  myTunnel\_exact.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 {  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);  }}  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.myTunnel);  packet.emit(hdr.ipv4);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 3)Calculator.p4

**Implementing a P4 Calculator**

**Introduction**

The objective of this tutorial is to implement a basic calculator using a custom protocol header written in P4. The header will contain an operation to perform and two operands. When a switch receives a calculator packet header, it will execute the operation on the operands, and return the result to the sender.

## About program

As a first step, compile the incomplete calc.p4 and bring up a switch in Mininet to test its behavior.

1. In your shell, run:

make

This will:

o compile calc.p4, and

o start a Mininet instance with one switches (s1) connected to two hosts (h1, h2).

o The hosts are assigned IPs of 10.0.1.1 and 10.0.1.2.

2. We've written a small Python-based driver program that will allow you to test your calculator. You can run the driver program directly from the Mininet command prompt:

mininet> h1 python calc.py >

|  |
| --- |
| Code: #include <core.p4>  #include <v1model.p4>  \* Define the headers the program will recognize  \* Standard ethernet header  header ethernet\_t {  bit<48> dstAddr;  bit<48> srcAddr;  bit<16> etherType; }  /\*  \* This is a custom protocol header for the calculator. We'll use  \* ethertype 0x1234 for is (see parser)  \*/  const bit<16> P4CALC\_ETYPE = 0x1234;  const bit<8> P4CALC\_P = 0x50; // 'P'  const bit<8> P4CALC\_4 = 0x34; // '4'  const bit<8> P4CALC\_VER = 0x01; // v0.1  const bit<8> P4CALC\_PLUS = 0x2b; // '+'  const bit<8> P4CALC\_MINUS = 0x2d; // '-'  const bit<8> P4CALC\_AND = 0x26; // '&'  const bit<8> P4CALC\_OR = 0x7c; // '|'  const bit<8> P4CALC\_CARET = 0x5e; // '^'  header p4calc\_t {  bit<8> p;  bit<8> four;  bit<8> ver;  bit<8> op;  bit<32> operand\_a;  bit<32> operand\_b;  bit<32> res;  }  /\*  \* All headers, used in the program needs to be assembed into a single struct. \* We only need to declare the type, but there is no need to instantiate it,  \* because it is done "by the architecture", i.e. outside of P4 functions  \*/  struct headers {  ethernet\_t ethernet;  p4calc\_t p4calc;  }  /\*  \* All metadata, globally used in the program, also needs to be assembed  \* into a single struct. As in the case of the headers, we only need to  \* declare the type, but there is no need to instantiate it,  \* because it is done "by the architecture", i.e. outside of P4 functions  \*/  struct metadata {  /\* In our case it is empty \*/  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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) {  P4CALC\_ETYPE : check\_p4calc;  default : accept;  }  }  state check\_p4calc {  transition select(packet.lookahead<p4calc\_t>().p,  packet.lookahead<p4calc\_t>().four,  packet.lookahead<p4calc\_t>().ver) {  (P4CALC\_P, P4CALC\_4, P4CALC\_VER) : parse\_p4calc;  default : accept;  }  }  state parse\_p4calc {  packet.extract(hdr.p4calc);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {    action send\_back(bit<32> result) {  bit<48> tmp;  /\* Put the result back in \*/  hdr.p4calc.res = result;  /\* Swap the MAC addresses \*/  tmp = hdr.ethernet.dstAddr;  hdr.ethernet.dstAddr = hdr.ethernet.srcAddr;  hdr.ethernet.srcAddr = tmp;  /\* Send the packet back to the port it came from \*/  standard\_metadata.egress\_spec = standard\_metadata.ingress\_port;  }  action operation\_add() {  send\_back(hdr.p4calc.operand\_a + hdr.p4calc.operand\_b);  }  action operation\_sub() {  send\_back(hdr.p4calc.operand\_a - hdr.p4calc.operand\_b);  }  action operation\_and() {  send\_back(hdr.p4calc.operand\_a & hdr.p4calc.operand\_b);  }  action operation\_or() {  send\_back(hdr.p4calc.operand\_a | hdr.p4calc.operand\_b);  }  action operation\_xor() {  send\_back(hdr.p4calc.operand\_a ^ hdr.p4calc.operand\_b);  }  action operation\_drop() {  mark\_to\_drop(standard\_metadata);  }  table calculate {  key = {  hdr.p4calc.op : exact;  }  actions = {  operation\_add;  operation\_sub;  operation\_and;  operation\_or;  operation\_xor;  operation\_drop;  }  const default\_action = operation\_drop();  const entries = {  P4CALC\_PLUS : operation\_add();  P4CALC\_MINUS: operation\_sub();  P4CALC\_AND : operation\_and();  P4CALC\_OR : operation\_or();  P4CALC\_CARET: operation\_xor();  }  }  apply {  if (hdr.p4calc.isValid()) {  calculate.apply();  } else {  operation\_drop();  } }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  packet.emit(hdr.ethernet);  packet.emit(hdr.p4calc);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 4)Explicit congestion notification.p4

**Implementing ECN**

**Introduction**

The objective of this tutorial is to extend basic L3 forwarding with an implementation of Explicit Congestion Notification (ECN).

ECN allows end-to-end notification of network congestion without dropping packets. If an end-host supports ECN, it puts the value of 1 or 2 in the ipv4.ecn field. For such packets, each switch may change the value to 3 if the queue size is larger than a threshold. The receiver copies the value to sender, and the sender can lower the rate.

## About program

A complete ecn.p4 will contain the following components:

1. Header type definitions for Ethernet (ethernet\_t) and IPv4 (ipv4\_t).

2. Parsers for Ethernet, IPv4,

3. An action to drop a packet, using mark\_to\_drop().

4. An action (called ipv4\_forward), which will:

i. Set the egress port for the next hop.

ii. Update the ethernet destination address with the address of the next hop. iii. Update the ethernet source address with the address of the switch. iv. Decrement the TTL.

5. An egress control block that checks the ECN and standard\_metadata.enq\_qdepth and sets the ipv4.ecn.

6. A deparser that selects the order in which fields inserted into the outgoing packet. 7. A package instantiation supplied with the parser, control, checksum verification and recomputation and deparser.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<8> TCP\_PROTOCOL = 0x06;  const bit<16> TYPE\_IPV4 = 0x800;  const bit<19> ECN\_THRESHOLD = 10;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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<6> diffserv;  bit<2> 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;  }  struct metadata {  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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, }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  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 = NoAction();  }  apply {  if (hdr.ipv4.isValid()) {  ipv4\_lpm.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) {  action mark\_ecn() {  hdr.ipv4.ecn = 3;  }  apply {  if (hdr.ipv4.ecn == 1 || hdr.ipv4.ecn == 2){  if (standard\_metadata.enq\_qdepth >= ECN\_THRESHOLD){  mark\_ecn();  } }  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  update\_checksum(  hdr.ipv4.isValid(),  { hdr.ipv4.version,  hdr.ipv4.ihl,  hdr.ipv4.diffserv,  hdr.ipv4.ecn,  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); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 5)Firewall.p4

**Implementing A Basic Stateful Firewall**

**Introduction**

The objective of this exercise is to write a P4 program that implements a simple stateful firewall. To do this, we will use a bloom filter. This exercise builds upon the basic exercise so be sure to complete that one before trying this one.

We will use the pod-topology for this exercise, which consists of four hosts connected to four switches, which are wired up as they would be in a single pod of a fat tree topology.

## About program

The firewall.p4 file contains a skeleton P4 program with key pieces of logic replaced by TODO comments. Your implementation should follow the structure given in this file --- replace each TODO with logic implementing the missing piece.

A complete firewall.p4 will contain the following components:

1. Header type definitions for Ethernet (ethernet\_t), IPv4 (ipv4\_t) and TCP (tcp\_t).

2. Parsers for Ethernet, IPv4 and TCP that populate ethernet\_t, ipv4\_t and tcp\_t fields.

3. An action to drop a packet, using mark\_to\_drop().

4. An action (called compute\_hashes) to compute the bloom filter's two hashes using hash algorithms crc16 and crc32. The hashes will be computed on the packet 5-tuple consisting of IPv4 source and destination addresses, source and destination port numbers and the IPv4 protocol type.

5. An action (ipv4\_forward) and a table (ipv4\_lpm) that will perform basic IPv4 forwarding (adopted from basic.p4).

6. An action (called set\_direction) that will simply set a one-bit direction variable as per the action's parameter.

7. A table (called check\_ports) that will read the ingress and egress port of a packet (after IPv4 forwarding) and invoke set\_direction. The direction will be set to 1, if the packet is incoming into the internal network. Otherwise, the direction will be set to 0. To achieve this, the file pod-topo/s1-runtime.json contains the appropriate control plane entries for the check\_ports table.

8. A control that will:

i. First apply the table ipv4\_lpm if the packet has a valid IPv4 header.

ii. Then if the TCP header is valid, apply the check\_ports table to determine the direction.

iii. Apply the compute\_hashes action to compute the two hash values which are the bit positions in the two register arrays of the bloom filter

(reg\_pos\_one and reg\_pos\_two). When the direction is 1 i.e. the packet is incoming into the internal network, compute\_hashes will be invoked by

swapping the source and destination IPv4 addresses and the source and destination ports. This is to check against bloom filter's set bits when the TCP connection was initially made from the internal network.

iv. **TODO:** If the TCP packet is going out of the internal network and is a SYN packet, set both the bloom filter arrays at the computed bit positions (reg\_pos\_one and reg\_pos\_two). Else, if the TCP packet is entering the internal network, read both the bloom filter arrays at the computed bit positions and drop the packet if either is not set.

9. A deparser that emits the Ethernet, IPv4 and TCP headers in the right order. 10. A package instantiation supplied with the parser, control, and deparser.

In general, a package also requires instances of checksum verification and recomputation controls. These are not necessary for this tutorial and are replaced with instantiations of empty controls.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  /\* CONSTANTS \*/  const bit<16> TYPE\_IPV4 = 0x800;  const bit<8> TYPE\_TCP = 6;  #define BLOOM\_FILTER\_ENTRIES 4096  #define BLOOM\_FILTER\_BIT\_WIDTH 1  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }  header tcp\_t{  bit<16> srcPort;  bit<16> dstPort;  bit<32> seqNo;  bit<32> ackNo;  bit<4> dataOffset;  bit<4> res;  bit<1> cwr;  bit<1> ece;  bit<1> urg;  bit<1> ack;  bit<1> psh;  bit<1> rst;  bit<1> syn;  bit<1> fin;  bit<16> window;  bit<16> checksum;  bit<16> urgentPtr;  }  struct metadata {  /\* empty \*/  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  tcp\_t tcp;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 select(hdr.ipv4.protocol){  TYPE\_TCP: tcp;  default: accept;  }  }  state tcp {  packet.extract(hdr.tcp);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  register<bit<BLOOM\_FILTER\_BIT\_WIDTH>>(BLOOM\_FILTER\_ENTRIES) bloom\_filter\_1; register<bit<BLOOM\_FILTER\_BIT\_WIDTH>>(BLOOM\_FILTER\_ENTRIES) bloom\_filter\_2; bit<32> reg\_pos\_one; bit<32> reg\_pos\_two;  bit<1> reg\_val\_one; bit<1> reg\_val\_two;  bit<1> direction;  action drop() {  mark\_to\_drop(standard\_metadata);  }  action compute\_hashes(ip4Addr\_t ipAddr1, ip4Addr\_t ipAddr2, bit<16> port1, bit<16> port2){ //Get register position  hash(reg\_pos\_one, HashAlgorithm.crc16, (bit<32>)0, {ipAddr1,  ipAddr2,  port1,  port2,  hdr.ipv4.protocol},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  hash(reg\_pos\_two, HashAlgorithm.crc32, (bit<32>)0, {ipAddr1,  ipAddr2,  port1,  port2,  hdr.ipv4.protocol},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  }  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();  }  action set\_direction(bit<1> dir) {  direction = dir;  }  table check\_ports {  key = {  standard\_metadata.ingress\_port: exact;  standard\_metadata.egress\_spec: exact;  }  actions = {  set\_direction;  NoAction;  }  size = 1024;  default\_action = NoAction();  }  apply {  if (hdr.ipv4.isValid()){  ipv4\_lpm.apply();  if (hdr.tcp.isValid()){  direction = 0; // default  if (check\_ports.apply().hit) {  // test and set the bloom filter  if (direction == 0) {  compute\_hashes(hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.tcp.srcPort, hdr.tcp.dstPort); }  else {  compute\_hashes(hdr.ipv4.dstAddr, hdr.ipv4.srcAddr, hdr.tcp.dstPort, hdr.tcp.srcPort); }  // Packet comes from internal network  if (direction == 0){  // If there is a syn we update the bloom filter and add the entry if (hdr.tcp.syn == 1){  bloom\_filter\_1.write(reg\_pos\_one, 1);  bloom\_filter\_2.write(reg\_pos\_two, 1);  }  }  // Packet comes from outside  else if (direction == 1){  // Read bloom filter cells to check if there are 1's  bloom\_filter\_1.read(reg\_val\_one, reg\_pos\_one);  bloom\_filter\_2.read(reg\_val\_two, reg\_pos\_two);  // only allow flow to pass if both entries are set  if (reg\_val\_one != 1 || reg\_val\_two != 1){  drop(); } }  }  }  }  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  packet.emit(hdr.tcp);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 6)Link monitor.p4

**Implementing Link Monitoring**

**Introduction**

The objective of this exercise is to write a P4 program that enables a host to monitor the utilization of all links in the network. This exercise builds upon the basic IPv4 forwarding exercise so be sure to complete that one before attempting this one. Specifically, we will

modify the basic P4 program to process a source routed probe packet such that it is able to pick up the egress link utilization at each hop and deliver it to a host for monitoring purposes.

## About program

The link\_monitor.p4 file contains a skeleton P4 program with key pieces of logic replaced by TODO comments. Your implementation should follow the structure given in this file--- replace each TODO with logic implementing the missing piece.

Here are a few more details about the design:

**Parser**

● The parser has been extended support parsing of the source routed probe packets. The parser is the most complicated part of the design so spend a bit of time reading over it. Note that it does not contain any TODO comments so there is nothing you need to change here.

● To parse the probe packets, we use the hdr.probe.hop\_cnt to determine how many hops the packet has traversed prior to reaching the switch. If this is the first hop then there will not be any probe\_data in the packet so we skip that state and transition directly to the parse\_probe\_fwd state. In the parse\_probe\_fwd state, we use the hdr.probe.hop\_cnt field to figure out which egress\_spec header field to use to perform forwarding and we save that port value into a metadata field which is subsequently used to perform forwarding.

**Ingress Control**

● The ingress control block looks very similar to the basic exercise. The only difference is that the apply block contains another condition to forward probe packets using the egress\_spec field extracted by the parser. It also increments

the hdr.probe.hop\_cnt field.

**Egress Control**

● This is where the interesting stateful processing occurs. It uses

the byte\_cnt\_reg register to count the number of bytes that have passed through each port since the last probe packet passed through the port.

● It adds a new probe\_data header to the packet and filld out the bos (bottom of stack) field, as well as the swid (switch ID) field.

● TODO: your job is to fill out the rest of the probe packet fields in order to ensure that you can properly measure link utilization.

**Deparser**

● Simply emits all headers in the correct order.

● Note that emitting a header stack will only emit the headers within the stack that are actually marked as valid.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  /\* CONSTANTS \*/  const bit<16> TYPE\_IPV4 = 0x800;  const bit<8> TYPE\_TCP = 6;  #define BLOOM\_FILTER\_ENTRIES 4096  #define BLOOM\_FILTER\_BIT\_WIDTH 1  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }  header tcp\_t{  bit<16> srcPort;  bit<16> dstPort;  bit<32> seqNo;  bit<32> ackNo;  bit<4> dataOffset;  bit<4> res;  bit<1> cwr;  bit<1> ece;  bit<1> urg;  bit<1> ack;  bit<1> psh;  bit<1> rst;  bit<1> syn;  bit<1> fin;  bit<16> window;  bit<16> checksum;  bit<16> urgentPtr;  }  struct metadata {  /\* empty \*/  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  tcp\_t tcp;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 select(hdr.ipv4.protocol){  TYPE\_TCP: tcp;  default: accept;  }  }  state tcp {  packet.extract(hdr.tcp);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  register<bit<BLOOM\_FILTER\_BIT\_WIDTH>>(BLOOM\_FILTER\_ENTRIES) bloom\_filter\_1; register<bit<BLOOM\_FILTER\_BIT\_WIDTH>>(BLOOM\_FILTER\_ENTRIES) bloom\_filter\_2; bit<32> reg\_pos\_one; bit<32> reg\_pos\_two;  bit<1> reg\_val\_one; bit<1> reg\_val\_two;  bit<1> direction;  action drop() {  mark\_to\_drop(standard\_metadata);  }  action compute\_hashes(ip4Addr\_t ipAddr1, ip4Addr\_t ipAddr2, bit<16> port1, bit<16> port2){ //Get register position  hash(reg\_pos\_one, HashAlgorithm.crc16, (bit<32>)0, {ipAddr1,  ipAddr2,  port1,  port2,  hdr.ipv4.protocol},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  hash(reg\_pos\_two, HashAlgorithm.crc32, (bit<32>)0, {ipAddr1,  ipAddr2,  port1,  port2,  hdr.ipv4.protocol},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  }  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();  }  action set\_direction(bit<1> dir) {  direction = dir;  }  table check\_ports {  key = {  standard\_metadata.ingress\_port: exact;  standard\_metadata.egress\_spec: exact;  }  actions = {  set\_direction;  NoAction;  }  size = 1024;  default\_action = NoAction();  }  apply {  if (hdr.ipv4.isValid()){  ipv4\_lpm.apply();  if (hdr.tcp.isValid()){  direction = 0; // default  if (check\_ports.apply().hit) {  // test and set the bloom filter  if (direction == 0) {  compute\_hashes(hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.tcp.srcPort, hdr.tcp.dstPort); }  else {  compute\_hashes(hdr.ipv4.dstAddr, hdr.ipv4.srcAddr, hdr.tcp.dstPort, hdr.tcp.srcPort); }  // Packet comes from internal network  if (direction == 0){  // If there is a syn we update the bloom filter and add the entry if (hdr.tcp.syn == 1){  bloom\_filter\_1.write(reg\_pos\_one, 1);  bloom\_filter\_2.write(reg\_pos\_two, 1);  } }  // Packet comes from outside  else if (direction == 1){  // Read bloom filter cells to check if there are 1's  bloom\_filter\_1.read(reg\_val\_one, reg\_pos\_one);  bloom\_filter\_2.read(reg\_val\_two, reg\_pos\_two);  // only allow flow to pass if both entries are set  if (reg\_val\_one != 1 || reg\_val\_two != 1){  drop();  } } }  } }  } }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  packet.emit(hdr.tcp);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main |

# 7)Load balancing.p4

In this exercise, you will implement a form of load balancing based on a simple version of Equal-Cost Multipath Forwarding. The switch you will implement will use two tables to forward packets to one of two destination hosts at random. The first table will use a hash function (applied to a 5-tuple consisting of the source and destination IP addresses, IP protocol, and source and destination TCP ports) to select one of two hosts. The second table will use the computed hash value to forward the packet to the selected host.

## About program

The load\_balance.p4 file contains a skeleton P4 program with key pieces of logic replaced by TODO comments. These should guide your implementation---replace each TODO with logic implementing the missing piece.

A complete load\_balance.p4 will contain the following components:

1. Header type definitions for Ethernet (ethernet\_t) and IPv4 (ipv4\_t).

2. Parsers for Ethernet and IPv4 that populate ethernet\_t and ipv4\_t fields. 3. An action to drop a packet, using mark\_to\_drop().

4. **TODO:** An action (called set\_ecmp\_select), which will:

i. Hashes the 5-tuple specified above using the hash extern

ii. Stores the result in the meta.ecmp\_select field

5. **TODO:** A control that:

i. Applies the ecmp\_group table.

ii. Applies the ecmp\_nhop table.

6. A deparser that selects the order in which fields inserted into the outgoing packet.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }  header tcp\_t {  bit<16> srcPort;  bit<16> dstPort;  bit<32> seqNo;  bit<32> ackNo;  bit<4> dataOffset;  bit<3> res;  bit<3> ecn;  bit<6> ctrl;  bit<16> window;  bit<16> checksum;  bit<16> urgentPtr;  }  struct metadata {  bit<14> ecmp\_select;  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  tcp\_t tcp;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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) {  0x800: parse\_ipv4;  default: accept;  }  }  state parse\_ipv4 {  packet.extract(hdr.ipv4);  transition select(hdr.ipv4.protocol) {  6: parse\_tcp;  default: accept;  }  }  state parse\_tcp {  packet.extract(hdr.tcp);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action set\_ecmp\_select(bit<16> ecmp\_base, bit<32> ecmp\_count) {  hash(meta.ecmp\_select,  HashAlgorithm.crc16,  ecmp\_base,  { hdr.ipv4.srcAddr,  hdr.ipv4.dstAddr,  hdr.ipv4.protocol,  hdr.tcp.srcPort,  hdr.tcp.dstPort },  ecmp\_count);  }  action set\_nhop(bit<48> nhop\_dmac, bit<32> nhop\_ipv4, bit<9> port) {  hdr.ethernet.dstAddr = nhop\_dmac;  hdr.ipv4.dstAddr = nhop\_ipv4;  standard\_metadata.egress\_spec = port;  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;  }  table ecmp\_group {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  drop;  set\_ecmp\_select;  }  size = 1024;  }  table ecmp\_nhop {  key = {  meta.ecmp\_select: exact;  }  actions = {  drop;  set\_nhop;  }  size = 2;  }  apply {  if (hdr.ipv4.isValid() && hdr.ipv4.ttl > 0) {  ecmp\_group.apply();  ecmp\_nhop.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) {    action rewrite\_mac(bit<48> smac) {  hdr.ethernet.srcAddr = smac;  }  action drop() {  mark\_to\_drop(standard\_metadata);  }  table send\_frame {  key = {  standard\_metadata.egress\_port: exact;  }  actions = {  rewrite\_mac;  drop;  }  size = 256;  }  apply {  send\_frame.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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  packet.emit(hdr.tcp);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 8)Multi-Hop Route Inspection.p4

**Implementing MRI**

**Introduction**

The objective of this tutorial is to extend basic L3 forwarding with a scaled-down version of In-Band Network Telemetry (INT), which we call Multi-Hop Route Inspection (MRI).

MRI allows users to track the path and the length of queues that every packet travels through. To support this functionality, you will need to write a P4 program that appends an ID and queue length to the header stack of every packet. At the destination, the sequence of switch IDs correspond to the path, and each ID is followed by the queue length of the port at switch.

## About program

A complete mri.p4 will contain the following components:

1. Header type definitions for Ethernet (ethernet\_t), IPv4 (ipv4\_t), IP Options (ipv4\_option\_t), MRI (mri\_t), and Switch (switch\_t).

2. Parsers for Ethernet, IPv4, IP Options, MRI, and Switch that will

populate ethernet\_t, ipv4\_t, ipv4\_option\_t, mri\_t, and switch\_t.

3. An action to drop a packet, using mark\_to\_drop().

4. An action (called ipv4\_forward), which will:

i. Set the egress port for the next hop.

ii. Update the ethernet destination address with the address of the next hop. iii. Update the ethernet source address with the address of the switch. iv. Decrement the TTL.

5. An ingress control that:

i. Defines a table that will read an IPv4 destination address, and invoke either drop or ipv4\_forward.

ii. An apply block that applies the table.

6. At egress, an action (called add\_swtrace) that will add the switch ID and queue depth.

7. An egress control that applies a table (swtrace) to store the switch ID and queue depth, and calls add\_swtrace.

8. A deparser that selects the order in which fields inserted into the outgoing packet. 9. A package instantiation supplied with the parser, control, checksum verification and recomputation and deparser.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<8> UDP\_PROTOCOL = 0x11;  const bit<16> TYPE\_IPV4 = 0x800;  const bit<5> IPV4\_OPTION\_MRI = 31;  #define MAX\_HOPS 9  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* typedef bit<9> egressSpec\_t;  typedef bit<48> macAddr\_t;  typedef bit<32> ip4Addr\_t;  typedef bit<32> switchID\_t;  typedef bit<32> qdepth\_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;  }  header ipv4\_option\_t {  bit<1> copyFlag;  bit<2> optClass;  bit<5> option;  bit<8> optionLength;  }  header mri\_t {  bit<16> count;  }  header switch\_t {  switchID\_t swid;  qdepth\_t qdepth;  }  struct ingress\_metadata\_t {  bit<16> count;  }  struct parser\_metadata\_t {  bit<16> remaining;  }  struct metadata {  ingress\_metadata\_t ingress\_metadata;  parser\_metadata\_t parser\_metadata;  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  ipv4\_option\_t ipv4\_option;  mri\_t mri;  switch\_t[MAX\_HOPS] swtraces;  }  error { IPHeaderTooShort }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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);  verify(hdr.ipv4.ihl >= 5, error.IPHeaderTooShort);  transition select(hdr.ipv4.ihl) {  5 : accept;  default : parse\_ipv4\_option;  }  }  state parse\_ipv4\_option {  packet.extract(hdr.ipv4\_option);  transition select(hdr.ipv4\_option.option) {  IPV4\_OPTION\_MRI: parse\_mri;  default: accept;  }  }  state parse\_mri {  packet.extract(hdr.mri);  meta.parser\_metadata.remaining = hdr.mri.count;  transition select(meta.parser\_metadata.remaining) {  0 : accept;  default: parse\_swtrace;  }  }  state parse\_swtrace {  packet.extract(hdr.swtraces.next);  meta.parser\_metadata.remaining = meta.parser\_metadata.remaining - 1; transition select(meta.parser\_metadata.remaining) {  0 : accept;  default: parse\_swtrace;  }  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  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 = NoAction();  }  apply {  if (hdr.ipv4.isValid()) {  ipv4\_lpm.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) {  action add\_swtrace(switchID\_t swid) {  hdr.mri.count = hdr.mri.count + 1;  hdr.swtraces.push\_front(1);  // According to the P4\_16 spec, pushed elements are invalid, so we need // to call setValid(). Older bmv2 versions would mark the new header(s) // valid automatically (P4\_14 behavior), but starting with version 1.11,  // bmv2 conforms with the P4\_16 spec.  hdr.swtraces[0].setValid();  hdr.swtraces[0].swid = swid;  hdr.swtraces[0].qdepth = (qdepth\_t)standard\_metadata.deq\_qdepth;  hdr.ipv4.ihl = hdr.ipv4.ihl + 2;  hdr.ipv4\_option.optionLength = hdr.ipv4\_option.optionLength + 8;  hdr.ipv4.totalLen = hdr.ipv4.totalLen + 8;  }  table swtrace {  actions = {  add\_swtrace;  NoAction;  }  default\_action = NoAction();  }  apply {  if (hdr.mri.isValid()) {  swtrace.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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  packet.emit(hdr.ipv4\_option);  packet.emit(hdr.mri);  packet.emit(hdr.swtraces);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 9)Multicast.p4

**Implementing Multicast**

**Introduction**

The objective of this exercise is to write a P4 program that multicasts packets to a group of ports.

Upon receiving an Ethernet packet, the switch looks up the output port based on the destination MAC address. If it is a miss, the switch broadcast packets on ports belonging to a multicast group (if ingress port appears in the group, the packet will be dropped in the egress pipeline).

Your switch will have a single table, which the control plane will populate with static rules. Each rule will map an Ethernet MAC address to the output port. We have already defined the control plane rules, so you only need to implement the data plane logic of your P4 program.

## About program

The multicast.p4 file contains a skeleton P4 program with key pieces of logic replaced by TODO comments. Your implementation should follow the structure given in this file--- replace each TODO with logic implementing the missing piece.

A complete multicast.p4 will contain the following components:

1. Header type definitions for Ethernet (ethernet\_t)

2. An action to drop a packet, using mark\_to\_drop().

3. **TODO:** An action (called multicast) that sends multiple copies of packets to a group of output ports.

4. **TODO:** Add the multicast action to the list of available actions

5. **TODO:** Set multicast as default action for table mac\_lookup

6. **TODO:** Add port 4 to the multicast group in file sig-topo/s1-runtime.json

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }  struct metadata {  /\* empty \*/  }  struct headers {  ethernet\_t ethernet;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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) {  default : accept;  }  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action multicast() {  standard\_metadata.mcast\_grp = 1;  }  action mac\_forward(egressSpec\_t port) {  standard\_metadata.egress\_spec = port;  }  table mac\_lookup {  key = {  hdr.ethernet.dstAddr : exact;  }  actions = {  multicast;  mac\_forward;  drop;  }  size = 1024;  default\_action = multicast;  }  apply {  if (hdr.ethernet.isValid())  mac\_lookup.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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  apply {  // Prune multicast packet to ingress port to preventing loop  if (standard\_metadata.egress\_port == standard\_metadata.ingress\_port) drop();  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  packet.emit(hdr.ethernet);  }  }\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 10)Quality of Service.p4

**Implementing QOS**

**Introduction**

The objective of this tutorial is to extend basic L3 forwarding with an implementation of Quality of Service (QOS) using Differentiated Services.

Diffserv is simple and scalable. It classifies and manages network traffic and provides QOS on modern IP networks.

As before, we have already defined the control plane rules for routing, so you only need to implement the data plane logic of your P4 program.

## About program

The qos.p4 file contains a skeleton P4 program with key pieces of logic replaced by TODO comments. These should guide your implementation---replace each TODO with logic implementing the missing piece.

First we have to change the ipv4\_t header by splitting the TOS field into DiffServ and ECN fields. Remember to update the checksum block accordingly. Then, in the egress control block we must compare the protocol in IP header with IP protocols. Based on the traffic classes and priority, the diffserv flag will be set.

A complete qos.p4 will contain the following components:

1. Header type definitions for Ethernet (ethernet\_t) and IPv4 (ipv4\_t). 2. Parsers for Ethernet, IPv4,

3. An action to drop a packet, using mark\_to\_drop().

4. An action (called ipv4\_forward), which will:

i. Set the egress port for the next hop.

ii. Update the ethernet destination address with the address of the next hop.

iii. Update the ethernet source address with the address of the switch. iv. Decrement the TTL.

5. An ingress control block that checks the protocols and sets the ipv4.diffserv. 6. A deparser that selects the order in which headers are inserted into the outgoing packet.

7. A package instantiation supplied with the parser, control, checksum verification and recomputation and deparser.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  /\* IP protocols \*/  const bit<8> IP\_PROTOCOLS\_ICMP = 1;  const bit<8> IP\_PROTOCOLS\_IGMP = 2;  const bit<8> IP\_PROTOCOLS\_IPV4 = 4;  const bit<8> IP\_PROTOCOLS\_TCP = 6;  const bit<8> IP\_PROTOCOLS\_UDP = 17;  const bit<8> IP\_PROTOCOLS\_IPV6 = 41;  const bit<8> IP\_PROTOCOLS\_GRE = 47;  const bit<8> IP\_PROTOCOLS\_IPSEC\_ESP = 50;  const bit<8> IP\_PROTOCOLS\_IPSEC\_AH = 51;  const bit<8> IP\_PROTOCOLS\_ICMPV6 = 58;  const bit<8> IP\_PROTOCOLS\_EIGRP = 88;  const bit<8> IP\_PROTOCOLS\_OSPF = 89;  const bit<8> IP\_PROTOCOLS\_PIM = 103;  const bit<8> IP\_PROTOCOLS\_VRRP = 112;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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<6> diffserv;  bit<2> 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;  }  struct metadata {  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  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;  }  /\* Default Forwarding \*/  action default\_forwarding() {  hdr.ipv4.diffserv = 0;  }  /\* Expedited Forwarding \*/  action expedited\_forwarding() {  hdr.ipv4.diffserv = 46;  }  /\* Voice Admit \*/  action voice\_admit() {  hdr.ipv4.diffserv = 44;  }  /\* Assured Forwarding \*/  /\* Class 1 Low drop probability \*/  action af\_11() {  hdr.ipv4.diffserv = 10;  }  /\* Class 1 Med drop probability \*/  action af\_12() {  hdr.ipv4.diffserv = 12;  }  /\* Class 1 High drop probability \*/  action af\_13() {  hdr.ipv4.diffserv = 14;  }  /\* Class 2 Low drop probability \*/  action af\_21() {  hdr.ipv4.diffserv = 18;  }  /\* Class 2 Med drop probability \*/  action af\_22() {  hdr.ipv4.diffserv = 20;  }  /\* Class 2 High drop probability \*/  action af\_23() {  hdr.ipv4.diffserv = 22;  }  /\* Class 3 Low drop probability \*/  action af\_31() {  hdr.ipv4.diffserv = 26;  }  /\* Class 3 Med drop probability \*/  action af\_32() {  hdr.ipv4.diffserv = 28;  }  /\* Class 3 High drop probability \*/  action af\_33() {  hdr.ipv4.diffserv = 30;  }  /\* Class 4 Low drop probability \*/  action af\_41() {  hdr.ipv4.diffserv = 34;  }  /\* Class 4 Med drop probability \*/  action af\_42() {  hdr.ipv4.diffserv = 36;  }  /\* Class 4 High drop probability \*/  action af\_43() {  hdr.ipv4.diffserv = 38;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  ipv4\_forward;  drop;  NoAction;  }  size = 1024;  default\_action = NoAction();  }  apply {  if (hdr.ipv4.isValid()) {  if (hdr.ipv4.protocol == IP\_PROTOCOLS\_UDP) {  expedited\_forwarding();  }  else if (hdr.ipv4.protocol == IP\_PROTOCOLS\_TCP) {  voice\_admit();  }  ipv4\_lpm.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 {  update\_checksum(  hdr.ipv4.isValid(),  { hdr.ipv4.version,  hdr.ipv4.ihl,  hdr.ipv4.diffserv,  hdr.ipv4.ecn,  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 11)Reflector.p4

**Packet Reflector**

**Introduction**

The packet reflector will be our very first P4 exercise. The main objective of this exercise is to show you how to create simple topologies with hosts and p4 switches and how to add links between them. Then you will implement a very simple p4 program that makes switches bouncing back packets to the interface the packets came from.

## About program

Follow these instructions to create a mininet network and run reflector.p4:

1. To create the topology described in p4app.json, you just have to call p4run, which by default will check if the file p4app.json exists in the path:

sudo p4run

This will call a python script that parses the configuration file, creates a virtual network of hosts and p4 switches using mininet, compile the p4 program and load it in the switch

2. At this point you will have a small topology that consists of a host h1 and a p4 switch s1. You can get a terminal in h1 by either typing xterm h1 in the CLI, or by using the mx command that comes already installed in the VM:

mx h1

3. From the h1 terminal you can run send\_receive.py, a small python script that sends packets to the switch and prints if the packets get reflected. Since the switch with

the initial code in reflector.p4 is not doing anything, all the packets it receives will get dropped and nothing will get reflected back to h1.

4. Close all the host-terminals and type quit to leave the mininet CLI and clean the network.

mininet> quit

Your program should do the following:

1. Parse the ethernet header and make a transition to MyIngress. Note that the definition of the ethernet header and the headers struct is already defined for you.

2. Swap the packet's ethernet addresses. You can use an action or simply write the code directly in the control apply.

Hint: you can define and use local variables to swap the addresses: bit<48> tmpAddr;

3. Use the *ingress\_port* as *egress\_port*. The value of the ingress\_port will be stored in the packet metadata, in the variable standard\_metadata.ingress\_port.

To set a packet's output port, you need to set standard\_metadata.egress\_spec metadata

field. For more information about the standard metadata fields read: simple switch documentation.

4. Deparse the ethernet header.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* typedef bit<48> macAddr\_t;  header ethernet\_t {  macAddr\_t dstAddr;  macAddr\_t srcAddr;  bit<16> etherType;  }  struct metadata {  /\* empty \*/  }  struct headers {  ethernet\_t ethernet;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* parser MyParser(packet\_in packet,  out headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {  state start{  packet.extract(hdr.ethernet);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  action swap\_mac(){  macAddr\_t tmp;  tmp = hdr.ethernet.srcAddr;  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  hdr.ethernet.dstAddr = tmp;  }  apply {  // Swap MAC addresses.  swap\_mac();  //Set Output port == Input port  standard\_metadata.egress\_spec = standard\_metadata.ingress\_port;  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  // parsed headers have to be added again into the packet  packet.emit(hdr.ethernet);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 12)Repeater.p4

**Introduction**

In the second introductory exercise we will use our first table and conditional statements in a control block. In this exercise you will make a two-port switch act as a packet repeater, in other words, when a packet enters port 1 it has to be leave from port 2 and vice versa.

Once you have the repeater.p4 program finished you can test its behaviour:

1. Start the topology (this will also compile and load the program).

sudo p4run

2. Get a terminal in h1 and h2 using mx:

mx h1

mx h2 #in different terminal windows

Or directly from the mininet prompt using xterm:

> mininet xterm h1 h2

3. Run receive.py app in h2.

4. Run send.py in h1:

python send.py 10.0.0.2 "Hello H2"

5. Since the switch will always forward traffic from h1 to h2 and vice versa, we can test the repeater with other applications such as: ping, iperf, etc. The mininet CLI provides some helpers that make very easy such kind of tests:

> mininet h1 ping h2

> mininet iperf h1 h2

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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) {  state start{  transition accept;  } }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  action forward(bit<9> egress\_port){  standard\_metadata.egress\_spec = egress\_port;  }  table repeater {  key = {  standard\_metadata.ingress\_port: exact;  }  actions = {  forward;  NoAction; }  size = 2;  default\_action = NoAction;  }  apply {  repeater.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 {  /\* Deparser not needed \*/ }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 13)Verifying checksum.p4

**Introduction**

In this example we show how to verify IPv4 checksums and drop packets accordingly.

How to run

Start the topology:

sudo p4run

Start a receiver at h2:

mx h2

python receive.py

Send valid and invalid packets to h2 from h1:

mx h1

python send.py 10.0.1.2 valid/invalid

You will observe that only packets with a valid checksum get forwarded.

|  |
| --- |
| Code: #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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){  TYPE\_IPV4: ipv4;  default: accept;  }  }  state ipv4 {  packet.extract(hdr.ipv4);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action ipv4\_forward(macAddr\_t dstAddr, egressSpec\_t port) {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl -1;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  ipv4\_forward;  drop;  NoAction;  }  size = 1024;  default\_action = NoAction();  }  apply {  //only if IPV4 the rule is applied. Therefore other packets will not be forwarded. if (hdr.ipv4.isValid() && standard\_metadata.checksum\_error == 0){  ipv4\_lpm.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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* control MyDeparser(packet\_out packet, in headers hdr) {  apply {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 14)Meter

# (Direct meter.p4 & Indirect meter.p4)

**Introduction**

This program illustrates as simply as possible how to use meters in P4 with bmv2. bmv2 uses two-rate three-color meters as described here.

For each incoming packet the m\_read table is applied and the appropriate meter (based on the packet's source MAC address) is executed. Based on the observed traffic rate for this sender and the meter's configuration, executing the meter will yield one of 3 values: 0 (*GREEN*), 1 (*YELLOW*) or 2 (*RED*). This value will be copied to metadata field meta.meter\_tag. Note that if no meter was associated to the sender's MAC address, the table will be a no-op. This table also redirects all packets - with a known source MAC address- to port 2 of the switch.

After that, the packet will go through a second table, m\_filter, which can either be a no-op or drop the packet based on how the packet was tagged by the meter. If you take a look at the runtime commands and the default action in the p4 programm we wrote for this example, you will see that we configure the table to drop all the packets for which the color is not *GREEN* (i.e. all packets for which meta.meter\_tag is not 0).

The commands.txt file also gives you the meter configuration. In this case, the first rate is 0.5 packets per second, with a burst size of 1, and the second rate is 10 packets per second, with a burst size of 1 also. Feel free to play with the numbers, but these play nicely with the demonstration below.

**How to run**

There are two examples, one using a direct and one using an indirect meter. You can start them using:

$ sudo ./run.sh --config p4app\_direct.json

$ sudo ./run.sh --config p4app\_indirect.json

In the mininet CLI, you can start the demo script which periodically sends packets from the host 1 interface and listens for packets on the host 2 interface. The script takes the time interface (in seconds) as argument, e.g.:

mininet> sh ./send\_and\_receive.py 1

(Works for both direct and indirect meters)

|  |
| --- |
| Code:Direct meter.p4 #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 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;  }  struct metadata {  bit<32> meter\_tag;  }  struct headers {  ethernet\_t ethernet;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* parser MyParser(packet\_in packet,  out headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {  state start {  packet.extract(hdr.ethernet);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  direct\_meter<bit<32>>(MeterType.packets) my\_meter;  action drop() {  mark\_to\_drop(standard\_metadata);  }  action m\_action() {  my\_meter.read(meta.meter\_tag);  }  table m\_read {  key = {  hdr.ethernet.srcAddr: exact;  }  actions = {  m\_action;  NoAction;  }  default\_action = NoAction;  meters = my\_meter;  size = 16384;  }  table m\_filter {  key = {  meta.meter\_tag: exact;  }  actions = {  drop;  NoAction; }  default\_action = drop;  size = 16;  }  apply {  // Same egress port for all packets in this example  standard\_metadata.egress\_spec = 2;  // Check meter  m\_read.apply();  // Filter based on meter status  m\_filter.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 ER\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; Indirect meter.p4 #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }  struct metadata {  bit<32> meter\_tag;  }struct headers {  ethernet\_t ethernet;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* parser MyParser(packet\_in packet,  out headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {  state start {  packet.extract(hdr.ethernet);  transition accept;  }}  \*\*\*\*\*\*\*\* 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) {  meter(32w16384, MeterType.packets) my\_meter;  action drop() {  mark\_to\_drop(standard\_metadata);  }  action m\_action(bit<32> meter\_index) {  my\_meter.execute\_meter<bit<32>>(meter\_index, meta.meter\_tag);  }  table m\_read {  key = {  hdr.ethernet.srcAddr: exact;  }  actions = {  m\_action;  NoAction;  }  default\_action = NoAction;  size = 16384;  }  table m\_filter {  key = {  meta.meter\_tag: exact;  }  actions = {  drop;  NoAction;  }  default\_action = drop;  size = 16;  } apply {  // Same egress port for all packets in this example  standard\_metadata.egress\_spec = 2;  // Check meter  m\_read.apply();  // Filter based on meter status  m\_filter.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 {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 15)Counters

# (Direct counter.p4 & Indirect counter.p4)

Simple example of direct and indirect counters that count the packets and bytes arriving at each ingress port.

**How to run**

Run the topology, by starting either the direct or indirect examples:

sudo p4run --config p4app\_direct.json

sudo p4run --config p4app\_indirect.json

Testing the counters

Use e.g. ping to send packets. The switch will count all packets and then drop them (because forwarding is not implemented).

Access counter values (for direct and indirect counters):

$ simple\_switch\_CLI --thrift-port 9090

$ RuntimeCmd: counter\_read MyIngress.port\_counter 0

(where the 1 is the entry handle for a given table entry. For example the first entry added in a table will have a handle of 0)

Using a simple control plane to read the counters

The following controller program will establish a connection with the switch and through a thrift API it will read all the counter values.

$ python read\_counters.py [direct/indirect]

|  |
| --- |
| Code:(Direct counter.p4) #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }  struct metadata {  bit<32> meter\_tag;  }  struct headers {  ethernet\_t ethernet;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* parser MyParser(packet\_in packet,  out headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {  state start {  packet.extract(hdr.ethernet);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  direct\_counter(CounterType.packets\_and\_bytes) direct\_port\_counter;  table count\_table {  key = {  standard\_metadata.ingress\_port: exact;  }  actions = {  NoAction;  }  default\_action = NoAction;  counters = direct\_port\_counter;  size = 512;  }  apply {  count\_table.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 {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; (Indirect counter.p4) #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }  struct metadata {  bit<32> meter\_tag;  }  struct headers {  ethernet\_t ethernet;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* parser MyParser(packet\_in packet,  out headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {  state start {  packet.extract(hdr.ethernet);  transition accept; }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  counter(512, CounterType.packets\_and\_bytes) port\_counter;  action drop() {  mark\_to\_drop(standard\_metadata);  }  apply {  port\_counter.count((bit<32>)standard\_metadata.ingress\_port); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 16)Source routing.p4

**Introduction**

The objective of this exercise is to implement source routing. With source routing, the source host guides each switch in the network to send the packet to a specific port. The host puts a stack of output ports in the packet. In this example, we just put the stack after Ethernet header and select a special etherType to indicate that. Each switch pops an item from the stack and forwards the packet according to the specified port number.

Your switch must parse the source routing stack. Each item has a bos (bottom of stack) bit and a port number. The bos bit is 1 only for the last entry of stack. Then at ingress, it should pop an entry from the stack and set the egress port accordingly. The last hop may also revert back the etherType to TYPE\_IPV4.

## About program

The source\_routing.p4 file contains a skeleton P4 program with key pieces of logic replaced by TODO comments. These should guide your implementation---replace each TODO with logic implementing the missing piece.

A complete source\_routing.p4 will contain the following components:

1. Header type definitions for Ethernet (ethernet\_t) and IPv4 (ipv4\_t) and

Source Route (srcRoute\_t).

2. **TODO:** Parsers for Ethernet and Source Route that

populate ethernet and srcRoutes fields.

3. An action to drop a packet, using mark\_to\_drop().

4. **TODO:** An action (called srcRoute\_nhop), which will:

i. Set the egress port for the next hop.

ii. remove the first entry of srcRoutes

5. A control with an apply block that:

i. checks the existence of source routes.

ii. **TODO:** if statement to change etherent.etherType if it is the last hop iii. **TODO:** call srcRoute\_nhop action

6. A deparser that selects the order in which fields inserted into the outgoing packet. 7. A package instantiation supplied with the parser, control, and deparser.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  const bit<16> TYPE\_SRCROUTING = 0x1234;  #define MAX\_HOPS 9  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 srcRoute\_t {  bit<1> bos;  bit<15> port;  }  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;  srcRoute\_t[MAX\_HOPS] srcRoutes;  ipv4\_t ipv4;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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\_SRCROUTING: parse\_srcRouting;  default: accept;  }  }  state parse\_srcRouting {  packet.extract(hdr.srcRoutes.next);  transition select(hdr.srcRoutes.last.bos) {  1: parse\_ipv4;  default: parse\_srcRouting;  }  }  state parse\_ipv4 {  packet.extract(hdr.ipv4);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action srcRoute\_nhop() {  standard\_metadata.egress\_spec = (bit<9>)hdr.srcRoutes[0].port;  hdr.srcRoutes.pop\_front(1);  }  action srcRoute\_finish() {  hdr.ethernet.etherType = TYPE\_IPV4;  } action update\_ttl(){  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;  }  apply {  if (hdr.srcRoutes[0].isValid()){  if (hdr.srcRoutes[0].bos == 1){  srcRoute\_finish();  }  srcRoute\_nhop();  if (hdr.ipv4.isValid()){  update\_ttl();  }  }else{  drop();  }  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  packet.emit(hdr.ethernet);  packet.emit(hdr.srcRoutes);  packet.emit(hdr.ipv4);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 17)IP Forwarding.P4

**Introduction**

The objective of this exercise is to write a P4 program that implements basic forwarding. To keep things simple, we will just implement forwarding for IPv4.

With IPv4 forwarding, the switch must perform the following actions for every packet: (i) update the source and destination MAC addresses, (ii) decrement the time-to-live (TTL) in the IP header, and (iii) forward the packet out the appropriate port.

Your switch will have a single table, which the control plane will populate with static rules. Each rule will map an IP address to the MAC address and output port for the next hop. We have already defined the control plane rules, so you only need to implement the data plane logic of your P4 program.

|  |
| --- |
| Code: #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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;  }\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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){  TYPE\_IPV4: ipv4;  default: accept;  }  }  state ipv4 {  packet.extract(hdr.ipv4);  transition accept;  }}  \*\*\*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  } action ipv4\_forward(macAddr\_t dstAddr, egressSpec\_t port) {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl -1;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  ipv4\_forward;  drop;  NoAction;  }  size = 1024;  default\_action = NoAction();  }  apply {  //only if IPV4 the rule is applied. Therefore other packets will not be forwarded. if (hdr.ipv4.isValid()){  ipv4\_lpm.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 {  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);  }  } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* control MyDeparser(packet\_out packet, in headers hdr) {  apply {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 18)Digest Messages.p4

**Introduction**

Similarly to the [copy\_to\_cpu](https://github.com/nsg-ethz/p4-learning/blob/master/examples/copy_to_cpu) example in this example we show how to use the bmv2 digest extern to send information to the control plane using an out of band channel. In the digest we send a tuple with: (random\_number, src\_ip, dst\_ip).

Run the topology:

sudo p4run

Run the very small controller code that receives packets from the switch:

sudo python get\_digest.py

Any packet will trigger a digest so we can simply run a ping between h1 and h2, from the CLI

mininet> h1 ping h2 -c1

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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>tos;  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 learn\_t {  bit<8> digest;  bit<32> srcIP;  bit<32> dstIP;  }  struct metadata {  learn\_t learn;  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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){  TYPE\_IPV4: ipv4;  default: accept;  }  } state ipv4 { packet.extract(hdr.ipv4);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\* 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) {    action drop() {  mark\_to\_drop(standard\_metadata);  }  action ipv4\_forward(macAddr\_t dstAddr, egressSpec\_t port)  {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table  hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl -1;  random(meta.learn.digest,(bit<8>) 0, (bit<8>) 255);  meta.learn.srcIP = hdr.ipv4.srcAddr;  meta.learn.dstIP = hdr.ipv4.dstAddr;  //digest packet  digest(1, meta.learn);  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  ipv4\_forward;  drop;  NoAction;  }  size = 1024;  default\_action = NoAction();  }  apply {  //only if IPV4 the rule is applied. Therefore other packets will not be forwarded.  if (hdr.ipv4.isValid()){  ipv4\_lpm.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 {  update\_checksum(  hdr.ipv4.isValid(),  { hdr.ipv4.version,  hdr.ipv4.ihl,  hdr.ipv4.tos,  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 19)Heavy Hitter

**Introduction**

Uses a counting bloom filter to block flows for which the switch has observed more than a certain number of packets (default 1000).

**How to run**

Run the topology:

sudo p4run

Run the receiver and the sending scripts in h2 and h1 respectively:

mx h2

python receive.py 5000

Send 1500 packets using the same 5-tuple. Only the first 1000 will be received by h2.

mx h1

python send.py 10.0.2.2 5000 1500

|  |
| --- |
| Code: #include <core.p4>  #include <v1model.p4>  /\* CONSTANTS \*/  const bit<16> TYPE\_IPV4 = 0x800;  const bit<8> TYPE\_TCP = 6;  #define BLOOM\_FILTER\_ENTRIES 4096  #define BLOOM\_FILTER\_BIT\_WIDTH 32  #define PACKET\_THRESHOLD 1000  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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;  }  header tcp\_t{  bit<16> srcPort;  bit<16> dstPort;  bit<32> seqNo;  bit<32> ackNo;  bit<4> dataOffset;  bit<4> res;  bit<1> cwr;  bit<1> ece;  bit<1> urg;  bit<1> ack;  bit<1> psh;  bit<1> rst;  bit<1> syn;  bit<1> fin;  bit<16> window;  bit<16> checksum;  bit<16> urgentPtr;  }  struct metadata {  /\* empty \*/  bit<32> output\_hash\_one;  bit<32> output\_hash\_two;  bit<32> counter\_one;  bit<32> counter\_two;  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  tcp\_t tcp;  }\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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){  TYPE\_IPV4: ipv4;  default: accept; }  }  state ipv4 {  packet.extract(hdr.ipv4);  transition select(hdr.ipv4.protocol){  TYPE\_TCP: tcp;  default: accept; }  }  state tcp {  packet.extract(hdr.tcp);  transition accept; }  }\*\*\*\*\*\*\*\*\*\* 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) {  register<bit<BLOOM\_FILTER\_BIT\_WIDTH>>(BLOOM\_FILTER\_ENTRIES) bloom\_filter;  action drop() {  mark\_to\_drop(standard\_metadata);  }  action \_update\_bloom\_filter(){  //Get register position  hash(meta.output\_hash\_one, HashAlgorithm.crc16, (bit<16>)0, {hdr.ipv4.srcAddr,  hdr.ipv4.dstAddr,  hdr.tcp.srcPort,  hdr.tcp.dstPort,  hdr.ipv4.protocol},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  hash(meta.output\_hash\_two, HashAlgorithm.crc32, (bit<16>)0, {hdr.ipv4.srcAddr,  hdr.ipv4.dstAddr,  hdr.tcp.srcPort,  hdr.tcp.dstPort,  hdr.ipv4.protocol},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  //Read counters  bloom\_filter.read(meta.counter\_one, meta.output\_hash\_one);  bloom\_filter.read(meta.counter\_two, meta.output\_hash\_two);  meta.counter\_one = meta.counter\_one + 1;  meta.counter\_two = meta.counter\_two + 1;  //write counters  bloom\_filter.write(meta.output\_hash\_one, meta.counter\_one);  bloom\_filter.write(meta.output\_hash\_two, meta.counter\_two);  }  action ipv4\_forward(macAddr\_t dstAddr, egressSpec\_t port) {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table  hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl -1;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  ipv4\_forward;  drop;  NoAction; }  size = 1024;  default\_action = NoAction(); }  table update\_bloom\_filter{  actions = {  \_update\_bloom\_filter;  }  size = 1;  default\_action = \_update\_bloom\_filter();  }  table drop\_table{  actions = {  drop; }  size =1;  default\_action = drop();  }  apply {  if (hdr.ipv4.isValid()){  if (hdr.tcp.isValid()){  update\_bloom\_filter.apply();  //only if IPV4 the rule is applied. Therefore other packets will not be forwarded.  if ( (meta.counter\_one > PACKET\_THRESHOLD && meta.counter\_two > PACKET\_THRESHOLD) ){  drop\_table.apply();  return;}  }  ipv4\_lpm.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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  packet.emit(hdr.tcp); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 

# 20)IP Forwarding two tables.p4

**Introduction**

The objective of this exercise is to write a P4 program that implements basic forwarding but using two tables. This program is a very simple extension of the ip\_forwarding example.

**How to run**

Run the topology:

sudo p4run

Try to ping from one host to another:

mininet> h1 ping h2

Ping from all host pairs to test for connectivity:

mininet> pingall

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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 \*/  bit<8> nhop\_index;  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  }\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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){  TYPE\_IPV4: ipv4;  default: accept; }  }  state ipv4 {  packet.extract(hdr.ipv4);  transition accept;  }  }\*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action set\_nhop\_index(bit<8> index){  meta.nhop\_index = index;  }  action \_forward(macAddr\_t dstAddr, egressSpec\_t port) {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table  hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl -1;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  set\_nhop\_index;  drop;  NoAction; }  size = 1024;  default\_action = NoAction();  }  table forward {  key = {  meta.nhop\_index: exact; }  actions = {  \_forward;  NoAction;  }  size = 64;  default\_action = NoAction();  }  apply {  //only if IPV4 the rule is applied. Therefore other packets will not be forwarded.  if (hdr.ipv4.isValid()){  if (ipv4\_lpm.apply().hit) {  forward.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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 21)Stateful Firewall.p4

**Introduction**

Switch S1 is acting as a stateful firewall, it will only allow connections to be establish from h1. h2 is only able to reply to connections once they have been established from h1.

Note: This stateful firewall is implemented 100% in the dataplane, meanning that to check if a connection was established a bloom filter is used. Thus, there is a probability of having collisions that would let unwanted flows pass. Of course this is just a toy example for educational purposes. A more realistic implementation would require the use of the Control Plane.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  /\* CONSTANTS \*/  const bit<16> TYPE\_IPV4 = 0x800;  const bit<8> TYPE\_TCP = 6;  #define BLOOM\_FILTER\_ENTRIES 4096  #define BLOOM\_FILTER\_BIT\_WIDTH 1  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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;  }  header tcp\_t{  bit<16> srcPort;  bit<16> dstPort;  bit<32> seqNo;  bit<32> ackNo;  bit<4> dataOffset;  bit<4> res;  bit<1> cwr;  bit<1> ece;  bit<1> urg;  bit<1> ack;  bit<1> psh;  bit<1> rst;  bit<1> syn;  bit<1> fin;  bit<16> window;  bit<16> checksum;  bit<16> urgentPtr;  }  struct metadata {  bit<32> register\_position\_one;  bit<32> register\_position\_two;  bit<1> register\_cell\_one;  bit<1> register\_cell\_two;  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  tcp\_t tcp;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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){  TYPE\_IPV4: ipv4;  default: accept; }  }  state ipv4 {  packet.extract(hdr.ipv4);  transition select(hdr.ipv4.protocol){  TYPE\_TCP: tcp;  default: accept;  } }  state tcp {  packet.extract(hdr.tcp);  transition accept; }  }  \*\*\*\*\*\*\*\*\*\* 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) {  register<bit<BLOOM\_FILTER\_BIT\_WIDTH>>(BLOOM\_FILTER\_ENTRIES) bloom\_filter;  action drop() {  mark\_to\_drop(standard\_metadata);  }  action set\_allowed(){  //Get register position  hash(meta.register\_position\_one, HashAlgorithm.crc16, (bit<32>)0, {hdr.ipv4.srcAddr,  hdr.ipv4.dstAddr,  hdr.tcp.srcPort,  hdr.tcp.dstPort,  hdr.ipv4.protocol},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  hash(meta.register\_position\_two, HashAlgorithm.crc32, (bit<32>)0, {hdr.ipv4.srcAddr,  hdr.ipv4.dstAddr,  hdr.tcp.srcPort,  hdr.tcp.dstPort,  hdr.ipv4.protocol},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  //set bloom filter fields  bloom\_filter.write(meta.register\_position\_one, 1);  bloom\_filter.write(meta.register\_position\_two, 1);  }  action check\_if\_allowed(){  //Get register position  hash(meta.register\_position\_one, HashAlgorithm.crc16, (bit<32>)0, {hdr.ipv4.dstAddr,  hdr.ipv4.srcAddr,  hdr.tcp.dstPort,  hdr.tcp.srcPort,  hdr.ipv4.protocol},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  hash(meta.register\_position\_two, HashAlgorithm.crc32, (bit<32>)0, {hdr.ipv4.dstAddr,  hdr.ipv4.srcAddr,  hdr.tcp.dstPort,  hdr.tcp.srcPort,  hdr.ipv4.protocol},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  //Read bloom filter cells to check if there are 1's  bloom\_filter.read(meta.register\_cell\_one, meta.register\_position\_one);  bloom\_filter.read(meta.register\_cell\_two, meta.register\_position\_two);  }  action ipv4\_forward(macAddr\_t dstAddr, egressSpec\_t port) {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table  hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl -1;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  ipv4\_forward;  drop;  NoAction;  }  size = 1024;  default\_action = NoAction();  }  apply {  if (hdr.ipv4.isValid()){  if (hdr.tcp.isValid()){  // Packet comes from internal network  if (standard\_metadata.ingress\_port == 1)  {  //If there is a syn we update the bloom filter and add the entry  if (hdr.tcp.syn == 1){  set\_allowed();  } }  // Packet comes from outside  else if (standard\_metadata.ingress\_port == 2){  check\_if\_allowed();  // we let the flow pass  if (meta.register\_cell\_one != 1 || meta.register\_cell\_two != 1){  drop();  return;  } } }  ipv4\_lpm.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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  packet.emit(hdr.tcp); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 22)Ternary Match.p4

**Introduction**

Very simple forwarding program that uses a ternary match. See s1-commands.txt to see how to populate tables with ternary matches using the CLI API. You will see that matches are of the form value&mask, for example: 0x00000000&&&0x80000000, and the last action parameter is used as priority (lower better).

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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){  TYPE\_IPV4: ipv4;  default: accept; }  }  state ipv4 {  packet.extract(hdr.ipv4);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action ipv4\_forward(macAddr\_t dstAddr, egressSpec\_t port) {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table  hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl -1;  }  table ternary\_table {  key = {  hdr.ipv4.dstAddr: ternary;}  actions = {  ipv4\_forward;  drop;  NoAction;  }  size = 1024;  default\_action = NoAction();  }  apply {  //only if IPV4 the rule is applied. Therefore other packets will not be forwarded.  if (hdr.ipv4.isValid()){  ternary\_table.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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 

# 

# 23)IP forwarding 2 examples.p4

**Hierarchical Forwarding**

Run the topology:

sudo p4run

Runs controller and populates 50k entries:

sudo python controller populate 50000

Try to ping from one host to another:

mininet> h1 ping h2

Simultaneously ping h2 and h3

mx h1

ping 10.0.2.2

#another terminal

mx h1

ping 10.250.250.2

To fail the link between s1 and s2

mininet> link s1 s2 down

Update the switch so it reroutes the traffic and see how long does it take until h2 and h3 are reachable again.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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 \*/  bit<8> nhop\_index;  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  }\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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){  TYPE\_IPV4: ipv4;  default: accept;  }  }  state ipv4 {  packet.extract(hdr.ipv4);  transition accept;  }  }  \*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action set\_nhop\_index(bit<8> index){  meta.nhop\_index = index;  }  action \_forward(macAddr\_t dstAddr, egressSpec\_t port) {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table  hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl -1;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  set\_nhop\_index;  drop;  NoAction;  }  size = 100000;  default\_action = NoAction();  }  table forward {  key = {  meta.nhop\_index: exact;  }  actions = {  \_forward;  NoAction; }  size = 64;  default\_action = NoAction(); }  apply {  //only if IPV4 the rule is applied. Therefore other packets will not be forwarded.  if (hdr.ipv4.isValid()){  if (ipv4\_lpm.apply().hit) {  forward.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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  }  }\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# EXTRA EXAMPLES

# TO KNOW MORE ABOUT P4:

# \*L2 Basic Forwarding.p4

**Introduction**

In today's first exercise we will implement a very basic layer 2 forwarding switch. In order to tell the switch how to forward frames, the switch needs to know in which port it can find a given MAC address (hosts). Real life switches automatically learn this mapping by using the l2 learning algorithm (we will see this later today). In order to familiarize ourselves with tables and how to map ethernet addresses to a given host (port) we will implement a very basic l2 forwarding that statically maps mac addresses to ports.

In summary, your tasks are:

1. Define the ethernet header type and an empty metadata struct called meta. Then define the headers struct with an ethernet header.
2. Parse the ethernet header.
3. Define a match-action table to make switch behave as a l2 packet forwarder. The destination Mac address of each packet should tell the switch which output port use. You can use your last exercise as a reminder, or check the [documentation](https://github.com/nsg-ethz/p4-learning/blob/master/documentation/control-plane.md).
4. Define the action the table will call for matching entries. The action should get the output port index as a parameter and set it to the egress\_spec switch's metadata field.
5. Apply the table you defined.
6. Deparse the ethernet header to add it back to the wire.
7. Write the s1-commands.txt file. This file should contain all the cli commands needed to fill the forwarding table you defined in 3.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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;  }  struct metadata {  /\* empty \*/  }  struct headers {  ethernet\_t ethernet;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  parser MyParser(packet\_in packet,  out headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {  state start {  packet.extract(hdr.ethernet);  transition accept;  }}  \*\*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action forward(bit<9> egress\_port) {  standard\_metadata.egress\_spec = egress\_port;  }  table dmac {  key = {  hdr.ethernet.dstAddr: exact;  }  actions = {  forward;  NoAction;  }  size = 256;  default\_action = NoAction; }  apply {  dmac.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 {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 

# \*L2 learning.p4

**Introduction**

This is the last exercise of our Basic L2 Switch series. In the first exercise we implemented a very basic l2 forwarding switch, then in the second exercise we made the switch a bit more realistic and added the feature of forwarding packets for unknown and broadcast destinations.

L2 learning works as follows:

1. For every packet the switch receives, it checks if it has seen the src\_mac address before. If its a new mac address, it sends to the controller a tuple with (mac\_address, ingress\_port). The controller receives the packet and adds two rules into the switch's tables. First it tells the switch that src\_mac is known. Then, in another table it adds an entry to map the mac address to a port (this table would be the same we used in the previous exercises).
2. The switch also checks if the dst\_mac is known (using a normal forwarding table), if known the switch forwards the packet normally, otherwise it broadcasts it.

Tasks to be performed:-

1. Define a cpu\_t header that will be added to our original packet. This header needs two fields, one for the source mac address, and one for the input port (48 and 16 bits respectively). Remember to cast the standard\_metadata.ingress\_port before assigning it to this header field (the standard metadata field is 9 bits, but we need to send a multiple of 8 to the controller, and thus we use 16 bits).
2. Cloned packets get all the metadata reset. If we want to be able to know the ingress\_port for our cloned packet we will need to put that in a metadata field.
3. Add the new header to the headers struct.
4. Define a normal forwarding table, and call it dmac. The table should match to the packet's destination mac address, and call a function forward that sets the output port. Set NoAction as default. Copy this from the previous exercise.
5. Define a table named broadcast that matches to ingress\_port and calls the action set\_mcast\_grp which sets the multicast group for the packet, if needed. Define also the set\_mcast\_grp action. Copy this from the previous exercise.
6. Define a third and new table (and name it smac). This new table will be used to match source mac addresses. If there is a match nothing should happen, if there is a miss, an action mac\_learn should be called. The mac\_learn action should set the metadata field you defined in 3 to standard\_metadata.ingress\_port and call clone3 with CloneType.I2E and mirroring ID = 100.
7. Write the apply logic. First apply the smac table. Then the dmac and if it does not have a hit, apply the broadcast table.
8. When you call clone3 the packet gets copied to the egress pipeline. Here you have to do several things.
   1. First check that the instance\_type is equal to 1 (which means that the packet is an ingress clone).
   2. Now you will use the cpu header you defined in 2 to add the learning information we want to send to the controller. To enable the header you need to set it valid using setValid(). Fill the cpu headers fields with the mac source port and ingress\_port.
   3. Finally set the hdr.ethernet.etherType to 0x1234. The controller uses to filter packets.
9. Emit the new header you created (only valid headers are put back to the packet).
10. Implement the controllers learning function:

Reset the switch state.

Add Broadcast groups automatically.

Add the mirror session ID and map it to the CPU\_PORT.

It will listen for learning packets and will parse them.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  const bit<16> TYPE\_BROADCAST = 0x1234;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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;  }  struct learn\_t {  bit<48> srcAddr;  bit<9> ingress\_port;  }  struct metadata {  /\* empty \*/  learn\_t learn;  }  struct headers {  ethernet\_t ethernet;  }\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  parser MyParser(packet\_in packet,  out headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {  state start {  packet.extract(hdr.ethernet);  transition accept;  }  }  \*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action mac\_learn(){  meta.learn.srcAddr = hdr.ethernet.srcAddr;  meta.learn.ingress\_port = standard\_metadata.ingress\_port;  digest(1, meta.learn);  }  table smac {  key = {  hdr.ethernet.srcAddr: exact;  }  actions = {  mac\_learn;  NoAction;  }  size = 256;  default\_action = mac\_learn;  }  action forward(bit<9> egress\_port) {  standard\_metadata.egress\_spec = egress\_port;  }  table dmac {  key = {  hdr.ethernet.dstAddr: exact;  }  actions = {  forward;  NoAction;  }  size = 256;  default\_action = NoAction;  }  action set\_mcast\_grp(bit<16> mcast\_grp) {  standard\_metadata.mcast\_grp = mcast\_grp;  }  table broadcast {  key = {  standard\_metadata.ingress\_port: exact;  }  actions = {  set\_mcast\_grp;  NoAction;  }  size = 256;  default\_action = NoAction; }  apply {  smac.apply();  if (dmac.apply().hit){  //  }  else { broadcast.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 {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# \*L2 flooding.p4

**Introduction**

In the previous exercise we implemented a very basic l2 forwarding switch that only knows how to forward packets for which it knows the MAC destination address. In this exercise we will move one step forward towards our more realistic l2 switch. When a l2 switch does not know to which port to forward a frame or the MAC destination address is ff:ff:ff:ff:ff:ff the switch sends the packet to all the ports but the one it came from.

In this exercise, first you will have to implement a simplified version in which packets get forwarded to all ports, once that works, you will have to implement the real l2 flooding, in which packets do not get sent to the port they came from.

**Flooding to all ports**

To complete this exercise we will need to define multicast groups, a feature provided by the simple\_switch target. Multicast enables us to forward packets to multiple ports. You can find some documentation on how to set multicast groups in the [simple switch](https://github.com/nsg-ethz/p4-learning/blob/master/documentation/simple-switch.md#creating-multicast-groups) documentation.

Your tasks are:

1. Read the documentation section that talks about multi cast.
2. Define a multicast group with id=1. Create a multicast node that contains all the ports and associate it with the multicast group.
3. Define a match-action table to make switch behave as a l2 packet forwarder. The destination mac address of each packet should tell the switch witch output port use.
4. **Hint**: you can directly copy the table you defined in the previous exercise, and populate it with the same mac to port entries.
5. Add an extra action to the table and name it broadcast. This action should be called when there is no hit in the forwarding table (unknown Mac or ff:ff:ff:ff:ff:ff). You can set it as a default action either directly in the table description or using the table\_set\_default cli command.
6. Define the broadcast action. This action has to set the standard\_metadata.mcast\_grp to the multicast group id we want to use (in our case 1).
7. Apply the table.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  const bit<16> TYPE\_BROADCAST = 0x1234;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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;  }  struct metadata {  /\* empty \*/  }  struct headers {  ethernet\_t ethernet;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  parser MyParser(packet\_in packet,  out headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {  state start {  packet.extract(hdr.ethernet);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action forward(bit<9> egress\_port) {  standard\_metadata.egress\_spec = egress\_port;  }  action broadcast() {  standard\_metadata.mcast\_grp = 1;  }  table dmac {  key = {  hdr.ethernet.dstAddr: exact; }  actions = {  forward;  broadcast;  NoAction;  }  size = 256;  default\_action = NoAction; }  apply {  dmac.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 {  //parsed headers have to be added again into the packet.  packet.emit(hdr.ethernet); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 

# \*Multiprotocol label switching.p4 (MPLS)

In this exercise you will see a different forwarding technique: Multiprotocol Label Switching. Your final objective will be to implement a basic version of MPLS forwarding in the data plane.

MPLS attaches labels to data packets to drive the packet forwarding decisions. That is, instead of forwarding packets based on their IP addresses, switches forward packets just by looking up the contents of the MPLS labels that are attached to packets.

As you will see, MPLS has numerous benefits such as the possibility of creating end-to-end circuits for all types of packets or an extensive support for traffic engineering.

**MPLS Terminology**

Before we begin, we need some vocabulary to understand the concepts better:

● Multiprotocol Label Switching (MPLS): A highly scalable, data-carrying mechanism that is independent of any data link layer protocol.

● Label Edge Router (LER): A router that operates at the edges of an MPLS network. An LER determines and applies the appropriate labels and forwards the labeled packets into the MPLS domain.

● Label Switch Router (LSR): A router that switches the labels that are used to route packets through an MPLS network. You can understand LSRs as *all* the MPLS-capable switches in the network. LERs are also LSRs.

● Label Switched Path (LSP): A route through an MPLS network, defined by a signaling protocol such as the Border Gateway Protocol (BGP). The path is set up based on criteria in the forwarding equivalence class (FEC).

● Forwarding Equivalence Class (FEC): A set of packets with similar characteristics that might be bound to the same MPLS label. **An FEC tends to correspond to a label switched path (LSP); however, an LSP might be used for multiple FECs.**

MPLS mainly relies on three features:

1. A Match-Action Table that maps classes of packets into the desired labels (i.e., FTN).
2. Labels that are carried by the packets.
3. A Match-Action Table that maps labels to egress ports to execute the forwarding (i.e., the NHLFE).
4. As we do in order exercises, we provide you some files that will help you through the exercise.

● p4app.json: describes the topology that you will use throughout the exercise.

● p4src/basics.p4: contains the p4 program skeleton that you will use as a starting point for this first section of the exercise (mock MPLS).

● p4src/stacked.p4: contains the p4 program skeleton that you will use as a starting point for the second section of the exercise (realistic MPLS).

In this exercise, you will work with the following topology. Your objective is to enable communication from h1 to h2 and h3, by implementing different functionalities of MPLS.

|  |
| --- |
| Code: /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  const bit<16> TYPE\_MPLS = 0x8847;  #define CONST\_MAX\_PORTS 32  #define CONST\_MAX\_LABELS 10  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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 mpls\_t  {  bit<20> label;  bit<3>exp;  bit<1>s;  bit<8>ttl;  }  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 {  bit<1> is\_ingress\_border;  bit<1> is\_egress\_border;  }  struct headers {  ethernet\_t ethernet;  mpls\_t mpls;  ipv4\_t ipv4;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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\_MPLS: parse\_mpls;  TYPE\_IPV4: parse\_ipv4;  default: accept;  } }  state parse\_mpls {  packet.extract(hdr.mpls);  transition parse\_ipv4;  }  state parse\_ipv4 {  packet.extract(hdr.ipv4);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action set\_is\_ingress\_border(){  meta.is\_ingress\_border = (bit<1>)1;  }  table check\_is\_ingress\_border {  key = {  standard\_metadata.ingress\_port: exact;  }  actions = {  NoAction;  set\_is\_ingress\_border;  }  default\_action = NoAction;  size = CONST\_MAX\_PORTS;  }  action add\_mpls\_header(bit<20> tag) {  hdr.mpls.setValid();  hdr.mpls.label = tag;  hdr.mpls.s = 0;  hdr.mpls.ttl = 255;  hdr.ethernet.etherType = TYPE\_MPLS;  }  table fec\_to\_label {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  NoAction;  add\_mpls\_header;  }  default\_action = NoAction;  size = CONST\_MAX\_LABELS;  }  action mpls\_forward(macAddr\_t dstAddr, egressSpec\_t port)  {  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  hdr.ethernet.dstAddr = dstAddr;  standard\_metadata.egress\_spec = port;  hdr.mpls.ttl = hdr.mpls.ttl - 1;  }  table mpls\_tbl {  key = {  hdr.mpls.label: exact;  }  actions = {  mpls\_forward;  drop;  }  default\_action = drop;  size = CONST\_MAX\_LABELS;  }  action ipv4\_forward(macAddr\_t dstAddr, egressSpec\_t port) {  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  hdr.ethernet.dstAddr = dstAddr;  standard\_metadata.egress\_spec = port;  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  ipv4\_forward;  drop;  }  default\_action = drop;  size = 128;  }  apply {  // We check if it is an ingress border port  check\_is\_ingress\_border.apply();  if(meta.is\_ingress\_border == 1){  // We need to check if the header is valid since mpls label is based on dst ip  if(hdr.ipv4.isValid()){  // We add the label based on the destination  fec\_to\_label.apply();  }  }  // We select the egress port based on the mpls label  if(hdr.mpls.isValid()){  mpls\_tbl.apply();  }  // We implement normal forwarding  else if (hdr.ipv4.isValid())  {  ipv4\_lpm.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) {  action is\_egress\_border(){  // We remove the mpls header  hdr.mpls.setInvalid();  hdr.ethernet.etherType = TYPE\_IPV4;  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;  }  table check\_is\_egress\_border {  key = {  standard\_metadata.egress\_port: exact;  }  actions = {  NoAction;  is\_egress\_border;  }  default\_action = NoAction;  size = CONST\_MAX\_PORTS;  } apply {  // We check if it is an egress border port  if (hdr.mpls.isValid()){  check\_is\_egress\_border.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 {  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.mpls);  packet.emit(hdr.ipv4); }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 

# 

# 

# 

# 

# 

# 

# 

# \*Recirculate.p4

**Introduction**

In this example we show how to recirculate packets.

The program itself does nothing. It just recirculates all packets 5 times and finally sends them to port 2. You can see that we use metadata structs to carry the recirculate counter in order to track how many times the packet has been recirculated.

The program also shows you a trick to debug programs. De debugging trick uses a table to match some fields so their value gets displayed in the log file (log/s1.log).

**How to run**

Start topology:

sudo p4run

Send some traffic. And then check the log file at log/s1.log, you will see how the counter increases at each recirculation when matching the debug table.

|  |
| --- |
| Code: #include <core.p4>  #include <v1model.p4>  const bit<16> TYPE\_IPV4 = 0x800;  #define RECIRCULATE\_TIMES 5  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  typedef bit<9> egressSpec\_t;  typedef bit<48> macAddr\_t;  typedef bit<32> ip4Addr\_t;  /\* TCP/IP Headers \*/  header ethernet\_t {  macAddr\_t dstAddr;  macAddr\_t srcAddr;  bit<16> etherType;  }  header ipv4\_t {  bit<4>version;  bit<4>ihl;  bit<8>tos;  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 {  bit<8> counter;  }  struct headers {  ethernet\_t ethernet;  ipv4\_t ipv4;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  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){  TYPE\_IPV4: ipv4;  default: accept;  } }  state ipv4 {  packet.extract(hdr.ipv4);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4); }  }  \*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  apply {  standard\_metadata.egress\_spec = 2;  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\* 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) {  //To debug  table debug {  key = {  meta.counter: exact;}  actions = {  NoAction;}  size=1;  default\_action=NoAction();  }  apply {  debug.apply();  if (meta.counter < RECIRCULATE\_TIMES){  meta.counter = meta.counter + 1;  recirculate({meta.counter});  }  }  }  \*\*\*\*\*\*\*\*\*\* 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 {  update\_checksum(  hdr.ipv4.isValid(),  {  hdr.ipv4.version,  hdr.ipv4.ihl,  hdr.ipv4.tos,  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# 

# 

# \*Recirculate and add headers.p4

**Introduction**

In this example we show how to recirculate packets and add a headers at each recirculation. At each recirculation it reads the content of a register and adds it to the new header.

To test the program we provide a sending carrier application that sends special packets that indicate how many times do they have to be recirculated.

We also provide a script to automatically fill a register[i]=i values where i in [0,127].

**How to run**

Start topology:

sudo p4run

Fill register:

python fill\_register.py

Send Magic packets:

mx h1

python send\_carrier.py <num\_recirculations>

You can monitor packets at the output port of the switch to see that the number of bytes increases as we recirculate more. You can also inspect the value of those packets and verify that they carry the register's content using wireshark or any other tool.

|  |
| --- |
| Code: #include <core.p4>  #include <v1model.p4>  #include "include/constants.p4"  #include "include/headers.p4"  #include "include/parsers.p4"  \*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  apply {  standard\_metadata.egress\_spec = 2;  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\* 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) {  register<bit<8>>(128) recirculate\_register;  apply {  if (hdr.recirculate\_header.isValid()) {  if (meta.recirculate\_meta.stack\_length < hdr.recirculate\_header.length) {  hdr.recirculate\_last.setValid();  recirculate\_register.read(hdr.recirculate\_last.value, (bit<32>)meta.recirculate\_meta.stack\_length);  meta.recirculate\_meta.stack\_length = meta.recirculate\_meta.stack\_length +1;  meta.recirculate\_meta.stack\_length\_tmp = meta.recirculate\_meta.stack\_length;  if (meta.recirculate\_meta.stack\_length < hdr.recirculate\_header.length) {  recirculate(meta);  // This should in theory work but there is a bug in p4c  //recirculate(meta.recirculate\_meta);  }  }  }  }  }  \*\*\*\*\*\*\*\*\* 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 {  update\_checksum(  hdr.ipv4.isValid(),  {  hdr.ipv4.version,  hdr.ipv4.ihl,  hdr.ipv4.tos,  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# \*Flowlet Switching.p4

**Introduction**

In the previous exercise we implemented ECMP, a very basic (but widely used) technique to load balance traffic across multiple equal cost paths. ECMP works very well when it has to load balance many small flows with similar sizes (since it randomly maps them to one of the possible paths). However, real traffic does not look as described above, real traffic is composed by many small flows, but also but very few that are quite bigger. This makes ECMP suffer from a well-known performance problem such as hash collisions, in which few big flows end up colliding in the same path. In this exercise we will use state and information provided by the simple\_switch's standard\_metadata to fix the collision problem of ECMP, by implementing flowlet switching on top.

Flowlet switching leverages the burstiness of TCP flows to achieve a better load balancing. TCP flows tend to come in bursts (for instance because a flow needs to wait to get window space). Every time there is gap which is big enough (i.e., 50ms) between packets from the same flow, flowlet switching will rehash the flow to another path (by hashing an ID value together with the 5-tuple).

## About program

To successfully complete the exercise you have to do the following:

1. Like in the previous exercise, header definitions are already provided.
2. Define the parser that is able to parse packets up to tcp. Note that for simplicity we do not consider udp packets in this exercise. This time you must define the parser in: p4src/include/parsers.p4.
3. Define the deparser. Just emit all the headers.
4. Copy the tables and actions from the previous exercise. You will have to slightly modify them.
5. Define two registers flowlet\_to\_id and flowlet\_time\_stamp (for register sizing use the constant defined at the beginning of flowlet\_switching.p4 file: REGISTER\_SIZE, TIMESTAMP\_WIDTH, ID\_WIDTH). We will use this two registers to keep two things:
   1. In flowlet\_to\_id register we keep the id (a random generated number) of each flowlet, this id is now added to the hash function that devices the output port. As long as this id does not change, packets for that flow will stay in the same path.
   2. In flowlet\_time\_stamp register we keep the last timestamp for the last observed packet belonging to a flow.
6. Define an action to read the flowlet's register values (read\_flowlet\_registers). In this action you will have to hash the 5-tuple of every packet the index you will use to read the flowlet registers (to save the index you will need to define a new metadata field with a width size of 14 bits). Using the index you got from the hash function read flowlet id and last timestamp and save them in a metadata field (you also have to define them). Finally, update the timestamp register using standard\_metadata.ingress\_global\_timestamp.
7. Define another action to update the flowlet id (update\_flowlet\_id). We will use this action to update flowlet ids when needed. In this action you just have to generate a random number, and then save it in the flowlet to id register (using the id you already computed previously).
8. Modify the hash function you defined in the ECMP exercise (ecmp\_group), now instead of just hashing the 5-tuple, you have to add the metadata field where you store the flowlet\_id you read from the register (or you just updated).
9. Define the ingress control logic (keep the logic from the ecmp example and add):

Before applying the ipv4\_lpm table:

* 1. Read the flowlet registers (calling the action)
  2. Compute the time difference between now and the last packet observed for the current flow.
  3. Check if the time difference is bigger than FLOWLET\_TIMEOUT (define at the beginning of the file with a default value of 200ms).
  4. Update the flowlet id if the difference is bigger. Updating the flowlet id will make the hash function output a new value.
  5. Apply ipv4\_lpm and ecmp\_group is the same way you did in ecmp.

1. Copy the sX-commands.txt from the previous exercise.

|  |
| --- |
| Code: #include <core.p4>  #include <v1model.p4>  //My includes  #include "include/headers.p4"  #include "include/parsers.p4"  #define REGISTER\_SIZE 8192  #define TIMESTAMP\_WIDTH 48  #define ID\_WIDTH 16  #define FLOWLET\_TIMEOUT 48w200000  \*\*\*\*\*\*\*\*\* 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) {  register<bit<ID\_WIDTH>>(REGISTER\_SIZE) flowlet\_to\_id;  register<bit<TIMESTAMP\_WIDTH>>(REGISTER\_SIZE) flowlet\_time\_stamp;  action drop()  {  mark\_to\_drop(standard\_metadata);  }  action read\_flowlet\_registers(){  //compute register index  hash(meta.flowlet\_register\_index, HashAlgorithm.crc16,  (bit<16>)0,  { hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.tcp.srcPort, hdr.tcp.dstPort,hdr.ipv4.protocol},  (bit<14>)8192);  //Read previous time stamp  flowlet\_time\_stamp.read(meta.flowlet\_last\_stamp, (bit<32>)meta.flowlet\_register\_index);  //Read previous flowlet id  flowlet\_to\_id.read(meta.flowlet\_id, (bit<32>)meta.flowlet\_register\_index);  //Update timestamp  flowlet\_time\_stamp.write((bit<32>)meta.flowlet\_register\_index, standard\_metadata.ingress\_global\_timestamp);  }  action update\_flowlet\_id()  {  bit<32> random\_t;  random(random\_t, (bit<32>)0, (bit<32>)65000);  meta.flowlet\_id = (bit<16>)random\_t;  flowlet\_to\_id.write((bit<32>)meta.flowlet\_register\_index, (bit<16>)meta.flowlet\_id);  }  action ecmp\_group(bit<14> ecmp\_group\_id, bit<16> num\_nhops){  hash(meta.ecmp\_hash,  HashAlgorithm.crc16,  (bit<1>)0,  {  hdr.ipv4.srcAddr,  hdr.ipv4.dstAddr,  hdr.tcp.srcPort,  hdr.tcp.dstPort,  hdr.ipv4.protocol,  meta.flowlet\_id},  num\_nhops);  meta.ecmp\_group\_id = ecmp\_group\_id;  }  action set\_nhop(macAddr\_t dstAddr, egressSpec\_t port)  {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table  hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;  }  table ecmp\_group\_to\_nhop {  key = {  meta.ecmp\_group\_id: exact;  meta.ecmp\_hash: exact;  }  actions = {  drop;  set\_nhop;  } size = 1024;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  set\_nhop;  ecmp\_group;  drop;  }  size = 1024;  default\_action = drop;  }  apply {  if (hdr.ipv4.isValid()){  @atomic {  read\_flowlet\_registers();  meta.flowlet\_time\_diff = standard\_metadata.ingress\_global\_timestamp - meta.flowlet\_last\_stamp;  //check if inter-packet gap is > 100ms  if (meta.flowlet\_time\_diff > FLOWLET\_TIMEOUT){  update\_flowlet\_id();  }  }  switch (ipv4\_lpm.apply().action\_run){  ecmp\_group: {  ecmp\_group\_to\_nhop.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 {  update\_checksum(  hdr.ipv4.isValid(),  { hdr.ipv4.version,  hdr.ipv4.ihl,  hdr.ipv4.dscp,  hdr.ipv4.ecn,  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);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# \*Traceroutable.p4

**Introduction**

In this exercise, we will extend our P4 router with an additional feature: responding to traceroute packets, i.e. packets where the IPv4 TTL (time to live) value is equal to 1. If a router receives such a packet, it generates an ICMP Time Exceeded message and sends it back to the original sender of the expired packet. Thus, our objective is to make the traceroute or similar tools work within our network.

In brief, traceroute works as follows: it sends several packets starting with TTL=1 and it increases the TTL by 1 for each consequent packet. As you already know, each internet router decreases the IP TTL by 1 and sends back ICMP messages when it reaches 0. By sending packets with an increasing TTL, traceroute is able to trace the path packets take from the source to a given destination.

**Understanding traceroute**

Before starting with the implementation, we will have a look at the packets that traceroute sends and receives. To do this, open a terminal and type sudo wireshark & to open Wireshark. Then, select the interface eth0 and start capturing traffic. You will see a lot of packets being sent and received, to remove them you can apply the following filter at the filter bar: ip.ttl < 10 || icmp. Now run the following command in the terminal:

sudo traceroute -n -q 1 -f 1 -m 1 -T ethz.ch

This will send a single traceroute (TCP) packet with TTL=1 towards ethz.ch. In Wireshark, you should see a TCP packet as well as the corresponding ICMP reply. Take a close look at these to packets to understand how a router builds these ICMP replies. You can use the ICMP replied by ethz.ch as a reference. For example, you can see how the IP header changes, what the ICMP header and body contain, etc.

Once you completed your implementation, you can test the program using the traceroute.py script or the real traceroute tool. If you use the traceroute tool, bear in mind that we only implemented replies to TCP packets while the tool sends UDP packets by default. Add the -T parameter to use TCP packets.

1. Start the topology (this will also compile and load the program).

sudo p4run

1. Run the controller.

python routing-controller.py

1. Check that you can ping:

mininet> pingall

1. Traceroute between two hosts:

You can either use our own implementation (traceroute.py) or the default traceroute tool:

for sport in range(6000,6020):

print traceroute(dst="10.6.2.2",sport=sport, dport=80)

mininet> h1 traceroute -n -w 0.5 -q 1 -T --sport=<src\_port> --port=<dst\_port> 10.6.2.2

(to make it faster, we disable DNS lookups (-n), decrease the waiting time -w, send only one packet per TTL (-q) and a fixed source and destination port (--sport,--dport) to get the actual path of a flow despite ECMP)

|  |
| --- |
| Code: #include <core.p4>  #include <v1model.p4>  //My includes  #include "include/headers.p4"  #include "include/parsers.p4"  #define IP\_ICMP\_PROTO 1  #define ICMP\_TTL\_EXPIRED 11  \*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action ecmp\_group(bit<14> ecmp\_group\_id, bit<16> num\_nhops){  hash(meta.ecmp\_hash,  HashAlgorithm.crc16,  (bit<1>)0,  { hdr.ipv4.srcAddr,  hdr.ipv4.dstAddr,  hdr.tcp.srcPort,  hdr.tcp.dstPort,  hdr.ipv4.protocol},  num\_nhops);  meta.ecmp\_group\_id = ecmp\_group\_id;  }  action set\_nhop(macAddr\_t dstAddr, egressSpec\_t port) {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table  hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;  }  table ecmp\_group\_to\_nhop {  key = {  meta.ecmp\_group\_id: exact;  meta.ecmp\_hash: exact;  }  actions = {  drop;  set\_nhop;  }  size = 1024;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;}  actions = {  set\_nhop;  ecmp\_group;  drop;  }  size = 1024;  default\_action = drop;  }  action set\_src\_icmp\_ip (bit<32> src\_ip){  hdr.ipv4\_icmp.srcAddr = src\_ip;  }  table icmp\_ingress\_port {  key = {  standard\_metadata.ingress\_port: exact;  } actions = {  set\_src\_icmp\_ip;  NoAction;  }  size=64; default\_action=NoAction;  }  apply {  //Only forward packets if they are IP and TTL > 1  if (hdr.ipv4.isValid() && hdr.ipv4.ttl > 1){  switch (ipv4\_lpm.apply().action\_run){  ecmp\_group: {  ecmp\_group\_to\_nhop.apply();  }  }  }  //Traceroute Logic (only for TCP probes)  else if (hdr.ipv4.isValid() && hdr.tcp.isValid() && hdr.ipv4.ttl == 1){  // Set new headers valid  hdr.ipv4\_icmp.setValid();  hdr.icmp.setValid();  // Set egress port == ingress port  standard\_metadata.egress\_spec = standard\_metadata.ingress\_port;  //Ethernet: Swap map addresses  bit<48> tmp\_mac = hdr.ethernet.srcAddr;  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  hdr.ethernet.dstAddr = tmp\_mac;  //Building new Ipv4 header for the ICMP packet  //Copy original header (for simplicity)  hdr.ipv4\_icmp = hdr.ipv4;  //Set destination address as traceroute originator  hdr.ipv4\_icmp.dstAddr = hdr.ipv4.srcAddr;  //Set src IP to the IP assigned to the switch interface  icmp\_ingress\_port.apply();  //Set protocol to ICMP  hdr.ipv4\_icmp.protocol = IP\_ICMP\_PROTO;  //Set default TTL  hdr.ipv4\_icmp.ttl = 64;  //And IP Length to 56 bytes (normal IP header + ICMP + 8 bytes of data)  hdr.ipv4\_icmp.totalLen= 56;  //Create ICMP header with  hdr.icmp.type = ICMP\_TTL\_EXPIRED;  hdr.icmp.code = 0;  //make sure all the packets are length 70.. so wireshark does not complain when tpc options,etc  truncate((bit<32>)70); }  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  update\_checksum(  hdr.ipv4.isValid(),  {  hdr.ipv4.version,  hdr.ipv4.ihl,  hdr.ipv4.dscp,  hdr.ipv4.ecn,  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);  update\_checksum(  hdr.ipv4\_icmp.isValid(),  {  hdr.ipv4\_icmp.version,  hdr.ipv4\_icmp.ihl,  hdr.ipv4\_icmp.dscp,  hdr.ipv4\_icmp.ecn,  hdr.ipv4\_icmp.totalLen,  hdr.ipv4\_icmp.identification,  hdr.ipv4\_icmp.flags,  hdr.ipv4\_icmp.fragOffset,  hdr.ipv4\_icmp.ttl,  hdr.ipv4\_icmp.protocol,  hdr.ipv4\_icmp.srcAddr,  hdr.ipv4\_icmp.dstAddr },  hdr.ipv4\_icmp.hdrChecksum,  HashAlgorithm.csum16);  update\_checksum(  hdr.icmp.isValid(),  { hdr.icmp.type,  hdr.icmp.code,  hdr.icmp.unused,  hdr.ipv4.version,  hdr.ipv4.ihl,  hdr.ipv4.dscp,  hdr.ipv4.ecn,  hdr.ipv4.totalLen,  hdr.ipv4.identification,  hdr.ipv4.flags,  hdr.ipv4.fragOffset,  hdr.ipv4.ttl,  hdr.ipv4.protocol,  hdr.ipv4.hdrChecksum,  hdr.ipv4.srcAddr,  hdr.ipv4.dstAddr,  hdr.tcp.srcPort,  hdr.tcp.dstPort,  hdr.tcp.seqNo  },  hdr.icmp.checksum,  HashAlgorithm.csum16);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

# \*Simple routing (ECMP.P4)

**Introduction**

The goal of this exercise is to implement and provide a control plane to the ECMP routing exercise from 2 weeks ago. Unlike in the previous exercises where you specified the entries for the forwarding tables manually, we will now implement a controller that generates and installs forwarding rules automatically, based on the network topology.

In traditional networks the control plane is the brain of any networking device. The control plane is in charge of deciding where packets have to be sent. Distributed control planes exchange topology information with other devices and compute which is the best way of sending traffic based on some routing protocol (RIP, OSPF, BGP).

To simplify things a lot in this exercise we will not implement a distributed and dynamic control plane like the ones mentioned above, but something simple, centralized and static. However, your controller should be able to automatically populate the ECMP exercise tables for any topology.

**Implementing the router's control plane program**

The main task of the controller (we provide a skeleton in routing-controller.py) is to translate the network topology (stored in topology.db) to match-action table entries. For example for the topology that we used in last week's ECMP exercise, it should run the following commands to fill the ipv4\_lpm and ecmp\_group\_to\_nhop tables in switch s1

You have to write your controller application in the routing-controller.py file that we already provided you. You will see that we already implemented some small functions that use the Topology and SimpleSwitchAPI objects from p4utils. Among others, the provided functions do:

1. connect\_to\_switches(): function that establishes a connection with the simple switch thrift server using the SimpleSwitchAPI object and saves those objects in the self.controllers dictionary. This dictionary has the form of: {'sw\_name' : SimpleSwitchAPI()}.
2. reset\_states(): iterates over the self.controllers object and runs the reset\_state function which will empty the state (registers, tables, etc) for every switch.
3. set\_table\_defaults(): for each p4 switch it sets the default action for ipv4\_lpm and ecmp\_group\_to\_nhop tables.

In this exercise your task is to implement the route function which is in charge of populating the table entries such that you can route traffic using the shortest path in the network. Furthermore, if multiple equal cost paths are found you have to assign them to an ECMP group.

At a high level, the route function should do the following:

1. Iterate over all pairs of switches in the topology
2. Compute all the shortest paths between each of these pairs of switches
3. Install the table entries needed depending on the following 3 scenarios:
   1. If source switch and destination switch are the same. Install an entry for each directly connected host: You need host ip (use /32), mac address, and in which port index it is connected to the switch.
   2. If there is a single path between src switch and destination switch and the destination switch has direct hosts connected: this time use the next hop to get the output port and the destination mac address.
   3. If there are multiple paths between src switch and destination switch and the destination switch has direct hosts connected: create a ecmp group (as in the example above) for all multiple next hops needed to reach the destination switch. If for the same source switch the same multiple hops have to be used for another destination use the already defined ecmp group.

To get information about the shortest paths, ip addresses, mac addresses, port indexes and how nodes are connected between each other you will have to strongly utilize the topology object from p4-utils. To implement the routing function you will have to strongly utilize the topology object from p4-utils.

|  |
| --- |
| Code: #include <core.p4>  #include <v1model.p4>  //My includes  #include "include/headers.p4"  #include "include/parsers.p4"  \*\*\*\*\*\*\*\*\*\*\* 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) {  action drop() {  mark\_to\_drop(standard\_metadata);  }  action ecmp\_group(bit<14> ecmp\_group\_id, bit<16> num\_nhops){  hash(meta.ecmp\_hash,  HashAlgorithm.crc16,  (bit<1>)0,  { hdr.ipv4.srcAddr,  hdr.ipv4.dstAddr,  hdr.tcp.srcPort,  hdr.tcp.dstPort,  hdr.ipv4.protocol},  num\_nhops);  meta.ecmp\_group\_id = ecmp\_group\_id;  }  action set\_nhop(macAddr\_t dstAddr, egressSpec\_t port) {  //set the src mac address as the previous dst, this is not correct right?  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  //set the destination mac address that we got from the match in the table  hdr.ethernet.dstAddr = dstAddr;  //set the output port that we also get from the table  standard\_metadata.egress\_spec = port;  //decrease ttl by 1  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;  }  table ecmp\_group\_to\_nhop {  key = {  meta.ecmp\_group\_id: exact;  meta.ecmp\_hash: exact;  }  actions = {  drop;  set\_nhop;  }  size = 1024;  }  table ipv4\_lpm {  key = {  hdr.ipv4.dstAddr: lpm;  }  actions = {  set\_nhop;  ecmp\_group;  drop;  }  size = 1024;  default\_action = drop;  }  apply {  if (hdr.ipv4.isValid()){  switch (ipv4\_lpm.apply().action\_run){  ecmp\_group: {  ecmp\_group\_to\_nhop.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 {  update\_checksum(  hdr.ipv4.isValid(),  {  hdr.ipv4.version,  hdr.ipv4.ihl,  hdr.ipv4.dscp,  hdr.ipv4.ecn,  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);  }  }\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //switch architecture  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main; |

**END**