**FINAL PROJECT**

**Name:** Burhanuddin Rangwala, Mohammad Amin Shaikh

**Domain:** Programmable software defined Networks

**Project Aim:** Investigate what kind of local network information on the switch can be obtained using P4.

**PROJECT IMPLEMENTATION IDEA FOR COUNTER**

* Counter is a stateful object of P4 which essentially counts the number of packets that the switch encounters.
* At first it was suggested to use UDP server to receive the network information, but the road hit a rock when writing a python client inside a BmV2 switch was a dead end.
* Instead, a workaround python script is devised which will send a packet like **probe packet** which contains the required headers in which this information will be stored by the switch on encountering this packet.
* For the switch to perform this function we will write a P4 code with required parser for our probe packet, a match-action pipeline, counter, bloom-filter or register array.
* Match-action pipeline will contain a single function called **tally** which will increment the counter associated to the match key which is **ingress port.**
* Simultaneously, we will compute a hash on **ingress port** using **hash** function provided by v1model library. This hash will serve as key to bloom-filter which will store the value of the number of packets encountered by that port by incrementing the value stored in bloom-filter on every table hit **count\_table** which manages **tally** function.
* After incrementing the count, the result is read from the filter and is put into header called **n\_packets** which has a 16-bit field called **num\_packets** where the value will be stored and will be directed towards the same port through which the probe packet came initially.
* As elicited earlier that a python script will be sending the packet, that same python script will receive the packet sent by the switch after putting in the required data and will be displayed.

**PROJECT IMPLEMENTATION IDEA FOR QUEUE DEPTH**

* Queue depth is pivotal in case of congestion control as the number it is holding determines the packet sending rate of the sender to avoid network congestion.
* This value can be accessed through P4 using struct **standard\_metadata\_t** which stores this value in the field **enq\_depth.**
* Same idea as counter was thought to be implemented but the issue faced was the number of bits allotted to **enq\_depth** is 19 which is an odd number and so it is non-viable for header accepted by BMV2 switch.
* So, it became difficult to implement the idea. Hence, it could not be incorporated into the header.

**PROJECT IMPLEMENTATION IDEA FOR METER**

* Meters are stateful objects that measure the data rate, either in packets or bytes per second, and output the result as one of three colours: red, yellow, or green, which are encoded as a 2-bit-wide field.
* For our implementation we will modify the probe packet header which we used for counter and add a new field called **meter\_val** and a new table **m\_read** in P4 code.
* The key to m\_read table is source mac address as we will mark the sender with the above-mentioned colour coding using the **extern meter object** provided by v1model.p4 library which keeps track of number of bytes each sender is sending using the meter and marks sender as **0(green), 1(yellow), 2(red).**
* Each table entry will be associated with independent meters just like counters.
* Whenever a hit occurs the meter value of the source is read and is stored in meta data with variable name **meter\_tag.** This value will subsequently be stored in **meter\_val** of the header **n\_packets.**
* We will run the same python script with modified header and read the value **meter\_val** along with **num\_packets** and display both values.
* Also, we have instructed switch to drop packets of sources whose meter value is 1 or 2 meaning those who have moderate and high traffic rate respectively.

**P4 CODE**

| /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  #define BLOOM\_FILTER\_ENTRIES 4096  #define BLOOM\_FILTER\_BIT\_WIDTH 16  const bit<16> TYPE\_IPV4 = 0x800;  const bit<16> TYPE\_S = 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;  }  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 pack\_tot{  bit<16> num\_packets;  bit<32> meter\_val;  }  struct metadata {  /\* empty \*/  bit<32> meter\_tag;  }  struct headers {  ethernet\_t ethernet;  pack\_tot pkts;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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\_S: n\_pack;  default: accept;  }  }  state n\_pack{  packet.extract(hdr.pkts);  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) direct\_port\_counter;  direct\_meter<bit<32>>(MeterType.packets) my\_meter;  register<bit<BLOOM\_FILTER\_BIT\_WIDTH>>(BLOOM\_FILTER\_ENTRIES) bloom\_filter\_1;  bit<32> position;  bit<16> result;  bit<19> res\_queue\_len;  action compute\_hashes(){  //Get register position  hash(position, HashAlgorithm.crc16, (bit<32>)0, {standard\_metadata.ingress\_port},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  }  action compute\_hashes\_custom(bit<9> port){  //Get register position  hash(position, HashAlgorithm.crc32, (bit<32>)0, {port},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  //bloom\_filter\_1.write(position, 0);  }  action m\_action(){  my\_meter.read(meta.meter\_tag);  }    action drop() {  mark\_to\_drop(standard\_metadata);  }  action send\_back(){  bit<48> temp = hdr.ethernet.srcAddr;  bit<48> srcAddr = hdr.ethernet.dstAddr;  hdr.ethernet.dstAddr = temp;  standard\_metadata.egress\_spec = standard\_metadata.ingress\_port;  }    action tally(){  direct\_port\_counter.count();  }  table m\_read {  key = {  hdr.ethernet.srcAddr: exact;  }  actions = {  m\_action;  NoAction;  }  default\_action = NoAction;  meters = my\_meter;  size = 16384;  }  table count\_table {  key = {  standard\_metadata.ingress\_port: exact;  }  actions = {  tally;  NoAction;  }  default\_action = NoAction;  counters = direct\_port\_counter;  size = 512;  }    apply {  if(count\_table.apply().hit){  compute\_hashes\_custom(standard\_metadata.ingress\_port);  bloom\_filter\_1.read(result, position);  if(result >= 0){  bloom\_filter\_1.write(position, (result + 1));  }else{  bloom\_filter\_1.write(position, 1);  }  bloom\_filter\_1.read(result, position);  hdr.pkts.num\_packets = result;  if(m\_read.apply().hit){  hdr.pkts.meter\_val = meta.meter\_tag;  }else{  hdr.pkts.meter\_val = 4;  }  send\_back();  }  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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 {  //hdr.pkts.queue\_len = standard\_metadata.enq\_qdepth;  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\* 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.pkts);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main;  The python script that is sending packets to the switch  #!/usr/bin/env python  import argparse  import sys  import socket  import random  import struct  from scapy.all import sendp, send, get\_if\_list, get\_if\_hwaddr,srp1  from scapy.all import Packet  from scapy.all import Ether, IP, UDP, TCP  from scapy.fields import IntField  from scapy.packet import bind\_layers  class Num\_Packet(Packet):  name = "Num\_Packet"  fields\_desc = [IntField("num\_packets",0), IntField("meter\_val", 0)]  bind\_layers(Ether, Num\_Packet, type=0x1234)  def get\_if():  ifs=get\_if\_list()  iface=None # "h1-eth0"  for i in get\_if\_list():  if "eth0" in i:  iface=i  break  if not iface:  print "Cannot find eth0 interface"  exit(1)  return iface  def main():  #addr = socket.gethostbyname(sys.argv[1])  iface = get\_if()  #print "sending on interface %s to %s" % (iface, str(addr))  pkt = Ether(src=get\_if\_hwaddr(iface), dst='00:04:00:00:00:00')  pkt = pkt /Num\_Packet(num\_packets = 0, meter\_val = 0)  pkt = pkt/' '  pkt.show2()  resp = srp1(pkt, iface=iface, timeout = 1, verbose=False)  resp.show2()  if resp:  num = resp[Num\_Packet]  if num:  print "number of packets till now",num.num\_packets>>16  print "the meter value is:", num.meter\_val>>32  else:  print "cannot find Num\_Packet header in the packet"  else:  print "Didn't receive response"  if \_\_name\_\_ == '\_\_main\_\_':  main() |
| --- |

**OUTPUT IMAGES:**

1. Python script running on Host1 connected to port1 will send the packet and get the counter and information for port 1.

Text

Description automatically generated

**To Verify we will run the BMV2 runtime client to extract the value of counte**Text

Description automatically generated

1. We will be running similar script on host 2 to get data for port 2

Text

Description automatically generated

**To verify our result we will look for counter value of port 2 in runtime-Client**

**Text

Description automatically generated**

**The above image is that of runtime Client**

1. We will be running similar script on host 3 to get data for port 3

Text

Description automatically generated

**To verify our results we will check counter value for port 3 in runtime-client**

Text

Description automatically generated

1. We will run a similar script on host 4 to get information for port 4

Text

Description automatically generated

**To verify our results we will check counter value for port 3 in runtime-client**

Text

Description automatically generated

**COMMANDS USED TO RUN THIS PROJECT**

* **make run:** This command will initialize our topology and mininet environment.
* **./send.py:** This command is used to run our python script which will send and receive our custom packet to retrieve counter and meter values.