P4 to implement a Simple Switch and to apply Diverting Traffic Functionality

Chethan Nagesh Pal
Student Id: 01741616
University of Massschusetts Lowell

Akshit Rastogi
Student Id: 01755668
University of Massachusetts Lowell

Joseph Kuncheria
Student Id: 01704690
University of Massachusetts Lowell

ABSTRACT

P4 is a high-level language for programming protocol-independent packet processors. P4 is a domain-specific programming language designed for expressing how packets are processed by the data plane of a programmable network element such as hardware or software switches, network interface cards, routers or network function appliances. P4 has three main characteristics**:** Programmers can change the way switches process packets. Switches should not be tied to any specific network protocol. Programmers can able to describe the packet processing functionality independent of the hardware.

I. INTRODUCTION

One of the most active areas of computer networking today is Software Defined Networking (SDN). SDN has separated the two core functions of a network-processing element (router): the control plane and the data plane. Traditionally both these functions were implemented on the same device; SDN decoupled them and allowed a variety of control plane implementations for each data plane. The characteristics of SDN is the Open Flow protocol, which specifies the API between the control plane and the data plane. Compared to the state-of-the-art method of programming network processing systems (e.g., writing microcode on top of custom hardware ASICS), P4 takes SDN to the next level by bringing programmability to the forwarding plane.

P4 provides significant advantages:
i) **Flexibility:** P4 makes many packet-forwarding policies expressible as programs; contrast this to traditional switches, which expose fixed-function forwarding engines to their users.
ii) **Expressiveness:** P4 programs may express sophisticated hardware-independent packet processing algorithms using solely general-purpose operations and table lookups.
iii) **Resource mapping and management:** P4 programs express resource usage in symbolic terms (e.g., IPv4 source address); compilers map such user-defined fields to available hardware resources and manage resource allocation and scheduling.
iv) **Software engineering:** P4 programs provide important benefits such as type checking, information hiding, and software reuse.
v) **Component libraries:** P4 supports the creation of libraries that wrap hardware-specific functions or standard protocol implementations into portable high-level P4 constructs.
vi) **Decoupling hardware and software evolution:** P4 is to a large degree architecture independent, allowing separate hardware and software upgrade cycles.
vii) **Debugging:** Manufacturers can provide their customers' software models of target architectures to aid in the development and debugging of P4 programs.

A P4-programmable switch differs from a traditional switch in two essential ways:

The data plane functionality is not fixed in advance but is defined by the a P4 program. The data plane is configured at initialization time to implement the functionality described by the P4 program (shown by the long red arrow) and has no built-in knowledge of existing network protocols.

The control plane communicates with the data plane using the same channels as in a fixed-function device, but the set of tables and other objects in the data plane are no longer fixed, since they are defined by a P4 program. The P4 compiler generates the API that the control plane uses to communicate with the data plane.

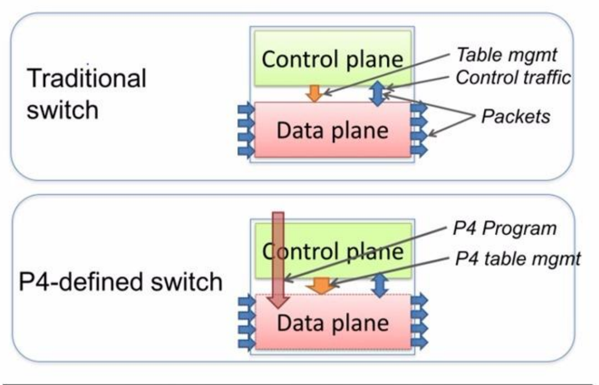


Figure 1. Traditional switches vs. programmable switches.

II. FORWARDING MODEL

The forwarding model is controlled by two types of operations:

A. *Configure*

Configure operations program the parser, set the order of match+action stages, and specify the header fields processed by each stage.

B. *Populate*

Populate operations add (and remove) entries to the match+action tables that were specified during configuration. Population determines the policy applied to packets at any given time.

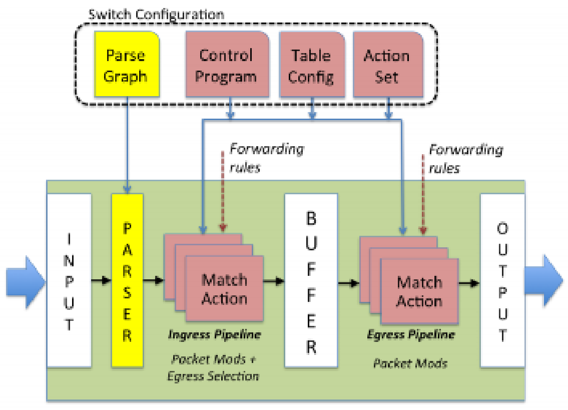


Figure 2

Packet are first handled by the parser. The parser recognizes and extracts fields from the header, and thus defines the protocols supported by the switch. The extracted header fields are then passed to the match+action tables.

The match+action tables are divided between ingress and egress. While both may modify the packet header, ingress match+action determines the egress port(s) and determines the queue into which the packet is placed. Based on ingress processing, the packet may be forwarded, replicated (for multicast, span, or to the control plane), dropped, or trigger flow control. Egress match+action performs per-instance modifications to the packet header – e.g., for multicast copies. Action tables (counters, policers, etc.) can be associated with a flow to track frame-to-frame state.

Packets can carry additional information between stages, called metadata, which is treated identically to packet header fields.

III. PROGAMMING LANGUAGE

P4 PROGRAMMING

• Headers: A header definition describes the sequence and structure of a series of fields. It includes specification of field widths and constraints on field values.
• Parsers: A parser definition specifies how to identify headers and valid header sequences within packets.
• Tables: Match+action tables are the mechanism for performing packet processing. The P4 program defines the fields on which a table may match and the actions it may execute.

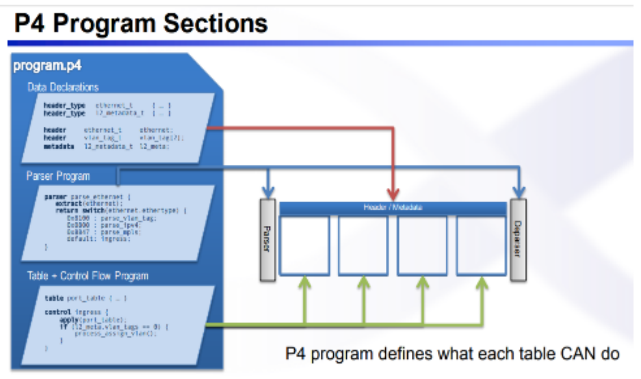


Figure 3

TOPOLOGY CREATION

The supporting files can be downloaded from P4.org.
2 hosts and 1 switch:
This topology has 2 hosts: h1 and h2 and one switch: s1. So, to build this topology and to connect the host to the switches, we run the script file run_demo.bash which exists in the P4.org in the Directory: targets/simple_router.

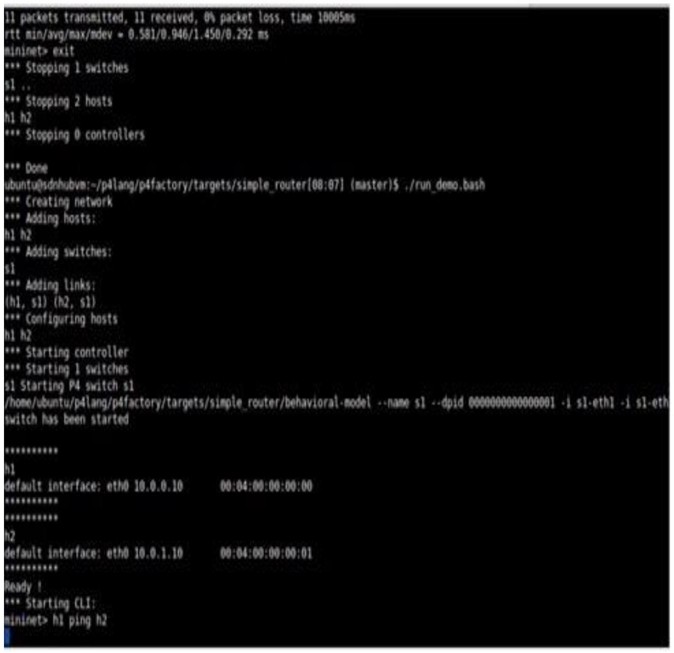
Working:
After running this bash file, we can link h1 to s1 and h2 to s2. In addition to this, we are creating a controller. Now, if we try to ping h1 with h2. The communication does not go through and no packets of h1 are received at the h2. This is because there are no table entries in switch which provide the destination address of h2. So, in typical network devices have control plane that provides information that is required to build the forwarding table. They also consist of a data plane that consults the forwarding table. Using this, the network devices decide where to send the packets.

Software Defined Networking abstracts this concept and places the Control Plane functions on an SDN controller. The SDN controller can be a server running SDN software. The Controller communicates with a physical or virtual switch Data Plane through a protocol called OpenFlow. OpenFlow conveys the instructions to the data plane on how to forward data.

In our project, we are using OpenFlow protocol to establish communication between h1 and h2. By running the run_add_demo_entries.bash file., we are populating the data plane. Any new incoming packets from the h1 will now pass through the parser which extracts the header sequence and passes this packet to the match action table which is present in the data plane. This action table provides us the further information om how the packet should be processed.

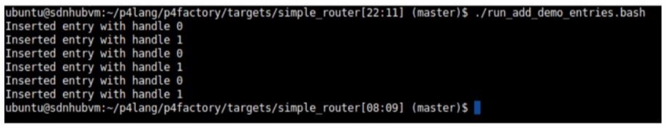
This implementation sets apart SDN networking with the traditional network. Here, you can see the top-down approach as explained in the figure 1.

IV. RESULTS



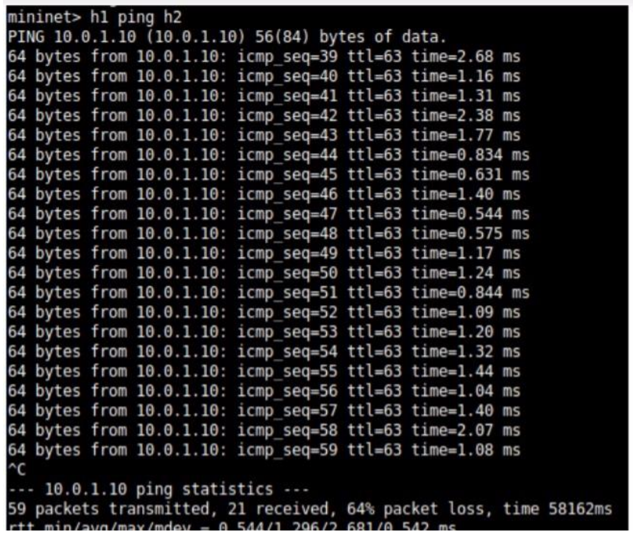
```
11 packets transmitted, 11 received, 0% packet loss, time 10005ms
rtt min/avg/max/mdev = 0.581/0.946/1.450/0.292 ms
mininet> exit
*** Stopping 1 switches
s1 ..
*** Stopping 2 hosts
h1 h2
*** Stopping 0 controllers

*** Done
ubuntu@sdnhubvm:~/p4lang/p4factory/targets/simple_router[08:07] (master)$ ./run_demo.bash
*** Creating network
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
*** Starting 1 switches
s1 Starting P4 switch s1
/home/ubuntu/p4lang/p4factory/targets/simple_router/behavioral-model --name s1 --dpid 0000000000000001 -i s1-eth1 -i s1-eth
switch has been started

**********
h1
default interface: eth0 10.0.0.10      00:04:00:00:00:00
**********
**********
h2
default interface: eth0 10.0.1.10      00:04:00:00:00:01
**********
Ready !
*** Starting CLI:
mininet> h1 ping h2
```

Figure 5. Topology createdof 2 hosts and 1 switch



```
ubuntu@sdnhubvm:~/p4lang/p4factory/targets/simple_router[22:11] (master)$ ./run_add_demo_entries.bash
Inserted entry with handle 0
Inserted entry with handle 1
Inserted entry with handle 0
Inserted entry with handle 1
Inserted entry with handle 0
Inserted entry with handle 1
ubuntu@sdnhubvm:~/p4lang/p4factory/targets/simple_router[08:09] (master)$
```

Figure 6. Populating the data plane



```
mininet> h1 ping h2
PING 10.0.1.10 (10.0.1.10) 56(84) bytes of data.
64 bytes from 10.0.1.10: icmp_seq=39 ttl=63 time=2.68 ms
64 bytes from 10.0.1.10: icmp_seq=40 ttl=63 time=1.16 ms
64 bytes from 10.0.1.10: icmp_seq=41 ttl=63 time=1.31 ms
64 bytes from 10.0.1.10: icmp_seq=42 ttl=63 time=2.38 ms
64 bytes from 10.0.1.10: icmp_seq=43 ttl=63 time=1.77 ms
64 bytes from 10.0.1.10: icmp_seq=44 ttl=63 time=0.834 ms
64 bytes from 10.0.1.10: icmp_seq=45 ttl=63 time=0.631 ms
64 bytes from 10.0.1.10: icmp_seq=46 ttl=63 time=1.40 ms
64 bytes from 10.0.1.10: icmp_seq=47 ttl=63 time=0.544 ms
64 bytes from 10.0.1.10: icmp_seq=48 ttl=63 time=0.575 ms
64 bytes from 10.0.1.10: icmp_seq=49 ttl=63 time=1.17 ms
64 bytes from 10.0.1.10: icmp_seq=50 ttl=63 time=1.24 ms
64 bytes from 10.0.1.10: icmp_seq=51 ttl=63 time=0.844 ms
64 bytes from 10.0.1.10: icmp_seq=52 ttl=63 time=1.09 ms
64 bytes from 10.0.1.10: icmp_seq=53 ttl=63 time=1.20 ms
64 bytes from 10.0.1.10: icmp_seq=54 ttl=63 time=1.32 ms
64 bytes from 10.0.1.10: icmp_seq=55 ttl=63 time=1.44 ms
64 bytes from 10.0.1.10: icmp_seq=56 ttl=63 time=1.04 ms
64 bytes from 10.0.1.10: icmp_seq=57 ttl=63 time=1.40 ms
64 bytes from 10.0.1.10: icmp_seq=58 ttl=63 time=2.07 ms
64 bytes from 10.0.1.10: icmp_seq=59 ttl=63 time=1.08 ms
^C
--- 10.0.1.10 ping statistics ---
59 packets transmitted, 21 received, 64% packet loss, time 58162ms
rtt min/avg/max/mdev = 0.544/1.296/2.681/0.542 ms
```

Figure 7. Packets received courtesy the forwarding action of the P4 switch

DIVERTING TRAFFIC

Until now, we created a topology and we were able to ping hosts h1 with h2 using match action table of the data plane. Here we are utilizing the characteristics of P4 programming and we are diverting the packets that is going to a specific destination to whichever destination we desire. The application of this functionality can be used to divert the traffic that is directed for a particular host and reduce the load on the host. For example, diverting the traffic of ticket requests on the airlines server during the festive season and avoiding the server getting crashed.

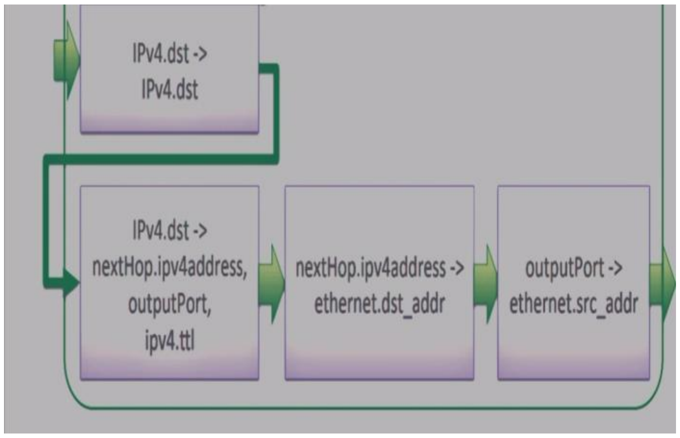
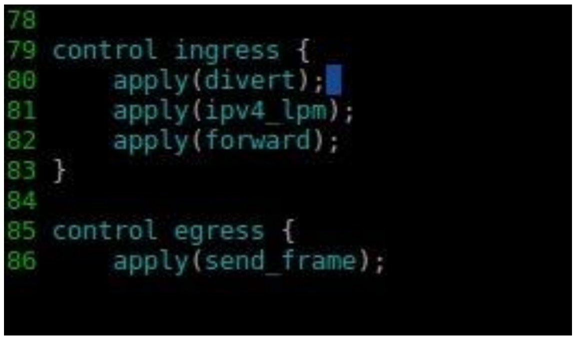

IPv4.dst -> IPv4.dst
IPv4.dst -> nextHop.ipv4address, outputPort, ipv4.ttl
nextHop.ipv4address -> ethernet.dst_addr
outputPort -> ethernet.src_addr

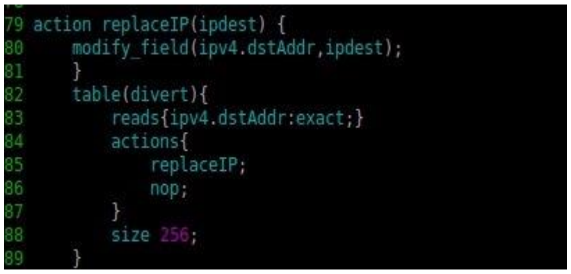
Figure 8. Diverting traffic

This functionality is being implemented in our project by calling apply(divert) in the control ingress and adding replaceIp action table in simple_router. p4.



```

control ingress {
    apply(divert);
    apply(ipv4_lpm);
    apply(forward);
}

control egress {
    apply(send_frame);
```

Figure 9. Adding apply(divert) function in the control ingress function.



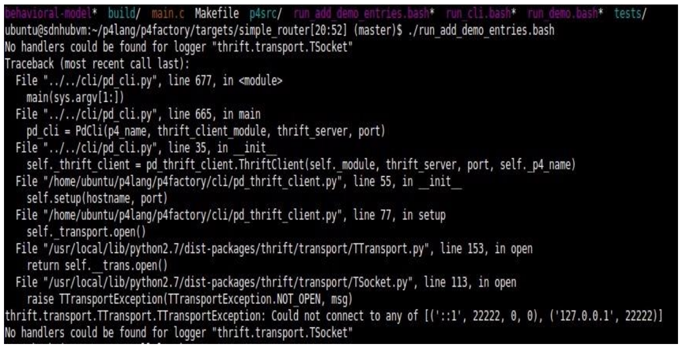
```
action replaceIP(ipdest) {
    modify_field(ipv4.dstAddr,ipdest);
    }
    table(divert){
        reads{ipv4.dstAddr:exact;}
        actions{
            replaceIP;
            nop;
        }
        size 256;
    }
```

Figure 10. replaceIp action table.

I. DIVERTING TRAFFIC RESULT

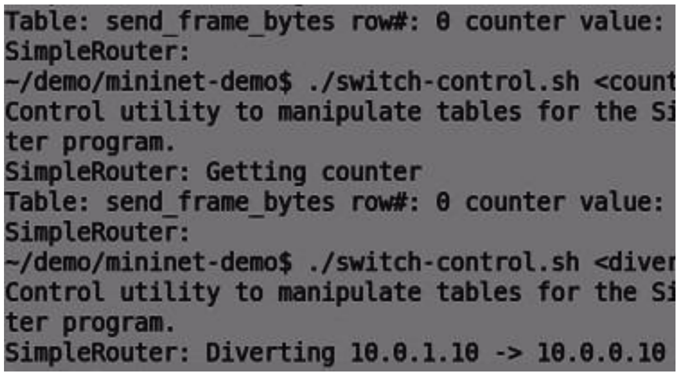
TROUBLESHOOTING

Once we add this new simple_router.p4 and we achieve successful compilation. But we are facing an error while adding the table entries in the data plane.



```
behavioral-model*  build/  main.c  Makefile  p4src/  run_add_demo_entries.bash*  run_cli.bash*  run_demo.bash*  tests/
ubuntu@sdnhubvm:~/p4lang/p4factory/targets/simple_router[20:52] (master)$ ./run_add_demo_entries.bash
No handlers could be found for logger "thrift.transport.TSocket"
Traceback (most recent call last):
  File "../../cli/pd_cli.py", line 677, in <module>
    main(sys.argv[1:])
  File "../../cli/pd_cli.py", line 665, in main
    pd_cli = PdCli(p4_name, thrift_client_module, thrift_server, port)
  File "../../cli/pd_cli.py", line 35, in __init__
    self._thrift_client = pd_thrift_client.ThriftClient(self._module, thrift_server, port, self._p4_name)
  File "/home/ubuntu/p4lang/p4factory/cli/pd_thrift_client.py", line 55, in __init__
    self.setup(hostname, port)
  File "/home/ubuntu/p4lang/p4factory/cli/pd_thrift_client.py", line 77, in setup
    self._transport.open()
  File "/usr/local/lib/python2.7/dist-packages/thrift/transport/TTransport.py", line 153, in open
    return self.__trans.open()
  File "/usr/local/lib/python2.7/dist-packages/thrift/transport/TSocket.py", line 113, in open
    raise TTransportException(TTransportException.NOT_OPEN, msg)
thrift.transport.TTransport.TTransportException: Could not connect to any of [('::1', 22222, 0, 0), ('127.0.0.1', 22222)]
No handlers could be found for logger "thrift.transport.TSocket"
```

Figure 11. Error while adding table entries



```
Table: send_frame_bytes row#: 0 counter value:
SimpleRouter:
~/demo/mininet-demo$ ./switch-control.sh <count
Control utility to manipulate tables for the Si
ter program.
SimpleRouter: Getting counter
Table: send_frame_bytes row#: 0 counter value:
SimpleRouter:
~/demo/mininet-demo$ ./switch-control.sh <diver
Control utility to manipulate tables for the Si
ter program.
SimpleRouter: Diverting 10.0.1.10 -> 10.0.0.10
```

Figure 12. Diverting traffic result

We were not able to rectify this error. So, we are adding the replica of the result that was achieved by the cisco network, which would resemble our result, provided if were successful in removing the above-mentioned error.

In this result, you can see that we are able to divert the traffic that was destined to 10.0.1.10 to 10.0.0.10. That is, we are sending the packet from h1 to h1 only.

V. CONCLUSION

Through this project, we got to know learnt about the P4 programming concepts. We learnt about the headers, parsers, and the "top-down" approach which involves control plane and the data plane. We were able to come up with a functionality wherein we were able to divert the traffic destined to a host to the desired host of our choice.

References

[1] Webite: https://rickmur.com/what-is-p4/

[2] Website: https://blogs.vmware.com/research/2017/04/07/programming-networks-p4/

[3] Website: https://www.barefootnetworks.com