P4 Traffic Based Routing

ECEN 5273

April 14th, 2020

Kristina Brunsgaard

Matt Menten

INTRODUCTION

Software Defined Networking is a relatively new concept, one that appeared within the past two decades. It's an approach to network management that allows for programmable, dynamic network configuration. Originally, basic ethernet/IP was straightforward and easy to manage, but as more devices are added to the network and new control requirements are implemented, complexity has increased. [1]

Because independent companies build their own network devices that maintain their own routing protocols, it's become especially difficult to create new protocols and ensure standard packet processing across all switches. Thus, The P4 Language Consortium developed an open-source programming language called P4 to help solve this problem. P4 controls packet forwarding in networking devices by parsing packets into fields and then matching on these fields to select actions. It's target independent, meaning it can be compiled for a variety of different switches. It's also protocol independent such that the language has no native support for generic protocols. The programmer instead develops headers and parses the packet into these fields as desired. In this way, any type of protocol can be supported, including proprietary protocols. Finally, P4 is reconfigurable, and can change how it processes packets after deployment, without interrupting the flow of packets.

In an effort to better understand the P4 programming language, we are implementing two different methods of packet forwarding. Both of these methods focus on route optimization. One for specific applications/protocols and the second for specific hosts.

App Specific Routing

We will implement application-preferred routing that uses dedicated switches for certain application types. To start, we will implement this on the transport layer between udp and tcp. We will design a topology with different switch routes, a shorter one for tcp and a longer route for udp, allowing a tiered hierarchy depending on the protocol. Thus, we will load balance based on the type of packet rather than across the whole topology. If time permits, we would like to implement this on the application layer to select between http, ftp and other application protocols.

Host Tier Path Selection

Host tier path selection also focuses on exploiting the network topology to influence the paths traffic takes between hosts. Here, there is an optimal path between Host X and Host Y, as well as suboptimal (longer) paths. Hosts are separated into "tiers". A high-tier host will have its traffic routed along the optimal path, while a low-tier host's traffic will be routed suboptimally.

[1] https://cseweb.ucsd.edu/classes/fa16/cse291-g/applications/ln/SDN.pdf

The switches in the network are populated with forwarding rules that determine how IP traffic is routed. Each switch is configured by an SDN controller, which sets rules dictating how traffic should be forwarded for each host, based on its tier. The advantage of P4 and SDN in this scenario is that reconfiguration can happen on the fly. We demonstrate that hosts can be “promoted” to a higher tier during runtime. Promotion is visible by examining the latency and bandwidth of a host’s traffic, both before and after promotion.

Motivation

Our motive for exploring these solutions was primarily for personal learning. After reading the P4 paper, we wanted to experiment with the language, and the SDN environment in general. We saw the project as an opportunity to learn a new programming language and implement our own optimizations.

However, our work is also applicable beyond education. Specifically, for private networks, such as those in companies and universities, optimizing the paths taken by specific applications can help reduce congestion. For instance, high amounts of video streaming traffic could negatively impact the performance of users doing other tasks, like file transfers. Our solution could solve an issue like this by having dedicated paths for video streaming, which in turn would reduce congestion for other workloads. Of course, this is within the scope of a private network, so there could still be congestion at the edge connected to the public internet.

For host tiers, a similar benefit is achievable. Stock traders working in a brokerage could be given shorter paths through the company network, which would reduce latency-sensitive transactions. Researchers at a university could have their traffic routed optimally, since their work is of a higher priority than the average user. Our general design is applicable in any situation where certain applications, or certain users, wish to be given preference in their routes. These are just a few motivating examples.

Snapshot of Results

Upon implementing and evaluating our designs, we have determined that they are successful. We demonstrate that certain types of traffic can be routed differently than others, reducing latency for protocols with high priority. We also demonstrate that different hosts in a network can have their traffic routed differently, even when they are connected to the same switch. Similarly, high priority hosts achieve higher bandwidth and lower latency as opposed to their lower-tier counterparts.

RELATED WORK

M. Palesi, R. Holsmark, S. Kumar and V. Catania, "Application Specific Routing Algorithms for Networks on Chip," in IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 3, pp. 316-330, March 2009.

https://ieeexplore.ieee.org/document/4553701

This paper focuses around application-specific routing, but for networks on a chip. Instead of typical routing across physically separated hosts, networks on a chip are concerned with routing information between modules on a piece of hardware, usually a system on a chip (SoC). Even so, the paper is still applicable to our project because it focuses on optimizing routing for certain applications and exploits topology to do so.

Y. Liu, D. Niu and B. L, “Delay-Optimized Video Traffic Routing in Software-Defined Inter-Datacenter Networks,” IEEE 2016

https://sites.ualberta.ca/~dniu/Homepage/Publications_files/yliu-tmm16.pdf

This paper focuses on optimizing video traffic routing within datacenter networks. It is similar in scope to our project because it addresses route optimization within a private network using the control plane. The authors propose a delay-sensitive routing mechanism using SDN to “to explicitly differentiate path selection for different sessions according to their delay-sensitivities.”

Khan, Sahrish & Tanvir, Sadaf & Javaid, Nadeem. (2016). Routing Techniques in Software Defined Networks: A Survey.

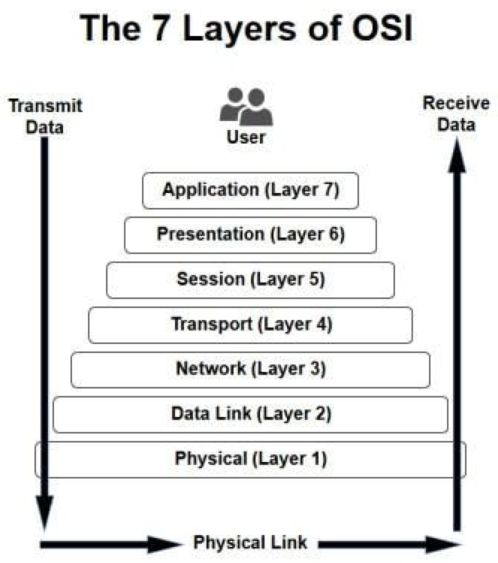
https://www.researchgate.net/publication/301757741_Routing_Techniques_in_Software_Defined_Networks_A_Survey

This paper is a general survey of routing techniques using software defined networking. The authors acknowledge that SDN concepts have created a new avenue for route selection, other than traditional BGP-based approaches within the dataplane. This is similar to our project, which focuses on route optimization for certain hosts or applications because it describes methods in which the control plane can assign its own routing rules. This paper is good background information on the general processes behind the solutions implemented in our project.

BACKGROUND

One essential component of packet forwarding is the actual composition of the header and how it must be interpreted to accurately receive the data. Packet headers contain data that is formatted to be used in a packet-switched network. This header contains control information that distinguishes the packet format as well as the payload that contains the user-desired data.

The seven-layer OSI model of networking can be found in Table 1 below. This model works in such a way that control is passed from a higher layer to a lower layer and then back up the hierarchy. The packets described above fall into the transport layer. Packet contents vary, however they typically contain at least a source and destination address, error detection and correction, hop limit, length, and the payload. Often, such as in the case of IP packets, the IP packet is the payload of an Ethernet frame that contains its own header and payload. Because we will exclusively be dealing with ethernet frames, we will continue to describe in detail these headers and disregard other packets such as the NASA Deep Space Network, MPEG packetized stream or NICAM.



[2]

The two packet header fields that are of most importance in this paper include the following...

Ethertype is a two-octet field that selects between the following protocols. Here are some common ones:
VLAN (0x9100): VLAN-tagged frame with double tagging
IPV4 (0x0800): Internet Protocol version 4
IPV6 (0x86DD): Internet Protocol Version 6
ARP (0x0806): Address Resolution Protocol
FCOE (0x8906): FC0E Initialization protocol

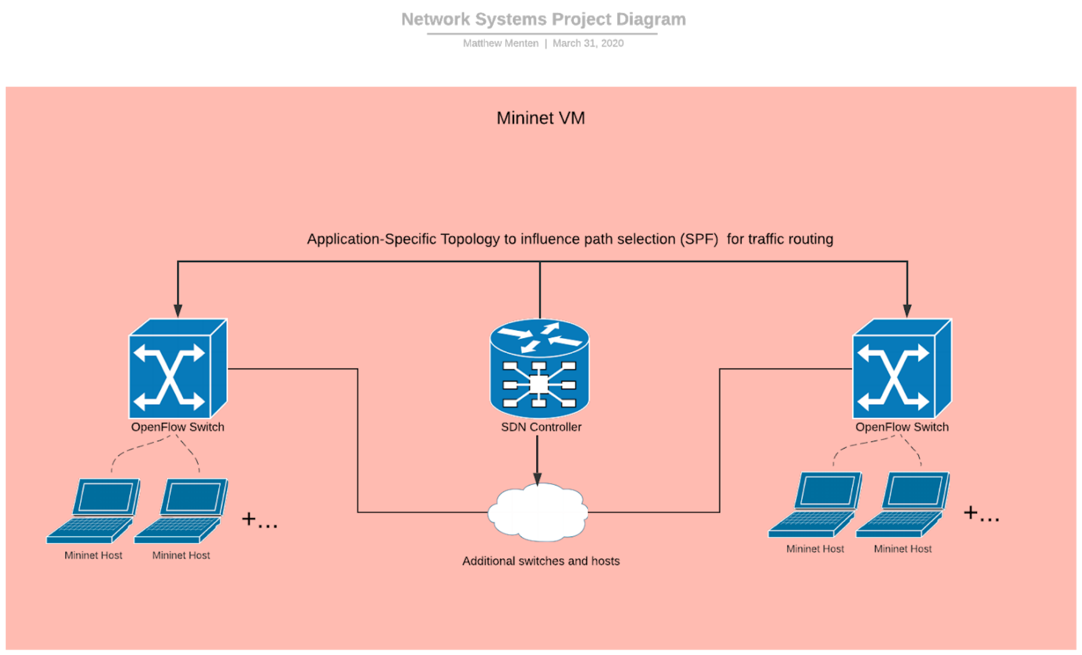
Internet protocol numbers are 8 bits wide and identify the next level protocol. Here are some common ones:
ICMP (0x1): Internet Control Message Protocol
TCP (0x6): Transmission Control Protocol
UDP (0x11): User Datagram Protocol
GRE (0x2F): Generic Routing Encapsulation
SCTP (0x84): Stream Control Transmission Protocol

[2] https://www.webopedia.com/quick_ref/OSI_Layers.asp

SYSTEM DESIGN

The topology and P4 routing was emulated in mininet, which is a lightweight network emulation environment that allows you to launch a virtual network with switches, hosts, and an SDN controller. The solution architecture is depicted in figure 1 below.

The components work together by first building the topology using a json file and commands file for each switch in mininet. Then, the P4 program is installed in the BMV2 software switch. This is a P4 software switch written in C++11 that takes a JSON file created from a P4 compiler and implements packet processing behavior.



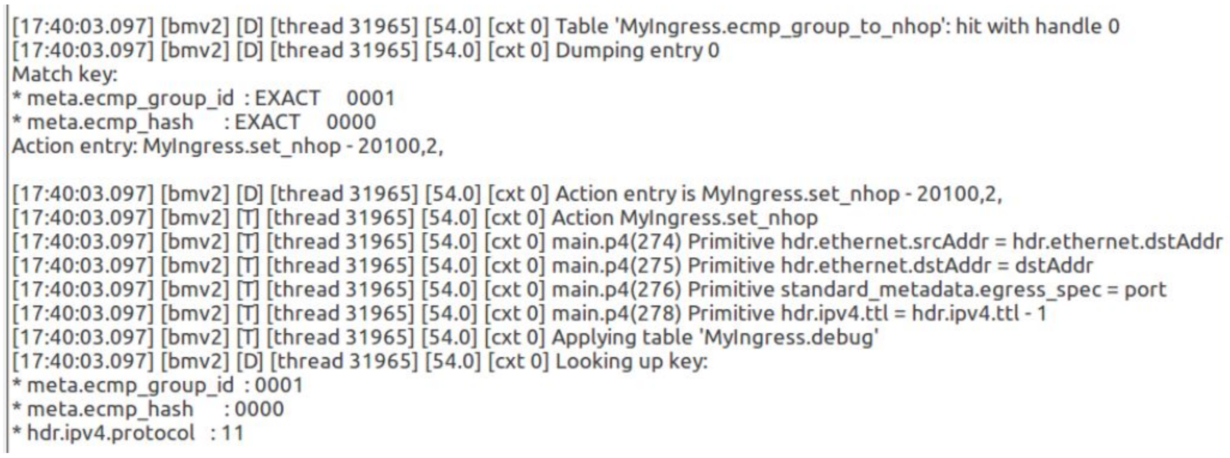
Network Systems Project Diagram
Matthew Menten | March 31, 2020
Mininet VM
Application-Specific Topology to influence path selection (SPF) for traffic routing
OpenFlow Switch
SDN Controller
OpenFlow Switch
Mininet Host
Mininet Host
+...
Additional switches and hosts
Mininet Host
Mininet Host
+...

Figure 1: Solution architecture

Originally, we planned in addition to implement a routing protocol designed for better bandwidth and congestion control by informing switches about bandwidth usage, speed caps, and additional statistics. We quickly realized this was too much to accomplish in the limited time we had and focused on two routing protocols, app specific routing and host tier selection.

Other problems we faced included with actually using the P4 compiler. There are a lot of tutorials available for P4, however they recently changed P4 in the last year to compile slightly differently. Because of this, compiling the tutorials using the wrong compiler resulted in errors. It took awhile to determine the cause of this.

Another problem we faced was figuring out how to debug P4 programs. Supposedly P4 includes a debugger, and we believed we had it enabled however it never displayed log statements. There was very little documentation on the debugger, so because of this we ended up instead creating a debug match and action table that matched on values we were interested in printing. Then, upon running mininet, a switch log file is created that traces the packet through P4. See Figure 2 below for an example of this.



```
[17:40:03.097] [bmv2] [D] [thread 31965] [54.0] [cxt 0] Table 'MyIngress.ecmp_group_to_nhop': hit with handle 0
[17:40:03.097] [bmv2] [D] [thread 31965] [54.0] [cxt 0] Dumping entry 0
Match key:
* meta.ecmp_group_id  : EXACT     0001
* meta.ecmp_hash      : EXACT     0000
Action entry: MyIngress.set_nhop - 20100,2,

[17:40:03.097] [bmv2] [D] [thread 31965] [54.0] [cxt 0] Action entry is MyIngress.set_nhop - 20100,2,
[17:40:03.097] [bmv2] [T] [thread 31965] [54.0] [cxt 0] Action MyIngress.set_nhop
[17:40:03.097] [bmv2] [T] [thread 31965] [54.0] [cxt 0] main.p4(274) Primitive hdr.ethernet.srcAddr = hdr.ethernet.dstAddr
[17:40:03.097] [bmv2] [T] [thread 31965] [54.0] [cxt 0] main.p4(275) Primitive hdr.ethernet.dstAddr = dstAddr
[17:40:03.097] [bmv2] [T] [thread 31965] [54.0] [cxt 0] main.p4(276) Primitive standard_metadata.egress_spec = port
[17:40:03.097] [bmv2] [T] [thread 31965] [54.0] [cxt 0] main.p4(278) Primitive hdr.ipv4.ttl = hdr.ipv4.ttl - 1
[17:40:03.097] [bmv2] [T] [thread 31965] [54.0] [cxt 0] Applying table 'MyIngress.debug'
[17:40:03.097] [bmv2] [D] [thread 31965] [54.0] [cxt 0] Looking up key:
* meta.ecmp_group_id  : 0001
* meta.ecmp_hash      : 0000
* hdr.ipv4.protocol   : 11
```

Figure 2: p4s.s1.log output

App Specific Routing

This protocol implementation is broken down into sub elements seen below to better describe the system design. Essentially, packets are sent using Send_mri.py and Receive_mri.py sniffs on the desired port for these packets. P4 actually processes the packets and is made up of a header, parser, ingress processing, and egress processing.

Send_mri.py

This python file is modified from a file provided by P4 Tutorials and can be found here. It takes three parameters, the destination ip address, the message to send, the type of packet (tcp or udp), and the number of packets to send. It uses a python packet called scapy to send packets. It also calls a function IPOption_MRI, that adds the value 31 following the destination address into the options field, to be parsed in the parsing function. The packets are sent from source port 1234 to destination port 4321.

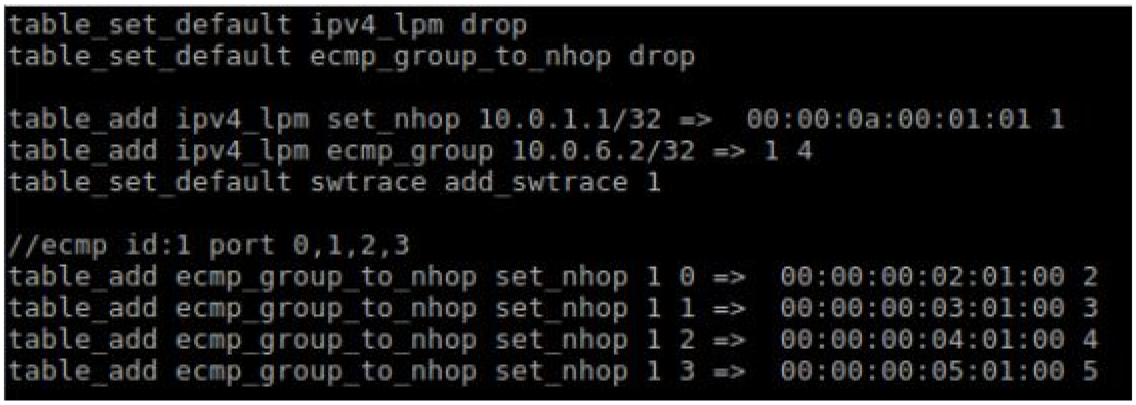
Receive_mri.py

This python file is again modified from a P4 Tutorials file found here. It sniffs for all packets on port 4321, which is the port where Send_mri.py sends it's packets. Pkt.show2, another scapy function, shows the packet contents, however on the assembled packet. This is relevant as the P4 deparser emits the packet contents, which include an mri header and swtrace.

Topology

Mininet is a virtual network that allows you to emulate a networking topology and implement routing protocols. The main topology file is passed as a json object and defines the hosts, switches, and links between the hosts and switches. Each host has its own configuration file where unique functions to the P4 configuration file are added.

For example, figure 3 shows the configuration of switch 1. The function add_swtrace adds the switch id 1 to this switch. Then it adds the routing table so switch 1 knows how to get to switch 2, 3, 4, and 5. The default values are set as drop packet in case the match action table fails to match on a value.



```
table_set_default ipv4_lpm drop
table_set_default ecmp_group_to_nhop drop

table_add ipv4_lpm set_nhop 10.0.1.1/32 =>  00:00:0a:00:01:01 1
table_add ipv4_lpm ecmp_group 10.0.6.2/32 => 1 4
table_set_default swtrace add_swtrace 1

//ecmp id:1 port 0,1,2,3
table_add ecmp_group_to_nhop set_nhop 1 0 =>  00:00:00:02:01:00 2
table_add ecmp_group_to_nhop set_nhop 1 1 =>  00:00:00:03:01:00 3
table_add ecmp_group_to_nhop set_nhop 1 2 =>  00:00:00:04:01:00 4
table_add ecmp_group_to_nhop set_nhop 1 3 =>  00:00:00:05:01:00 5
```

Figure 3: s1-commands.txt

The topology of the switches for the optimal submission are configured as in Figure 4 below. This topology was selected as a demonstration of app specific routing. The intention is to assign certain switches as dedicated to certain application packets. Below, tcp packets take the route from h1 to sw1 to sw2 to sw7 to sw8 to sw9 to sw6 to h2. Udp packets take a much shorter route, from h1 to s1 to s3 to s6 to h2. In this way the packets are tiered by priority level. S4 and S5 were configured as part of the topology in case, if given enough time, I could include additional packet types.

I had difficulty implementing this topology and ultimately stuck with the original topology that doesn't include s2, s8, s9. It still achieves the same application specific routing goal, however doesn't apply a shorter route to one over the other.

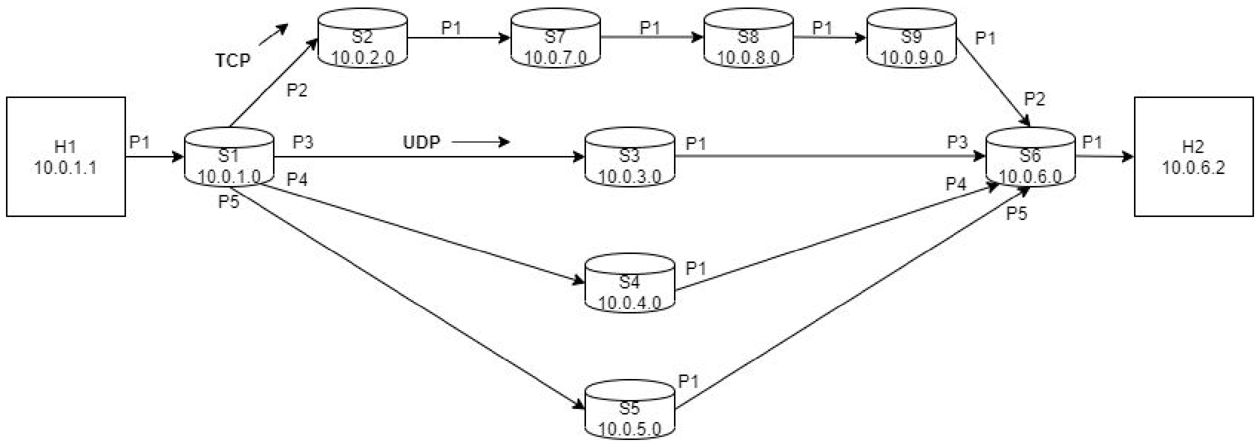
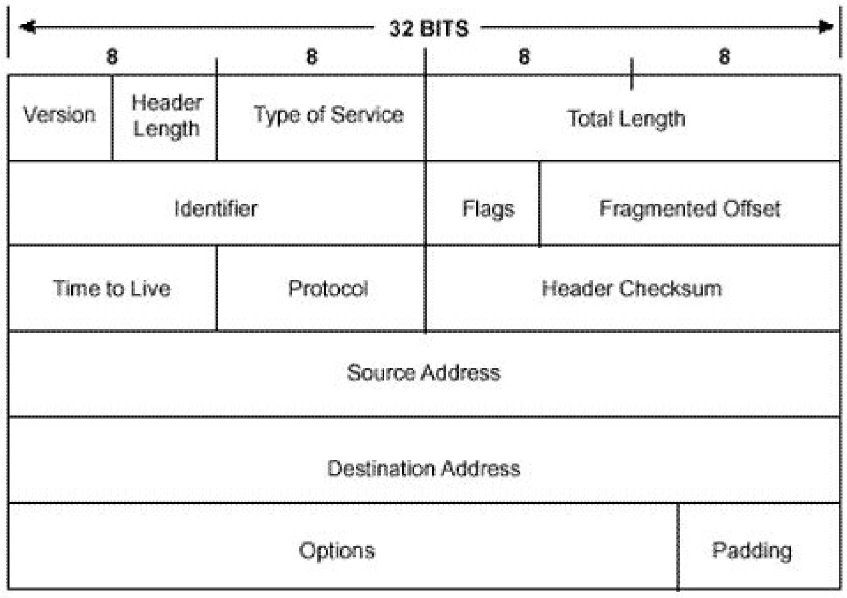


Figure 4: Topology configuration for P4

Header

In order to eventually parse the header for an Ethernet frame, we must first use the P4 language to define how the headers are to be extracted. As described in the background section, headers describe how the data is formatted. Figure 5 shows the fields within an IP packet header.



32 BITS

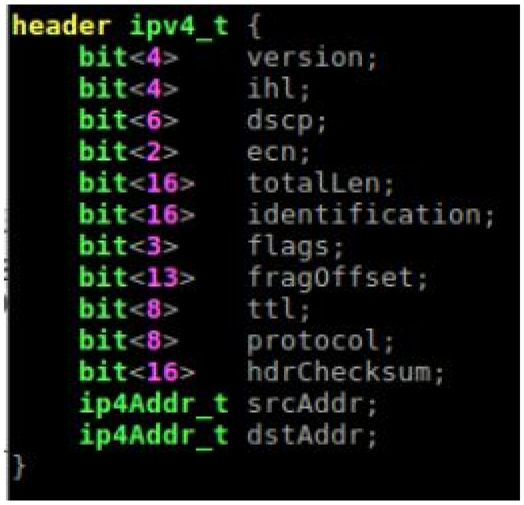
8	8	8	8
Version / Header Length	Type of Service	Total Length	
Identifier		Flags	Fragmented Offset
Time to Live	Protocol	Header Checksum	
Source Address			
Destination Address			
Options			Padding

[3]

Figure 5: IP Packet Header

For our purposes, we need headers to hold ethernet, ipv4, tcp, and udp. An example of defining a header in P4 can be found in figure 6.

[3] https://flylib.com/books/en/2.298.1.25/1/



```
header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<6>    dscp;
    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;
}
```

Figure 6: Ipv4 Header

We also define custom headers called ip4_option, mri, swtraces to hold specialty values within our P4 program. These will be discussed in further detail in the following sections.

Parser

The parser acts as a state machine to follow the designated path depending on what kind of packet comes in. The state transitions are selected by extracting header information and then using transition select to match that header and select the appropriate next state.

This parser is standard with extracting ethernet and ipv4 packet information, however also includes some specialty parsing. Some of this functionality comes from a P4 Tutorial found here. Specifically, Parse_ipv4_option looks to see if the packet contains an mri (Multi Route Hop Inspection) header. The mri value is 31 and is inserted into the packet options field prior to being sent. If this header exists, we go to parse_swtrace which continues to call itself until the value “remaining” is 0, indicating we’ve parsed all the switch ids. Recall that these ids get instantiated in the topology, with switch 1 having id 1, switch 2 id 2 and so on.

Ingress Processing

The Ingress processing occurs on packets coming into the switch and whose destination is within the host network. This is where the packets get routed to the next hop and ultimately it’s final destination.

This application specific routing protocol resembles load balancing somewhat. Load balancing often uses equal cost multipath forwarding to select a different route for each packet, thus avoiding congestion. This is done by hashing values specific to that packet such as addresses, ports, or its protocol and taking the modulus of the number of paths to select between them. Although similar to what we want, this doesn’t guarantee a dedicated path for each packet and can cause collisions. Therefore, we instead “hardcoded’ assigned values to each path. If our packet is TCP, we assign ecmp_hash to the value 3. If the packet is UDP we assign ecmp_hash to the value 0. This sends that packet down either the tcp or udp path as described in the topology.

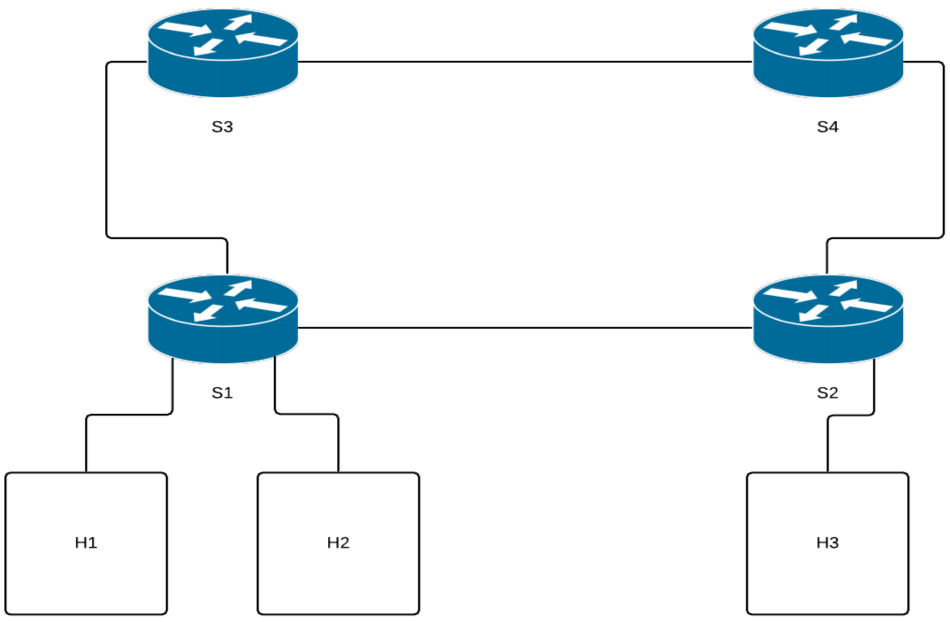
Egress Processing

Egress processing is the opposite of ingress processing and occurs on packets going out of the switch, directed toward an external network. We utilize egress processing here to attach the switch id the packet was directed through in order to print out the final path of the packet. This will ultimately be used for validation, to ensure tcp packets follow the tcp path and udp packets follow the udp path. Code-wise, we simply match on the function add_swtrace. This function increases the queue by 1, to show the congestion of packets. Push_front moves the previous id over one to make space for the current switch and count. We add the next switch and count to the header, add two to the internet header length as we added queue and switch values, increase option length, and finally increase the total length. Now, when we implement the deparser we also emit the swtraces header containing the route.

Host Tier Path Selection

For Host Tier Path Selection, a different topology was used. As before, the subsections below explain each part of the solution in detail. The main difference with this solution is that it focuses more on the control plane and dynamically configuring switches during runtime.

Topology



Establishing Tiers

As seen in the topology above, switch 1 is connected to hosts 1 and 2. Switch 2 is connected to host 3. To make things simple, testing was done by sending traffic from either H1 or H2 to H3. Routing differences were observed by comparing the paths taken by H1 and H2's traffic, since they are connected to the same edge switch. The optimal path from H1/H2 to H3 is via S1 and S2. Any paths involving S3 and S4 are suboptimal.

We created two separate tiers of hosts, one high-tier and one low-tier. In all implementations, H1 and H3 are high-tier hosts, while H2 is low-tier. Tiers are established by populating each switch's forwarding table with specific rules. Each switch has a single table, which is keyed on the source-destination address pair. For example, on S1, a rule states that traffic originating from H1 destined for H3 will be sent to S2. A separate rule on S1 states that traffic originating from H2 destined for H3 will be sent to S3, since H2 is a low-tier host.

Static Switch Configuration

In our repository, there are two folders under the HostTierPathSelection directory, one for static switch configurations, and one for dynamic. The static switch configuration was mostly used for testing and proof-of-concept. Here, we simply established our topology and the switch rules that forward traffic differently for H2 vs. H1. As mentioned briefly before, all traffic from H1 to H3 takes the following path: H1 -> S1 -> S2 -> H3. Since H2 is a low-tier host it takes a longer path to H3, namely: H2 -> S1 -> S3 -> S4 -> S2 -> H3.

The file 'basic.p4' was created based on a basic connectivity example from the P4 tutorials. The ingress processing table was modified to match on both source and destination addresses. The files sX-runtime.json define the forwarding rules for each switch and are gathered by the SDN controller to populate the switch tables on initialization.

Dynamic Switch Configuration

After implementing host tiers using a static configuration of the topology and switch rules, we knew that our design worked. We then modified the controller to allow for host promotion. To accomplish this, we initialize the SDN controller with multiple topologies. A topology defines the hosts and switches in the network, the links between them, and maps switches to their corresponding files that contain the table rules.

To achieve dynamic reconfiguration of the switch rules, we define multiple topologies with the same hosts, switches and links, but with different mappings to the switch rule files. When the network is initially created, the switch rules define H2 to be a low-tier host. During runtime, a user may issue a command to promote H2. In this case, the SDN controller will reconfigure the switches to give H2 optimal routes using a separate topology file.

The creation of separate topologies, and corresponding switch rules, which define the tiers of each host can be automated. For example, a single JSON or YAML file could define the static information, like hosts, switches and links. A script could parse this file, and be provided with options regarding the tier status of each host. It would dynamically create the switch rules based on the host tier options provided. We leave this automation as future work.

EVALUATION/FINDINGS

App Specific Routing

This is a new routing protocol implementation, rather than an attempt at performance improvement on an existing protocol. Therefore, our evaluation comes down to whether or not the implementation works and how to validate it.

To set up the following test bed, download this VM link and Import the virtual machine into VirtualBox. Do this by opening VirtualBox, select “File > Import Appliance”, and navigate to the downloaded file. Then pull the App Specific Routing repository into the vm. Type sudo make to build the topology and start mininet. Then, open a terminal for host 1 and host by typing xterm h1 h2.

To send one packet to host 2 type the following into host 1:

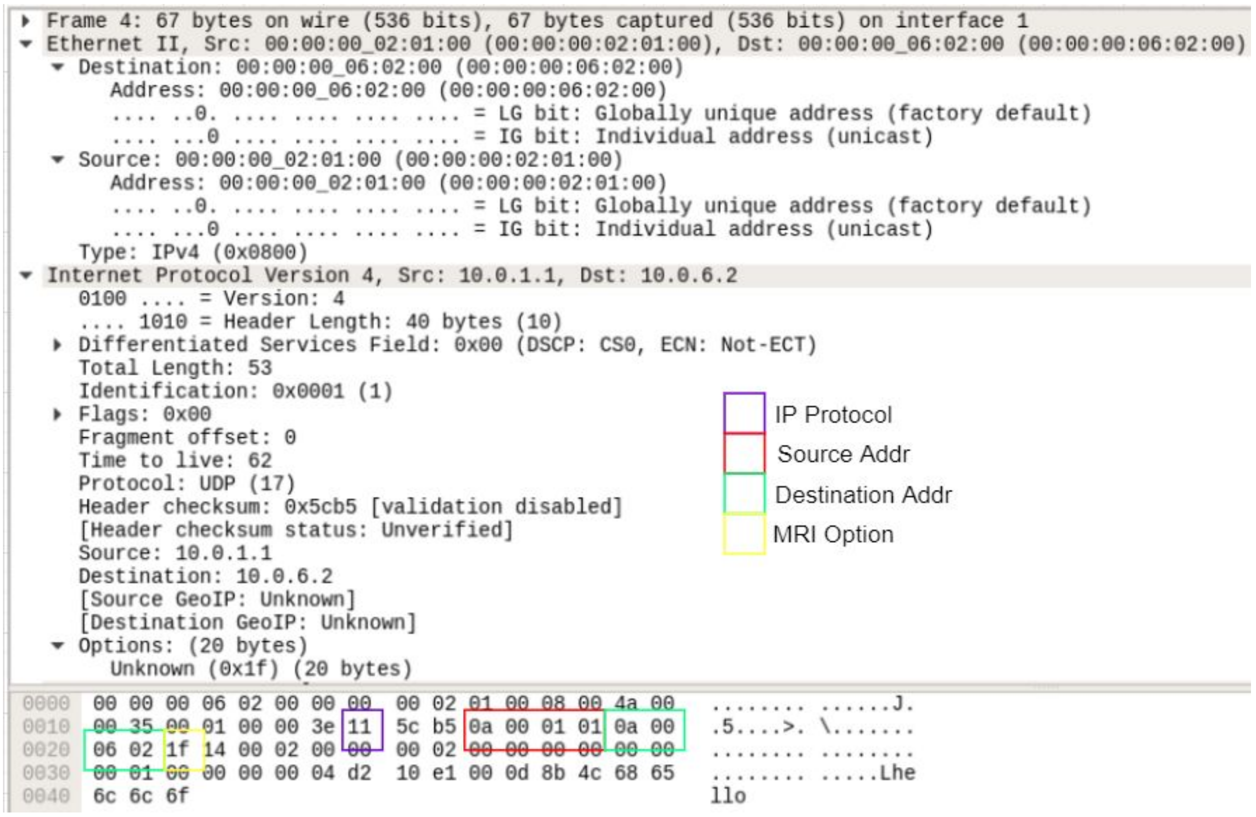
```
./send_mri.py 10.0.6.2 “hello” tcp 1
```

Note that tcp can be replaced with udp to send udp packets.

To receive packets type the following into host 2:

```
./receive_mri.py
```

To verify we receive the packets we expect, we use send_mri to send a tcp packet with the payload “hello” as in figure 8, and a udp packet with the payload “hello” as in figure 7.

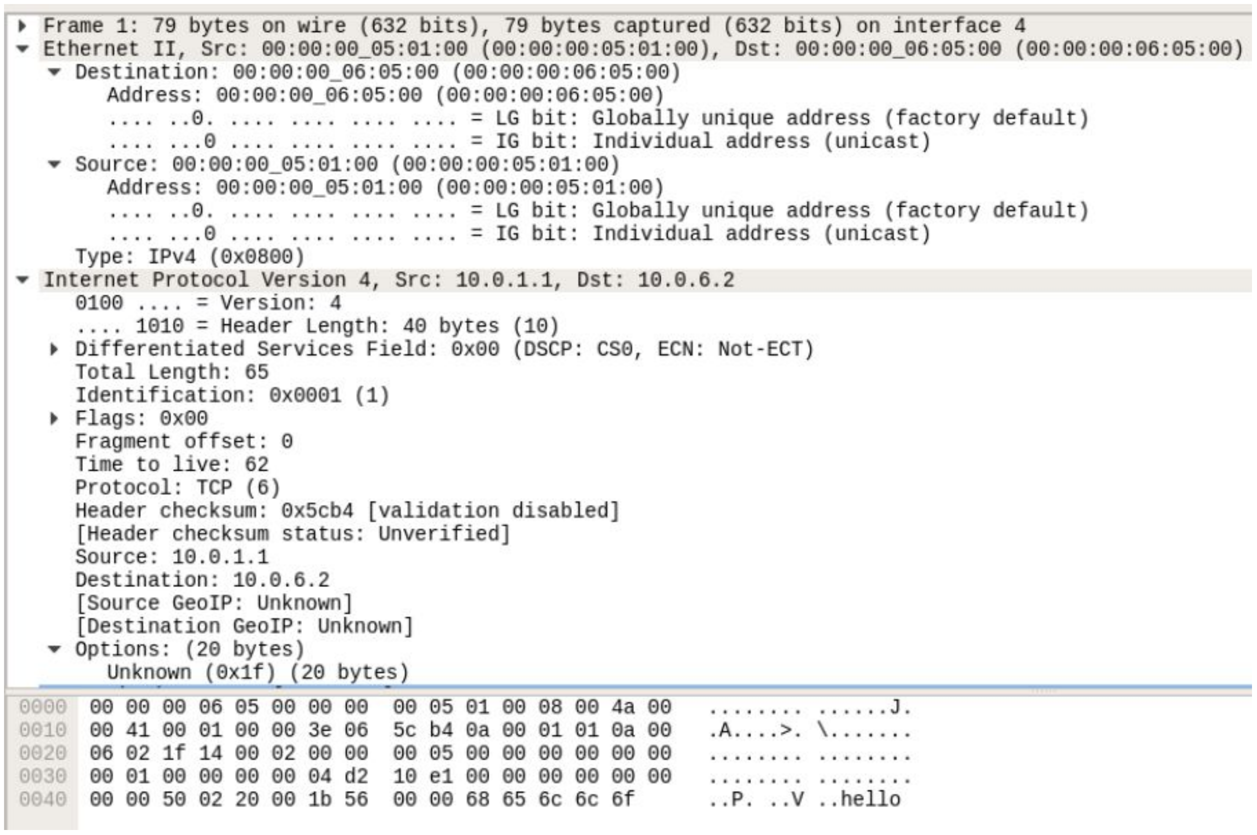


```
▸ Frame 4: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface 1
▾ Ethernet II, Src: 00:00:00_02:01:00 (00:00:00:02:01:00), Dst: 00:00:00_06:02:00 (00:00:00:06:02:00)
  ▾ Destination: 00:00:00_06:02:00 (00:00:00:06:02:00)
      Address: 00:00:00_06:02:00 (00:00:00:06:02:00)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  ▾ Source: 00:00:00_02:01:00 (00:00:00:02:01:00)
      Address: 00:00:00_02:01:00 (00:00:00:02:01:00)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IPv4 (0x0800)
▾ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.6.2
    0100 .... = Version: 4
    .... 1010 = Header Length: 40 bytes (10)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 53
    Identification: 0x0001 (1)
  ▸ Flags: 0x00
    Fragment offset: 0
    Time to live: 62
    Protocol: UDP (17)
    Header checksum: 0x5cb5 [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.0.1.1
    Destination: 10.0.6.2
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
  ▾ Options: (20 bytes)
      Unknown (0x1f) (20 bytes)

0000  00 00 00 06 02 00 00 00  00 02 01 00 08 00 4a 00   ........ ......J.
0010  00 35 00 01 00 00 3e 11  5c b5 0a 00 01 01 0a 00   .5....>. \.......
0020  06 02 1f 14 00 02 00 00  00 02 00 00 00 00 00 00   ........ ........
0030  00 01 00 00 00 00 04 d2  10 e1 00 0d 8b 4c 68 65   ........ .....Lhe
0040  6c 6c 6f                                           llo
```

Figure 7: Wireshark capture of a UDP packet send with the payload “hello” sniffing on sw6-eth1

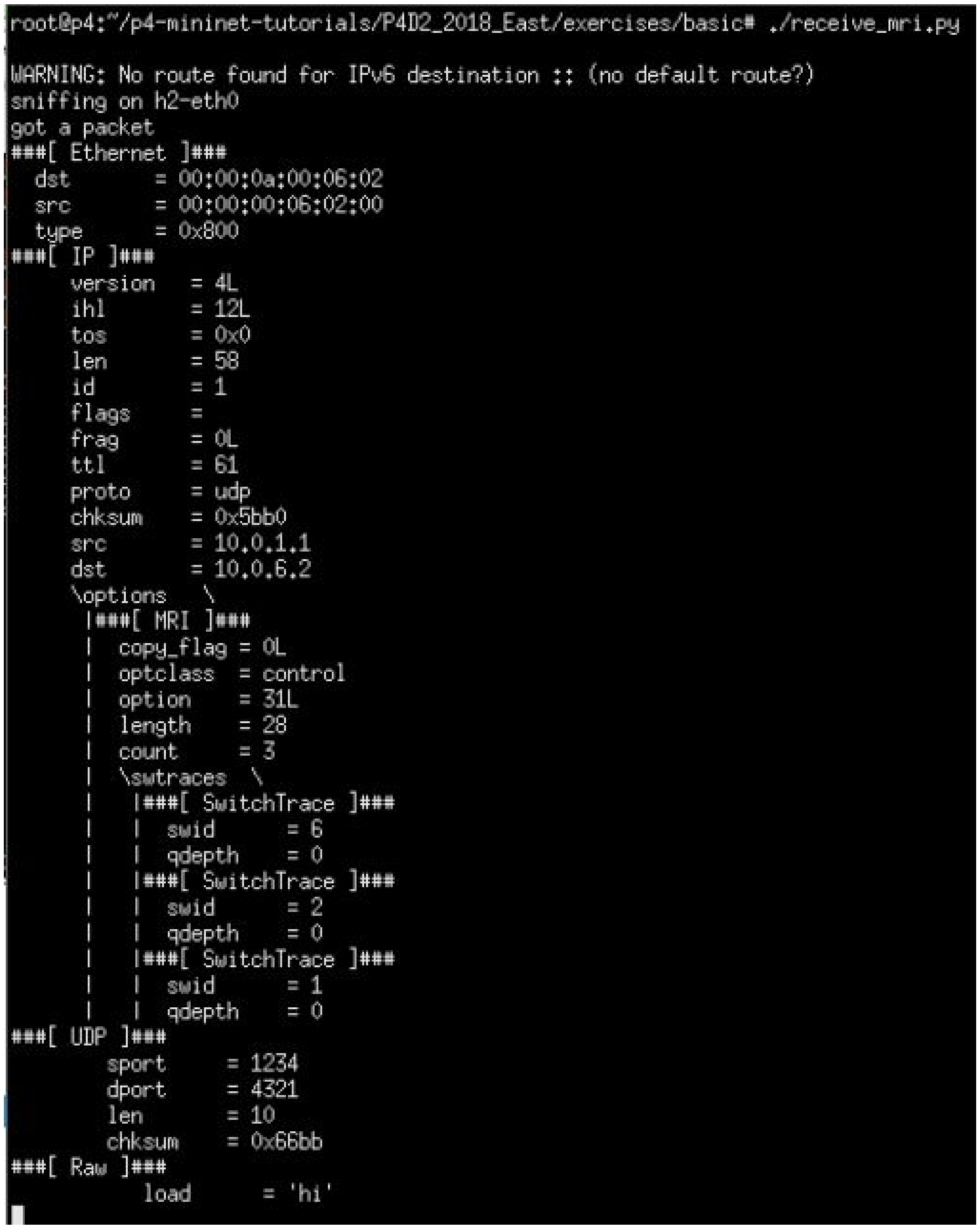
Analyzing the udp packet, we see the destination is 10.0.6.2 as expected, and the source is 10.0.1.1 as expected. Note that in this implementation the source isn’t updated as packets are passed and stays with the original host source. The IP protocol is 11, indicating it is indeed a udp packet. The value 1f in the options header is the mri value indicating we’re sending an mri packet that contains swtraces. Finally, we see the payload is at the very end. Looking at the TCP packet, we see almost the same as in the UDP however the ip protocol is 6 instead.



```
▸ Frame 1: 79 bytes on wire (632 bits), 79 bytes captured (632 bits) on interface 4
▾ Ethernet II, Src: 00:00:00_05:01:00 (00:00:00:05:01:00), Dst: 00:00:00_06:05:00 (00:00:00:06:05:00)
  ▾ Destination: 00:00:00_06:05:00 (00:00:00:06:05:00)
      Address: 00:00:00_06:05:00 (00:00:00:06:05:00)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  ▾ Source: 00:00:00_05:01:00 (00:00:00:05:01:00)
      Address: 00:00:00_05:01:00 (00:00:00:05:01:00)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IPv4 (0x0800)
▾ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.6.2
    0100 .... = Version: 4
    .... 1010 = Header Length: 40 bytes (10)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 65
    Identification: 0x0001 (1)
  ▸ Flags: 0x00
    Fragment offset: 0
    Time to live: 62
    Protocol: TCP (6)
    Header checksum: 0x5cb4 [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.0.1.1
    Destination: 10.0.6.2
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
  ▾ Options: (20 bytes)
      Unknown (0x1f) (20 bytes)
0000  00 00 00 06 05 00 00 00  00 05 01 00 08 00 4a 00   ........ ......J.
0010  00 41 00 01 00 00 3e 06  5c b4 0a 00 01 01 0a 00   .A....>. \.......
0020  06 02 1f 14 00 02 00 00  00 05 00 00 00 00 00 00   ........ ........
0030  00 01 00 00 00 00 04 d2  10 e1 00 00 00 00 00 00   ........ ........
0040  00 00 50 02 20 00 1b 56  00 00 68 65 6c 6c 6f      ..P. ..V ..hello
```

Figure 8: Wireshark capture of a TCP packet send with the payload "hello" sniffing on sw6-eth1

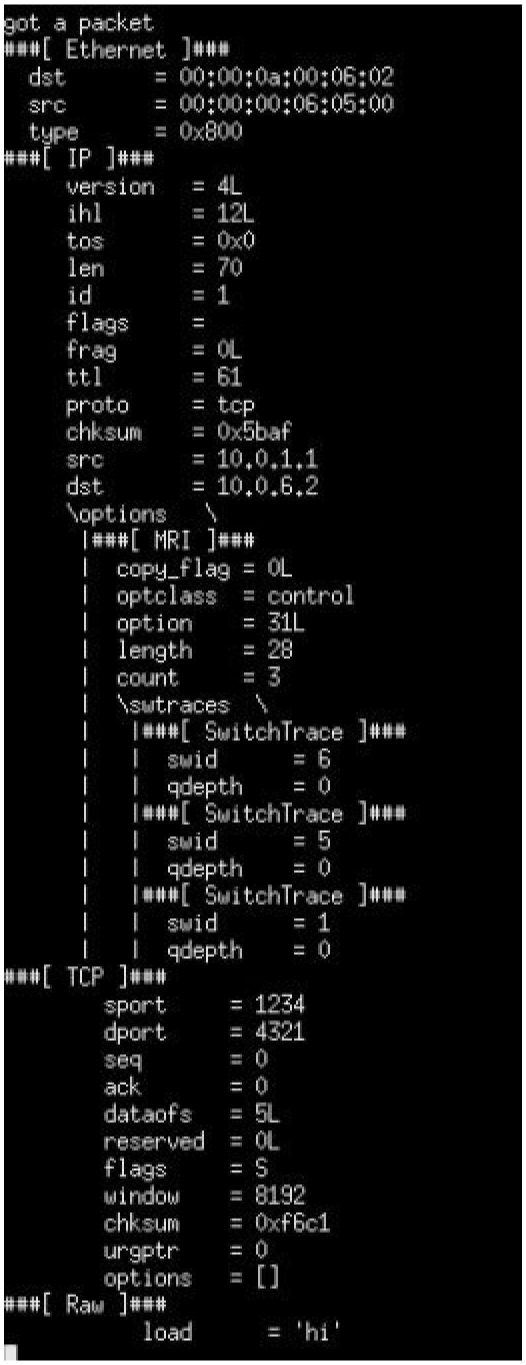
Now, by sending a packet and receiving it using receive_mri we can see the contents from scapy's show function which shows the developed view of the packet. It now contains the mri and switch trace output we added in the egress processing. In figure 7, the trace path is sw1 to sw2 to sw6 for a UDP packet. In figure 8, the trace path is sw1 to sw5 to sw6 for a TCP packet. Thus, the path's are selected as desired, with switches dedicated to certain ip protocols. Thus, we have validated the P4 program does what we expected it to do. Note that this is a baseline match action table and could be built on to match on layer 4 protocols instead.



```
root@p4:~/p4-mininet-tutorials/P4D2_2018_East/exercises/basic# ./receive_mri.py

WARNING: No route found for IPv6 destination :: (no default route?)
sniffing on h2-eth0
got a packet
###[ Ethernet ]###
  dst       = 00:00:0a:00:06:02
  src       = 00:00:00:06:02:00
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 12L
     tos       = 0x0
     len       = 58
     id        = 1
     flags     =
     frag      = 0L
     ttl       = 61
     proto     = udp
     chksum    = 0x5bb0
     src       = 10.0.1.1
     dst       = 10.0.6.2
     \options   \
      |###[ MRI ]###
      |  copy_flag = 0L
      |  optclass  = control
      |  option    = 31L
      |  length    = 28
      |  count     = 3
      |  \swtraces  \
      |   |###[ SwitchTrace ]###
      |   |  swid      = 6
      |   |  qdepth    = 0
      |   |###[ SwitchTrace ]###
      |   |  swid      = 2
      |   |  qdepth    = 0
      |   |###[ SwitchTrace ]###
      |   |  swid      = 1
      |   |  qdepth    = 0
###[ UDP ]###
        sport     = 1234
        dport     = 4321
        len       = 10
        chksum    = 0x66bb
###[ Raw ]###
           load      = 'hi'
```

Figure 9: receive_mri.py output of udp packet with payload of “hi”



```
got a packet
###[ Ethernet ]###
  dst       = 00:00:0a:00:06:02
  src       = 00:00:00:06:05:00
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 12L
     tos       = 0x0
     len       = 70
     id        = 1
     flags     =
     frag      = 0L
     ttl       = 61
     proto     = tcp
     chksum    = 0x5baf
     src       = 10.0.1.1
     dst       = 10.0.6.2
     \options   \
      |###[ MRI ]###
      |  copy_flag = 0L
      |  optclass  = control
      |  option    = 31L
      |  length    = 28
      |  count     = 3
      |  \swtraces  \
      |   |###[ SwitchTrace ]###
      |   |  swid      = 6
      |   |  qdepth    = 0
      |   |###[ SwitchTrace ]###
      |   |  swid      = 5
      |   |  qdepth    = 0
      |   |###[ SwitchTrace ]###
      |   |  swid      = 1
      |   |  qdepth    = 0
###[ TCP ]###
        sport     = 1234
        dport     = 4321
        seq       = 0
        ack       = 0
        dataofs   = 5L
        reserved  = 0L
        flags     = S
        window    = 8192
        chksum    = 0xf6c1
        urgptr    = 0
        options   = []
###[ Raw ]###
           load      = 'hi'
```

Figure 10: receive_mri.py output of tcp packet with payload of “hi”

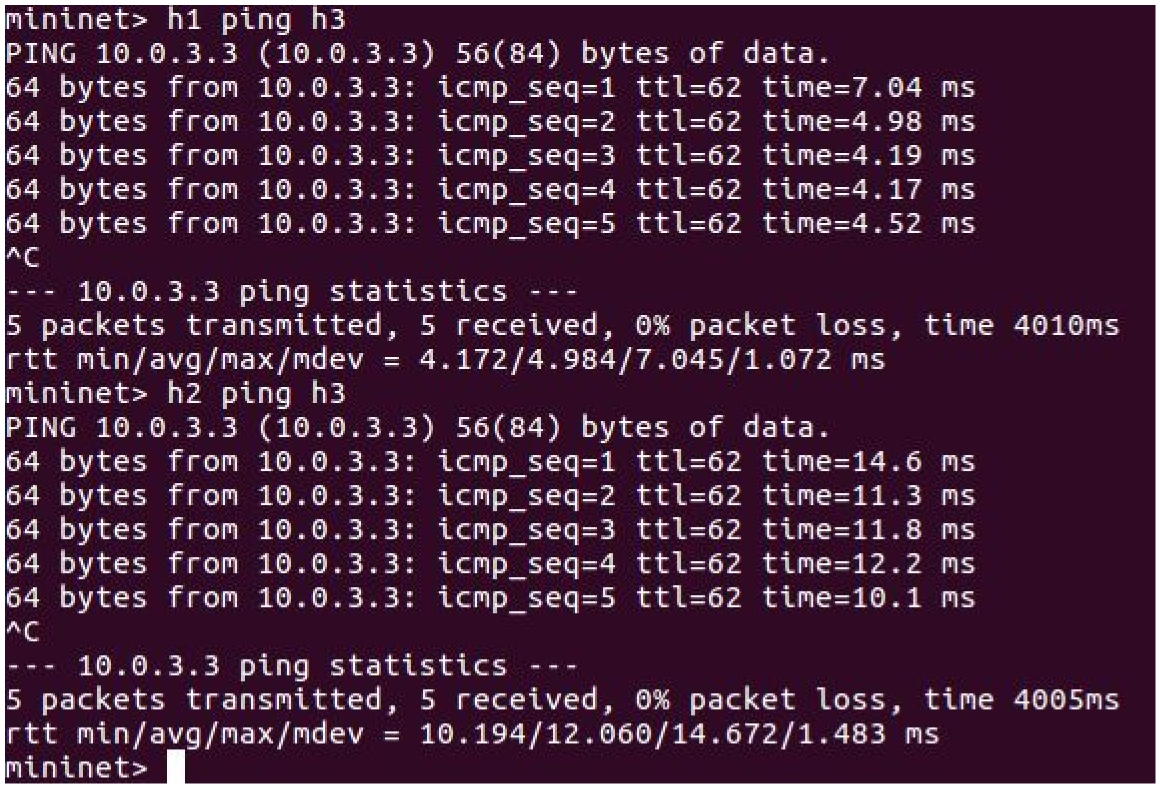
Host Tier Path Selection

Again, this is a new type of routing, as opposed to an optimization of existing routing, like BGP. The evaluation was done by comparing the paths taken by high vs. low tier hosts. Additionally, we added latency to the sub-optimal paths in order to highlight the difference between the host tiers.

Before we could compare the difference in performance, we had to make sure that each host's traffic was taking the intended route based on its tier. As mentioned and pictured in the System Design section, we verified routes by analyzing the log files for each switch. In the logs, we were able to see the source and destination address of each packet, as well as the switch port that it was ultimately forwarded out on. We used this to verify that high-tier hosts were routed along the optimal path and low-tier hosts were not.

Afterwards, we used both ping and iperf to measure the differences between high and low-tier hosts in terms of latency and achievable bandwidth. Again, sending traffic from H1 to H3 was our control/baseline, while sending traffic from H2 to H3 illustrates the sub-optimal routing given to low-tier hosts.

H1 to H3 Ping, H2 (low tier) to H3 Ping

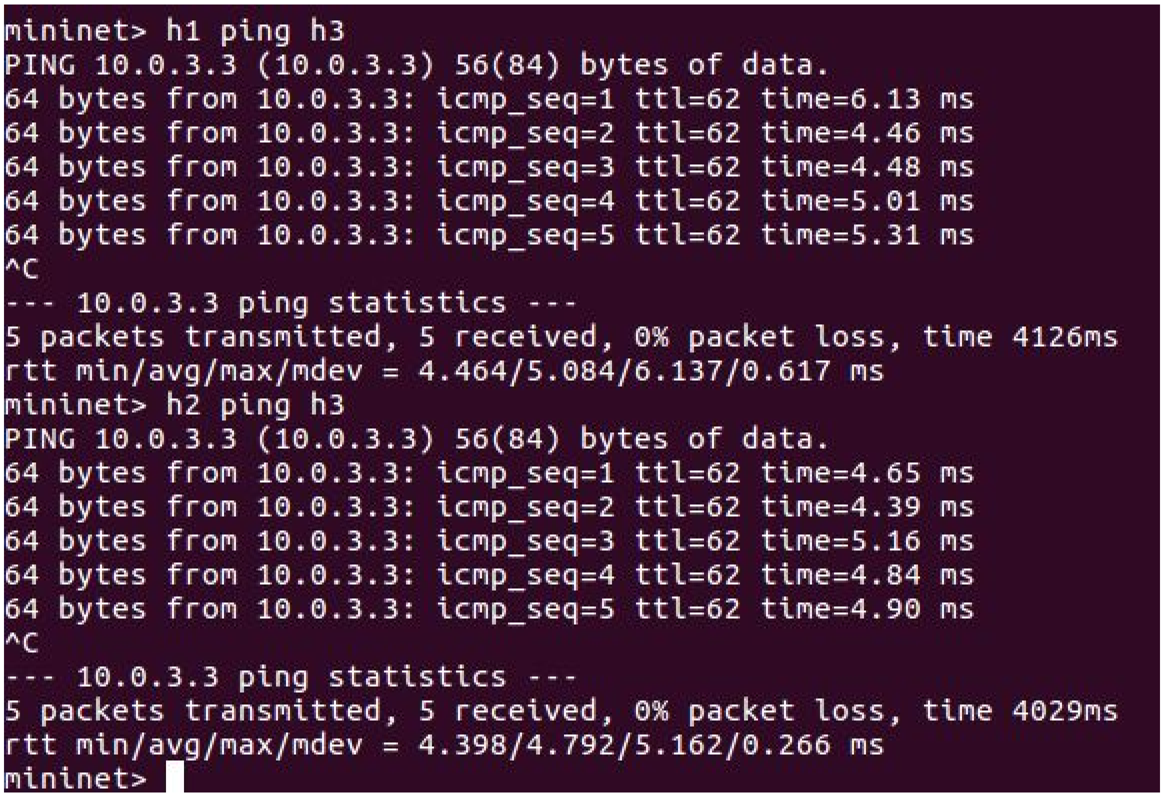


```
mininet> h1 ping h3
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=62 time=7.04 ms
64 bytes from 10.0.3.3: icmp_seq=2 ttl=62 time=4.98 ms
64 bytes from 10.0.3.3: icmp_seq=3 ttl=62 time=4.19 ms
64 bytes from 10.0.3.3: icmp_seq=4 ttl=62 time=4.17 ms
64 bytes from 10.0.3.3: icmp_seq=5 ttl=62 time=4.52 ms
^C
--- 10.0.3.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4010ms
rtt min/avg/max/mdev = 4.172/4.984/7.045/1.072 ms
mininet> h2 ping h3
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=62 time=14.6 ms
64 bytes from 10.0.3.3: icmp_seq=2 ttl=62 time=11.3 ms
64 bytes from 10.0.3.3: icmp_seq=3 ttl=62 time=11.8 ms
64 bytes from 10.0.3.3: icmp_seq=4 ttl=62 time=12.2 ms
64 bytes from 10.0.3.3: icmp_seq=5 ttl=62 time=10.1 ms
^C
--- 10.0.3.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 10.194/12.060/14.672/1.483 ms
mininet> 
```

Here, we can see that latency between H1 and H3 is significantly lower than between H2 and H3. The average RTT for H1-H3 is 4.98 ms, while the average RTT for H2-H3 is 12.06 ms, which is 2.4x longer.

H1 to H3 Ping, H2 (Promoted) to H3 Ping

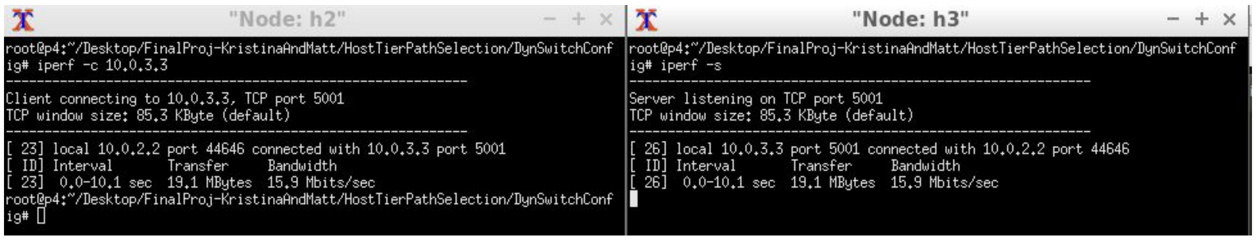
On the following page.



```
mininet> h1 ping h3
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=62 time=6.13 ms
64 bytes from 10.0.3.3: icmp_seq=2 ttl=62 time=4.46 ms
64 bytes from 10.0.3.3: icmp_seq=3 ttl=62 time=4.48 ms
64 bytes from 10.0.3.3: icmp_seq=4 ttl=62 time=5.01 ms
64 bytes from 10.0.3.3: icmp_seq=5 ttl=62 time=5.31 ms
^C
--- 10.0.3.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4126ms
rtt min/avg/max/mdev = 4.464/5.084/6.137/0.617 ms
mininet> h2 ping h3
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=62 time=4.65 ms
64 bytes from 10.0.3.3: icmp_seq=2 ttl=62 time=4.39 ms
64 bytes from 10.0.3.3: icmp_seq=3 ttl=62 time=5.16 ms
64 bytes from 10.0.3.3: icmp_seq=4 ttl=62 time=4.84 ms
64 bytes from 10.0.3.3: icmp_seq=5 ttl=62 time=4.90 ms
^C
--- 10.0.3.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4029ms
rtt min/avg/max/mdev = 4.398/4.792/5.162/0.266 ms
mininet>
```

Once H2 has been promoted to a high-tier host, we see that the latency differences are minimal. In fact, in this test H2 actually had a lower average RTT, although the difference is not significant.

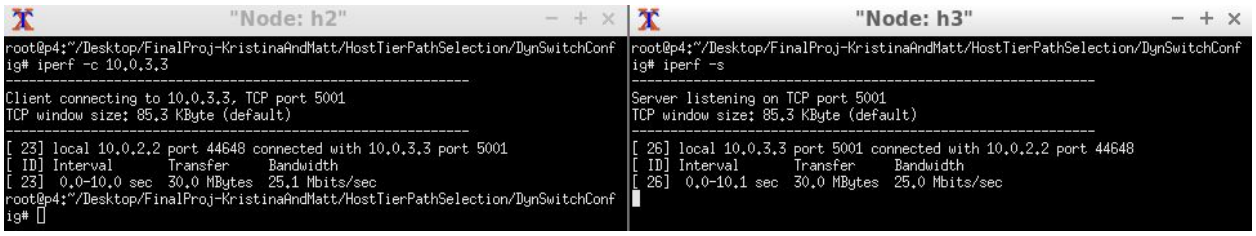
Iperf H2 (low-tier) to H3



```
root@p4:~/Desktop/FinalProj-KristinaAndMatt/HostTierPathSelection/DynSwitchConf
ig# iperf -c 10.0.3.3
------------------------------------------------------------
Client connecting to 10.0.3.3, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 23] local 10.0.2.2 port 44646 connected with 10.0.3.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 23]  0.0-10.1 sec  19.1 MBytes  15.9 Mbits/sec
root@p4:~/Desktop/FinalProj-KristinaAndMatt/HostTierPathSelection/DynSwitchConf
ig# 
```

```
root@p4:~/Desktop/FinalProj-KristinaAndMatt/HostTierPathSelection/DynSwitchConf
ig# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 26] local 10.0.3.3 port 5001 connected with 10.0.2.2 port 44646
[ ID] Interval       Transfer     Bandwidth
[ 26]  0.0-10.1 sec  19.1 MBytes  15.9 Mbits/sec
```

Iperf H2 (promoted) to H3



```
root@p4:~/Desktop/FinalProj-KristinaAndMatt/HostTierPathSelection/DynSwitchConf
ig# iperf -c 10.0.3.3
------------------------------------------------------------
Client connecting to 10.0.3.3, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 23] local 10.0.2.2 port 44648 connected with 10.0.3.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 23]  0.0-10.0 sec  30.0 MBytes  25.1 Mbits/sec
root@p4:~/Desktop/FinalProj-KristinaAndMatt/HostTierPathSelection/DynSwitchConf
ig# 
```

```
root@p4:~/Desktop/FinalProj-KristinaAndMatt/HostTierPathSelection/DynSwitchConf
ig# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 26] local 10.0.3.3 port 5001 connected with 10.0.2.2 port 44648
[ ID] Interval       Transfer     Bandwidth
[ 26]  0.0-10.1 sec  30.0 MBytes  25.0 Mbits/sec
```

Here, we see that promoting H2 not only decreases latency, it also increases the total bandwidth achievable between H2 and H3. While H2 was low-tier, it achieved only 15.9 Mb/s to H3. After promotion, this number jumped to 25.1 Mb/s.

REVIEW OF TEAM MEMBER WORK

Kristina

I completed the application specific routing portion of this project which involved completing the following:

Checkpoint 1 (3/29):

- From P4 mininet tutorial complete:
    - Basic Forwarding
    - Basic Tunneling
    - P4 Runtime
    - Explicit Congestion Notification
- Create P4 file skeleton for basic routing in P4

Checkpoint 2 (4/16):

- Write P4 parser
- Learn how to debug P4
- Develop match/action tables that route traffic based on application type

Final Project (4/30):

- Add multi route hop inspection
- Debug main.p4
- Work on presentation slides
- Final report

Matt

I helped formulate the design ideas behind the two solutions and completed the host tier path selection portion of the project.

Checkpoint 1 (3/29):

- Created basic design regarding the problem areas and proposed solutions
- Architecture diagramming
- Mininet setup w/ help from Kristina
- Background research and methods of evaluation

Checkpoint 2 (4/16):

- P4 research (load balancing via ECMP, MRI tracerouting)

- Refactored design based on research. Old solutions were flawed and could not be implemented
- Topology design and preferred paths through them
- More mininet testing using various P4 tutorials, as well as gaining familiarity with evaluation tools and debugging P4 compilation

Final Project (4/30):

- Changed design approach for host tier path selection once again
- Implementing switch rules and P4 tables to achieve host tier path selection in a static setup
- Modified SDN controller to allow for dynamic switch reconfiguration. This allowed for host promotion during runtime
- Worked on presentation slides
- Finalized code and performed evaluation of host tier path selection
- Final report

CONCLUSION

After performing our final evaluations, we determined that our solutions were successful. We were able to achieve preferential routing based on both application/protocol type and host tiers. We found that preferred protocols and hosts achieved higher bandwidth and lower latency due to the paths they are given through the network.

Each solution's evaluation section went into more detail regarding statistics, however the key result is not the numbers observed in testing, but instead in the idea behind the solutions. Large scale implementations of our solutions would likely have vastly different results, as they are heavily dependent on the underlying topology.

On that note, we also determined that network topology is the driving force behind these solutions, and that carefully designing a topology for a specific use-case is of paramount importance. Similar to how a Clos topology is ideal for datacenters, the topologies we have designed are ideal for preferential route optimization. Although they are small-scale, the ideas behind the topologies can be implemented in real world settings.

For future work, we acknowledge that application-specific routing can be extended to match on layer 4 protocols such as RSTP, FTP, HTTP, etc. This would only require updates to the P4 tables matching keys, and the switch rules corresponding to those keys. For host tier path selection, we note that creating additional topologies and switch rules could be automated based on user-defined host tier mappings. This would allow an even more streamlined reconfiguration of switching logic during runtime.

Overall, we learned a lot doing this project and we're glad we got more experience with P4, software defined networking and testbed environments.