MPLS implementation using P4

Antoine Sauray

June 16, 2017

Contents

1 Definitions

P4

P4 is a programming language designed to allow programming of packet forwarding planes. In contrast to a general purpose language such as C or Python, P4 is a domain-specific language with a number of constructs optimized around network data forwarding. P4 is an open-source, permissively licensed language and is maintained by a non-profit organization called the P4 Language Consortium.

Multi protocol label switching (MPLS)

Multiprotocol Label Switching (MPLS) is a type of data-carrying technique for high-performance telecommunications networks. MPLS directs data from one network node to the next based on short path labels rather than long network addresses, avoiding complex lookups in a routing table. The labels identify virtual links (paths) between distant nodes rather than endpoints. MPLS implements three types of operations, which are swap, push and pop.

Mininet and bmv2

In order to execute P4 code, we need to virtualize a network. Using Mininet, we are able to create a virtualized network easily, and therefore to create topologies that suit our needs. The version 2 of the Behavioral Model (BMV2) is the second version of the P4 software switch. It will allow us to a simulate p4 switch on our network.

2 Simulated network

Topology

We propose a topology in order to test our P4 code. It can be created using the Python script "run_network.py".

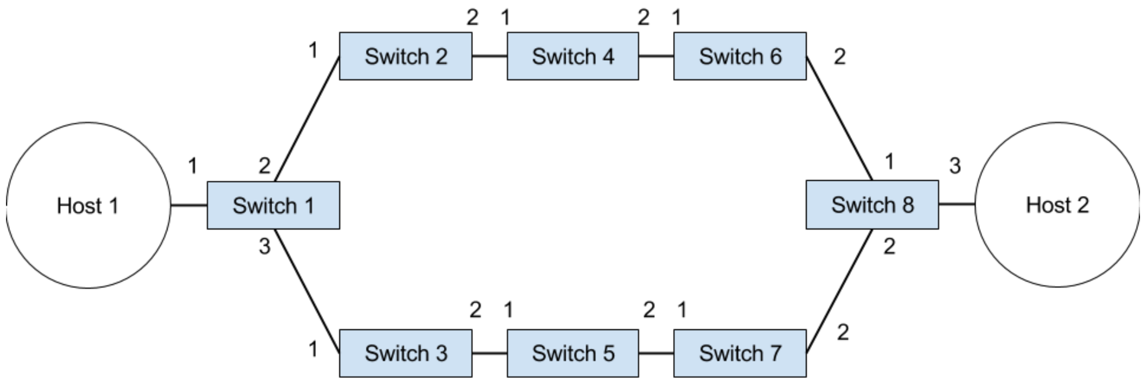


Figure 1: Proposed topology to test MPLS operations

Compile and run P4 code on a switch

in the Git repository, you will find a script named "compile.sh". This script compiles the P4 code in p4src/simple_router.p4 and generate a file named simple_router.json. This file matters because it is what will be loaded by the previous script "run_network.py" on each of the switch in the network. Also, please note that "run_network.py" has been modified to perform all the cleaning operations for you. It will handle sigint in order to remove your switches and will also perform cleaning on start if necessary.

```
git clone git@gitlab.testbed.se:berpec/p4_segmented_routing.git
cd mpls_implementation/src/mpls/
```

```
./compile.sh
python run_network.py
```

Applying P4 tables

Once the network started, the run_network.py script will load the tables in the switches. You can configure it by changing the command files. The thrift port relate to the way mininet works. It exposes each switch on the host through this thrift port. You can call the client by connecting to this port.

```
    # switch1
configure_dp("commands/commands-push.txt", thrift_port)
# switch2
configure_dp("commands/commands-push.txt", thrift_port+1)
# switch3
configure_dp("commands/commands-push.txt", thrift_port+2)
# switch4
configure_dp("commands/commands-pop.txt", thrift_port+3)
# switch5
configure_dp("commands/commands-pop.txt", thrift_port+4)
# switch6
configure_dp("commands/commands-pop.txt", thrift_port+5)
# switch7
configure_dp("commands/commands-swap.txt", thrift_port+6)
# switch8
configure_dp("commands/commands-s8.txt", thrift_port+7)
```

3 Implementing MPLS operations

Before applying actions, we need to match our input. This is the table that allows us to match a label and an ingress port to an action.

```
table mpls_exact {
reads {
    mpls.label : exact;
    standard_metadata.ingress_port : exact;
}
actions {
    mpls_swap;
```

```
                mpls_push;
                mpls_pop;
                _drop;
            }
            size: 1024;
        }
```

Swap

The swap is the easiest operation to implement. It takes the incoming port and a label, and forwards on an outgoing port with a new label. The code is as follows.

```
        action mpls_swap(new_label, port) {
            // set the metadata for the type of operation
            modify_field(mpls_metadata.mpls_type, MPLS_PUSH);
            // decrement ttl
            add_to_field(mpls.ttl, -1);
            // change the label to the new one
            modify_field(mpls.label, new_label);
            // set the outgoing port
            modify_field(standard_metadata.egress_spec, port);
        }
```

The commands for the table are the following

```
        table_add mpls_exact mpls_swap 200 1 => 200 2
```

For an incoming mpls packet with label 200 on port 1, forward it on port 2 with label 200.

Push

The push operation is more subttle due to a P4 limitation. The principle of push is to add a new MPLS header between the Ethernet header and the current MPLS header.

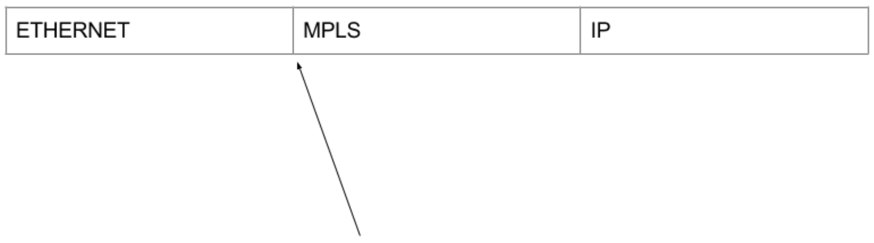


Figure 2: Where we want to add the new MPLS header

Though the problem with is P4 version 14, is that we cannot add a new header that is the same as the one we are currently parsing. This code does not work.

```
        action mpls_push(new_label, port, cos, ttl) {
            // set the metadata for the type of operation
            modify_field(mpls_metadata.mpls_type, MPLS_PUSH);
            // push a new mpls
            add_header(mpls);
            modify_field(mpls.label, new_label);
            modify_field(mpls.s, 0);
            modify_field(mpls.cos, cos);
            modify_field(mpls.ttl, ttl);
            add_to_field(mpls.ttl, -1);

            modify_field(standard_metadata.egress_spec, port);
        }
    }
```

P4 does not have such notion of variables. It calls headers by their name. In that case there is an unexpected behavior. Are we modifying the current MPLs or the new one we just added ? To fix this problem, the proposed solution of P4 programmers is to create a second header for MPLS, which contains the same fields but is named differently. In this example, we will name it mpls_between.

```
    header mpls_t mpls;
    header mpls_t mpls_between;
```

Another issue with P4 is that it does not allow us to create a header that is not used in the deparsing process. In our case, this mpls_between is used only for creation. We use the orignal mpls header for deparsing instead. There is a proposed hack to make this work.

```
parser parse_ethernet {
    extract(ethernet);
    return select(latest.etherType) {
        MPLS_unicast : parse_mpls;
        MPLS_multicast : parse_mpls;
         // HACK because of P4 : 0x8840 must never be reached
        0x8840 : parse_mpls_between;
        default: ingress;
    }
}
```

This idea behind this code is to find an ethernet type that will never be met in our case. For instance, we use 0x8840 here. In this code, parse_mpls will be called when we meet this type. In practice, because it never happens we will never never reach this part. P4 does is not aware of this and therefore allows the compilation process to finish successfully. it is very important that this 0x8840 type is never sent on the network otherwise the behavior will be unexpected, because the switch will try to parse a MPLS header where there is not. So here is the correct implementation.

```
    action mpls_push(new_label, port, cos, ttl) {
        // set the metadata for the type of operation
        modify_field(mpls_metadata.mpls_type, MPLS_PUSH);
        // push a new mpls header
        add_header(mpls_between);
        // change the label, bottom of stack, class of service and time to li
        modify_field(mpls_between.label, new_label);
        modify_field(mpls_between.s, 0);
        modify_field(mpls_between.cos, cos);
        modify_field(mpls_between.ttl, ttl);

        add_to_field(mpls.ttl, -1);
```

```
                modify_field(standard_metadata.egress_spec, port);
            }
```

Pop

The pop operation needs to remove the first packet of the MPLS stack. We need to handle a few cases.

1. If the current header is the not last (s, the bottom of stack bit, is set to 0), then we can remove the header.
2. If the current header is the last (s, the bottom of stack bit, is set to 1), then we remove the header but we need to change the Ethernet type because we are now in a situation where there is no more MPLS header to be parsed.

Table 1: Packet before and after Pop

(a)

ETHERNET	MPLS	IP

(b)

ETHERNET	IP

The issue here is we are not allowed to perform conditional branching in actions. We would like to do this.

```
        action mpls_pop(port) {
            modify_field(mpls_metadata.mpls_type, MPLS_POP);
            add_to_field(mpls.ttl, -1);
            if (mpls.s == 1) {
                // need to put the right type
                // once we remove all the mpls headers
                /  from the stack
                ethernet.etherType = ETHERTYPE_IPV4
            }
            remove_header(mpls);
            modify_field(standard_metadata.egress_spec, port);
        }
```

But this code is not correct. The solution we propose is to perform the conditional branching in the egress control. We will use the type of action for MPLS there and when we meet a pop action and a mpls bottom of stack, we will change the ethernet type to ipv4 (or ipv6 if we wanted to).

```
action set_ether_ipv4(){
    modify_field(ethernet.etherType, ETHERTYPE_IPV4);
}

table ether_type {
    actions {
        set_ether_ipv4;
    }
    size: 256;
}
control egress {
    if(mpls_metadata.mpls_type == MPLS_POP and mpls.s==1) {
        apply(ether_type);
    }
    apply(send_frame);
}
```

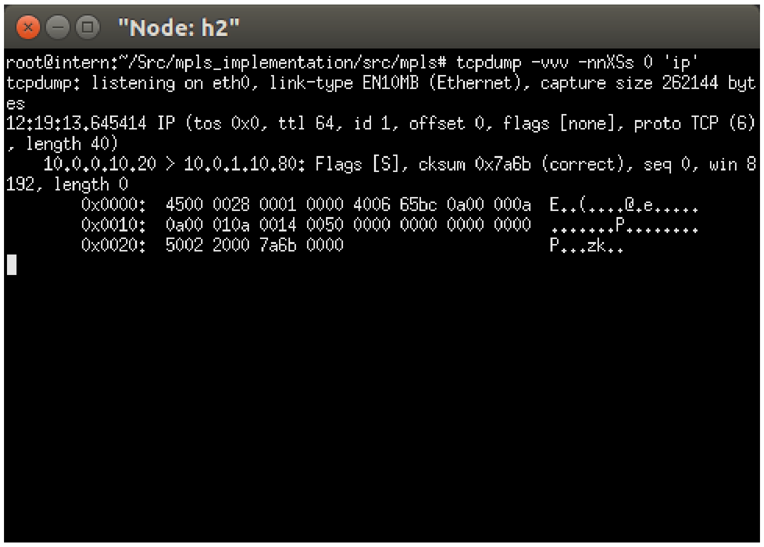
There is one thing to mention though with this approach. Because our table ether_type has nothing to match against, it will always be missed, and therefore our action will never be called. The workaround for this is to define the action set_ether_ipv4 as a default action for the table, which means that when the table will be missed, and it will, the default action to be executed is the one we want. We do it with the following code in the command file.

```
table_set_default ether_type set_ether_ipv4
```

Also, to specify the pop operation, we use this command.

```
table_add mpls_exact mpls_pop 200 1 => 2
```

For an incoming mpls packet with label 200 on port 1, forward it on port 2.



```
root@intern:~/Src/mpls_implementation/src/mpls# tcpdump -vvv -nnXSs 0 'ip'
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144 byt
es
12:19:13.645414 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6)
, length 40)
    10.0.0.10.20 > 10.0.1.10.80: Flags [S], cksum 0x7a6b (correct), seq 0, win 8
192, length 0
        0x0000:  4500 0028 0001 0000 4006 65bc 0a00 000a  E..(....@.e.....
        0x0010:  0a00 010a 0014 0050 0000 0000 0000 0000  .......P........
        0x0020:  5002 2000 7a6b 0000                      P...zk..
```

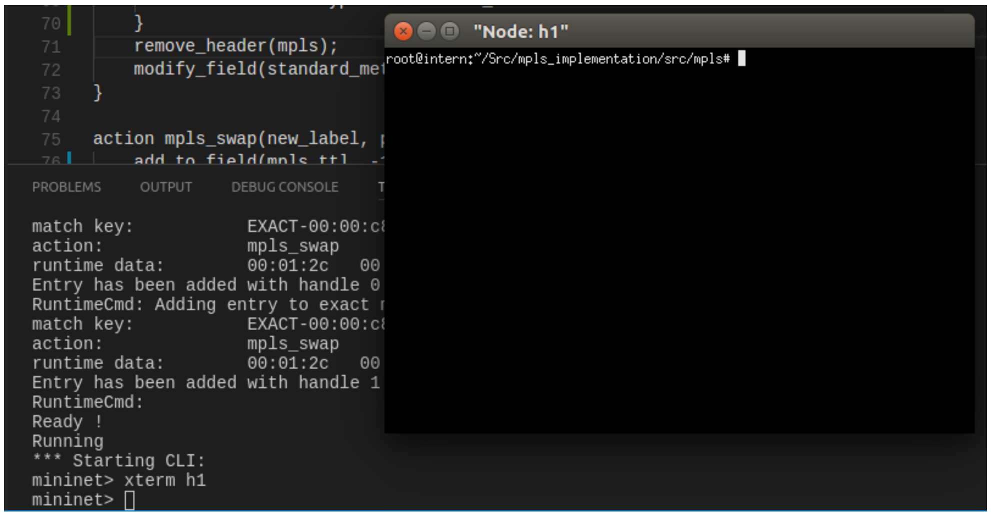
Figure 3: You can see that the packet arrives as ip on host 2

4 Testing, logging and monitoring

Testing

in the git repository, you will find a script called "test.py". This script forges MPLS packets. In order to test your network, you need to start it from HOST 1. After you run the "run_network.py" script, it opens the mininet client. Type the following lines of code.

```
xterm h1
python test.py
```



```
    }
    remove_header(mpls);
    modify_field(standard_me
}

action mpls_swap(new_label, 
    add_to_field(mpls_ttl, -
```

```
match key:           EXACT-00:00:c
action:              mpls_swap
runtime data:        00:01:2c   00
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact 
match key:           EXACT-00:00:c
action:              mpls_swap
runtime data:        00:01:2c   00
Entry has been added with handle 1
RuntimeCmd:
Ready !
Running
*** Starting CLI:
mininet> xterm h1
mininet> 
```

```
root@intern:~/Src/mpls_implementation/src/mpls# 
```

Figure 4: Terminal access on host 1 using mininet client

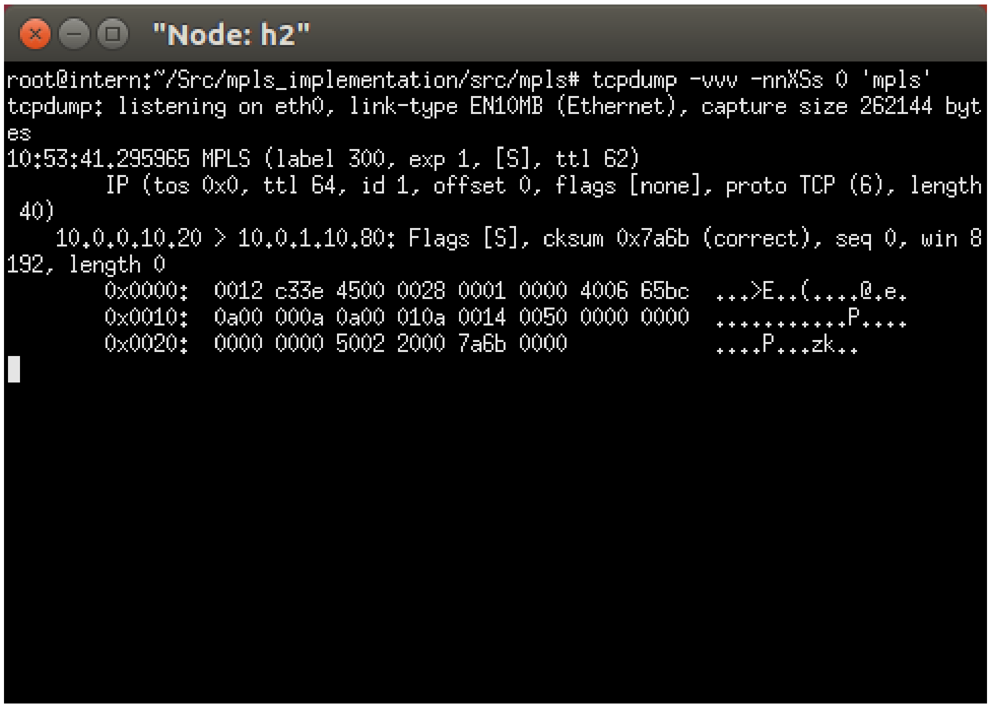
It will execute the test script on HOST 1. The problem is that right now, you cannot see what is going on. We will talk about this in the logging section.

Logging

Packets on Host 2

First, you need to spawn a terminal for host 2 using the "xterm h2" command in the mininet client. Then you need to execute the following tcpdump command in order to see the traffic.

```
xterm h2
tcpdump -vvv -nnXSs 0 'mpls'
```



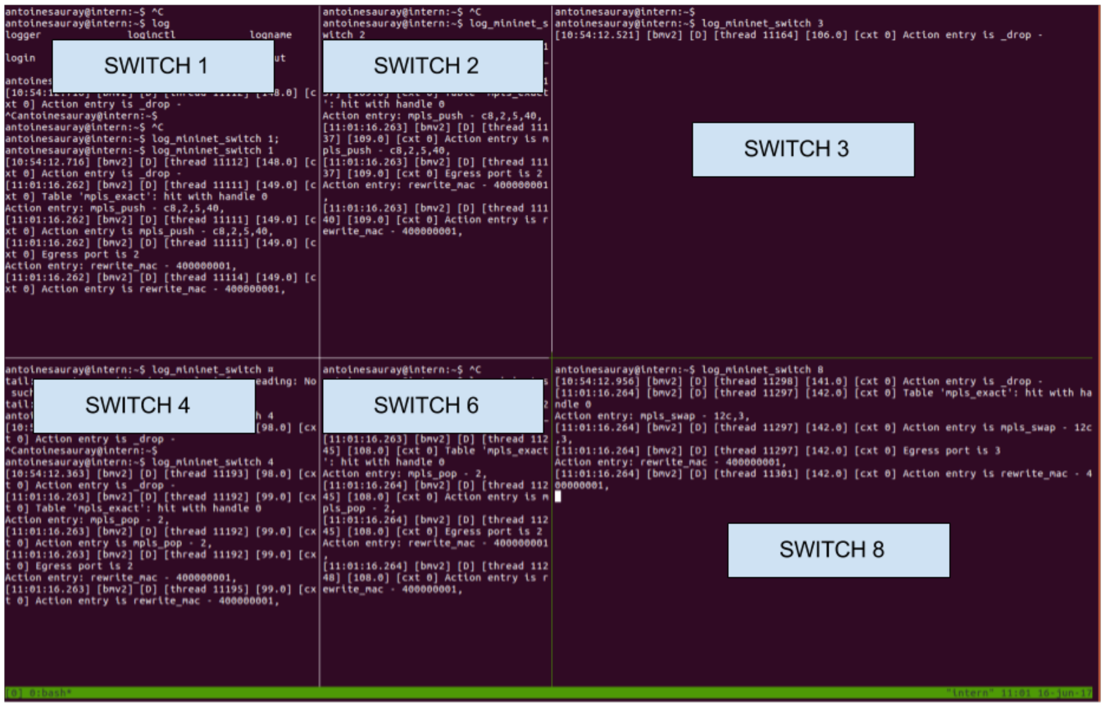
```
root@intern:~/Src/mpls_implementation/src/mpls# tcpdump -vvv -nnXSs 0 'mpls'
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144 byt
es
10:53:41.295965 MPLS (label 300, exp 1, [S], ttl 62)
        IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length
 40)
    10.0.0.10.20 > 10.0.1.10.80: Flags [S], cksum 0x7a6b (correct), seq 0, win 8
192, length 0
        0x0000:  0012 c33e 4500 0028 0001 0000 4006 65bc  ...>E..(....@.e.
        0x0010:  0a00 000a 0a00 010a 0014 0050 0000 0000  ...........P....
        0x0020:  0000 0000 5002 2000 7a6b 0000            ....P...zk..
```

Figure 5: Logs on Host 2 using tcpdump after host 1 executed test.py, we can see the mpls packet and its fields

Switch logging and route discovery

We propose a script named "log_mininet_switch.sh" which allows to log any P4 execution information. It can serve multiple purposes.

1. Debug your P4 code on any switch on the network
2. Discover which switches handled your traffic



```
antoinesauray@intern:~$ ^C
antoinesauray@intern:~$ log
logger              loginctl            logname

login                                        ut

antoines
[10:54:1                                  148.0] [c
xt 0] Action entry is _drop -
^Cantoinesauray@intern:~$
antoinesauray@intern:~$ ^C
antoinesauray@intern:~$ log_mininet_switch 1;
antoinesauray@intern:~$ log_mininet_switch 1
[10:54:12.716] [bmv2] [D] [thread 11112] [148.0] [c
xt 0] Action entry is _drop -
[11:01:16.262] [bmv2] [D] [thread 11111] [149.0] [c
xt 0] Table 'mpls_exact': hit with handle 0
Action entry: mpls_push - c8,2,5,40,
[11:01:16.262] [bmv2] [D] [thread 11111] [149.0] [c
xt 0] Action entry is mpls_push - c8,2,5,40,
[11:01:16.262] [bmv2] [D] [thread 11111] [149.0] [c
xt 0] Egress port is 2
Action entry: rewrite_mac - 400000001,
[11:01:16.262] [bmv2] [D] [thread 11114] [149.0] [c
xt 0] Action entry is rewrite_mac - 400000001,
```

```
antoinesauray@intern:~$ ^C
antoinesauray@intern:~$ log_mininet_s
witch 2

'; hit with handle 0
Action entry: mpls_push - c8,2,5,40,
[11:01:16.263] [bmv2] [D] [thread 111
37] [109.0] [cxt 0] Action entry is m
pls_push - c8,2,5,40,
[11:01:16.263] [bmv2] [D] [thread 111
37] [109.0] [cxt 0] Egress port is 2
Action entry: rewrite_mac - 400000001
,
[11:01:16.263] [bmv2] [D] [thread 111
40] [109.0] [cxt 0] Action entry is r
ewrite_mac - 400000001,
```

```
antoinesauray@intern:~$
antoinesauray@intern:~$ log_mininet_switch 3
[10:54:12.521] [bmv2] [D] [thread 11164] [106.0] [cxt 0] Action entry is _drop -
```

```
antoinesauray@intern:~$ log_mininet_switch #
tail:                                    eading: No
 such
tail:
antoi                                    h 4
[10:5                                    [98.0] [cx
t 0] Action entry is _drop -
^Cantoinesauray@intern:~$
antoinesauray@intern:~$ log_mininet_switch 4
[10:54:12.363] [bmv2] [D] [thread 11193] [98.0] [cx
t 0] Action entry is _drop -
[11:01:16.263] [bmv2] [D] [thread 11192] [99.0] [cx
t 0] Table 'mpls_exact': hit with handle 0
Action entry: mpls_pop - 2,
[11:01:16.263] [bmv2] [D] [thread 11192] [99.0] [cx
t 0] Action entry is mpls_pop - 2,
[11:01:16.263] [bmv2] [D] [thread 11192] [99.0] [cx
t 0] Egress port is 2
Action entry: rewrite_mac - 400000001,
[11:01:16.263] [bmv2] [D] [thread 11195] [99.0] [cx
t 0] Action entry is rewrite_mac - 400000001,
```

```
antoinesauray@intern:~$ ^C

[11:01:16.263] [bmv2] [D] [thread 112
45] [108.0] [cxt 0] Table 'mpls_exact
': hit with handle 0
Action entry: mpls_pop - 2,
[11:01:16.264] [bmv2] [D] [thread 112
45] [108.0] [cxt 0] Action entry is m
pls_pop - 2,
[11:01:16.264] [bmv2] [D] [thread 112
45] [108.0] [cxt 0] Egress port is 2
Action entry: rewrite_mac - 400000001
,
[11:01:16.264] [bmv2] [D] [thread 112
48] [108.0] [cxt 0] Action entry is r
ewrite_mac - 400000001,
```

```
antoinesauray@intern:~$ log_mininet_switch 8
[10:54:12.956] [bmv2] [D] [thread 11298] [141.0] [cxt 0] Action entry is _drop -
[11:01:16.264] [bmv2] [D] [thread 11297] [142.0] [cxt 0] Table 'mpls_exact': hit with ha
ndle 0
Action entry: mpls_swap - 12c,3,
[11:01:16.264] [bmv2] [D] [thread 11297] [142.0] [cxt 0] Action entry is mpls_swap - 12c
,3,
[11:01:16.264] [bmv2] [D] [thread 11297] [142.0] [cxt 0] Egress port is 3
Action entry: rewrite_mac - 400000001,
[11:01:16.264] [bmv2] [D] [thread 11301] [142.0] [cxt 0] Action entry is rewrite_mac - 4
00000001,
```

```
[0] 0:bash*                                              "intern" 11:01 16-Jun-17
```

Figure 6: Easy logging of each switch in the network

On this image, you can see traffic going from host 1 to switch 1, to switch 2, to switch 4, to switch 6 and finally to switch 8. On each of the switches you can see the match action the packet has gone through and what happened to it. Also, as you see, this script works well when used with a terminal multiplexer such as tmux.

Conclusion

You now has a fully working version of MPLS using P4. The next step will be to implement segment routing, a new technology which will allow to have FRR protection for any topology, simpler to operate and more scalable.