OVS Faucet Tutorial

This tutorial demonstrates how Open vSwitch works with a controller, using the Faucet controller as a simple way to get started. It was tested with the "master" branch of Open vSwitch and version 1.6.7 of Faucet in October 2017. It does not use advanced or recently added features in OVS or Faucet, so other versions of both pieces of software are likely to work equally well.

The goal of the tutorial is to demonstrate Open vSwitch and Faucet in an end-to-end way, that is, to show how it works from the Faucet controller configuration at the top, through the OpenFlow flow table, to the datapath processing. Along the way, in addition to helping to understand the architecture at each level, we discuss performance and troubleshooting issues. We hope that this demonstration makes it easier for users and potential users to understand how Open vSwitch works and how to debug and troubleshoot it.

We provide enough details in the tutorial that you should be able to fully follow along by following the instructions.

Setting Up OVS

This section explains how to set up Open vSwitch for the purpose of using it with Faucet for the tutorial.

You might already have Open vSwitch installed on one or more computers or VMs, perhaps set up to control a set of VMs or a physical network. This is admirable, but we will be using Open vSwitch in a different way to set up a simulation environment called the OVS "sandbox". The sandbox does not use virtual machines or containers, which makes it more limited, but on the other hand it is (in this writer's opinion) easier to set up.

There are two ways to start a sandbox: one that uses the Open vSwitch that is already installed on a system, and another that uses a copy of Open vSwitch that has been built but not yet installed. The latter is more often used and thus better tested, but both should work. The instructions below explain both approaches:

1. Get a copy of the Open vSwitch source repository using Git, then `cd` into the new directory:

   ```
   $ git clone https://github.com/openvswitch/ovs.git
   $ cd ovs
   ```

   The default checkout is the master branch. You can check out a tag (such as v2.8.0) or a branch (such as origin/branch-2.8), if you prefer.

2. If you do not already have an installed copy of Open vSwitch on your system, or if you do not want to use it for the sandbox (the sandbox will not disturb the functionality of any existing switches), then proceed to step 3. If you do have an installed copy and you want to use it for the sandbox, try to start the sandbox by running:

   ```
   $ tutorial/ovs-sandbox
   ```

   If it is successful, you will find yourself in a subshell environment, which is the sandbox (you can exit with `exit` or Control+D). If so, you're finished and do not need to complete the rest of the steps. If it fails, you can proceed to step 3 to build Open vSwitch anyway.

3. Before you build, you might want to check that your system meets the build requirements. Read :doc:`/intro/install/general` to find out. For this tutorial, there is no need to compile the Linux kernel module, or to use any of the optional libraries such as OpenSSL, DPDK, or libcap-ng.

4. Configure and build Open vSwitch:

```
$ ./boot.sh
$ ./configure
$ make -j4
```

5. Try out the sandbox by running:

```
$ make sandbox
```

   You can exit the sandbox with `exit` or Control+D.

Setting up Faucet

This section explains how to get a copy of Faucet and set it up appropriately for the tutorial. There are many other ways to install Faucet, but this simple approach worked well for me. It has the advantage that it does not require modifying any system-level files or directories on your machine. It does, on the other hand, require Docker, so make sure you have it installed and working.

It will be a little easier to go through the rest of the tutorial if you run these instructions in a separate terminal from the one that you're using for Open vSwitch, because it's often necessary to switch between one and the other.

1. Get a copy of the Faucet source repository using Git, then `cd` into the new directory:

```
$ git clone https://github.com/faucetsdn/faucet.git
$ cd faucet
```

   At this point I checked out the latest tag:

```
$ git checkout v1.6.7
```

2. Build a docker container image:

```
$ docker build -t faucet/faucet .
```

   This will take a few minutes.

3. Create an installation directory under the `faucet` directory for the docker image to use:

```
$ mkdir inst
```

   The Faucet configuration will go in `inst/faucet.yaml` and its main log will appear in `inst/faucet.log`. (The official Faucet installation instructions call to put these in `/etc/ryu/faucet` and `/var/log/ryu/faucet`, respectively, but we avoid modifying these system directories.)

4. Create a container and start Faucet:

```
$ docker run -d --name faucet -v `pwd`/inst/:/etc/ryu/faucet/ -v `pwd`/inst/:/var/lo
```

5. Look in `inst/faucet.log` to verify that Faucet started. It will probably start with an exception and traceback because we have not yet created `inst/faucet.yaml`.

6. Later on, to make a new or updated Faucet configuration take effect quickly, you can run:

```
$ docker exec faucet pkill -HUP -f faucet.faucet
```

Another way is to stop and start the Faucet container:

```
$ docker restart faucet
```

You can also stop and delete the container; after this, to start it again, you need to rerun the `docker run` command:

```
$ docker stop faucet
$ docker rm faucet
```

Overview

Now that Open vSwitch and Faucet are ready, here's an overview of what we're going to do for the remainder of the tutorial:

1. Switching: Set up an L2 network with Faucet.
2. Routing: Route between multiple L3 networks with Faucet.
3. ACLs: Add and modify access control rules.

At each step, we will take a look at how the features in question work from Faucet at the top to the data plane layer at the bottom. From the highest to lowest level, these layers and the software components that connect them are:

- Faucet, which as the top level in the system is the authoritative source of the network configuration.

  Faucet connects to a variety of monitoring and performance tools, but we won't use them in this tutorial. Our main insights into the system will be through `faucet.yaml` for configuration and `faucet.log` to observe state, such as MAC learning and ARP resolution, and to tell when we've screwed up configuration syntax or semantics.

- The OpenFlow subsystem in Open vSwitch. OpenFlow is the protocol, standardized by the Open Networking Foundation, that controllers like Faucet use to control how Open vSwitch and other switches treat packets in the network.

  We will use `ovs-ofctl`, a utility that comes with Open vSwitch, to observe and occasionally modify Open vSwitch's OpenFlow behavior. We will also use `ovs-appctl`, a utility for communicating with `ovs-vswitchd` and other Open vSwitch daemons, to ask "what-if?" type questions.

  In addition, the OVS sandbox by default raises the Open vSwitch logging level for OpenFlow high enough that we can learn a great deal about OpenFlow behavior simply by reading its log file.

- Open vSwitch datapath. This is essentially a cache designed to accelerate packet processing. Open vSwitch includes a few different datapaths, such as one based on the Linux kernel and a userspace-only datapath (sometimes called the "DPDK" datapath). The OVS sandbox uses the latter, but the principles behind it apply equally well to other datapaths.

At each step, we discuss how the design of each layer influences performance. We demonstrate how Open vSwitch features can be used to debug, troubleshoot, and understand the system as a whole.

Switching

Layer-2 (L2) switching is the basis of modern networking. It's also very simple and a good place to start, so let's set up a switch with some VLANs in Faucet and see how it works at each layer. Begin by putting the following into `inst/faucet.yaml`:

```
dps:
    switch-1:
        dp_id: 0x1
        timeout: 3600
        arp_neighbor_timeout: 3600
        interfaces:
            1:
                native_vlan: 100
            2:
                native_vlan: 100
            3:
                native_vlan: 100
            4:
                native_vlan: 200
            5:
                native_vlan: 200
vlans:
    100:
    200:
```

This configuration file defines a single switch ("datapath" or "dp") named `switch-1`. The switch has five ports, numbered 1 through 5. Ports 1, 2, and 3 are in VLAN 100, and ports 4 and 5 are in VLAN 2. Faucet can identify the switch from its datapath ID, which is defined to be 0x1.

> ***Note***
>
> This also sets high MAC learning and ARP timeouts. The defaults are 5 minutes and about 8 minutes, which are fine in production but sometimes too fast for manual experimentation. (Don't use a timeout bigger than about 65000 seconds because it will crash Faucet.)

Now restart Faucet so that the configuration takes effect, e.g.:

```
$ docker restart faucet
```

Assuming that the configuration update is successful, you should now see a new line at the end of `inst/faucet.log`:

```
Oct 14 22:36:42 faucet INFO     Add new datapath DPID 1 (0x1)
```

Faucet is now waiting for a switch with datapath ID 0x1 to connect to it over OpenFlow, so our next step is to create a switch with OVS and make it connect to Faucet. To do that, switch to the terminal where you checked out OVS and start a sandbox with `make sandbox` or *tutorial/ovs-sandbox*` (as explained earlier under Setting Up OVS). You should see something like this toward the end of the output:

```
---------------------------------------------------------------------
You are running in a dummy Open vSwitch environment.  You can use
ovs-vsctl, ovs-ofctl, ovs-appctl, and other tools to work with the
dummy switch.

Log files, pidfiles, and the configuration database are in the
"sandbox" subdirectory.
```

```
Exit the shell to kill the running daemons.
blp@sigabrt:~/nicira/ovs/tutorial(0)$
```

Inside the sandbox, create a switch ("bridge") named `br0`, set its datapath ID to 0x1, add simulated ports to it named `p1` through `p5`, and tell it to connect to the Faucet controller. To make it easier to understand, we request for port `p1` to be assigned OpenFlow port 1, `p2` port 2, and so on. As a final touch, configure the controller to be "out-of-band" (this is mainly to avoid some annoying messages in the `ovs-vswitchd` logs; for more information, run `man ovs-vswitchd.conf.db` and search for `connection_mode`):

```
$ ovs-vsctl add-br br0 \
        -- set bridge br0 other-config:datapath-id=0000000000000001 \
        -- add-port br0 p1 -- set interface p1 ofport_request=1 \
        -- add-port br0 p2 -- set interface p2 ofport_request=2 \
        -- add-port br0 p3 -- set interface p3 ofport_request=3 \
        -- add-port br0 p4 -- set interface p4 ofport_request=4 \
        -- add-port br0 p5 -- set interface p5 ofport_request=5 \
        -- set-controller br0 tcp:127.0.0.1:6653 \
        -- set controller br0 connection-mode=out-of-band
```

> ***Note***
>
> You don't have to run all of these as a single `ovs-vsctl` invocation. It is a little more efficient, though, and since it updates the OVS configuration in a single database transaction it means that, for example, there is never a time when the controller is set but it has not yet been configured as out-of-band.

Now, if you look at `inst/faucet.log` again, you should see that Faucet recognized and configured the new switch and its ports:

```
Oct 14 22:50:08 faucet.valve INFO     DPID 1 (0x1) Cold start configuring DP
Oct 14 22:50:08 faucet.valve INFO     DPID 1 (0x1) Configuring VLAN 100 vid:100 ports:Po
Oct 14 22:50:08 faucet.valve INFO     DPID 1 (0x1) Configuring VLAN 200 vid:200 ports:Po
Oct 14 22:50:08 faucet.valve INFO     DPID 1 (0x1) Port Port 1 up, configuring
Oct 14 22:50:08 faucet.valve INFO     DPID 1 (0x1) Port Port 2 up, configuring
Oct 14 22:50:08 faucet.valve INFO     DPID 1 (0x1) Port Port 3 up, configuring
Oct 14 22:50:08 faucet.valve INFO     DPID 1 (0x1) Port Port 4 up, configuring
Oct 14 22:50:08 faucet.valve INFO     DPID 1 (0x1) Port Port 5 up, configuring
```

Over on the Open vSwitch side, you can see a lot of related activity if you take a look in `sandbox/ovs-vswitchd.log`. For example, here is the basic OpenFlow session setup and Faucet's probe of the switch's ports and capabilities:

```
rconn|INFO|br0<->tcp:127.0.0.1:6653: connecting...
vconn|DBG|tcp:127.0.0.1:6653: sent (Success): OFPT_HELLO (OF1.4) (xid=0x1):
 version bitmap: 0x01, 0x02, 0x03, 0x04, 0x05
vconn|DBG|tcp:127.0.0.1:6653: received: OFPT_HELLO (OF1.3) (xid=0x2f24810a):
 version bitmap: 0x01, 0x02, 0x03, 0x04
vconn|DBG|tcp:127.0.0.1:6653: negotiated OpenFlow version 0x04 (we support version 0x05
rconn|INFO|br0<->tcp:127.0.0.1:6653: connected
vconn|DBG|tcp:127.0.0.1:6653: received: OFPT_ECHO_REQUEST (OF1.3) (xid=0x2f24810b): 0 by
vconn|DBG|tcp:127.0.0.1:6653: sent (Success): OFPT_ECHO_REPLY (OF1.3) (xid=0x2f24810b):
```

```
vconn|DBG|tcp:127.0.0.1:6653: received: OFPT_FEATURES_REQUEST (OF1.3) (xid=0x2f24810c):
vconn|DBG|tcp:127.0.0.1:6653: sent (Success): OFPT_FEATURES_REPLY (OF1.3) (xid=0x2f24810
 n_tables:254, n_buffers:0
 capabilities: FLOW_STATS TABLE_STATS PORT_STATS GROUP_STATS QUEUE_STATS
vconn|DBG|tcp:127.0.0.1:6653: received: OFPST_PORT_DESC request (OF1.3) (xid=0x2f24810d)
vconn|DBG|tcp:127.0.0.1:6653: sent (Success): OFPST_PORT_DESC reply (OF1.3) (xid=0x2f248
 1(p1): addr:aa:55:aa:55:00:14
     config:     PORT_DOWN
     state:      LINK_DOWN
     speed: 0 Mbps now, 0 Mbps max
 2(p2): addr:aa:55:aa:55:00:15
     config:     PORT_DOWN
     state:      LINK_DOWN
     speed: 0 Mbps now, 0 Mbps max
 3(p3): addr:aa:55:aa:55:00:16
     config:     PORT_DOWN
     state:      LINK_DOWN
     speed: 0 Mbps now, 0 Mbps max
 4(p4): addr:aa:55:aa:55:00:17
     config:     PORT_DOWN
     state:      LINK_DOWN
     speed: 0 Mbps now, 0 Mbps max
 5(p5): addr:aa:55:aa:55:00:18
     config:     PORT_DOWN
     state:      LINK_DOWN
     speed: 0 Mbps now, 0 Mbps max
 LOCAL(br0): addr:c6:64:ff:59:48:41
     config:     PORT_DOWN
     state:      LINK_DOWN
     speed: 0 Mbps now, 0 Mbps max
```

After that, you can see Faucet delete all existing flows and then start adding new ones:

```
vconn|DBG|tcp:127.0.0.1:6653: received: OFPT_FLOW_MOD (OF1.3) (xid=0x2f24810e): DEL tabl
vconn|DBG|tcp:127.0.0.1:6653: received: OFPT_BARRIER_REQUEST (OF1.3) (xid=0x2f24810f):
vconn|DBG|tcp:127.0.0.1:6653: sent (Success): OFPT_BARRIER_REPLY (OF1.3) (xid=0x2f24810f
vconn|DBG|tcp:127.0.0.1:6653: received: OFPT_FLOW_MOD (OF1.3) (xid=0x2f248110): ADD pric
vconn|DBG|tcp:127.0.0.1:6653: received: OFPT_FLOW_MOD (OF1.3) (xid=0x2f248111): ADD tabl
...
```

OpenFlow Layer

Let's take a look at the OpenFlow tables that Faucet set up. Before we do that, it's helpful to take a look at `docs/architecture.rst` in the Faucet documentation to learn how Faucet structures its flow tables. In summary, this document says:

Table 0

Port-based ACLs

Table 1

Ingress VLAN processing

Table 2

VLAN-based ACLs

Table 3

Ingress L2 processing, MAC learning

Table 4

L3 forwarding for IPv4

Table 5

L3 forwarding for IPv6

Table 6

Virtual IP processing, e.g. for router IP addresses implemented by Faucet

Table 7

Egress L2 processing

Table 8

Flooding

With that in mind, let's dump the flow tables. The simplest way is to just run plain `ovs-ofctl dump-flows`:

```
$ ovs-ofctl dump-flows br0
```

If you run that bare command, it produces a lot of extra junk that makes the output harder to read, like statistics and "cookie" values that are all the same. In addition, for historical reasons `ovs-ofctl` always defaults to using OpenFlow 1.0 even though Faucet and most modern controllers use OpenFlow 1.3, so it's best to force it to use OpenFlow 1.3. We could throw in a lot of options to fix these, but we'll want to do this more than once, so let's start by defining a shell function for ourselves:

```
$ dump-flows () {
  ovs-ofctl -OOpenFlow13 --names --no-stat dump-flows "$@" \
    | sed 's/cookie=0x5adc15c0, //'
}
```

Let's also define `save-flows` and `diff-flows` functions for later use:

```
$ save-flows () {
  ovs-ofctl -OOpenFlow13 --no-names --sort dump-flows "$@"
}
$ diff-flows () {
  ovs-ofctl -OOpenFlow13 diff-flows "$@" | sed 's/cookie=0x5adc15c0 //'
}
```

Now let's take a look at the flows we've got and what they mean, like this:

```
$ dump-flows br0
```

First, table 0 has a flow that just jumps to table 1 for each configured port, and drops other unrecognized packets. Presumably it will do more if we configured port-based ACLs:

```
priority=9099,in_port=p1 actions=goto_table:1
priority=9099,in_port=p2 actions=goto_table:1
priority=9099,in_port=p3 actions=goto_table:1
priority=9099,in_port=p4 actions=goto_table:1
priority=9099,in_port=p5 actions=goto_table:1
priority=0 actions=drop
```

Table 1, for ingress VLAN processing, has a bunch of flows that drop inappropriate packets, like those that claim to be from a broadcast source address (why not from all multicast source addresses, though?):

```
table=1, priority=9099,dl_src=ff:ff:ff:ff:ff:ff actions=drop
table=1, priority=9001,dl_src=0e:00:00:00:00:01 actions=drop
table=1, priority=9099,dl_dst=01:80:c2:00:00:00 actions=drop
table=1, priority=9099,dl_dst=01:00:0c:cc:cc:cd actions=drop
table=1, priority=9099,dl_type=0x88cc actions=drop
```

Table 1 also has some more interesting flows that recognize packets without a VLAN header on each of our ports (`vlan_tci=0x0000/0x1fff`), push on the VLAN configured for the port, and proceed to table 3. Presumably these skip table 2 because we did not configure any VLAN-based ACLs. There is also a fallback flow to drop other packets, which in practice means that if any received packet already has a VLAN header then it will be dropped:

```
table=1, priority=9000,in_port=p1,vlan_tci=0x0000/0x1fff actions=push_vlan:0x8100,set_fi
table=1, priority=9000,in_port=p2,vlan_tci=0x0000/0x1fff actions=push_vlan:0x8100,set_fi
table=1, priority=9000,in_port=p3,vlan_tci=0x0000/0x1fff actions=push_vlan:0x8100,set_fi
table=1, priority=9000,in_port=p4,vlan_tci=0x0000/0x1fff actions=push_vlan:0x8100,set_fi
table=1, priority=9000,in_port=p5,vlan_tci=0x0000/0x1fff actions=push_vlan:0x8100,set_fi
table=1, priority=0 actions=drop
```

> ***Note***
>
> The syntax `set_field:4196->vlan_vid` is curious and somewhat misleading. OpenFlow 1.3 defines the `vlan_vid` field as a 13-bit field where bit 12 is set to 1 if the VLAN header is present. Thus, since 4196 is 0x1064, this action sets VLAN value 0x64, which in decimal is 100.

Table 2 isn't used because there are no VLAN-based ACLs. It just has a drop flow:

```
table=2, priority=0 actions=drop
```

Table 3 is used for MAC learning but the controller hasn't learned any MAC yet. We'll come back here later:

```
table=3, priority=0 actions=drop
table=3, priority=9000 actions=CONTROLLER:96,goto_table:7
```

Tables 4, 5, and 6 aren't used because we haven't configured any routing:

```
table=4, priority=0 actions=drop
table=5, priority=0 actions=drop
table=6, priority=0 actions=drop
```

Table 7 is used to direct packets to learned MACs but Faucet hasn't learned any MACs yet, so it just sends all the packets along to table 8:

```
table=7, priority=0 actions=drop
table=7, priority=9000 actions=goto_table:8
```

Table 8 implements flooding, broadcast, and multicast. The flows for broadcast and flood are easy to understand: if the packet came in on a given port and needs to be flooded or broadcast, output it to all the other ports in the same VLAN:

```
table=8, priority=9008,in_port=p1,dl_vlan=100,dl_dst=ff:ff:ff:ff:ff:ff actions=pop_vlan,
table=8, priority=9008,in_port=p2,dl_vlan=100,dl_dst=ff:ff:ff:ff:ff:ff actions=pop_vlan,
table=8, priority=9008,in_port=p3,dl_vlan=100,dl_dst=ff:ff:ff:ff:ff:ff actions=pop_vlan,
table=8, priority=9008,in_port=p4,dl_vlan=200,dl_dst=ff:ff:ff:ff:ff:ff actions=pop_vlan,
table=8, priority=9008,in_port=p5,dl_vlan=200,dl_dst=ff:ff:ff:ff:ff:ff actions=pop_vlan,
table=8, priority=9000,in_port=p1,dl_vlan=100 actions=pop_vlan,output:p2,output:p3
table=8, priority=9000,in_port=p2,dl_vlan=100 actions=pop_vlan,output:p1,output:p3
table=8, priority=9000,in_port=p3,dl_vlan=100 actions=pop_vlan,output:p1,output:p2
table=8, priority=9000,in_port=p4,dl_vlan=200 actions=pop_vlan,output:p5
table=8, priority=9000,in_port=p5,dl_vlan=200 actions=pop_vlan,output:p4
```

Note

These flows could apparently be simpler because OpenFlow says that `output:<port>` is ignored if `<port>` is the input port. That means that the first three flows above could apparently be collapsed into just:

```
table=8, priority=9008,dl_vlan=100,dl_dst=ff:ff:ff:ff:ff:ff actions=pop_vlan,output:p
```

There might be some reason why this won't work or isn't practical, but that isn't obvious from looking at the flow table.

There are also some flows for handling some standard forms of multicast, and a fallback drop flow:

```
table=8, priority=9006,in_port=p1,dl_vlan=100,dl_dst=33:33:00:00:00:00/ff:ff:00:00:00:00
table=8, priority=9006,in_port=p2,dl_vlan=100,dl_dst=33:33:00:00:00:00/ff:ff:00:00:00:00
table=8, priority=9006,in_port=p3,dl_vlan=100,dl_dst=33:33:00:00:00:00/ff:ff:00:00:00:00
table=8, priority=9006,in_port=p4,dl_vlan=200,dl_dst=33:33:00:00:00:00/ff:ff:00:00:00:00
table=8, priority=9006,in_port=p5,dl_vlan=200,dl_dst=33:33:00:00:00:00/ff:ff:00:00:00:00
table=8, priority=9002,in_port=p1,dl_vlan=100,dl_dst=01:80:c2:00:00:00/ff:ff:ff:00:00:00
table=8, priority=9002,in_port=p2,dl_vlan=100,dl_dst=01:80:c2:00:00:00/ff:ff:ff:00:00:00
table=8, priority=9002,in_port=p3,dl_vlan=100,dl_dst=01:80:c2:00:00:00/ff:ff:ff:00:00:00
table=8, priority=9004,in_port=p1,dl_vlan=100,dl_dst=01:00:5e:00:00:00/ff:ff:ff:00:00:00
table=8, priority=9004,in_port=p2,dl_vlan=100,dl_dst=01:00:5e:00:00:00/ff:ff:ff:00:00:00
table=8, priority=9004,in_port=p3,dl_vlan=100,dl_dst=01:00:5e:00:00:00/ff:ff:ff:00:00:00
table=8, priority=9002,in_port=p4,dl_vlan=200,dl_dst=01:80:c2:00:00:00/ff:ff:ff:00:00:00
table=8, priority=9002,in_port=p5,dl_vlan=200,dl_dst=01:80:c2:00:00:00/ff:ff:ff:00:00:00
table=8, priority=9004,in_port=p4,dl_vlan=200,dl_dst=01:00:5e:00:00:00/ff:ff:ff:00:00:00
table=8, priority=9004,in_port=p5,dl_vlan=200,dl_dst=01:00:5e:00:00:00/ff:ff:ff:00:00:00
table=8, priority=0 actions=drop
```

Tracing

Let's go a level deeper. So far, everything we've done has been fairly general. We can also look at something more specific: the path that a particular packet would take through Open vSwitch. We can use OVN `ofproto/trace` command to play "what-if?" games. This command is one that we send directly to `ovs-vswitchd`, using the `ovs-appctl` utility.

Note

`ovs-appctl` is actually a very simple-minded JSON-RPC client, so you could also use some other utility that speaks JSON-RPC, or access it from a program as an API.

The `ovs-vswitchd`(8) manpage has a lot of detail on how to use `ofproto/trace`, but let's just start by building up from a simple example. You can start with a command that just specifies the datapath (e.g. `br0`), an input port, and nothing else; unspecified fields default to all-zeros. Let's look at the full output for this trivial example:

```
$ ovs-appctl ofproto/trace br0 in_port=p1
Flow: in_port=1,vlan_tci=0x0000,dl_src=00:00:00:00:00:00,dl_dst=00:00:00:00:00:00,dl_typ

bridge("br0")
-------------
 0. in_port=1, priority 9099, cookie 0x5adc15c0
    goto_table:1
 1. in_port=1,vlan_tci=0x0000/0x1fff, priority 9000, cookie 0x5adc15c0
    push_vlan:0x8100
    set_field:4196->vlan_vid
    goto_table:3
 3. priority 9000, cookie 0x5adc15c0
    CONTROLLER:96
    goto_table:7
 7. priority 9000, cookie 0x5adc15c0
    goto_table:8
 8. in_port=1,dl_vlan=100, priority 9000, cookie 0x5adc15c0
    pop_vlan
    output:2
    output:3

Final flow: unchanged
Megaflow: recirc_id=0,eth,in_port=1,vlan_tci=0x0000,dl_src=00:00:00:00:00:00,dl_dst=00:0
Datapath actions: push_vlan(vid=100,pcp=0),pop_vlan,2,3
This flow is handled by the userspace slow path because it:
        - Sends "packet-in" messages to the OpenFlow controller.
```

The first line of output, beginning with `Flow:`, just repeats our request in a more verbose form, including the L2 fields that were zeroed.

Each of the numbered items under `bridge("br0")` shows what would happen to our hypothetical packet in the table with the given number. For example, we see in table 1 that the packet matches a flow that push on a VLAN header, set the VLAN ID to 100, and goes on to further processing in table 3. In table 3, the packet gets sent to the controller to allow MAC learning to take place, and then table 8 floods the packet to the other ports in the same VLAN.

Summary information follows the numbered tables. The packet hasn't been changed (overall, even though a VLAN was pushed and then popped back off) since ingress, hence `Final flow: unchanged`. We'll look at the `Megaflow` information later. The `Datapath actions` summarize what would actually happen to such a packet. Finally, the note at the end gives a hint that this flow would not perform well for large volumes of traffic, because it has to be handled in the switch's slow path since it sends OpenFlow messages to the controller.

> ***Note***
>
> This performance limitation is probably not problematic in this case because it is only used for MAC learning, so that most packets won't encounter it. However, the Open vSwitch 2.9 release (which is upcoming as of this writing) will likely remove this performance limitation anyway.

Triggering MAC Learning

We just saw how a packet gets sent to the controller to trigger MAC learning. Let's actually send the packet and see what happens. But before we do that, let's save a copy of the current flow tables for later comparison:

```
$ save-flows br0 > flows1
```

Now use `ofproto/trace`, as before, with a few new twists: we specify the source and destination Ethernet addresses and append the `-generate` option so that side effects like sending a packet to the controller actually happen:

```
$ ovs-appctl ofproto/trace br0 in_port=p1,dl_src=00:11:11:00:00:00,dl_dst=00:22:22:00:00
```

The output is almost identical to that before, so it is not repeated here. But, take a look at `inst/faucet.log` now. It should now include a line at the end that says that it learned about our MAC 00:11:11:00:00:00, like this:

```
Oct 15 01:16:23 faucet.valve INFO     DPID 1 (0x1) learned 00:11:11:00:00:00 on Port 1 c
```

Now compare the flow tables that we saved to the current ones:

```
diff-flows flows1 br0
```

The result should look like this, showing new flows for the learned MACs:

```
+table=3 priority=9098,in_port=p1,dl_vlan=100,dl_src=00:11:11:00:00:00 cookie=0x5adc15c0
+table=7 priority=9099,dl_vlan=100,dl_dst=00:11:11:00:00:00 cookie=0x5adc15c0 idle_timeo
```

To demonstrate the usefulness of the learned MAC, try tracing (with side effects) a packet arriving on `p2` (or `p3`) and destined to the address learned on `p1`, like this:

```
$ ovs-appctl ofproto/trace br0 in_port=p2,dl_src=00:22:22:00:00:00,dl_dst=00:11:11:00:00
```

The first time you run this command, you will notice that it sends the packet to the controller, to learn `p2`'s 00:22:22:00:00:00 source address:

```
bridge("br0")
-------------
 0. in_port=2, priority 9099, cookie 0x5adc15c0
    goto_table:1
 1. in_port=2,vlan_tci=0x0000/0x1fff, priority 9000, cookie 0x5adc15c0
    push_vlan:0x8100
    set_field:4196->vlan_vid
```

```
    goto_table:3
 3. priority 9000, cookie 0x5adc15c0
    CONTROLLER:96
    goto_table:7
 7. dl_vlan=100,dl_dst=00:11:11:00:00:00, priority 9099, cookie 0x5adc15c0
    pop_vlan
    output:1
```

If you check `inst/faucet.log`, you can see that p2's MAC has been learned too:

```
Oct 15 01:24:01 faucet.valve INFO     DPID 1 (0x1) learned 00:22:22:00:00:00 on Port 2 c
```

Similarly for `diff-flows`:

```
$ diff-flows flows1 br0
+table=3 priority=9098,in_port=p1,dl_vlan=100,dl_src=00:11:11:00:00:00 cookie=0x5adc15c0
+table=3 priority=9098,in_port=p2,dl_vlan=100,dl_src=00:22:22:00:00:00 cookie=0x5adc15c0
+table=7 priority=9099,dl_vlan=100,dl_dst=00:11:11:00:00:00 cookie=0x5adc15c0 idle_timeo
+table=7 priority=9099,dl_vlan=100,dl_dst=00:22:22:00:00:00 cookie=0x5adc15c0 idle_timeo
```

Then, if you re-run either of the `ofproto/trace` commands (with or without `-generate`), you can see that the packets go back and forth without any further MAC learning, e.g.:

```
$ ovs-appctl ofproto/trace br0 in_port=p2,dl_src=00:22:22:00:00:00,dl_dst=00:11:11:00:00
Flow: in_port=2,vlan_tci=0x0000,dl_src=00:22:22:00:00:00,dl_dst=00:11:11:00:00:00,dl_typ

bridge("br0")
-------------
 0. in_port=2, priority 9099, cookie 0x5adc15c0
    goto_table:1
 1. in_port=2,vlan_tci=0x0000/0x1fff, priority 9000, cookie 0x5adc15c0
    push_vlan:0x8100
    set_field:4196->vlan_vid
    goto_table:3
 3. in_port=2,dl_vlan=100,dl_src=00:22:22:00:00:00, priority 9098, cookie 0x5adc15c0
    goto_table:7
 7. dl_vlan=100,dl_dst=00:11:11:00:00:00, priority 9099, cookie 0x5adc15c0
    pop_vlan
    output:1

Final flow: unchanged
Megaflow: recirc_id=0,eth,in_port=2,vlan_tci=0x0000/0x1fff,dl_src=00:22:22:00:00:00,dl_d
```

Performance

We've already seen one factor that can be important for performance: Open vSwitch forces any flow that sends a packet to an OpenFlow controller into its "slow path", which means that processing packets in the flow will be orders of magnitude slower than otherwise. This distinction between "slow path" and "fast path" is the key to making sure that Open vSwitch performs as fast as possible.

In addition to sending packets to a controller, some other factors can force a flow or a packet to take the slow path. As one example, all CFM, BFD, LACP, STP, and LLDP processing takes place in the slow path, in the cases where Open vSwitch processes these protocols itself instead of delegating to controller-written flows. As a second example, any flow that modifies ARP fields is processed in the slow path. These are corner cases that are unlikely to cause performance problems in practice because these

protocols send packets at a relatively slow rate, and users and controller authors do not normally need to be concerned about them.

To understand what cases users and controller authors should consider, we need to talk about how Open vSwitch optimizes for performance. The Open vSwitch code is divided into two major components which, as already mentioned, are called the "slow path" and "fast path" (aka "datapath"). The slow path is embedded in the `ovs-vswitchd` userspace program. It is the part of the Open vSwitch packet processing logic that understands OpenFlow. Its job is to take a packet and run it through the OpenFlow tables to determine what should happen to it. It outputs a list of actions in a form similar to OpenFlow actions but simpler, called "ODP actions" or "datapath actions". It then passes the ODP actions to the datapath, which applies them to the packet.

> ***Note***
>
> Open vSwitch contains a single slow path and multiple fast paths. The difference between using Open vSwitch with the Linux kernel versus with DPDK is the datapath.

If every packet passed through the slow path and the fast path in this way, performance would be terrible. The key to getting high performance from this architecture is caching. Open vSwitch includes a multi-level cache. It works like this:

1. A packet initially arrives at the datapath. Some datapaths (such as DPDK and the in-tree version of the OVS kernel module) have a first-level cache called the "microflow cache". The microflow cache is the key to performance for relatively long-lived, high packet rate flows. If the datapath has a microflow cache, then it consults it and, if there is a cache hit, the datapath executes the associated actions. Otherwise, it proceeds to step 2.
2. The datapath consults its second-level cache, called the "megaflow cache". The megaflow cache is the key to performance for shorter or low packet rate flows. If there is a megaflow cache hit, the datapath executes the associated actions. Otherwise, it proceeds to step 3.
3. The datapath passes the packet to the slow path, which runs it through the OpenFlow table to yield ODP actions, a process that is often called "flow translation". It then passes the packet back to the datapath to execute the actions and to, if possible, install a megaflow cache entry so that subsequent similar packets can be handled directly by the fast path. (We already described above most of the cases where a cache entry cannot be installed.)

The megaflow cache is the key cache to consider for performance tuning. Open vSwitch provides tools for understanding and optimizing its behavior. The `ofproto/trace` command that we have already been using is the most common tool for this use. Let's take another look at the most recent `ofproto/trace` output:

```
$ ovs-appctl ofproto/trace br0 in_port=p2,dl_src=00:22:22:00:00:00,dl_dst=00:11:11:00:00
Flow: in_port=2,vlan_tci=0x0000,dl_src=00:22:22:00:00:00,dl_dst=00:11:11:00:00:00,dl_typ

bridge("br0")
-------------
 0. in_port=2, priority 9099, cookie 0x5adc15c0
    goto_table:1
 1. in_port=2,vlan_tci=0x0000/0x1fff, priority 9000, cookie 0x5adc15c0
    push_vlan:0x8100
    set_field:4196->vlan_vid
    goto_table:3
 3. in_port=2,dl_vlan=100,dl_src=00:22:22:00:00:00, priority 9098, cookie 0x5adc15c0
    goto_table:7
 7. dl_vlan=100,dl_dst=00:11:11:00:00:00, priority 9099, cookie 0x5adc15c0
```

```
    pop_vlan
    output:1

Final flow: unchanged
Megaflow: recirc_id=0,eth,in_port=2,vlan_tci=0x0000/0x1fff,dl_src=00:22:22:00:00:00,dl_d
```

This time, it's the last line that we're interested in. This line shows the entry that Open vSwitch would insert into the megaflow cache given the particular packet with the current flow tables. The megaflow entry includes:

- `recirc_id`. This is an implementation detail that users don't normally need to understand.
- `eth`. This just indicates that the cache entry matches only Ethernet packets; Open vSwitch also supports other types of packets, such as IP packets not encapsulated in Ethernet.
- All of the fields matched by any of the flows that the packet visited:

  `in_port`
  : In tables 0, 1, and 3.

  `vlan_tci`
  : In tables 1, 3, and 7 (`vlan_tci` includes the VLAN ID and PCP fields and``dl_vlan`` is just the VLAN ID).

  `dl_src`
  : In table 3

  `dl_dst`
  : In table 7.

- All of the fields matched by flows that had to be ruled out to ensure that the ones that actually matched were the highest priority matching rules.

The last one is important. Notice how the megaflow matches on `dl_type=0x0000`, even though none of the tables matched on `dl_type` (the Ethernet type). One reason is because of this flow in OpenFlow table 1 (which shows up in `dump-flows` output):

```
table=1, priority=9099,dl_type=0x88cc actions=drop
```

This flow has higher priority than the flow in table 1 that actually matched. This means that, to put it in the megaflow cache, `ovs-vswitchd` has to add a match on `dl_type` to ensure that the cache entry doesn't match LLDP packets (with Ethertype 0x88cc).

> ***Note***
>
> In fact, in some cases `ovs-vswitchd` matches on fields that aren't strictly required according to this description. `dl_type` is actually one of those, so deleting the LLDP flow probably would not have any effect on the megaflow. But the principle here is sound.

So why does any of this matter? It's because, the more specific a megaflow is, that is, the more fields or bits within fields that a megaflow matches, the less valuable it is from a caching viewpoint. A very specific megaflow might match on L2 and L3 addresses and L4 port numbers. When that happens, only packets in one (half-)connection match the megaflow. If that connection has only a few packets, as many connections do, then the high cost of the slow path translation is amortized over only a few packets, so the average cost of forwarding those packets is high. On the other hand, if a megaflow only matches a relatively small number of L2 and L3 packets, then the cache entry can potentially be used by many individual connections, and the average cost is low.

For more information on how Open vSwitch constructs megaflows, including about ways that it can make megaflow entries less specific than one would infer from the discussion here, please refer to the 2015 NSDI paper, "The Design and Implementation of Open vSwitch", which focuses on this algorithm.

Routing

We've looked at how Faucet implements switching in OpenFlow, and how Open vSwitch implements OpenFlow through its datapath architecture. Now let's start over, adding L3 routing into the picture.

It's remarkably easy to enable routing. We just change our `vlans` section in `inst/faucet.yaml` to specify a router IP address for each VLAN and define a router between them. The `dps` section is unchanged:

```
dps:
    switch-1:
        dp_id: 0x1
        timeout: 3600
        arp_neighbor_timeout: 3600
        interfaces:
            1:
                native_vlan: 100
            2:
                native_vlan: 100
            3:
                native_vlan: 100
            4:
                native_vlan: 200
            5:
                native_vlan: 200
vlans:
    100:
        faucet_vips: ["10.100.0.254/24"]
    200:
        faucet_vips: ["10.200.0.254/24"]
routers:
    router-1:
        vlans: [100, 200]
```

Then we restart Faucet:

```
$ docker restart faucet
```

> ***Note***
>
> One should be able to tell Faucet to re-read its configuration file without restarting it. I sometimes saw anomalous behavior when I did this, although I didn't characterize it well enough to make a quality bug report. I found restarting the container to be reliable.

OpenFlow Layer

Back in the OVS sandbox, let's see how the flow table has changed, with:

```
$ diff-flows flows1 br0
```

First, table 3 has new flows to direct ARP packets to table 6 (the virtual IP processing table), presumably to handle ARP for the router IPs. New flows also send IP packets destined to a particular Ethernet address to table 4 (the L3 forwarding table); we can make the educated guess that the Ethernet address is the one used by the Faucet router:

```
+table=3 priority=9131,arp,dl_vlan=100 actions=resubmit(,6)
+table=3 priority=9131,arp,dl_vlan=200 actions=resubmit(,6)
+table=3 priority=9099,ip,dl_vlan=100,dl_dst=0e:00:00:00:00:01 actions=resubmit(,4)
+table=3 priority=9099,ip,dl_vlan=200,dl_dst=0e:00:00:00:00:01 actions=resubmit(,4)
```

The new flows in table 4 appear to be verifying that the packets are indeed addressed to a network or IP address that Faucet knows how to route:

```
+table=4 priority=9131,ip,dl_vlan=100,nw_dst=10.100.0.254 actions=resubmit(,6)
+table=4 priority=9131,ip,dl_vlan=200,nw_dst=10.200.0.254 actions=resubmit(,6)
+table=4 priority=9123,ip,dl_vlan=200,nw_dst=10.100.0.0/24 actions=resubmit(,6)
+table=4 priority=9123,ip,dl_vlan=100,nw_dst=10.100.0.0/24 actions=resubmit(,6)
+table=4 priority=9123,ip,dl_vlan=200,nw_dst=10.200.0.0/24 actions=resubmit(,6)
+table=4 priority=9123,ip,dl_vlan=100,nw_dst=10.200.0.0/24 actions=resubmit(,6)
```

Table 6 has a few different things going on. It sends ARP requests for the router IPs to the controller; presumably the controller will generate replies and send them back to the requester. It switches other ARP packets, either broadcasting them if they have a broadcast destination or attempting to unicast them otherwise. It sends all other IP packets to the controller:

```
+table=6 priority=9133,arp,arp_tpa=10.100.0.254 actions=CONTROLLER:96
+table=6 priority=9133,arp,arp_tpa=10.200.0.254 actions=CONTROLLER:96
+table=6 priority=9132,arp,dl_dst=ff:ff:ff:ff:ff:ff actions=resubmit(,8)
+table=6 priority=9131,arp actions=resubmit(,7)
+table=6 priority=9131,ip actions=CONTROLLER:96
+table=6 priority=9131,icmp actions=CONTROLLER:96
```

Note

There's one oddity here in that ICMP packets can match either the `ip` or `icmp` entry, which both have priority 9131. OpenFlow says, "If there are multiple matching flow entries with the same highest priority, the selected flow entry is explicitly undefined." In this case, it probably doesn't matter, since both flows have the same actions, but if Faucet wants to keep track of ICMP statistics separately from other IP packets, then it should install the `ip` flow with a lower priority than the `icmp` flow.

Performance is clearly going to be poor if every packet that needs to be routed has to go to the controller, but it's unlikely that's the full story. In the next section, we'll take a closer look.

Tracing

As in our switching example, we can play some "what-if?" games to figure out how this works. Let's suppose that a machine with IP 10.100.0.1, on port `p1`, wants to send a IP packet to a machine with IP 10.200.0.1 on port `p4`. Assuming that these hosts have not been in communication recently, the steps to accomplish this are normally the following:

1. Host 10.100.0.1 sends an ARP request to router 10.100.0.254.
2. The router sends an ARP reply to the host.
3. Host 10.100.0.1 sends an IP packet to 10.200.0.1, via the router's Ethernet address.
4. The router broadcasts an ARP request to `p4` and `p5`, the ports that carry the 10.200.0.<x> network.
5. Host 10.200.0.1 sends an ARP reply to the router.
6. Either the router sends the IP packet (which it buffered) to 10.200.0.1, or eventually 10.100.0.1 times out and resends it.

Let's use `ofproto/trace` to see whether Faucet and OVS follow this procedure.

Before we start, save a new snapshot of the flow tables for later comparison:

```
$ save-flows br0 > flows2
```

Step 1: Host ARP for Router

Let's simulate the ARP from 10.100.0.1 to its gateway router 10.100.0.254. This requires more detail than any of the packets we've simulated previously:

```
$ ovs-appctl ofproto/trace br0 in_port=p1,dl_src=00:01:02:03:04:05,dl_dst=ff:ff:ff:ff:ff
```

The important part of the output is where it shows that the packet was recognized as an ARP request destined to the router gateway and therefore sent to the controller:

```
6. arp,arp_tpa=10.100.0.254, priority 9133, cookie 0x5adc15c0
   CONTROLLER:96
```

The Faucet log shows that Faucet learned the host's MAC address, its MAC-to-IP mapping, and responded to the ARP request:

```
Oct 15 19:01:16 faucet.valve INFO     DPID 1 (0x1) Adding new route 10.100.0.1/32 via 10
Oct 15 19:01:16 faucet.valve INFO     DPID 1 (0x1) Responded to ARP request for 10.100.0
Oct 15 19:01:16 faucet.valve INFO     DPID 1 (0x1) learned 00:01:02:03:04:05 on Port 1 o
```

We can also look at the changes to the flow tables:

```
$ diff-flows flows2 br0
+table=3 priority=9098,in_port=1,dl_vlan=100,dl_src=00:01:02:03:04:05 hard_timeout=3600
+table=4 priority=9131,ip,dl_vlan=100,nw_dst=10.100.0.1 actions=set_field:4196->vlan_vid
+table=4 priority=9131,ip,dl_vlan=200,nw_dst=10.100.0.1 actions=set_field:4196->vlan_vid
+table=7 priority=9099,dl_vlan=100,dl_dst=00:01:02:03:04:05 idle_timeout=3600 actions=po
```

The new flows include one in table 3 and one in table 7 for the learned MAC, which have the same forms we saw before. The new flows in table 4 are different. They matches packets directed to 10.100.0.1 (in two VLANs) and forward them to the host by updating the Ethernet source and destination addresses appropriately, decrementing the TTL, and skipping ahead to unicast output in table 7. This means that packets sent **to** 10.100.0.1 should now get to their destination.

Step 2: Router Sends ARP Reply

`inst/faucet.log` said that the router sent an ARP reply. How can we see it? Simulated packets just get dropped by default. One way is to configure the dummy ports to write the packets they receive to a file. Let's try that. First configure the port:

```
$ ovs-vsctl set interface p1 options:pcap=p1.pcap
```

Then re-run the "trace" command:

```
$ ovs-appctl ofproto/trace br0 in_port=p1,dl_src=00:01:02:03:04:05,dl_dst=ff:ff:ff:ff:ff
```

And dump the reply packet:

```
$ /usr/sbin/tcpdump -evvvr sandbox/p1.pcap
reading from file sandbox/x.pcap, link-type EN10MB (Ethernet)
15:34:14.172222 0e:00:00:00:00:01 (oui Unknown) > 00:01:02:03:04:05 (oui Unknown), ether
```

We clearly see the ARP reply, which tells us that the Faucet router's Ethernet address is 0e:00:00:00:00:01 (as we guessed before from the flow table.

Let's configure the rest of our ports to log their packets, too:

```
$ for i in 2 3 4 5; do ovs-vsctl set interface p$i options:pcap=p$i.pcap; done
```

Step 3: Host Sends IP Packet

Now that host 10.100.0.1 has the MAC address for its router, it can send an IP packet to 10.200.0.1 via the router's MAC address, like this:

```
$ ovs-appctl ofproto/trace br0 in_port=p1,dl_src=00:01:02:03:04:05,dl_dst=0e:00:00:00:00
Flow: ip,in_port=1,vlan_tci=0x0000,dl_src=00:01:02:03:04:05,dl_dst=0e:00:00:00:00:01,nw_

bridge("br0")
-------------
 0. in_port=1, priority 9099, cookie 0x5adc15c0
    goto_table:1
 1. in_port=1,vlan_tci=0x0000/0x1fff, priority 9000, cookie 0x5adc15c0
    push_vlan:0x8100
    set_field:4196->vlan_vid
    goto_table:3
 3. ip,dl_vlan=100,dl_dst=0e:00:00:00:00:01, priority 9099, cookie 0x5adc15c0
    goto_table:4
 4. ip,dl_vlan=100,nw_dst=10.200.0.0/24, priority 9123, cookie 0x5adc15c0
    goto_table:6
 6. ip, priority 9131, cookie 0x5adc15c0
    CONTROLLER:96

Final flow: ip,in_port=1,dl_vlan=100,dl_vlan_pcp=0,vlan_tci1=0x0000,dl_src=00:01:02:03:0
Megaflow: recirc_id=0,eth,ip,in_port=1,vlan_tci=0x0000/0x1fff,dl_src=00:01:02:03:04:05,d
Datapath actions: push_vlan(vid=100,pcp=0)
This flow is handled by the userspace slow path because it:
        - Sends "packet-in" messages to the OpenFlow controller.
```

Observe that the packet gets recognized as destined to the router, in table 3, and then as properly destined to the 10.200.0.0/24 network, in table 4. In table 6, however, it gets sent to the controller. Presumably, this is because Faucet has not yet resolved an Ethernet address for the destination host 10.200.0.1. It probably sent out an ARP request. Let's take a look in the next step.

Step 4: Router Broadcasts ARP Request

The router needs to know the Ethernet address of 10.200.0.1. It knows that, if this machine exists, it's on port `p4` or `p5`, since we configured those ports as VLAN 200.

Let's make sure:

```
$ /usr/sbin/tcpdump -evvvr sandbox/p4.pcap
reading from file sandbox/p4.pcap, link-type EN10MB (Ethernet)
15:55:42.977504 0e:00:00:00:00:01 (oui Unknown) > Broadcast, ethertype ARP (0x0806), len
```

and:

```
$ /usr/sbin/tcpdump -evvvr sandbox/p5.pcap
reading from file sandbox/p5.pcap, link-type EN10MB (Ethernet)
15:55:42.977568 0e:00:00:00:00:01 (oui Unknown) > Broadcast, ethertype ARP (0x0806), len
```

For good measure, let's make sure that it wasn't sent to `p3`:

```
$ /usr/sbin/tcpdump -evvvr sandbox/p3.pcap
reading from file sandbox/p3.pcap, link-type EN10MB (Ethernet)
```

Step 5: Host 2 Sends ARP Reply

The Faucet controller sent an ARP request, so we can send an ARP reply:

```
$ ovs-appctl ofproto/trace br0 in_port=p4,dl_src=00:10:20:30:40:50,dl_dst=0e:00:00:00:00
Flow: arp,in_port=4,vlan_tci=0x0000,dl_src=00:10:20:30:40:50,dl_dst=0e:00:00:00:00:01,ar

bridge("br0")
-------------
 0. in_port=4, priority 9099, cookie 0x5adc15c0
    goto_table:1
 1. in_port=4,vlan_tci=0x0000/0x1fff, priority 9000, cookie 0x5adc15c0
    push_vlan:0x8100
    set_field:4296->vlan_vid
    goto_table:3
 3. arp,dl_vlan=200, priority 9131, cookie 0x5adc15c0
    goto_table:6
 6. arp,arp_tpa=10.200.0.254, priority 9133, cookie 0x5adc15c0
    CONTROLLER:96

Final flow: arp,in_port=4,dl_vlan=200,dl_vlan_pcp=0,vlan_tci1=0x0000,dl_src=00:10:20:30:
Megaflow: recirc_id=0,eth,arp,in_port=4,vlan_tci=0x0000/0x1fff,dl_src=00:10:20:30:40:50,
Datapath actions: push_vlan(vid=200,pcp=0)
This flow is handled by the userspace slow path because it:
        - Sends "packet-in" messages to the OpenFlow controller.
```

It shows up in `inst/faucet.log`:

```
Oct 16 23:02:22 faucet.valve INFO     DPID 1 (0x1) ARP response 10.200.0.1 (00:10:20:30:
Oct 16 23:02:22 faucet.valve INFO     DPID 1 (0x1) learned 00:10:20:30:40:50 on Port 4 o
```

and in the OVS flow tables:

```
$ diff-flows flows2 br0
+table=3 priority=9098,in_port=4,dl_vlan=200,dl_src=00:10:20:30:40:50 hard_timeout=295 a
...
+table=4 priority=9131,ip,dl_vlan=200,nw_dst=10.200.0.1 actions=set_field:4296->vlan_vid
+table=4 priority=9131,ip,dl_vlan=100,nw_dst=10.200.0.1 actions=set_field:4296->vlan_vid
...
+table=4 priority=9123,ip,dl_vlan=100,nw_dst=10.200.0.0/24 actions=goto_table:6
+table=7 priority=9099,dl_vlan=200,dl_dst=00:10:20:30:40:50 idle_timeout=295 actions=pop
```

Step 6: IP Packet Delivery

Now both the host and the router have everything they need to deliver the packet. There are two ways it might happen. If Faucet's router is smart enough to buffer the packet that trigger ARP resolution, then it might have delivered it already. If so, then it should show up in `p4.pcap`. Let's take a look:

```
$ /usr/sbin/tcpdump -evvvr sandbox/p4.pcap ip
reading from file sandbox/p4.pcap, link-type EN10MB (Ethernet)
```

Nope. That leaves the other possibility, which is that Faucet waits for the original sending host to re-send the packet. We can do that by re-running the trace:

```
$ ovs-appctl ofproto/trace br0 in_port=p1,dl_src=00:01:02:03:04:05,dl_dst=0e:00:00:00:00
Flow: udp,in_port=1,vlan_tci=0x0000,dl_src=00:01:02:03:04:05,dl_dst=0e:00:00:00:00:01,nw

bridge("br0")
-------------
 0. in_port=1, priority 9099, cookie 0x5adc15c0
    goto_table:1
 1. in_port=1,vlan_tci=0x0000/0x1fff, priority 9000, cookie 0x5adc15c0
    push_vlan:0x8100
    set_field:4196->vlan_vid
    goto_table:3
 3. ip,dl_vlan=100,dl_dst=0e:00:00:00:00:01, priority 9099, cookie 0x5adc15c0
    goto_table:4
 4. ip,dl_vlan=100,nw_dst=10.200.0.1, priority 9131, cookie 0x5adc15c0
    set_field:4296->vlan_vid
    set_field:0e:00:00:00:00:01->eth_src
    set_field:00:10:20:30:40:50->eth_dst
    dec_ttl
    goto_table:7
 7. dl_vlan=200,dl_dst=00:10:20:30:40:50, priority 9099, cookie 0x5adc15c0
    pop_vlan
    output:4

Final flow: udp,in_port=1,vlan_tci=0x0000,dl_src=0e:00:00:00:00:01,dl_dst=00:10:20:30:40
Megaflow: recirc_id=0,eth,ip,in_port=1,vlan_tci=0x0000/0x1fff,dl_src=00:01:02:03:04:05,d
Datapath actions: set(eth(src=0e:00:00:00:00:01,dst=00:10:20:30:40:50)),set(ipv4(dst=10.
```

Finally, we have working IP packet forwarding!

Performance

Take another look at the megaflow line above:

```
Megaflow: recirc_id=0,eth,ip,in_port=1,vlan_tci=0x0000/0x1fff,dl_src=00:01:02:03:04:05,d
```

This means that (almost) any packet between these Ethernet source and destination hosts, destined to the given IP host, will be handled by this single megaflow cache entry. So regardless of the number of UDP packets or TCP connections that these hosts exchange, Open vSwitch packet processing won't need to fall back to the slow path. It is quite efficient.

> ***Note***
>
> The exceptions are packets with a TTL other than 64, and fragmented packets. Most hosts use a constant TTL for outgoing packets, and fragments are rare. If either of those did change, then that would simply result in a new megaflow cache entry.

The datapath actions might also be worth a look:

```
Datapath actions: set(eth(src=0e:00:00:00:00:01,dst=00:10:20:30:40:50)),set(ipv4(dst=10.
```

This just means that, to process these packets, the datapath changes the Ethernet source and destination addresses and the IP TTL, and then transmits the packet to port `p4` (also numbered 4). Notice in particular that, despite the OpenFlow actions that pushed, modified, and popped back off a VLAN, there is nothing in the datapath actions about VLANs. This is because the OVS flow translation code "optimizes out" redundant or unneeded actions, which saves time when the cache entry is executed later.

> ***Note***
>
> It's not clear why the actions also re-set the IP destination address to its original value. Perhaps this is a minor performance bug.

ACLs

Let's try out some ACLs, since they do a good job illustrating some of the ways that OVS tries to optimize megaflows. Update `inst/faucet.yaml` to the following:

```
dps:
    switch-1:
        dp_id: 0x1
        timeout: 3600
        arp_neighbor_timeout: 3600
        interfaces:
            1:
                native_vlan: 100
                acl_in: 1
            2:
                native_vlan: 100
            3:
```

```
                native_vlan: 100
            4:
                native_vlan: 200
            5:
                native_vlan: 200
vlans:
    100:
        faucet_vips: ["10.100.0.254/24"]
    200:
        faucet_vips: ["10.200.0.254/24"]
routers:
    router-1:
        vlans: [100, 200]
acls:
    1:
        - rule:
            dl_type: 0x800
            nw_proto: 6
            tp_dst: 8080
            actions:
                allow: 0
        - rule:
            actions:
                allow: 1
```

Then restart Faucet:

```
$ docker restart faucet
```

On port 1, this new configuration blocks all traffic to TCP port 8080 and allows all other traffic. The resulting change in the flow table shows this clearly too:

```
$ diff-flows flows2 br0
-priority=9099,in_port=1 actions=goto_table:1
+priority=9098,in_port=1 actions=goto_table:1
+priority=9099,tcp,in_port=1,tp_dst=8080 actions=drop
```

The most interesting question here is performance. If you recall the earlier discussion, when a packet through the flow table encounters a match on a given field, the resulting megaflow has to match on that field, even if the flow didn't actually match. This is expensive.

In particular, here you can see that any TCP packet is going to encounter the ACL flow, even if it is directed to a port other than 8080. If that means that every megaflow for a TCP packet is going to have to match on the TCP destination, that's going to be bad for caching performance because there will be a need for a separate megaflow for every TCP destination port that actually appears in traffic, which means a lot more megaflows than otherwise. (Really, in practice, if such a simple ACL blew up performance, OVS wouldn't be a very good switch!)

Let's see what happens, by sending a packet to port 80 (instead of 8080):

```
$ ovs-appctl ofproto/trace br0 in_port=p1,dl_src=00:01:02:03:04:05,dl_dst=0e:00:00:00:00

bridge("br0")
-------------
 0. in_port=1, priority 9098, cookie 0x5adc15c0
```

```
    goto_table:1
 1. in_port=1,vlan_tci=0x0000/0x1fff, priority 9000, cookie 0x5adc15c0
    push_vlan:0x8100
    set_field:4196->vlan_vid
    goto_table:3
 3. ip,dl_vlan=100,dl_dst=0e:00:00:00:00:01, priority 9099, cookie 0x5adc15c0
    goto_table:4
 4. ip,dl_vlan=100,nw_dst=10.200.0.0/24, priority 9123, cookie 0x5adc15c0
    goto_table:6
 6. ip, priority 9131, cookie 0x5adc15c0
    CONTROLLER:96

Final flow: tcp,in_port=1,dl_vlan=100,dl_vlan_pcp=0,vlan_tci1=0x0000,dl_src=00:01:02:03:
Megaflow: recirc_id=0,eth,tcp,in_port=1,vlan_tci=0x0000/0x1fff,dl_src=00:01:02:03:04:05,
Datapath actions: push_vlan(vid=100,pcp=0)
```

Take a look at the Megaflow line and in particular the match on `tp_dst`, which says `tp_dst=0x0/0xf000`. What this means is that the megaflow matches on only the top 4 bits of the TCP destination port. That works because:

```
  80 (base 10) == 0001,1111,1001,0000 (base 2)
8080 (base 10) == 0000,0000,0101,0000 (base 2)
```

and so by matching on only the top 4 bits, rather than all 16, the OVS fast path can distinguish port 80 from port 8080. This allows this megaflow to match one-sixteenth of the TCP destination port address space, rather than just 1/65536th of it.

> ***Note***
>
> The algorithm OVS uses for this purpose isn't perfect. In this case, a single-bit match would work (e.g. tp_dst=0x0/0x1000), and would be superior since it would only match half the port address space instead of one-sixteenth.

For details of this algorithm, please refer to `lib/classifier.c` in the Open vSwitch source tree, or our 2015 NSDI paper "The Design and Implementation of Open vSwitch".

Finishing Up

When you're done, you probably want to exit the sandbox session, with Control+D or `exit`, and stop the Faucet controller with `docker stop faucet; docker rm faucet`.

Further Directions

We've looked a fair bit at how Faucet interacts with Open vSwitch. If you still have some interest, you might want to explore some of these directions:

- Adding more than one switch. Faucet can control multiple switches but we've only been simulating one of them. It's easy enough to make a single OVS instance act as multiple switches (just `ovs-vsctl add-br` another bridge), or you could use genuinely separate OVS instances.
- Additional features. Faucet has more features than we've demonstrated, such as IPv6 routing and port mirroring. These should also interact gracefully with Open vSwitch.

- Real performance testing. We've looked at how flows and traces **should** demonstrate good performance, but of course there's no proof until it actually works in practice. We've also only tested with trivial configurations. Open vSwitch can scale to millions of OpenFlow flows, but the scaling in practice depends on the particular flow tables and traffic patterns, so it's valuable to test with large configurations, either in the way we've done it or with real traffic.