frenetic

Programmers Guide

Craig Riecke

with

Others

Draft of June 9, 2016

Contents

Chapter 1

Quick Start

In this book, you will use the open source software Frenetic to create a fully programmable network. For the moment, let's assume you're familiar with Software Defined Networking and the OpenFlow protocol, and just dive right in. (If you're not, don't worry! We'll introduce some bedrock concepts in the next chapter and explain everything that happened here.)

1.1 Installation

There are several ways to get started with Frenetic, but the easiest is to use Frenetic VM. Frenetic itself only runs on Linux, but the Frenetic VM will run on any host system that supports VirtualBox, including Windows, Mac OS X and practically any version of Linux. Keeping Frenetic it in its own VM will keep your own system clean and neat. Later on, if you want to install Frenetic on a bare metal Ubuntu Linux server or network device, you can use the instructions in 13.1.

First you'll need the following prerequisites:

- Install VirtualBox from https://www.virtualbox.org/wiki/Downloads. Use the latest version platform package appropriate for your system.
- Install Vagrant at http://www.vagrantup.com/downloads. Vagrant automates the process of building VM's from scratch, and Frenetic VM uses it to build its own environment.
- Install Frenetic VM from https://github.com/frenetic-lang/frenetic-vm. You can simply use the Download Zip button and unzip to an appropriate directory on your system, like frenetic-vm. Then from a terminal or command prompt:

```
$ cd /path/to/frenetic-vm
$ vagrant up
 ... lots of text
```

The build process may take 15 minutes to an hour, depending on the speed of your system and Internet connection.

1.2 What Do You Get With Frenetic VM?

At the end of the process you will have a working copy of Frenetic with lots of useful open source infrastructure:

Mininet software simulates a test network inside of Linux. It can model a topology with many switches and hosts. Writing a network application and throwing it into production is …well, pretty risky, but running it on Mininet first can be a good test for how it works beforehand.

Wireshark captures and analyzes network traffic. It's a great debugging tool, and very useful for sifting through piles of network packets.

Frenetic . This layer provides an easy-to-use programmable layer on top of ODL. Its main job is to shuttle OpenFlow messages between ODL and your application, and to translate the language NetKAT into OpenFlow flow tables. We'll see the differences between the two as we go.

Hmmm, that's a lot of software - what do *you* bring to the table? You write your network application in Python, using the Frenetic framework. As you'll see, it's quite easy to build a network device from scratch, and easy to grow it organically to fit your requirements. Python is fairly popular, and knowing it will give you a head start into Frenetic programming. But if you're a Python novice that's OK. As long as you know one object-oriented language fairly well, you should be able to follow the concepts. We'll introduce you to useful Python features, like list comprehensions, as we go.

1.3 An Attempt at Hello World

So let's dive right in. We'll set up a Mininet network with one switch and two hosts. First you should work from the directory where you installed Frenetic VM.

```
 $ cd /path/to/frenetic-vm
```

And start up the VM:

```
$ vagrant up
 ringing machine 'default' up with 'virtualbox' provider...
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 22 => 2222 (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection timeout. Retrying...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Setting hostname...
==> default: Mounting shared folders...
    default: /vagrant => /Users/cr396/frenetic-vm
    default: /home/vagrant/src => /Users/cr396/frenetic-vm/src
==> default: Machine already provisioned. Run `vagrant provision` or...
==> default: to force provisioning. Provisioners marked to run always...
```

Then log in to the VM. At this point your command prompt will change to `vagrant@frenetic` to distinguish it from your host machine.

```
 $ vagrant ssh
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.16.0-30-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Last login: Tue Oct  6 10:35:06 2015 from 10.0.2.2
vagrant@frenetic:~$
```

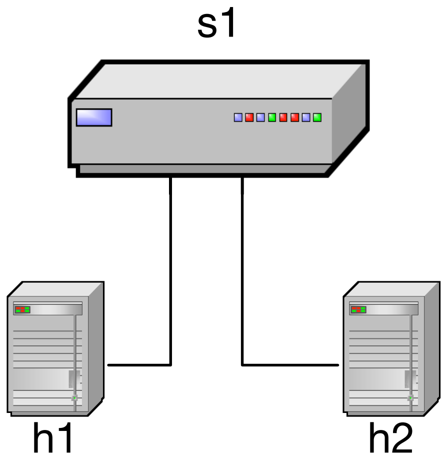
So you are now working inside an Ubuntu-based VM.

Let's start up a Mininet network with one switch and two nodes.

```
vagrant@frenetic:~$ sudo mn --topo=single,2 --controller=remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
```

```
s1 ...
*** Starting CLI:
mininet>
```

The topology Mininet constructs looks like this:



The prompt changes to `mininet>` to show your working in Mininet. The error message `Unable to contact controller at 127.0.0.1:6633` looks a little ominous, but it's not fatal.

You now have an experimental network with two hosts named h1 and h2. To see if there's connectivity between them, use the command `h1 ping h2` which means "On host h1, ping the host h2."

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
^C
--- 10.0.0.2 ping statistics ---
6 packets transmitted, 0 received, +3 errors, 100\% packet loss, time 5014ms
pipe 3
```

The ping gets executed over and over again, but the `Destination Host Unreachable` message shows it's clearly not working. So we press CTRL-C to stop and quit out of Mininet:

```
mininet> quit
```

So by default, hosts can't talk over the network to each other. We're going to fix that by writing a *network application*. Frenetic will act as the controller on the network, and the network application tells Frenetic how to behave.

1.4 A Repeater

You will write your network application in Python, using the Frenetic framework. Mininet is currently running in our VM under its own terminal window, and we can leave it like that. We'll do our programming in another window, so start up another one and log into our VM:

```
 $ cd /path/to/frenetic-vm
 $ vagrant ssh
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.16.0-30-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Last login: Tue Oct  6 10:35:06 2015 from 10.0.2.2
vagrant@frenetic:~$
```

Get the sample code from the Frenetic Github repository:

```
vagrant@frenetic:~$ git clone https://github.com/frenetic-lang/manual
Cloning into 'manual'...
remote: Counting objects: 102, done.
remote: Total 102 (delta 0), reused 0 (delta 0), pack-reused 102
Receiving objects: 100% (102/102), 1.46 MiB | 728.00 KiB/s, done.
Resolving deltas: 100% (47/47), done.
Checking connectivity... done.
vagrant@frenetic:~$ cd manual/programmers_guide
vagrant@frenetic:~/manual/programmers_guide$ cd code/quick_start
```

The following code is in quick_start/repeater.py:

```python
import frenetic
from frenetic.syntax import *

class RepeaterApp(frenetic.App):

  client_id = "quick_start"

  def connected(self):
    self.update( id >> SendToController("repeater_app") )

  def packet_in(self, dpid, port_id, payload):
    out_port_id = 2 if port_id == 1 else 1
    self.pkt_out(dpid, payload, SetPort(out_port_id) )

app = RepeaterApp()
app.start_event_loop()
```

Lines 1-2 are pretty much the same in every Frenetic network application. Similarly, lines 15-16 are similar in most cases. The meat of the application is an object class named RepeaterApp, whose base class is `frenetic.App`. A frenetic application can hook code into different points of the network event cycle. In our Repeater network app, the only two events we're interested in here are `connected`, which is fired when a switch connects for the first time to Frenetic, and `packet_in`, which is fired every time a packet bound for a controller arrives.

The code in `connected` is called a *handler* and here it merely directs the switch to send all packets to our application. The code in `packet_in` is also a handler, and here it implements a *repeater*. A repeater, sometimes called a *hub*, is the oldest type of network device. In a 2-port repeater, if a packet enters on port 1, it should get copied out to port 2. Conversely, if a packet enters on port 2, it should get copied out to port 1. If there were more ports in our switch, we'd write a more sophisticated repeater – one that outputs the packet to all ports except the one on which it arrived (called the *ingress port*). We'll do that in section 3.1

`pkt_out` is a method provided by Frenetic to actually send the packet out the switch. It takes three parameters: a switch, a packet, and a policy. Here the policy sends the packet out to port `out_port_id`.

1.5 Running The Repeater Application

So let's get this running in a lab setup. Three programs need to be running: Mininet, Frenetic, and our new Repeater application. For now, we'll run them in three separate command lines, having typed `vagrant ssh` in each to login to the VM.

In the first terminal window, we'll start up Frenetic:

```
vagrant@frenetic:~src/frenetic$ ./frenetic.native http-controller \
  > --verbosity debug
 [INFO] Calling create!
 [INFO] Current uid: 1000
 [INFO] Successfully launched OpenFlow controller with pid 3062
 [INFO] Connecting to first OpenFlow server socket
 [INFO] Failed to open socket to OpenFlow server: (Unix.Unix_error...
 [INFO] Retrying in 1 second
 [INFO] Successfully connected to first OpenFlow server socket
 [INFO] Connecting to second OpenFlow server socket
 [INFO] Successfully connected to second OpenFlow server socket
```

In the second, we'll start up Mininet with the same configuration as before:

```
vagrant@frenetic:~$ sudo mn --topo=single,2 --controller=remote
*** Creating network
*** Adding controller
```

The following will appear in your Frenetic window to show a connection has been made:

```
 [INFO] switch 1 connected
[DEBUG] Setting up flow table
+----------------------+
| 1 | Pattern | Action |
|----------------------|
|             |        |
+----------------------+
```

And in the third, we'll start our repeater application:

```
vagrant@frenetic:~/manual/code/quick_start$ python repeater.py
Starting the tornado event loop (does not return).
```

The following will appear in the Frenetic window.

```
 [INFO] GET /version
 [INFO] POST /quick_start/update_json
 [INFO] GET /quick_start/event
 [INFO] New client quick_start
[DEBUG] Installing policy
drop | port := pipe(repeater_app) | port := pipe(repeater_app)
[DEBUG] Setting up flow table
+---------------------------------------+
| 1 | Pattern | Action                  |
|---------------------------------------|
|             | Output(Controller(128)) |
+---------------------------------------+
```

And finally, we'll pop over to the Mininet window and try our connection test once more:

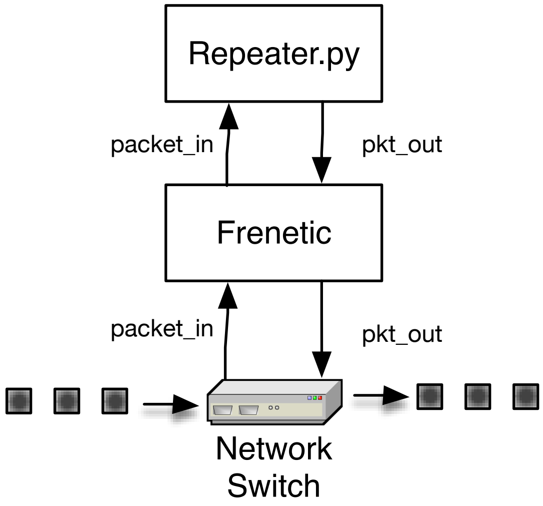
```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=149 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=97.2 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=88.7 ms
```

Ah, much better! Our pings are getting through. You can see evidence of this in the Frenetic window:

```
 [INFO] GET /quick_start/event
 [INFO] POST /pkt_out
[DEBUG] SENDING PKT_OUT
 [INFO] GET /quick_start/event
 [INFO] POST /pkt_out
[DEBUG] SENDING PKT_OUT
 [INFO] GET /quick_start/event
 [INFO] POST /pkt_out
[DEBUG] SENDING PKT_OUT
```

1.6 Summary

You now have a working SDN, or Software Defined Network! Like much software, it works in layers:



1. At the bottom is your switches and wires. In our lab setup, Mininet and OpenVSwitch is a substitute for this layer.

2. In the middle is Frenetic. It talks the OpenFlow protocol to the switches (or to Mininet) – this is called the Southbound interface. It also accepts its own language called NetKAT from network applications – this is called the Northbound interface.

3. At the very top is your network application, which you write in Python. It defines how packets are dealt with.

We wrote a very simple network application that emulates a network repeater. It responds to the `packet_in` event coming from the switches through Frenetic when a packet arrives at the switch. And it sends the `pkt_out` message to send the packet back out through Frenetic to the switch. When you're done, your network application can be deployed to a real production network.

Obviously you can do much more than just simple repeating with SDN! We'll cover that next with some background on OpenFlow and NetKAT, the underlying language of Frenetic.

Chapter 2

NetKAT

Software Defined Networking, or SDN, is a huge paradigm shift in the computing world. With traditional pre-SDN networking, you buy expensive, proprietary ”boxes” from major vendors, plugging them in, configuring them, and hoping they meet your needs. But traditional networking suffers from these maladies:

- The devices are flexible only within narrow configuration parameters. Special requirements, like preventing certain kinds of devices from mobility, or configuring the spanning tree to prefer certain paths, are either impossible or expensive.
- While the devices are powered by software, there’s no good way to examine the underlying code or prove it’s correct.
- Upgrades tend to be the forklift-variety, since mixing and matching old and new hardware is a dicey proposition . . . not to mention mixing hardware from different vendors.
- Configuration is not easily automated. Solutions are often proprietary, require special programming languages, and are not interchangeable. Because of this, modern data center virtualization is more difficult.
- Adding support for new protocols is slow and expensive.

With SDN, data centers can step off the proprietary network treadmill. Its a shift similar to the personal computer revolution of the 1980’s. Before that, IBM and similar mainframes controlled the computer space, and required the same kinds of forklift upgrades networks do. The IBM PC opened up the architecture to competitors, who could then design and build extensions that made it more useful. This created a snowball effect, with hardware extensions like the mouse and Ethernet cards opening the way for new software like Microsoft Windows and the Netscape browser.

SDN opens network devices in a similar manner, allowing them to be extended and manipulated in interesting, user-defined ways. Although the term SDN has been hijacked to mean many things, it most often refers to OpenFlow-based devices and related software. OpenFlow is an open protocol defined by the Open Network Foundation.

Frenetic is an OpenFlow controller, meaning it converses in the OpenFlow protocol to network switches. In turn, Frenetic exposes an API that can be used to write network programs easily. It works with any network switch that understands the OpenFlow 1.0 protocol – both hardware devices like the HP 2920 and software "devices" like Open vSwitch. So let's take a brief look at OpenFlow itself.

2.1 Introduction to OpenFlow

Every network device – from the lowliest repeater, to firewalls and load balancers, all the way up to the most complex router – has two conceptual layers:

The data plane performs actions on packets. It manipulates headers, copies packets to outgoing (or egress) ports, or drops packets. It consults control tables - MAC address tables, ARP caches, OSPF tables, etc. - to guide it.

The control plane manipulates the control tables themselves. People, in turn, may manipulate the control plane through configuration software. Or packets may do it: specialized ones like OSPF neighbor exchange, ARP requests, or just examining plain ol' packets themselves. But the control plane never actually touches the packets.

The OpenFlow protocol makes the control plane *programmable*. Rather than relying on the entire program being *inside* the box, you write a program that advises the control plane running *outside* the box. It's like an advisor that makes arbitrarily complex control table manipulations. The programmable piece runs on any standard computer, and is collectively called the *controller*.

The controller can be written in any language and run on any computer . . . the only requirement is it must speak the OpenFlow protocol to the device. You can think of these two pieces working in a tandem through an OpenFlow conversation:

Switch: I just got a packet, but I don't know what to do with it. It came in port 3 from Ethernet mac address 10:23:10:59:12:fb and it's going to mac address 5c:fb:12:59:10:23.

Controller: OK. I'll memorize that the 10:23:10:59:12:fb address is on port 3. But I don't know which port has a device with address 5c:fb:12:59:10:23. So just send it out all ports on the switch except port 3.

Switch: OK. . . . Ooops, here's another packet I don't know what to do with. It came in port 5 from Ethernet mac address 5c:fb:12:59:10:23 and it's going to mac address 10:23:10:59:12:fb.

Controller: Oh yeah. That looks like a reply. I'll memorize that the 5c:fb:12:59:10:23 address is on port 5. Meanwhile, I know the destination is on port 3. Install a rule so all packets going to that mac address go out port 3, then forward this packet out port 3 as well.

Switch: OK!

Controller: How many packets have went out port 3, by the way?

Switch: 82,120.

Switch: (To itself) I just saw a packet destined for Ethernet mac address 10:23:10:59:12:fb:5c, but I have a rule for dealing with it. I'm gonna send it out port 3.

OpenFlow boils down control plane functionality to a common core set of actions. A list of rules and actions that can be handled in the device itself are kept in a *flow table*. Any complex decisions that can't be handled independently by the control plane may be offloaded to the controller. In a well-designed Software Defined Network, the controller gets involved only when necessary. After all, the conversation between the device and the controller takes time, and anything that can eliminate this conversation makes the packets go faster. So in the example above, a rule for any packets going to 10:23:10:59:12:fb:5c to be output on port 5 keeps all the processing on the switch, and out of the controller. That makes it really fast.

So, central to the OpenFlow model is the *flow table*. Flow tables have *entries*, sometimes called *flow rules*, that are consulted for making decisions. A sample flow table might look like this:

Match	Actions	Priority

dl_src = 10:23:10:59:12:fb:5c,
dl_type = 0x806
	OFPAT_OUTPUT(9)	100

nw_src = 128.56.0.0/16,
dl_type = 0x800
	OFPAT_SET_DL_DST(5c:fb:12:59:10:23),
	OFPAT_OUTPUT(1)
		90

Wildcard	OFPAT_OUTPUT(Controller)	1

The main components of a flow entry are:

Matches specifies patterns of packet header and metadata values. OpenFlow 1.0 defines 12 different fields to match: some popular ones are the Input Port, the Ethernet Destination mac address, and the TCP destination port. The match values can either be exact (like 10:23:10:59:12:fb:5c above) or wild carded (like 128.56.0.0/16 for a particular Ip subnet).

Actions tell what to do if the match occurs, for example: send a packet out a port, or update packet header information, or send a packet to the controller.

Priorities define the order that matches are consulted. When more than one entry matches a particular packet, the entry with the highest priority wins.

In our example above, the controller installed a flow entry matching Ethernet Destination 10:23:10:59:12:fb:5c, and an instruction applying the action "Output it through Port 3".

Suppose you wanted to write your own controller from scratch. You could do that just by talking the OpenFlow protocol. Let's say you wrote this program in Node.js and placed it on the server "controller.example.com", listening on TCP port 6633. Then you'd just point your OpenFlow network device at controller.example.com:6633. Your program could install flow table entries into the network device over the OpenFlow protocol.

Hmmm. Sounds pretty easy, but . . .

2.2 OpenFlow is Difficult

From a programmer's perspective, a table looks awfully primitive. Tables are easy for switch hardware to interpret, and the faster they can interpret and carry out rules, the faster the packets travel. It's like machine language, where the CPU interprets simple instructions very quickly, and even in parallel.

Programming OpenFlow tables directly, you begin to find out the subtle missing details:

- You can only match packets with $=$. There's no $\neq$.
- There is an implicit AND in all match rows and an implicit OR between all rules. You cannot arbitrarily place AND's and OR's in match rules.
- You cannot match a field against a set of values. You have to write one rule per value.

Because tables often have thousands of rules, they are difficult to construct and debug. In the programming world, *modularization* aids both of these problems since smaller units of code are easier to understand.

OpenFlow tables have no inherent grouping mechanism, but we could simply modularize them by constructing small tables that do target packet processing. Smoosh them together into one big OpenFlow table when we're done, right?

But as the paper Foster et al. [2013] points out in section IIA, even simple modules can be difficult to *compose*. Suppose your SDN switch needed to do two things: repeat all traffic, but drop all HTTP packets coming from port 2 (a makeshift firewall). The repeater table might look something like this:

Match	Actions	Priority
in_port=1	OFPAT_OUTPUT(2)	200
in_port=2	OFPAT_OUTPUT(1)	100

And the firewall table might look like this:

Match	Actions	Priority

in_port = 2,
tp_src_port = 80,
dl_type = 0x800,
nw_proto = 0x1
	None	100

If we simply smooshed the two tables together, the firewall rule would never fire because the first rule in the repeater table overshadows it. In this case, reordering the priorities might work, but it's impossible to do this correctly without a spec to guide it.

Finally, it's difficult to reason about OpenFlow tables. While it's true that the set of possible packets is a finite set, it's still a large set. One could loop over all header values (200 bits worth in an OpenFlow 1.0 structured packet) and give the corresponding actions. But it's tough to actually enumerate all these cases.

Frenetic obeys mathematically-defined rules about packets, and its algorithms are provably correct, which you can see in the paper Smolka et al. [2015] And as outlined in Foster et al. [2015], you can prove properties like loop-freeness and connectivity about NetKAT programs.

2.3 Predicates

No matter what controller you use, underneath it all, you still have OpenFlow tables. So how does the Frenetic controller improve things?

Other SDN controllers like OpenDaylight, RYU, and Beacon force you to manipulate OpenFlow tables directly. Frenetic works at a higher abstraction level. Instead of writing programs that directly call OpenFlow primitives, you write programs in the NetKAT language.

Frenetic's main job is to compile NetKAT predicates and policies into OpenFlow flow tables. It directly communicates with the switch hardware (southbound) and to your application (northbound). Applications talk to Frenetic with standard HTTP, REST, and JSON. The JSON-based NetKAT dialect is available to anyone, and any programming language that can talk HTTP and JSON can talk to Frenetic. In this manual, we use the Python bindings for NetKAT because they're easy to use and extend, and they come bundled with Frenetic. This saves you from dealing with the esoterica of HTTP communication and JSON formatting.

So let's look at NetKAT predicates first. A *predicate* is a clause used to match packets. The base language is pretty straightforward:

SwitchEq(n)	Matches packets that arrive on switch n, where n is the Datapath ID of the switch.
PortEq(n)	Matches packets that arrive on port n. Generally ports are numbered 1-m, where m is the number of interfaces, but they don't need to be consecutive.
EthSrcEq(mac)	Matches packets whose Ethernet Mac source address is mac, which is a string in the standard form $nn : nn : nn : nn : nn : nn$ where the n's are lowercase hexadecimal digits.
EthDstEq(mac)	Matches packets whose Ethernet Mac destination address is mac.
VlanEq($vlan$)	Matches packets whose VLAN is $vlan$, and integer from 1-4096. Packets without a VLAN are never matched by this predicate.
VlanPcpEq(p)	Matches packets whose VLAN Priority Code Point is p. Packets without a VLAN are never matched by this predicate.
EthTypeEq(t)	Matches packets whose Ethernet Type is t, where t is a 32 bit integer. Popular values of t are 0x800 for IP and 0x806 for ARP.
IPProtoEq(p)	Matches packets whose IP Protocol is p, a number from 0-255. Popular values are 1 for ICMP, 6 for TCP and 17 for UDP.
IP4SrcEq($addr$, $mask$)	Matches packets whose IP source address is $addr$. If $mask$ is provided, a number from 1-32, this matches all hosts on a network whose first $mask$ bits match the host. If it's omitted, the entire address is matched – i.e. only one IP host.
IP4DstEq($addr$, $mask$)	Matches packets whose IP destination address is $addr$. Follows same rules as IpSrcEq.
TCPSrcPortEq(p)	Matches packets whose TCP source port is p, an integer from 0-65535.
TCPDstPortEq(p)	Matches packets whose TCP destination port is p, an integer from 0-65535. Popular values are 80 for HTTP, 22 for SSH, and 443 for SSL.

If you're familiar with OpenFlow, this list should look familiar to you – it's the same list of fields you can use in an OpenFlow flow table. One special case is SwitchEq, matching a switch id, which we'll talk about in a second.

Unlike OpenFlow matches, NetKAT predicates can contain a list of values. There is an explicit OR between the values, so only one value needs to match. The following are equivalent predicates (the | means OR, as we'll see shortly):

```
PortEq(1) | PortEq(2) | PortEq(3)
PortEq(1, 2, 3)
PortEq( [1,2,3] )
```

You can use boolean operators to combine predicates into bigger ones:

$p1$ & $p2$	Matches packets that satisfy $p1$ AND $p2$
And([$p1$, $p2$, . . . , pn])	Matches packets that satisfy all the predicates: $p1$ & $p2$ & . . . & pn
$p1 \mid p2$	Matches packets that satisfy $p1$ OR $p2$
Or([$p1$, $p2$, . . . , pn])	Matches packets that satisfy one of the predicates: $p1 \mid p2 \mid \ldots \mid pn$
~$p1$	Matches packets that DO NOT satisfy $p1$
Not($p1$)	Synonym for ~$p1$

For the operator versions of these, the precedence is the same as for most Boolean operators in normal programming languages: `Not`, then `And`, then `Or`. In the function versions, the precedence is determined by how you write the function compositions: e.g. `And(Or(\dots ))` applies the `Or` first, but `Or(And(\dots ))` applies the `And` first.

As a shortcut, each of the field-matching predicates `FieldEq` has an analagous `FieldNotEq` predicate which matches all values *except* the given ones.

So here are some examples in Python code:

```
import frenetic
from frenetic.syntax import *

# Note this program doesn't actually do anything

# Match packets from a particular mac address
src_match = EthSrc("10:23:10:59:12:fb:5c")

# Match packets from a particular port that are either IP or ARP packets
port_ip_arp_match = PortEq(9) & EthType(0x800, 0x806)

# Matches packets from a particular port on switches 2 or 3 only
port_switch_match = PortEq(8) & SwitchEq(2, 3)

# Matches packets from all ports except 2 or 3
non_router_match = PortNotEq(2, 3)

# Matches broadcast packets or packets from a particular port
# or packets with a particular Vlan
all_criteria = [ EthSrc("ff:ff:ff:ff:ff:ff"), PortEq(1), VlanEq(2345) ]
brd_or_port_match = Or( all_criteria )
```

Note that you can assign predicates to Python variables. They are never actually matched against packets in Python, however . . . they are always interpreted on the switch itself as part of a rule.

One predicate requires some explanation: `SwitchEq`. An OpenFlow flow table belongs to one and only one switch, but a NetKAT program belongs to every switch connected to that controller. So a predicate tagged with `SwitchEq` will limit a particular match to a particular switch. Any predicates that don't have a `SwitchEq` predicate will apply to *all* switches in the network.

Finally, there are a few special predicates:

`true`	Matches all packets
`Id()`	Matches all packets
`false`	Matches no packets
`Drop()`	Matches no packets

Why would you need these? They're useful for "catch all" rules that appear last in a list. A good example is our repeater, where we had an id rule that matched all packets and forwarded them to the controller.

2.4 Policies

NetKAT predicates are powerless by themselves. To make them work, you need to use them in NetKAT *policies*. A policy is like a command, and policies are compiled down to OpenFlow actions, and are used in table rules or the `packet_out` command. But just as NetKAT predicates are more powerful than OpenFlow matches, NetKAT policies are more powerful than OpenFlow action lists.

Filter(p)	Select packets that match NetKAT predicate p, and quietly forget the rest
`id`	Lets all packets through. Equivalent to `Filter(True))`
`drop`	Drops all packets. Equivalent to `Filter(False))`
SetPort($p_1, p_2, \ldots, p_n$)	Set the output port for the packet to the specified ports. The packet is effectively copied to each port.
SendToController(tag)	Send packet to controller with tag tag
SendToQuery(tag)	Send packet to a statistics bucket named tag
SetEthSrc(mac)	Set Ethernet Mac source address to mac
SetEthDst(mac)	Set Ethernet Mac destination address to mac.
SetVlan($vlan$)	Set packet VLAN to $vlan$. Note this is not a Vlan push - it overwrites whatever Vlan is in the packet (if there is one).
SetVlanPcp(p)	Set VLAN Priority Code Point to p.

SetEthType(t)	Set Ethernet Type to t, where t is a 32 bit integer.
SetIPProto(p)	Set IP Protocol to p.
SetIP4Src($addr$)	Set IP source address to $addr$. Note there is no mask here, as in the equivalent predicate.
SetIP4Dst($addr$)	Set IP destination address to $addr$.
SetTCPSrcPort(p)	Sets TCP source port to p.
SetTCPDstPort(p)	Sets TCP destination port to p.

Note that Set policies mirror each Eq predicate, so for example the predicate VlanEq($vlan$) has a matching SetVlan($vlan$). The exception is Switch. You can't just set the Switch to some ID – that would be analogous to teleporting a packet from one switch to another! To send a packet to a different switch, you also use Send, but you are restricted to switches that are directly connected to the current switch, and you must know out which port to send it. We'll cover strategies for dealing with this in Chapter **??**.

And just as you can combine predicates with Boolean operators, you can combine policies with NetKAT policy operators:

<table>
<tr><td>$pol1$ | $pol2$</td><td>Copy the packet and apply both $pol1$ and $pol2$ to it. This is called <i>parallel composition</i>.</td></tr>
<tr><td>Union([$pol1$, $pol2$, . . . $poln$])</td><td>Copy the packet n times and apply policy $pol[i]$ to copy i. Equivalent to $pol1$ | $pol2$ | $poln$</td></tr>
<tr><td>$pol1$ >> $pol2$</td><td>Apply the policy $pol1$ to the packet, then apply $pol2$ to it This is called <i>sequential composition</i>.</td></tr>
<tr><td>Seq([$pol1$, $pol2$, . . . $poln$])</td><td>Apply each of the policies $pol1$, $pol2$, ..., $poln$ to the packet, in order Equivalent to $pol1$ >> $pol2$ >> . . . $poln$</td></tr>
<tr><td>IfThenElse($pred$, $pol1$, $pol2$)</td><td>If packet matches predicate $pred$, then apply policy $pol1$, or else apply $pol2$. Either $pol1$ or $pol2$ is applied, but never both.</td></tr>
</table>

The >> should look familiar to C++ programmers. Like in C++, the >> operator changes a piece of data, then forwards it to the next step in the chain, one after the other. It's especially helpful in I/O, where you build a string from pieces, then send it to the output device (file or screen) as the last step.

The | symbol is somewhat like the equivalent in UNIX shell programming: the components run in parallel. However, unlike |, in NetKAT you are actually running separate copies of each policy without any connections between them. In other words, you don't send packets from the output of one into the input of another. (Note that | is also the OR symbol in NetKAT predicates, but NetKAT distinguishes between the two in its parser.)

The difference between sequential and parallel composition is subtle, and we'll talk about it more in Section 3.2

2.5 Commands and Hooks

Just like NetKAT predicates, NetKAT policies don't do anything by themselves. We need to install policies as switch rules or use them in `pkt_out` operations.

And so we come to the last level of a net app: commands and hooks. Commands are instructions from the net app to the switches (via Frenetic). OpenFlow calls these *controller-to-switch messages*. Hooks are instructions from the switches to the net app (again, via Frenetic), and OpenFlow calls these *switch-to-controller messages.*

The commands are:

`pkt_out(`*sw*`,` *payload*`,` *plist*`,` *inport*`)`	Send a packet out switch with DPID *sw*. We'll describe this in detail below.
`update(`*policy*`)`	Update all switches with the given NetKAT policy. This is the equivalent of setting the OpenFlow flow tables of all switches in one shot.
`port_stats(`*sw*`,` *port*`)`	Get statistics for a particular switch and port.
`query(`*label*`)`	Get the current statistics from a labelled bucket. We'll cover this in Chapter 9.
`current_switches()`	Gets a list of DPID's of all current, operating switches, and the operating ports for each. This is most useful in the `connected()` hook.
`config(`*compiler_options*`)`	Set Frenetic compiler options. This is an instruction to Frenetic, not the switch.

You call commands through Python method calls, e.g. `self.update(`*policy*`)`. The hooks are:

`connected()`	Called when Frenetic has finished startup and some (perhaps not all) switches have connected to it.
`packet_in(`*sw*`,` *port*`,` *payload*`)`	Called when a packet has arrived on switch *sw*, port *port* and a matching policy had a `SendToController` action This is described in detail below.
`switch_up(`*sw*`)`	Called when a switch has been initialized and is ready for commands. Some switches send this message periodically to verify the controller is operational and reachable.

switch_down(sw)	Called when a switch has been powered-down gracefully.
port_up(sw, $port$)	Called when a port has been activated.
port_down(sw, $port$)	Called when a port has been de-activated - most of the time, that means the link status is down, the network cord has been unplugged, the host connected to that port has been powered-off, or the port has been reconfigured.

Your net app may be interested in one or more of these hooks. To add code for a hook, you write a *handler* which implements the hook's signature. Following Python conventions, it must be named exactly the same as the hook. However, you don't need to provide handlers for every hook. If you don't provide one, Frenetic uses its own default handler – in the case of `connected()`, for example, it merely logs a message to the console.

We'll see most of these commands and hooks used in the next few chapters. But since `pkt_out()` and `packet_in()` are the most crucial for net apps, we'll describe them first here.

2.5.1 The packet_in Hook

`packet_in()` is used to inspect network packets. Note that *not all packets* coming in through all switches arrive here – indeed, if they did, your controller would be horribly slow (as our Repeater example is, but we'll see how to improve it in a bit.) A packet will be sent to the controller when it matches a rule with a `SendToController` policy.

Once at the controller, Frenetic will deliver the packet to `packet_in()`. The default handler will simply log a message and drop the packet, but of course that's not very interesting. The handler `packet_in` is called with three parameters:

sw	The DPID of the switch where the packet arrived.
port	The port number of the port where the packet arrived.
payload	The raw packet data.

There are two formats for the packet data: *buffered* or *unbuffered*. Most switches will only send unbuffered data, meaning the entire network packet - header and data - will be transferred to the controller. We'll talk about buffering in Section 2.5.3.

If you don't need to examine any packet data, you can simply drop $packet$, or pass it directly to `pkt_out`, as we did in our Repeater app:

```
...
class RepeaterApp(frenetic.App):

...
    def packet_in(self, dpid, port_id, payload):
        out_port = 2 if port_id = 1 else 1
        self.pkt_out(dpid, payload, SetPort(out_port_id) )

...
```

Here, we merely send the payload out unchanged. If it came in as buffered data, it will be sent out as buffered data. If it came in unbuffered, it will be sent out unbuffered. Pretty simple.

If you need to examine the packet inside the payload, you'll need some assistance. Frenetic has a simple API for that, demonstrated here:

```
from frenetic.packet import *
...
    def packet_in(self, dpid, port_id, payload):
        pkt = Packet.from_payload(dpid, port_id, payload)
        src_mac = pkt.ethSrc
        if pkt.ethType == 0x806:
          src_ip = pkt.ip4Src

...
```

`pkt = Packet.from_payload(dpid, port_id, payload)` parses the raw `payload`. The result is a Python object whose fields match the OpenFlow match fields. They are named the same as they appear in `Eq` predicates and `Set` policies except for initial lower-case letters. So the `IP4Src` field predicate is `IP4SrcEq`, the policy is `SetIP4Src`, and the object attribute is `pkt.ip4Src`

2.5.2 The pkt_out Command

`pkt_out` is used to send out packets. Most of the time, the packets you send are packets you received through `packet_in`. But there's nothing stopping from you sending arbitrarily-constructed packets from here as well.

The command takes the following parameters:

`sw`	The DPID of the switch from where the packet should be sent.
`payload`	The raw packet data.
`policy_list`	A NetKAT policy or list of policies to apply to the packet data.
`in_port`	Port ID from which to send it. This parameter is optional, and only applies to buffered packets presumably sitting at a particular port on the switch waiting to be released.

The policy_list effectively tells the switch how to act on the packet. Most NetKAT policies are usable here including SetIP4Src and SendToController. The exceptions are:

- Filter is not usable. To optionally send or not send packets, use the packet parsing library from RYU to examine the packet and make decisions.
- Only simple policies are doable, so you can't use policy operators like Union and IfThenElse. You can use a Seq but only as the outermost policy operator. You cannot use it in nested policies.

What if you want to modify a packet before sending it out? There are actually two ways to do it, each appropriate for a particular use case:

Direct Modification where you set particular data in the packet itself. With a Frenetic Packet object, you set the attributes directly, then use Packet.to_payload() to reserialize it back into a payload. Direct modification is only available for unbuffered packets.

Policy Modification is achieved through the Policy list, and is limited to modification through NetKAT policies. It can be done on buffered or unbuffered packets.

In general, Policy Modification is preferable since it works on buffered and unbuffered packets, and saves you some programming steps. For example, a common routing function is to change the destination MAC address for a next hop router. Here's an example of doing this with Policy Modification:

```
    def packet_in(self, dpid, port_id, payload):
        (next_hop_mac, next_port) = calculate_next_hop(payload)
        self.pkt_out(dpid, new_payload,
          SetEthDst(next_hop_mac) >> SetPort(next_port)
        )
```

For those times when you need Direct Modification, here's an example of how to do it:

```
from frenetic.packet import *
...

    def packet_in(self, dpid, port_id, payload):
        pkt = Packet.from_payload(dpid, port_id, payload)
        # In this example, only ARP packets come to the controller.  We
        # double-check this is the case
        assert pkt.ethType == 0x806

        # Per Openflow 1.0, the ARP opcode is in pkt.ipProto
        # Change request into reply and vice-versa
        pkt.ipProto =  2 if pkt.ipProto == 1 else 2
        # Reserialize it back and send it out port 1
        new_payload = pkt.to_payload()
        self.pkt_out(dpid, new_payload, SetPort(1) )
...
```

You can also create packets from scratch with a variation of Direct Modification.

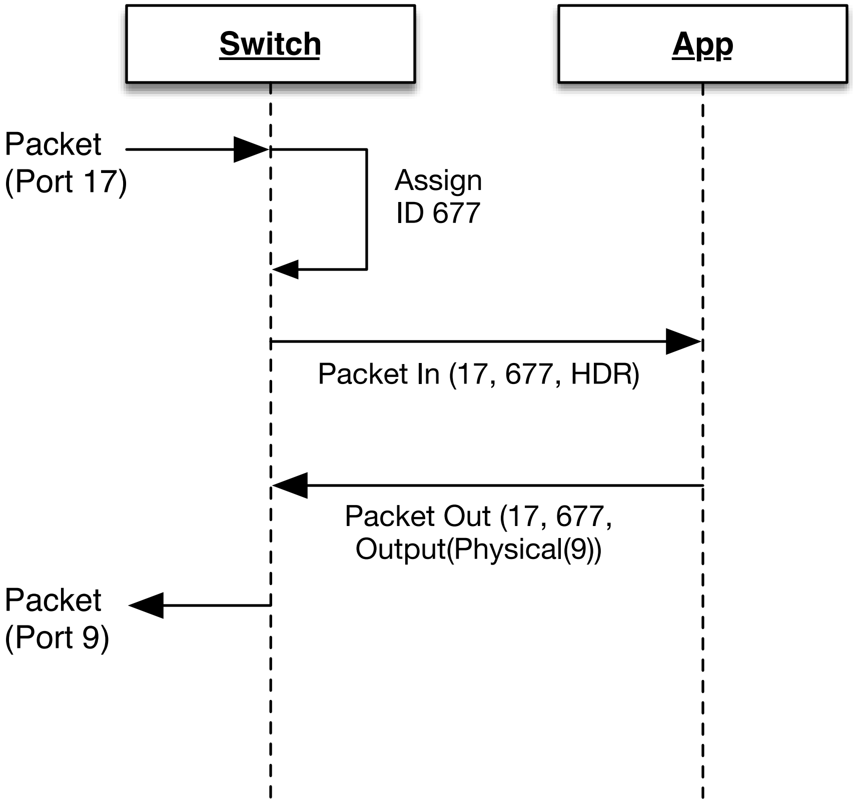
```
from frenetic.packet import *
...
    def send_arp_reply(port, src_mac, dst_mac, src_ip, dst_ip):
        pkt = Packet(
            ethSrc=src_mac, ethDst=dst_mac, ethType=0x806,
            ip4Src=src_ip, ip4Dst=dst_ip, ipProto=2
        )

        # Reserialize it back and send it out specified port
        new_payload = pkt.to_payload()
        self.pkt_out(dpid, new_payload, SetPort(port) )
...
```

2.5.3 Buffering

Packet buffering is often a settable option in a switch's OpenFlow configuration. It's especially useful when packets are large, as in Jumbo Frames. Most switching decisions made in the controller look only at the packet headers, not the data, so why should you have to send it? By buffering the entire packet, the switch only sends the headers to the controller.

A buffered Packet Out *must always* be preceded by a buffered Packet In. That's because you need to send back two things: the incoming port id, and the buffer id. The switch actually has separate buffer lists for every port on the switch, and sending it back with the proper port helps it match it to the correct buffer. This sequence diagram shows how it commonly works:



Here, the HDR is the first 128 bytes of the packet, enough to hold the Ethernet and IP header information, typically. The app doesn't send any of this header information back to the switch - just the instruction `SetPort(9)` to direct the switch's data plane.

What if we *never* send back the Packet Out? Buffer contents are not held indefinitely – they timeout after a certain period. At that point, the buffered packet will drop out. If we send a Packet Out for that buffer after the timeout period, the Packet Out will generate an error (which Frenetic ignores). A similar fate awaits Packet Outs that fabricate a random buffer id, or send it to the wrong port.

So where is the buffer id in the `PacketOut` call? It's embedded in the `payload` object. When a `PacketIn` delivers a buffered payload, you can simply it send it back out the `PacketOut` call, which sends the buffer id along with it.

2.6 OpenFlow Constructs Not Supported by NetKAT

NetKAT supports the most popular features of OpenFlow 1.0. But there a few things it doesn't do:

- It can't output to special pseudo-ports NORMAL, FLOOD, etc. The semantics of sending to these ports depends on the spanning tree, VLAN's, and other settings that NetKAT is not aware of, so it cannot reason about them.

- It can't set the size for buffered packet data in `SendToController`. All buffered packets get sent with the default 128 bytes, usually enough to encapsulate the header information.
- Frenetic must talk to switches through an unsecured channel. TLS is not supported.
- Most controller-to-switch messages like Features, Configuration or Barrier are not supported.
- Some switch-to-controller messages like Flow Removed and Vendor are ignored by Frenetic.
- Some OpenFlow actions are not supported, like Enqueue.

We haven't discussed some implemented features like Statistics yet, but those will be described in chapter 9.

Chapter 3

NetKAT Principles

3.1 Efficient SDN

In a nutshell, the `packet_in()` hook receives network packets and the `pkt_out()` command sends network packets. In theory, you could use these two to implement arbitrarily-complex network clients and servers. You could build switches and routers, but also HTTP servers, Email servers, Database servers, or any other network server.

That said, you probably wouldn't want to. OpenFlow and Frenetic are optimized for small, very selective packet inspections and creations. The more packets you inspect through `packet_in()`, the slower your controller will be, and the more likely that packets will be dropped or sent out of sequence.

Principle 1 *Keep as much traffic out of the controller as possible. Instead, program NetKAT policies to make most of the decisions inside the switch.*

So let's go back to our Repeater application:

```
import frenetic
from frenetic.syntax import *

class RepeaterApp(frenetic.App):

  client_id = "quick_start"

  def connected(self):
    self.update( id >> SendToController("repeater_app") )

  def packet_in(self, dpid, port_id, payload):
    out_port_id = 2 if port_id == 1 else 1
    self.pkt_out(dpid, payload, SetPort(out_port_id) )

app = RepeaterApp()
app.start_event_loop()
```

Here, *every single packet* goes from the switch to Frenetic to the net app and back out. That's horribly inefficient, and unnecessarily so since all the decisions can be made inside the switch. So let's write a more efficient one.

The following code is in `netkat_principles/repeater2.py`:

```python
import frenetic
from frenetic.syntax import *

class RepeaterApp2(frenetic.App):

    client_id = "repeater"

    def connected(self):
        rule_port_one = Filter(PortEq(1)) >> SetPort(2)
        rule_port_two = Filter(PortEq(2)) >> SetPort(1)
        self.update( rule_port_one | rule_port_two )

app = RepeaterApp2()
app.start_event_loop()
```

This program takes principle 1 very seriously, to the point where *no* packets arrive at the controller. All of the configuration of the switch is done up front.

The `Filter(PortEq(1)) >> SetPort(2)` policy is a pretty common pattern in NetKAT. You first whittle down the incoming flood of packets to a certain subset with `Filter`. Then you apply a policy or series of policies, `SetPort` being the most popular. We'll look at combining policies in section 3.2.

If you've worked with OpenFlow, you might wonder how the NetKAT rules get translated to OpenFlow rules. In this example, it's fairly straightforward. You get two OpenFlow rules in the rule table, which you can see in the Frenetic debug window:

```
 [INFO] POST /repeater/update_json
 [INFO] GET /repeater/event
 [INFO] New client repeater
[DEBUG] Installing policy
drop | (filter port = 1; port := 2 | filter port = 2; port := 1)
[DEBUG] Setting up flow table
+-------------------------+
| 1 | Pattern | Action    |
|-------------------------|
| InPort = 1  | Output(2) |
|-------------------------|
| InPort = 2  | Output(1) |
|-------------------------|
|             |           |
+-------------------------+
```

But this is not true in general. One NetKAT rule may expand into many, many OpenFlow rules. And it may go the opposite direction to: where different NetKAT rules are combined to create one OpenFlow rule. It's the same thing that happens with most compiled languages – the rules that govern the compiled code are non-trivial. If they were easy, you wouldn't need a compiler to do it!

There are two problems with `RepeaterApp2`:

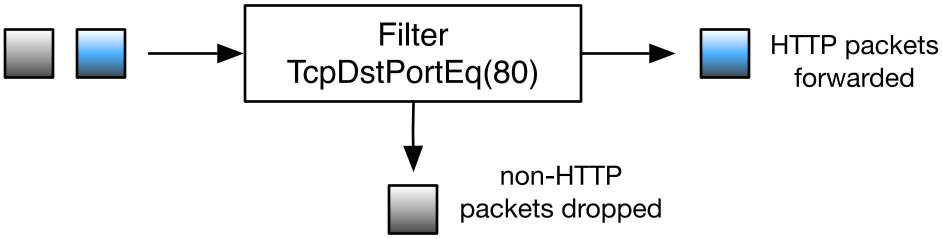
- It works on a two port switch, but not anything bigger. And the ports absolutely have to be numbered 1 and 2 . . . otherwise, the whole program doesn't work. And those ports need to be functioning.
- More subtly, this program can drop packets. There is a short lag between when the switches come up and the `self.update()` installs the policies. During this lag, packets will arrive at the controllerand get dropped by the default `packet_in` handler in Frenetic.

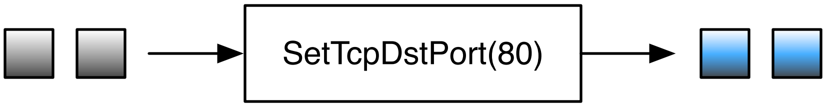
We will correct both of these problems in section 3.3

3.2 Combining NetKAT Policies

In our Repeater2 network app, the two rules have the `Seq` operator >> in them, then the two rules are joined together with `Union` or `|`. But what is the difference between the two? When do you use one or the other?

To illustrate this, let's go back to NetKAT basics. At the lowest level there are two types of building blocks: filters and modifications.



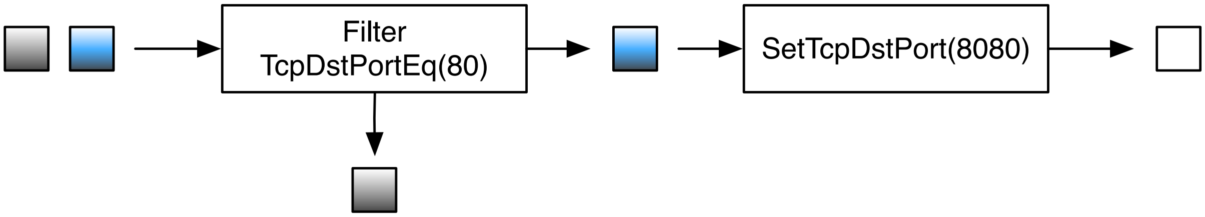


A filter, shown at the top, takes as input a stream of packets. Packets that match its criteria are sent to the next policy, packets that don't match are dropped. Two special policies, `id` and `drop`, let through all or no packets respectively.

A modification, meaning any NetKAT policy beginning with `Set`, changes packet header values. In the bottom example, the `TcpSrcPort` header gets changed to 80. This modification happens on a logical copy of the packet. That means subsequent `Filters` match against the original `TcpSrcPort` value in the header, not the one we just changed it to. But most of the time, you can ignore this subtlety because filters precede modifications.

Modifications to different headers can generally be specified in any order. If two modifications to the same header value occur one after the other, the last one wins.

You combine these building blocks with `Seq` and `Union`. Here is a `Seq` of two policies, a filter and a modification.



In sequences, policies are chained together in a straight line, and packets travel through each of one of the policies in order. Combining a filter plus modifications is very common in NetKAT programs, and we generally use `Seq` to combine them into a *rule*.

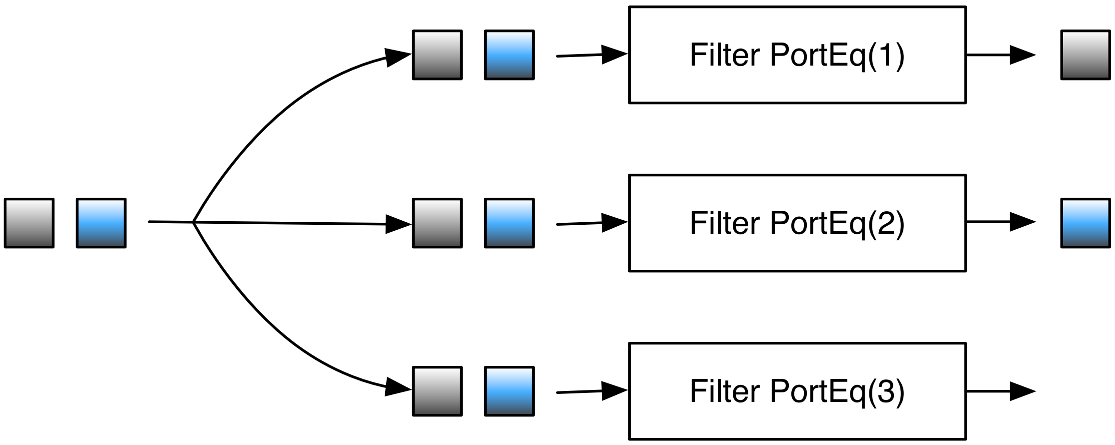
A rule is like an OpenFlow flow table entry, but with more options. If you think about it, a switch receives a huge firehose blast of packets, and you want the switch to perform a targeted action on a very small subset of those packets. In other words, you want a *filter* and some *actions*. It's like the MapReduce paradigm, but in reverse order - you first *reduce* the stream to a manageable level, then *map* the result by performing actions on it.

With `Seq`, order matters. So the rule `drop >> Filter(EthTypeEq(0x806))` drops ALL packets, no matter the type. The second filter is never reached. In general, putting all the filters before the actions in a sequence chain is the clearest way to write it.

The following principle is a good guideline:

Principle 2 *Use >> between filters and all actions.*

Here is a Union of policies.



A `Union` of policies makes logical copies of packets and sends the copy of each packet through each policy in parallel. In the above example, the two incoming packets are copied three times, one for each rule. In this case, the top filter only forwards a copy of the first packet matching `PortEq(1)`. The middle filter only forwards a copy of the second packet matching `PortEq(2)` The bottom filter doesn't forward any packets at all.

Principle 3 *Use* `|` *between rules that DO NOT overlap. Use* `IfThenElse` *to combine rules that DO overlap.*

We'll explain `IfThenElse` in a little bit.

You might think, "All that packet copying must be really tough on the switch." In fact, all the packet copying is conceptual, it doesn't necessarily happen in the switch. Frenetic turns NetKAT programs into flow tables, and these flow tables generally work through sophisticated TCAM's, Hash tables, and pipelines that eliminate actual packet copying. So don't worry about stringing 20,000 policies with a `Union`. Your switch can handle it.

`SetPort` is a little different. Other modifications like `SetTcpDstPort` act on a single packet at a time. A single `SetPort` is like this - we merely want to change the egress port of the packet itself. But if you want to send it out multiple ports at once, as in flooding, you can send a list of ports to `SetPort`.

Now let's look at `Union` and `IfThenElse`. As we have seen in Chapter 2, Union is parallel composition: we effectively make copies of each packet and apply the rules to each packet. So in our repeater application, the rules are:

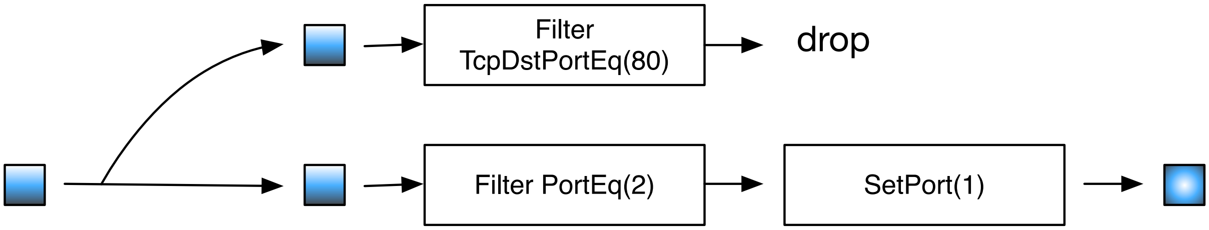
- `Filter(PortEq(1)) >> SetPort(2)` Sends traffic from port 1 out port 2

- `Filter(PortEq(2)) >> SetPort(1)` Sends traffic from port 2 out port 1

A packet cannot match both rules – in other words, there is no overlap. So a `Union` between these two rules is appropriate. You make a copy of each packet, send it through each of the rules. One or the other will match and send it out the appropriate port, the other will simply drop it.

Suppose you have two rules:

- `Filter(TcpDstPortEq(80)) >> drop` Drops non-encrypted HTTP traffic
- `Filter(PortEq(2)) >> SetPort(1)` Sends port 2 traffic out port one.

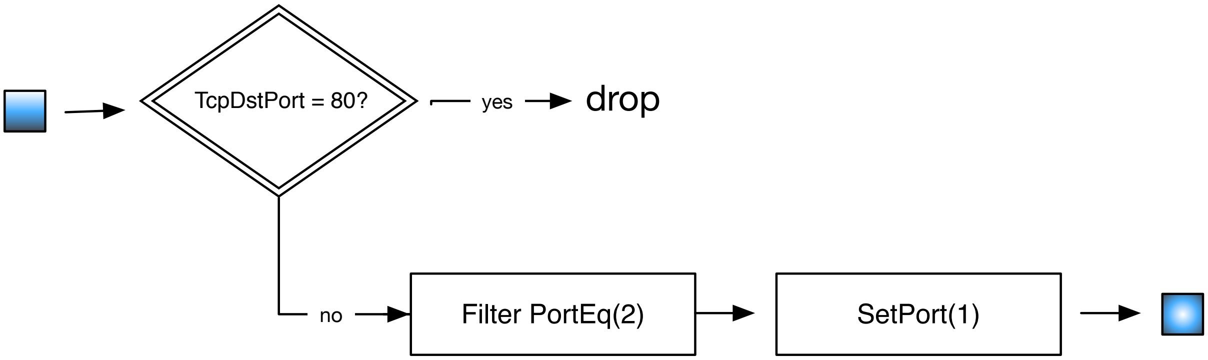


These two rules overlap. A packet can both have a TCP destination port of 80 and arrive on switch 1, port 2. What happens if we combine them with `Union`, and a packet like that arrives at the switch? The packet will be copied twice, then:

- `Filter(TcpDstPortEq(80)) >> drop` will drop the first copy of the packet
- `Filter(PortEq(2)) >> SetPort(1)` will send out the second copy to port 1

But that's probably not what you intended. You probably want the HTTP rule to take precedence over the sending rule. In such cases, `IfThenElse` makes the precedence crystal clear:

```
IfThenElse(TcpDstPortEq(80), drop, Filter(PortEq(2)) >> SetPort(1))}
```



The predicate is tested, and if it matches, the first policy is performed, otherwise the second is performed.

That's pretty powerful, and in fact we can write the repeater app to use `IfThenElse` instead of Union:

```
IfThenElse(PortEq(1), SetPort(2), Filter(PortEq(2)) >> SetPort(1)
```

So why not just use IfThenElse for combining rules all the time? In a real switch, you might have hundreds of non-overlapping rules. Writing them as:

```
IfThenElse(predicate1), action1, IfThenElse(predicate2, action2, IfThenElse #....
```

is doable but a lot more verbose. It also hides the fact that the rules are non-overlapping, and this is useful information when rummaging through an SDN app.

IfThenElse is really syntactic sugar. The statement:

```
IfThenElse(pred, true_policy, false_policy)
```

is just a shortcut for:

```
Filter(pred) >> true_policy | Filter(~ pred) >> false_policy
```

but it's a lot shorter and easier to understand, especially for programmers accustomed to seeing If-Then-Else constructs in other programming languages.

3.3 Keeping It Stateless

So let's kick it up a notch. Our repeater is fast, but pretty static – it only works with two ports, and only ports that are numbered 1 and 2. So let's extend our repeater to do two things:

1. On connection, read the list of currently available ports and build the initial NetKAT policy accordingly.
2. When ports go up or down, adjust the policy dynamically.

The first task requires us to inquire which ports a switch has. Frenetic provides a current_switches() method that does exactly that. The best time to call it is on the connected() hook since, at that point, Frenetic knows the switches and ports out there.

The following code is in netkat_principles/repeater3.py:

```
import sys, logging
import frenetic
from frenetic.syntax import *

class RepeaterApp3(frenetic.App):

  client_id = "repeater"
  def connected(self):
    def handle_current_switches(switches):
      logging.info("Connected to Frenetic - Switches: "+str(switches))
    self.current_switches(callback=handle_current_switches)

logging.basicConfig(stream = sys.stderr, \
  format='%(asctime)s [%(levelname)s] %(message)s', level=logging.INFO \
)
app = RepeaterApp3()
app.start_event_loop()
```

current_switches() is an asynchronous call, meaning you don't get the results immediately. Instead, we pass a callback procedure named handle_current_switches, which Frenetic calls when the current_switches() call is complete and has results. In our first pass, the callback is just a one line procedure that prints the results to the log (the screen by default). Although we could define handle_current_switches() outside the connected() method, placing it inside clarifies the connection between the two.

So let's start up Mininet with 4 hosts this time:

```
vagrant@frenetic:~/workspace$ sudo mn --topo=single,4 --controller=remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Start up Frenetic:

```
vagrant@frenetic:~/workspace$ ~/src/frenetic/frenetic.native http-controller
 [INFO] Calling create!
 [INFO] Current uid: 1000
 [INFO] Successfully launched OpenFlow controller with pid 2035
 [INFO] Connecting to first OpenFlow server socket
 [INFO] Failed to open socket to OpenFlow server: (Unix.Unix_error "...
 [INFO] Retrying in 1 second
```

```
[INFO] Successfully connected to first OpenFlow server socket
[INFO] Connecting to second OpenFlow server socket
[INFO] Successfully connected to second OpenFlow server socket
[INFO] switch 1 connected
```

And start up our application:

```
vagrant@frenetic:~/workspace$ python repeater3.py
No client_id specified. Using 0323e4bc31c94e81939d9f51640f1f5b
Starting the tornado event loop (does not return).
2016-04-12 09:51:14,578 [INFO] Connected to Frenetic - Switches: {1: [4, 2, 1, 3]}
```

The `handle_current_switches` callback gets called with a Python dictionary. The keys in this dictionary are the datapath ID's (dpid's) of each switch – in our case, we only have one switch with a dpid of 1. The value associated with the key is a list of ports that the switch has operational. In this case, we have four ports labelled 1 through 4 (they will be in some random order in the list, as you see above in `[4, 2, 1, 3]`).

Great! Once we have the list of ports, we can construct policies. If a packet comes in on port 1, it should be repeated to all the ports that are not 1, e.g. 2, 3 and 4, and so on. The NetKAT policy, written out manually, will look like this:

```
Filter(PortEq(1)) >> SetPort(2,3,4) |
Filter(PortEq(2)) >> SetPort(1,3,4) |
Filter(PortEq(3)) >> SetPort(1,2,4) |
Filter(PortEq(4)) >> SetPort(1,2,3)
```

Or equivalently with lists:

```
Filter(PortEq(1)) >> SetPort( [2,3,4] ) |
Filter(PortEq(2)) >> SetPort( [1,3,4] ) |
Filter(PortEq(3)) >> SetPort( [1,2,4] ) |
Filter(PortEq(4)) >> SetPort( [1,2,3] )
```

A combination of Frenetic and Python lists makes this easy to construct. Suppose you have the list named `sw` with value `[1, 2, 3, 4]` from the callback. The following Python list comprehension:

```
SetPort( [p for p in sw] )
```

returns the list:

```
SetPort( [1,2,3,4] )
```

Now suppose we're looking only at port 1. We want all of the ports here except 1. A little tweak to the list comprehension:

```
SetPort( [p for p in sw if p != in_port] )
```

Removes the input port from the list, leaving:

```
SetPort( [2,3,4] )
```

So here is our repeater that installs an initial configuration. The following code is in `netkat_principles/repeater4.py`:

```
import sys, logging
import frenetic
from frenetic.syntax import *

class RepeaterApp4(frenetic.App):

  client_id = "repeater"

  def port_policy(self, in_port, all_ports):
    return \
      Filter(PortEq(in_port)) >> \
      SetPort( [p for p in all_ports if p != in_port] )

  def all_ports_policy(self, all_ports):
    return Union( self.port_policy(p, all_ports) for p in all_ports )

  def policy(self, switches):
    # We take advantage of the fact there's only one switch
    dpid = switches.keys()[0]
    return self.all_ports_policy(switches[dpid])

  def connected(self):
    def handle_current_switches(switches):
      logging.info("Connected to Frenetic - Switches: "+str(switches))
      self.update(self.policy(switches))
    self.current_switches(callback=handle_current_switches)

logging.basicConfig(stream = sys.stderr, \
  format='%(asctime)s [%(levelname)s] %(message)s', level=logging.INFO \
)

app = RepeaterApp4()
app.start_event_loop()
```

Now it's just a hop, skip and jump to a fully dynamic repeater. First we need to capture packets from ports that we haven't seen yet. We could write an extremely long filter, filtering out every port that's not on the list, but since port numbers can be 32-bits long, that's gonna be pretty huge:

```
Filter(PortEq(5, 6, 7, ..., 4294967295) >> SendToController("repeater_app")
```

It's easier just to write an overlapping rule. Recall that `id` is a filter that matches all packets:

```
id >> SendToController("repeater_app")
```

But obviously, this overlaps with the rules for known ports in our repeater. So we use an `IfThenElse` to resolve the overlap:

```
  def all_ports_policy(self, all_ports):
    return Union( self.port_policy(p, all_ports) for p in all_ports )

  def known_ports_pred(self, all_ports):
    return PortEq(all_ports)

  def policy(self):
    return IfThenElse( \
      self.known_ports_pred(self.all_ports), \
      self.all_ports_policy(self.all_ports), \
      SendToController("repeater_app") \
    )
```

We take this opportunity to save the port list in an instance variable `self.all_ports`. This instance variable is the beginning of a Network Information Base, or NIB – it encapsulates the known state of the network at this time. We're going to see the NIB a lot in future apps. You can think of our app as a function with the NIB as input and a NetKAT program as output.

Now how we do learn about new ports? A packet arriving at `pkt_in` will signal we're seeing a new port. So suppose we're seeing a packet on port 40. If you think about, the entire NetKAT program will change from:

```
Filter(PortEq(1)) >> SetPort(2,3,4) |
Filter(PortEq(2)) >> SetPort(1,3,4) |
Filter(PortEq(3)) >> SetPort(1,2,4) |
Filter(PortEq(4)) >> SetPort(1,2,3)
```

to:

```
Filter(PortEq(1)) >> SetPort(2,3,4,40) |
Filter(PortEq(2)) >> SetPort(1,3,4,40) |
Filter(PortEq(3)) >> SetPort(1,2,4,40) |
Filter(PortEq(4)) >> SetPort(1,2,3,40) |
Filter(PortEq(40)) >> SetPort(1,2,3,4)
```

Fortunately, we have all the logic we need in `self.all_ports_policy` already. We just need to pass it an amended list of known ports. Not a problem!

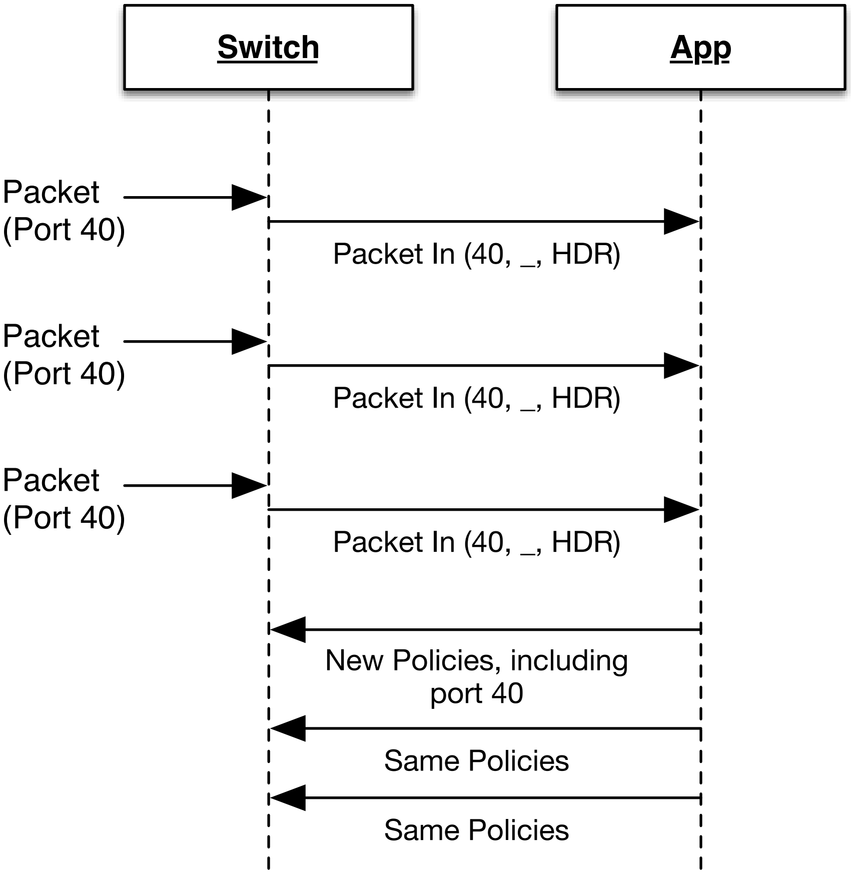
```
  def packet_in(self, dpid, port_id, payload):
    self.all_ports.add(port_id)
    self.update(self.policy())
```

What do we do with the packet we just received? We could just drop it and hope the host is smart enough to resend it. But that's not very hospitable.

What if we just do a `pkt_out` immediately afterward? After all, we just installed a rule that will deal with it appropriately, right?

```
  def packet_in(self, dpid, port_id, payload):
    self.all_ports.add(port_id)
    self.update(self.policy())
    self.pkt_out(dpid, payload, ??? )
```

There are two problems. First, it's unclear what the action should be on `pkt_out`. If we send an empty list of actions, the OpenFlow switch will interpret it as "drop the packet.". Second, there's a timing problem. Even though we just sent a self.update, the call is asynchronous with no response when it's done updating the OpenFlow flow table. It could take a microsecond, it could take an hour . . . we just don't know. This timing problem actually can cause more problems. What if, before the new rules are installed, the switch receives 100 more packets? pkt_in will be called 100 times, and each time the policy will be recalculated and sent to the switch. That could cause major problems since installing switch rules can be a CPU-intensive process on the switch.



Hence the following principle:

Principle 4 *When you install a new switch policy, do not assume it'll be installed right away.*

We can solve this with two quick fixes. One is to make a quick check that port has not actually be learned yet. The other is to add actions to pkt_out that emulate the new rule.

Here's our final repeater program in `netkat_principles/repeater5.py`:

```
import sys, logging
import frenetic
from frenetic.syntax import *

class RepeaterApp5(frenetic.App):

  client_id = "repeater"

  def port_policy(self, in_port, all_ports):
    return \
      Filter(PortEq(in_port)) >> \
      SetPort( [p for p in all_ports if p != in_port] )
  def all_ports_policy(self, all_ports):
    return Union( self.port_policy(p, all_ports) for p in all_ports )

  def known_ports_pred(self, all_ports):
    return PortEq(all_ports)

  def policy(self):
    return IfThenElse( \
      self.known_ports_pred(self.all_ports), \
      self.all_ports_policy(self.all_ports), \
      SendToController("repeater_app") \
    )

  def connected(self):
    def handle_current_switches(switches):
      logging.info("Connected to Frenetic - Switches: "+str(switches))
      # We take advantage of the fact there's only one switch
      dpid = switches.keys()[0]
      self.all_ports = switches[dpid]
      self.update(self.policy())
    self.current_switches(callback=handle_current_switches)

  def packet_in(self, dpid, port_id, payload):
    if port_id not in self.all_ports:
      self.all_ports.append(port_id)
      self.update(self.policy())
    flood_actions = SetPort( [p for p in self.all_ports if p != in_port] )
    self.pkt_out(dpid, payload, flood_actions )

logging.basicConfig(\
  stream = sys.stderr, \
  format='%(asctime)s [%(levelname)s] %(message)s', level=logging.INFO \
)
app = RepeaterApp5()
app.start_event_loop()
```

In mininet with a simple 4-host topology, we can test our repeater by adding a 5th host and port:

```
mininet> py net.addHost('h5')
<Host h5:  pid=3187>
mininet> py net.addLink(s1, net.get('h5'))
<mininet.link.Link object at 0x7fd432ace810>
mininet> py s1.attach('s1-eth5')
mininet> py net.get('h5').cmd('ifconfig h5-eth0 10.5')
```

The new port causes the switch to send an OpenFlow portUp message to our app. We don't have a handler for the `port_up` hook, so the default message appears:

```
port_up(switch_id=1, port_id=5)
```

Back in Mininet, let's try to ping from our new host:

```
mininet> h5 ping -c 1 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=996 ms

--- 10.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 996.738/996.738/996.738/0.000 ms
```

Looks good! A combination of the new packet rules and the `pkt_out` is sending the packets to the correct place.

3.4 Summary

In building a dynamic repeater, we have learned the following NetKAT principles:

Principle 1 *Keep as much traffic out of the controller as possible. Instead, program NetKAT policies to make most of the decisions inside the switch.*

Principle 2 *Use >> between filters and all actions.*

Principle 3 *Use* `|` *between rules that DO NOT overlap. Use* `IfThenElse` *to combine rules that DO overlap.*

Principle 4 *When you install a new switch policy, do not assume it'll be installed right away.*

And we'll learn a fifth in the next chapter:

Principle 5 *Do not rely on network state. Always assume the current packet is the first one you're seeing.*

These principles are meant as guidelines, not straitjackets. As we build network applications over the next few chapters, we may find good reasons to violate them to achieve shorter code or better modularity. That's OK.

Now we have a robust repeater that acts mostly correctly. There is a still a small timing problem. If a packet arrives from port 1 bound for port 40 in that short interval before the rules for port 40 are installed, it will be copied to ports 2, 3, and 4 but not 40. There's little we can do in the context of our simple repeater.

But of course, modern network equipment doesn't act like a repeater. That's 1980's technology! Repeaters do a lot of unnecessary packet copying, flooding hosts with packets that are clearly destined for them. So our next project will be building an L2 learning switch. In that context, we'll correct some of the remaining timing flaws. And the result will be much more efficient.

Chapter 4

Learning Switch

4.1 Design

Layer 2, or L2, switching revolutionized networking in the 1990's. As LAN traffic grew, hub performance rapidly degraded as collisions became more and more frequent. L2 switches effectively cut the LAN into logical segments, performing the forwarding between them. This dramatically reduced the number of collisions, and also cut down on the traffic that individual NIC's had to filter out. Just as the Plain Old Telephone System evolved from party lines to direct lines, LAN hardware evolved from Hubs to L2 switches, improving security, speed and signal quality.

Of course, L2 switches were more technically sophisticated than hubs. They required a processor, memory, and some form of software. In order to know which segments to forward traffic, they needed to watch the Ethernet MAC addresses of traffic and theremember their respective ports. In other words, the switch *learns* the MAC-to-port mappings, and thus L2 switches are called learning switches.

We can simulate the L2 switch with Frenetic. By doing so, as well see in Section 4.4, we can add features to the switch with just a little extra programming effort. At a high-level, you can think of a Frenetic network application as:

$$netkat = f(nib, env)$$

Where f is your application, nib is the Network Information Base – the information you have dynamically determined in your network through packets received by `pkt_in` – and env is other information (fixed configuration files, out-of-band network measurements, or whatever you want). The output, $netkat$ is the NetKAT program.

Naturally, nib is critical to a good design. You don't need to capture all aspects of the network, only those needed to properly form switch rules. In an L2 switch, we are really only interested in three pieces of data in a packet:

- The source MAC address

- The destination MAC address

- The switch port connected to the host with a particular MAC address

So that's all the data we want to capture for the NIB. Here's a rough design for how we want the switch to behave:

```
if port_for_mac(EthSrc) == None:
  learn(EthSrc, Port)
if port_for_mac(EthDst) != None:
  pkt_out(payload, port)
else
  pkt_out(payload, all_ports_except(port))
```

Admittedly this is pretty sketchy, but it covers the interesting cases. In particular, it covers Ethernet broadcasts to MAC address ff:ff:ff:ff:ff:ff just by the fact that a source MAC will never equal ff:ff:ff:ff:ff:ff. And flooding is exactly what you want to do in that case.

So our NIB must maintain at least a list of MAC-to-port mappings. In our Repeater app, our NIB was a single instance variable in the application itself: `self.ports`, which held a list of connected ports on the switch. Now we'll evolve a little. In what will become a standard part of our network apps, we'll model the NIB as a separate object class in Python. The following code is in `l2_learning_switch/network_information_base.py`:

```
class NetworkInformationBase(object):

  # hosts is a dictionary of MAC addresses to ports
  #  { "11:11:11:11:11": 2, ...}
  hosts = {}

  # ports on switch
  ports = []

  def __init__(self, logger):
    self.logger = logger

  def learn(self, mac, port_id):
    # Do not learn a mac twice
    if mac in self.hosts:
      return

    self.hosts[mac] = port_id
    self.logger.info(
      "Learning: "+mac+" attached to ( "+str(port_id)+" )"
    )

  def port_for_mac(self, mac):
    if mac in self.hosts:
      return self.hosts[mac]
    else:
      return None

  def mac_for_port(self, port_id):
    for mac in self.hosts:
      if self.hosts[mac] == port_id:
        return mac
    return None

  def unlearn(self, mac):
    if mac in self.hosts:
      del self.hosts[mac]

  def all_mac_port_pairs(self):
    return [ (mac, self.hosts[mac]) for mac in self.hosts.keys() ]

  def all_learned_macs(self):
    return self.hosts.keys()

  def set_ports(self, list_p):
    self.ports = list_p

  def add_port(self, port_id):
    if port_id not in self.ports:
      self.ports.append(port_id)

  def delete_port(self, port_id):
    if port_id in self.ports:
      self.ports.remove(port_id)

  def all_ports_except(self, in_port_id):
    return [p for p in self.ports if p != in_port_id]

  def switch_not_yet_connected(self):
    return self.ports == []
```

That encapsulates the state in one place, making it easy to change underlying data structures later. It also separates the NIB details from the NetKAT details, making it easier to reuse the NIB in other applications.

4.2 A First Pass

One of the problems with our switch pseudocode design is it doesn't fit our notions of NetKAT very well. NetKAT programs do not have variables, so they can't remember MAC-to-port mappings on their own. So it appears that every packet must pass through the controller so we can make decisions. Processing every single packet through the controller clearly violates the First NetKAT principle, but we can leave that aside for now. It'll be instructive to build an easy but inefficient L2 switch first.

The following code is in `l2_learning_switch/learning1.py`:

```python
import sys, logging
import frenetic
from frenetic.syntax import *
from frenetic.packet import *
from network_information_base import *

class LearningApp1(frenetic.App):

  client_id = "l2_learning"

  def __init__(self):
    frenetic.App.__init__(self)
    self.nib = NetworkInformationBase(logging)

  def connected(self):
    def handle_current_switches(switches):
      logging.info("Connected to Frenetic - Switches: "+str(switches))
      dpid = switches.keys()[0]
      self.nib.set_ports( switches[dpid] )
      self.update( id >> SendToController("learning_app") )
    self.current_switches(callback=handle_current_switches)

  def packet_in(self, dpid, port_id, payload):
    nib = self.nib

    pkt = Packet.from_payload(dpid, port_id, payload)
    src_mac = pkt.ethSrc
    dst_mac = pkt.ethDst

    # If we haven't learned the source mac, do so
    if nib.port_for_mac( src_mac ) == None:
      nib.learn( src_mac, port_id)

    # Look up the destination mac and output it through the
    # learned port, or flood if we haven't seen it yet.
    dst_port = nib.port_for_mac( dst_mac )
    if  dst_port != None:
      actions = SetPort(dst_port)
    else:
      actions = SetPort( nib.all_ports_except(port_id) )
    self.pkt_out(dpid, payload, actions )

if __name__ == '__main__':
  logging.basicConfig(\
    stream = sys.stderr, \
    format='%(asctime)s [%(levelname)s] %(message)s', level=logging.INFO \
  )
  app = LearningApp1()
  app.start_event_loop()
```

There are a couple of new details to note:

- The `__init__` constructor must call the superclass constructor to properly initialize.
- Because we are writing in classes, we now distinguish the main loop of this application with a check on `__main__`.
- We are using the Frenetic `Packet` object discussed in Section 2.5.1
- The `packet_in` looks almost exactly like our pseudocode design

Starting up Mininet, Frenetic and our application respectively, we try a `pingall` in Mininet and see the following on the console:

```
vagrant@frenetic:~/manual/programmers_guide/code/l2_learning_switch$ python learning1.py
Starting the tornado event loop (does not return).
2016-04-14 12:49:17,228 [INFO] Connected to Frenetic - Switches: {1: [4, 2, 1, 3]}
2016-04-14 12:49:17,229 [INFO] Learning: 9a:0f:ec:39:54:f5 attached to ( 1 )
2016-04-14 12:49:17,258 [INFO] Learning: be:3f:5a:90:8a:ac attached to ( 2 )
2016-04-14 12:49:17,303 [INFO] Learning: 3a:a4:6b:e6:24:25 attached to ( 3 )
2016-04-14 12:49:17,343 [INFO] Learning: f2:a7:c0:cb:90:23 attached to ( 4 )
```

The switch works perfectly! But it's a huge violation of Principle 1: all the traffic goes through the controller.

4.3 A More Efficient Switch

Once we've learned a MAC-to-port mapping, we shouldn't have to go to the controller for packets destined for that MAC. The switch should handle it by itself.

This is actually pretty straightforward. If we know that MAC 11:11:11:11:11:11 is on port 2, we can handle it with the following NetKAT program:

```
Filter(EthDstEq("11:11:11:11:11:11")) >> SetPort(2)
```

And we just need one of these rules for each MAC we've learned. But all of these rules are non-overlapping because they involve different values for `EthDst`. So we just Union them all together and that's our entire NetKAT program.

So let's write some methods for calculating the policies We'll add this code to learning1 (listed in `l2_learning_switch/learning2.py`):

```
  def policy_for_dest(self, mac_port):
    (mac, port) = mac_port
    return Filter(EthDstEq(mac)) >> SetPort(port)

  def policies_for_dest(self, all_mac_ports):
    return [ self.policy_for_dest(mp) for mp in all_mac_ports ]

  def policy(self):
    return Union( self.policies_for_dest(self.nib.all_mac_port_pairs()) )
```

Note here that `(mac, port) = mac_port` unpacks the tuple `mac_port` into two variables `mac` and `port`.

When shall we install these rules? We could install them on every incoming packet, but that's a little overkill. We really only need to recalculate them when we see a newly learned MAC and port. So we add them to that conditional:

```
    if nib.port_for_mac( src_mac ) == None:
      nib.learn( src_mac, port_id)
      self.update(self.policy())
```

Now run it and try a pingall from Mininet:

```
vagrant@frenetic:~/manual/programmers_guide/code/l2_learning_switch$ python learning1.py
Starting the tornado event loop (does not return).
2016-04-14 13:33:22,965 [INFO] Connected to Frenetic - Switches: {1: [2, 4, 1, 3]}
2016-04-14 13:33:26,447 [INFO] Learning: 86:d8:df:f0:95:75 attached to ( 1 )
2016-04-14 13:33:26,453 [INFO] Learning: 4a:1c:9e:9b:50:7c attached to ( 2 )
... STOP
```

Uh oh. Why did we only learn the first two ports? Let's look at the Frenetic console for a clue:

```
[DEBUG] Installing policy
drop |
(filter ethDst = 4a:1c:9e:9b:50:7c; port := 2 |
 filter ethDst = 86:d8:df:f0:95:75; port := 1)
[DEBUG] Setting up flow table
+----------------------------------------+
| 1 | Pattern                | Action    |
|----------------------------------------|
| EthDst = 4a:1c:9e:9b:50:7c | Output(2) |
|----------------------------------------|
| EthDst = 86:d8:df:f0:95:75 | Output(1) |
|----------------------------------------|
|                            |           |
+----------------------------------------+
```

Can you see the problem? There's no longer a rule to send packets to the controller. If packets are destined for the first two MAC addresses, that's not a problem. But if they're destined for other MAC addresses, it *is* a problem.

If that's true, why didn't it stop after the first packet? Remember NetKAT Principle 4, which states there is a lag time between rule sending and rule installation. In this case:

1. The first packet came to the controller, causing a rule regeneration. These rules are sent to the switch, but are not installed yet.

2. The second packet came to the controller, causing a second rule regeneration.

3. The rules from the first packet are installed on the switch, effectively shutting off any more packets from going to the controller.

4. The rules from the second packet are installed.

One thing that definitely *won't* work is to add the following rule with a Union:

```
id >> SendToController("learning_app")
```

The `id` filter matches all packets, and therefore overlaps every other rule. Even if we place this rule as the last rule in a set of Unions, *that does not guarantee it'll be fired last.* Frenetic does not guarantee the OpenFlow rules will follow the order of the NetKAT rules.

There are a few ways to solve this problem, but we'll try an easy one first. In Chapter 2, we mentioned briefly that for every `FieldEq` NetKAT predicate, there is a corresponding `FieldNotEq` predicate. We can use that in our policy, as we see in `learning3.py`:

```
  def policy(self):
    return \
      IfThenElse(
        EthSrcNotEq( self.nib.all_learned_macs() ) |
          EthDstNotEq( self.nib.all_learned_macs() ),
        SendToController("learning_app"),
        Union( self.policies_for_dest(self.nib.all_mac_port_pairs()) )
      )
```

Basically, we want to see all packets with an unfamiliar Ethernet source MAC (because we want to learn the port) or destination MAC (because we need to flood it out all ports). Although we could set up NetKAT policies to handle the latter case, it tends to be overkill since an unlearned destination MAC usually replies soon afterwards, the MAC is learned, and everything is good.

Notice how the this fairly simple NetKAT policy becomes a large OpenFlow table:

```
[DEBUG] Setting up flow table
+------------------------------------------------------+
| 1 | Pattern                | Action                  |
|------------------------------------------------------|
| EthDst = 00:00:00:00:00:01 | Output(1)               |
| EthSrc = 00:00:00:00:00:01 |                         |
|------------------------------------------------------|
| EthDst = 00:00:00:00:00:01 | Output(1)               |
| EthSrc = 00:00:00:00:00:02 |                         |
|------------------------------------------------------|
| EthDst = 00:00:00:00:00:01 | Output(1)               |
| EthSrc = 00:00:00:00:00:03 |                         |
|------------------------------------------------------|
| EthDst = 00:00:00:00:00:01 | Output(1)               |
| EthSrc = 00:00:00:00:00:04 |                         |
|------------------------------------------------------|
| EthDst = 00:00:00:00:00:01 | Output(Controller(128)) |
|------------------------------------------------------|
...
```

In fact, the table will have n^2 entries where n is the number of learned MACs. It's a lot easier to write the NetKAT rules than all these OpenFlow rules.

4.4 Timeouts and Port Moves

Our learning switch works fine if MAC-to-port assignments never change. But a network is usually more fluid than that:

- Users unplug a host from one port and plug it into another. In our application, packets will continue to go to the old port.
- Users replace one host (and associated MAC) with another host in the same port. In our application, the old MAC will continue to take up rule space on the switch, making it more confusing to debug.

Fortunately, plugging and unplugging hosts sends OpenFlow events `port_up` and `port_down`, respectively. We can write hooks that control MAC learning and unlearning. The following code is in `learning4.py`:

```
  def port_up(self, dpid, port_id):
    # Just to be safe, in case we have old MACs mapped to this port
    self.nib.unlearn( self.nib.mac_for_port(port_id) )
    self.nib.add_port(port_id)
    self.update(self.policy())

if __name__ == '__main__':
```

```
logging.basicConfig(\
  stream = sys.stderr, \
  format='%(asctime)s [%(levelname)s] %(message)s', level=logging.INFO \
```

When we make a port change, we call `update()` to recalculate and send the NetKAT rules down to the switch. This keeps the forwarding tables in sync with the NIB.

If we can rely on `port_up` and `port_down` events, this approach would work fine. However, in the real world, the following things can happen:

- The `port_up` or `port_down` message might not fire. Since they rely on power being present or absent, they are not always reliable.

- The messages might arrive in the wrong order, as in the port "flip-flopping" between active and inactive status.

Modern switches solve these issues by holding MAC-to-port mappings for a certain amount of time, then timing them out and (if they're still connected) relearning them. Pure OpenFlow flow rules emulate this by assigning a timeout value to each flow rule. But NetKAT doesn't have timeouts, and indeed since NetKAT policies don't map one-to-one to flow table rules, it'd be difficult to pass this information on.

But our application obeys the following principle:

Principle 5 *Do not rely on network state. Always assume the current packet is the first one you're seeing.*

That means we can restart the application at any time. This clears out the NIB and sets the flow table back to its initial "send all packets to controller" state. There will be a slight performance penalty as MAC-to-port mappings are relearned, but eventually the NIB will be filled wth all current MAC addresses . . . and no outdated ones.

Following this principle yields another benefit: fault tolerance. If the switch loses connectivity with our application, the switch will continue to function. No new MACs will be learned, but otherwise the network will continue to run. When the application returns, it will start with a clear NIB and relearn MACs.

In other words, a fully populated NIB is not critical to switch operation. It makes things much faster by providing the basis for the NetKAT rules. But it's not so important that it needs to be persisted or shared, making the overall design much simpler.

4.5 Summary

Our learning switch application is a mere 130 lines of code, but it does a lot:

- Mac addresses are learned in the NIB as packets arrive at the switch

- The flow table is updated to match the NIB

- Broadcast packets are automatically forwarded to all ports
- It handles hosts moved to other ports or replaced with other devices
- It is fault tolerant, and can easily tolerate restarts

The latest traditional switches can do even more. For example, you can plugin a Wireless Access Port or WAP to a switch port. Practically, the WAP acts like a multiplexer allowing multiple MACs to attach to a single port. It turns out our application handle multiple MACs mapped to a port, although it's difficult to model this in Mininet. We can enforce certain security constraints, like the number of MACs on a particular WAP, simply by changing the NIB class.

It's this kind of flexibility that makes SDN an important evolutionary move. In the next chapter, we'll inject some more intelligence into the network to handle Virtual LANs, or VLANs.

Chapter 5

Handling VLANs

5.1 VLAN Uses

Back in the late 1990's when LAN switching became more prevalent, network architects ran into balancing problems. Enterprise divisions and departments wanted to keep their LANs separate for security and speed reasons. For example, departments that generated heavy traffic during certain times of the month (like the Accounting department) needed to be segregated from others to prevent intolerable slowdowns in other departments.

The problem was network boundaries had to match physical switch boundaries. If you bought switches with 48 ports, and a department had 12 hosts, then 36 would remain unused. Or, worse, if a department grew from 48 to to 49 ports, you had to make an emergency network buy.

Networks often solved these problems by adding routers, splitting department subnets and letting the routers handle traffic between them. But routing can be expensive and slow, especially for large bursts of traffic. Routing makes sense between networks you want to keep segregated, but it's overkill between two hosts that you know you want to keep together.

Virtual LAN's, called VLAN's, evolved to solve these problems. You can think of a VLAN as a logical network, mappable onto physical switches. A VLAN is like an elastic switch – you assign real ports to a particular VLAN, then can add or remove them at will.

VLANs can span multiple switches. So if you needed that 49th port for a department, you could assign an unused port on a different switch to that VLAN, and the host on it would act as if it were connected to the same switch. Of course, there's technically more to it than that, especially since you need to get the packet from one switch to the other. But as a network operator, VLANs freed you from worrying about those details.

OpenFlow-based switches and Frenetic can integrate with standard VLAN technology easily. In this chapter, we'll first simulate a simple VLAN in software. Then we'll use standard 802.1Q VLANs to make a more flexible setup. Then we'll use VLAN technology between switches.

5.2 A Simple, Fixed VLAN

You've probably noticed that, as we're designing networking applications we start with something static and hard-coded, then gradually move to something configurable and dynamic. This kind of organic growth makes network application design more agile. Trying to design and implement everything at once tends to make debugging difficult.

So with VLANs, we'll start with a simple rule:

- Odd-numbered ports (1, 3, 5, . . .) will get mapped to VLAN 1001
- Even-numbered ports (2, 4, 6, . . .) will get mapped to VLAN 1002

The VLAN ids 1001 and 1002 will only be used internally for this first application. Later, we'll use them as actual 802.1Q VLAN ids, since they fit into the range 1..4096.

Hosts in each VLAN will talk only to hosts in their own VLAN, never to hosts in the other VLAN. We'll talk about how to connect the two VLANs together a little later.

So in NetKAT, the L2 switch basic rule looks like this:

```
Filter(EthDstEq("11:11:11:11:11:11")) >> SetPort(2)
```

Note that the destination, port 2, is on VLAN 1002, to which all even-numbered ports belong. To segregate the networks, we'll simply extend the rule to look like this:

```
Filter(PortEq(2,4,6,8,10)) >> Filter(EthDstEq("11:11:11:11:11:11")) >> SetPort(2)
```

Where we cobble together $2, 4, 6 \ldots$ from the even-numbered ports we know about on the switch.

If the destination is unknown, instead of flooding packets out *all* ports of the switch, we'll flood over ports of the switch *in the same VLAN*. In other words, the Packet Out action will be:

```
SetPort(2,4,...)
```

We'll add some methods onto the `NetworkInformationBase` object to do some VLAN mapping. The following code is in `code/handling_vlans/network_information_base_static.py`:

```
  def vlan_of_port(self, port_id):
    return 1001 if port_id % 2 == 1 else 1002

  def ports_in_vlan(self, vlan):
    ports_have_remainder = 1 if vlan == 1001 else 0
    return [ p for p in self.ports if p % 2 == ports_have_remainder ]

  def all_vlan_ports_except(self, vlan, in_port_id):
    return [ p for p in self.ports_in_vlan(vlan) if p != in_port_id ]
```

The `policy_for_dest` method gets extended with the new filtering. The following code is in `code/handling_vlans/vlan1.py`:

```
def policy_for_dest(self, mac_port):
  (mac, port) = mac_port
  mac_vlan = self.nib.vlan_of_port(port)
  ports_in_vlan = self.nib.ports_in_vlan(mac_vlan)
  return \
    Filter(PortEq(ports_in_vlan)) >> \
    Filter(EthDstEq(mac)) >> \
    SetPort(port)
```

And the `packet_in` handler does some extra VLAN handling. Note that if the destination port is already learned, we verify it's connected to the same VLAN as the source. That way, hosts cannot just forge destination MAC addresses in the other VLAN and subvert security.

```
def packet_in(self, dpid, port_id, payload):
  nib = self.nib

  # If we haven't learned the ports yet, just exit prematurely
  if nib.switch_not_yet_connected():
    return

  pkt = Packet.from_payload(dpid, port_id, payload)
  src_mac = pkt.ethSrc
  dst_mac = pkt.ethDst
  src_vlan = self.nib.vlan_of_port(port_id)

  # If we haven't learned the source mac, do so
  if nib.port_for_mac( src_mac ) == None:
    nib.learn( src_mac, port_id)
    self.update(self.policy())

  # Look up the destination mac and output it through the
  # learned port, or flood if we haven't seen it yet.
  dst_port = nib.port_for_mac( dst_mac )
  if  dst_port != None:
    # Don't output it if it's on a different VLAN
    dst_vlan = nib.vlan_of_port(dst_port)
    if src_vlan == dst_vlan:
      actions = SetPort(dst_port)
    else:
      actions = [ ]
  else:
    actions = SetPort( nib.all_vlan_ports_except(src_vlan, port_id) )
  self.pkt_out(dpid, payload, actions )
```

Otherwise, learning switch internals stay the same. Using a `single,4` topology in Mininet, a pingall:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X h3 X
h2 -> X X h4
h3 -> h1 X X
h4 -> X h2 X
*** Results: 66\% dropped (4/12 received)
mininet>
```

reveals that only odd-numbered ports talk to odd-numbered ports, and only even-numbered ports talk to even-numbered ports. We now have two segregated VLANs.

5.3 Standard, Dynamic VLANs

So now we want to make our setup a little more flexible. Our program effectively codes the port-to-VLAN mappings. We *could* also store this mapping in a configuration file or a database. But even that's too static. We want network operators to be able to change VLAN-to-port mappings on the fly, simply by changing the interface configuration on the switch.

So how are VLANs assigned to ports? The procedure varies from switch to switch, but in most cases you create an *untagged* interface. On an HP switch, for example, you configure the interface information:

```
coscintest-sw-ithaca# config
coscintest-sw-ithaca(vlan-1)# vlan 1001
coscintest-sw-ithaca(vlan-1001)# untagged A1
```

This assigns the port A1 to VLAN 1001. On OpenVSwitch, which is the OpenFlow switch we're using under Mininet, the configuration is similar:

> *NOTE: The following tag setting commands don't seem to work correctly on the OpenVSwitch version found on Frenetic VM. Packet In messages have no VLAN tag. One nasty workaround is to use a custom Mininet topology, where a separate driver attached to an IP address on each Mininet host sends VLAN-tagged packets. This is not only a pain to model, but it's not how real OpenFlow switches (like the Dell and HP) handle VLANs.*

```
mininet> sh ovs-vsctl set port s1-eth1 tag=1001
mininet> sh ovs-vsctl set port s1-eth2 tag=1002
mininet> sh ovs-vsctl set port s1-eth3 tag=1001
mininet> sh ovs-vsctl set port s1-eth4 tag=1002
mininet> sh ovs-vsctl show
ae82ea91-2820-4b9f-9b01-fd8e897675b9
    Bridge ”s1”
        Controller ”tcp:127.0.0.1:6633”
        Controller ”ptcp:6634”
        fail_mode: secure
        Port ”s1-eth4”
            tag: 1002
            Interface ”s1-eth4”
        Port ”s1”
            Interface ”s1”
                type: internal
        Port ”s1-eth2”
            tag: 1002
            Interface ”s1-eth2”
        Port ”s1-eth1”
            tag: 1001
            Interface ”s1-eth1”
        Port ”s1-eth3”
            tag: 1001
            Interface ”s1-eth3”
    ovs_version: ”2.1.3”
```

That means any packets going into this port without any VLAN information (i.e. untagged) are implicitly tagged with a fixed VLAN id. This VLAN id is visible in NetKAT predicates, and in the `packet_in` hook, even though they technically don't belong to the packet that entered the switch. (This is a rare instance where packet modifications are applied before Frenetic gets ahold of them.)

Outgoing packets on these ports have VLAN information stripped from them. That's because we generally want hosts to be encumbered by VLAN information. This allows us to move the host from one VLAN to another without reconfiguring it.

Learning VLAN-to-port mappings is very similar to learning MAC-to-port mappings. In fact, we can do it all at the same time. We assume that an incoming packet for port p with a VLAN tag of v can be learned as a VLAN-to-port mapping. If a port is ever reassigned to another VLAN, the same `port_down` and `port_up` hooks will be fired as for MAC moves, so we can unlearn the VLAN mappings at the same time.

We'll store the VLAN-to-port mappings in a Python dictionary, similar to our MAC-to-port mappings. One difference is the values of this dictionary will be lists, since a VLAN (the key of our dictionary) will be assigned to many ports. This code will will replace our static VLAN view in `code/handling_vlans/network_information_base_static.py`:

```
# VLAN Handling, dynamic version
  # vlans is a dictionary of VLANs to lists of ports
  #  { ”1001”: [1,3], ”1002”: [2,4] ...}

  vlans = {}
  def vlan_of_port(self, port_id):
    for vl in self.vlans:
      if port_id in self.vlans[vl]:
        return vl
    return None

  def ports_in_vlan(self, vlan):
    return self.vlans[vlan] if vlan in self.vlans else None

  def all_vlan_ports_except(self, vlan, in_port_id):
    return [ p for p in self.ports_in_vlan(vlan) if p != in_port_id ]
```

Then we tweak the `learn` method to learn both MACs and VLANs:

```
  def learn(self, mac, port_id, vlan):
    # Do not learn a mac twice
    if mac in self.hosts:
      return

    self.hosts[mac] = port_id
    if vlan not in self.vlans:
      self.vlans[vlan] = []
    self.vlans[vlan].append(port_id)
    self.logger.info(
      "Learning: "+mac+" attached to ( "+str(port_id)+" ), VLAN "+str(vlan)
    )
```

And in the `delete_port` method, we clean up any lingering VLAN-to-port mappings for that port:

```
  def delete_port(self, port_id):
    if port_id in ports:
      self.ports.remove(port_id)
      for vl in self.vlans:
        if port_id in self.vlans[vl]:
          self.vlans[vl].remove(port_id)
```

Since we now have packets tagged with VLANs, the NetKAT policies no longer need to reference list of ports. They will change to look like:

```
Filter(VlanEq(1002)) >> Filter(EthDstEq("11:11:11:11:11:11")) >> SetPort(2)
```

Which we enshrine in the `policy_for_dest` method, listed in `code/handling_vlans/vlan2.py`:

```
  def policy_for_dest(self, mac_port):
    (mac, port) = mac_port
    mac_vlan = self.nib.vlan_of_port(port)
    return \
      Filter(VlanEq(mac_vlan)) >> \
      Filter(EthDstEq(mac)) >> \
      SetPort(port)
```

In the `packet_in` handler, we read the VLAN from the packet instead of precomputing it:

```
    pkt = Packet.from_payload(dpid, port_id, payload)
    src_mac = pkt.ethSrc
    dst_mac = pkt.ethDst
    src_vlan = self.nib.vlan_of_port(port_id)

    # If we haven't learned the source mac, do so
```

And now our switch handles VLANs dynamically. You can create new VLANs, assign ports to them, rearrange or dismantle them altogether, all without restarting the network application.

5.4 Summary

Virtual LANs, or VLANs, introduce a level of indirection in creating a network. Before them, physical switches forced physical boundaries on the design. Overprovisioning meant many ports stood unused while other switches were filled to the brim.

In this chapter we created:

- An application that simulates VLANs algorithmically
- And an extnded, dynamic application that reads VLAN information from packets and adjust accordingly, similar to our learning switch.

Note that you can combine these two approaches. You can enforce certain rules, such as ”Ports 1-35 will always be access ports assigned to VLANs 1000-1999, ports 35 and up will be on the management VLAN 1.” This is an excellent use case for SDN – your rules will never be hampered by an inflexible switch.

Up until this point, we have been working with one switch, which makes things easy to design. But multiple switches will extend the range of our network greatly. They come with their own sets of challenges though, which we’ll work through in our next chapter.

Chapter 6

Multi-Switch Topologies

Up until now, we've been working with a one-switch network. Indeed, we can divide any SDN network into one-switch subnets with a separate controller and network application for each. We can even share network information between these applications through a database or special network protocols.

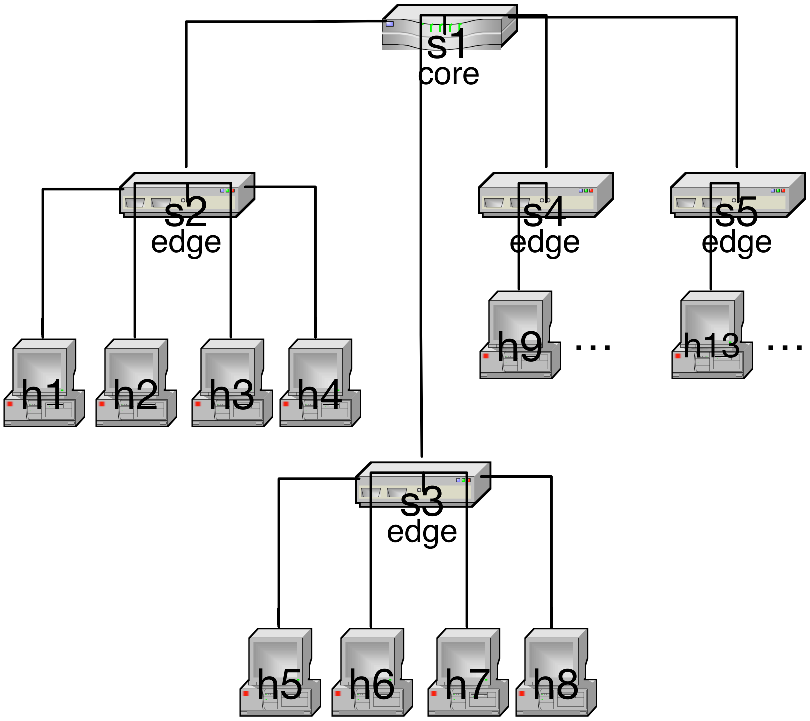
But this makes the SDN unncessarily complex and expensive. By using a single controller and application connected to a collection of switches, we gain the following:

- A simpler network application architecture. Similar switches can share policy templates.
- A cheaper infrastsucture by using one controller server (we can add redundant controllers if fault tolerance is required).
- A larger overall network view. This eliminates the need for complex protocols like Spanning Tree Protocol (STP).

The main thing in handling multi-switch networks is to avoid loops and broadcast storms. These two problems can slow a network to a crawl.

6.1 A Simple Core/Edge Network

Let's start with a fairly standard network topology.



In this network, each of the four switches has four hosts of its own – commonly one or more switches serve the hosts on a particular building floor. These are called *edge switches* because they live on the edge of the network. Edge switches are easy to spot: they have end hosts connected to them.

A *core switch* connects the edge switches in a bundle. It generally doesn't have hosts hooked up to it – only edge switches.

We start with the NIB from an L2 learning network and add `dpid` identifiers to the data structures.

The following code is in `multiswitch_topologies/network_information_base.py`:

```
class NetworkInformationBase(object):

  # hosts is a dictionary of MAC addresses to (dpid, port) tuples
  #  { "11:11:11:11:11:11": (1234867, 1) ...}
  hosts = {}

  # dictionary of live ports on each switch
  # { "11:11:11:11:11:11": [1,2], ...}
  ports = {}

  # For this incarnation, we assume the switch with dpid = 1 is the core switch
  core_switches = set([1])
  edge_switches = set([2,3,4,5])
```

```
# And hardcode the router ports
port_for_dpid = {2:1, 3:2, 4:3, 5:4}

# And the edge switches all have the same port as their uplink
edge_uplink_port = 5

def __init__(self, logger):
  self.logger = logger

def core_switch_dpids(self):
  return list(self.core_switches)

def edge_switch_dpids(self):
  return list(self.edge_switches)

def core_port_for_edge_dpid(self, dpid):
  return self.port_for_dpid[dpid]

def uplink_port_for_dpid(self, dpid):
  return self.edge_uplink_port

def is_internal_port(self, dpid, port_id):
  return dpid == self.core_switch_dpids()[0] or port_id == self.edge_uplink_port

def learn(self, mac, dpid, port_id):
  # Do not learn a mac twice
  if mac in self.hosts:
    return

  cd = (dpid, port_id)
  self.hosts[mac] = cd
  self.logger.info("Learning: "+mac+" attached to "+str(cd))

def port_for_mac_on_switch(self, mac, dpid):
  return self.hosts[mac][1] \
    if mac in self.hosts and self.hosts[mac][0] == dpid else None

def mac_for_port_on_switch(self, dpid, port_id):
  for mac in self.hosts:
    if self.hosts[mac][0] == dpid and self.hosts[mac][1] == port_id:
      return mac
  return None

def unlearn(self, mac):
  if mac in self.hosts:
    del self.hosts[mac]

def all_mac_port_pairs_on_switch(self, dpid):
  return [
    (mac, self.hosts[mac][1])
    for mac in self.hosts.keys() if self.hosts[mac][0] == dpid
  ]

def all_learned_macs_on_switch(self, dpid):
  return [
    mac for mac in self.hosts.keys() if self.hosts[mac][0] == dpid
  ]

def all_learned_macs(self):
  return self.hosts.keys()

def all_mac_dpid_pairs(self):
  return [
    (mac, self.hosts[mac][0]) for mac in self.hosts.keys()
  ]

def set_all_ports(self, switch_list):
  self.ports = switch_list

def add_port(self, dpid, port_id):
  if port_id not in self.ports[dpid]:
    self.ports.append(port_id)

def delete_port(self, dpid, port_id):
  if port_id in self.ports:
    self.ports.remove(port_id)

def all_ports_except(self, dpid, in_port_id):
  return [p for p in self.ports[dpid] if p != in_port_id]

def switch_not_yet_connected(self):
  return self.ports == {}
```

The core switch will have different rules than the edge switches, so we need to distinguish it from the others. To make things easy, we hard code the DPID of the core switch. (We'll see how to obtain this information dynamically in a later section). In a Mininet Tree topology, the core switch always has DPID 1.

So let's work from the edge switches inwards. Edge switches basically learn its directly connected hosts, like a regular L2 switch. In fact, the NetKAT rules for the edge switch look like L2 rules except for the addition of `SwitchEq`. For each learned MAC, we have a rule.

```
Filter(SwitchEq(dpid) & EthDstEq(mac)) >> SetPort(port)
```

Like an L2 switch, all packets bound to/from an unlearned MAC will go the controller and get flooded out all non-ingress ports. The big difference, which is practically invisible in the rule, is the packet will also go to the core switch. That way, if the packet is bound for a host on the opposite side of the network, it will hop to the core switch, then to the destination switch.

The edge switch policy is set in the application `multiswitch_topologies/multiswitch1.py`:

```
def policy_for_dest(self, dpid, mac_port):
  (mac, port) = mac_port
  return Filter(EthDstEq(mac)) >> SetPort(port)

def policies_for_dest(self, dpid, all_mac_ports):
  return [ self.policy_for_dest(dpid, mp) for mp in all_mac_ports ]

def policy_for_edge_switch(self, dpid):
  return \
    Filter(SwitchEq(dpid)) >> \
    IfThenElse(
      (EthSrcNotEq( self.nib.all_learned_macs_on_switch(dpid) ) |
      EthDstNotEq( self.nib.all_learned_macs_on_switch(dpid) )),
      SendToController("multiswitch"),
      Union( self.policies_for_dest(dpid, self.nib.all_mac_port_pairs_on_switch(dpid)) )
    )

def policy_for_edge_switches(self):
  return Union(self.policy_for_edge_switch(dpid) for dpid in self.nib.edge_switch_dpids())
```

And packets for unlearned MACs are handled by `packet_in`:

```
def packet_in(self, dpid, port_id, payload):
  nib = self.nib

  # If we haven't learned the ports yet, just exit prematurely
  if nib.switch_not_yet_connected():
    return

  pkt = Packet.from_payload(dpid, port_id, payload)
  src_mac = pkt.ethSrc
  dst_mac = pkt.ethDst

  # If we haven't learned the source mac, do so
  if nib.port_for_mac_on_switch( src_mac, dpid ) == None:
    # Don't learn the mac for packets coming in from internal ports
    if nib.is_internal_port(dpid, port_id):
      pass
    else:
      nib.learn( src_mac, dpid, port_id )
      self.update(self.policy())

  # Look up the destination mac and output it through the
  # learned port, or flood if we haven't seen it yet.
  dst_port = nib.port_for_mac_on_switch( dst_mac, dpid )
  if  dst_port != None:
    actions = SetPort(dst_port)
```

Here, we have to be a bit careful. In a one-switch setup, we simply learn all packets coming in on all ports. But in a multi-switch setup, we only want to learn MACs from packets coming from directly connected hosts. A packet coming from a core switch comes into the *uplink port* sometimes called an *internal port* because it's on an internal edge of the topology graph. Such a packet represent many MACs from many hosts. Even though those hosts may have been learned on their ingress switches, preventing learning here, timing may cause the rules not to be installed yet. To keep things perfectly clear, we explicitly deny learning of MAC's arriving on an internal port. (In our first carnation, these ports are hardcoded, but we'll make it dynamic later).

Now for the core switch. In our first incarnation, we'll take a naive approach. We'll make it like a repeater – packets coming in port p will be flooded out all non-p ports. This would cause problems in a topology with loops, but our fixed topology has no loops in it and so is safe. The core switch policy looks like this:

```
  def flood_core_switch(self, dpid, port_id):
    outputs = SetPort( self.nib.all_ports_except(dpid, port_id) )
    return Filter(PortEq(port_id)) >> outputs

  def policy_for_core_switch(self):
    dpid = self.nib.core_switch_dpids()[0]
    return Filter(SwitchEq(dpid)) >> \
      Union(self.flood_core_switch(dpid, p) for p in self.nib.ports[dpid])
```

And finally because the core and edge switch policies are disjoint, we can tie them together with a `Union`:

```
  def policy(self):
    return self.policy_for_core_switch() | self.policy_for_edge_switches()
```

We start up the Mininet topology and the Pingall pings all 16^2 host pairs in order. The Mininet topology `tree,2,4` means a tree toplogy with two levels and fanout four, meaning four edge switches and four hosts connected to each edge switch.

```
vagrant@frenetic:~$ sudo mn --topo=tree,2,4 --controller=remote --mac
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Adding switches:
s1 s2 s3 s4 s5
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s1, s5) (s2, h1) (s2, h2) (s2, h3) ....
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
```

```
*** Starting controller
c0
*** Starting 5 switches
s1 s2 s3 s4 s5 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16

   ... more Pings
```

6.2 Network-Wide Learning

Out first application works, but it's pretty inefficient. In particular:

- The core switch simply floods to all the switches, even when it knows where the desination switch and port is. If it knows the egress switch, it should send it to that switch.
- Packets with an unknown destination (but a known source) get flooded out the ingress switch. These packets come to the controller, but they don't really need to. They could just as well be flooded by a switch rule.

So let's write some rules to handle these cases. We need to be careful not to create loops or broadcast storms in the process.

First we'll do the core switch. For every destination MAC we have learned, we know its connected to switch sw at port p. Actually port p doesn't matter to the core switch – it just needs to know which core port will get it to the egress switch sw. Suppose that port is cp. Then the rule will look like:

```
Filter(SwitchEq(core) & EthDstEq(mac)) >> SetPort(cp)
```

First, let's factor out the port flooding policy. This policy will flood all packets to all other ports on the switch. It will be our fallback policy for all destination MACs we don't know.

The following code is in `multiswitch_topologies/multiswitch2.py`:

```
  def policy_flood_one_port(self, dpid, port_id):
    outputs = SetPort( self.nib.all_ports_except(dpid, port_id) )
    return Filter(PortEq(port_id)) >> outputs
  def policy_flood(self, dpid):
    return Union(
      self.policy_flood_one_port(dpid, p)
      for p in self.nib.ports[dpid]
    )
```

Next, this policy constructs the learned MAC rules. Since there's no overlap here, a simple `Union` can be used to combine them. (We have factored out the `SwitchEq` filter because we'll use it for the entire switch rule.)

```
  def policy_for_dest_on_core(self, mac, dpid, core_dpid):
    return Filter(EthDstEq(mac)) >> \
      SetPort( self.nib.core_port_for_edge_dpid(dpid) )

  def policies_for_dest_on_core(self, core_dpid):
    return Union(
      self.policy_for_dest_on_core(mac, dpid, core_dpid)
      for (mac, dpid) in self.nib.all_mac_dpid_pairs()
    )
```

Finally, the entire core switch policy puts them together. Flooding rules and learned MAC rules have considerable overlap – you can imagine a packet destined for a learned MAC (which matches a learned MAC rule) which comes in a core switch port (which matches a flooding rule). So we use `IfThenElse` to disambiguate them. And here we pop on the filter for the core switch, neatly factoring it out of the individual rules:

```
  def policy_for_core_switch(self):
    core_dpid = self.nib.core_switch_dpids()[0]
    return \
      Filter(SwitchEq(core_dpid)) >> \
      IfThenElse(
        EthDstEq(self.nib.all_learned_macs()),
        self.policies_for_dest_on_core(core_dpid),
        self.policy_flood(core_dpid)
      )
```

OK, now for the edge switches. We basically want to refactor the rule into the following psuedocode:

```
if unlearned(src_mac) and not internal_port(port):
  learn(src_mac)    # Send to controller
elif on_switch(dpid, dst_mac):
  send_directly()
else:
  flood()
```

Like our previous application, we have to be careful not to learn MACs arriving on internal ports. But here, since we're relying on the rules to do the flooding, not the controller, we have to explicitly filter out internal ports from the learning process with NetKAT predicates.

The new edge switch rule follows this psuedocode skeleton:

```
 def policy_for_edge_switch(self, dpid):
   nib = self.nib
   return \
     Filter(SwitchEq(dpid)) >> \
     IfThenElse(
       EthSrcNotEq( nib.all_learned_macs_on_switch(dpid) ) &
         PortNotEq(nib.uplink_port_for_dpid(dpid)),
       SendToController("multiswitch"),
       IfThenElse(
         EthDstEq( nib.all_learned_macs_on_switch(dpid) ),
         Union( self.policies_for_dest(dpid, nib.all_mac_port_pairs_on_switch(dpid)) ),
         self.policy_flood(dpid)
       )
     )
```

The remaining code stays the same. Now doing a Ping All in Mininet is a much faster experience – once all the ports are learned (h1 pings each host in turn first, so that does the learning), no packets hit the controller. Very nice!

> *NOTE: ARP doesn't work very well with this, but it's unclear why. h1 sends an ARP request for h2, . . . but it's unclear whether the request is flooded properly by the controller, or whether the reply comes back. This is true for all hosts, intranet or internet. But once the flow table is complete, everything works fine including (presumably) ARP requests. Use the arp command switch in Mininet and the problem doesn't happen at all.*

6.3 Calculating Loop-Free Paths

To get to this point, we've had to effectively hard-code the network topology into the program. The next step is to extend it to work with *any* topology.

The basic problem is: given two host endpoints and a topology, what is the shortest path through the switches connecting those hosts? Once you have that, you can generate a rule for each switch – basically the rule will forward packets one hop closer to the destination. In effect, our first programs hard-coded the shortest path solutions because they were trivial for that topology:

- If the destination is on the same switch as the source, just forward it out the right port.

- If it's not, forward it to the core switch, then to the destination edge switch, then out the right port.

If you had never worked with computer networks you'd think we could stop there. Computing shortest paths is an easy graph theoretical problem, and we could just have a library do it. Not only that, you could just connect every switch to every other switch, and the number of hops between any two hosts would be two.

The only trouble is ... there is flooding. If every switch is connected to every switch, and you copy a packet to all the ports at once, it'll eventually arrive back at the source switch and get flooded again and again. This is called a *broadcast storm* and it'll eventually grind the network to a halt. And you can't just get rid of flooding. Even if get rid of Ethernet broadcasts, flooding is a major part of L2 learning – it's how you deal with packets going to as-yet-unlearned destinations.

One way to have total connectivity without loops is to calculate the *spanning tree* of the topology. In a spanning tree, all switches are indirectly connected by some path, but the total collection of paths contains no loops. As long as flooding occurs only along the paths of the spanning tree (and not out the ingress port too, of course), no broadcast storms are possible. This spanning tree might not be optimal in a shortest path sense – it might force some host-to-host paths to be longer than they could be. But the added safety is well-worth the cost.

Traditional Ethernet networks use a protocol like STP to cooperatively decide on the spanning tree. STP is interesting but it's complex and slow to converge. The problem is no one switch has a total view of the network – it only knows its neighbors.

SDN doesn't have that problem. The controller effectively can use a global view of the network, from which it calculates the spanning tree and pushes the resulting rules to all the switches at once.

But how does it get that global view of the topology, especially when switches can dynamically enter and leave the network? Let's defer that question to the next section, and for now focus on a fixed toplogy, but one that can be defined outside of the program itself. Once we build the infrastructure that calculates the spanning tree and forwarding rules from this topology, we can generalize it to dynamic toplogies.

There are many different file formats for specifying a network topology. We're going to use the popular *GraphViz* file format. GraphViz files can be read from lots of different programming languages, and can easily be turned into visible graphs. In Python, it's also easy to interface GraphViz files with the popular graph theory library NetworkX. We can then leverage NetworkX's spanning tree and shortest-path algorithms rather than writing our own. Nice!

To make this topology more interesting, we're going to use a Mininet tree topology with depth 3 and fanout 3. That'll give us 1 at the top level, three at the second level, and nine at the third level for a total of $1 + 3 + 9 = 13$ switches and three hosts connected to the 9 level 3 switches, for a total of 27 hosts:

```
vagrant@frenetic:~$ sudo mn --topo=tree,3,3 --controller=remote --mac
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25 h26 h27
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13
*** Adding links:
(s1, s2) (s1, s6) (s1, s10) (s2, s3) (s2, s4) (s2, s5) (s3, h1) (s3, h2) (s3, h3) (s4, h4)
(s4, h5) (s4, h6) (s5, h7) (s5, h8) (s5, h9) (s6, s7) (s6, s8) (s6, s9) (s7, h10) (s7, h11)
(s7, h12) (s8, h13) (s8, h14) (s8, h15) (s9, h16) (s9, h17) (s9, h18) (s10, s11) (s10, s12)
(s10, s13) (s11, h19) (s11, h20) (s11, h21) (s12, h22) (s12, h23) (s12, h24) (s13, h25)
(s13, h26) (s13, h27)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25 h26 h27
*** Starting controller
c0
*** Starting 13 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 ...
*** Starting CLI:
mininet>
```

Now the Mininet tree toplogy has no loops, but we can add one on the sly:

```
mininet> py net.addLink(s2, s6)
<mininet.link.Link object at 0x7fd432ace810>
mininet> link s2 s6 up
mininet> py s2.attach('s2-eth5')
```

Next we write a GraphViz file that mirrors this topology, including the extra edge. So here is our DOT file, from `multiswitch_topologies/multiswitch_topo.dot`:

```
strict graph multiswitch {
  /* Level 1 */
  1 ;

  /* Level 2 */
  2 ;
  6 ;
  10 ;

  /* Level 3 */
  3 ;
  4 ;
  5 ;

  7 ;
  8 ;
  9 ;

  11 ;
  12 ;
  13 ;

  /* Level 1 -> Level 2 */
  1 -- 2 [ src_port = 1,  dport = 4 ];
  1 -- 6 [ src_port = 2,  dport = 4 ];
  1 -- 10 [ src_port = 3,  dport = 4 ];

  /* Level 2 -> Level 3 */
  2 -- 3 [ src_port = 1,  dport = 4 ];
  2 -- 4 [ src_port = 2,  dport = 4 ];
  2 -- 5 [ src_port = 3,  dport = 4 ];
  2 -- 6 [ src_port = 5,  dport = 5 ];  /* Extra - forms loop */

  6 -- 7 [ src_port = 1,  dport = 4 ];
  6 -- 8 [ src_port = 2,  dport = 4 ];
  6 -- 9 [ src_port = 3,  dport = 4 ];

  10 -- 11 [ src_port = 1,  dport = 4 ];
  10 -- 12 [ src_port = 2,  dport = 4 ];
  10 -- 13 [ src_port = 3,  dport = 4 ];
}
```

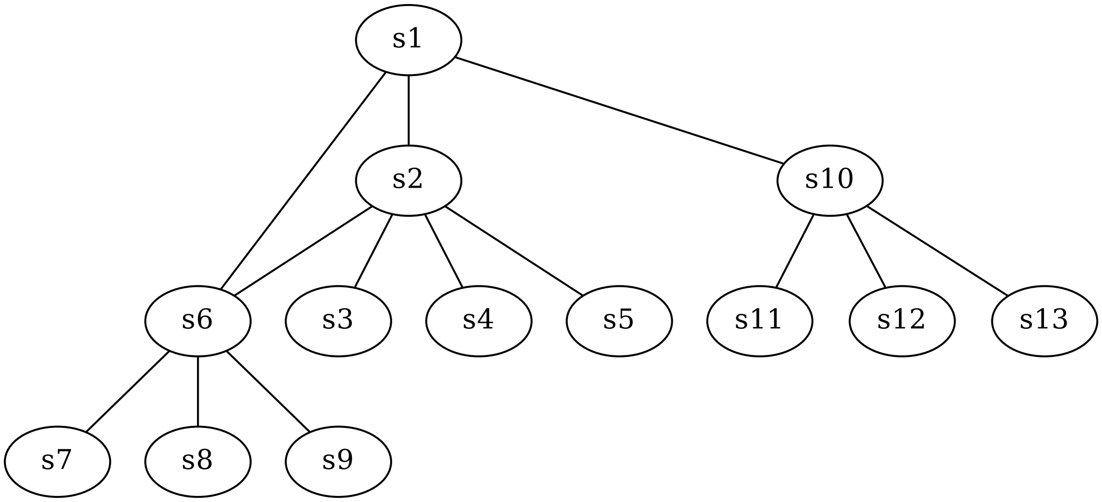
This file is pretty straightforward to read:

- The switches are listed by themselves first. We'll keep it simple and name each one after the DPID that Mininet will naturally assign to it: e.g. s13 will have DPID 13.
- The edges are listed next, each edge being a pair of switches separated by two dashes.
- Each edge has two attributes: `src_port` is the port used on the first switch and `dport` is the port on the second switch.

And you can turn this into a PDF-formatted graph with the command:

```
dot -Tpdf multiswitch_topo.dot -o multiswitch_topo.pdf
```

Which looks like this:



For the code, we'll start with the NIB. We'll still keep the distinction between core and edge switches: the edge switches being where hosts are connected, and core switches being those on the internal portion fo the network. Only this time we'll leave these sets blank and fill the in from the topology.

The following code is in `multiswitch_topologies/network_information_base_from_file.py`:

```
core_switches = set()
edge_switches = set()
```

Next we build a data structure for holding the direct connections between hosts and switches, or between switches and switches. This is a dictionary whose keys are MAC addresses for hosts and DPID's for switches (there is no overlap between the two). The value for each key is itself a dictionary of destinations reachable from that source (the keys are, again, MAC addresses and DPID's), and the port they go out on. Each host is assumed to have one switch connection on port 0, although this is really a placeholder because a host is not a switch and therefore has no rules of its own: it can only send packets to its connected switch.

```
# For each switch, returns a dictionary of destination switches to ports.
# So in the following, a packet in 18237640987 can get to switch 9287354
# by going out port 7.  This effectively turns the undirected edges of the
# topo graph into bidirectional edges.  Later we'll add learned MAC addresses
# to this dictionary as well.
#  { 18237640987: { 9287354: 7, 09843509: 5, ... }}
port_mappings = {}
```

Note that this data is separate from the spanning tree, so just because we have a direct connection from one host/switch to another doesn’t mean we’ll actually use it!

The `ports` data structure is the same one we’ve used in previous network applications, but we need a parallel structure to track *enabled* ports. When we calculate the spanning tree, it’s possible that some core switch ports need to be virtually disabled because they form redundant paths. For example, in our network, the edge s2-s6 is not on the spanning tree, and so the ports for that edge (port 5 on s2 and port 5 on s6) will need to be virtually disabled - no floodable traffic will go that port, and all packets arriving on that port (there shouldn’t be any, but you never know) will be dropped.

```
# On core switches, we pretend that edges not living on the spanning
# tree are disabled - we don't send or flood traffic to it, and we drop
# all incoming traffic from it.  So this structure keeps track of all
# ports that are enabled - each enabled_ports[sw] is a subset of ports[sw]
enabled_ports = {}
```

The uplink port on each of the edge switches needs to be calculated and tracked, since MAC learning cannot occur on that port.

```
# Uplink port from each edge switch to the nearest core switch
uplink_port = {}
```

The `init` function is more involved, reading the topology from our DOT file and building the intermediate data structures.

```
def __init__(self, logger, topology_file=”multiswitch_topo.dot”):
  self.logger = logger

  self.logger.info(”---> Reading Topology from ”+topology_file)
  self.agraph = pgv.AGraph(topology_file)

  # It's faster to denormalize this now
  self.logger.info(”---> Remembering internal ports”)
  switches = [ int(sw) for sw in self.agraph.nodes()]
  for e in self.agraph.edges():
    # Parse the source and destination switches
    source_dpid = int(e[0])
    dest_dpid = int(e[1])
    source_port = int(e.attr[”src_port”])
    dest_port = int(e.attr[”dport”])

    # Add a mapping for the forward directions source->dest
    self.add_port_mapping(source_dpid, dest_dpid, source_port)

    # Add a mapping for the reverse direction dest->source
    self.add_port_mapping(dest_dpid, source_dpid, dest_port)
  # Calculate the core/edge attribute.  Edge switches have only one
  # connection to another switch
  for sw in self.port_mappings:
    if len(self.port_mappings[sw]) == 1:
      self.edge_switches.add(sw)
      # Edge switches have only one entry in port_mappings[dpid], so we get it here
      connected_sw = self.port_mappings[sw].keys()[0]
      self.uplink_port[sw] = self.port_mappings[sw][connected_sw]
    else:
      self.core_switches.add(sw)

  self.logger.info("---> Calculating spanning tree")
  nxgraph = nx.from_agraph(self.agraph)
  self.nx_topo = nx.minimum_spanning_tree(nxgraph)
  nx.write_edgelist(self.nx_topo, sys.stdout)

  self.logger.info("---> Enabling only those ports on the spanning tree")
  for (from_dpid, to_dpid) in self.nx_topo.edges():
    # We look up the port mapping from the port-mapping dictionary instead of
    # from the graph attributes because NetworkX flips the src and dest node
    # arbitrarily in an undirected graph
    from_dpid_int = int(from_dpid)
    to_dpid_int = int(to_dpid)
    from_port = self.port_mappings[from_dpid_int][to_dpid_int]
    if from_dpid_int not in self.enabled_ports:
      self.enabled_ports[from_dpid_int] = []
    self.enabled_ports[from_dpid_int].append(from_port)

    to_port = self.port_mappings[to_dpid_int][from_dpid_int]
    if to_dpid_int not in self.enabled_ports:
      self.enabled_ports[to_dpid_int] = []
    self.enabled_ports[to_dpid_int].append(to_port)
```

The last few lines calculate the minimal spanning tree for the topology (minimal meaning using the least cost, and in our case all edges have cost 1). Then we calculate all the enabled ports whose edges live on this spanning tree. We print the spanning tree to the log for reference:

```
vagrant@frenetic:~/manual/programmers_guide/code/multiswitch_topologies$ python multiswitch3.py
2016-05-27 14:49:11,162 [INFO] ---> Reading Topology from multiswitch_topo.dot
2016-05-27 14:49:11,163 [INFO] ---> Remembering internal ports
2016-05-27 14:49:11,164 [INFO] ---> Calculating spanning tree
11 10 {'dport': u'4', 'src_port': u'1'}
10 1 {'dport': u'4', 'src_port': u'3'}
10 13 {'dport': u'4', 'src_port': u'3'}
10 12 {'dport': u'4', 'src_port': u'2'}
1 2 {'dport': u'4', 'src_port': u'1'}
1 6 {'dport': u'4', 'src_port': u'2'}
```

```
3 2 {'dport': u'4', 'src_port': u'1'}
2 5 {'dport': u'4', 'src_port': u'3'}
2 4 {'dport': u'4', 'src_port': u'2'}
7 6 {'dport': u'4', 'src_port': u'1'}
6 8 {'dport': u'4', 'src_port': u'2'}
6 9 {'dport': u'4', 'src_port': u'3'}
Starting the tornado event loop (does not return).
```

Note that the edges from s1-s2 and s1-s6 are present, but the edge s2-s6 is not listed. It has been dropped calcuating the minimal spanning tree. This final topology graph now has no loops.

This basic graph won't change. When we connect hosts to the edge switches, it doesn't change the spanning tree because hosts only add one edge to the graph. That edge must be part of the spanning tree in order to reach the new host, but it cannot form a loop because the rest of the graph has no loops and the new host was previously unconnected.

Knowing all this, we can redo the MAC learning function:

```
  def learn(self, mac, dpid, port_id):
    # Do not learn a mac twice
    if mac in self.hosts:
      return

    # Compute next hop table: from each switch, which port do you need to go next to get to destination?
    self.nx_topo.add_node(mac)
    dpid_str = str(dpid)
    self.nx_topo.add_edge(dpid_str, mac)
    # Note we don't need a mapping from mac to switch because we never see packets going INTO a host
    self.port_mappings[dpid][mac] = port_id

    # Return shortest paths from each source in the graph to mac in the form
    # [ src: to1, from2: to2 ..., to[n]: dest ]
    spaths = nx.shortest_path(self.nx_topo, None, mac, None)
    next_hop_table = { }
    for from_dpid in self.switches():
      # If we're on the switch the Mac is connected to, just add the next_hop
      if from_dpid == dpid:
        next_hop_table[from_dpid] = port_id
      else:
        sw = str(from_dpid)
        # If there are no paths from sw, just skip it
        if sw in spaths:
          next_sw = spaths[sw][1]  # element 0 is the start, element 1 is the next hop
          # Convert this back to a dpid
          next_dpid = int(next_sw)
          # The next hop switch along the shortest path from sw to mac.
          next_hop_table[from_dpid] = self.port_mappings[from_dpid][next_dpid]

    cd = (dpid, port_id, next_hop_table)
    self.hosts[mac] = cd
    self.logger.info("Learning: "+mac+" attached to "+str(cd))
```

The main addition is the `next_hop` dictionary to the MAC. This dictionary basically tells you how to go from any switch to this MAC. Knowing this, you can easily trace a path from any source host through as set of switches, and finally to this destination MAC. The path is guaranteed not to have any loops no matter which switch you start from.

A set of utility functions gathers important information for calculating the switch forwarding rules:

```
  def core_switch_dpids(self):
    return list(self.core_switches)

  def edge_switch_dpids(self):
    return list(self.edge_switches)

  def core_port_for_edge_dpid(self, dpid):
    return self.port_for_dpid[dpid]

  def next_hop_port(self, mac, core_dpid):
    return self.hosts[mac][2][core_dpid]

  def uplink_port_for_dpid(self, dpid):
    return self.uplink_port[dpid]

  def is_internal_port(self, dpid, port_id):
    return (dpid in self.core_switches) or (port_id == self.uplink_port_for_dpid(dpid))

  def switches(self):
    return self.ports.keys()
```

And a new function returns the enabled ports for a particular switch, so rational flooding can occur. This is important for the core switches. The edge switches merely flood to all ports since the host ports and the uplink ports are always part of the spanning tree.

```
  def all_enabled_ports_except(self, dpid, in_port_id):
    ports_to_flood = self.enabled_ports[dpid] if dpid in self.core_switches else self.ports[dpid]
    return [p for p in ports_to_flood if p != in_port_id]
```

Our new learning switch program won't change much. It'll delegate most of the complex stuff to the NIB, which has precalculated all the routes for us.

The following code is in `multiswitch_topologies/multiswitch3.py`:

```
  def policy_for_dest_on_core(self, mac, core_dpid):
    return Filter(EthDstEq(mac)) >> \
      SetPort( self.nib.next_hop_port(mac, core_dpid) )

  def policies_for_dest_on_core(self, core_dpid):
    return Union(
      self.policy_for_dest_on_core(mac, core_dpid)
      for mac in self.nib.all_learned_macs()
    )
```

In this example, the core switch rule calculation has been delegated to the `next_hop` calculations in the NIB.

The flooding policy for edge switches doesn't change because each edge switch has only hosts (which must receive the flooded packets) and one uplink port (which also must receive the flooded packet), modulo the ingress port as always. Core switch flooding, however, must respect the spanning tree so as not to introduce loops:

```
def policy_flood_one_port(self, dpid, port_id):
  outputs = SetPort( self.nib.all_enabled_ports_except(dpid, port_id) )
  return Filter(PortEq(port_id)) >> outputs
```

And with that, our network keeps itself loop-free and functional. If the topology changes, we simply change the GraphViz file and restart that network application, which then recalculates the spanning tree and relearns the host MAC addresses. One can build a very large L2 network with lots of switches connected arbitrarily in this manner. And it's fairly easy to extend this program for fault tolerance – if a port on a core switch goes down, we can update the graph, recalculate the spanning tree, enable previously-disabled ports and keep the network running smoothly.

6.4 Dynamic Topology

The last step in our journey is to dynamically calculate the topology. Our job is actually half-done already – we add and drop hosts dynamically as they appear or disappear. Because each learned MAC has a next hop table, when the MAC disappears from the learned host list, the rules disappear as well.

> *NOTE: This seciton is dependent on the Frenetic GUI server, which is still in Alpha. It relies on the REST call to /topology, which returns a JSON list of switches and edges. The problem is it's a non-trivial exercise to make this work alongside other network programs, so it's unclear whether this is workable in this setup. It may be more doable to just have multiswitch4.py send out its own probes, like the Python discovery app does.*

Chapter 7

Routing

7.1 Design

Up until now, we've been dealing with OSI Layer 2 technologies – those that operate at the Ethernet packet level. Now we'll step one layer up the stack to Layer 3: Internet Protocol or IP.

From the IP perspective, the Internet is just a bunch of LAN's organized into networks, then further divided into subnets. For our purposes, we'll concentrate on IP Version 4 or IPv4, which is currently the most popular of the two IP versions (the other being IP Version 6 or IPv6).

A particular IPv4 address, for example 10.0.1.3, may be part of a subnet with other hosts. We may set up a subnet labelled 10.0.1.0/24, which means the first 24 bits comprise the subnet number, and the last $32 - 24 = 8$ bits are the host number. In our example, the subnet number is 10.0.1 and the host is 3. Because the host number is 8 bits, this means our example host lives in a subnet with up to 256 neighbors. Some neighbors are reserved:

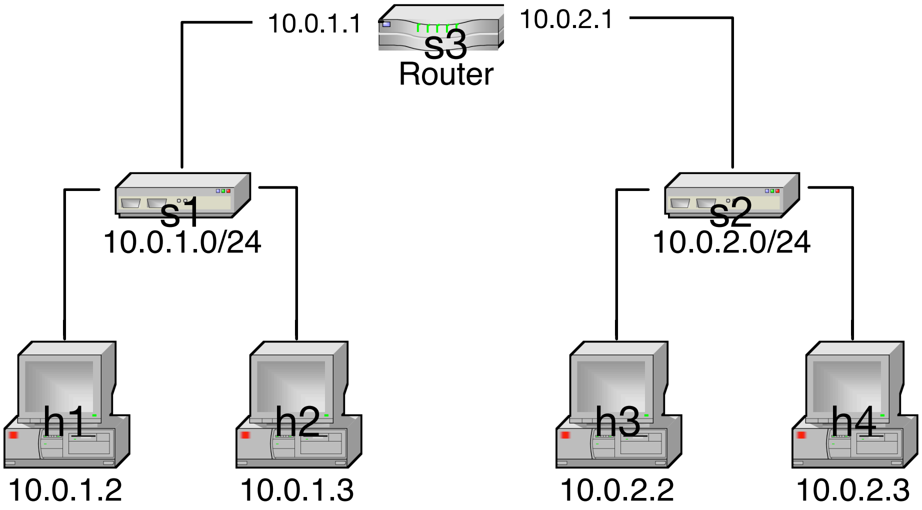
Host 0 in this case 10.0.1.0, usually has no host assigned to it.

Host 1 in this case 10.0.1.1, is usually the default gateway of the subnet, which we'll see in a minute.

Host n - 1 in this case 10.0.1.255, is the subnet broadcast address.

This leaves you with n - 3 "real" neighbor hosts. You can think of a subnet as a gated community where the most common action is to talk with your neighbors, but not with those outside your subnet. To do the latter, you need an intermediary . . . in IP, that intermediary is a router.

So suppose we have two subnets, each with two hosts, organized like this:



There is no such thing as speaking IP *natively* over a network. You must speak the wire protocol, which in most cases is Ethernet. Because IP applications like your browser and Ping can only speak in IP addresses, there must be a translation mechanism between IP addresses and Ethernet addresses. That translation mechanism is Address Resolution Protocol or ARP.

A typical conversation between two hosts on the same subnet, looks something like this:

1. Host 10.0.1.2 wants to communicate with 10.0.1.3, but doesn't know it's MAC address.
2. It broadcasts over Ethernet an ARP request, "Who has 10.0.1.3" ?
3. Assuming 10.0.1.3 is up, it sends back an ARP reply, "10.0.1.3 is at 11:22:33:44:55:66", which is presumably its own MAC address.
4. Host 10.0.1.2 then constructs an Ethernet packet with destination 11:22:33:44:55:66 and sends it off.

We haven't had to think about ARP in previous chapters, because the hosts have seamlessly handled it for us. Our L2 learning switch handles Ethernet broadcasts and Ethernet unicasts just fine, so every part of the conversation above was handled by it seamlessly.

Now, throw a router into the mix and the conversation gets slightly more complicated. Host IP stacks now distinguish between hosts on their own subnet and hosts on other subnets. If 10.0.1.2 wants to talk to 10.0.2.3, the host won't just send out an ARP request for 10.0.2.3 – it's on another network. Hosts can have routing tables to tell it where to direct inter-network traffic, but in most cases the routing table contains one entry: the default gateway. The default gateway is usually set by DHCP, but it's generally a special IP address on the subnet where the router lives. In this case the conversation becomes:

1. Host 10.0.1.2 wants to communicate with 10.0.2.3, but doesn't know it's MAC address. Since 10.0.2.3 is on different subnet, and there's no routing table entry for a subnet including 10.0.2.3, it decides to send to a default gateway 10.0.1.1
2. It broadcasts over Ethernet an ARP request, "Who has 10.0.1.1?"
3. The router sends back an ARP reply, "10.0.1.1 is at 11:00:00:00:00:00", which is the MAC address of the router port hooked into subnet 10.0.1.0/24.
4. Host 10.0.1.1 then constructs an Ethernet packet with destination 11:00:00:00:00:00 and sends it off.

And now the router can do its thing:

1. The destination IP is 10.0.2.3, and the router knows it has subnet 10.0.2.0/24 on port 2. But it doesn't know the MAC address of 10.0.2.3.
2. The router broadcasts an ARP request over port 2 "Who has 10.0.2.3?"
3. That host responds with an ARP reply "10.0.2.3 is at ff:ee:dd:cc:bb:aa", which is its own MAC address.
4. Router constructs an Ethernet packet with the router MAC address connected to that subnet as its source, and ff:ee:dd:cc:bb:aa as its destination and sends it off.

A couple of things to notice here:

- Only IP traffic gets routed.
- Only Ethernet sources and destinations are changed in the packet. The IP addresses stay the same no matter where they are in the process.
- A router must buffer packets until the ARP replies return. For subsequent requests, it caches the IP-to-MAC translation, like a learning switch does with MACs.
- If the router receives a packet bound for a network not directly connected to it, the router itself has a default gateway it can send to.

So the router does three basic things: answers ARP requests, sends ARP requests, and routes a packet to the "next hop". Of course real routers do much more than that: they maintain routing tables, translate network protocols, drop blatantly malicious traffic, and so on. But we'll concentrate on the three core router functions here.

7.2 Modeling The Topology

Every OpenFlow enabled device is called a *switch*, but you should not confuse it with a traditional L2 switch. An OpenFlow switch can model just about any network device including firewalls, load balancers, and – as we'll see in this chapter – routers.

In Chapter **??**, we saw two ways of modeling the network topology: statically with a `.dot` file and dynamically via Frenetic. One advantage of a static topology is you can share it between Mininet and your application. That way you can model more difficult topologies completely, and only change one file to change the design.

So here is our DOT file, from `routing/topology.dot`:

```
strict graph routing {
  s_router [ dpid="01:01:00:00:00:00:00", router=true ];
  s1 [ dpid="01:01:00:00:00:00:01" ];
  s2 [ dpid="01:01:00:00:00:00:02" ];

  s_router -- s1 [ src_port = 1, dport = 3 ];
  s_router -- s2 [ src_port = 2, dport = 3 ];

  h1 [ mac="00:00:01:00:00:02", ip="10.0.1.2/24", gateway="10.0.1.1" ];
  h2 [ mac="00:00:01:00:00:03", ip="10.0.1.3/24", gateway="10.0.1.1" ];
  s1 -- h1 [ src_port = 1, dport = 0 ];
  s1 -- h2 [ src_port = 2, dport = 0 ];

  h3 [ mac="00:00:02:00:00:02", ip="10.0.2.2/24", gateway="10.0.2.1" ];
  h4 [ mac="00:00:02:00:00:03", ip="10.0.2.3/24", gateway="10.0.2.1"];
  s2 -- h3 [ src_port = 1, dport = 0 ];
  s2 -- h4 [ src_port = 2, dport = 0 ];
}
```

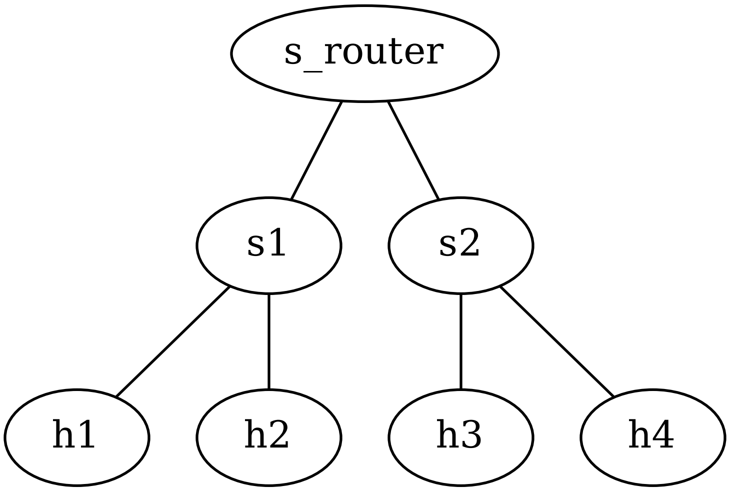
We've added a few more attributes to accommodate IP. In particular:

router is set to true on the device acting as the router

ip addresses are assigned to each host

gateway addresses is the default gateway to which the host sends packets. All packets not bound for hosts in the same subnet go here.

Here's the GraphViz generated diagram from the above:



Until now we've been using Mininet's predefined topologies: simple and tree. We can define a custom Mininet toplogy by writing some Python code calling the Mininet API's.

The following code is in `routing/mn_dot_topology.py`:

```
import re, sys, os
from networkx import *
import pygraphviz as pgv
from net_utils import NetUtils

# Mininet imports
from mininet.log import lg, info, error, debug, output
from mininet.util import quietRun
from mininet.node import Host, OVSSwitch, RemoteController
from mininet.cli import CLI
from mininet.net import Mininet

def start(ip="127.0.0.1",port=6633):

  ctrlr = lambda n: RemoteController(n, ip=ip, port=port, inNamespace=False)
  net = Mininet(switch=OVSSwitch, controller=ctrlr, autoStaticArp=False)
  c1 = net.addController('c1')

  topo_agraph = pgv.AGraph("topology.dot")
  for node in topo_agraph.nodes():
    if node.startswith("s"):
      net.addSwitch(node, dpid=str(node.attr['dpid']))
    else:
      net.addHost(
        node,
        mac=node.attr['mac'],
        ip=node.attr['ip'],
        defaultRoute="dev "+node+"-eth0 via "+node.attr['gateway']
      )

  for link in topo_agraph.edges():
    (src_node, dst_node) = link
    net.addLink(src_node, dst_node,
      int(link.attr['src_port']),
      int(link.attr['dport'])
    )

  # Set up logging etc.
  lg.setLogLevel('info')
  lg.setLogLevel('output')

  # Start the network
  net.start()

  # Enter CLI mode
  output("Network ready\n")
  output("Press Ctrl-d or type exit to quit\n")
  CLI(net)

os.system("sudo mn -c")
start()
```

This program relies on the `agraph` API, which we used in Chapter **??**, and the Mininet API, documented at: http://mininet.org/api/annotated.html. The interesting calls are:

addSwitch() which adds a new Mininet switch with a particular DPID.

addHost() which adds a host

addLink() which adds virtual "wire" between a host and a switch, or two switches.

In this program, we're careful to set the port ids to some defaults. Although that's not technically necessary, since L2 learning will establish a table of MAC-to-port mappings, it makes debugging easier.

To run this Mininet topology, you simply run this python file as root:

```
vagrant@frenetic:~/manual/programmers_guide/code/routing$ sudo python mn_dot_topology.py
*** Removing excess controllers/ofprotocols/ofdatapaths/pings/noxes

  ... a lot of cleanup messages

*** Cleanup complete.
Unable to contact the remote controller at 127.0.0.1:6633
```

```
Network ready
Press Ctrl-d or type exit to quit
mininet>
```

Finally, we set up a static routing table. Subnets directly connected to a router are usually configured statically in this manner, while indirectly-connected subnet entries are handled by a route advertising protocol like OSPF. Our test network won't have the latter.

We use simple JSON to define the subnets in `routing/routing_table.json`:

```
[
  { "subnet":"10.0.1.0/24",
    "router_mac":"01:01:01:01:01:01",
    "router_port": 1,
    "gateway": "10.0.1.1"
  },
  { "subnet":"10.0.2.0/24",
    "router_mac":"02:02:02:02:02:02",
    "router_port": 2,
    "gateway": "10.0.2.1"
  }
]
```

This information isn't needed to *build* the network, but it's crucial for its operation.

7.3 A Better Router

One of the problems with traditional routing is the ARP process. The router must have gobs of memory to buffer packets waiting for ARP replies . . . and those replies may never arrive! But here, SDN can help us build a faster router.

In Chapter 6 we saw how a global network view helps us build a loop-free network without the hassle of spanning tree. We can use this same global network view to build routing without relying as much on ARP.

The key is in the L2 learning tables. If 10.0.1.2 wants to communicate with 10.0.2.3, we probably know both of their MAC addresses. 10.0.1.2 cannot simply address its packet directly to that MAC because it's not on the same switch (and Ethernet, by itself, is not routable). But the router can use 10.0.2.3's MAC address to perform the second hop. It can skip the buffering and ARP request step altogether.

But what if the MAC address of 10.0.2.3 is *not* known? This looks like the case in the L2 learning switch when we simply "punt" and flood the packet over all ports. Unfortunately the router can't do that with L3 packets. If you simply stamp an Ethernet broadcast address in the same packet as the destination IP, the end hosts will see the packet, note that the IP and MAC addresses don't match, and drop the packet. In this case, the router *does* need to enqueue packets, send the ARP request, monitor the response, and release the packets when their destination MAC is known.

Under these conditions, the L2 switches need very few modifications. When a host needs to send an internetwork packet, it always starts by sending the packet to the default gateway. From that perspective, the default gateway is no different than any other host. If the sender knows its MAC address, it simply sends it. If it doesn't, it broadcasts an ARP request for it and the router responds.

The router needs to implement a rule for each learned MAC in the network. For each mac and address ip, you first determine the router port rp and the MAC address of that port $rtrmac$. That can be calculated since we know which subnet is connected to each router port. Then we write the rule:

```
Filter(EthTypeEq(0x800) & IP4DstEq(ip)) >> \
  SetEthSrc(rtrmac) >> SetEthDst(mac) >> SetPort(rp)
```

Two more things: we need to catch all remaining unlearned IP destinations, and we need to catch all ARP requests so that we can answer requests for the default gateway:

```
Filter(EthTypeEq(0x800, 0x806) ) >> SendToController("router")
```

Note that we don't need to *answer* all ARP requests, necessarily . . . and in fact we'll see ARP requests for intra-network hosts because they are broadcast over the subnet. We can ignore those since they'll be answered by the host itself. The only ones we reply to are those for the default gateway.

7.4 Modularization

When we get done, our network application will perform the job of both the switches and the router. Up until now, we have written our applications as one subclass of `Frenetic.App`, but here it makes sense to modularize the application into a switch part and a router part. We call these handlers, and they implement the same method signatures that Frenetic applications do. They are not subclasses of `Frenetic.App` – making them subclasses would give them each their own event loops and asynchronous calls, which simply makes things too complicated. However, they could be turned into their own freestanding Frenetic applications simply by making them subclasses.

The main application looks like this, from `routing/routing1.py`:

```
import sys,logging,datetime
from network_information_base import *
from frenetic import *
from frenetic.packet import *
from switch_handler import *
from router_handler import *
from tornado.ioloop import IOLoop

class RoutingApp(frenetic.App):

  client_id = "routing"

  def __init__(self, topo_file="topology.dot", routing_table_file="routing_table.json"):
    frenetic.App.__init__(self)
    self.nib = NetworkInformationBase(logging, topo_file, routing_table_file)

    self.switch_handler = SwitchHandler(self.nib, logging, self)
    self.router_handler = RouterHandler(self.nib, logging, self)

  def policy(self):
    return Union([
      self.switch_handler.policy(),
      self.router_handler.policy(),
    ])

  def update_and_clear_dirty(self):
    self.update(self.policy())
    self.nib.clear_dirty()

  def connected(self):
    def handle_current_switches(switches):
      logging.info("Connected to Frenetic - Switches: "+str(switches))
      self.nib.set_all_ports( switches )
      self.update( self.policy() )
    self.current_switches(callback=handle_current_switches)

  def packet_in(self, dpid, port, payload):
    pkt = Packet.from_payload(dpid, port, payload)
    self.switch_handler.packet_in(pkt, payload)
    self.router_handler.packet_in(pkt, payload)

    if self.nib.is_dirty():
      logging.info("Installing new policy")
      # This doesn't actually wait two seconds, but it serializes the updates
      # so they occur in the right order
      IOLoop.instance().add_timeout(datetime.timedelta(seconds=2), self.update_and_clear_dirty)

  def port_down(self, dpid, port_id):
    self.nib.unlearn( self.nib.mac_for_port_on_switch(dpid, port_id) )
    self.nib.delete_port(dpid, port_id)
    self.update_and_clear_dirty()

  def port_up(self, dpid, port_id):
    # Just to be safe, in case we have old MACs mapped to this port
    self.nib.unlearn( self.nib.mac_for_port_on_switch(dpid, port_id) )
    self.nib.add_port(dpid, port_id)
    self.update_and_clear_dirty()

if __name__ == '__main__':
  logging.basicConfig(\
    stream = sys.stderr, \
    format='%(asctime)s [%(levelname)s] %(message)s', level=logging.INFO \
  )
  app = RoutingApp()
  app.start_event_loop()
```

Notice how it creates one NIB, and passes these to both switch and router handlers. This allows them to share state. But learning a new MAC on the switch should trigger a policy recalculation on the router. Rather than coding this dependency into the router (which then couples it to the switch handler), we handle the recalculation here.

The recalculation uses a trick from the Tornado Python library. If you simply call the `update_and_clear_dirty()` function, many successive packets may cause the updates to happen in a random order since the requests are handled asynchronously. This can cause older calculated rules sets to overwrite newer ones. By using `IOLoop.add_timeout()`, we enqueue all the recalculations so they occur in order.

There's not a lot of code in this app – most of the actual work is delegated to the handlers `SwitchHandler` and `RouterHandler`. Each handler does two main tasks: (a) review incoming packets and (b) contribute their portion of the network-wide policy based on the NIB. Since the switches and router policies are non-overlapping (they have different `SwitchEq` filters) simply `Union`'ing them together gives you the network-wide policy. In this app, all the handler workflow is hard-coded, but you can imagine a more dynamic main program that would register handlers and dynamically delegate events based on signatures. That's overkill for our routing application.

The NIB, in `routing/network_information_base.py` looks a lot like the NIB we use in multiswitch handling. IP information, though, makes the tables a bit wider, so we move to using objects instead of tuples. These two classes model devices (connected hosts and gateways) and subnets.

```
class ConnectedDevice(object):
  dpid = None
  port_id = None
  ip = None
  mac = None

  def __init__(self, dpid, port_id, ip, mac):
    self.dpid = dpid
    self.port_id = port_id
    self.ip = ip
    self.mac = mac
  def __str__(self):
    return str(self.ip)+"/"+self.mac+ \
      " attached to ( "+str(self.dpid)+" , "+str(self.port_id)+" )"

class Subnet(object):
  dpid = None
  subnet_cidr = None
  router_port = None
  gateway = None

  def __init__(self, subnet_cidr, router_port, router_mac, gateway):
    self.subnet_cidr = subnet_cidr
    self.router_mac = router_mac
    self.router_port = router_port
    self.gateway = gateway
```

The hosts table is now a dictionary of MAC addresses to `ConnectedDevice` instances.

The initialization procedure reads the fixed configuration from the Routing Table and topology files. It follows the same general outline as the Mininet custom configurator.

```
  def __init__(self, logger, topo_file, routing_table_file):
    self.logger = logger

    # Read the Fixed routing table first
    f = open(routing_table_file, "r")
    routing_config = json.load(f)
    f.close()
    for rt in routing_config:
      sn = Subnet(rt["subnet"],rt["router_port"],rt["router_mac"],rt["gateway"])
      self.subnets.append(sn)

    # Read the fixed configuration from the topology file
    topo_agraph = pgv.AGraph(topo_file)
    switchnames = {}

    # Read router name and dpid from topo
    for node in topo_agraph.nodes():
      if node.startswith("s"):
        if node.attr["router"]:
          router_name = str(node)
          self.router_dpid = NetUtils.int_from_mac_colon_hex(node.attr["dpid"])
        else:
          switchnames[str(node)] = NetUtils.int_from_mac_colon_hex(node.attr["dpid"])

    # Mark all the internal ports in the switches, since these need special attention
    for link in topo_agraph.edges():
      (src_node, dst_node) = link
      if str(src_node) == router_name:
        dpid = switchnames[ str(dst_node) ]
        if dpid not in self.internal_ports:
```

```
        self.internal_ports[dpid] = []
      self.internal_ports[dpid].append(int(link.attr['dport']))
```

The learning procedure adds the IP field, which may be passed in as None for non-IP packets. The first packet from a device might very well be non-IP, as in a DHCP request (even though DHCP asks for an IP address, the DHCP packet itself is not an IP packet). That might trigger learning on the L2 switch, which records its port and MAC, but no IP address. The router can later call this procedure to fill in the missing IP.

The NIB now has dirty flag which learning/unlearning a MAC can set or. The main handler uses this to determine whether to do a wholesale recalculation of the switch and router policies.

```
  def is_dirty(self):
    return self.dirty

  def set_dirty(self):
    self.dirty = True

  def clear_dirty(self):
    self.dirty = False
```

The switch handler is virtually identical to the switching application of Chapter **??**. The main difference is extra handling in the MAC learning procedure – if the packet is an IP packet, we extract the extra IP address and learn that as well. One important except is the *internal port*, meaning any ports connected to other switches or routers. Because the routing process changes the Ethernet Source MAC, we can't count on it to match the Source IP address, which itself never changes. So though we learn the MAC and port of an internal port, we ignore the IP address of any packets arriving there.

This code is from routing/switch_handler.py:

```
  def packet_in(self, pkt, payload):
    nib = self.nib

    # If we haven't learned the ports yet, just exit prematurely
    if nib.switch_not_yet_connected():
      return

    # If this packet was received at the router, just ignore
    if pkt.switch == self.nib.router_dpid:
      return

    src_mac = pkt.ethSrc
    dst_mac = pkt.ethDst

    # If we haven't learned the source mac
    if nib.port_for_mac_on_switch( src_mac, pkt.switch ) == None:
```

```
    # If this is an IP or ARP packet, record the IP address too
    src_ip = None

    # Don't learn the IP-mac for packets coming in from the router port
    # since they will be incorrect.  (Learn the MAC address though)
    if nib.is_internal_port(pkt.switch, pkt.port):
      pass
    elif pkt.ethType == 0x800 or pkt.ethType == 0x806:   # IP or ARP
      src_ip = pkt.ip4Src

    nib.learn( src_mac, pkt.switch, pkt.port, src_ip)

  # Look up the destination mac and output it through the
  # learned port, or flood if we haven't seen it yet.
  dst_port = nib.port_for_mac_on_switch( dst_mac, pkt.switch )
  if  dst_port != None:
    actions = SetPort(dst_port)
  else:
    actions = SetPort(nib.all_ports_except(pkt.switch, pkt.port))
  self.main_app.pkt_out(pkt.switch, payload, actions )
```

The interesting processing happens in `routing/router_handler.py`. First, we set up objects to hold queued packets waiting for an ARP reply:

```
def __init__(self, dpid, port_id, payload):
  self.dpid = dpid
  self.port_id = port_id
  self.payload = payload
```

Policies are constructed from the learned MAC table, similarly to the switches. The big differences are we look at the entire network-wide MAC table, and we look only at those entries with IP addresses.

```
def policy_for_learned_ip(self, cd):
  sn = self.nib.subnet_for(cd.ip)
  return Filter( EthTypeEq(0x800) & IP4DstEq(cd.ip) ) >> \
    SetEthSrc(sn.router_mac) >> SetEthDst(cd.mac) >> \
    SetPort(sn.router_port)

def policies_for_learned_ips(self):
  return Union(
    self.policy_for_learned_ip(cd)
    for cd in self.nib.all_learned_macs_with_ip()
  )

def learned_dst_ips(self, all_learned_ips):
  return Or( IP4DstEq(ip) for ip in all_learned_ips )
```

```
def policy_for_router(self):
  all_learned_ips = self.nib.all_learned_ips()
  return IfThenElse(
    EthTypeEq(0x800) & ~ self.learned_dst_ips( all_learned_ips ),
    SendToController("router"),
    self.policies_for_learned_ips()
  )

def policy_for_arp(self):
  return Filter(EthTypeEq(0x806)) >> SendToController("router")

def policy(self):
  return Filter( SwitchEq(self.nib.router_dpid)) >> \
    Union( [ self.policy_for_router(), self.policy_for_arp() ])
```

Here you can see the learned MAC policies, the catch-all subnet policies, and the ARP policy are installed. ARP replies are constructed from scratch using the Frenetic Packet object described in Section 2.5.1:

```
def arp_reply(self, dpid, port, src_mac, src_ip, target_mac, target_ip):
  # Note for the reply we flip the src and target, as per ARP rules
  arp_reply_pkt = Packet(
    ethSrc=target_mac, ethDst=src_mac, ethType = 0x806,
    ip4Src=target_ip, ip4Dst=src_ip, ipProto=2
  )
  payload = arp_reply_pkt.to_payload()
  self.main_app.pkt_out(dpid, payload, SetPort(port))

def arp_request(self, dpid, port, src_mac, src_ip, target_ip):
  arp_request_pkt = Packet(
    ethSrc=src_mac, ethDst="ff:ff:ff:ff:ff:ff", ethType = 0x806,
    ip4Src=src_ip, ip4Dst=target_ip, ipProto=1
  )
  payload = arp_request_pkt.to_payload()
  self.main_app.pkt_out(dpid, payload, SetPort(port))
```

The Packet In handler deals primarily with ARP requests and replies:

```
def packet_in(self, pkt, payload):
  nib = self.nib

  # If we haven't learned the ports yet, just exit prematurely
  if nib.switch_not_yet_connected():
    return
```

```
# If this packet was not received at the router, just ignore
if pkt.switch != self.nib.router_dpid:
  return

src_mac = pkt.ethSrc
dst_mac = pkt.ethDst

if pkt.ethType == 0x806: # ARP
  reply_sent = False
  if pkt.ipProto == 1:
    self.logger.info("Got ARP request from "+pkt.ip4Src+" for "+pkt.ip4Dst)
    for sn in self.nib.subnets:
      if pkt.ip4Dst == sn.gateway:
        self.logger.info("ARP Reply sent")
        self.arp_reply( pkt.switch, pkt.port, src_mac, pkt.ip4Src, sn.router_mac, pkt.ip4Dst)
        reply_sent = True
    if not reply_sent:
      self.logger.info("ARP Request Ignored")

  # For ARP replies, see if the reply was meant for us, and release any queued packets if so
  elif pkt.ip4Src in self.arp_requests:
    src_ip = pkt.ip4Src
    self.logger.info("ARP reply for "+src_ip+" received.  Releasing packets.")
    dev = nib.hosts[src_mac]
    if src_ip != dev.ip:
      nib.learn( src_mac, dev.dpid, dev.port_id, src_ip )
    self.release_waiting_packets(src_ip)

  else:
    self.logger.info("ARP reply from "+pkt.ip4Src+" for "+pkt.ip4Dst+" ignored.")
```

And with IP packets:

```
elif pkt.ethType == 0x800:  # IP
  src_ip = pkt.ip4Src
  dst_ip = pkt.ip4Dst

  # Now send it out the appropriate interface, if we know it.
  dst_mac = nib.mac_for_ip(dst_ip)
  if dst_mac == None:
    # We don't know the mac address, so we send an ARP request and enqueue the packet
    self.enqueue_waiting_packet(dst_ip, pkt.switch, pkt.port, payload)

  # This will be fairly rare, in the case where we know the destination IP but the rule
  # hasn't been installed yet.  In this case, we emulate what the rule should do.
  else:
    sn = nib.subnet_for(dst_ip)
    if sn != None:
      actions = [SetEthSrc(sn.router_mac), SetEthDst(dst_mac), SetPort(sn.router_port)]
      self.main_app.pkt_out(pkt.switch, payload, actions)
    else:
      # Usually we would send the packet to the default gateway, but in this case.
      self.logger.info("Packet for destination "+dst_ip+" on unknown network dropped")
else:
  self.logger.info("Router: Got packet with Ether type "+str(pkt.ethType))
```

The queuing and dequeuing procedures are straightforward. Note that a huge stream of packets to a disconnected IP address could easily overwhelm this implementation, and so caps on the queues should probably be enforced:

```
def enqueue_waiting_packet(self, dst_ip, dpid, port_id, payload):
  self.logger.info("Packet for unknown IP "+dst_ip+" received.  Enqueuing packet.")
  if dst_ip not in self.arp_requests:
    sn = self.nib.subnet_for(dst_ip)
    if sn == None:
      self.logger.info("ARP request for "+dst_ip+" has no connected subnet ")
      return

    self.logger.info("Sending ARP Request for "+dst_ip)
    self.arp_request(self.nib.router_dpid, sn.router_port, sn.router_mac, sn.gateway, dst_ip)
    self.arp_requests[dst_ip] = []

  self.arp_requests[dst_ip].append( WaitingPacket(dpid, port_id, payload) )

def release_waiting_packets(self, dst_ip):
  if dst_ip not in self.arp_requests:
    self.logger.info("ARP reply from "+dst_ip+" released no waiting packets ")
    return

  sn = self.nib.subnet_for(dst_ip)
  if sn == None:
    self.logger.info("ARP reply from "+dst_ip+" has no connected subnet ")
    return

  dst_mac = self.nib.mac_for_ip(dst_ip)
  if dst_mac == None:
    self.logger.info("ARP reply from "+dst_ip+" has no MAC address ")
    return

  for wp in self.arp_requests[dst_ip]:
    actions = [
      SetEthSrc(sn.router_mac),
      SetEthDst(dst_mac),
      Output(Physical(sn.router_port))
    ]
    self.main_app.pkt_out(wp.dpid, wp.payload, actions)

  self.logger.info("All packets for IP "+dst_ip+" released.")
  del self.arp_requests[dst_ip]
```

7.5 Summary

The IP routing process need a summary.

In the next chapters, we build on layer three technology. In chapter **??** we build Network Address Translation which is like routing but changes the IP address and ports as well. From there it's a small step to Firewall and Load Balance technology.

Chapter 8

Routing Variants

In Layer 3 routing, you bascially compute next hop information based on the IP destination address, shifting the ethernet source and destination as you move the packet along. In routing, you always keep the remaining headers values, especially the IP source and destination address, constant.

A few useful variants of Layer 3 routing change other packet headers as well:

- In *load balancing*, a few servers with different IP addresses can act as exact copies of on another. A single artificial IP address, called the *front-end* IP, is where clients send their packets, and a load balancer changes the front-end IP to a real server IP as it moves the packet. Ths load balancer can partition user traffic to maximize efficiency.

- In *network address translation*, or NAT, a group of client machines share a single externally visible, public IP address. But the machines themselves use private, non-routable IP addresses. The NAT box translates these private addresses to the public one on outgoing packets, and the reverse on incoming ones. NAT enables conversations originating on the private network only.

- In a *firewall*, NAT is augmented with security features to allow incoming traffic as well. The traffic must follow certain rules – for example, only port 80 on a certain IP.

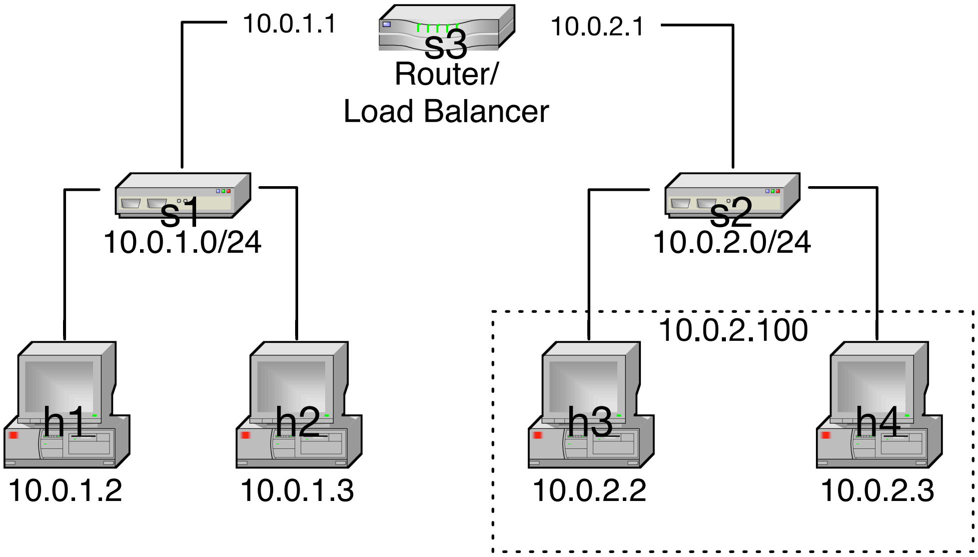
In previous chapters, we always wrote our programs from a blank slate. Because load balancers, NAT boxes and firewalls are so much like routing, we will take a different approach. We will treat the router handler as a superclass, and write our specialized devices as subclasses of it. In addition, we'll use the NIB developed for routers and subclass it to add any additional information needed for the specialized box.

Object oriented concepts in something new that SDN brings to the table. With hardware-based traditional network devices, you could only extend them in ways the vendor allowed. This is like the days of canned, non object-oriented software – changing the behavior of small parts was virtually impossible without the source code. With Frenetic, you can build a library of easily extendible network pieces, and use them for purposes far beyond what the original authors intended.

8.1 Load Balancing

A *load balancer* distributes client requests to certain servers invisibly. When we enter a request to http://www.google.com, which is one IP address, but the request is handled by one of thousands of servers at Google. This is good for the client, who then doesn't have to try certain backend IP addresses until they find a fast server. But it's also good for the server owner's security – by not publicly posting their back end IP's, they can prevent coordinated attacks on certain servers. (Of course, you can attack the load balancer, but this box usually has more intelligence to detect such things).

So let's take a small example. In the following network:



Clients in the 10.0.1.0/24 network issue a request to 10.0.2.100, called the *front-end IP*. No actual host owns this front-end IP, and network administrators must take steps to assure it's not re-used by some real host. The IP's of the two hosts in the 10.0.2.0/24 network: h3 at 10.0.2.2 and h4 at 10.0.2.3 are called *back-end IP's*. In our example, h3 and h4 are exact equivalents, and can service the same set of requests. It doesn't matter which host serves the content.

We use this information to power the load balancer. So if h3 and h4 are equivalent, how does the load balancer decide where to send a request for 10.0.2.100? The load balancer can be as simple or as sophisticated as it wants:

- In *Round Robin* scheduling, each subsequent request goes to the next server in the chain. The first goes to h3, the second to h4, the third back to h3, and so on.

- In *Sticky Round Robin*, clients are assigned to a server on first request. So if h2 asks for 10.0.2.100 first, it will be assigned to h3. Subsequent requests from h2 will continue to go to h3. A request from h1 might then go to h4.

- In *Least Utilized*, each request is assigned the server that has the lowest utilization by some measure (least CPU usage, least network traffic, etc.)

Different network applications benefit from different strategies. For example, web apps that use server-side state tend to use Sticky Round Robin. This assures that subsequent requests from a client can access the same server-side state without complex state-distribution mechansims. Applications that have heavy CPU usage like reporting benefit from a Least Utilized strategy. This prevents someone with a long-running report from slowing down the system unnecessarily for others with short requests.

With SDN, you can write arbitrarily complex load balancing algorithms. For our example, we will use Sticky Round Robin. A Load Balancer must do three things:

- When the first request comes in, it must select a back-end IP to rewrite. It notes this in a table.

- It then rewrites the packet with the back-end IP, does the typical routing thing of altering the MAC addresses, and forwards the packet. It must do this because the back end host must receive a request bound for its own IP - if it receives one for 10.0.2.100, it'll simply drop the packet.

- Lastly, when the reply comes, it must reverse the step. It rewrites the source IP back to the front-end IP 10.0.2.100, then does the typical routing thing. If it does not do this, the host will receive a reply from a back-end IP, which it never sent to, and will probably drop the packet.

So let's get to it! In the first incarnation of or load balancer, as in the past, we'll do load balancing tasks solely in the controller. Once that's debugged, we can move these steps to rules in the flow table. To configure the load balancer, as in the past we'll use a JSON file.

Here's one for our sample network, from `routing_variants/load_balancer.json`:

```
{ "frontend": "10.0.2.100",
  "backends": [ "10.0.2.3", "10.0.2.2" ]
}
```

Our Network Information Base will use the router's NIB as a superclass, and add load balancing attributes. The main addition is a map of back-end IP's to clients, and a pointer to the "current" back-end IP entry.

The following code is in from `routing_variants/load_balancer_nib.py`:

```
import sys
sys.path.append("../routing")
from network_information_base import *
```

```
class LoadBalancerNIB(NetworkInformationBase):

  lb_config = {}

  # Map source IP's to a host so they stay sticky
  # { "10.0.1.2": "10.0.2.2", "10.0.1.3": "10.0.2.3" }
  backend_map = {}

  # index into lb_config["farm"] for the next target host
  current_backend_index= 0

  def __init__(self, logger, topo_file, routing_table_file, load_balancer_file):
    super(LoadBalancerNIB,self).__init__(logger, topo_file, routing_table_file)

    # Read the load_balancer configuration
    f = open(load_balancer_file, "r")
    self.lb_config = json.load(f)
    f.close()

  def backend_ip(self, src_ip):
    if src_ip not in self.backend_map:
      backends = self.lb_config["backends"]
      n_backends = len(backends)
      self.backend_map[src_ip] = backends[self.current_backend_index]
      self.current_backend_index = (self.current_backend_index + 1) % n_backends

    return self.backend_map[src_ip]

  def lb_frontend_ip(self):
    return self.lb_config["frontend"]

  def lb_backend_ips(self):
    return self.lb_config["backends"]
```

Note hwo we define a subclass of Routing's `NetworkInformationBase` class, which Python finds by the addition of `../routing` to the class path. There are more sophistcated ways to structure your network function library, and you do that down the road.

The method `backend_ip` does the heavy lifting of assigning a client IP to a back-end server (if it hasn't already been assigned). If this were a more sophisticated load balancing algorithm, this method would bear the brunt of it. Our version is fairly simple.

Next we define the network application itself. The following code is in from `routing_variants/load_bala`

```
from frenetic.packet import *
import sys,logging,datetime,time
sys.path.append("../routing")
from load_balancer_nib import *
from switch_handler import *
from load_balancer_handler import *
from tornado.ioloop import IOLoop
```

```
class LoadBalancerApp(frenetic.App):

  client_id = "load_balancer"

  def __init__(self,
    topo_file="../routing/topology.dot",
    routing_table_file="../routing/routing_table.json",
    load_balancer_file="load_balancer.json"
    ):
    frenetic.App.__init__(self)
    self.nib = LoadBalancerNIB(
      logging,
      topo_file, routing_table_file, load_balancer_file
    )

    self.switch_handler = SwitchHandler(self.nib, logging, self)
    self.load_balancer_handler = LoadBalancerHandler(self.nib, logging, self)

  def policy(self):
    return self.switch_handler.policy() | self.load_balancer_handler.policy()

  def update_and_clear_dirty(self):
    self.update(self.policy())
    self.nib.clear_dirty()

  def connected(self):
    def handle_current_switches(switches):
      logging.info("Connected to Frenetic - Switches: "+str(switches))
      self.nib.set_all_ports( switches )
      self.update( self.policy() )
      # Wait a few seconds for the policy to be installed
      logging.info("Pausing 2 seconds to allow rules to be installed")
      IOLoop.instance().add_timeout(datetime.timedelta(seconds=2),
        self.load_balancer_handler.connected
      )
    self.current_switches(callback=handle_current_switches)

  def packet_in(self, dpid, port, payload):
    pkt = Packet.from_payload(dpid, port, payload)
    self.switch_handler.packet_in(pkt, payload)
    self.load_balancer_handler.packet_in(pkt, payload)

    if self.nib.is_dirty():
      # This doesn't actually wait two seconds, but it serializes the updates
      # so they occur in the right order
      IOLoop.instance().add_timeout(datetime.timedelta(seconds=2),
        self.update_and_clear_dirty
      )

  def port_down(self, dpid, port_id):
    self.nib.unlearn( self.nib.mac_for_port_on_switch(dpid, port_id) )
    self.nib.delete_port(dpid, port_id)
    self.update_and_clear_dirty()

  def port_up(self, dpid, port_id):
    # Just to be safe, in case we have old MACs mapped to this port
    self.nib.unlearn( self.nib.mac_for_port_on_switch(dpid, port_id) )
    self.nib.add_port(dpid, port_id)
    self.update_and_clear_dirty()

if __name__ == '__main__':
  logging.basicConfig(\
    stream = sys.stderr, \
    format='%(asctime)s [%(levelname)s] %(message)s', level=logging.INFO \
  )
  app = LoadBalancerApp()
  app.start_event_loop()
```

This main application uses similar techniques as the Routing main application. It delegates the Packet In handler, for example, to the packet-in handlers of the Switches and Load Balancer. It uses the switch mechanism from the Routing application verbatim – no changes are necessary.

One interesting addition is to the pythonconnected() handler. Here the load balancer handler will do some extra setup work. When a load balancer routes a first request, it calculates the back-end IP, but it must also know the MAC address of that back-end IP (because the load balancer is also a router). It could just flood the request to all back-end hosts, but that could cause security problems. Instead, the load balancer attempts to calculate these MAC addresses up front. In a more sophsticated load balancer, it could also ask all back-end hosts “are you up?” and remove back-end hosts from the pool that are not. In our simple case, we’ll just send an ARP request to all back-end hosts and hope-for-the-best.

The following code is in from `routing_variants/load_balancer_handler.py`:

```
import sys
sys.path.append(”../routing”)
from router_handler import *

class LoadBalancerHandler(RouterHandler):

  def where_is(self, src_ip):
    sn = self.nib.subnet_for(src_ip)
    self.logger.info("Sending out ARP Request for IP "+src_ip)
    self.arp_request(self.nib.router_dpid, sn.router_port, sn.router_mac, sn.gateway, src_ip)

  def connected(self):
    # We send out ARP requests to all back-end IP's so we minimize waiting
    # for first-time requests
    self.logger.info("Sending out ARP Requests to all Backend IPs")
    for backend_ip in self.nib.lb_backend_ips():
      self.where_is(backend_ip)
```

Like the load balancer NIB, `LoadBalancerHandler` subclasses the equivalent functionality from the Routing handler. That way it can override what it needs to and delegate the rest.

So the load balancer sends out ARP requests to all back-end IP's. It does not bother waiting for the response – when the hosts respond, the MAC and port will be learned as if it were any other host. Note that this code uses the ARP request methods from the router – there's no need to rewrite those.

The load balancer must intercept traffic to the front-end IP, or from the back-end IP's. But the router already handles this traffic natively, so we can't `Union` these rules in. Instead we use an `IfThenElse` to split out the traffic.

```
  def pred_for_load_balancer(self):
    frontend_pred = IP4DstEq(self.nib.lb_frontend_ip())
    backend_pred = Or([IP4SrcEq(backend_ip) for backend_ip in self.nib.lb_backend_ips()])
    return frontend_pred | backend_pred

  def policy(self):
    return Filter( SwitchEq(self.nib.router_dpid)) >> \
      IfThenElse(
        self.pred_for_load_balancer(),
        SendToController("router"),
        self.policy_for_router() | self.policy_for_arp()
      )
```

The `policy` method here overrides the one in `routing_handler`. So load balanced IP traffic is sent to the controller, and the rest is delegated back to the IP or ARP handling policies of the router. Lastly, the `packet_in` handler:

```
  def packet_in(self, pkt, payload):
    nib = self.nib

    # If we haven't learned the ports yet, just exit prematurely
    if nib.switch_not_yet_connected():
      return

    # If this packet was not received at the router, just ignore
    if pkt.switch != self.nib.router_dpid:
      return

    src_mac = pkt.ethSrc
    dst_mac = pkt.ethDst

    lb_processed = False
    if pkt.ethType == 0x800:  # IP
      src_ip = pkt.ip4Src
      dst_ip = pkt.ip4Dst

      # If the packet is bound for a front-end IP, rewrite it to the
      # back end IP
      frontend_ip = self.nib.lb_frontend_ip()
      if dst_ip == frontend_ip:
        sn = self.nib.subnet_for(frontend_ip)
        backend_ip = nib.backend_ip(src_ip)
        backend_mac = nib.mac_for_ip(backend_ip)
        if backend_mac == None:
          self.where_is(backend_ip)
        else:
          self.logger.info("Rerouting request for "+frontend_ip+" to "+backend_ip+"/"+backend_mac)
          self.logger.info("Using subnet on router port "+str(sn.router_port)+" mac "+sn.router_mac)
          actions = [
            SetIP4Dst(backend_ip),
            SetEthSrc(sn.router_mac),
            SetEthDst(backend_mac),
            SetPort(sn.router_port)
          ]
          self.main_app.pkt_out(pkt.switch, payload, actions)

        # We consider the packet processed even if we don't know the mac
        lb_processed = True

      # If the packet is coming from a back-end IP, rewrite it with
      # the front-end IP.
      if src_ip in nib.lb_backend_ips():
        dst_mac = nib.mac_for_ip(dst_ip)
        if dst_mac == None:
          self.where_is(dst_ip)
        else:
          sn = nib.subnet_for(dst_ip)
          self.logger.info("Rewriting reply from "+src_ip+" to "+frontend_ip)
          actions = [
            SetIP4Src(frontend_ip),
            SetEthSrc(sn.router_mac),
            SetEthDst(dst_mac),
            SetPort(sn.router_port)
          ]
          self.main_app.pkt_out(pkt.switch, payload, actions)
        lb_processed = True

    # Punt all non-load balancer packets to the router
    if not lb_processed:
      super(LoadBalancerHandler,self).packet_in(pkt, payload)
```

Does the heavy lifting of rewriting the IP's. Note that we have fail-safe mechanisms in place just in case we don't know the MAC for the back-end IP. In this case, we send out an ARP request and drop the packet (TCP will re-send the request, and hopefully we'll have the MAC and port learned by then). Non-load-balanced traffic is delegated to the Packet In handler of the router.

So how does this work? Let's first start up our custom topology from the routing chapter:

```
vagrant@frenetic:~/manual/programmers_guide/code$ cd routing
vagrant@frenetic:~/manual/programmers_guide/code/routing$ sudo python mn_dot_topology.py
*** Removing excess controllers/ofprotocols/ofdatapaths/pings/noxes
killall controller ofprotocol ofdatapath ping nox_core lt-nox_core ovs-openflowd ...
killall -9 controller ofprotocol ofdatapath ping nox_core lt-nox_core ...
pkill -9 -f ”sudo mnexec”
*** Removing junk from /tmp
rm -f /tmp/vconn* /tmp/vlogs* /tmp/*.out /tmp/*.log
*** Removing old X11 tunnels
*** Removing excess kernel datapaths
ps ax | egrep -o 'dp[0-9]+' | sed 's/dp/nl:/'
***  Removing OVS datapaths
ovs-vsctl --timeout=1 list-br
ovs-vsctl --if-exists del-br s1 -- --if-exists del-br s2 -- --if-exists del-br s_router
ovs-vsctl --timeout=1 list-br
*** Removing all links of the pattern foo-ethX
ip link show | egrep -o '([-_.[:alnum:]]+-eth[[:digit:]]+)'
ip link show
*** Killing stale mininet node processes
pkill -9 -f mininet:
*** Shutting down stale tunnels
pkill -9 -f Tunnel=Ethernet
pkill -9 -f .ssh/mn
rm -f ~/.ssh/mn/*
*** Cleanup complete.
Unable to contact the remote controller at 127.0.0.1:6633
Network ready
Press Ctrl-d or type exit to quit
mininet>
```

And then start up Frenetic, and our load balancer application. First we'll use h2 to ping our front-end IP. This will be assigned our first back-end IP, 10.0.2.3. Then we'll ping it from h1, which is assigned to the other back-end IP, 10.0.2.2.

```
mininet> h2 ping -c 1 10.0.2.100
PING 10.0.2.100 (10.0.2.100) 56(84) bytes of data.
64 bytes from 10.0.2.100: icmp_seq=1 ttl=64 time=85.9 ms

--- 10.0.2.100 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 85.960/85.960/85.960/0.000 ms
mininet> h1 ping -c 1 10.0.2.100
```

```
PING 10.0.2.100 (10.0.2.100) 56(84) bytes of data.
64 bytes from 10.0.2.100: icmp_seq=1 ttl=64 time=328 ms
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 85.960/85.960/85.960/0.000 ms
mininet>
```

Note that our ping requests don't mention any of the back-end IP's. The client doesn't even know they exist. But the log shows evidence that they're being rerouted to different hosts:

```
vagrant@frenetic:~/manual/programmers_guide/code/routing_variants$ python load_balancer1.py
Starting the tornado event loop (does not return).
2016-06-09 11:42:31,831 [INFO] Connected to Frenetic - Switches: {282574488338432: [2, 1],
  282574488338433: [2, 1, 3], 282574488338434: [2, 1, 3]}
2016-06-09 11:42:31,832 [INFO] Pausing 2 seconds to allow rules to be installed
2016-06-09 11:42:33,832 [INFO] Sending out ARP Requests to all Backend IPs
2016-06-09 11:42:33,833 [INFO] Sending out ARP Request for IP 10.0.2.3
2016-06-09 11:42:33,834 [INFO] Sending out ARP Request for IP 10.0.2.2
2016-06-09 11:42:33,843 [INFO] Learning: None/02:02:02:02:02:02 attached to ( 282574488338434 , 3 )
2016-06-09 11:42:33,889 [INFO] Learning: 10.0.2.3/00:00:02:00:00:03 attached to ( 282574488338434 , 2 )
2016-06-09 11:42:33,929 [INFO] Learning: 10.0.2.2/00:00:02:00:00:02 attached to ( 282574488338434 , 1 )
2016-06-09 11:42:33,934 [INFO] ARP reply from 10.0.2.3 for 10.0.2.1 ignored.
2016-06-09 11:42:33,975 [INFO] ARP reply from 10.0.2.2 for 10.0.2.1 ignored.
...
2016-06-09 11:42:55,592 [INFO] Rerouting request for 10.0.2.100 to 10.0.2.3/00:00:02:00:00:03
2016-06-09 11:42:55,592 [INFO] Using subnet on router port 2 mac 02:02:02:02:02:02
2016-06-09 11:42:55,640 [INFO] Rewriting reply from 10.0.2.3 to 10.0.2.100
...
2016-06-09 11:43:20,753 [INFO] Rerouting request for 10.0.2.100 to 10.0.2.2/00:00:02:00:00:02
2016-06-09 11:43:20,754 [INFO] Using subnet on router port 2 mac 02:02:02:02:02:02
2016-06-09 11:43:20,800 [INFO] Rewriting reply from 10.0.2.2 to 10.0.2.100
```

And a quick check of the statistics in both back end hosts bears that out:

```
mininet> h3 ifconfig
h3-eth0   Link encap:Ethernet  HWaddr 00:00:02:00:00:02
          inet addr:10.0.2.2  Bcast:10.0.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:420 (420.0 B)  TX bytes:378 (378.0 B)

mininet> h4 ifconfig
h4-eth0   Link encap:Ethernet  HWaddr 00:00:02:00:00:03
          inet addr:10.0.2.3  Bcast:10.0.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:518 (518.0 B)  TX bytes:476 (476.0 B)
```

OK, so this works fine, but it forces all load balancer traffic through the controller. We can now enshrine these actions in rules. A simple change to our `policy` handler will take care of that.

> *NOTE: This program doesn't work yet.*

The following code is in from `routing_variants/load_balancer_handler2.py`:

```
def policy(self):
  return Filter( SwitchEq(self.nib.router_dpid)) >> \
    IfThenElse(
      self.pred_for_assigned_clients(),
      self.assigned_clients_policy(),
      IfThenElse(
        self.pred_for_unassigned_clients(),
        SendToController("router"),
        self.policy_for_router() | self.policy_for_arp()
      )
    )
```

First we split the load balanced cases in two: one for assigned clients, and one for not-yet-assigned clients. The predicate for assigned clients as in two cases: the request case and the response case (here we assume all requests come from a client and go to a frontend IP, and all replies are the reverse direction):

```
def pred_for_assigned_clients(self):
  frontend_ip = self.nib.lb_frontend_ip()
  preds = []
  for (client_ip, backend_ip) in self.nib.backend_map.iteritems():
    # Forward direction: client -> front-end IP
    preds.append(IP4SrcEq(client_ip) & IP4DstEq(frontend_ip))
    # Backward direction: back-end IP -> client
    preds.append(IP4SrcEq(backend_ip) & IP4DstEq(client_ip))
  return Or(preds)
```

We factor out the request and response policies from the Packet In procedure into their own methods.

```
def request_policy(self, backend_ip):
  sn = self.nib.subnet_for(backend_ip)
  dst_mac = self.nib.mac_for_ip(backend_ip)
```

```
    if dst_mac == None:
      return None
    return Seq([
      SetIP4Dst(backend_ip),
      SetEthSrc(sn.router_mac),
      SetEthDst(dst_mac),
      SetPort(sn.router_port)
    ])

  def response_policy(self, client_ip):
    frontend_ip = self.nib.lb_frontend_ip()
    sn = self.nib.subnet_for(client_ip)
    dst_mac = self.nib.mac_for_ip(client_ip)
    if dst_mac == None:
      return None
    return Seq([
      SetIP4Src(frontend_ip),
      SetEthSrc(sn.router_mac),
      SetEthDst(dst_mac),
      SetPort(sn.router_port)
    ])
```

And that way we can use them both in the new rules:

```
  def assigned_clients_policy(self):
    frontend_ip = self.nib.lb_frontend_ip()
    pols = []
    for (client_ip, backend_ip) in self.nib.backend_map.iteritems():
      # Forward direction: client -> front-end IP
      pols.append(
        Filter(IP4SrcEq(client_ip) & IP4DstEq(frontend_ip)) >> \
        self.request_policy(backend_ip)
      )
      # Backward direction: back-end IP -> client
      pols.append(
        Filter(IP4SrcEq(backend_ip) & IP4DstEq(client_ip)) >> \
        self.response_policy(client_ip)
      )
    return Union(pols)
```

and in the Packet Out command of the Packet In handler:

```
  def packet_in(self, pkt, payload):
    nib = self.nib

    # If we haven't learned the ports yet, just exit prematurely
    if nib.switch_not_yet_connected():
      return
    # If this packet was not received at the router, just ignore
    if pkt.switch != self.nib.router_dpid:
      return

    src_mac = pkt.ethSrc
    dst_mac = pkt.ethDst

    lb_processed = False
    if pkt.ethType == 0x800:  # IP
      src_ip = pkt.ip4Src
      dst_ip = pkt.ip4Dst

      # If the packet is bound for a front-end IP, rewrite it to the
      # back end IP
      frontend_ip = self.nib.lb_frontend_ip()
      if dst_ip == frontend_ip:
        backend_ip = nib.backend_ip(src_ip)
        request_policy = self.request_policy(backend_ip)
        if request_policy == None:
          self.logger.error("Ooops can't find MAC of frontend IP")
          self.where_is(backend_ip)
        else:
          self.main_app.pkt_out(pkt.switch, payload, request_policy)

        # We consider the packet processed even if we don't know the mac
        lb_processed = True

      # If the packet is coming from a back-end IP, rewrite it with
      # the front-end IP.
      if src_ip in nib.lb_backend_ips():
        response_policy = self.response_policy(src_ip)
        if response_policy == None:
          self.where_is(dst_ip)
        else:
          self.main_app.pkt_out(pkt.switch, payload, response_policy)
        lb_processed = True

    # Punt all non-load balancer packets to the router
    if not lb_processed:
      super(LoadBalancerHandler2,self).packet_in(pkt, payload)
```

8.2 Network Address Translation, or NAT

8.3 Firewalls

Chapter 9

Gathering Statistics

9.1 Port Statistics

Frenetic has two statistics-gathering mechanisms:

- The `port_stats` command gets per-port statistics.
- The `SetQuery` policy gathers user-defined statistics. You can use this in switch policies or `pkt_out` actions, and aggregate them.

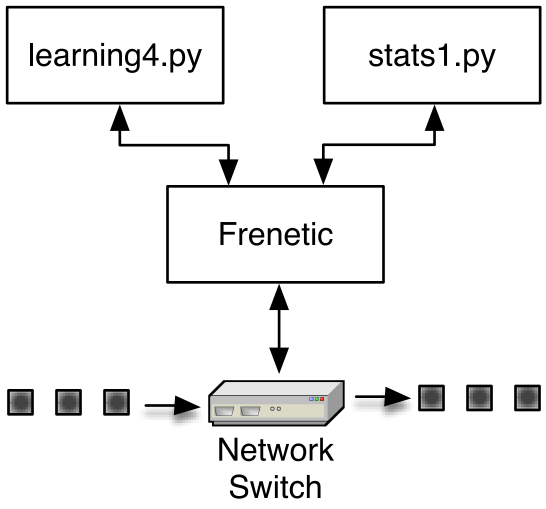
> *SetQuery is not in the current Python bindings, but will be. See Issue #491 on Github*

We could add the statistics-gathering commands to an existing network application, like our learning switch. But Frenetic allows you more flexibility.

We're going to write a completely separate application and run it as a separate process, but point it at the same Frenetic instance as the learning switch. The architecture looks something like this:

This yields some nice advantages:

- We can combine the statistics application with other network applications. It's like a library of network functions.
- The resulting applications are much smaller, focused, and easy to understand and debug.
- If the applications have large NetKAT policies, the process of separating them makes the updating them a bit faster.
- If the applications are CPU intensive, you can run them on different servers or in different VM's.



One caveat is that all NetKAT policies from the applications are `Union`'ed together. Therefore, according to NetKAT Principle 3, they should not have overlapping rules. If this is a problem, you can use concerns to modularize your application instead, which we'll cover in Chapter **??**.

The NIB for our statistics gathering process is fairly simple – we save just the switch DPID and port list. Although we could share the NIB with the other applications, keeping it separate frees us from worrying about locks and so forth

The following code is in `gathering_statistics/network_information_base.py`:

```
class NetworkInformationBase(object):

  # ports on switch
  ports = []

  def __init__(self, logger):
    self.logger = logger

  def set_dpid(self, dpid):
    self.dpid = dpid

  def get_dpid(self):
    return self.dpid

  def switch_not_yet_connected(self):
    return self.ports == []

  def set_ports(self, list_p):
    self.ports = list_p

  def add_port(self, port_id):
    if port_id not in ports:
      self.ports.append(port_id)

  def delete_port(self, port_id):
    if port_id in ports:
      self.ports.remove(port_id)

  def all_ports(self):
    return self.ports
```

The statistics program itself doesn't need to send any NetKAT policies. We set a 5 second timer, and send the `port_stats` command each time, logging the response:

The following code is in `gathering_statistics/stats1.py`:

```
import sys, logging
import frenetic
from frenetic.syntax import *
from network_information_base import *
from tornado.ioloop import PeriodicCallback, IOLoop
from functools import partial

class StatsApp1(frenetic.App):

  client_id = "stats"

  def __init__(self):
    frenetic.App.__init__(self)
    self.nib = NetworkInformationBase(logging)

  def connected(self):
    def handle_current_switches(switches):
      logging.info("Connected to Frenetic - Switches: "+str(switches))
      dpid = switches.keys()[0]
      self.nib.set_dpid(dpid)
      self.nib.set_ports( switches[dpid] )
      PeriodicCallback(self.count_ports, 5000).start()
    self.current_switches(callback=handle_current_switches)

  def print_count(self, future, switch):
    data = future.result()
    logging.info("Count %s@%s: {rx_bytes = %s, tx_bytes = %s}" % \
      (switch, data['port_no'], data['rx_bytes'], data['tx_bytes']) \
    )

  def count_ports(self):
    switch_id = self.nib.get_dpid()
    for port in self.nib.all_ports():
      ftr = self.port_stats(switch_id, str(port))
      f = partial(self.print_count, switch = switch_id)
      IOLoop.instance().add_future(ftr, f)
```

```
if __name__ == '__main__':
  logging.basicConfig(\
    stream = sys.stderr, \
    format='%(asctime)s [%(levelname)s] %(message)s', level=logging.INFO \
  )
  app = StatsApp1()
  app.start_event_loop()
```

Having started Mininet, Frenetic, and the Learning switch `learning4.py` from Section 4.4, we can now start our app independently:

```
vagrant@frenetic:~/manual/programmers_guide/code/gathering_statistics$ python stats1.py
Starting the tornado event loop (does not return).
2016-04-29 10:33:37,321 [INFO] Connected to Frenetic - Switches: {1: [4, 2, 1, 3]}
2016-04-29 10:33:42,333 [INFO] Count 1@3: {rx_bytes = 9590, tx_bytes = 10206}
2016-04-29 10:33:42,333 [INFO] Count 1@4: {rx_bytes = 8414, tx_bytes = 9100}
2016-04-29 10:33:42,334 [INFO] Count 1@2: {rx_bytes = 9562, tx_bytes = 10276}
2016-04-29 10:33:42,334 [INFO] Count 1@1: {rx_bytes = 10024, tx_bytes = 9184}
```

Doing a `pingall` in the Mininet window will make the statistics go up.

What per-port statistics are available? The following table gives you a list:

port_no Port number

rx_packets Number of packets received

tx_packets Number of packets transmitted

rx_bytes Number of bytes received

tx_bytes Number of bytes transmitted

rx_dropped Number of packets attempted to receive, but dropped

tx_dropped Number of packets attempted to transmit, but dropped

rx_errors Number of packets errored upon receive

tx_errors Number of packets errored upon transmit

rx_fram_err Number of packets received with frame errors

rx_over_err Number of packets received with buffer overrun errors

rx_crc_err Number of packets received with CRC errors

collisions Number of collisions detected

These are the per-port statistics defined in OpenFlow 1.0.

9.2 Queries

You can also gather statistics based on NetKAT rules. This is called a *query* and you can think of a query as just another location to send packets, like SetPort or SetPipe. So Principle TODO applies here. Recall that a packet can have only one destination: a port, a pipe or a query. If you want packets to go to more than one destination, you need to use Union between them so packet copies are made.

Unlike per-port statistics, a query only counts two things:

0 Number of packets

1 Number of bytes

You call the self.query() command to get the current counts for this query at any time. The following code is in gathering_statistics/stats2.py:

```
import sys, logging
import frenetic
from frenetic.syntax import *
from network_information_base import *
from tornado.ioloop import PeriodicCallback, IOLoop
from functools import partial

class StatsApp2(frenetic.App):

  client_id = "stats"

  def __init__(self):
    frenetic.App.__init__(self)

  def repeater_policy(self):
    return Filter(PortEq(1)) >> SetPort(2) | Filter(PortEq(2)) >> SetPort(1)

  def http_predicate(self):
    return PortEq(1) & EthTypeEq(0x800) & IPProtoEq(6) & TCPDstPortEq(80)

  def policy(self):
    return IfThenElse(
      self.http_predicate(),
      SendToQuery("http")) | SetPort(2),
      self.repeater_policy()
    )

  def connected(self):
    def handle_current_switches(switches):
      logging.info("Connected to Frenetic - Switches: "+str(switches))
      self.update( self.policy() )
      PeriodicCallback(self.query_http, 5000).start()
    self.current_switches(callback=handle_current_switches)
  def print_count(self, future):
    data = future.result()
    logging.info("Count: {packets = %s, bytes = %s}" % \
      (data[0], data[1]) \
    )

  def query_http(self):
    ftr = self.query("http")
    IOLoop.instance().add_future(ftr, self.print_count)

if __name__ == '__main__':
  logging.basicConfig(\
    stream = sys.stderr, \
    format='%(asctime)s [%(levelname)s] %(message)s', level=logging.INFO \
  )
  app = StatsApp2()
  app.start_event_loop()
```

So here, we set up a query called `"http"` and use it as a location to send packets. The rule counts only HTTP requests initiated from port 1. Every five minutes, we ask for a current count from the query. We can simulate an HTTP in Mininet like this:

```
vagrant@frenetic:~$ sudo mn --topo=single,2 --controller=remote --mac
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> h2 python -m SimpleHTTPServer 80 &
mininet> h1 curl 10.0.0.2
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>

 ... More output
```

And the output changes when the request is initiated:

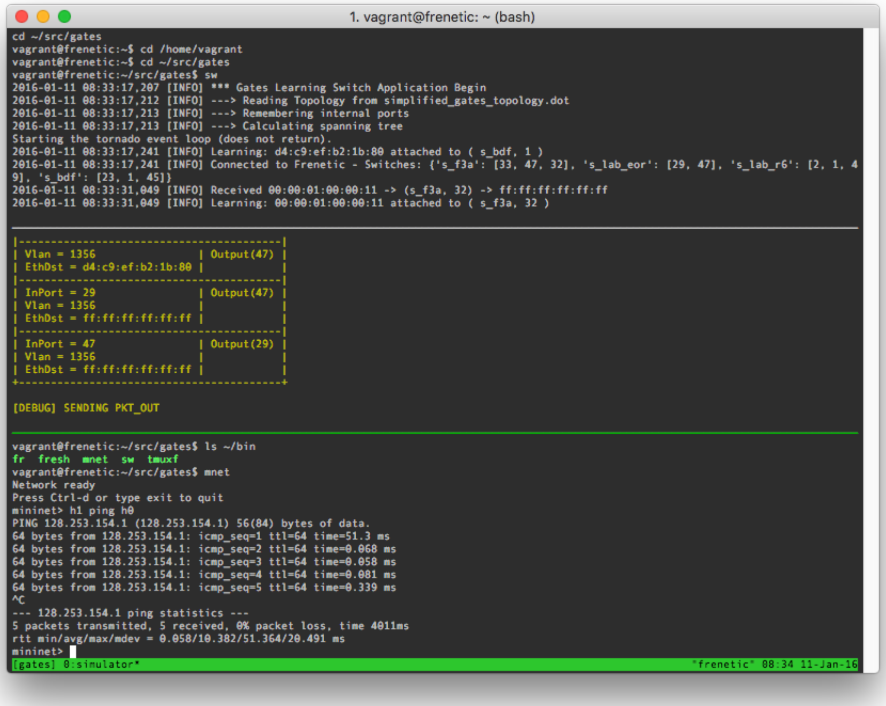
```
Starting the tornado event loop (does not return).
2016-05-11 14:51:46,565 [INFO] Connected to Frenetic - Switches: {1: [2, 4, 1, 3]}
2016-05-11 14:52:06,572 [INFO] Count: {packets = 0, bytes = 0}
2016-05-11 14:52:11,571 [INFO] Count: {packets = 0, bytes = 0}
2016-05-11 14:58:18,802 [INFO] Count: {packets = 11, bytes = 806}
2016-05-11 14:58:23,802 [INFO] Count: {packets = 11, bytes = 806}
```

Chapter 10

SDN Development Tools

10.1 Tmux

Tmux stands for *terminal multiplexor*, and it's indispensible for all kinds of multi-program development. It's a Window Manager for the command line, of sorts. With tmux you can split the screen into *panes*, each of which can run a different shell.



1. vagrant@frenetic: ~ (bash)

```
cd ~/src/gates
vagrant@frenetic:~$ cd /home/vagrant
vagrant@frenetic:~$ cd ~/src/gates
vagrant@frenetic:~/src/gates$ sw
2016-01-11 08:33:17,207 [INFO] *** Gates Learning Switch Application Begin
2016-01-11 08:33:17,212 [INFO] ---> Reading Topology from simplified_gates_topology.dot
2016-01-11 08:33:17,213 [INFO] ---> Remembering internal ports
2016-01-11 08:33:17,213 [INFO] ---> Calculating spanning tree
Starting the tornado event loop (does not return).
2016-01-11 08:33:17,241 [INFO] Learning: d4:c9:ef:b2:1b:80 attached to ( s_bdf, 1 )
2016-01-11 08:33:17,241 [INFO] Connected to Frenetic - Switches: {'s_f3a': [33, 47, 32], 's_lab_eor': [29, 47], 's_lab_r6': [2, 1, 4
9], 's_bdf': [23, 1, 45]}
2016-01-11 08:33:31,049 [INFO] Received 00:00:01:00:00:11 -> (s_f3a, 32) -> ff:ff:ff:ff:ff:ff
2016-01-11 08:33:31,049 [INFO] Learning: 00:00:01:00:00:11 attached to ( s_f3a, 32 )
```

```
|---------------------------------------|
| Vlan = 1356               | Output(47) |
| EthDst = d4:c9:ef:b2:1b:80 |           |
|---------------------------------------|
| InPort = 29               | Output(47) |
| Vlan = 1356               |           |
| EthDst = ff:ff:ff:ff:ff:ff |           |
|---------------------------------------|
| InPort = 47               | Output(29) |
| Vlan = 1356               |           |
| EthDst = ff:ff:ff:ff:ff:ff |           |
+---------------------------------------+

[DEBUG] SENDING PKT_OUT
```

```
vagrant@frenetic:~/src/gates$ ls ~/bin
fr  fresh  mnet  sw  tmuxf
vagrant@frenetic:~/src/gates$ mnet
Network ready
Press Ctrl-d or type exit to quit
mininet> h1 ping h0
PING 128.253.154.1 (128.253.154.1) 56(84) bytes of data.
64 bytes from 128.253.154.1: icmp_seq=1 ttl=64 time=51.3 ms
64 bytes from 128.253.154.1: icmp_seq=2 ttl=64 time=0.068 ms
64 bytes from 128.253.154.1: icmp_seq=3 ttl=64 time=0.058 ms
64 bytes from 128.253.154.1: icmp_seq=4 ttl=64 time=0.081 ms
64 bytes from 128.253.154.1: icmp_seq=5 ttl=64 time=0.339 ms
^C
--- 128.253.154.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4011ms
rtt min/avg/max/mdev = 0.058/10.382/51.364/20.491 ms
mininet> 
[gates] 0:simulator*                                                    "frenetic" 08:34 11-Jan-16
```

In Frenetic development, it's helpful to run tmux with at least 3 panes, which you can see in the example above:

1. One pane runs your network application, e.g. `python repeater.py`

2. One pane runs Frenetic, e.g. `frenetic http-server --verbosity debug`

3. One pane runs Mininet

Tmux is very personalizable and configurable. But if you haven't used tmux before, here is a good setup to get you started.

1. Install tmux on Frenetic-vm with `sudo apt-get install tmux`

2. Create a file in your home directory, ./tmux.conf with the line `set -g prefix C-a`. This maps the tmux prefix key to Ctrl + A , which is much easier to reach than the default Ctrl + B

3. Type `tmux` to start.

Ctrl + A is called the *prefix key*, and we'll denote it as Prefix below. Because you will be typing the prefix a lot, it's really helpful to map your Caps Lock key to Ctrl , if you haven't already done so. On a Mac, for example, you can go to Apple Menu → System Preferences → Keyboard → Keyboard Tab → Modifier Keys and select Control for Caps Lock.

Once there, you can use the following key combinations:

- Prefix = splits the pane at the cursor into two panes: one above the cursor and one below. You can split existing panes as many times as you want, all the way down to panes with one line (which are probably not very useful).

- Prefix ↑ moves the cursor to the pane above. (If you're on the top pane already, the cursor moves to the bottom-most pane.)

- Prefix ↓ moves the cursor to the pane below. (If you're on the bottom pane already, the cursor moves to the top-most pane.)

- Prefix Z zooms the current pane, so that it takes up the entire window. The other panes continue to run, even though they're not visible. Pressing Prefix Z again unzooms the window.

- Prefix D detaches from the Tmux session. You can start it up again later, even after having logged off the Frenetic VM, by using `tmux attach`.

Because tmux operates outside the normal window manager realms, you can no longer scroll up or down in a pane using scroll bars. But tmux has a scrolling mechanism inside itself which scrolls panes independently.

- Prefix , [enters scroll mode. You can see a cursor position status at the top right hand corner of the pane: 67/900 means you're on line 67 of 900 lines in the pane.

- once in scroll mode:
  - ↑ moves the cursor one line up.
  - ↓ moves the cursor one line down.
  - Page ↑ moves the cursor one page up.
  - Page ↓ moves the cursor one page down.
  - Esc leaves scroll mode and scrolls all the way down to the bottom

If you end up doing the same keystrokes each time you start up an SDN session, you can automate it with tmuxinator software.

The book Hogan [2012] is a great introduction and reference to tmux.

10.2 Open VSwitch Utilities

10.3 TCPDump

10.4 Mininet Network Modelling

Chapter 11

Frenetic REST API

The Frenetic Python language bindings are only one way to write network applications. The Frenetic HTTP Controller accepts NetKAT policies, configuration settings through JSON, and pushes network events like Packet In's through JSON. So any programming language that can speak HTTP and JSON (pretty much all of them) can talk to Frenetic.

11.1 REST Verbs

The following verbs are implemented in the HTTP Controller:

POST /*client_id*/update_json	Update the network-wide NetKAT policy, compile the tables, and send the resulting tables to the switches.
GET /*client_id*/event	Get the latest waiting network events
GET /current_switches	Return a list of connected switches and ports
POST /pkt_out	Send a packet out via OpenFlows Packet Out message
GET /version	Return Frenetic version
GET/config	Get current Frenetic configuration
POST /config	Update current Frenetic configuration
GET /query/*name*	Get current statistics bucket contents
GET /port_stats/*sw*/*port*	Get port statistics contents for switch *sw* and port *port*
POST /*client_id*/update	Same as update_json but the policy is in Raw NetKAT format

Of these verbs, you'll be calling update_json and event the most, so we'll discuss these in detail. The rest are outlined in Section 12.4.

11.2 Pushing Policies

You push policies to Frenetic by using the `POST /`*client_id*`/update_json` verb. The *client_id* is identical to the *client_id* instance variable set in our Python language examples. To make the policy updates smaller, you can split the policy into multiple client id's, then call `update\_json` on just that portion to update the policy. Frenetic recompiles all the policies together with an implicit `Union` – so there can be no overlapping rules in each of the clients.

The data you push is a JSON representation of the NetKAT policy. For example, this policy using the Python bindings:

```
Filter(PortEq(1)) >> SetPort(2) |
Filter(PortEq(2)) >> SetPort(1)
```

Will look like this in JSON:

```
{  "type": "union",
   "pols": [
     { "type": "seq",
       "pols": [
         { "type": "filter",
           "pred": {
              "type": "test",
              "header": "port",
              "value": { "type": "physical", "port": 1 }
           }
         },
         { "type": "mod",
           "header": "port",
           "value": { "type": "physical", "port": 2 }
         }
       ]
     },
     { "type": "seq",
       "pols": [
         { "type": "filter",
           "pred": {
              "type": "test",
              "header": "port",
              "value": { "type": "physical", "port": 2 }
           }
         },
         { "type": "mod",
           "header": "port",
           "value": { "type": "physical", "port": 1 }
         }
       ]
     }
   ]
}
```

Figuring out the JSON by hand is pretty laborious. So you can make the Python REPL and Frenetic bindings do the work for you like this:

```
vagrant@frenetic:~$ python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import frenetic
>>> from frenetic.syntax import *
>>> pol = Filter(PortEq(1)) >> SetPort(2) | Filter(PortEq(2)) >> SetPort(1)
>>> pol.to_json()
{'pols': [{'pols': [{'pred': {'header': 'location', 'type': 'test', 'value':
{'type': 'physical', 'port': 1}}, 'type': 'filter'}, {'header': 'location',
'type': 'mod', 'value': {'type': 'physical', 'port': 2}}], 'type': 'seq'},
{'pols': [{'pred': {'header': 'location', 'type': 'test', 'value': {'type':
'physical', 'port': 2}}, 'type': 'filter'}, {'header': 'location', 'type':
'mod', 'value': {'type': 'physical', 'port': 1}}], 'type': 'seq'}], 'type':
'union'}
>>>
```

The complete catalog of JSON syntax for NetKAT predicates and policies is in Chapter 12

11.3 Incoming Events

Calling the `GET /`*client_id*`/event` verb gets a list of incoming network events. The call will block until events become available, which is why the Python bindings are implemented asynchrononously through callbacks. The events come back as a JSON array, so there could be more than one.

All incoming events are listed in Section 12.3. The most prevalent one is `packet_in`, so your call to `GET /`*client_id*`/event` might return something like this:

```
[
  { "type": "packet in",
    "switch id": 1981745,
    "port id": 1,
    "payload": {
```

```
   "id": 19283745,
   "buffer": "AB52DF57B12BF87216345"
  }
 }
 { "type": "packet in",
  "switch id": 923462,
  "port id": 2,
  "payload": {
   "buffer": "AB52DF57B12BF87216345"
  }
 },
]
```

The payload may be either buffered or unbuffered – the absence of an `id` attribute indicates the packet is unbuffered, in which case the `buffer` attribute is the entire contents of the packet. (See Section 2.5.3 for an explanation of how buffering works.)

The contents are encoded in Base 64, and most languages have decoders for this type. You will need to write or acquire a packet parser to extract and test data from inside the packet – RYU's packet library is good on the Python side.

Chapter 12

Frenetic/NetKAT Reference

12.1 NetKAT Predicates

12.1.1 Primitives

Drop

PYTHON	`Drop` `false`
RAW	`false`
REST	`{ ”type”: ”false” }`

`Drop` matches no packets. This is a *predicate*, as opposed to the lower-case equivalent `drop` which is a *policy* that drops all packets.

Note in Python, `Id` and `Drop` are the only predicates that don’t require parantheses.

EthDst

PYTHON	`EthDstEq(”72:00:08:bc:5f:a0”)`
	`EthDstEq(”72:00:08:bc:5f:a0”, ”82:9b:41:a6:16:f8”)`
	`EthDstEq([”72:00:08:bc:5f:a0”, ”82:9b:41:a6:16:f8”])`
		`EthDstNotEq(”72:00:08:bc:5f:a0”)`
		`EthDstNotEq(”72:00:08:bc:5f:a0”, ”82:9b:41:a6:16:f8”)`
		`EthDstNotEq([”72:00:08:bc:5f:a0”, ”82:9b:41:a6:16:f8”])`

RAW	`ethDst = 125344472129440`	
REST	`{ ”type”: ”test”, ”header”: ”ethdst”, ”value”: 125344472129440 }`	

`EthDstEq` matches packets with a particular Ethernet MAC destination address, or from a set of addresses – if the parameter is a list of addresses, there is an implicit OR between them.

In Python, a MAC address must be specified in colon-separated 6 byte hexadecimal. This is the most common format for MAC address display on network devices (although Cisco tends to list them in dotted notation with 2 byte boundaries). They must be passed as strings. In Raw or REST-based NetKAT, you must send the 48-bit MAC as an integer.

EthSrc

PYTHON	`EthSrcEq(”72:00:08:bc:5f:a0”)`
	`EthSrcEq(”72:00:08:bc:5f:a0”, ”82:9b:41:a6:16:f8”)`
	`EthSrcEq([”72:00:08:bc:5f:a0”, ”82:9b:41:a6:16:f8”])`
		`EthSrcNotEq(”72:00:08:bc:5f:a0”)`
		`EthSrcNotEq(”72:00:08:bc:5f:a0”, ”82:9b:41:a6:16:f8”)`
		`EthSrcNotEq([”72:00:08:bc:5f:a0”, ”82:9b:41:a6:16:f8”])`

RAW	`ethSrc = 125344472129440`	
REST	`{ "type": "test", "header": "ethsrc", "value": 125344472129440 }`	

`EthSrcEq` matches packets with a particular Ethernet MAC source address, or from a set of addresses – if the parameter is a list of addresses, there is an implicit OR between them.

In Python, a MAC address must be specified in colon-separated 6 byte hexadecimal. This is the most common format for MAC address display on network devices (although Cisco tends to list them in dotted notation with 2 byte boundaries). They must be passed as strings. In Raw or REST-based NetKAT, you must send the 48-bit MAC as an integer.

EthType

PYTHON	`EthTypeEq(0x800)`
	`EthTypeEq(0x800, 0x806)`
	`EthTypeEq([0x800, 0x806])`
		`EthTypeNotEq(0x800)`
		`EthTypeNotEq(0x800, 0x806)`
		`EthTypeNotEq([0x800, 0x806])`

RAW	`ethTyp = 2048`	
REST	`{ "type": "test", "header": "ethtype", "value": 2048 }`	

`EthType` matches packets with a particular Ethernet frame type, or from a set of frame types – if the parameter is a list of types, there is an implicit OR between them. The frame type is a 32 bit integer as defined by IEEE, and the common ones are listed at http://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml

Python can accept any valid 32 bit integer representation for the frame type – it's common to pass them as hexadecimal values because they're easy to remember. You can also use constants defined in `ryu.packet.ether_types`. Raw and REST-based NetKAT requires them to be specified in decimal.

Certain OpenFlow match rules have *dependencies* on the Ethernet type. In other words, it requires certain Ethernet type matches to be specified when other fields are matched. For example, if the IPv4 source address is matched, then an Ethernet Type match must also be specified as IP, e.g. ethernet type 0x800. This is not required in NetKAT - the Frenetic compiler will automatically compile the correct dependencies for you.

With VLAN tagged packets, OpenFlow matches the type of the *enclosed packet*. So for example, an IP packet wrapped with a VLAN tag will match `EthType(0x800)` even though the actual Ethernet type of the entire packet is 0x8100 (the IEEE 802.1q Ethernet type for VLAN packets).

Id

```
PYTHON  Id
        true

RAW     true

REST     { "type": "true” }
```

`Id` matches all packets. This is a *predicate*, as opposed to the lower-case equivalent `id` which is a *policy* that accepts all packets.

Note in Python, `Id` and `Drop` are the only predicates that don't require parantheses.

IP4Dst

```
PYTHON  IP4DstEq(”192.168.57.100”)    IP4DstNotEq(”192.168.57.100”)
        IP4DstEq(”192.168.57.0”, 24)  IP4DstNotEq(”192.168.57.0”, 24)

RAW     ip4Dst = 192.168.57.100       ip4Dst = 192.168.57.0/24

REST     { "type": "test", "header": "ip4dst",
         "value": {"addr": "192.168.57.100", "mask": 32 } }
         { "type": "test", "header": "ip4dst",
         "value": {"addr": "192.168.57.0", "mask": 24 } }
```

`IP4Dst` matches packets with a particular IP destination address, or destination network. It only matches IP v4 packets, as per Openflow 1.0.

`IP4DstEq` cannot accept a list of values as other predicates can. Instead, you can specify a subnet – a range of IP addresses using a bit mask value. This is identical to using CIDR format, so that `IP4DstEq("192.168.57.0", 24)` matches IP addresses in subnet `192.168.57.0/24` – e.g. IP addresses `192.168.57.1` to `192.168.57.255`. Leaving out the subnet mask in Python or Raw NetKAT is equivalent to specifiying a mask of 32 - meaning all 32 bits of the IP address are used in the match. (The 32 must be explicitly specified in REST-based NetKAT).

IPProto

PYTHON	`IPProtoEq(6)`	`IPProtoNotEq(6)`
	`IPProtoEq(6,17)`	`IPProtoNotEq(6,17)`
	`IPProtoEq([6,17])`	`IPProtoNotEq([6,17])`
RAW	`ipProto = 6`	
REST	`{ "type": "test", "header": "ipproto", "value": 6 }`	

`IPProtoEq` matches packets with a certain IP Protocol, or set of protocols – if a list is specified, there is an implicit OR between them. It only matches IP v4 packets, as per OpenFlow 1.0 specs.

The IP Protocol is a number from 1-255. A complete list of common protocols are listed in https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers. In Python, you can also match against the constants in `ryu.packet.in_proto`.

IP4Src

PYTHON	`IP4SrcEq("192.168.57.100")`	`IP4SrcNotEq("192.168.57.100")`
	`IP4SrcEq("192.168.57.0", 24)`	`IP4SrcNotEq("192.168.57.0", 24)`
RAW	`ip4Src = 192.168.57.100`	`ip4Src = 192.168.57.0/24`

REST	`{ "type": "test", "header": "ip4src",`
	`"value": {"addr": "192.168.57.100", "mask": 32 } }`
	`{ "type": "test", "header": "ip4src",`
	`"value": {"addr": "192.168.57.0", "mask": 24 } }`	

`IP4Src` matches packets with a particular IP source address, or source network. It only matches IP v4 packets, as per Openflow 1.0.

`IP4SrcEq` cannot accept a list of values as other predicates can. Instead, you can specify a subnet – a range of IP addresses using a bit mask value. This is identical to using CIDR format, so that `IP4SrcEq("192.168.57.0", 24)` matches IP addresses in subnet `192.168.57.0/24` – e.g. IP addresses `192.168.57.1` to `192.168.57.255`. Leaving out the subnet mask in Python or Raw NetKAT is equivalent to specifiying a mask of 32 - meaning all 32 bits of the IP address are used in the match. (The 32 must be explicitly specified in REST-based NetKAT).

Port

```
PYTHON  PortEq(1)        PortNotEq(1)
        PortEq(1,2,3)    PortNotEq(1,2,3)
        PortEq([1,2,3])  PortNotEq([1,2,3])

RAW     port = 1

REST     { "type": "test", "header": "location",
         "value": { "type": "physical", "port": 1 } }
```

`PortEq` matches packets that arrive on a particular port, or on a particular set of ports – if the parameter is a list of ports, there is an implicit OR between them.

A port number must be an integer from 1 …65520, the upper limit of physical port numbers according to the OpenFlow 1.0 spec. It can also be a string convertible to an integer in this range. The port number is not checked against any known list of port numbers on the switch, so you can insert rules for non-operational ports (they obviously won't match any packets until the port becomes operational).

`PortEq` predicates are usually combined with `SwitchEq` predicates because packet processing is switch-and-port-specific. However, it may be used alone if, for example, a packet on port 47 is processed exactly the same on every switch (for example, port 47 is the trunk port between switches).

Switch

```
PYTHON  SwitchEq(1981745)            SwitchNotEq(1981745)
        SwitchEq(1981745, 887345)    SwitchNotEq(1981745, 887345)
        SwitchEq([1981745, 887345])  SwitchNotEq([1981745, 887345])

RAW     switch = 1981745

REST     { "type": "test", "header": "switch", "value": 1981745 }
```

`SwitchEq` matches packets that arrive on a particular switch or set of switches. Switches are identified by a 64-bit number called a *datapath ID* or *DPID*. On OpenVSwitch, the default OpenFlow switch provided by Mininet, DPID's are generally small integers corresponding to the switch name – e.g. `s1` in Mininet has the DPID 1. Physical switches generally append the OpenFlow instance number with the overall MAC id of the switch so it's globally unique. But this varies from switch to switch.

`SwitchEq` predicates are usually combined with `PortEq` predicates because packet processing is switch-and-port-specific.

TCPDstPort

```
PYTHON  TCPDstPortEq(80)         TCPDstPortNotEq(80)
        TCPDstPortEq(80,443)     TCPDstPortNotEq(80,443)
        TCPDstPortEq([80,443])   TCPDstPortNotEq([80,443])

RAW     tcpDstPort = 80

REST     { "type": "test", "header": "tcpdstport", "value": 80 }
```

`TCPDstPortEq` matches packets with a certain TCP or UDP destination port, or set of ports – if the parameter is a list of ports, there is an implicit OR between them. Note this only matches TCP or UDP packets, which must be IP packets. A complete list of common destination ports are listed in https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

TCPSrcPort

```
PYTHON  TCPSrcPortEq(5000)           TCPSrcPortNotEq(5000)
        TCPSrcPortEq(5000,5001)      TCPSrcPortNotEq(5000,5001)
        TCPSrcPortEq([5000,5001])    TCPSrcPortNotEq([5000,5001])

RAW     tcpSrcPort = 5000

REST     { "type": "test", "header": "tcpsrcport", "value": 5000 }
```

`TCPSrcPortEq` matches packets with a certain TCP or UDP source port, or set of ports – if the parameter is a list of ports, there is an implicit OR between them. Note this only matches TCP or UDP packets, which must be IP packets.

`TCPSrcPort` matches are rarely used in OpenFlow since TCP and UDP source ports are essentially random numbers assigned by the client. They can be used for "per-flow" rules where a rule lasts for the duration of a conversation between a client and a server.

Vlan

```
PYTHON  VlanEq(1001)          VlanNotEq(1001)
        VlanEq(1001,1002)     VlanNotEq(1001,1002)
        VlanEq([1001,1002])   VlanNotEq([1001,1002])

RAW     vlanId = 1001

REST    { "type": "test", "header": "vlan", "value": 1001 }
```

`VlanEq` matches packets that arrive with a particular VLAN tag, or on a particular set of VLAN tags – if the parameter is a list of vlans, there is an implicit OR between them.

The VLAN id, as defined by IEE 802.1q is a 12 bit value from 1 . . . 4095. VLAN matches are only applicable to packets with an actual VLAN tag, so the Ethernet Type is 0x8100. However, in such cases, OpenFlow exposes the Ethernet Type of the *enclosed packet*. So for example, an IP packet wrapped with a VLAN tag will match `EthType(0x800)`.

In most physical switches, access ports are tagged with a particular VLAN. So though the packet coming from the host is untagged, the tagging occurs at the switch before the OpenFlow engine is invoked. That way you can assign access ports to VLANs, and any packets coming in on that port will match the appropriate `VlanEq`. This is one of the few cases where packet headers are manipulated outside the OpenFlow engine.

VlanPcp

```
PYTHON  VlanPcpEq(1)       VlanPcpNotEq(1)
        VlanPcpEq(1,2)     VlanPcpNotEq(1,2)
        VlanPcpEq([1,2])   VlanPcpNotEq([1,2])

RAW     vlanPcp = 1

REST    { "type": "test", "header": "vlanpcp", value: 1 }
```

`VlanPcpEq` matches packets that arrive with a particular VLAN Priority Code Point (PCP), or on a particular set of VLAN PCP's – if the parameter is a list of PCP's, there is an implicit OR between them. Only VLAN tagged packets will match this predicate.

The PCP as defined by IEEE 802.1q must be an integer from 0-7. 0 is the default, generally meaning best effort delivery.

12.1.2 Combinations

And

PYTHON	$pred_1$ `&` $pred_2$ `And([`$pred_1$`,` $pred_2$`, . . . ,` $pred_n$`])`
RAW	$pred_1$ `and` $pred_2$
REST	{ `"type": "and", "preds": [` $pred_1$`,` $pred_2$`, . . .` $pred_n$ `]` }

`And` is the Boolean conjunction of predicates.

Not

PYTHON	`~`$pred_1$ `Not(`$pred_1$`)`
RAW	`not` $pred_1$
REST	{ `"type": "neg", "pred":` $pred_1$ }

`Not` is the Boolean negation of a single predicate.

Or

PYTHON	$pred_1 \mid pred_2$ `Or([`$pred_1$`,` $pred_2$`, . . . ,` $pred_n$`])`
RAW	$pred_1$ `or` $pred_2$
REST	{ `"type": "or", "preds": [` $pred_1$`,` $pred_2$`, . . .` $pred_n$ `]` }

`Or` is the Boolean disjunction of predicates. In Python, if all of your disjunction terms involve the same field, you can use the list form of the simple predicate as a shortcut (except for `IpSrcEq` and `IpDstEq`).

Note that in Python `Or` shares a symbol with `Union`, but the parser keeps the meaning straight.

12.2 Policies

12.2.1 Primitives

drop

PYTHON	`drop`
RAW	`drop`
REST	`{ "type": "filter", "pred": { "type": false } }`

`drop` is a *policy* that drops all packets. It is equivalent to `Filter(false)`.

Note that in Python, `id` and `drop` are the only policies that begin with a lower case letter, and do not require parantheses.

Filter

PYTHON	`Filter(`*pred*`)`
RAW	`filter` *pred*
REST	`{ "type": "filter", "pred":` *pred* `}`

`Filter` accepts all packets that match the predicate *pred* and rejects everything else. It is described in detail in Section 3.2.

id

PYTHON	`id`
RAW	`id`
REST	`{ "type": "filter", "pred": { "type": true } }`

`drop` is a *policy* that drops all packets. It is equivalent to `Filter(true)`.

Note that in Python, `id` and `drop` are the only policies that begin with a lower case letter, and do not require parantheses.

SendToController

PYTHON	`SendToController("http")`
RAW	`port := pipe("http")`

REST	`{ "type": "mod", "header": "location",`
	`"value": { "type": "pipe", "name": "http" } }`

`SendToController` sends the packet to the controller.

The pipe name is an arbitrary string. You can use it to distinguish . . . TODO

A packet may have only one destination: a port, a query bucket, or the controller. So if you use `SetPort`, `SendToController`, or `SendToQuery` in a sequence, the last one will win. If you want a packet to go to more than one destination, use `Union` between them.

> *Note: I'm not sure how you distinguish packets using pipes since they're not part of the packet and you can't do matches on them.*

SendToQuery

PYTHON	`SendToQuery("internet")`
RAW	`port := query("internet")`

REST	`{ "type": "mod", "header": "location",`
	`"value": { "type": "query", "name": "internet" } }`

`SendToQuery` counts packets and bytes that meet certain match criteria. This is described in Section 9.2.

A packet may have only one destination: a port, a query bucket, or the controller. So if you use `SetPort`, `SendToController`, or `SendToQuery` in a sequence, the last one will win. If you want a packet to go to more than one destination, use `Union` between them.

SetEthDst

PYTHON	`SetEthDst("72:00:08:bc:5f:a0")`
RAW	`ethDst := 125344472129440`
REST	`{ "type": "mod", "header": "ethdst", "value": 125344472129440 }`

`SetEthDst` sets the Ethernet MAC destination address for a packet.

In Python, a MAC address must be specified in colon-separated 6 byte hexadecimal. This is the most common format for MAC address display on network devices (although Cisco tends to list them in dotted notation with 2 byte boundaries). They must be passed as strings. In Raw or REST-based NetKAT, you must send the 48-bit MAC as an integer.

SetEthSrc

PYTHON	`SetEthSrc(”72:00:08:bc:5f:a0”)`
RAW	`ethSrc := 125344472129440`
REST	{ `"type": "mod", "header": "ethsrc", "value": 125344472129440` }

`SetEthDst` sets the Ethernet MAC source address for a packet.

In Python, a MAC address must be specified in colon-separated 6 byte hexadecimal. This is the most common format for MAC address display on network devices (although Cisco tends to list them in dotted notation with 2 byte boundaries). They must be passed as strings. In Raw or REST-based NetKAT, you must send the 48-bit MAC as an integer.

SetEthType

PYTHON	`SetEthType(0x800)`
RAW	`ethTyp := 2048`
REST	{ `"type": "mod", "header": "ethtype", "value": 2048` }

`SetEthType` sets the Ethernet type for a packet.

Note that for VLAN-tagged packets, you set the Ethernet type for the *inner* packet, not 0x8100 for VLAN. To tag it with a VLAN, you simply use the policy `SetVlan`, and that assigns the correct Ethernet type to the outer packet.

SetIP4Dst

PYTHON	`SetIP4Dst(”192.168.57.100”)`
RAW	`ip4Dst := 192.168.57.100`

REST	`{ "type": "mod", "header": "ip4dst",`
	`"value": {"addr": "192.168.57.100" } }`

`SetIP4Dst` sets the IP v4 destination address. Note that unlike the `IP4DstEq` predicate, there is no option for masking. You must set a fixed IP address with all bits present.

The proper Ethernet type for IP packets must already be set in the incoming packet, or through the `SetEthType` policy. Otherwise, the packet will likely be dropped by a host or router before it reaches its destination.

SetIPProto

Python	`SetIPProto(6)`
Raw	`ipProto := 6`
REST	`{ "type": "mod", "header": "ipProto", "value": 6 }`

`SetIPProto` sets the IP v4 protocol type.

The proper Ethernet type for IP packets must already be set in the incoming packet, or through the `SetEthType` policy. Otherwise, the packet will likely be dropped by a host or router before it reaches its destination.

SetIP4Src

Python	`SetIP4Src("192.168.57.100")`
Raw	`ip4Src := 192.168.57.100`

REST	`{ "type": "mod", "header": "ip4src",`
	`"value": {"addr": "192.168.57.100" } }`

`SetIP4Src` sets the IP v4 destination address. Note that unlike the `IP4SrcEq` predicate, there is no option for masking. You must set a fixed IP address with all bits present.

The proper Ethernet type for IP packets must already be set in the incoming packet, or through the `SetEthType` policy. Otherwise, the packet will likely be dropped by a host or router before it reaches its destination.

SetPort

PYTHON

```
SetPort(1)

SetPort(1,2,3)

SetPort([1,2,3])
```

RAW

```
port := 1
```

REST

```
{ "type": "mod", "header": "location",
"value": { "type": "physical", "port": 1 } }
```

`SetPort` sets the port destination for this packet.

A port number must be an integer from 1 ...65520, the upper limit of physical port numbers according to the OpenFlow 1.0 spec. It can also be a string convertible to an integer in this range. The port number is not checked against any known list of port numbers on the switch, but any attempt to send a packet over a non-existent or non-operational port will fail silently.

`SetPort` is the only modification policy that can take a list of arguments. When more than one port is specified, there is an implicit `Union` between them – in other words, packets are copied to each one of the ports.

A packet may have only one destination: a port, a query bucket, or the controller. So if you use `SetPort`, `SendToController`, or `SendToQuery` in a sequence, the last one will win. If you want a packet to go to more than one destination, use `Union` between them.

SetTCPDstPort

PYTHON

```
SetTCPDstPort(80)
```

RAW

```
tcpDstPort := 80
```

REST

```
{ "type": "mod", "header": "tcpdstport", "value": 80 }
```

`SetTCPDstPort` sets the TCP or UDP destination port.

The proper Ethernet type for IP packets must already be set in the incoming packet, or through the `SetEthType` policy. Also, the proper IP Protocol (TCP or UDP) must already be set in incoming packet, or through the `SetIPProto` policy. Otherwise, the packet will likely be dropped by a host or router before it reaches its destination.

SetTCPSrcPort

PYTHON	`SetTCPSrcPort(5000)`
RAW	`tcpSrcPort := 5000`
REST	`{ "type": "mod", "header": "tcpsrcport", "value": 5000 }`

`SetTCPDstPort` sets the TCP or UDP source port.

The proper Ethernet type for IP packets must already be set in the incoming packet, or through the `SetEthType` policy. Also, the proper IP Protocol (TCP or UDP) must already be set in incoming packet, or through the `SetIPProto` policy. Otherwise, the packet will likely be dropped by a host or router before it reaches its destination.

SetVlan

PYTHON	`SetVlan(1001)`
RAW	`vlanId := 1001`
REST	`{ "type": "mod", "header": "vlan", "value": 1001 }`

`SetVlan` sets the VLAN tag for the packet. If there is no VLAN tag, it automatically sets the `EthType` to 0x8100, and pushes the current `EthType` into the inner packet. If there is already a VLAN tag, it merely overwrites the VLAN id.

SetVlanPcp

PYTHON	`SetVlanPcp(1)`
RAW	`vlanPcp := 1`
REST	`{ "type": "mod", "header": "vlanpcp", "value": 1 }`

`SetVlanPcp` sets the VLAN priority for the packet. If there is no VLAN tag, it automatically sets the `EthType` to 0x8100, pushes the current `EthType` into the inner packet and sets the VLAN id to 0. If there is already a VLAN tag, it merely overwrites the existing VLAN PCP.

12.2.2 Combinations

IfThenElse

PYTHON	`IfThenElse(`$pred$`,` $truepol$ $falsepol$`)`
RAW	`if` $pred$ `then` $truepol$ `else` $falsepol$

`IfThenElse` tests a predicate and executes $truepol$ if the predicate is true or $falsepol$ if it is not. This is explained further in Section 3.2.

Seq

PYTHON	pol_1 `>>` pol_2 `Seq([`pol_1`,` pol_2`, . . . ,` pol_n`])`
RAW	pol_1 `;` pol_2
REST	`{ "type": "seq", "pols": [` pol_1`,` pol_2`, . . .` pol_n `] }`

`Seq` is the sequential composition of policies. The packet is pushed through each of the policies one after the other in the order listed. This is explained further in Section 3.2.

Union

PYTHON	pol_1 `|` pol_2 `Union([`pol_1`,` pol_2`, . . . ,` pol_n`])`
RAW	pol_1 `|` pol_2
REST	`{ "type": "union", "pols": [` pol_1`,` pol_2`, . . .` pol_n `] }`

`Union` is the parallel composition of policies. n copies of the packet are sent through each of the listed policies in parallel. This is explained further in Section 3.2.

12.3 Events

In Python, each Frenetic event has a hook listed below. If the application defines a handler with the same signature, that handler is called on the event. If the user doesn't define a handler, a default handler is invoked which simply logs the event.

In REST, calling the URL `/events/$client\_id$` will return the JSON data below if the event has been fired.

12.3.1 connected

PYTHON `connected()`

There is no `connected` event in REST. To simulate it, you simply call GET on the URL `/version`. If a response comes back (there is no data), then you are connected to Frenetic.

12.3.2 packet_in

PYTHON `packet_in(`*switch_id*`, `*port_id*`, `*payload*`)`

REST
```
 { "type": "packet_in", "switch_id": 1981745,
"port_id": 1, "payload": {
"id": 19283745, "buffer": "AB52DF57B12BF87216345" }}
```

`packet_in` is described in detail in Section 2.5.1.

In the Python version, *switch_id* is a 64-bit DPID per OpenFlow specs, and *port_id* is a 32-bit integer. *payload* is a Python object of either class `Buffered` or `NotBuffered`. Buffered objects have attributes `buffer_id` and`buffer` with the decoded data. NotBuffered objects only have a `data` attribute with the decoded data.

In the REST version, the presence of the `id` attribute means the packet is Buffered. The `buffer` attribute is the packet contents encoded in Base64.

The Frenetic Packet class, also described in Section 2.5.1 eases the programming in packet processing.

12.3.3 port_down

PYTHON `port_down(`*switch_id*`, `*port_id*`)`

REST
```
{ "type": "port_down", "switch_id": 1981745, "port_id": 1 }
```

The `port_down` event is fired when the port is disconnected, deactivated, reconfigured, or removed from the OpenFlow engine.

12.3.4 port_up

PYTHON	`port_up(`*switch_id*, *port_id*`)`
REST	`{ "type": "port_up", "switch_id": 1981745, "port_id": 1 }`

The `port_up` event is fired when the port is connected, activated, reconfigured, or assigned to the OpenFlow engine and is ready to use.

12.3.5 switch_down

PYTHON	`switch_down(`*switch_id*`)`
REST	`{ "type": "switch_down", "switch_id": 1981745 }`

The `switch_down` event is fired when the switch is gracefully stopped, or the OpenFlow engine has been stopped.

12.3.6 switch_up

PYTHON	`switch_up(`*switch_id*, *ports*`)`
REST	`{ "type": "switch_up", "switch_id": 1981745, "ports": [1,2] }`

The `switch_up` event is fired when the switch and OpenFlow engine are ready to use. The operational ports connected to OpenFlow are sent in `ports`.

12.4 Commands

Commands are called from Python by calling the method listed below. Commands may send a reply, as in `port_stats` or not, as in `pkt_out`.

Commands in REST are sent to the listed URL via GET or POST. The JSON data listed is an example request (for POSTs) or response (for GETs).

12.4.1 current_switches

PYTHON	`current_switches()`

REST	`GET /current_switches`	`[ {"switch_id": 1981745, "ports": [1,2] },`
		`{"switch_id": 9435797, "ports": [1] } ]`

The `current_switches` command retrieves a dictionary of operational, OpenFlow enabled switches and their operational ports. The dictionary key is the DPID of the switch and the value is the list of port numbers for that switch.

12.4.2 config

PYTHON	`config(`*options*`)`

REST	`POST /config`	`{"cache_prepare":"keep", "field_order":"default",`
		`"remove_tail_drops": false, "dedup_flows": true,`
		`"optimize": true }`

`config` sets compiler options for Frenetic. These options are applied on the next `update` command.

cache_prepare ("empty" or "keep", defaults to empty): If keep, keep old policies after calling `update` command. There is an implicit Union between the old and new policies in the new setup.

dedup_flows (boolean, defaults to true): If true, remove any OpenFlow table rules that are exactly alike.

field_order ("default", "heuristic", or a list of < separated fields, defaults to heuristic): Set field order priority. On occasion, setting this may reduce the OpenFlow table size. The heuristic setting attempts the optimal ordering based on the fields in the policy.

optimize (boolean, defaults to true): If true, attempt to optimize the number of OpenFlow rules.

remove_tail_drops (boolean, defaults to false): If true, remove any drop rules from the end of the OpenFlow table. This is necessary on switches like the Dell where the ACL table incorrectly prioritizes itself over all L2 and L3 table rules.

12.4.3 pkt_out

PYTHON	`pkt_out(`*switch_id*, *payload*, *plist*, *inport*`)`

REST	`POST /pkt_out`	`{"switch":1981745, "in_port":1,`
		`"actions": [` pol_1, pol_2 . . . pol_n `],`
		`"payload": { "id": 19283745,`
		`"buffer": "AB52DF57B12BF87216345" } }`

`pkt_out`, which in many ways is the analogue of the `packet_in` hook, sending a packet out to the switch for processing.

The Python parameters and REST attributes are the same as their `packet_in` counterparts. The exception is `actions` which is a Python or JSON list of policies. The policies you can use here are limited . . . the limitations are described in detail in Section 2.5.1.

The Frenetic Packet class, also described in Section 2.5.1 eases the programming in packet processing.

12.4.4 port_stats

Python		`port_stats(`*switch_id*, *port_id*`)`
REST	`GET /port_stats/`*switch_id*/*port_id*	`[ { "port_no":1, . . . }, . . . ]`

`port_stats` retrieves current port-level statistics for a certain switch and port. Sending a `port_id` of 0 retrieves stats of each operational port on that switch. Statistics attributes include:

port_no Port number

rx_packets Number of packets received

tx_packets Number of packets transmitted

rx_bytes Number of bytes received

tx_bytes Number of bytes transmitted

rx_dropped Number of packets attempted to receive, but dropped

tx_dropped Number of packets attempted to transmit, but dropped

rx_errors Number of packets errored upon receive

tx_errors Number of packets errored upon transmit

rx_fram_err Number of packets received with frame errors

rx_over_err Number of packets received with buffer overrun errors

rx_crc_err Number of packets received with CRC errors

collisions Number of collisions detected

In Python, the stats are returned as a list of dictionaries, one dictionary for each port requested. The dict keys are the attributes listed above.

12.4.5 query

PYTHON		`query(`*label*`)`
REST	`GET /query/`*label*	`{ "packets":1000, "bytes": 8000 }`

`query` retrieves statistics from a query bucket named `label`. This label should have been set up as a `SendToQuery(label)` policy.

12.4.6 update

PYTHON		`update(`*policy*`)`
REST	`POST /update_json/`*client_id*	*policy*

`update` sends a NetKAT policy to Frenetic, which will compile it into OpenFlow flow tables for each connected switch. In REST, the policy itself is the JSON packet representing the policy – generally the outermost envelope is a policy combinator like `Union`.

12.5 Frenetic Command Line

12.5.1 Common Options

All forms of the Frenetic command line accept the following parameters:

–verbosity Log level, which can be debug, info, error. Defaults to info.

–log Path to write logs to, or the special paths stdout and stderr. Defaults to stderr.

12.5.2 Command Line Compiler

```
./frenetic.native dump [ local | global | virtual ]
```

The commmand line compiler accepts Raw NetKAT files and compiles them into OpenFlow flow tables. The command line options vary depending on the type of compiler used:

Local

```
./frenetic.native dump local FILE
```

–dump-fdd dump a dot file encoding of the intermediate representation (FDD) generated by the local compiler

–json Parse input file as JSON.

–no-tables Do not print tables.

–print-fdd print an ASCI encoding of the intermediate representation (FDD) generated by the local compiler

–switches n number of switches to dump flow tables for (assuming switch-numbering 1,2,...,n)

Global

```
./frenetic.native dump global FILE
```

–dump-auto dump a dot file encoding of the intermediate representation generated by the global compiler (symbolic NetKAT automaton)

–dump-fdd dump a dot file encoding of the intermediate representation (FDD) generated by the local compiler

–json Parse input file as JSON.

–no-tables Do not print tables.

–print-auto print an ASCI encoding of the intermediate representation generated by the global compiler (symbolic NetKAT automaton)

–print-fdd print an ASCI encoding of the intermediate representation (FDD) generated by the local compiler

Virtual

```
./frenetic.native dump virtual FILE
```

–dump-fdd dump a dot file encoding of the intermediate representation (FDD) generated by the local compiler

–peg file Physical egress predicate. If not specified, defaults to peg.kat

–ping file Physical ingress predicate. If not specified, defaults to ping.kat

–print-fdd print an ASCI encoding of the intermediate representation (FDD) generated by the local compiler

–print-global-pol print global NetKAT policy generated by the virtual compiler

–ptopo file Physical topology. If not specified, defaults to ptopo.kat

–veg file Virtual egress predicate. If not specified, defaults to veg.kat

–ving file Virtual ingress predicate. If not specified, defaults to ving.kat

–ving-pol file Virtual ingress policy. If not specified, defaults to ving_pol.kat

–vrel file Virtual-physical relation. If not specified, defaults to vrel.kat

–vtopo file Virtual topology. If not specified, defaults to vtopo.kat

12.5.3 Compile Server

```
./frenetic.native compile-server [ --http-port=9000 ]
```

The compile server is an HTTP server, but it only compiles NetKAT policies into OpenFlow Flow Tables and outputs the response. It does not connect to any OpenFlow switch or pass back OpenFlow events. It responds to the following REST commands:

POST /compile Accepts NetKAT Raw format input and returns JSON-based flow table

POST /compile_pretty Accepts NetKAT Raw format input and returns human-readable flow table

GET /config Returns JSON-based current comiler options

POST /config Changes current comiler options. See command `config` above.

GET /*switch_id***/flow_table** Compiles current policy and returns JSON-based flow table

POST /update Accepts NetKAT Raw format input, but does not compile or return output. It is expected that an update will be followed by a `GET /`*switch*`/flow_table` to trigger the compilation

12.5.4 HTTP Controller

```
./frenetic.native http-controller [ --http-port=9000 ] [ --openflow-port=6633 ]
```

The HTTP Controller is the option we've been using the most in this book. It accepts REST commands to run the controller, and speaks OpenFlow to the switch. You can adjust the default port numbers as above.

12.5.5 Shell

```
./frenetic.native shell
```

The Shell is a REPL for Frenetic. Like the HTTP server, it connects via OpenFlow to switches. It can load Raw NetKAT policies, compile them to the server, and print out current flow-tables. It does not respond to OpenFlow events, however, like the HTTP server does. It's a good tool for debugging.

To see shell commands, start up Frenetic shell and type `help`.

Chapter 13

Productionalizing

Once you have your Frenetic-based application written and debugged on Mininet, you can run it in on a physical network testbed or in production.

13.1 Installing Frenetic on Bare Metal Linux

We've been running all of our experiments in Virtual Machines, and you can certainly use VM's for production as well. You simply need to set up port forwarding for the OpenFlow port (6633 by default on OpenFlow 1.0). To do this, you can add the following to the file `Vagrantfile`:

```
# Add this to forward OpenFlow traffic
config.vm.network "forwarded_port", guest: 6633, host: 6633
```

In production, however, the speed of delivering Packet In's, Packet Out's, and rules to switches is crucial so removing the extra VM network stack overhead makes sense.

To install Frenetic and your network application on a bare metal server, first install Ubuntu Server 14.04. This is the OS used by Frenetic VM.

Next install Git from your administrative login:

```
ubuntu@ubuntu:~$ sudo apt-get update
ubuntu@ubuntu:~$ sudo apt-get install git
\end{console}

Retrieve the code from the Frenetic-vm Github repository:

\begin{minted}{console}
ubuntu@ubuntu:~$ git clone https://github.com/frenetic-lang/frenetic-vm
\end{console}

Then run the administrative provisioning script.  This installs all the supporting software for
```

```
Frenetic, and should take somewhere from 5 to 30 minutes.

\begin{minted}{console}
ubuntu@ubuntu:~$ cd frenetic-vm
ubuntu@ubuntu:~/frenetic-vm$ sudo ./root-bootstrap.sh
\end{console}

Create a non-root user under which Frenetic and the application will run.  In our examples, we will
use \codefilename{frenetic} user:

\begin{minted}{console}
ubuntu@ubuntu:~$ sudo adduser frenetic
Adding user `frenetic' ...
Adding new group `frenetic' (1001) ...
Adding new user `frenetic' (1001) with group `frenetic' ...
Creating home directory `/home/frenetic' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for frenetic
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n]
```

Add the `frenetic` user to do the `/etc/sudoers` file so it can perform administrative tasks. This is required for the next scripts to run, and if you wish, you can revoke sudo permissions afterwards (they are not required to run Frenetic or the network application):

```
ubuntu@ubuntu:~$ sudo adduser frenetic sudo
Adding user `frenetic' to group `sudo' ...
Adding user frenetic to group sudo
Done.
```

Copy the `user-bootstrap.sh` script to the `frenetic` user's workspace:

```
ubuntu@ubuntu:~/frenetic-vm$ sftp frenetic@localhost
The authenticity of host 'localhost (127.0.1.1)' can't be established.
ECDSA key fingerprint is 3c:f4:ba:46:f0:b7:1f:74:cd:34:33:f9:12:5e:3a:3e.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
frenetic@localhost's password:
Connected to localhost.
sftp> put user-bootstrap.sh
```

```
Uploading user-bootstrap.sh to /home/frenetic/user-bootstrap.sh
user-bootstrap.sh                                                                                 100%  296
sftp> quit
```

Then switch over to the `frenetic` workspace and run the script. This installs and compiles the latest version of Frenetic, and takes about 15 minutes.

```
ubuntu@ubuntu:~$ sudo -i -u frenetic
frenetic@ubuntu:~$ mkdir src
frenetic@ubuntu:~$ chmod u+x user-bootstrap.sh
frenetic@ubuntu:~$ ./user-bootstrap.sh
```

Lastly, you need to retrieve your network application code and store it in the `frenetic` user's workspace. We generally store network applications in Github and just use Git tools to retrieve the latest Master copy. That way it's easy to update later. In the examples below, we'll assume the network application is in `src/l2_learning_switch/learning4.py`.

13.2 Control Scripts

Ubuntu 14.04 includes the Upstart utility to wrap programs as daemons. This is handy for starting and stopping both Frenetic and your network application. Creating Upstart scripts for both handles logging, and automatically starting up the daemons from a cold boot.

Here is a sample Upstart script for Frenetic, which you can place in `/etc/init/frenetic.conf` (you must do this with `sudo`):

```
# Frenetic startup script
#

description "Frenetic OpenFlow Controller"

start on runlevel [2345]
stop on runlevel [!2345]

respawn
respawn limit 10 5
umask 022

# log output to /var/log/upstart
console log

setuid frenetic
chdir /home/frenetic/src/frenetic
exec ./frenetic.native http-controller --verbosity debug
```

And here is one for your network application. We'll assume the network application is in src/l2_learning_switch/learning4.py.

```
# L2 Learning Switch network application
#
# Uses Frenetic as the OpenFlow Controller

description "L2 Learning Swtich"

start on runlevel [2345]
stop on runlevel [!2345]

respawn
respawn limit 10 5
umask 022

# log output to /var/log/upstart
console log

setuid frenetic
chdir /home/frenetic/src/l2_learning_switch
exec python learning4.py
```

And you can control both processes with standard Ubuntu commands:

sudo service frenetic start Starts the Frenetic controller. Note, only one copy can be running, so issuing it when Frenetic is already running does nothing.

sudo service frenetic stop Stops the Frenetic controller.

sudo service frenetic status Displays the status of Frenetic - whether it's running or not.

The commands are similar for the network application l2_learning_switch.

One nice feature of Upstart is logging. Using the Upstart script for Frenetic as above places the logs in /var/log/upstart/frenetic.log. By default logs are rotated daily, compresssed, and deleted after 7 days.

Bibliography

Nate Foster, Michael J. Freedman, Arjun Guha, Rob Harrison, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Mark Reitblatt, Jennifer Rexford, Cole Schlesinger, Alec Story, and David Walker. Languages for software-defined networks. *IEEE Communications*, 51(2):128–134, Feb 2013.

Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for netkat. *SIGPLAN Not.*, 50(1): 343–355, Jan 2015. ISSN 0362-1340. doi: 10.1145/2775051.2677011. URL `http://doi.acm.org/10.1145/2775051.2677011`.

Brian P. Hogan. *tmux: Productive Mouse-Free Development*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2012. ISBN 978-1-93435-696-8. URL `http://www.pragprog.com/titles/bhtmux/tmux`.

Steffen Smolka, Spiridon Aristides Eliopoulos, Nate Foster, and Arjun Guha. A fast compiler for netkat. *CoRR*, abs/1506.06378, 2015. URL `http://arxiv.org/abs/1506.06378`.