frenetic >>

Programmers Guide

Craig Riecke

with

Others

Draft of April 27, 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 15.1.

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:~$
```

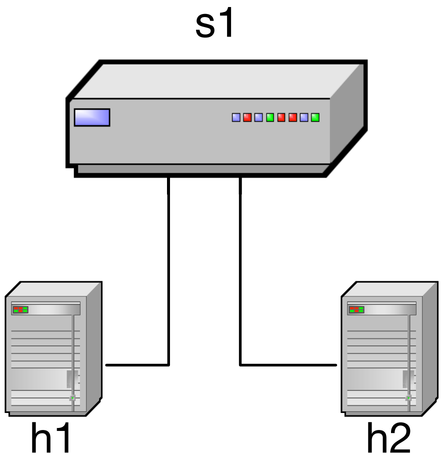
So you are now working inside an Ubuntu-based VM.

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`:

```
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, [ Output(Physical(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:

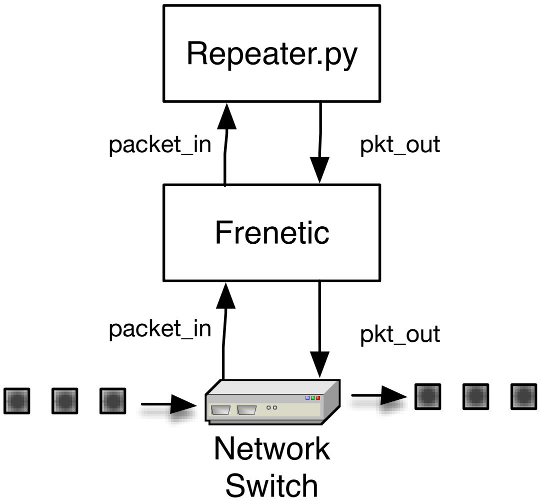
```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=149 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=97.2 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=88.7 ms
```

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 matcha 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.
IPSrcEq($addr$, $mask$)	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.
IPDstEq($addr$, $mask$)	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 ˜$p1$

The precedence is the same as for most Boolean operators in normal programming languages: Not, then And, then Or.

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`(n)	Set the output port for the packet to port n.
`SendToController`(tag)	Send packet to controller with tag 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.
SetIPSrc($addr$)	Set IP source address to $addr$. Note there is no mask here, as in the equivalent predicate.
SetIPDst($addr$)	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 6.

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

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

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 user-defined statistics associated with a particular label. We'll cover this in Chapter 8.
`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, [ Output(Physical(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 payload, you'll need some assistance. The payload is the raw network packet data, not parsed or translated at all. So for the Ethernet Source address, for example, you'd need to examine bytes 14 through 19 (byte ordering starts at 0). Working at this low-level is exceedingly error prone, so Frenetic leverages the RYU Packet library. RYU is an open source project spearheaded by NTT (Nippon Telephone and Telegraph) and its Python packet parsing library is very solid and complete.

Frenetic provides a simple API on top of RYU Packet.

```
# You must import the proper protocol from the RYU packet library to use it
from ryu.lib.packet import ethernet, arp
...
    def packet_in(self, dpid, port_id, payload):
        ethernet_packet = self.packet(payload, 'ethernet')
        src_mac = ethernet_packet.src
        if ethernet_packet.ethertype = 0x806:
          arp_packet = self.packet(payload, 'arp')
          src_ip = arp_packet.src_ip

...
```

`p = self.packet(payload, protocol)` turns the raw `payload` into a parsed packet of type `protocol`. From there, the fields of `p` are the parsed values of that protocol. You can think of `p` as a view into `payload` with the `protocol` lens attached. In the example above, for example, `payload` is both an Ethernet packet and an ARP packet, and the variables `ethernet_packet` and `arp_packet` provide respective views into it. If `payload` is not parseable into that protocol, `None` is returned.

http://ryu-zhdoc.readthedocs.org/en/latest/library_packet_ref.html contains a complete reference for RYU packets. Here are the most popular protocols and fields:

ethernet

`dst`	Destination mac address string, formatted as ”08:60:6e:7f:74:e7”
`src`	Source mac address string
`ethertype`	Ethernet frame type. Popular values are 0x800 for IP version 4, or 0x806 for ARP.

Constructor: `ethernet.ethernet(dst, src, ethertype)`

vlan

`vid`	VLAN id
`pcp`	Priority Code Point
`ethertype`	Ethernet frame type. The outer Ethernet packet has ethertype 0x8100, so this is the ethertype of the packet’s data.

Constructor: `ethernet.vlan(pcp, vid, ethertype)`

ipv4

`proto`	The IP version 4 protocol. Popular values are 6 for TCP and 17 for UDP.
`src`	Source address, a 32 bit integer
`dst`	Destination address, a 32 bit integer

Constructor: `ipv4.ipv4(proto, src, dst)`

arp

`opcode`	Popular values are `arp.ARP_REQUEST` and `arp.ARP_REPLY`.
`src_mac`	Source mac address string, formatted as ”08:60:6e:7f:74:e7”
`src_ip`	Source IP address, a 32 bit integer
`dst_mac`	Destination mac address string
`dst_ip`	Destination IP address

Constructor: `arp.arp_ip(opcode, src_mac, src_ip, dst_mac, dst_ip)`

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, wrapped in a Buffered or Unbuffered object type.
`policy_list`	Python list of NetKAT policies (actions) 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 `SetIPSrc` 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.
- `SetPort` is not available, but `Output(Physical(`p`))` can be substituted. The difference is subtle. `SetPort` can be used anywhere in a policy sequence, and can be followed by more packet modifications. `Output` sends the packet out immediately, and according to OpenFlow it is always the last action executed if present.
- Only simple policies are doable, so you can't use policy operators like `Union`, `Seq` and `IfThenElse`. Instead, you can send multiple policies in a list, where there's an implied `Seq` operator between them.

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, reserialize it through RYU's packet library API, and send it in `payload`. This is the only way to modify data that's not accessible to a NetKAT policy - e.g. the IPv4 source address is settable by NetKAT policy `SetIPv4`, but there's no equivalent policy for the ARP opcode. 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 all packets, and saves you the costly step of parsing and reserializing the packet in the controller. 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), Output(Physical(next_port)) ]
        )
```

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

```
from ryu.lib.packet import ethernet, arp
...

    def payload(self, e, pkt):
        p = packet.Packet()
        p.add_protocol(e)
        p.add_protocol(pkt)
        p.serialize()
        return NotBuffered(binascii.a2b_base64(binascii.b2a_base64(p.data)))

    def packet_in(self, dpid, port_id, payload):
        ethernet_packet = self.packet(payload, 'ethernet')
        arp_packet = self.packet(payload, 'arp')
        # Flip a request into a reply and vice versa
        arp_packet.opcode =  \
          arp.ARP_REPLY if arp_packet.opcode == arp.ARP_REQUEST \
          else arp.ARP_REPLY
            # Reserialize it back into a raw packet
        new_payload = self.payload(ethernet_packet, arp_packet)
        self.pkt_out(dpid, new_payload, [ Output(Physical(1)) ])
...
```

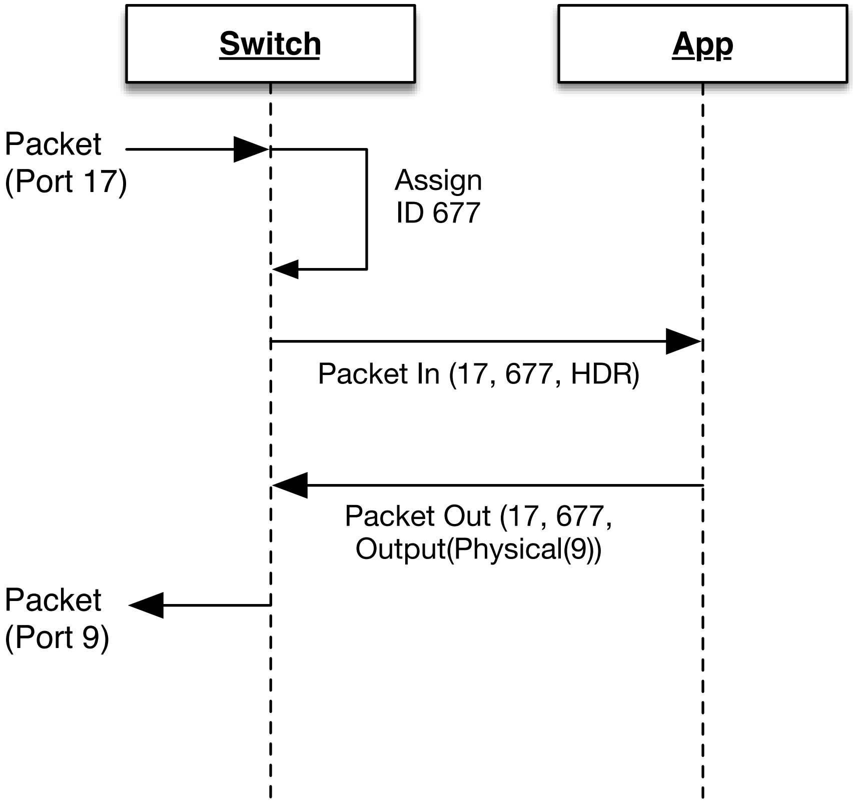
You can also create packets from scratch with a variation of Direct Modification. First create the packet views with standard RYU Packet library constructors. Then just call `self.payload()` on them to create the raw packet and send it.

```
from ryu.lib.packet import ethernet, arp
...
    def send_arp_reply(port, src_mac, dst_mac, src_ip, dst_ip):
        # Immediately send an ARP reply when a port comes up.
        e = ethernet.ethernet(dst=dst_mac, src=src_mac, ethertype=0x806)
        a = arp.arp_ip(arp.ARP_REPLY, src_mac, src_ip, dst_mac, dst_ip)
            # Serialize it into a raw packet
        new_payload = self.payload(ethernet_packet, arp_packet)
        self.pkt_out(dpid, new_payload, [ Output(Physical(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 `Output(Physical(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.
- It can't retrieve flow table counters. Since one NetKAT policy may expand to many OpenFlow rules, or even be optimized to OpenFlow Rules, there is no good way to map and retrieve this data.
- 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 Error, 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 8.

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, [ Output(Physical(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`:

```
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`:

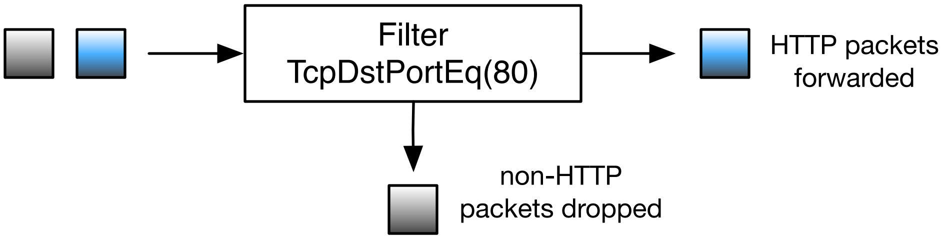
- It works on a two port switch, but not anything bigger. And the ports absolutely have to be numbered 1 and 2 . . . otherwise, the whole program doesn't work. And those ports need to be functioning.
- More subtly, this program can drop packets. There is a short lag between when the switches come up 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.

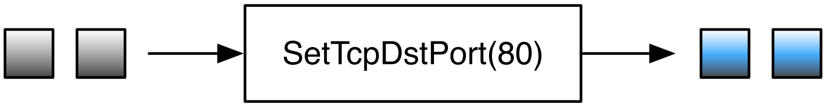
We will correct both of these problems in section 3.3

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.



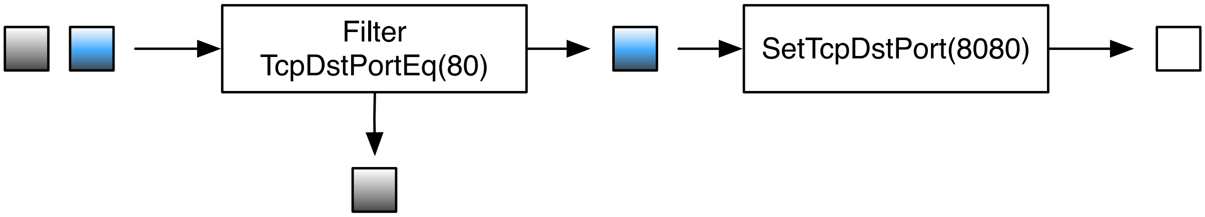


A filter, shown at the top, takes as input a stream of packets. Packets that match its criteria are sent to the next policy, packets that don't match are dropped. Two special policies, `id` and `drop`, let through all or no packets respectively.

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*.

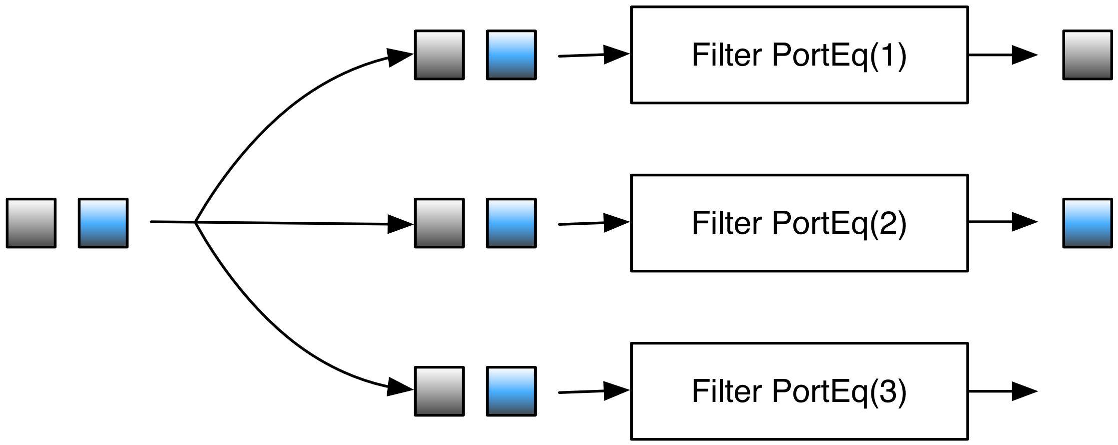
A rule is like an OpenFlow flow table entry, but with more options. If you think about it, a switch receives a huge firehose blast of packets, and you want the switch to perform a targeted action on a very small subset of those packets. In other words, you want a *filter* and some *actions*. It's like 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 except multiple* `SetPorts`*.*

`SetPort` is an exception to the rule that we'll see in a minute. Here is a Union of policies.



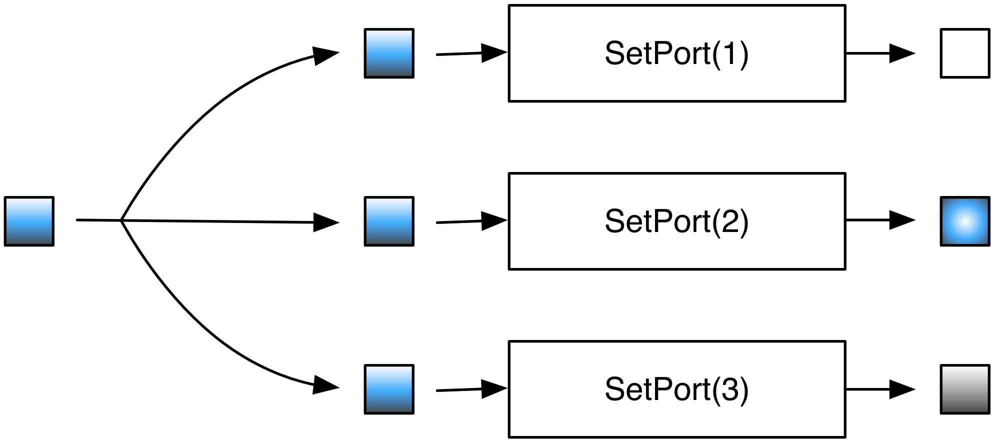
A `Union` of policies makes logical copies of packets and sends the copy of each packet through each policy in parallel. In the above example, the two incoming packets are copied three times, one for each rule. In this case, the top filter only forwards a copy of the first packet matching `PortEq(1)`. 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 multiple SetPorts and rules that DO NOT overlap. Use* `IfThenElse` *to combine rules that DO overlap.*

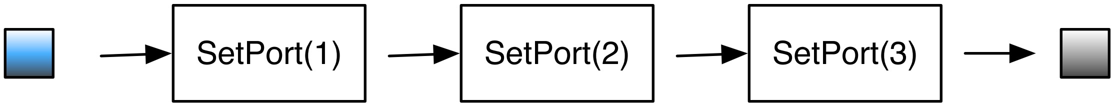
We'll explain `IfThenElse` in a little bit.

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 need to use multiple `SetPort` actions.



A Union of Three SetPorts Sends Three Packets



A Seq of Three SetPorts Sends One Packet

If you string these `SetPorts` together in a `Seq`, the single packet will be assigned each output port in turn, and the last assigned egress port will "win". This is probably not what you want. On the other hand, stringing these together with `Union` makes more sense. You actually WANT to make extra copies of the packet, and send each copy out a different port.

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:

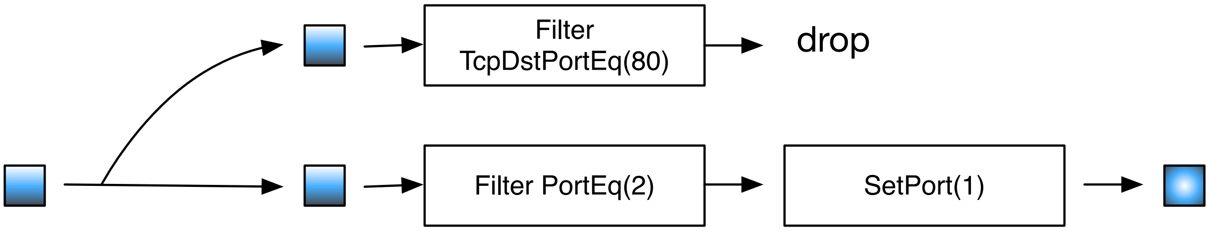
- `Filter(PortEq(1)) >> SetPort(2)` Sends traffic from port 1 out port 2
- `Filter(PortEq(2)) >> SetPort(1)` Sends traffic from port 2 out port 1

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.

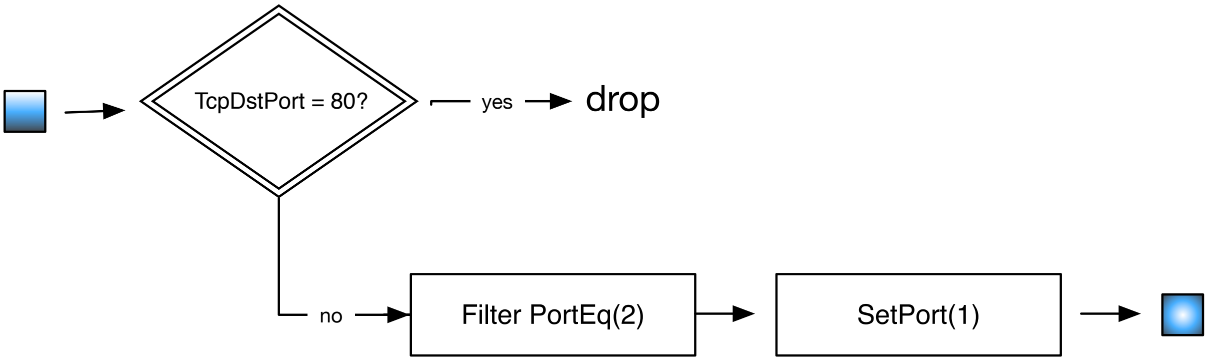


These two rules overlap. A packet can both have a TCP destination port of 80 and arrive on switch 1, port 2. What happens if we combine them with `Union`, and a packet like that arrives at the switch? The packet will be copied twice, then:

- `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) | SetPort(3) | SetPort(4)) |
Filter(PortEq(2)) >> (SetPort(1) | SetPort(3) | SetPort(4)) |
Filter(PortEq(3)) >> (SetPort(1) | SetPort(2) | SetPort(4)) |
Filter(PortEq(4)) >> (SetPort(1) | SetPort(2) | SetPort(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); SetPort(2); SetPort(3); SetPort(4) ]
```

Now suppose we're looking only at port 1. We want all of the `SetPort`'s here except `SetPort(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); SetPort(3); SetPort(4) ]
```

And now you can pass this to the NetKAT `Union` operator:

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

. . . which creates a parallel composition out of the entire list:

```
SetPort(2) | SetPort(3) | SetPort(4)
```

You can pass a Python list of policies to either `Union` or `Seq`, making it easy to build very large policies from component parts very quickly.

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)) >> \
      Union( 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 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())
```

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

to:

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

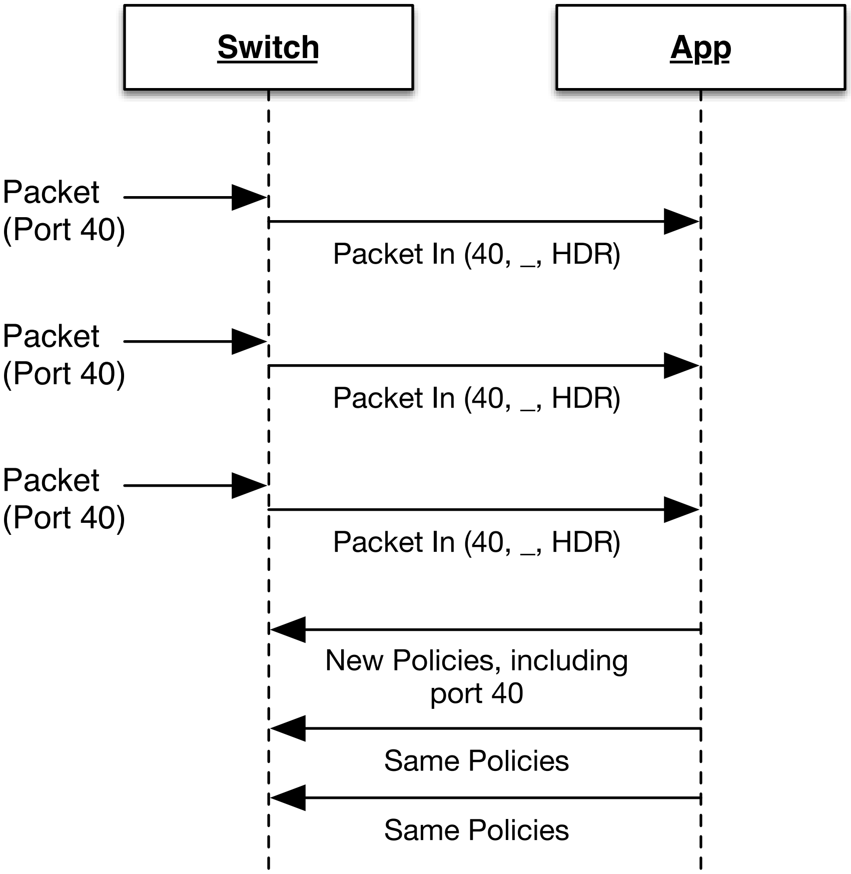
```
  def packet_in(self, dpid, port_id, payload):
    self.all_ports.add(port_id)
    self.update(self.policy())
```

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)) >> \
      Union( 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 = [ Output(Physical(p)) for p in self.all_ports if p != port_id ]
    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 except multiple* `SetPorts`*.*

Principle 3 *Use* `|` *between multiple SetPorts and 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():

  # hosts is a dictionary of MAC addresses to ports
  #  { "11:11:11:11:11:11": 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 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_except(self, in_port_id):
    return [p for p in self.ports if p != in_port_id]
```

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 ryu.lib.packet import ethernet
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

    # Parse the interesting stuff from the packet
    ethernet_packet = self.packet(payload, "ethernet")
    src_mac = ethernet_packet.src
    dst_mac = ethernet_packet.dst

    # 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 = [ Output(Physical(dst_port)) ]
    else:
      actions = [ Output(Physical(p)) for p in 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 RYU packet parsing routines 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(
        EthDstNotEq( self.nib.all_learned_macs() ),
        SendToController("learning_app"),
        Union( self.policies_for_dest(self.nib.all_mac_port_pairs()) )
      )
```

TODO: The first ping from h1 to h2 fails the first time, even though all the subsequent ones work, and the seond pingall works 100 percent.

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_down(self, dpid, port_id):
  self.nib.unlearn( self.nib.mac_for_port(port_id) )
  self.nib.delete_port(port_id)
  self.update(self.policy())

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())
```

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:

```
Output(Physical(2)) | Output(Physical(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

  # Parse the interesting stuff from the packet
  ethernet_packet = self.packet(payload, "ethernet")
  src_mac = ethernet_packet.src
  dst_mac = ethernet_packet.dst
  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 = [ Output(Physical(dst_port)) ]
    else:
      actions = [ ]
  else:
    actions = [ Output(Physical(p)) for p in 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:

```
    # Parse the interesting stuff from the packet
    ethernet_packet = self.packet(payload, "ethernet")
    vlan_packet = self.packet(payload, "vlan")
    src_mac = ethernet_packet.src
    dst_mac = ethernet_packet.dst
    src_vlan = vlan_packet.vid
```

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

Spanning Tree Alternatives

6.1 What's Wrong with STP Protocols?

6.2 Design

6.3 Calculating Shortest Paths

6.4 Core/Edge Separation

Chapter 7

Proxy ARP

7.1 The ARP Protocol

7.2 Design

7.3 Snooping on ARP Requests and Replies

7.4 Replying

7.5 Maintaining State Across Restarts

Chapter 8

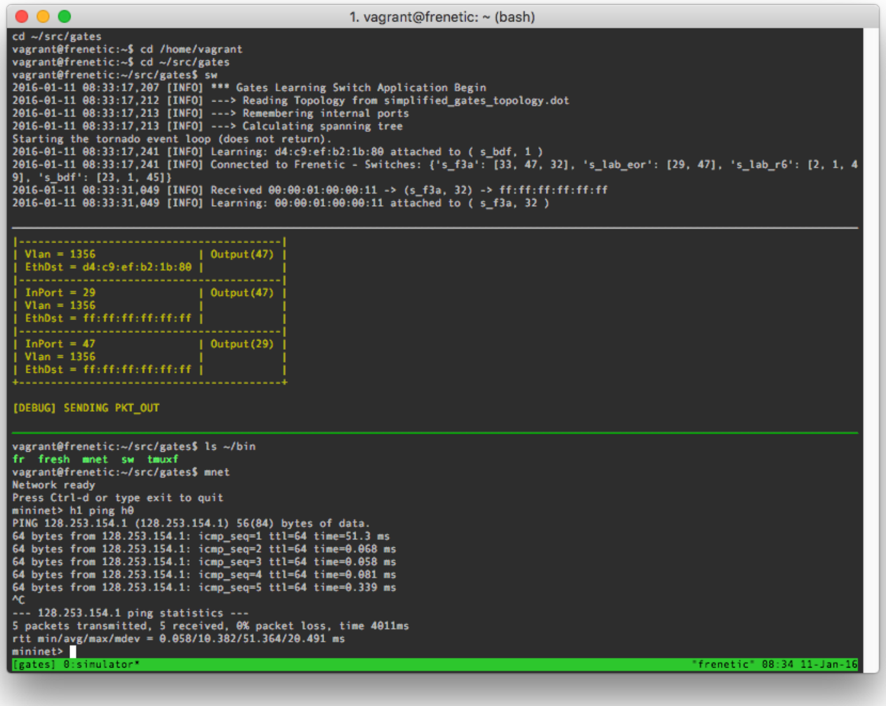
Gathering Statistics

Chapter 9

SDN Development Tools

9.1 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.



```
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.

9.2 Open VSwitch Utilities

9.3 TCPDump

9.4 Mininet Network Modelling

Chapter 10

Network Address Translation

10.1 Why Do We Need NAT?

10.2 Design

Chapter 11

Routing

11.1 Design

11.2 Configuring Route Tables

11.3 Using Link State

Chapter 12

Modularization

12.1 Sharing Actions in Rules and Packet Outs

12.2 Subclassing

12.3 Multiple Network Apps

12.4 Clustering

Chapter 13

Frenetic REST API

13.1 REST URL's

13.2 Incoming, Northbound Events

13.3 Messages

Chapter 14

Frenetic/NetKAT Reference

14.1 NetKAT Predicates

14.1.1 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”)     EthSrcNotEq(”72:00:08:bc:5f:a0”)
        EthSrcEq(”72:00:08:bc:5f:a0”,     EthSrcNotEq(”72:00:08:bc:5f:a0”,
        ”82:9b:41:a6:16:f8”)              ”82:9b:41:a6:16:f8”)
        EthSrcEq([”72:00:08:bc:5f:a0”,    EthSrcNotEq([”72:00:08:bc:5f:a0”,
        ”82:9b:41:a6:16:f8”])             ”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)           EthTypeNotEq(0x800)
        EthTypeEq(0x800, 0x806)    EthTypeNotEq(0x800, 0x806)
        EthTypeEq([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.

IPDst

PYTHON	`IPDstEq("192.168.57.100")`
	`IPDstEq("192.168.57.0", 24)`
		`IPDstNotEq("192.168.57.100")`
		`IPDstNotEq("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 } }`	

`IPDst` matches packets with a particular IP destination address, or destination network. It only matches IP v4 packets, as per Openflow 1.0.

`IPDstEq` 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 `IPDstEq("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`.

IPSrc

```
PYTHON  IPSrcEq(”192.168.57.100”)     IPSrcNotEq(”192.168.57.100”)
        IPSrcEq(”192.168.57.0”, 24)   IPSrcNotEq(”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 }}
```

`IPSrc` matches packets with a particular IP source address, or source network. It only matches IP v4 packets, as per Openflow 1.0.

`IPSrcEq` 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 `IPSrcEq("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": "port", "value": 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).

SwitchEq

```
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.

14.1.2 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.

14.2 Policies

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.

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.

SetIPDst

Python	`SetIPDst(”192.168.57.100”)`
Raw	`ip4Dst := 192.168.57.100`
REST	`{ "type": "mod", "header": "ip4dst", "value": {"addr": "192.168.57.100" } }`

`SetIPDst` sets the IP v4 destination address. Note that unlike the `IPDstEq` 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.

SetIPSrc

Python	`SetIPSrc(”192.168.57.100”)`
Raw	`ip4Src := 192.168.57.100`
REST	`{ "type": "mod", "header": "ip4src", "value": {"addr": "192.168.57.100" } }`

`SetIPSrc` sets the IP v4 destination address. Note that unlike the `IPSrcEq` 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)`
RAW	`port := 1`
REST	`{ "type": "mod", "header": "port", "value": 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.

14.3 Events

14.4 Commands

14.5 Compiler Directives

14.6 Frenetic Command Line

Chapter 15

Productionalizing

15.1 Installing Frenetic on Bare Metal Linux

15.2 Control Scripts

15.3 Logging

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`.