frenetic >>

Programmers Guide

Craig Riecke

with

Others

Draft of March 30, 2016

Contents

Chapter 1

Quick Start

In this book, you will use Frenetic to create a full-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 itself. 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.

- 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. This is more reliable than downloading a multi-gigabyte VM file.

- 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 builds a test network inside of Linux. It can simulate 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. We'll use it throughout this book.

Wireshark captures and analyzes network traffic. It's a great debugging tool, and very necessary for sifting through large amounts of data.

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
```

Then 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. You don’t really need to know Ubuntu, but just know that Mac OS and Windows commands won’t necessarily work here.

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 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 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 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 act.

1.4 A Repeater

You 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:~$
```

Create a tutorial directory for your use:

```
vagrant@frenetic:~$ mkdir tutorial
vagrant@frenetic:~$ cd tutorial
```

Now we'll write our first network application. You can use your favorite Unix text editor - vim and nano are already installed, or you can install your own favorite one with Ubuntu's *apt-get* commands. Nano is a nice editor if your feeling indecisive:

```
vagrant@frenetic:~/tutorial$ nano repeater.py
```

Now type in the following network application:

```
import sys
sys.path.append(’../src/frenetic/lang/python’)
import frenetic
from frenetic.syntax import *

class RepeaterApp(frenetic.App):

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

    def packet_in(self, dpid, port_id, payload):
        out_port = 2 if port_id = 1 else 1
```

```
13          self.pkt_out(dpid, payload, [ Output(Physical(out_port_id)) ] )
14
15  app = RepeaterApp()
16  app.start_event_loop()
```

Lines 1-4 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` merely directs the switch to send all packets to our application. The interesting code is in `packet_in` and implements a *repeater*. A repeater is the oldest type of network device, and is sometimes called a *hub*. In a 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*).

`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:~$ cd src/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:~/tutorial$ python repeater.py
No client_id specified. Using d7650d3aebfc4bd9a708eef1382041ba
Starting the tornado event loop (does not return).
```

The following will appear in the Frenetic window.

```
 [INFO] GET /version
 [INFO] POST /d7650d3aebfc4bd9a708eef1382041ba/update_json
 [INFO] GET /d7650d3aebfc4bd9a708eef1382041ba/event
 [INFO] New client d7650d3aebfc4bd9a708eef1382041ba
[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:

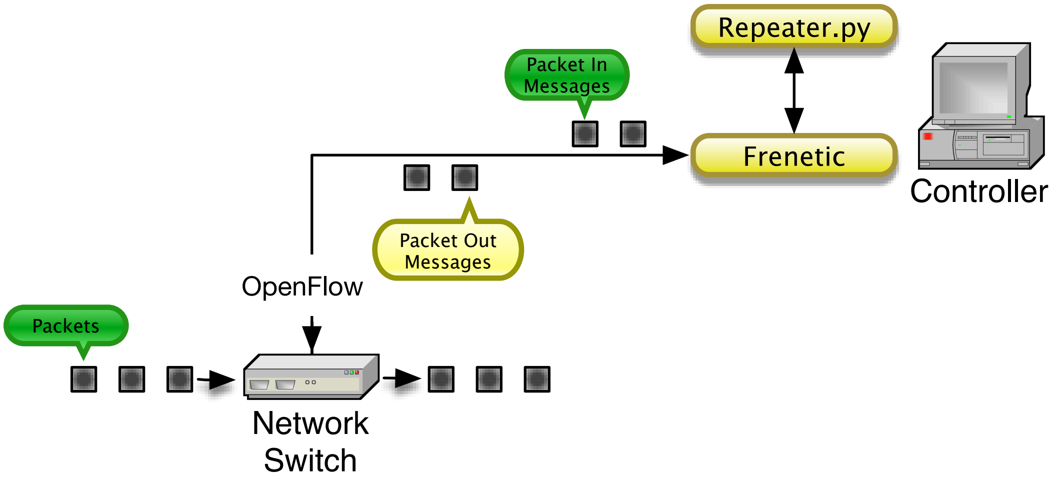
```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=149 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=97.2 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=88.7 ms
```

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

```
 [INFO] GET /d7650d3aebfc4bd9a708eef1382041ba/event
 [INFO] POST /pkt_out
[DEBUG] SENDING PKT_OUT
 [INFO] GET /d7650d3aebfc4bd9a708eef1382041ba/event
 [INFO] POST /pkt_out
[DEBUG] SENDING PKT_OUT
 [INFO] GET /d7650d3aebfc4bd9a708eef1382041ba/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. Frenetic-vm makes installing and testing all the pieces straightforward. 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. Traditional networking involves 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 workable 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 Personal Computer 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-enabled network software and devices. OpenFlow is an open protocol defined by the Open Network Foundation that manipulates the control plane of a network intermediary.

Frenetic is an OpenFlow controller, meaning it talks the OpenFlow protocol to network intermediaries. In turn, it exposes an API that can be used to write network programs easily. It works with any network intermediary 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.

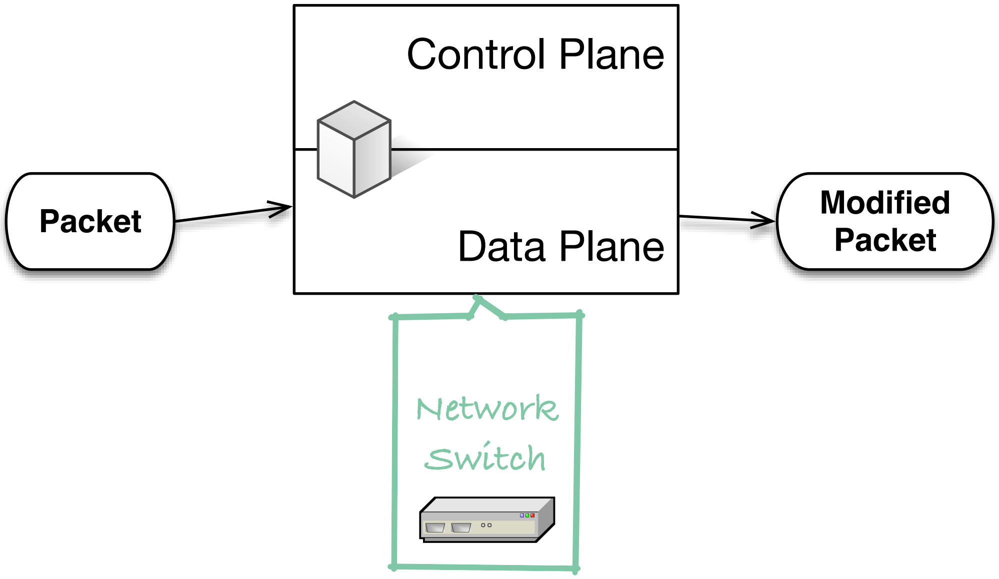
The control plane manipulates the control tables themselves. People, in turn, may manipulate the control plane through configuration software. Or packets may do it: specialized ones like OSPF neighbor exchange, ARP requests, or just examining plain ol' packets themselves. But they never actually touch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

This separation is only conceptual. You'd be hard pressed to open a network device, point to a chip and say, "That's the data plane." It helps in understanding a device, though, because they have different goals:

The data plane 's job is to move data as quickly as possible. It relies more on fast table lookups than complex algorithms.

The control plane works more flexibly, yet conservatively. Control tables should not change often, and when they do, appropriate checks and balances should be applied to ensure the data plane keeps working. For example, the Spanning Tree Protocol (or STP) ensures packets are routed along the shortest path with no loops. Calculating the spanning tree is the control plane's job, as its complex. But once that's calculated, the data plane can use it to forward packets quickly.

A traditional network device looks like this. The control plane is closed and contained fully within the box.



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 and runs outside the box. It's like an advisor that makes aribtrarily 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:

Device: 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.

Device: 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.

Device: OK!

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

Device: 82,120.

Device: (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:

A match specifies patterns of packet header and metadata values. OpenFlow 1.3 defines 40 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. Instructions can apply actions (send a packet out a port, or write some header information, or send a packet to the controller), invoke groups (like a function call in a programming language), or set variables.

A priority defines 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”.

OpenFlow's flow table model is *abstract*. An OpenFlow device is not necessarily going to find a RAM chip with the matches, instructions and priorities ... although a pure software switch like Open vSwitch might mirror it quite closely. Instead, the controller asks the device to install entries, and the device accommodates it by placing entries in its own tables. For example, a real network device might have an L3 table that matches subnets with the various ports that have IP gateways. A programmer, knowing this table exists, can write instructions that match on those ports, and place it directly in the table accordingly.

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 6653. Then you'd just point your OpenFlow network device at controller.example.com:6653. Then 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. “That's not code, that's data,” you might say. But a table is very 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.

You don't program in machine language most of the time, though, and you shouldn't have to program directly in OpenFlow tables. Why not?

- The set of matches and actions is very limited
- They are difficult to modularize and compose
- They are difficult to prove correct.

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, but you can't be more flexible than that.
- Matching against a set of values requires you 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. 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. Here, instead of writing programs that directly call OpenFlow primitives, you write programs in the NetKAT language. These programs are called *network applications* or more succinctly *net apps*.

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 net apps (northbound). Net apps talk to Frenetic with good ol' fashioned 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. This match only makes sense when EthTypeEq(0x800, 0x806) (IP or ARP).
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. This match only makes sense when EthTypeEq(0x800, 0x806) (IP or ARP).
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.

For each rule, OpenFlow allows only one kind of boolean operator: AND. If the Open-

Flow rule specifies match criteria $p1, p2, \ldots, pn$, the packet must match $p1$ AND $p2$ AND . . . AND pn. NetKAT is much more expressive, allowing the following boolean operators between predicates:

$p1$ & $p2$	Matches packets that satisfy $p1$ AND $p2$
And([$p1$, $p2$, ..., pn])	Matches packets that satisfy all the predicates: $p1$ AND $p2$ AND . . . AND pn
$p1$ | $p2$	Matches packets that satisfy $p1$ OR $p2$
Or([$p1$, $p2$, ..., pn])	Matches packets that satisfy one of the predicates: $p1$ OR $p2$ OR . . . OR 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`.

A bunch of predicates, stitched together with Boolean operators, can be used wherever a simple predicate is used. Furthermore, predicates can be assigned to Python variables. So here are some examples in Python code:

```python
import sys
sys.path.append('../src/frenetic/lang/python')
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 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 )
```

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:

`id`	Matches all packets
`drop`	Match 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 useless by themselves. To make them work, you need to form them into NetKAT *policies*. A policy is like a command. Often they are compiled down to OpenFlow actions, and in fact there's a lot of overlap between the two concepts. 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
`SetPort`(n)	Set the output port for the packet to port n.
`SendToController`(tag)	After all actions have been performed, 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. This action is only done when EthTypeEq([0x800, 0x806]) (IP or ARP).
`SetIPSrc`($addr$)	Set IP source address to $addr$. Note there is no mask here, as in the equivalent predicate. This action is only done when EthTypeEq([0x800, 0x806]) (IP or ARP).
`SetIPDst`($addr$)	Set IP destination address to $addr$. Follows same rules as SetIpSrc.
`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`. This fields is not physical fields in a packet header - rather, it is metadata that OpenFlow fills in "invisibly" in a packet. So you can't set it to a particular value, like you can other header fields. 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 (e.g. you must know the topology). We'll cover strategies for dealing with this in 10.

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

$pol1 \mid pol2$	Copy the packet and apply both $pol1$ and $pol2$ to it. Also known as *parallel composition*.
`Union([`$pol1$`, `$pol2$`, ..., `$poln$`])`	Copy the packet n times and apply policy $pol[i]$ to copy i
$pol1$ `>>` $pol2$	Apply the policy $pol1$ to the packet, then apply $pol2$ to it Also known as *sequential composition*.
`Seq([`$pol1$`, `$pol2$`, ..., `$poln$`])`	Apply each of the policies $pol1, pol2, \ldots,$ $poln$ to the packet, in order
`IfThenElse(`p`, `$pol1$`, `$pol2$`)`	If packet matches predicate p, then apply policy $pol1$, or else apply $pol2$. Either $pol1$ or $pol2$ is applied, but never both. Equivalent to `Filter(`p`) >>` $pol1 \mid$ `Filter(~`p`) >>` $pol2$

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 actually 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.

Some examples will clarify.

2.5 Commands and Hooks

The truth is, just like NetKAT predicates, NetKAT policies don't do anything by themselves. 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 all 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 7.
`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 in your net app through plain ol' Python method calls, e.g. `self.update(`*polic*

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*`)`	A packet has arrived on DPID *sw*, port *port* and either the matching Policy had a SendToController policy, or the packet matched no rules and the TableMiss rule forwards TableMiss packets to the controller. 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 once every 5 minutes or some user-defined interval, and it doesn't necessarily mean it was down before this.
`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, or the host connected to that port has been powered-off.

Your net app may be interested in one or more of these hooks. To add code to it, 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 crucial to 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.) Packets arrive here in one of two ways:

- It may match an OpenFlow rule with the action `OFPAT_OUTPUT(OFPP_CONTROLLER)` also known in NetKAT as `SendToController`.
- It may not match any OpenFlow rule. In OpenFlow 1.0, the default action is to send the packet to the controller. In OpenFlow 1.3, the default action is to drop the packet, but it can be configured to send it to the controller instead.

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 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. Unbuffered data can be held at the controller, changed and resent, dropped, multiplied into many different packets, or any arbitrary action without penalty.

Buffered data, on the other hand, exchanges flexibility for some efficiency. Only a subset of buffered packet data is sent to the controller - the amount is configurable, but defaults to 128 bytes which is usually enough for the Ethernet and IP headers. If packets are large, or the channel between the switch and controller is slow, this can save much precious time. The buffered packet will wait at the switch until it either gets changed and sent (through `pkt_out` in the case of Frenetic), or explicitly dropped, or dropped due to timeout.

Although buffering data sounds like a good idea, it requires greater care on the net apps part to prevent duplicate packets, race conditions, and so on. It's also possible that truncated packets become unparseable - for example, if they get cut off in the middle of the IP header. Because of this, very few OpenFlow switches support buffering at all. If capacity is an issue, it's better to send as few packets to the controller as possible. A judicious use of NetKAT policies in the flow table will ensure this.

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. And in the cases where all you want to decide in the controller which ports to send out a packet, this is all the infrastructure you need.

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 = 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.

A complete reference for RYU packets is available at http://ryu-zhdoc.readthedocs.org/en/latest/li The following is a list of 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 from the switch. 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 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 library from RYU to 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 packet_in(self, dpid, port_id, payload):
        (ethernet_packet, arp_packet) =
                self.packet(payload, [’ethernet’, ’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)) ])
...
```

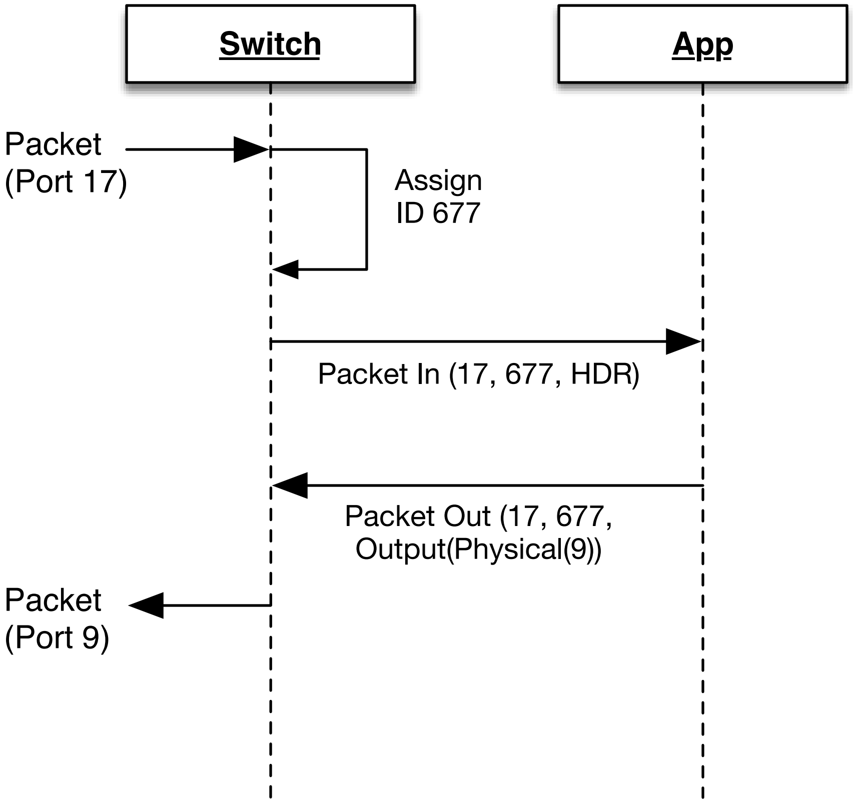
You can also create packets from scratch with a variation of Direct Modification. First create the packet views with standard RYU Packet library constructors. Then just call `self.payload()` on them to create the raw packet and send it.

```
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. The controller only does Policy Modifications to the buffered packet, saving the trouble of sending all of it back.

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 list 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 can'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 7.

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 naive Repeater application:

```
import sys
sys.path.append(’../src/frenetic/lang/python’)
import frenetic
from frenetic.syntax import *

class RepeaterApp(frenetic.App):

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

    def packet_in(self, dpid, port_id, payload):
        out_port = 2 if port_id = 1 else 1
        self.pkt_out(dpid, payload, [ Send(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 it unecessarily so since all the decisions can be made inside the switch. So let's write a more efficient one:

```
import sys
sys.path.append(’../src/frenetic/lang/python’)
import frenetic
from frenetic.syntax import *

class RepeaterApp2(frenetic.App):

    def connected(self):
        policy_port_one = Filter(PortEq(1)) >> Send(2)
        policy_port_two = Filter(PortEq(2)) >> Send(1)
        self.update( policy_port_one | policy_port_two )

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

Wow! That 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)) >> Send(2)` policy is a pretty common pattern in NetKAT. You first whittle down the incoming flood of packets to a certain subset with `Filter` and a predicate. Then you apply a policy or series of policies, `Send` being the most popular. We'll look at combining policies in the 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:

TODO

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 computer 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 in between when the switches come up and the `self.update()` installs the policies. During this lag, packets will arrive at the controller by default and 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 —. So when do you use one or the other? The following principle is easy to remember and apply.

Principle 2 *Use `Seq` or >> between filters and their actions to form a rule. Use `Union` to combine policies that DO NOT overlap. Use `IfThenElse` to combine policies that DO overlap.*

To see why this is so, let's go back to the definition of `Seq` and `Union`. A `Seq` of policies applies each policy sequentially to each packet. A `Union`, on the other hand, makes a duplicate of the packet and sends each copy through each policy simultaneously.

A `Union` after a `Filter` is useless because `Filter` does nothing but grab a subset of packets. `Union` makes two copies of the packet, one which goes through the filter, and one which does another action. But filtering, by itself, does nothing useful to the packet, and after it's over, if there are no further actions, the packet is dropped. If your complete policy was `Filter(SwitchEq(2) & PortEq(9))`, nothing will happen to the packet, even if it arrived on switch 2, port 9.

Suppose you have two rules:

- Filter(TcpPort(80)) ¿¿ drop Drops non-encrypted HTTP traffic
- Filter(SwitchEq(1) & PortEq(2)) ¿¿ Send(1) Sends port 2 traffic out port one.

Let's say you combine these with:

Union - all packets going to port 80 from port 2 will be copied twice. The first will be dropped by the first Filter, the second will be output by the second Filter, ensuring the packet will go out anyway.

Seq - all packets going to port 80 from port 2 will be dropped by the first rule and will never make it to the second.

TODO: Explain this better.

3.3 Keeping It Stateless

Principle 3 *When you install a new switch policy, do not assume it's there before the next matching packet arrives.*

Principle 4 *Build mechanisms to automatically recreate the state if the net app dies.*

Chapter 4

Learning Switch

4.1 Design

4.2 Bootup, Switch Information

4.3 Learning Ports

4.4 Timeouts and Moves

Chapter 5

Proxy ARP

5.1 The ARP Protocol

5.2 Design

5.3 Snooping on ARP Requests and Replies

5.4 Replying

5.5 Maintaining State Across Restarts

Chapter 6

Handling Vlans

6.1 Vlan Uses

6.2 Design

6.3 Maintaining Separate Vlan Tables

6.4 Tagging and Untagging

Chapter 7

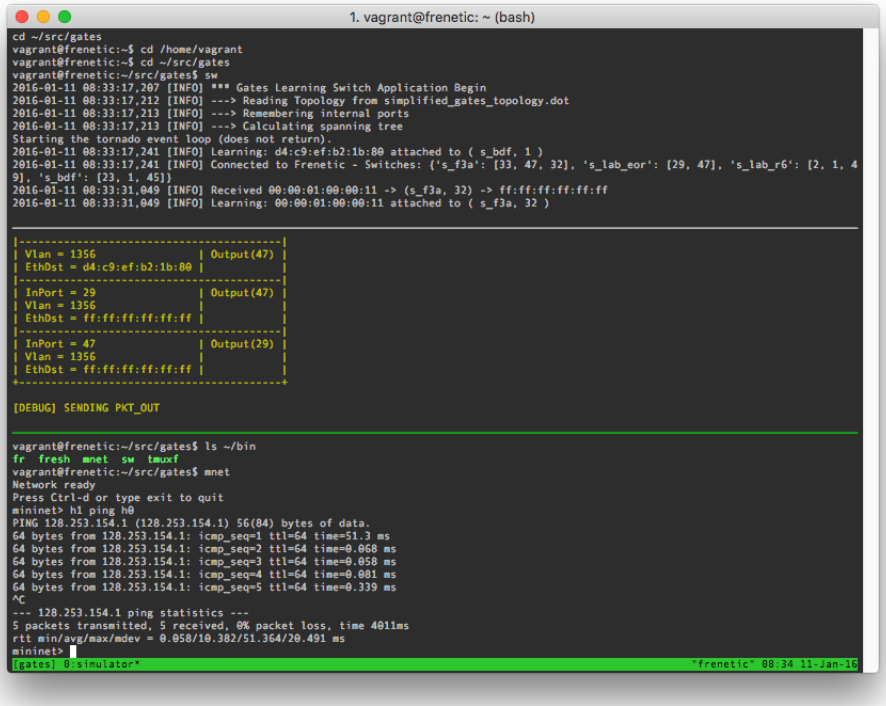
Gathering Statistics

Chapter 8

SDN Development Tools

8.1 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.

8.2 Open VSwitch Utilities

8.3 TCPDump

8.4 Mininet Network Modelling

Chapter 9

Network Address Translation

9.1 Why Do We Need NAT?

9.2 Design

Chapter 10

Spanning Tree Alternatives

10.1 What's Wrong with STP Protocols?

10.2 Design

10.3 Calculating Shortest Paths

10.4 Core/Edge Separation

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 Predicates

14.2 Policies

14.3 Event Hooks

14.4 Compiler Directives

14.5 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`.