Frenetic: A Network Programming Language

Walter Robert J. Harrison

Master's Thesis

In Partial Fulfillment of the Requirements

for the Master of Science in Engineering

Department of Computer Science

Princeton University

Co-advisers: Professor Jennifer Rexford

Professor David Walker

May 2011

Abstract

Network administrators must configure network devices to simultaneously provide several interrelated services such as routing, load balancing, traffic monitoring, and access control. Unfortunately, most interfaces for programming networks are defined at the low level of abstraction supported by the underlying hardware, leading to complicated programs with subtle bugs. We present Frenetic, a high-level language for OpenFlow networks that enables writing programs in a declarative and compositional style, with a simple "program like you see every packet" abstraction. Building on ideas from functional programming, Frenetic offers a rich pattern algebra for classifying packets into traffic streams and a suite of operators for transforming streams. The runtime system efficiently manages the low-level details of (un)installing packet-processing rules in the switches. We describe the design of Frenetic, an implementation on top of OpenFlow, and experiments and example programs that validate our design choices.

Acknowledgments

Frenetic is collaborative work with Nate Foster (Cornell University), Michael J. Freedman (Princeton University), Matthew L. Meola, Jennifer Rexford (Princeton University), and David Walker (Princeton University).

Contents

1 Introduction

Most modern networks consist of hardware and software components that are closed and proprietary. The difficulty of changing the underlying network has had a chilling effect on innovation, and forces network administrators to express complex policies through a frustratingly brittle interface. To address this problem, a number of researchers have proposed a new platform called OpenFlow to open up the software that controls the network [22]. OpenFlow defines a standard interface for installing flexible packet-handling rules in network switches. These rules are installed by a programmable *controller* that runs separately, on a stock machine [13]. OpenFlow is supported by a number of commercial Ethernet switch vendors, and several campus and backbone networks have deployed OpenFlow switches. Building on this platform, researchers have created a variety of controller applications that introduce new network functionality, like flexible access control [7, 24], Web server load balancing [14], energy-efficient networking [15], and seamless virtual-machine migration [11].

Unfortunately, while OpenFlow now makes it *possible* to implement exciting new network services, it does not make it *easy*. Programmers constantly grapple with several challenges:

Interactions between concurrent modules: Networks often perform multiple tasks, like routing, access control, and traffic monitoring. However, decoupling these tasks and implementing them independently in separate modules is effectively impossible, since packet-handling rules (un)installed by one module may interfere with overlapping rules installed by other modules.

Low-level interface to switch hardware: The OpenFlow rule algebra directly reflects the capabilities of the switch hardware (*e.g.*, bit patterns and integer priorities). Simple concepts such as set difference require multiple rules and priorities to implement correctly. Moreover, the more powerful "wildcard" rules are a limited hardware resource the programmer must manage by hand.

Two-tiered programming model: The controller only sees packets the switches do not know how to handle—in essence, application execution is split between the controller and the switches. As such, programmers must carefully avoid installing rules that hide important information from the controller.

To address these challenges, we present Frenetic, a new programming model for OpenFlow

networks. Frenetic is organized around two levels of abstraction: (1) a set of source-level operators for manipulating streams of network traffic, and (2) a run-time system that handles all of the details of installing and uninstalling low-level rules on switches. The source-level operators draw on previous work on declarative database query languages and functional reactive programming (FRP). These operators are carefully constructed to support the following key design principles:

Purely functional: The source-level abstractions are purely functional and shield programmers from the imperative nature of the underlying switches. Consequently, program modules may be written independently of one another and composed without unpredicatable effects or race conditions.

High-level, programmer-centric: Wherever possible, we first consider what *the programmer* might want to say, rather than how *the hardware* implements it. We provide intuitive, high-level primitives, even though they are not directly supported by the hardware.

See-every-packet abstraction: Programmers do not have to worry that installing packet-handling rules may prevent the controller from analyzing certain traffic. Frenetic supports the abstraction that every packet is available for analysis, side-stepping the many complexities of today's two-tiered programming model.

These principles are designed to make Frenetic programs robust, compact, and easy-to-understand, and, consequently, the Frenetic programmers writing them more productive. However, taking our "see every packet" abstraction too literally would lead to programs that process far more traffic on the controller than necessary. Instead, we give programmers a set of declarative query operators that ensure packet processing remains on the switches. The Frenetic run-time system keeps traffic in the "fast path" whenever possible, while ensuring the correct operation of all modules. In summary, this paper makes the following contributions:

Analysis of OpenFlow programming model (Section 3): We identify weaknesses of today's OpenFlow environment that modern programming-language principles can overcome.

Frenetic language (Section 4) and "subscribe" queries (Section 5): Applying ideas from the disparate fields of database query languages and functional reactive programming, we present a design for Frenetic, a language for programming OpenFlow networks.

Frenetic implementation (Section 6): We design and implement a library of high-level packet-processing operators and an efficient run-time system in Python. The run-time system handles

Integers	n	
Rules	r	$::= \langle pat, pri, t, [a_1, \ldots, a_n] \rangle$
Patterns	pat	$::= \{h_1 : n_1, \ldots, h_k : n_k\}$
Priorities	pri	$::= n$
Timeouts	t	$::= n \mid None$
Actions	a	$::= \mathsf{output}(op) \mid \mathsf{modify}(h, n)$
Header Fields	h	$::= \mathsf{in\_port} \mid \mathsf{vlan}\ \mathsf{dl\_src} \mid \mathsf{dl\_dst} \mid \mathsf{dl\_type} \mid$ $\mathsf{nw\_src} \mid \mathsf{nw\_dst} \mid \mathsf{nw\_proto} \mid \mathsf{tp\_src} \mid \mathsf{tp\_dst}$
Output Port	op	$::= n \mid \mathsf{flood} \mid \mathsf{controller}$
Packet Counts	ps	$::= n$
Byte Counts	bs	$::= n$

Figure 1: OpenFlow Syntax. Prefixes dl , nw, and tp denote data link (MAC), network (IP), and transport (TCP/UDP) respectively.

the translation from the high-level Frenetic constructs to the low-level OpenFlow rules without interaction from the programmer.

Evaluation (Section 7) and case studies (Section 8): We compare several Frenetic programs with conventional OpenFlow applications by measuring both the lines of code and the traffic handled by the controller. We also describe our experiences building four larger applications.

2 Background on OpenFlow and NOX

This section presents the key features of the OpenFlow platform. To keep the presentation simple, we have elided a few details that are not important for understanding Frenetic. Readers interested in a complete description may consult the OpenFlow specification [3].

Overview In an OpenFlow network, a centralized *controller* manages a distributed collection of *switches*. While packets flowing through the network may be processed by the centralized controller, doing so is orders of magnitude slower than processing those packets on the switches. Hence, one of the primary functions of the controller is to configure the switches so that they process the vast majority of packets and only a few packets from new or unexpected flows need to be handled on the controller.

Configuring a switch primarily involves installing entries in its *flow table*: a set of *rules* that specify how packets should be processed. A rule consists of a *pattern* that identifies a set of packets,

an integer *priority* that disambiguates rules with overlapping patterns, an optional integer *timeout* that indicates the number of seconds until the rule expires, and a list of *actions* that specifies how packets should be processed. For each rule in its flow table, the switch maintains a set of *counters* that keep track of basic statistics concerning the number and total size of packets processed.

Formally, rules are defined by the grammar in Figure 1. A pattern is a list of pairs of header fields and integer values, which are interpreted as equality constraints. For instance, the pattern $\{\mathsf{nw\_src} : \mathtt{192.168.0.100}, \mathsf{tp\_dst} : \mathtt{80}\}$ matches packets from source IP address `192.168.1.100` going to destination port `80`. We use standard notation for the values associated with header fields—*e.g.*, writing "`192.168.1.100`" instead of "`3232235876`." Any header fields not appearing in a pattern are unconstrained. We call rules with unconstrained header fields *wildcard rules*.

OpenFlow switches When a packet arrives at a switch, the switch processes the packet in three steps. First, it selects a rule from its flow table whose pattern matches the packet. If there are no matching rules, the switch sends the packet to the controller for further processing. Otherwise, if there are multiple matching rules, it picks the *exact-match* rule (*i.e.*, the rule whose pattern matches all of the header fields in the packet) if one exists, or a wildcard rule with highest priority if not. Second, it updates the byte and packet counters associated with the rule. Third, it applies each of the actions listed in the rule to the packet (or drops the packet if the list is empty). The action $\mathsf{output}(op)$ instructs the switch to forward the packet out on port op, which can either be a physical switch port n or one of the virtual ports flood or controller, where flood forwards the packet out on all physical ports (except the ingress port) and controller sends the packet to the controller. The action $\mathsf{modify}(h, n)$ instructs the switch to rewrite the header field h to n. The list of actions in a rule can contain both output and modify actions—*e.g.*, $[\mathsf{output}(2), \mathsf{output}(\mathsf{controller}), \mathsf{modify}(\mathsf{nw\_src}, \mathtt{10.0.0.1})]$ forwards packets out on switch port 2 and to the controller, and also rewrites their source IP address to `10.0.0.1`.

NOX Controller The controller manages the set of rules installed on the switches in the network by reacting to network events. Most controllers are currently based on NOX, which is a simple operating system for networks that provides some primitives for managing events as well as functions for communicating with switches [13]. NOX defines a number of events including,

- $packet\_in(s, n, p)$, triggered when switch s forwards a packet p received on physical port n to the controller,
- $stats\_in(s, xid, pat, ps, bs)$, triggered when switch s responds to a request for statistics about rules contained in pat, where xid is an identifier for the request,
- $flow\_removed(s, pat, ps, bs)$, triggered when a rule with pattern pat exceeds its timeout and is removed from s's flow table,
- $switch\_join(s)$, triggered when switch s joins the network,
- $switch\_leave(s)$, triggered when switch s leaves the network,
- $port\_change(s, n, u)$, triggered when the link attached to physical port n on switch s goes up or down, with u a boolean value representing the new status of the link,

and provides functions for sending messages to switches:

- $install(s, pat, pri, t, [a_1, \ldots, a_k])$, which installs a rule with pattern pat, priority pri, timeout t, and actions $[a_1, \ldots, a_n]$ in the flow table of switch s,
- $uninstall(s, pat)$, which removes all rules contained in pattern pat from the flow table of the switch,
- $send(s, p, a)$, which sends packet p to switch s and applies action a to it there, and
- $query\_stats(s, pat)$, which issues a request for statistics from all rules contained in pattern pat on switch s and returns a request identifier xid, which can be used to match up the asynchronous response from the switch.

The controller program defines a handler for each event, but is otherwise an arbitrary program.

Example To illustrate a simple use of OpenFlow, consider a controller program written in Python that implements a repeater. Suppose that the network has a single switch connected to a pool of internal hosts on port 1 and a wide-area network on port 2, as shown in Figure 2. The `repeater` function below installs rules on switch `s` that instruct the switch to forward packets from port 1 to port 2 and vice versa. The `switch_join` handler calls `repeater` when the switch joins the network.

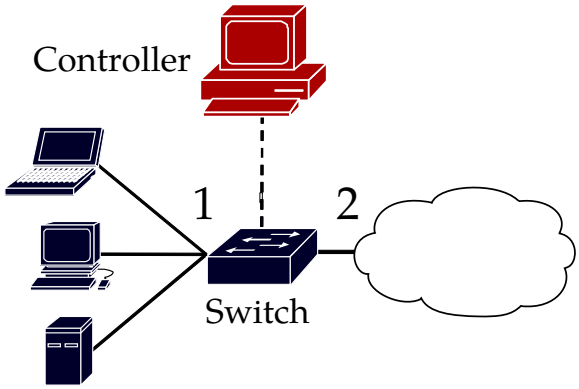


Figure 2: Simple network topology for the remaining examples

```
def repeater(s):
  pat1 = {IN_PORT:1}
  pat2 = {IN_PORT:2}
  install(s,pat1,DEFAULT,None,[output(2)])
  install(s,pat2,DEFAULT,None,[output(1)])

def switch_join(s):
  repeater(s)
```

Note that both calls to `install` use the `DEFAULT` priority level and `None` as the timeout, indicating that the rules are permanent.

3 Analysis of OpenFlow/NOX Difficulties

OpenFlow provides a standard interface for manipulating the rules installed on switches, which goes a long way toward making networks programmable. However, the programming model currently provided by NOX has several deficiencies that make it difficult to use in practice. While our analysis focuses solely on the NOX controller, other OpenFlow controllers such as Onix [17] and Beacon [1] suffer from similar issues. In this section, we describe three of the most substantial difficulties that arise when writing programs in NOX.

3.1 Interactions Between Concurrent Modules

The first issue is that NOX program pieces do not compose. Suppose that we want to extend the repeater to monitor the total number of bytes of incoming web traffic. Rather than counting the web traffic at the controller, a monitoring application could install rules for web traffic, and periodically poll the byte and packet counters associated with those rules to collect the necessary statistics:

```
def monitor(s):
  pat = {IN_PORT:2,TP_SRC:80}
  install(s, pat, DEFAULT, None, [])
  query_stats(s, pat)

def stats_in(s, xid, pat, ps, bs):
  print bs
  sleep(30)
  query_stats(s, pat)
```

The `monitor` function installs a rule that matches all incoming packets with TCP source port 80 and issues a query for the counters associated with that rule. The `stats_in` handler receives the response from the switch, prints the byte count to the console, sleeps for 30 seconds, and then issues the next query.

Ideally, we would be able to compose this program with the repeater program to obtain a program that forwards packets and monitors traffic:

```
def repeater_monitor_wrong(s):
  repeater(s)
  monitor(s)
```

Unfortunately, naively composing the two programs does *not* work due to interactions between the rules installed by each program. In particular, because the programs install overlapping rules on the switch, when a packet arrives from port 80 on the source host, the switch is free to process the packet using either rule. But using the `repeater` rule does not update the counters needed for monitoring, while using the `monitor` rule breaks the repeater program because the list of actions is empty (so the packet will be dropped).

To obtain the desired behavior, we have to manually combine the forwarding logic from the first program with the monitoring policy from the second:

```
def repeater_monitor(s):
  pat1 = {IN_PORT:1}
  pat2 = {IN_PORT:2}
  pat2web = {IN_PORT:2, TP_SRC:80}
  install(s, pat1, [output(2)], DEFAULT)
  install(s, pat2, [output(1)], DEFAULT)
  install(s, pat2web, [output(1)], HIGH)
  query_stats(s, pat2web)
```

Performing this combination is non-trivial: the `pat2web` rule needs to include the `output(1)` action from the `repeater` program, and must be installed with `HIGH` priority to resolve the overlap with the `pat2` rule. In general, composing OpenFlow programs requires careful, manual effort on the part of the programmer to preserve the semantics of the original programs. This makes it nearly impossible to factor out common pieces of functionality into reusable libraries and also prevents compositional reasoning about programs.

3.2 Low-Level Programming Interface

Another difficulty of writing NOX programs stems from the low-level nature of the programming interface, which is derived from the features of the switch hardware rather than being designed for ease of use. This makes programs unnecessarily complicated, as they must describe low-level details that do not affect the overall behavior of the program. For example, suppose that we want to extend the repeater and monitoring program to monitor all incoming web traffic *except* traffic destined for an internal server (connected to port 1) at address `10.0.0.9`. To do this, we need to "subtract" patterns, but the patterns in OpenFlow rules can only directly express positive constraints. To simulate the difference between two patterns, we have to install *two* overlapping rules on the switch, using priorities to disambiguate between them.

```
def repeater_monitor_noserver(s):
  pat1 = {IN_PORT:1}
  pat2 = {IN_PORT:2}
  pat2web = {IN_PORT:2, TP_SRC:80}
  pat2srv = {IN_PORT:2, NW_DST:10.0.0.9, TP_SRC:80}
  install(s, pat1, DEFAULT, None, [output(2)])
  install(s, pat2, DEFAULT, None, [output(1)])
  install(s, pat2web, MEDIUM, None, [output(1)])
  install(s, pat2srv, HIGH, None, [output(1)])
  query_stats(s, pat2web)
```

This program uses a separate rule to process web traffic going to the internal server—`pat2srv` matches incoming web packets going to the internal server, while `pat2web` matches all other incoming web packets. It also installs `pat2srv` at `HIGH` priority to ensure that the `pat2web` rule only processes (and counts!) packets going to hosts other than the internal server.

More generally, describing packets using the low-level patterns that OpenFlow switches support is cumbersome and error-prone. It forces programmers to use multiple rules and priorities to encode patterns that could be easily expressed using natural logical operations such as negation, difference, and union. It adds unnecessary clutter to programs that is distracting and further complicates reasoning about their behavior.

3.3 Two-Tiered System Architecture

A third challenge stems from the two-tiered architecture where a controller program manages the network by (un)installing switch-level rules. This indirection forces the programmer to specify the communication patterns between the controller and switch and deal with tricky concurrency issues such as coordinating asynchronous events. Consider extending the original repeater program to monitor the total amount of incoming traffic by destination host.

```
def repeater_monitor_hosts(s):
  pat = {IN_PORT:1}
  install(s, pat, DEFAULT, None, [output(2)])

def packet_in(s, inport, p):
  if inport == 2:
     m = dstmac(p)
     pat = {IN_PORT:2, DL_DST:m}
     install(s, pat, DEFAULT, None, [output(1)])
     query_stats(s, pat)
```

Unlike the previous examples, we cannot install all of the rules we need in advance because, in general, we will not know the address of each host *a priori*. Instead, the controller must dynamically install rules for the packets seen at run time.

The `repeater_monitor_hosts` function installs a single rule that handles all outgoing traffic. Initially, the flow table on the switch does not contain any entries for incoming traffic, so the switch sends all packets that arrive at ingress port 2 up to the controller. This causes the

`packet_in` handler to be invoked; it processes each packet by installing a rule that handles all future packets to the same host (identified by its MAC address). Note that the controller only sees one incoming packet per host—the rule processes all future traffic to that host directly on the switch.

As this example shows, NOX programs are actually implemented using *two* programs—one on the controller and another on the switch. While this design is essential for efficiency, the two-tiered architecture makes applications difficult to read and reason about, because the behavior of each program depends on the other—*e.g.*, installing/uninstalling rules on the switch changes which packets are sent up to the controller. In addition, the controller program must specify the communication patterns between the two programs and deal with subtle concurrency issues—*e.g.*, if we were to extend the example to monitor both incoming and outgoing traffic, the controller would have to issue multiple queries for the statistics for each host and synchronize the resulting callbacks.

Although OpenFlow makes it possible to manage networks using arbitrary general-purpose programs, its two-tiered architecture forces programmers to specify the asynchronous and event-driven interaction between the programs running on the controller and the switches in the network. In our experience, these details are a significant distraction and a frequent source of bugs.

4 Frenetic

Frenetic is a domain-specific language for programming OpenFlow networks, embedded in Python. The language is designed to solve the major OpenFlow/NOX programming problems outlined in the previous section. In particular, Frenetic introduces a set of *purely functional* abstractions that enable modular program development; defines *high-level*, *programmer-centric* packet-processing operators; and eliminates many of the difficulties of the two-tier programming model by introducing a *see-every-packet* programming paradigm. In this section, we explain the basics of the Frenetic language, and use a series of examples to illustrate how our design principles simplify NOX programming. However, these examples take the *see-every-packet* abstraction far too literally—they process every packet on the controller. In the next section, we will introduce additional features of Frenetic that preserve the key high-level abstractions, while also making it possible to reduce

the traffic handled by the controller to the levels seen by vanilla NOX programs.

4.1 Basic Concepts

Inspired by past work on functional reactive programming, Frenetic introduces three important datatypes for representing, transforming, and consuming streams of values.

Events represent discrete, time-varying streams of values. The type of all events carrying values of type α is written α E. To a first approximation, values of type α E can be thought of as possibly infinite lists of pairs (t, v) where t is a timestamp and v is a value of type α. Examples of primitive events available in Frenetic include `Packets`, which contains all of the packets flowing through the network; `Seconds`, which contains the number of seconds since the epoch; and `SwitchJoin` and `SwitchLeave`, which contain the identifiers of switches joining and leaving the network respectively.

Event functions transform events of one type into events of a possibly different type. The type of all event functions from α E to β E is written α β EF. Many of Frenetic's event functions are based on standard operators that have been proposed in previous work on FRP. For example, the simplest event function, `Lift(`f`)`, which is parameterized on an ordinary function f of type $\alpha \to \beta$, is an event function of type α β EF that works by applying f to each value in its input event. Frenetic also includes some novel event functions that are specifically designed for processing network traffic. For example, if g has type packet $\to$ bool then `Group(`g`)` splits the stream of packets into two streams, one for packets on which g returns true and one for packets on which g returns false. More generally, and precisely, if g has type packet $\to \alpha$, the result has type packet $(\alpha \times$ packet E$)$ EF. The elements of the resulting event are pairs of the form (v, e) where v is a value of type α and e is a nested event containing all the packets that g maps to v. We use `Group`, and its variants, to organize network traffic into streams of related packets that are processed in the same way.

A *listener* consumes an event stream and produces a side effect on the controller. The type of all listeners of events α E is written α L. Examples of listeners include `Print`, which has a polymorphic type α L and prints each value in its input to the console, and `Send`, which has type (switch $\times$ packet $\times$ action) L and sends a packet to a switch and applies an action to it there.

The rest of this section presents a series of examples that illustrate how these types fit together and demonstrate the main advantages of Frenetic's programming model over the Open-

Flow/NOX model. As in the previous section, we will assume the same network topology shown in Figure 2 For simplicity, we elide the details related to the switch joining and leaving the network and assume that a global variable `switch` is bound to its identifier.

4.2 The See-Every-Packet Abstraction

To get a taste of Frenetic, consider the web-monitoring program from the last section. Note that this program only does monitoring; we extend it with forwarding later in this section.

```
def web_monitor_ef():
  stats = ( Filter(inport_p(2) & srcport_p(80)) >>
            Lift(size) >>
            GroupByTime(30) >>
            Lift(sum) )
  return stats

def web_monitor():
  ( Packets() >>
    web_monitor_ef() >>
    Print() )
```

The top-level `web_monitor` function takes the event `Packets`, *which contains all packets flowing through the network* (!) and processes it using the `web_monitor_ef` event function. This yields an event `stats` containing the number of bytes of incoming web traffic in each 30-second window, which the program prints to the console by attaching to a `Print` listener.

The `web_monitor_ef` event function is structured as the composition of several smaller event functions—the infix operator `>>` composes event functions. `Filter` discards packets that do not match the predicate supplied as a parameter. `Lift` applies `size` to each packet in the result, yielding an event carrying packet sizes. `GroupByTime`, which has type α (α list) EF (and is derived from other Frenetic operators) divides the event of packet sizes into an event of lists containing the packet sizes in each 30-second window. The final event function, `Lift`, uses Python's built-in `sum` function to add up the packet sizes in each list, yielding an event of integers as the final result. Note that unlike the NOX program, which specified the layout of switch-level rules as well as the communication between the switch and controller (to retrieve counters from the switch), Frenetic's unified architecture makes it possible to express this program as a simple, declarative query.

4.3 High-Level Patterns

Frenetic includes a rich pattern algebra for describing sets of packets. Suppose that we want to change the monitoring program to exclude traffic to the internal server. In Frenetic, we can simply take the difference between the pattern describing incoming web traffic and the one describing traffic to the internal web server.

```
def monitor_noserver_ef():
  return( Filter((inport_p(2) & srcport_p(80)) - dstip_p("10.0.0.9")) >>
          Lift(size) >>
          GroupByTime(30) >>
          Lift(sum) )
```

The only change in this program compared to the previous one is the pattern passed to `Filter`. The "-" operator computes the difference between patterns and the run-time system takes care of the details related to implementing this pattern. Recall that crafting rules to implement the same behavior in NOX required simulating the difference using two rules at different priorities.

4.4 Compositional Semantics

Frenetic makes it easy to compose programs. Suppose that we want to extend the monitoring program from above to also behave like a repeater. In Frenetic, we just specify the forwarding rules and register them with the run-time system.

```
rules = [(switch, inport_p(1), [output(2)]),
         (switch, inport_p(2), [output(1)])]

def repeater():
  register_static_rules(rules)

def repeater_web_monitor():
   repeater()
   web_monitor()
```

The `register_static_rules` function takes a list of high-level rules (different than the low-level rules used in NOX) each containing a switch, a high-level pattern, and a list of actions, and installs them as the current forwarding policy in the Frenetic run-time. Note that the monitoring portion of the program does not need to change at all—the run-time ensures that there are no harmful interactions between the forwarding and monitoring components.

To illustrate the benefits of composition, let us carry the example a step further and extend it to monitor incoming traffic by host. Implementing this program in NOX would be difficult—we cannot run the two smaller programs side-by-side because the rules for monitoring web traffic overlap with the rules for monitoring traffic by host. We would have to rewrite both programs to ensure that the rules installed on the switch by the programs do not interfere with each other—*e.g.*, installing two rules for each host, one for web traffic and another for all other traffic. This could be made to work, but it would require a major effort from the programmer, who would need to understand the low-level implementations of both programs in full detail.

In contrast, extending the Frenetic program is simple. The following event function monitors incoming traffic by host.

```
def host_monitor_ef():
  return ( Filter(inport_p(2)) >>
           Group(dstmac_g()) >>
           RegroupByTime(60) >>
           Second(Lift(lambda l:sum(map(size,l)))) )
```

It uses `Filter` to obtain an event carrying all packets incoming on port 2, `Group` to aggregate these filtered packets into an event of pairs of destination MACs and nested events that contain all packets destined for that host, `RegroupByTime` to divide the nested event streams into an event of pairs of MACs and lists that contain all packets to that host in each 60-second window, and `Second` and `Lift` to add up the size of the packets in each window. The `RegroupByTime` event function (which like `GroupByTime` is a derived operator in Frenetic) has type $(\beta \times \alpha\ \mathsf{E})\ (\beta \times \alpha\ \mathsf{list})\ \mathsf{EF}$. It works by splitting the nested event stream into lists containing the values in each window. The `Second` event function takes an event function as an argument and applies it to the second component of each value in an event of pairs. In this example, the event function being applied is a lifted anonymous function, denoted by the Python keyword `lambda`, that returns the sum of sizes of packets in a list. Putting all of these together, we obtain an event function that transforms an event of packets into an event of pairs containing MACs and byte counts.

The top-level program `repeater_monitor_hosts` applies both stream functions to `Packets` and registers the forwarding policy with the run-time. Despite the slightly different functionality and polling intervals of the two programs, Frenetic allows these programs to be easily composed without any concerns about undesirable interactions or timing issues between them.

```
def repeater_monitor_hosts():
    repeater()
    stats1 = Packets() >> web_monitor_ef()
    stats2 = Packets() >> host_monitor_ef()
    Merge(stats1,stats2) >> Print()
```

Raising the level of abstraction frees programmers from worrying about low-level details and enables writing programs in a modular style. This represents a major advance over NOX, where programs must be written monolithically to avoid harmful interactions between the switch-level rules installed by different program pieces.

4.5 Learning Switch

So far, we have mostly focused on small examples that illustrate the main features of Frenetic. The last example in this section describes a more realistic application—an Ethernet learning switch. Learning switches provide easy plug-and-play functionality in local-area networks. When the switch receives a packet, it remembers the source host and ingress port that the packet came in on. Then, if the switch has previously received a packet from the destination host, it forwards the packet out on the port that it remembered for that host. Otherwise, it floods the packet out on all ports (other than the packet's ingress). In this way, over time, the switch learns the information needed to forward packets to each active host in the network and avoids unnecessary flooding.

Figure 3 gives the definition of a learning switch in Frenetic. Just like the other Frenetic programs we have seen, it is structured as the composition of several smaller event functions. It uses `Group` to aggregate the input event of packets by source MAC and `Regroup` to split the nested events whenever a packet from a given host appears on a different ingress port (*i.e.*, because the host has moved). This leaves an event of pairs (m, e) where m is a source MAC and e is a nested event containing packets that share the same source MAC address and ingress switch port. The `Ungroup` event function extracts the first packet from each nested event, yielding an event of pairs of MACs and packets. The `LoopPre` event function takes an initial value of type γ and an event function of type $(\alpha \times \gamma)\ (\beta \times \gamma)$ EF as an argument and produces an event function of type $\alpha\ \beta$ EF that works by looping the second component of each pair into the next iteration of the top-level event function. In this instance, it builds up a dictionary structure that associates a MAC address to a rule that forwards packets to that host (the helper `add_rule` inserts the rule into the dic-

```
# helper functions
def add_rule(((m,p),t)):
    a = forward(inport(header(p)))
    pat = dstmac_p(m)
    t[m] = (switch,pat,[a])
    return (t,t)

def complete_rules(t):
    l = t.values()
    ps = map(lambda r: r.pattern, l)
    r = (switch,reduce(diff_p, ps, true_p()),[flood()])
    l.append(r)
    return l

# main definition
def learning():
    ( Packets() >>
      Group(srcmac_g()) >>
      Regroup(lambda p1,p2: inport_r()) >>
      Ungroup(1, lambda n,p:p, None) >>
      LoopPre({}, Lift(add_rule)) >>
      Lift(complete_rules) >>
      Register() )
```

Figure 3: Simple Learning Switch

tionary). The `Lift` event function uses `complete_rules` to extract the list of rules from the dictionary and add a catch-all rule that floods packets to unknown hosts. Finally, the program registers the resulting rule list event in the Frenetic run-time. Note that unlike the previous examples, the rules generated for the learning switch are not static. The `Register` listener takes a rule list event and registers each new list as the forwarding policy in the run-time.

Frenetic includes a number of additional operators for manipulating event streams; Figure 4 lists a few of the most important operators and their types. Note that the composition operator `>>` is overloaded to work with events, event functions, and listeners.

5 Subscribe Queries

Each of the Frenetic programs in the previous section applies a user-defined event function to `Packets`, the built-in event containing every packet flowing through the network. These programs are easy to write and understand—much easier than their NOX counterparts—but implementing their semantics directly would require sending every packet to the controller, which

Events

Seconds	$\in$	int E
Packets	$\in$	packet E
SwitchJoin	$\in$	switch E
SwitchLeave	$\in$	switch E
PortChange	$\in$	(switch $\times$ int $\times$ bool) E

Event Functions

>>	$\in$	α E $\rightarrow$ α β EF $\rightarrow$ β E
Lift	$\in$	$(a \rightarrow \beta) \rightarrow \alpha$ β EF
>>	$\in$	α β EF $\rightarrow$ β γ EF $\rightarrow$ α γ EF
First	$\in$	α β EF $\rightarrow$ $(\alpha \times \gamma)$ $(\beta \times \gamma)$ EF
Second	$\in$	α β EF $\rightarrow$ $(\gamma \times \alpha)$ $(\gamma \times \beta)$ EF
Merge	$\in$	$(\alpha$ E $\times$ β E) $\rightarrow$ (α option $\times$ β option) E
LoopPre	$\in$	$(\gamma \times ((\alpha \times \gamma)$ $(\beta \times \gamma)$ EF)) $\rightarrow$ α β EF
Calm	$\in$	α α EF
Filter	$\in$	$(\alpha \rightarrow$ bool) $\rightarrow$ α α EF
Group	$\in$	$(\alpha \rightarrow \beta) \rightarrow \alpha$ $(\beta \times \alpha$ E) EF
Regroup	$\in$	$((\alpha \times \alpha) \rightarrow$ bool) $\rightarrow$ $(\beta \times \alpha$ E) $(\beta \times \alpha$ E) EF
Ungroup	$\in$	(int option $\times$ $(\gamma \times \alpha \rightarrow \gamma) \times \gamma) \rightarrow (\beta \times \alpha$ E) $(\beta \times \gamma)$ EF

Listeners

>>	$\in$	α E $\rightarrow$ α L $\rightarrow$ unit
Print	$\in$	α L
Register	$\in$	(packet $\times$ action list) list L
Send	$\in$	(switch $\times$ packet $\times$ action) L

Figure 4: Core Frenetic Operators

would lead to unacceptable performance.

Frenetic sidesteps this issue by providing programmers with a simple query language that allows them to succinctly express the packets and statistics needed in their programs. The runtime takes these queries and generates events that contain the appropriate data, using rules on the switch to move packet processing into the network and off of the controller.

Frenetic queries are expressed using orthogonal constructs for *filtering* using high-level patterns, *grouping* by one or more header fields, *splitting* by time or whenever a header field changes value, *aggregating* by number or size of packets, and *limiting* the number of values returned, for a given *interval* or *window* of time. The syntax of Frenetic queries is given in Figure 5. Each of the top-level constructs are optional, except for the Select, which identifies the type of values returned by the query—actual packets, byte counts, or packet counts. The infix operator $\star$ combines query operators. As an example, the following query generates an event that may be used in the learning switch:

```
Select(packets) *
GroupBy([srcmac]) *
SplitWhen([inport]) *
Limit(1)
```

It groups packets (using `Select`) by source MAC (using `GroupBy`), splits each group when the ingress port changes (using `SplitWhen`), and limits the number of packets in each group to one (using `Limit`). The event generated by this query contains pairs (m, e), where m is a MAC address and e is an event carrying the first packet sent from that host. We can use this event to rewrite the learning switch as follows:

```
def learning():
  ( Select(packets) *
    GroupBy([srcmac]) *
    SplitWhen([inport]) *
    Limit(1) >>
    Ungroup(1,lambda n,p:p,None) >>
    LoopPre({}, Lift(add_rule)) >>
    Lift(complete_rules) >>
    Register() )
```

In this program, the grouping and regrouping of packets is done using a query instead of an event function.

This revised program makes it easier for the run-time to determine which packets need to be sent up to the controller and which ones can be processed using rules on the switch. It also helps the programmer predict how their program will perform—in general, by using a subscribe query,

Queries	q	::= Select(a) * Where(qp) * GroupBy($[qh_1, \ldots, qh_k]$) * SplitWhen($[qh_1, \ldots, qh_k]$) * Every(n) * Limit(n)
Aggregates	a	::= packets | bytes | counts
Headers	qh	::= inport | srcmac | dstmac | ethtype | vlan | srcip | dstip | protocol | srcport | dstport
Patterns	qp	::= true_p() | qh_p(n) | qp & qp | qp _ qp | qp − qp | ~ qp

Figure 5: Frenetic query syntax

the run-time will move as much processing from the controller to the switches as possible. In this case, since the learning switch only needs a single packet from each host (as long as that host does not move to a different port on the switch), the run-time will indeed install switch-level rules that forward all subsequent traffic at the switch without having to send it to the contoller.

Queries can also subscribe to streams of traffic statistics. For example, the following query looks only at web traffic, groups by destination MAC, and aggregates the number of bytes every 60 seconds:

```
Select(bytes) *
Where(srcport_p(80)) *
GroupBy([dstmac]) *
Every(60)
```

Queries such as this can be used to implement many monitoring applications. The run-time can implement them efficiently by polling the counters associated with rules on the switch.

Subscribing to queries is fully compositional—a program can subscribe to multiple, overlapping events without worrying about harmful low-level interactions between the switch-level rules used to implement them. In addition, the policy for forwarding packets registered in the run-time does not affect the values sent to the subscribers. In contrast, in OpenFlow/NOX installing a rule can prevent future packets from being sent to the controller.

6 Frenetic Implementation

Frenetic provides high-level programming abstractions that free programmers from reasoning about many low-level details involving the underlying switch hardware. However, the need to deal with these details does not disappear just because the language raises the level of abstraction. The rubber meets the road in the implementation, which is described in this section.

We have implemented a complete working prototype of Frenetic as an embedded combinator library in Python. Figure 6 depicts its architecture, which consists of three main pieces: an implementation of the language itself, a run-time system, and NOX. The use of NOX is convenient but not essential—we borrow its OpenFlow API but could also use a different back-end.

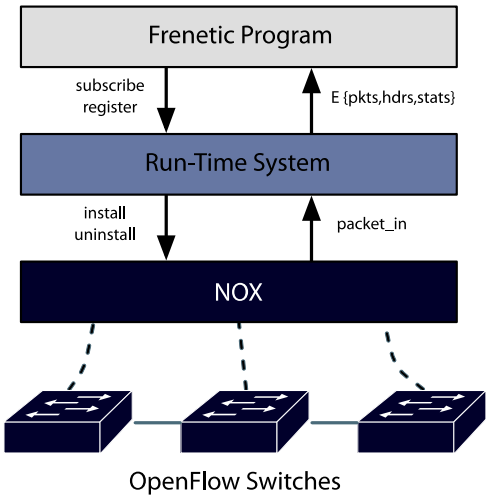


Figure 6: Frenetic architecture

6.1 The Run-time System

The core piece of the implementation is the run-time system, which sits between the high-level Frenetic program and NOX. The run-time system manages all of the bookkeeping related to installing and uninstalling rules on switches. It also generates the necessary communication patterns between switches and the controller.

6.1.1 Enforcing Language Abstractions

The Frenetic run-time is responsible for actually enforcing the high-level language features using the primitive OpenFlow rules and NOX API. Specifically, the run-time system must enforce the see-every-packet abstraction and compositionality between Frenetic programs. While the Subscribe queries defined in Section 5 ensure that a particular Frenetic program *can* be implemented on an OpenFlow enabled switch, these queries do not prescribe a particular implementation strategy for enforcing the above features. The primitive match and forward operations provided by the OpenFlow API will be insufficient, in general, to ensure that the high-level language features are enforced. Therefore, *some* degree of involvement by the controller will be required to enforce these features, but an implementation that requires *every* packet to be processed at the controller is unacceptable.

Using the information provided to the run-time by the Frenetic programs, we can define a

clear compositional semantics for programs and ensure that every program can still make use of the see-every-packet abstraction. For each program, a set of *packet subscribers* and *statistics subscribers* are defined by the subscribe queries. The subscribers articulate to the run-time the set of all packets and statistics required for a particular program. By taking the union of all suscribers over all programs, the run-time knows all packets that must be delivered to the controller and all statistics that must be collected. The run-time system can now enforce the language guarantees by ensuring that no rule is installed in a switch that would prevent any packet subscriber from receiving its matching packets at the controller. For statistics subscribers, the run-time system can begin to push work away from the controller by leveraging the built-in counters at OpenFlow switches. Therefore the run-time should install rules in the switches to collect statistics for the desired traffic provided that the rules do not interfere with any other packet subscriber. Traffic that is not a member of either of these two sets can safely be forwarded at the switches without any involvement by the controller and without violating our language guarantees. Unlike Frenetic programmers who can focus solely on *what* their programs do, we must now consider *how* we install rules in the network switches based on these guidelines.

6.1.2 Implementation Strategy

Typically, network programs are written in either a *proactive* or *reactive* manner. In the former case, a switch is populated with a given set of rules in advance and in the latter case, a switch starts with an empty flow table, consults the controller on any packet for which it has no matching flow table entry, and populates its flow table based on the controller's response. When populating these flow tables, network programmers can install either *wildcard* or *microflow* rules. Microflow rules specify an exact match of all OpenFlow supported header fields while wildcard rules leave some subset of header fields unconstrained to match any value. Since enumerating a set of switch-level rules in advance for all possible microflow combinations is difficult and space-inefficent, proactive approaches typically make use of wildcard rules. Reactive network programs often employ microflow style rules in order to make forwarding decisions at the finest level of granularity supported by OpenFlow. While these conventions pervade many OpenFlow applications in the literature, there is no restriction to adhere to them. For example, one could imagine a reactive program that installs wildcard rules.

While using proactive rules would allow us to keep more traffic in the high-speed dataplane and, consequently, provide better network throughput, we focus on a reactive, microflow based run-time for the initial prototype. This approach follows the implementation of Ethane [7] and many other OpenFlow-based applications [11, 14], and is well-suited for dynamic settings. Moreover, microflow rules can use the plentiful conventional memory (*e.g.*, SRAM) many switches provide for exact-match rules, as opposed to the small, expensive, power-hungry Ternary Content Addressable Memories (TCAMs) needed to support wildcard matches. Still, wildcard rules are more concise and well-suited for static settings. We plan to develop a more proactive, priority-based wildcard approach as part of Frenetic's run-time in the future. Longer term, we plan to extend the run-time to choose *adaptively* between exact-match and wildcard rules, depending on the capabilities of the individual switches in the network.

6.1.3 Reactive, Microflow Run-time Architecture

Currently, our implementation translates the high-level forwarding policy installed in the run-time into microflow rules at the switch. To accomplish this translation without violating the language guarantees as described above, the run-time maintains several global data structures:

- *rules*, a set of high-level rules that describe the current packet-forwarding policy,
- *flows*, a set of low-level rules that are currently installed on the switches in the network, and
- *subscribers*, a set of tuples of the form (q, e, cs, rs) where q is the query that defines the subscriber, e is the event for that subscriber, cs tracks byte and packet counts, and rs is a set of identifiers for outstanding requests for statistics,

The run-time can be partitioned roughly into three parts: the data structures described above, a packet-processing pipeline and a statistics monitoring facility. Figure 7 shows the architecture of the run-time system which we now describe in detail. At the start of the execution of a program, the flow table of each switch in the network is empty, so all packets are sent up to the controller and passed to the `packet_in` handler. Upon receiving a packet, the run-time system iterates through the set of packet subscribers and propagates the packet to each subscriber whose defining query depends on being provided with this packet. Next, the run-time consults the forwarding policy

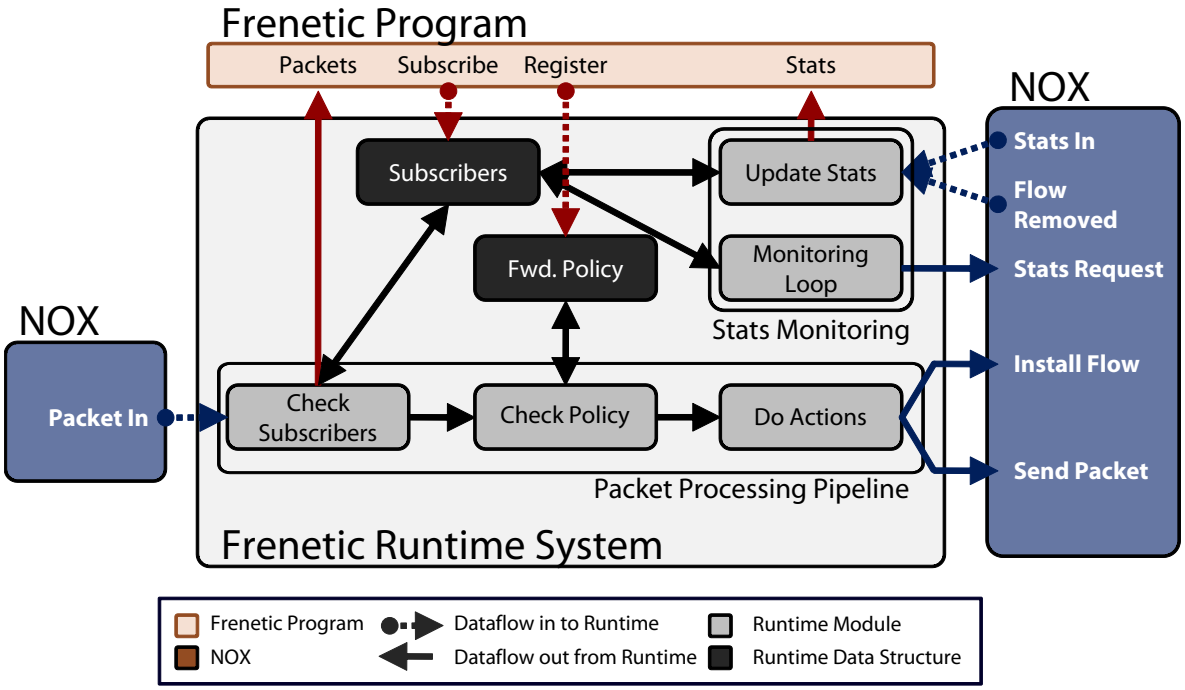


Figure 7: Frenetic Run-time System Architecture.

and collects the list of actions specified from all rules that the packet matches. Finally, it processes the packet in one of two ways: if the packet matched no registered packet subscribers, then the run-time installs a *microflow* rule that processes future packets with the same header fields on the switch. Alternatively, if the packet did match any registered packet subscriber, then the run-time sends the packet back to the switch and applies the actions there, but does not install a rule, as doing so would prevent future packets from being sent to the controller. In the case of a packet subscriber containing a Limit(n) clause in the defining query, once the n packets have been received, this query is no longer considered an active packet subscriber. In effect, this strategy dynamically unfolds the forwarding policy expressed in the high-level rules into switch-level rules, moving processing off the controller and onto switches in a way that does not interfere with any subscriber.

The run-time uses a different strategy to implement statistics subscribers, using the byte and packet counters maintained by the switches to calculate the values. The run-time system executes a loop that waits until the interval for a statistics subscriber has elapsed. At that point, it traverses

```
function packet_in(packet, inport)
  isSubscribed := false
  actions := []
  for (query, event, counters, requests) ∈ subs do
    if query.matches(packet.header) then
      event.push(packet)
      isSubscribed := true
  for rule ∈ rules do
    if (rule.pattern).matches(packet.header) then
      actions.append(rule.actions)
  if isSubscribed then
    send_packet(packet, actions)
  else
    install(packet.header, DEFAULT, None, actions)
    flows.add(packet.header)

function stats_in(xid, ps, bs)
  for (query, event, counters, requests) ∈ subs do
    if requests.contains(xid) then
      counters.add(ps, bs)
      requests.remove(xid)
      if requests.is_empty() then
        event.push(counters)

function stats_loop()
  while true do
    for (query, event, counters, requests) ∈ subs
    do
      if query.ready() then
        counters.reset()
        for pattern ∈ flows do
          if query.matches(pattern) then
            xid := stats_request(pattern)
            requests.add(xid)
    sleep(1)
```

Figure 8: Frenetic run-time system handlers

the *flows* set and issues a request for the byte and packet counters from each switch-level rule whose pattern matches the query, adding the request identifier to the set of outstanding requests maintained for this subscriber in *subscribers*. The `stats_in` handler receives the asynchronous replies to these requests, adds the byte and packet counters to the counters maintained for the subscriber in *subscribers*, and removes the request identifier from the set of outstanding requests. When the set of outstanding requests becomes empty, the run-time pushes the counters, which now contain the correct statistics, onto the appropriate subscriber's event stream.

Figure 8 gives pseudocode for the NOX handlers used in the Frenetic run-time system. These algorithms describe the basic behavior of the run-time, but elide some additional complications and details that the actual implementation has to deal with such as maintaining accurate counters across rule-set changes and spurious packets sent to the controller due to race conditions between the receipt of a message to install a rule and the arrival of the packet at the switch.

6.2 Combinator Library

The other major piece of the Frenetic implementation is the library of FRP operators themselves. This library defines representations for events, event functions, and listeners, as well as each of the primitives in Frenetic including `Lift`, `Filter`, `LoopPre`, etc. Unlike classic FRP implementa-

tions, which support both continuous streams called *behaviors* and discrete streams called *events*, Frenetic focuses almost exclusively on discrete streams. This means that the pull-based strategy used in most previous FRP implementations, which is optimized for behaviors, is not a good fit for Frenetic. Instead, our FRP library uses a push-based strategy to propagate values from input to output streams.

7 Experiments

In Section 5 we described Frenetic programs that made use of subscribe queries to communicate to the run-time what data the programs required from the network. In Section 6 we described the current implementation of the run-time system that can enforce the language guarantees while keeping traffic in the high-speed dataplane whenever possible. Now we seek to evaluate our design and current protype by comparing Frenetic programs with programs written with the pure NOX API only. We implemented several simple applications in Frenetic and compared them against equivalent NOX programs on three metrics: lines of code, traffic to controller, and aggregate traffic. The *lines of code* metric gives a measure of the complexity of each program, as well as the savings from code reuse when modules are composed. The *controller traffic* measures the total amount of communication between the switch and controller, which quantifies the overhead of managing switch-level rules using a run-time system. Finally, the *aggregate traffic* metric measures the total amount of traffic on every link in the network.

Setup We ran the experiments using the Mininet virtualization environment [18] on a Linux host with a 2.4GHz Intel Core2 Duo processor and 2GB of RAM. Although Mininet cannot provide performance fidelity, it does give accurate measurements of the volume of traffic flowing through the network. In each microbenchmark, we use a very simple virtual network topology consisting of a single network switch, four network hosts, and in the web statistics benchmark, a single web server.

Microbenchmarks. We compared the performance of Frenetic against NOX using microbenchmarks consisting of some monitoring component and some forwarding component:

- **All-Pairs Connectivity:** each host sends and receives ICMP (ping) packets to/from all other hosts. This benchmark tests whether the forwarding policy establishes basic connectivity. In this base case, there actually is no monitoring component and the microbenchmark merely executes the underlying forwarding policy.

- **Web Statistics:** each host generates a single request to a web server and the controller monitors the aggregate HTTP traffic every five seconds. This tests the performance of simple monitoring—a common network administration task.

- **Heavy Hitters:** each host sends and receives ICMP packets to/from a variety of other hosts in the network. The controller collects per-host statistics and reports the top-k traffic sources. This illustrates a more sophisticated monitoring application.

Note that none of these microbenchmarks specify the underlying policy used to forward packets in the network. We ran each microbenchmark using several different policies:

- **Hub:** The hub (HUB) policy floods packets received on one port out on all other ports, except the port the packet arrived on.

- **Learning Switch:** The learning switch (LSW) policy dynamically learns the association between hosts and ports as it sees traffic. It floods packets to unknown destinations but outputs packets to known hosts on the port the host is connected to.

- **Loop-Free Learning Switch:** The loop-free learning switch (LFL) learns the host-port mapping and also monitors the network topology and calculates a minimum spanning tree. This avoids forwarding loops when flooding packets.

We measured lines of code (up to 80 characters of properly-indented Python, excluding non-essential whitespace) as well as the total amount of controller traffic—control messages, switch responses, and whole packets sent to the controller on flow-table misses.

Results The results of our experiments are given in Table 1. They demonstrate a few key points. First, on these benchmarks, Frenetic performs comparably with hand-written NOX programs despite being implemented using a run-time system. Second, Frenetic provides substantial code

		Connectivity			Heavy Hitters			Web Stats		
		HUB	*LSW*	*LFL*	*HUB*	*LSW*	*LFL*	*HUB*	*LSW*	*LFL*
NOX	Lines of Code	20	55	75	110	198		104	135	
	Controller (kB)	8.8	9.7	22.2	8.4	10.7	$\star$	7.5	7.1	$\star$
	Aggregate (kB)	65.3	38.5	56.9	145.1	78.4		31.8	17.0	
Frenetic	Lines of Code	6	30	58	29	53	81	13	37	65
	Controller (kB)	8.8	11.8	12.4	10.8	11.8	12.3	5.8	6.8	7.4
	Aggregate (kB)	65.3	40.6	41.2	149.3	80.4	86.3	32.4	18.3	18.8

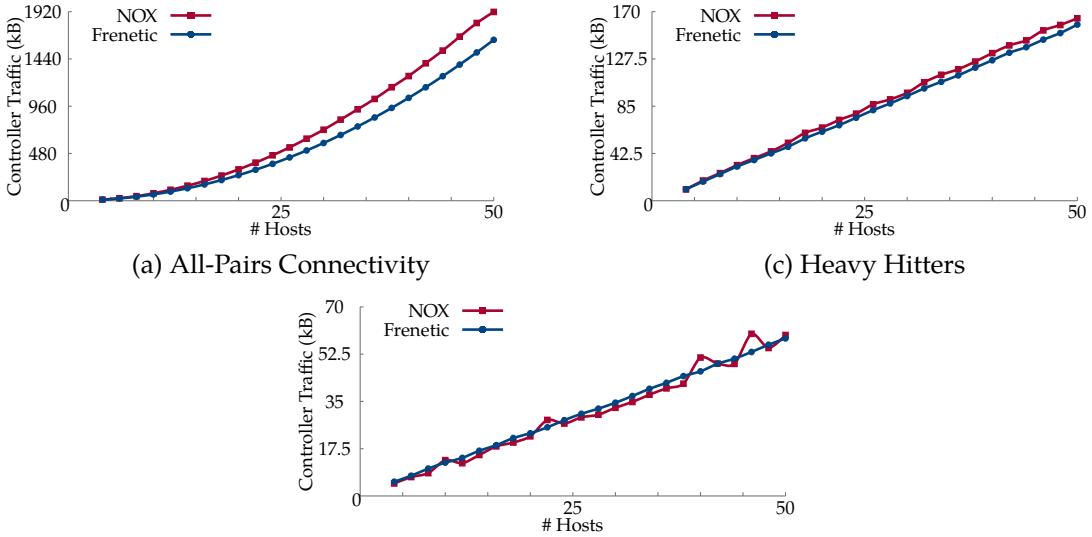
Table 1: Experimental results.

savings to the network programmer. In particular, Frenetic's compositional semantics allowed us to easily compose the monitoring modules with each of the forwarding policies—the size of each composition is exactly the sum of the sizes of the inputs (the monitoring queries for Web Stats and Heavy Hitters are 23 and 7 lines, respectively)—unlike the NOX programs, which had to be manually refactored to correctly implement each version of the microbenchmark.[1] Finally, the aggregate traffic statistics for LFL demonstrate that by using Frenetic, programmers can write sophisticated network programs that actually consume *less* network capacity than hand-written NOX programs. The reason for this difference is that the Frenetic LFL dynamically reacts to network events while the NOX version uses periodic polling to discover the network topology, which produces more total traffic on the network.

These microbenchmarks demonstrate that Frenetic's run-time system achieves adequate performance in some common scenarios. Of course, they are far from comprehensive. There are certainly many situations where Frenetic's run-time system does not perform as well as hand-written NOX programs—*e.g.*, when the optimal implementation of the forwarding policy uses wildcard rules. We plan to investigate other strategies for implementing the run-time system in the future.

Scalability Experiments. For each microbenchmark, we also conducted a scalability experiment to evaluate whether Frenetic programs would continue performing comparably to NOX programs as the number of hosts in the network grows. In each experiment, we used a single switch running

[1] In fact, refactoring the benchmarks to use the loop-free learning switch was sufficiently difficult that we did not complete it, despite the fact that NOX provides a topology module and we had already implemented hub and learning switch versions of the benchmarks.



(a) All-Pairs Connectivity

(c) Heavy Hitters

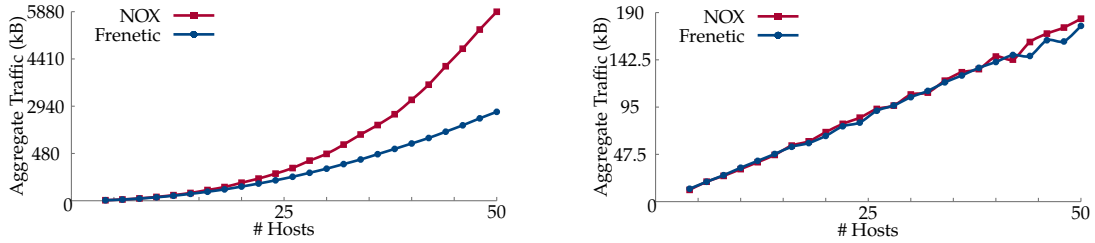
(b) Web Statistics

Figure 9: Scalability results.

the learning switch forwarding policy, but scaled the number of hosts up from 4 to 50. The results in Figure 9 confirm that Frenetic performance *scales* comparably—and in many cases better than—NOX. We hypothesize a simple reason for this difference: a common NOX idiom, which we used in our implementations of the NOX benchmarks, is to install rules with timeouts. This ensures that rules "self-destruct" without the programmer having to perform extra bookkeeping to remember all of the installed rules. However, such timeouts result in additional packets being sent to the controller, both in *flow_removed* messages and for subsequent flow setups. In contrast, Frenetic's run-time system reacts to changes in the forwarding policy and manages the set of installed rules automatically, obviating the need for such timeouts.

Additionally, we explored how increasing the number of hosts affects the aggregate traffic on the network. Figure 10 shows that while some microbenchmarks saw signficant savings over the pure NOX versions, others saw little to no change. However, we again see that as the network grows in number of hosts, Frenetic programs do not scale *worse* than NOX versions.

Controller Throughput. We also ran an experiment to measure the performance of the run-time system itself. Because the run-time processes the first packet in every flow, the overall throughput of the network is roughly proportional to the maximum throughput of the controller. We mea-



(a) All-Pairs Connectivity

(b) Web Statistics

Figure 10: Aggregate Traffic scalability results.

sured the throughput of the controller in terms of maximum *flow modifications per second (fmods/sec)* using the Cbench tool [5] from the OFlops suite. CBench measures the maximum number of instructions the controller can issue to switches in response to packets received. We compared NOX to the current, unoptimized Frenetic prototype both running a repeater hub. Frenetic performs at 85% of the peak throughput obtained using NOX. In the future, we expect we will be able to close this gap by optimizing the run-time. However, this result suggests that our current, unoptimized prototype already provides the benefits of a high-level language at a reasonable cost.

8 Case Studies

This section describes four more substantial network applications we have developed using Frenetic: the first two are applications that implement conventional network functionality while the latter two are more novel applications that make use of Frenetic's high level language features.

8.1 Centralized ARP Server

The Address Resolution Protocol (ARP) determines the network adapter address (MAC) of a given IP address in a broadcast local-area network (LAN). This protocol is integral to the proper operation of modern local-area networks. However, the broadcast nature of the protcol limits the size to which local-area networks can grow as the amount of broadcast traffic overwhelms the network [12]. A centralized ARP server can help mitigate the scalability problems by storing these bindings for clients and responding to requests itself thereby suppressing some broadcast traffic.

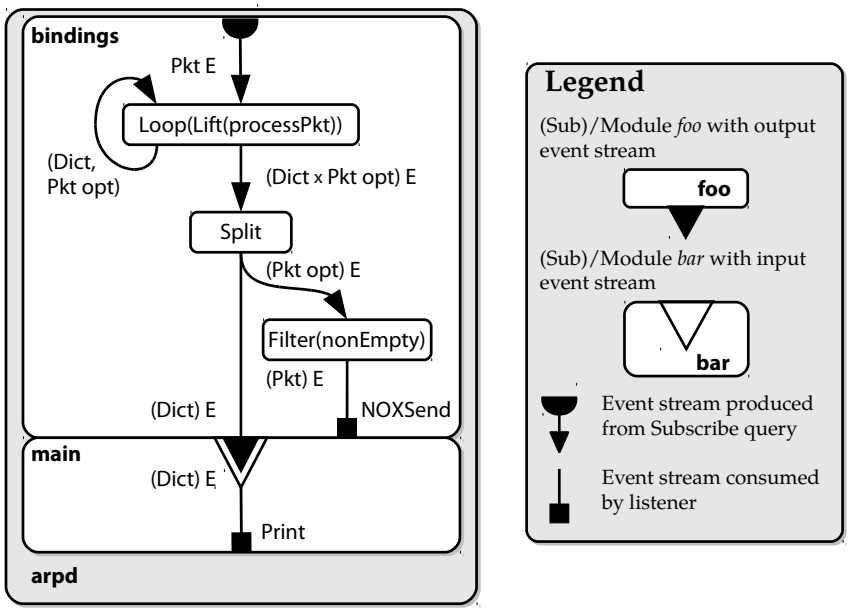


Figure 11: ARPd Functional Description. The confluence and divergence of arrows on a module in the diagram imply Merge and Split operations, respectively.

ARP Operation Clients resolve IP addresses by sending broadcast *requests* on the LAN and listening for *responses* indicating which physical address on the network possesses the given IP address. Once resolved, clients maintain a cache that maps IP addresses to MAC addresses and refresh this data after a given timeout expires by simply reissuing the query.

Module Description Figure 11 shows the functional description of the arpd module. In general, Frenetic modules consist of relatively few primary event streams or sub-modules which expose some functionality that might be desirable for other modules to have access. This application is relatively simple and consists of a single public event stream bindings and a main function to run the application in stand-alone mode using this event stream. Running an application in stand-alone mode means that the application's public event streams will not be used in some other module and the application's main module should describe (or call another module to describe) all desired network functionality.

The bindings module registers a query for all ARP traffic in the network, the results of which are subsequently sent to the processPacket lifted function. This function parses the ARP packet and determines whether it is a response or a request. This function also maintains a mapping of

the current IP to MAC bindings in the network by looping its output value back on its input using LoopPre. This function broadcasts requests out all switch ports when it does not know the MAC of the requested address, but returns the stored mapping otherwise. This application reduces the broadcast ARP traffic on the network and allows other Frenetic applications to make use of the current bindings as a publicly available event stream.

The main module simply consumes the output of bindings and displays all known IP, MAC pairs on the console. The main module also calls (not depicted above) the learning_switch.main submodule to serve as a forwarding policy when the application is run in stand-alone mode. Thus, a programmer could use the output of bindings as an input to another distinct module (as demonstrated later in the memcached application), or run the arpd module as a standalone application.

Limitations Currently, this module does not handle network dynamics and assumes static hosts. However, this more complicated program would be a straightforward extension of this program and easily implemented using the Frenetic language.

8.2 Dynamic Host Configuration

End-host configuration through the Dynamic Host Configuration Protocol (DHCP) has become a staple of modern network management because users demand the simplicity of "plug-and-play" connectivity. However, the policies that network managers use to determine how hosts are configured have become increasingly complex in recent years. While several commercial and free software solutions solve this problem, creating a Frenetic program that provides this functionality allows network managers to compose any other arbitrary network logic with the module to create custom dynamic configurations not possible with existing software solutions. Additionally, network operators can use the output (the set of current leases) of the DHCP module to create entirely new functionality unrelated to end-host configuration.

The DHCP Protocol Clients send DHCP *requests* to a well-known service on a broadcast address asking for configuration data. If available, a DHCP server answers this request with an *offer* of host configuration data and the client then confirms receipt and acceptance of the configuration. A DHCP server offers clients an IP address, subnet mask, default gateway, and other optional

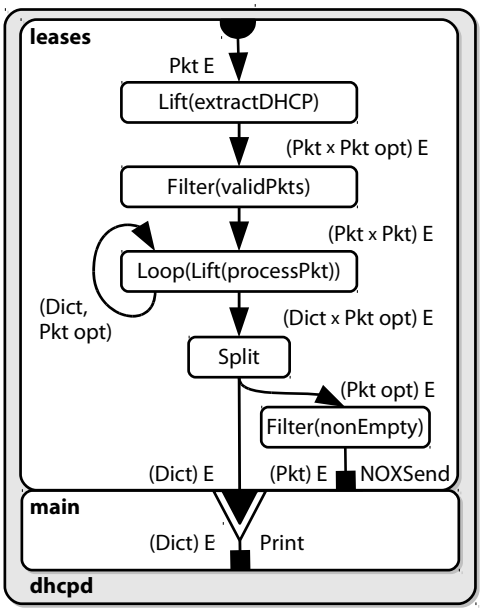


Figure 12: DHCPd Functional Description.

configuration instructions. The DHCP server draws this configuration from an operator-defined client *pool* and must maintain state about which addresses in the pool have already been *leased* to clients to avoid duplicating assignments.

Module Description Figure 12 shows the functional description of the dhcpd module which consists of two public sub-modules: leases and main. The leases event stream yields a Dict E containing the set of current leases indexed by client MAC address. The main sub-module simply takes as input the output from the leases module and prints that stream to the console.

Within the leases module, we start by subscribing to a query for DHCP requests which produces a stream of type packet E. We then parse packets of the event stream through two lifted functions to extract the DHCP packet from the raw frame and check for validity of the request. Valid DHCP requests are then passed to the processPacket function which maintains state about leases in the network and outputs the current leases indexed by client MAC address as well as an optional packet object for sending the response to the client. This stream is then split and the lease dictionary is returned as the output of this sub-module and the packet object is consumed by the NOXSend L which outputs the packet out of a switch to the client.

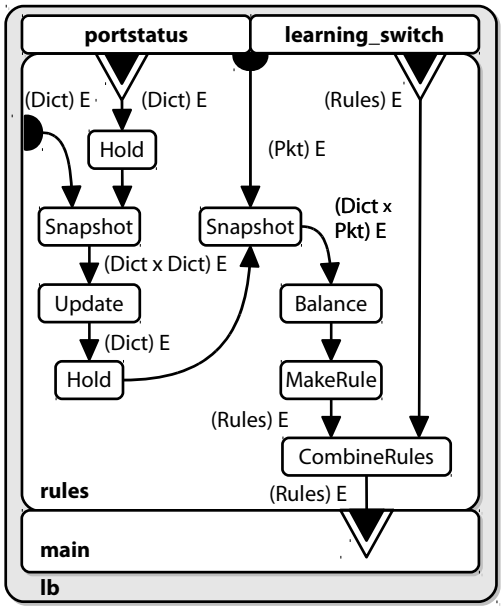


Figure 13: Parameterized Loadbalancer Functional Description.

Limitations Currently, the DHCP module only responds to a subset of the entire DHCP protocol, and does not support many DHCP/BOOTP options but this is not a limitation inherent to the design of the module, but only in the implementation of the additional functionality.

8.3 Parameterized Load Balancer

Load balancing is a common network administration task wherein all client requests for a single service are balanced across some set of replicas (typically a web server) according to a given statistic — *e.g.*, outgoing load, incoming load, server utilization, etc. The statistic and function (such as simple round-robin or minimal load) on which this balancing occurs can change over time and a generalized load balancer that can perform a variety of balancing strategies would reduce the time required to change strategies from hours to minutes.

Module Description Figure 13 shows the functional description of the module. The load balancer first registers two queries with the run-time: one query that listens for new client requests received, and one that queries the load on some configurable set of replica ports on a switch. The module combines the load query with the input from portstatus in the Update event function which

outputs an updated set of available replica ports and the current load on them. The Balance event function takes a single parameter f which is a configurable balancing function that returns the next output port for the appropriate replica that should be used. This function is then applied to each of the packets yielded from the query for client requests and the replica port loads provided by Update. The MakeRule event function then transforms the packet and output port generated by Balance into an updated network forwarding policy which is subsequently registered with the run-time system.

Limitations This module is currently written only for a single switch and only accounts for fail-stop network dynamics, which assumes that the only type of failure for a replica is a hard shut-down.

8.4 Routing Requests to Memcached Servers

Memcached [2] is a distributed key-value store used by many online services to cache data objects in memory. In a typical usage scenario, a collection of memcached servers handles *get* and *set* requests from clients, with the keyspace partitioned evenly across the servers. The current version of the Memcached application configures a *static* set of servers at each client, a restriction that prevents services from automatically adapting to new servers becoming available or existing servers failing. Ideally, a memcached cluster would automatically adjust the partitioning of the keyspace to account for server load and failure.

Traditional Memcached Operation Figure 14 shows an example topology in which memcached might be deployed. A back-end set of servers runs the memcached service on a particular port. In order to initialize, the memcached client must specify the list of servers to which it is connecting by IP address. Once initialized, the client routes *get* and *set* requests for a key to the appropriate server in this list according to some partitioning strategy. However, each time the server set changes, each client must reinitialize with the new set of servers. Memcached is a well-known application heavily used by online-service providers and a network solution that integrated off-the-shelf memcached clients and servers but could handle server churn would be a welcome addition to any online service provider.

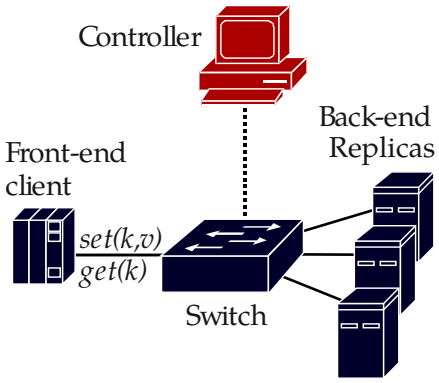


Figure 14: Memcached Application Topology

Dynamically Adjusting the Server Set We developed a novel solution to this problem in Frenetic that adapts dynamically to server churn by introducing a layer of indirection between the clients and servers. When configuring the memcached client, we specify a list of *virtual addresses (vid)* as opposed to the actual *physical addresses (pid)* of the servers. A centralized network controller dynamically assigns the set of *vid*s to the currently available subset of backend servers (*pid*s). The controller then subsequently installs OpenFlow rules in a switch that rewrite memcached requests to a particular *vid* with the actual *pid* of the server. When the set of available servers changes, the controller dynamically remaps the set of *vid*s onto the new set of *pid*s and changes the OpenFlow rules accordingly. Since this solution works entirely in the network and the list of *vid*s to which the client connects never changes, completely unmodified clients and servers can be used.

Module Description Figure 15 depicts the high-level structure of the Frenetic Memcached application. The dhcpd module at the top left of the figure was described in the previous section and provides the stream of type Dict E containing the current leases as input to memcache.main. This dictionary contains tuples of the form form (m, p, a), where m is a MAC address, p is a physical switch port, and a is the IP address leased to m.

The module portstatus at the top center monitors PortChange E events and produces an event stream with sets of active physical ports. This event stream is merged with the DHCP event stream and the result is supplied to the MakeState function. This function reconciles the set of known

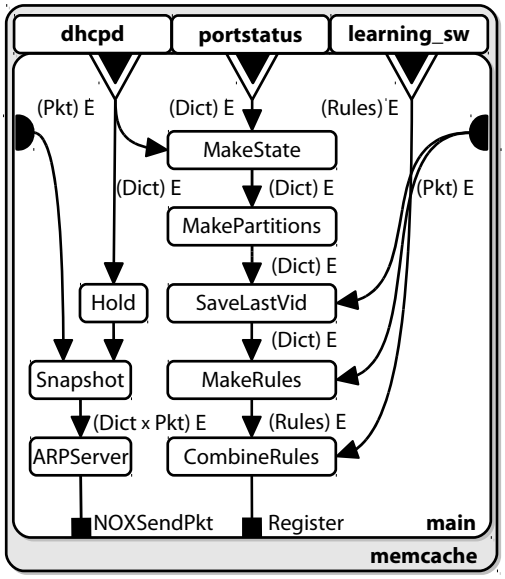


Figure 15: Memcached Functional Description. The confluence and divergence of arrows on a module in the diagram imply Merge and Split operations, respectively.

servers (from dhcp_server) with the set of active ports (from portstatus) and generates a single data structure that contains the current network state and set of available servers.

The current network state is then provided to the MakePartitions function which creates and adjusts the mapping between virtual and physical addresses. This module must avoid unnecessary disruption to the keyspace when the mapping of *vid*s to *pid*s changes. Therefore, this module must maintain state about the current partitioning to avoid remapping *vid*s unaffected by a particular network event.

The module MakeRules then converts the event with the current partitioning into a set of rules that will rewrite *vid*s to *pid*s and vice versa. Rewriting *vid*s to *pid*s is straightforward since there is a many-to-one mapping. However since each *pid* could (and will) have multiple *vid*s mapped to it, there is no way to install a set of OpenFlow level rules that matches only on the source *pid* and correctly rewrites packets for each different *vid* mapped to that *pid*. Consequently, SaveLastVid remembers the last *vid* requested for each *pid* and only installs that return-path rule.

The CombineRules then merges the stream of modification rules with the stream of forwarding rules provided from the learning_switch.rules module and registers this resultant rule set with the

run-time through the Register listener.

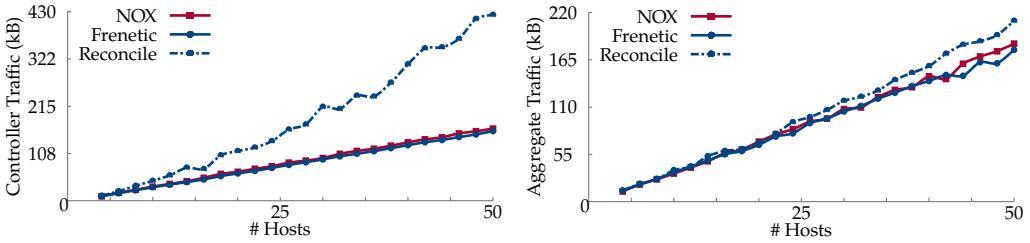
Limitations This module currently assumes a fail-stop failure model, however, our design permits a straightforward extension by adding a heartbeat mechanism to cope with soft failures. For instance, a module such as CheckServers could be interposed between MakeState and MakePartitions that sends heartbeat messages and subscribes to heartbeat responses. Because Frenetic supports a compositional style of programming, we believe this extension should be easy to integrate into our existing application.

9 Related Work

Frenetic's event functions are modeled after functional reactive languages such as Yampa and others [25, 10, 26, 23]. Its push-based implementation is based on FrTime [8] and is similar to self-adjusting computation [6]. The key differences between Frenetic and these previous languages are in the application domain and in the design of our query language and run-time system, which uses the capabilities of switches to avoid sending packets to the controller. The Flask [21] language applies FRP in a staged language to assemble efficient programs for sensor networks.

The most similar language to Frenetic is Nettle [28]. Nettle is also based on FRP, but it operates at a different level of abstraction than Frenetic: Nettle is an effective *substitute* for NOX; Frenetic, in contrast, *sits on top of* NOX, and, in the future, could potentially sit on top of Nettle. In other words, Nettle is designed to issue streams of (low-level) OpenFlow commands directly; it does not have any analogue of Frenetic's run-time system or its support for composition of possibly overlapping modules.

Another related language is NDLog, which has been used to specify and implement routing protocols, overlay networks, and services such as distributed hash tables [20, 19]. NDLog differs from Frenetic in that it is designed for distributed systems (rather than a centralized controller) and is based on logic programming. Also based on logic programming, FML focuses on specifying policies such as access control in OpenFlow networks [16]. Finally, the SNAC OpenFlow controller [4] provides a GUI for specifying access control policies using high-level patterns similar to the ones we have developed for Frenetic. However, SNAC provides a much less general



(a) Heavy Hitters (b) Web Statistics

Figure 16: Comparison of run-time switch ruleset reconciliation strategies.

programming environment than Frenetic.

One of the main challenges in the implementation of Frenetic involves splitting work between the (powerful but slow) controller and the (fast but limited) switches. A similar challenge appears in the implementation of Gigascope [9], a stream database for monitoring networks. However, Gigascope is less expressive than than Freneticas it only supports querying traffic and cannot be used to control the network itself.

10 Conclusions and Future Work

This paper describes the design and implementation of Frenetic, a new language for programming OpenFlow networks. Frenetic addresses some serious problems with the OpenFlow/NOX platform by providing a high-level, compositional, and unified programming model. It includes a collection of operators for transforming streams of network traffic, and a run-time system that manages the switch-level rules.

The experiments from Section 7 demonstrate that a highly unoptimized run-time system can compete comparably with programs written without the benefits of a run-time system. However, these results expose only one set of myriad design choices that we have made in the current prototype implementation. For instance, one such design decision we call the *rule reconciliation strategy*. Whenever the high-level forwarding policy changes, the switch-level rules installed may now contain stale actions. The run-time must then decide how to update the switches in the network. In the evaluated implemenation, we employ a "nuclear" option that simply empties all switch flow tables and allows normal network operation to rebuild them from scratch. Such an

option would certainly interrupt traffic on active network flows and would result in some retransmission of data. Another option might be to *reconcile* the rules and repopulate the entire flow table with (potentially) updated actions.

Figure 16 shows a subset of the microbenchmarks executed using the nuclear and reconcile strategies. We see that that the reconcile strategy causes additional communication between the controller and the network switches. However, our microbenchmarks do not contain long-lasting flows which would fully expose the problematic retransmission described above. We do see from this small example that the current design of the run-time system is hardly cemented and a more exhaustive analysis of the design and implementation of a network run-time system is warranted. Specifically, investigating the tradeoffs between network load, controller throughput, and update consistency across the network will require a much more thorough analysis on a physical testbed where experiments can be run with true performance fidelity.

In addition to maturing the run-time system, we are working to extend Frenetic in several directions. We are developing additional applications for a variety of common network tasks including fault-tolerant path computation, authentication and access control, and a framework inspired by FlowVisor [27] for ensuring isolation between programs. We are developing a front-end and an optimizer that will transform programs into a form that can be efficiently implemented on the run-time system. We are exploring a proactive strategy that eagerly generates rules based on the subscribers and forwarding policy and plan to compare the tradeoffs between rule generation strategies empirically. Finally, we aim to extend Frenetic's programmatic control of network elements beyond OpenFlow switches all the way to the end-hosts and, ultimately, to network services operated by groups of end-hosts.

References

[1] Beacon: A java-based openflow control platform. See `http://www.beaconcontroller.net`, Nov 2010.

[2] Memcached: A distributed memory object caching system. See `http://www.memcached.org`, Nov 2010.

[3] OpenFlow. See `http://www.openflowswitch.org`, Nov 2010.

[4] SNAC. See `http://snacsource.org/`, 2010.

[5] Openflow operations per second controller benchmark. See `http://www.openflow.org/wk/index.php/Oflops`, Mar 2011.

[6] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *TOPLAS*, 28:990–1034, November 2006.

[7] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *Trans. on Networking.*, 17(4), Aug 2009.

[8] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, pages 294–308, 2006.

[9] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, pages 647–651, New York, NY, USA, 2003. ACM.

[10] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 163–173, Jun 1997.

[11] David Erickson et al. A demonstration of virtual machine mobility in an OpenFlow network, Aug 2008. Demo at *ACM SIGCOMM*.

[12] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[13] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.

[14] Nikhil Handigol, Srinivasan Seetharaman, Mario Flajslik, Nick McKeown, and Ramesh Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow, Aug 2009. Demo at *ACM SIGCOMM*.

[15] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving energy in data center networks. In *NSDI*, Apr 2010.

[16] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *WREN*, pages 1–10, New York, NY, USA, 2009. ACM.

[17] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*, Oct 2010.

[18] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *HotNets*, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[19] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. *SIGOPS*, 39(5):75–90, 2005.

[20] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *SIGCOMM*, pages 289–300, New York, NY, USA, 2005. ACM.

[21] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. In *ICFP*, pages 335–346, 2008.

[22] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[23] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *OOPSLA*, pages 1–20, New York, NY, USA, 2009. ACM.

[24] Ankur Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: Dynamic access control in enterprise networks. In *WREN*, Aug 2009.

[25] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell Workshop*, pages 51–64, Pittsburgh, Pennsylvania, USA, Oct 2002. ACM Press.

[26] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *PADL*, Jan 1999.

[27] Rob Sherwood, Michael Chan, Glen Gibb, Nikhil Handigol, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, David Underhill, Kok-Kiong Yap, Guido Appenzeller, and Nick McKeown. Carving research slices out of your production networks with OpenFlow. *SIGCOMM CCR*, 40(1):129–130, 2010.

[28] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, Jan 2011.

A Frenetic Program Source Code

Licence Although this header is not repeated in each source listing, each of the following source files contain code licensed under the following terms:

```
################################################################################
# The Frenetic Project                                                         #
# frenetic@frenetic-lang.org                                                   #
################################################################################
# Licensed to the Frenetic Project by one or more contributors. See the        #
# NOTICE file distributed with this work for additional information            #
# regarding copyright and ownership. The Frenetic Project licenses this        #
# file to you under the following license.                                     #
#                                                                              #
# Redistribution and use in source and binary forms, with or without           #
# modification, are permitted provided the following conditions are met:       #
# - Redistributions of source code must retain the above copyright             #
#   notice, this list of conditions and the following disclaimer.              #
# - Redistributions in binary form must reproduce the above copyright          #
#   notice, this list of conditions and the following disclaimer in            #
#   the documentation or other materials provided with the distribution.       #
# - The names of the copyright holds and contributors may not be used to       #
#   endorse or promote products derived from this work without specific        #
#   prior written permission.                                                  #
#                                                                              #
# Unless required by applicable law or agreed to in writing, software          #
# distributed under the License is distributed on an "AS IS" BASIS, WITHOUT    #
# WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the     #
# LICENSE file distributed with this work for specific language governing      #
# permissions and limitations under the License.                               #
################################################################################
```

A.1 Centralized ARP Server - `arpd.py`

```
import nox.coreapps.examples.frenetic_util as util
import time as time
import nox.lib.openflow as openflow
from nox.coreapps.examples.frenetic_lib import *
from nox.coreapps.examples.frenetic_net import *
from nox.lib.packet.ethernet import ethernet
from nox.lib.packet.ipv4 import ipv4
from nox.lib.packet.arp import arp
from nox.lib.packet.packet_utils import *
from logging import getLogger

import learning_switch

log = getLogger('arpd')

# build_arp_reply : NOXPacket * MAC -> NOXPacket
# Construct an ARP reply based on on a request and a MAC in the NOX
# Packet format.
def build_arp_reply(req,srcmac):
    reply = arp()
    (reply.hwdst, reply.protodst, reply.hwsrc, reply.protosrc
      ) = (req.hwsrc, req.protosrc, octstr_to_array(srcmac), req.protodst)
    (reply.hwtype, reply.hwlen, reply.prototype, reply.protolen, reply.opcode
        ) = (reply.HW_TYPE_ETHERNET, 6, reply.PROTO_TYPE_IP, 4, reply.REPLY)
    frame = ethernet()
    (frame.dst,frame.src,frame.type) = (req.hwsrc,
        octstr_to_array(srcmac), ethernet.ARP_TYPE)
    frame.set_payload(reply)
    return frame

# arp_reply : Packet * NOXARPRequest * MAC -> Packet
# Construct an ARP reply based on a request and a MAC in the
# Frenetic Packet object format
def arp_reply(packet,request,srcmac):
    f = build_arp_reply(request,srcmac)
    return packet_of_raw_packet(switch(header(packet)),
            inport(header(packet)), f, len(f.tostring()))

# extractARP : Packet -> Packet * NOXARPPacket
# Extracts an ARP packet in NOX format from a Frenetic Packet object
def extractARP(packet):
    pkt = packet.payload
    d = None
    if pkt.parsed:
        d = pkt.find('arp')
    return d

# extractARPType : (ARPType * NOXPacket) -> NOXARPPacket option
# Extracts ARP packets of a given type
def extractARPType(typ,pkt):
    arpkt = extractARP(pkt)
    if arpkt is None:
        return None
    else:
        if ((arpkt.prototype == arpkt.PROTO_TYPE_IP) and
            (arpkt.opcode == typ)):
            return arpkt
        else:
            return None

# extractRequest : (NOXPacket) -> NOXARPPacket option
```

```
62  # Extracts ARP requests only from NOX Packets
63  def extractRequest(pkt):
64      return extractARPType(arp.REQUEST,pkt)
65
66  # extractResponse : (NOXPacket) -> NOXARPPacket option
67  # Extracts ARP replies only from NOX packets
68  def extractReply(pkt):
69      return extractARPType(arp.REPLY,pkt)
70
71  # fwd_arp_request : switch * port * Packet -> Packet
72  # Generate a Frenetic packet object for output
73  def fwd_arp_request(sw,p,pkt):
74      return packet_of_raw_packet(sw,p,pkt.payload, len(pkt.payload.tostring()))
75
76  # iptomac : ip * Dict -> mac
77  # Returns the MAC address bound to the requested IP
78  def iptomac(ip,table):
79      mac = None
80      for (m,(s,i,p)) in table.items():
81          if ip == i:
82              mac = m
83      return mac
84
85  # process : ARPPacket (ARPPacket * AddressBindings Dict) EF
86  # Process a given ARP request and generate a response
87  def processPkt((pkt,(d,last))):
88
89      pktOut = None
90      hdr = net.header(pkt)
91      req = extractRequest(pkt)
92      rep = extractReply(pkt)
93
94      # Received a request
95      # Check table and make a reply
96      if not(req is None):
97          reqip = ip_to_str(req.protodst)
98          srcip = ip_to_str(req.protosrc)
99          mac = iptomac(reqip,d)
100         ##Update ARP Data from Source
101         ##Should probably check fist, but just update for now
102         d[net.srcmac(hdr)] = (net.switch(hdr),srcip,net.inport(hdr))
103         if mac == None:
104             # No record of this host, forward a flood
105             log.info("IP %s NOT FOUND" % reqip)
106             pktOut = fwd_arp_request(net.switch(hdr),openflow.OFPP_FLOOD,pkt)
107         else:
108             log.info("SHORT CIRCUT REPLY to IP %s for IP %s with MAC %s" %
109                      (srcip,reqip,mac))
110             pktOut = arp_reply(pkt,req,mac)
111     # Received a reply
112     # Update table and forward
113     elif not(rep is None):
114         srcip = ip_to_str(rep.protosrc)
115         dstip = ip_to_str(rep.protodst)
116         ##Update ARP Data from Source and destination
117         ##Should probably check fist, but just update for now
118         d[net.srcmac(hdr)] = (net.switch(hdr),srcip,net.inport(hdr))
119         ##Lookup output port
120         dstmac = mac_to_str(rep.hwdst)
121         (sw,ip,p) = d[dstmac]
122         pktOut = pkt
123         ## Remember when sending a packet inport is really output
```

```
124             hdr.inport = p
125             pktOut.header = hdr
126             log.info("STORING MAC %s for IP %s" % (net.srcmac(hdr),srcip))
127         return ((d,pktOut),(d,pktOut))
128
129     # bindings : (AddressBindings Dict) E
130     # Returns the current mapping of MAC addresses to switch, and IP address
131     def bindings():
132         # arps : E ARPPacket
133         arps = (Select('packets') *
134                 Where(ethtype_fp(ethernet.ARP_TYPE)))
135         (d,pkt) = Split(arps >> (LoopPre(({},None), Lift(processPkt))))
136         (pkt >> Filter(lambda x: not(x is None)) >> NOXSendPkt())
137         return d
138
139     # main : unit
140     # Publically accessible function to run as stand-alone application with a
141     # learning forwarding policy
142     def main():
143         learning_switch.main()
144         (bindings() >> Print(">> ")
```

A.2 DHCP Server - dhcpd.py

```
import nox.coreapps.examples.frenetic_util as util
import time as time

from nox.coreapps.examples.frenetic_lib import *
from nox.coreapps.examples.frenetic_net import *
from nox.lib.packet.ethernet import ethernet
from nox.lib.packet.ipv4 import ipv4
from nox.lib.packet.dhcp import dhcp
import nox.lib.packet.packet_utils as putil
from array import *
from logging import getLogger
import random as random

log = getLogger('dhcpd')

def mip_from_mbits(mb):
    mip = 0
    for i in range(31,(32-mb-1),-1):
        mip = mip | (1 << i)
    return mip

def mipstr_from_mbits(mb):
    return putil.ip_to_str(mip_from_mbits(mb))

## CONSTANTS
[PENDING_STATE, ACTIVE_STATE] = range(0,2)

## TUNABLE PARAMETERS
## Configure the DHCP Pool and Default Gateway
POOL = '172.16.0.0/16'
[POOL_NET,POOL_MASKBITS] = POOL.split("/")
POOL_GW = '172.16.0.1'
POOL_START = '172.16.0.2'
POOL_END = '172.16.0.254'
POOL_GW_MAC = "00:ff:00:ff:00:ff"
# In Seconds
LEASE_TIME = 86399

## DERIVED PARAMETERS
POOL_GW_OFFSET = putil.ipstr_to_int(POOL_GW) - putil.ipstr_to_int(POOL_NET)
POOL_MASKBITS = int(POOL_MASKBITS)
POOL_MASK = mipstr_from_mbits(POOL_MASKBITS)
POOL_START_OFFSET = putil.ipstr_to_int(POOL_START) - putil.ipstr_to_int(POOL_NET)
POOL_END_OFFSET = putil.ipstr_to_int(POOL_END) - putil.ipstr_to_int(POOL_NET)
POOL_SEED = POOL_END_OFFSET - POOL_START_OFFSET

## Helper functions
# mip_from_mbits : MaskBits:int -> MaskAddress:ipint
def mip_from_mbits(mb):
    mip = 0
    for i in range(31,(32-mb-1),-1):
        mip = mip | (1 << i)
    return mip
# mipstr_from_mbits : MaskBits:int -> MaskAddress:ipstr
def mipstr_from_mbits(mb):
    return putil.ip_to_str(mip_from_mbits(mb))

# build_dhcp_pkt : Create a DHCP packet with the given parameters
def build_dhcp_pkt(m, a, sm, gwip, gwm, xid, msg):
    # Convert ipstr subnet mask, and gw ip into list of ints
    sm = map(int,sm.split('.'))
```

```
62      gwiplist = map(int, gwip.split('.'))
63      # Create DHCP message
64      dr = dhcp()
65      (dr.op, dr.hlen, dr.magic, dr.xid) = (2,6, dhcp.MAGIC, xid)
66      dr.yiaddr = putil.ipstr_to_int(a)
67      dr.siaddr = putil.ipstr_to_int(gwip)
68      dr.chaddr = putil.octstr_to_array(m)
69      # DHCP Option 53, DHCP Message Type
70      dr.addUnparsedOption(dhcp.MSG_TYPE_OPT, 1, [msg])
71      # DHCP Option 1, Subnet Mask
72      dr.addUnparsedOption(dhcp.SUBNET_MASK_OPT, 4, sm)
73      # DHCP Option 3, router
74      dr.addUnparsedOption(dhcp.GATEWAY_OPT, 4, gwiplist)
75      # DHCP Option 51, lease time
76      dr.addUnparsedOption(dhcp.REQUEST_LEASE_OPT, 4, [0,1,81,128])
77      dr.addUnparsedOption(dhcp.END_OPT,0,[])
78      # Fill the dhcp object arr field
79      dr.arr = dr.tostring()
80      # Create UDP datagram
81      ur = udp()
82      (ur.srcport,ur.dstport, ur.len) = (67,68, (udp.MIN_LEN + len(dr.tostring())))
83      ur.set_payload(dr)
84      # Fill the udp object arr field
85      ur.arr = ur.tostring()
86      # Create IPv4 packet
87      ipr = ipv4()
88      ipr.srcip = putil.ipstr_to_int(gwip)
89      ipr.dstip = putil.ipstr_to_int('255.255.255.255')
90      ipr.protocol = ipv4.UDP_PROTOCOL
91      ipr.iplen = ipv4.MIN_LEN + len(ur.tostring())
92      ipr.set_payload(ur)
93      ipr.arr = ipr.tostring()
94      # Calculate checksums now that packets have been filled
95      ipr.csum = ipr.checksum()
96      ur.csum = ur.checksum()
97      # Create ethernet frame
98      er = ethernet()
99      er.dst = putil.octstr_to_array(m)
100     er.src = putil.octstr_to_array(gwm)
101     er.type = ethernet.IP_TYPE
102     er.set_payload(ipr)
103     return er
104
105 # dhcp_offer: (Params) -> Packet
106 # Create a DHCP offer packet based on the given parameters
107 def dhcp_offer(dpid, port, m, a, sm, gwip, gwm, xid):
108     pkt = build_dhcp_pkt(m, a, sm, gwip, gwm, xid, dhcp.OFFER_MSG)
109     return packet_of_raw_packet(dpid, port, pkt, len(pkt.tostring()))
110
111 # dhcp_ack: (Params) -> Packet
112 # Create a DHCP ACK packet based on the given parameters
113 def dhcp_ack(dpid, port, m, a, sm, gwip, gwm, xid):
114     pkt = build_dhcp_pkt(m, a, sm, gwip, gwm, xid, dhcp.ACK_MSG)
115     return packet_of_raw_packet(dpid, port, pkt, len(pkt.tostring()))
116
117 # is_assigned: Leases * IP -> bool
118 # Determine if a given address is bound in the current leases
119 def is_assigned(l,i):
120     for (m,(sw,p,a,s)) in l.items():
121         if a == i:
122             return True
123     return False
```

```
124
125 # set_lease Leases * Switch * MAC * Port * IP -> Leases
126 # Add a binding to the current leases
127 def set_lease(l,sw,m,p,a):
128     l[m] = (sw,p,a, ACTIVE_STATE)
129     return l
130
131 # find_lease: Leases * MAC -> IP option
132 # Lookup the address bound the given MAC
133 def find_lease(l,m):
134     if l.has_key(m):
135         (sw,p,a,s) = l[m]
136         return a
137     else:
138         return None
139
140 # get_pending_lease : Leases * Switch * MAC * Port -> IP
141 # Add a tentative binding to the current leases awaiting confirmation
142 def get_pending_lease(l,sw,m,p):
143     if m in l.keys():
144         (switch, port, addr, status) = l[m]
145         if (sw == switch) and (p == port):
146             return addr
147         else:
148             l[m] = (sw, p, addr, status)
149             return addr
150     else:
151         addr = putil.ip_to_str(putil.ipstr_to_int(POOL_NET) +
152             POOL_START_OFFSET + random.randint(0,POOL_SEED))
153         excluded = map(lambda(xsw,xm,xa,xss):xa, l.values())
154         while addr in excluded:
155             addr = putil.ip_to_str(putil.ipstr_to_int(POOL_NET) +
156                 POOL_START_OFFSET + random.randint(0,POOL_SEED))
157         l[m] = (sw,p,addr, PENDING_STATE)
158     return (l,addr)
159
160 # confirm_lease : Leases * MAC -> Leases
161 # Confirm a pending binding in the current leases
162 def confirm_lease(l,m):
163     if m in l.keys():
164         (sw,p,a,s) = l[m]
165         l[m] = (sw,p,a,ACTIVE_STATE)
166     return l
167
168 # extractDHCP : Packet -> (Packet, NOXDHCPPacket option)
169 # Extracts a NOX DHCP packet from a Frenetic packet object
170 def extractDHCP(packet):
171     pkt = packet.payload
172     d = None
173     if pkt.parsed:
174         d = pkt.find('dhcp')
175         if not(d.parsedOptions.has_key(dhcp.MSG_TYPE_OPT)):
176             d = None
177     return (packet,d)
178
179 # Predicates on DHCP packet type
180 # d: PARSED and VALID DHCP packet
181 def isDiscover(d):
182     return (d.parsedOptions[dhcp.MSG_TYPE_OPT] == array('B',[dhcp.DISCOVER_MSG]))
183
184 def isRequest(d):
185     return (d.parsedOptions[dhcp.MSG_TYPE_OPT] == array('B',[dhcp.REQUEST_MSG]))
```

```
186
187 # processRequest : (Packet * Leases) (Leases, Packet) EF
188 # Process the request through the DHCP state machine
189 def processRequest(((packet,d),(leases,z))):
190     pktOut = None
191     dpid = switch(header(packet))
192     (m,p,txid) = (srcmac(header(packet)), inport(header(packet)), d.xid)
193
194     if isDiscover(d):
195         (leases,a) = get_pending_lease(leases,dpid, m, p)
196         log.info("INITIALOFFER %s:%s:%s" % (m,p,a))
197         pktOut = dhcp_offer(dpid,p,m,a,POOL_MASK, POOL_GW, POOL_GW_MAC, txid)
198
199     elif isRequest(d):
200         if d.parsedOptions.has_key(dhcp.REQUEST_IP_OPT):
201             req = putil.array_to_ipstr(d.parsedOptions[dhcp.REQUEST_IP_OPT])
202             a = find_lease(leases,m)
203             if a == None:
204                 # No active lease found for this client, but client wants req
205                 log.info("NO ACTIVE LEASE FOR CLIENT: %s" % m)
206                 if not(is_assigned(leases,req)) and in_network(req, POOL_GW, POOL_MASK):
207                     a = req
208                     log.info("*REQUESTED IP %s IS AVAILABLE" % req)
209                     # req is not assigned to any active lease, so ack
210                     leases = set_lease(leases,dpid,m,p,req)
211                     pktOut = dhcp_ack(dpid,p,m,req,POOL_MASK, POOL_GW, POOL_GW_MAC,txid)
212                 else:
213                     # req is already assigned so make new offer
214                     (leases,a) = get_pending_lease(leases,dpid,m,p)
215                     log.info("*REQUESTED IP %s IS UNAVAILABLE, COUNTEROFFER %s" %
216                              (req,a))
217                     pktOut = dhcp_offer(dpid,p,m,a,POOL_MASK, POOL_GW, POOL_GW_MAC, txid)
218             elif a == req:
219                 log.info("REQUEST FOR %s MATCHES LEASE FOR %s" % (a,m))
220                 # Active lease for mac m was found and that lease matches req
221                 # Send Ack
222                 confirm_lease(leases,m)
223                 pktOut = dhcp_ack(dpid,p,m,req,POOL_MASK, POOL_GW, POOL_GW_MAC,txid)
224
225             elif a != req:
226                 log.info("REQUEST FOR %s DOES NOT MATCH ACTIVE LEASE FOR %s" % (req,m))
227                 log.info("COUNTEROFFER %s WITH CURRENT ENTRY %s" % (m, a))
228                 # Active lease for mac m was found, but m requested a different value
229                 # Make counteroffer with current lease
230                 pktOut = leases,dhcp_offer(dpid,p,m,a,POOL_MASK, POOL_GW,
231                                            POOL_GW_MAC,txid)
232         else:
233             # Received a request without an IP, make new offer
234             (leases,a) = get_pending_lease(leases,dpid,m,p)
235             pktOut = leases,dhcp_offer(dpid,p,m,a,POOL_MASK, POOL_GW, POOL_GW_MAC,txid)
236     t = (leases,pktOut)
237     return (t,t)
238
239 def validDHCP((p,d)):
240     return not(d is None)
241
242 ## leases : (DHCP Lease Dictionary) E
243 ## Publically accessible event stream carrying the current leases confirmed
244 ## By the DHCP server
245 def leases():
246     dhcp_fp = (ethtype_fp(ethernet.IP_TYPE) & protocol_fp(ipv4.UDP_PROTOCOL) &
247                dstip_fp('255.255.255.255', "255.255.255.255") &
```

```
248                 srcip_fp('0.0.0.0', "255.255.255.255") &
249                 dstport_fp(67))
250
251     # dhcp_query : E Packet
252     dhcp_query = (Select('packets') * Where (dhcp_fp))
253
254     # Parse and push the query result through the DHCP state machine
255     # outputting the resulting packet
256     (l,p) = Split(dhcp_query >>
257                   Lift(extractDHCP) >>
258                   Filter(validDHCP) >>
259                   LoopPre(({},None),Lift(processRequest)))
260     p >> Filter(lambda x: not(x is None)) >> NOXSendPkt()
261     return l
262
263 ## main : unit
264 ## Publically accessible function to run module as a stand-alone application
265 def main():
266     # Invoke the leases event stream and display the results
267     leases() >> Print(">> ")
```

A.3 Parameterized Load Balancer - `lb.py`

```
import time as time
from logging import getLogger

import nox.coreapps.examples.frenetic_util as util
from nox.coreapps.examples.frenetic_lib import *
from nox.coreapps.examples.frenetic_net import *
import learning_switch

log = getLogger('loadbalanced')

## CONFIGURABLE PARAMETERS
# Define the switches and ports that contain replicas to be load balanced
REPLICA_PORTS = {102:[3,4,5]}
SWITCHES = REPLICA_PORTS.keys()

# Tabulate : (int * int * bool) (Dict) EF
# Takes an Event of key,value (switch, port) pairs to an Event of Dictionaries
# containing for each key (switch) a port-indexed list of single values
def Tabulate():
    def f(((sw,p,st),d)):
        if st:
            if d.has_key(sw):
                if not(d[sw].has_key(p)):
                    d[sw][p] = 0
            else:
                d[sw] = {p:0}
        else:
            if d.has_key(sw):
                if d[sw].has_key(p):
                    del d[sw][p]

        return (d,d)
    return LoopPre({},Lift(f))

# FilterRules: int list -> (Rule Dict) (Rule Dict) EF
# Takes an Event of Rule Sets and Filters out rules for switches
# contained in the list sw.
def FilterRules(sw):
    def f(rs):
        nrs = {}
        for k in rs.keys():
            if k not in sw:
                nrs[k] = rs[k]
        return nrs
    return Lift(f)

# rr: (Dict * int * Dict) -> int * Dict
# For a list of switches, returns the next port to be used in that
# switch's list in a round-robin fashion as well as a dictionary of
# state used locally

def rr(pd,switch,state):
    if state.has_key(switch):
        il = REPLICA_PORTS[switch].index(state[switch])
        length = len(REPLICA_PORTS[switch])
        op = REPLICA_PORTS[switch][(il+1)%length]
    else:
        op = REPLICA_PORTS[switch][0]
    state[switch] = op
    return (op,state)

```

```
# lmin: (Dict * int * Dict) -> int * Dict
# For a list of switches, returns the next port to be used in that
# switch's list based on load.

def lmin(pd,switch,state):
    op = None
    for sw in pd.keys():
        if sw == switch:
            l = pd[sw].items()
            l.sort(cmp=lambda (k1,i1),(k2,i2): cmp(i1,i2))
            ## There are some size/boundary conditions worth thinking about
            op = l[0][0]
            if state.has_key(switch):
                if state[switch] == op:
                    op = l[1][0]
                state[switch] = op
            else:
                state[switch] = op
    return (op,state)

# Balance: ((Dict * int * Dict) -> int * Dict) -> (Dict * Packet) (int * Packet) EF
# Returns an event function that takes an Event of network state and packets
# and returns an Event of output ports and packets
def Balance(f):
    def g(((d,pkt),(state,(pl,pkl)))):
        sw = switch(net.header(pkt))
        (outport,state) = f(d,sw,state)
        return ((state,(outport,pkt)),(state,(outport,pkt)))
    return (LoopPre(({},(None,None)),Lift(g)) >> Snd())

# PortStatus : (int list * int list) -> (Dict) E
# Takes a list of switches and replica ports and returns a dictionary
# of the current network state that is a subset of those switches and ports
def PortStatus(sl,pl):
    s = (PortEvents() >> Filter(lambda pe: portswitch(pe) in sl) >>
         Filter(lambda pe: portnum(pe) != 65534 and portnum(pe) in pl[portswitch(pe)])
         >> Lift(lambda pe: (portswitch(pe), portnum(pe), portenabled(pe)))
         >> Tabulate())
    return s

# MakeRule : (int * Packet) E -> (Rule list) E
# Takes an event of output ports and packets to an Event of Rules
# specifying the forwarding of the flow described by packet
def MakeRule():
    def f((op,pkt)):
        hdr = net.header(pkt)
        (client,inp,sw) = (srcip(hdr),inport(hdr),switch(hdr))
        rules = [Rule(srcip_fp(client) & inport_fp(inp), [forward(op)]),
                 Rule(dstip_fp(client) & inport_fp(op), [forward(inp)])]
        return (sw,rules)
    return Lift(f)

# addRule : (int * Rule list) -> Rule Dict
# Accumlator function from switches and rule lists to rule sets
def addRule((sw,rl),d):
    if d.has_key(sw):
        d[sw] += rl
    else:
        d[sw] = rl
    return d

# Update : (port Dict * status Dict) (port Dict) EF
```

```
124 # Update the current status of ports in the network with the lastest results
125 # from the stats query
126 def Update():
127     def f((pd,sd)):
128         for (sw,p) in sd.keys():
129             if p in REPLICA_PORTS[sw]:
130                 pd[sw][p] = sd[(sw,p)]
131         return pd
132     return Lift(f)
133 
134 # CombineRules : (Rule Dict * Rule Dict) (Rule Dict) EF
135 # Combine forwarding rules and modification rules as a simple union
136 def CombineRules():
137     def f((rsa,rsb)):
138         rsc = {}
139         if not(rsa is None):
140             for k in rsa.keys():
141                 if rsc.has_key(k):
142                     rsc[k] += rsa[k][:]
143                 else:
144                     rsc[k] = rsa[k][:]
145         if not(rsb is None):
146             for k in rsb.keys():
147                 if rsc.has_key(k):
148                     rsc[k] += rsb[k][:]
149                 else:
150                     rsc[k] = rsb[k][:]
151         return rsc
152     return Lift(f)
153 
154 # rules : (Rule Dict) E
155 # Public interface to the rules generated by the load balancer
156 def rules():
157     # fp: filter_pattern describing all replica ports on all participating
158     #     switches
159     fp = or_fp(map(lambda (k,v):and_fp([switch_fp(k),
160                 not_fp(or_fp(map(inport_fp,v)))]), REPLICA_PORTS.items()))
161 
162     # q: (Packet) E
163     # All first packets destined for replica ports
164     q = (Select('packets') *
165          Where(fp) *
166          GroupBy(['switch']) *
167          SplitWhen(['srcip']) *
168          Limit(1) >>
169          Snd())
170 
171     # stq: (Dict) E
172     # Status query that generates Dict with keys of (switch * inport) and
173     # values of sizes
174     stq = (Select('sizes') *
175            Where(or_fp(map(net.switch_fp,SWITCHES))) *
176            GroupBy(['switch','inport']) *
177            Every(5) >> Identity())
178 
179     # s: (Dict) E
180     # Current updated network status based on stats query and network events
181     s = (Snapshot(Hold({},PortStatus(SWITCHES,REPLICA_PORTS)),stq) >> Update())
182 
183     # rs: (RuleSet) E
184     # Apply the Balance function to the current network status and most recent
185     # query to a replica port to generate the appropriate rules
```

```
186     rs = (Snapshot(Hold({},s),q) >>
187           Balance(lmin) >> Filter(lambda (op,pkt):not(op is None)) >>
188             MakeRule() >> Accum({},addRule))
189 
190     return rs
191 
192 # main : unit
193 # Publicly accessible function to run module stand-alone with a default
194 # forwarding policy using the learning switch.
195 def main():
196     l = (learning_switch.rules() >> FilterRules(SWITCHES))
197     (StickyMerge(rules(),l) >> CombineRules() >> Probe(">>") >> Register())
```

A.4 Memcached Query Router - `memcached.py`

```
import nox.coreapps.examples.frenetic_util as util
import time as time
from nox.coreapps.examples.frenetic_lib import *
from nox.coreapps.examples.frenetic_net import *
from nox.lib.packet.ethernet import ethernet
from nox.lib.packet.ipv4 import ipv4
from nox.lib.packet.arp import arp
from nox.lib.packet.packet_utils import *
from logging import getLogger

import dhcpd
import learning_switch

log = getLogger('memcached')

## CONFIGURABLE PARAMETERS
CLIENT_PORTS = [1]
SERVER_PORTS = range(2,5)

## Set of virtual ID's for the memcache partition
## This is computed offline.
vidset = ['172.16.208.31', '172.16.124.217', '172.16.122.78', '172.16.243.69',
          '172.16.58.152', '172.16.42.15', '172.16.217.100', '172.16.160.188',
          '172.16.252.91', '172.16.151.40']

## HELPER FUNCTIONS

# Status : (k,v) Dict EF
# Maintains a single get/set value for single key
def Status():
    def f(((k,v),d)):
        if not d.has_key(k):
            d[k] = v
        d[k]=v
        return (d,d)
    return LoopPre({},Lift(f))

# LeftStickyMerge : a E * b E -> (a * b) E
# Merges two events such that the last value of a accompanies
# any value of b
def LeftStickyMerge(e1,e2):
    def f(((x,y),(xl,yl))):
        retx = xl if (x is None) else x
        return ((retx,y),(retx,y))
    return (Apply(Merge(e1,e2),LoopPre((None,None),Lift(f))))

# build_arp_reply : NOXPacket * MAC -> NOXPacket
# Construct an ARP reply based on on a request and a MAC in the NOX
# Packet format.
def build_arp_reply(req,srcmac):
    reply = arp()
    (reply.hwdst, reply.protodst, reply.hwsrc, reply.protosrc
      ) = (req.hwsrc, req.protosrc, octstr_to_array(srcmac), req.protodst)
    (reply.hwtype, reply.hwlen, reply.prototype, reply.protolen, reply.opcode
        ) = (reply.HW_TYPE_ETHERNET, 6, reply.PROTO_TYPE_IP, 4, reply.REPLY)
    frame = ethernet()
    (frame.dst,frame.src,frame.type) = (req.hwsrc,
        octstr_to_array(srcmac), ethernet.ARP_TYPE)
    frame.set_payload(reply)
    return frame

```

```
62  # arp_reply : Packet * NOXARPRequest * MAC -> Packet
63  # Construct an ARP reply based on a request and a MAC in the
64  # Frenetic Packet object format
65  def arp_reply(packet,request,srcmac):
66      f = build_arp_reply(request,srcmac)
67      return packet_of_raw_packet(switch(header(packet)),
68              inport(header(packet)), f, len(f.tostring()))
69
70  # extractARP : Packet -> Packet * NOXARPPacket
71  # Extracts an ARP packet in NOX format from a Frenetic Packet object
72  def extractARP(packet):
73      pkt = packet.payload
74      d = None
75      if pkt.parsed:
76          d = pkt.find('arp')
77          if d.opcode != arp.REQUEST:
78              d = None
79      return (packet,d)
80
81  # which_pid VirtualIP * PartitionDictionary -> PhysicalIP
82  # Looks up the Physical IP address corresponding to a given virtual IP address
83  # in the current partitioning
84  def which_pid(v,pd):
85      for p in  pd.keys():
86          (m,a,vids) = pd[p]
87          if v in vids:
88              return p
89
90  ## PRIMARY SUBMODULES
91
92  # MakeState : (LeasesDict * PortDict) (StateDict) EF
93  # Correlates lease data about physical IPs and MACs with status information
94  # about ports.
95  def MakeState():
96      def f(((dl,dp),state)):
97          # dl : DHCP lease dict, dp: Port status dict
98          # state : is network status table
99          if dp != None:
100             for p in dp.keys():
101                 (m,a,s) = (None,None,False)
102                 if state.has_key(p):
103                     (m,a,s) = state[p]
104                 state[p] = (m,a,dp[p])
105
106         if dl != None:
107             for m in dl.keys():
108                 (switch,port,addr,dstatus) = dl[m]
109                 if state.has_key(port):
110                     (mac,a,s) = state[port]
111                     if dstatus == dhcpd.ACTIVE_STATE:
112                        state[port] = (m,addr,s)
113         return (state,state)
114     return LoopPre({},Lift(f))
115
116 # MakePartitions : (StateDict) (PartitionDict) EF
117 # Creates the partitioning of the vidset onto the
118 # available servers from the network state while
119 # ensuring that only V/k servers are displaced
120 # in any repartitioning event.
121
122 def MakePartitions():
123     def f((state,last)):
```

```
124             (added,lost,next) = ({},{},{})
125             for p in state:
126                 (m,a,s) = state[p]
127                 if s and (a != None):
128                     if not(a in last.keys()):
129                         added[a] = (m,p,[])
130                     else:
131                         next[a] = last[a]
132                 elif (a != None):
133                     if (a in last.keys()):
134                         lost[a] = last[a]
135             ## Assuming that only one node is
136             ## lost or added at any one event
137             knext = len(next) + len(added)
138             klast = len(last)
139             if len(added) > 0:
140                 if klast > 0:
141                     n = (len(vidset)//knext)
142                     for a in added.keys():
143                         (m,p,v) = added[a]
144                         while len(v) < n:
145                             for k in next.keys():
146                                 (mk,pk,vk) = next[k]
147                                 v.extend(vk[len(vk)-1:])
148                                 next[k] = (mk,pk,vk[:-1])
149                         next[a] = (m,p,v[:])
150                 else:
151                     for a in added.keys():
152                         (m,p,v) = added[a]
153                         next[a] = (m,p,vidset[:])
154             elif len(lost) > 0:
155                 if knext > 0:
156                     for a in lost.keys():
157                         (m,p,v) = lost[a]
158                         while len(v) > 0:
159                             for k in next.keys():
160                                 (mk,pk,vk) = next[k]
161                                 vk.extend(v[:1])
162                                 next[k] = (mk,pk,vk[:])
163                                 v = v[1:]
164             return (next,next)
165         return LoopPre({},Lift(f))
166 
167 ## MakeRules : (int list * int list) -> (Partition Dict * VID list) (Rules Dict) EF
168 ## Generates the set of rewriting rules that correspond to the current mapping
169 ## of vids to pids.  This EF must output only one return rule per physical ID.
170 
171 def MakeRules(cp,sp):
172     def f((pd,vl)):
173         DPID = 101
174         rs = {DPID:[]}
175         d = {}
176         for a in pd.keys():
177             d[a] = None
178             (m,p,vids) = pd[a]
179             for v in vids:
180                 ## Rewrite vid -> pid in client requests
181                 fpin = (ethtype_fp(ethernet.IP_TYPE) &
182                       dstip_fp(v, '255.255.255.255'))
183                 ain = [modify(dstip_mod(a))]
184                 rs[DPID].append(Rule(fpin,ain))
185                 ## Rewrite pid -> vid in server responses
```

```
186             if vl.has_key(a):
187                 fpout = (ethtype_fp(ethernet.IP_TYPE) &
188                          srcip_fp(a, '255.255.255.255') & srcport_fp(11211))
189                 aout = [modify(srcip_mod(vl[a]))]
190                 rs[DPID].append(Rule(fpout,aout))
191             else:
192                 fpout = (ethtype_fp(ethernet.IP_TYPE) &
193                          srcip_fp(a, '255.255.255.255') & srcport_fp(11211))
194                 aout = [modify(srcip_mod(pd[a][0]))]
195                 rs[DPID].append(Rule(fpout,aout))
196 
197         return rs
198     return Lift(f)
199 
200 ## PortStatus : int list -> (ports Dict) E
201 ## Processes the stream of port events for input to the network state
202 def PortStatus(y):
203     return (PortEvents() >>
204                 (Filter(lambda x: True if portnum(x) in y else False) >>
205                  Lift(lambda pe: (portnum(pe), portenabled(pe))) >>
206                  status()))
207 
208 ## ARPServer : int list * int list -> (Dict * Dict) Packet EF
209 ## Handle ARP Reponses based on the partitioning and DHCP leases
210 def ARPServer(cp,sp):
211     def g(((dl,pd),(packet,req))):
212         in_port = inport(header(packet))
213         rip = ip_to_str(req.protodst)
214         pktOut = None
215         log.info("arpserver: REQUEST FROM PORT %s, FOR IP %s" % (in_port,rip))
216         if (in_port in cp) and (rip in vidset):
217             pid = which_pid(rip,pd)
218             (mac,a,vids) = pd[pid]
219             pktOut = arp_reply(packet,req,mac)
220             log.info("arpserver: SENDING MAC %s FOR PID %s IN RESPONSE" % (mac,pid))
221         elif (in_port in sp):
222             for m in dl.keys():
223                 (switch,port,addr,dstatus) = dl[m]
224                 if addr == rip:
225                     log.info("arpserver: SENDING MAC %s FOR CLIENT %s IN RESPONSE" % (m,addr))
226                     pktOut = arp_reply(packet,req,m)
227         return pktOut
228     return Lift(g)
229 
230 # CombineRules : (Rule Dict * Rule Dict) (Rule Dict) EF
231 # Combine forwarding rules and modification rules as a simple union
232 def CombineRules():
233     def f((rsa,rsb)):
234         rsc = {}
235         if not(rsa is None):
236             for k in rsa.keys():
237                 if rsc.has_key(k):
238                     rsc[k] += rsa[k][:]
239                 else:
240                     rsc[k] = rsa[k][:]
241         if not(rsb is None):
242             for k in rsb.keys():
243                 if rsc.has_key(k):
244                     rsc[k] += rsb[k][:]
245                 else:
246                     rsc[k] = rsb[k][:]
247         return rsc
```

```
248     return Lift(f)
249
250 # SaveLastVid : (Dict * IP) (Dict * Dict) EF
251 # Based on the partition dictionary and the most recent request
252 # Maintain a mapping of only one vid per pid to actually install
253 # return rules for.
254 def SaveLastVid():
255     def f(((pd,dstip),(pl,vl))):
256         (added,lost, next) = ({},{},{})
257         for k in pd.keys():
258             if not(pl.has_key(k)):
259                 added[k] = pd[k]
260         for k in pl.keys():
261             if not(pd.has_key(k)):
262                 lost[k] = pl[k]
263         if (added == {}) and (lost == {}):
264             next = vl.copy()
265             # The partition set has not changed
266             if not(dstip is None):
267                 pid = which_pid(dstip,pd)
268                 next[pid] = dstip
269             else:
270                 log.debug("THIS SHOULD NEVER OCCUR")
271         else:
272             # The partition set has changed!
273             for k in vl.keys():
274                 vid = vl[k]
275                 lastpid = k
276                 newpid = which_pid(vid,pd)
277                 if (lastpid != newpid):
278                     #Try to reassign
279                     if not(vl.has_key(newpid) and next.has_key(newpid)):
280                         # No recent request for new pid, reassign
281                         next[newpid] = vid
282                 else:
283                         next[k] = vl[k]
284             if not(dstip is None):
285                 # Update if there is a new vid
286                 pid = which_pid(dstip,pd)
287                 next[pid] = dstip
288         return ((pd,next),(pd,next))
289     return LoopPre(({},{}),Lift(f))
290
291 def main():
292     ## Define Memcached request pattern
293     mcreq_fp = (ethtype_fp(ethernet.IP_TYPE) &
294                 protocol_fp(ipv4.UDP_PROTOCOL) & dstport_fp(11211))
295
296     ## mc_reqs : Dstip E
297     ## Query that returns an event of memcache request destination IPs
298     mc_reqs = (Select('packets') *
299                Where(mcreq_fp) *
300                GroupBy(['srcip']) *
301                SplitWhen(['dstip']) *
302                Limit(1) >>
303                Ungroup(1,lambda n,p:dstip(header(p)), None) >>
304                Lift(lambda (x,y):y)
305                )
306
307     ## Create the partitioning of the virtual ID's
308     la,lb = Split(dhcpd.leases() >> Dup())
309     pma,pmb = Split(Merge(la, PortStatus(SERVER_PORTS)) >>
```

```
310                     MakeState() >> MakePartitions() >> Dup())
311
312         # Generate the rewriting rules based on the current vid partitioning and
313         # combine them with the forwarding rules from the learning switch.
314         last = (LeftStickyMerge(pma,mc_reqs) >> SaveLastVid())
315         modrules = (last >> MakeRules(CLIENT_PORTS,SERVER_PORTS))
316         fwdrules = learning_switch.rules()
317         (StickyMerge(modrules,fwdrules) >> CombineRules() >> Register())
318
319         # Print the current partitioning to the console for debuging and monitoring
320         (pma >>
321             Lift(lambda d: map(lambda (a,(m,p,v)):(a,v),d.items())) >> Print(">>"))
322
323         # Based on the current partitioning, and the DHCP server data, generate
324         # ARP responses for the ARP requests for vids
325         arp_requests = (Select('packets') *
326                         Where(ethtype_fp(ethernet.ARP_TYPE)) >>
327                         Lift(extractARP) >> Filter(lambda (p,d): not(d is None)))
328         (Snapshot(Hold({},StickyMerge(lb,pmb)),arp_requests) >>
329           ARPServer(CLIENT_PORTS, SERVER_PORTS) >>
330           Filter(lambda x:not(x is None)) >>
331           NOXSendPkt())
```