Frenetic: A High-Level Language for OpenFlow Networks

Nate Foster, Cornell University
Rob Harrison, Princeton University
Michael J. Freedman, Princeton University
Jennifer Rexford, Princeton University
David Walker, Princeton University

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 run-time 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.

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, which opens up the software that controls the network [19]. 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 [10]. 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 [5, 21], Web server load balancing [11], energy-efficient networking [12], and seamless virtual-machine migration [9].

Unfortunately, while OpenFlow now makes it *possible* to implement exciting new network services, it certainly 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 considered what *the programmer* might want to say, rather than how *the hardware* implements it. In general, this means we attempted to supply high-level primitives, even when they were not directly supported by the hardware.

See-every-packet abstraction: Programmers do not have to worry that installing packet-handling rules prevents 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): Using our combined expertise in programming languages and networking, 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 propose the Frenetic 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 reactively installs rules to minimize the traffic handled by the controller.

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 two large applications—a security monitor that detects "scanning" attacks and a distributed key-value storage system (Memcached).

Integers	n
Rules	$r ::= \langle pat, pri, t, [a_1, \dots, a_n] \rangle$
Patterns	$pat ::= \{h_1 : n_1, \dots, h_k : n_k\}$
Priorities	$pri ::= n$
Timeouts	$t ::= n \mid None$
Actions	$a ::= \mathsf{output}(op) \mid \mathsf{modify}(h, n)$
Header Fields	$h ::=$ in_port | vlan dl_src | dl_dst | dl_type | nw_src | nw_dst | nw_proto | tp_src | tp_dst
Output Port	$op ::= n \mid$ flood | 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.

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 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 {nw_src : 192.168.0.100, tp_dst : 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.*, $[\mathsf{output}(2), \mathsf{output}(\mathsf{controller}), \mathsf{modify}(\mathsf{nw\_src}, 10.0.0.1)]$ 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 events in the network. 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 [10]. 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:

- $install(s, pat, pri, t, [a_1, \dots, a_k])$, which installs a rule with pattern pat, priority pri, timeout t, and actions $[a_1, \dots, a_n]$ 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(a). 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.

```
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.

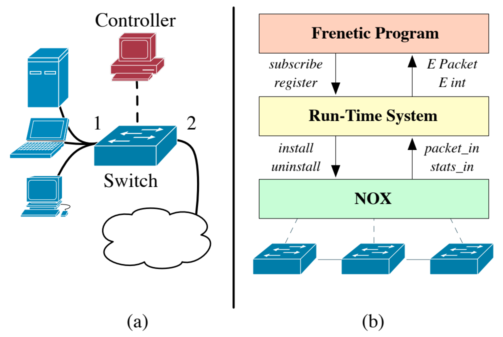

Figure 2. (a) Simple network topology (b) Frenetic architecture

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 current programming model 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 [14] and Beacon [1] suffer from similar issues because they share the same OpenFlow core. 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, pat2web, [output(1)], HIGH)
  install(s, pat2, [output(1)], DEFAULT)
  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 $\alpha \rightarrow \beta$, 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 $\rightarrow$ 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 $\rightarrow \alpha$, 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 events 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 OpenFlow/NOX model. As in the previous section, we will assume the network topology shown in Figure 2(a). 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():
  stats = Apply(Packets(), web_monitor_ef())
  Attach(stats,Print())
def web_monitor_ef():
  return (Filter(inport_p(2) & srcport_p(80)) |o|
          Lift(size) |o|
          GroupByTime(30) |o|
          Lift(sum))
```

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 it prints to the console by attaching a `Print` listener.

The `web_monitor_ef` event function is structured as the composition of several smaller event functions—the infix operator `|o|` 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) |o|
         GroupByTime(30) |o|
         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_monitor():
   register_rules(rules)
   stats = Apply(Packets(), monitor_ef())
   Attach(stats,Print())
```

The `register_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)) |o|
          Group(dstmac_g()) |o|
          RegroupByTime(60) |o|
          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. 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 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():
   register_rules(rules)
   stats1 = Apply(Packets(),web_monitor_ef())
   stats2 = Apply(Packets(),host_monitor_ef())
   Attach(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, the switch remembers the source host and ingress port that the packet came in on. Then,

```
# 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 definitions
def learning_ef():
  return (Group(srcmac_g()) |o|
          Regroup(inport_r()) |o|
          Ungroup(1,lambda n,p:p,None) |o|
          LoopPre({}, Lift(add_rule)) |o|
          Lift(complete_rules))
def learning():
  rules = Apply(Packet(),learning_ef())
  Attach(Register(),rules)
```

Figure 3. Frenetic learning switch

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

Apply $\in (\alpha$ E $\times\ \alpha\ \beta$ EF$) \rightarrow \beta$ E
Lift $\in (\alpha \rightarrow \beta) \rightarrow \alpha\ \beta$ EF
|o| $\in (\alpha\ \beta$ EF $\times\ \beta\ \gamma$ EF$) \rightarrow \alpha\ \gamma$ EF
First $\in \alpha\ \beta$ EF $\rightarrow (\alpha \times \gamma)\ (\beta \times \gamma)$ EF
Second $\in \alpha\ \beta$ EF $\rightarrow (\gamma \times \alpha)\ (\gamma \times \beta)$ EF
Merge $\in (\alpha$ E $\times\ \beta$ E$) \rightarrow (\alpha$ option $\times\ \beta$ option) E
LoopPre $\in (\gamma \times ((\alpha \times \gamma)\ (\beta \times \gamma)$ EF$)) \rightarrow \alpha\ \beta$ EF
Calm $\in \alpha\ \alpha$ EF
Filter $\in (\alpha \rightarrow$ bool$) \rightarrow \alpha\ \alpha$ 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

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

Figure 4. Core Frenetic Operators

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 route 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, the `learning_ef` event function 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 arguments 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 MAC addresses to a rule that forwards packets to that host (the helper `add_rule` inserts the rule into the dictionary). The final `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.

The top-level `learning` function applies `learning_ef` to `Packets` and 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.

4.6 Other Operators

Frenetic includes a number of additional operators, which space constraints prevent us from discussing in detail. Figure 4 lists a few of the most important operators and their types.

Queries $\quad q$ ::= Select(a) |x|
Where(qp) |x|
GroupBy$([qh_1, \ldots, qh_k])$ |x|
SplitOn$([qh_1, \ldots, qh_k])$ |x|
Every(n) |x|
Limit(n)

Aggregates $\quad a$ ::= packets | bytes | counts

Headers $\quad qh$ ::= inport | srcmac | dstmac | ethtype | vlan | srcip | dstip | protocol | srcport | dstport

Patterns $\quad qp$ ::= true_p() | qh_p(n) | qp & qp | qp | qp | qp − qp | ~ qp

Figure 5. Frenetic query syntax

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 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 Frenetic run-time 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. 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 `|x|` combines query operators. As an example, the following query generates an event that may be used in the learning switch:

```
q = (Select(packets) |x|
     GroupBy([srcmac]) |x|
     SplitOn([inport]) |x|
     Limit(1))
```

It groups packets (using `Select`) by source MAC (using `GroupBy`), splits each group when the ingress port changes (using `SplitOn`), 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():
  e = Subscribe(q)
  ef = (Ungroup(1,lambda n,p:p,None) |o|
        LoopPre({}, Lift(add_rule)) |o|
        Lift(complete_rules))
  rules = Apply(e,ef)
  Attach(rules,Register())
```

This program is simpler than the one given in the last section because the grouping and regrouping of packets is done by the query instead of the 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, the run-time will move as much processing from the controller to

```
function packet_in($packet$, $inport$)
  $isSubscribed$ := false
  $actions$ := []
  for ($query$, $event$, $counters$, $requests$) $\in$ $subscribers$ do
    if $query$.matches($packet.header$) then
      $event$.push($packet$)
      $isSubscribed$ := true
  for $rule \in rules$ 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 ($query$, $event$, $counters$, $requests$) $\in$ $subscribers$ 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 ($query$, $event$, $counters$, $requests$) $\in$ $subscribers$ do
      if $query.ready()$ then
        $counters.reset()$
        for $pattern \in flows$ do
          if $query$.matches($pattern$) then
            $xid$ := stats_request($pattern$)
            requests.add($xid$)
    sleep(1)
```

Figure 6. Frenetic run-time system handlers

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 most traffic without having to send it up to the controller.

Queries can 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) |x|
Where(srcport_p(80)) |x|
GroupBy([dstmac]) |x|
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 2(b) 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.

The core piece of the implementation is the run-time system, which sits between the high-level FRP 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. To do all of this, 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,

Currently, our implementation translates the high-level forwarding policy installed in the run-time into switch-level rules using a simple strategy that *reacts* to flows of network traffic as they occur. 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 subscribers and propagates the packet to each subscriber whose defining query depends on being provided with this packet. Next, it traverses the set of rules and collects the list of actions specified in all rules. Finally, it processes the packet in one of two ways: If there were no subscribers for the packet, then it installs a *microflow* rule that processes future packets with the same header fields on the switch. Alternatively, if there were subscribers for the packet, 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, and, by extension, the subscribers that need to be supplied with those packets. 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 aggregate 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 window for a statistics subscriber has expired. At that point, it traverses 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 id from the set of outstanding requests. When the set of outstanding requests becomes empty, it pushes the counters, which now contain the correct statistics, onto the subscriber's event stream.

Pseudocode for the NOX handlers used in the Frenetic run-time system are given in Figure 6. These algorithms describe the basic behavior of the run-time, but elide some additional complications and details with which the actual implementation has to deal. For example, if the forwarding policy changes—*e.g.*, because the `rule_listener` receives a new set of rules—the microflow rules that the run-time has installed on some of the switches in the network may be stale. To repair them, the run-time system traverses the set of flows, uninstalling stale rules and re-installing fresh ones using the actions specified in the updated policy. Of course, when the run-time uninstalls a rule on a switch due to a change in the high-level forwarding policy, the byte and packet counters associated with the switch-level rule should not be lost. Thus, the Frenetic run-time also defines a handler for `flow_removed` messages that receives the counters for uninstalled rules and adds them to the counters maintained for the subscriber on the controller. The run-time deals with several other complications, such as spurious packets that get 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.

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 implementations, 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.

The run-time's current use of microflow (exact-match) rules follows the approach of Ethane [5] and many OpenFlow-based applications [9, 11], 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.

7. Experiments

Section 4 described a *naive* version of Frenetic that leads to programs that forward all traffic to the controller. In Section 5 we described subscribe queries, which allow us to provide *optimized* functionality where programs process most packets directly in the switches. We now compare these two implementations of Frenetic to pure NOX programs. We evaluate the programs according to two metrics: *lines of code* and *total traffic* between the switch and the controller. The "lines of code" gives a sense of how much Frenetic simplifies the programmer's task, as well as the savings from code reuse in composed modules. The "total traffic" gauges how well Frenetic keeps traffic in the "fast path"—crucial to assessing the feasibility of using Frenetic in practice. We compare pure NOX, naive Frenetic, and optimized Frenetic programs using four microbenchmark experiments.

Setup We ran the experiments using the Mininet virtualization environment [15] 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.

Microbenchmarks We compared the performance of Frenetic against NOX using the following microbenchmarks:

LSW: In the learning switch (LSW) benchmark, the switch forwards packets from hosts it has learned directly but sends packets from unknown hosts to the controller. Our experiment sends ICMP echo request ("pings") between all-pairs of four hosts. This experiment evaluates the effectiveness of programs that reactively install microflow rules.

	LSW	*WSS*	*WSL*	*HHL*
Pure NOX				
Lines of Code	55	29	121	125
Controller Traffic (Bytes)	71224	1932	5300	18010
Naive Frenetic				
Lines of Code	15	7	19	36
Controller Traffic (Bytes)	120104	6590	14075	95440
Optimized Frenetic				
Lines of Code	14	5	16	32
Controller Traffic (Bytes)	70694	3912	5368	19360

Figure 7. Experimental results.

WSS: In the web statistics static (WSS) benchmark, the switch behaves like a repeater between a client and server. The forwarding rules do not change during the experiment but the controller collects HTTP request statistics every five seconds. Our experiment sends multiple HTTP requests to a single web server. This experiment evaluates the effectiveness of monitoring queries.

WSL: The web statistics learning (WSL) benchmark combines the forwarding logic from LSW with the monitoring logic from WSS. Our experiment sends multiple HTTP requests from three hosts to a single web server. This experiment illustrates the composition of two modules and the effectiveness of monitoring queries in a dynamic forwarding context.

HHL: The "heavy hitters" learning (HHL) benchmark combines the forwarding logic from LSW with a monitoring program that computes the top-k source MACs by total bytes sent. Our experiment sends a varying number of ICMP echo requests from three hosts, which are measured and tabulated on the controller. This experiment illustrates the behavior of a more sophisticated monitoring application.

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. We excluded the initial handshake, which is common to all switch-controller combinations.

Results The results of our experiments, shown in Figure 7, demonstrate that naive Frenetic achieves significant savings in code complexity over NOX, but sends more traffic to the controller. The data shows that naive Frenetic does not scale well. However, we also see that optimized Frenetic *does* perform comparably with the pure NOX implementations of each benchmark. Optimized Frenetic's performance in the *WSS* benchmark is an outlier because the benchmark exploits an inherent weakness in a reactive flow installation strategy. While the pure NOX program can install permanent static forwarding rules in the switch immediately after coming online, Frenetic's run-time waits for the switch to send a packet to the controller before consulting the forwarding rules that a program registers, even if they are static in nature.

Conversely, we also see in *LSW* where the optimized Frenetic implementation actually (slightly) outperforms the traditional NOX implementation in terms of traffic between the switch and the controller. To deal with the two-tiered architecture of OpenFlow and the finite size of the flow table, many NOX programs use flow timeouts to evict installed flows from the switch. Using timeouts ensures that installed rules "self-destruct" without the programmer needing to perform extra bookkeeping to remember all of the installed rules. However, such timeouts result in additional packets being sent to the controller for subsequent flow setups. In contrast,

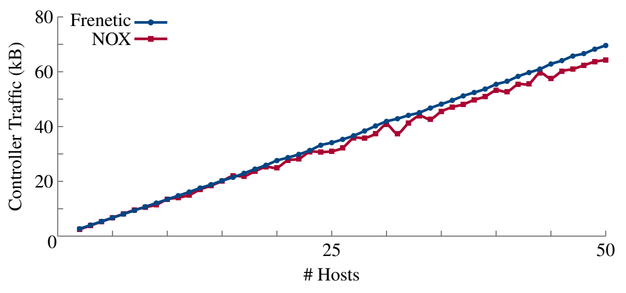

Figure 8. Controller traffic as number of hosts grows.

Frenetic's run-time system can react to changes in the forwarding logic and manage the set of installed rules, obviating the need for short flow timeouts, without burdening the programmer.

Scalability In addition to the micro-benchmarks we measured, we also conducted a host scalability analysis against a single benchmark to evaluate whether or not Frenetic programs could continue performing comparably to traditional NOX programs as the number of devices in the network grows. In the *WSL* experiment, we varied the number of hosts fetching web resources on a single switch. The results shown in Figure 8 demonstrate that not only does optimized Frenetic perform comparably with pure NOX in this benchmark, but it also *scales* with the pure NOX implementation.

8. Case Studies

This section describes two more substantial network applications we have developed in Frenetic. The first is a monitoring application that detects network scans as they occur and adaptively blocks the scanning host from sending additional packets into the network. The second is an in-network query router for the Memcached key-value store that makes it possible to seamlessly add and remove servers without rebooting clients.

Detecting and Blocking Scanning Traffic Malicious users often scan the the hosts in a network to identify machines that are vulnerable to attack. Although scanning also has legitimate uses, many operators block scans to prevent unknown users from probing their network. Using Frenetic, we implemented a proof-of-concept scan detector that can be composed with the learning switch from Section 4 to obtain a switch that adaptively responds to scans by blocking offending hosts from sending packets on the network.

The high-level architecture of the detector is shown in Figure 9(a). The LearningSwitch event function at the top-left of the diagram subscribes to an event with source MACs and produces an event carrying rules. The ScanDetect event function at the top-right subscribes to an event with pairs of source-destination MACs and produces a set of suspected scanners. We currently use a simple strategy to detect scanners—maintaining a table that counts the number of unique destination hosts contacted recently by each source and considering a host a scanner if the number exceeds a threshold—but plan to explore more sophisticated strategies in the future. Next, the Merge event function combines the rules and scanners and the FilterScanners event function transforms the rules to another set in which scanning hosts are explicitly prevented from sending traffic on the network. The Register operator at the bottom of the diagram represents a listener that registers this final set of rules with the run-time.

This application demonstrates how Frenetic facilitates building large programs in a compositional way. In particular, we did not have to make *any* modifications to the learning switch—were able to use it "off the shelf." It also shows the benefits of being able to

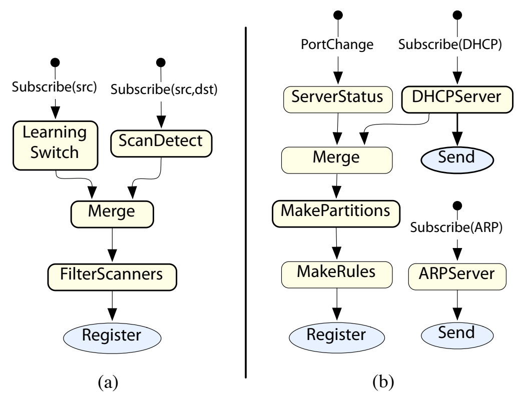

Figure 9. (a) Scan Detector (b) Memcached Request Router

subscribe to events identified by different, overlapping queries as manually crafting rules for these subscribers would be tedious and difficult to get right.

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 between the servers. The current Memcached 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.

We developed a novel "plug-and-play" solution to this problem in Frenetic that adapts dynamically to server churn. We introduce a layer of indirection between the clients and servers: clients are configured with a large number of *virtual* addresses and an OpenFlow switch translates between these virtual addresses and server's physical addresses. When a server fails, the controller program reassigns its virtual addresses to another server; when a new server becomes available, a virtual addresses belonging to some other server is remapped to it. As this solution works entirely in the network, it can be used with existing, unmodified servers and clients.

Figure 9(b) depicts the high-level structure of the Frenetic Memcached application. The DHCPServer event function at the top right of the figure implements a DHCP server that subscribes to DHCP requests, generates responses, and emits them onto the network using a Packet listener. The DHCP server also generates an event with tuples of the form (p, m, a), where p is a physical switch port, m is a MAC address, and a is the IP address assigned to m. The ServerStatus event function at the top left monitors PortChange events and produces an event with sets of active physical ports. This event is merged with the DHCP event and the result is supplied to the MakePartitions function, which reconciles the set of known servers with the set of active ports and generates a event with the mapping between virtual and physical addresses. MakeRules converts the event with the current partitioning into the forwarding policy, which is installed in the run-time using a Register listener. In the future, we plan to add a heartbeat mechanism to help cope with soft failures, where servers have not crashed outright but applications are unresponsive. 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 [8, 20, 22, 23]. Its push-based implementation is based on FrTime [6]. The key differences between

Frenetic and these previous languages are in the application domain (networking as opposed to animation, robotics and others) 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 [18] language applies FRP in a staged language to assemble efficient programs for sensor networks.

The most similar language to Frenetic is Nettle [25]. 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 [16, 17]. 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 [13]. 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 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 [7], a stream database for monitoring networks. In terms of expressiveness, Gigascope is more limited than Frenetic, as 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 handles all of the details related to installing and uninstalling switch-level rules.

We are currently working to extend Frenetic in several directions. We are developing applications for a variety of tasks including load balancing, authentication and access control, and a framework inspired by FlowVisor [24] 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. Finally, we are exploring a proactive strategy that generates rules from the registered subscribers and forwarding rules eagerly. We plan to compare the tradeoffs between different rule generation strategies empirically.

Acknowledgments We wish to thank Matt Meola and Minlan Yu for many helpful discussions. Our work is supported by ONR grant N00014-09-1-0770 *Networks Opposing Botnets*. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the ONR.

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] 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.

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

[7] 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.

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

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

[10] 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.

[11] 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*.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

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

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

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

[24] 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.

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