Frenetic: A Network Programming Language

Nate Foster
Cornell University

Rob Harrison
Princeton University

Michael J. Freedman
Princeton University

Christopher Monsanto
Princeton University

Jennifer Rexford
Princeton University

Alec Story
Cornell University

David Walker
Princeton University

Abstract

Modern networks provide a variety of interrelated services including routing, traffic monitoring, load balancing, and access control. Unfortunately, the languages used to program today's networks lack modern features—they are usually defined at the low level of abstraction supplied by the underlying hardware and they fail to provide even rudimentary support for modular programming. As a result, network programs tend to be complicated, error-prone, and difficult to maintain.

This paper presents Frenetic, a high-level language for programming distributed collections of network switches. Frenetic provides a declarative query language for classifying and aggregating network traffic as well as a functional, reactive combinator library for describing high-level packet-forwarding policies. Unlike prior work in this domain, these constructs are—by design—fully compositional, which facilitates modular reasoning and enables code reuse. This important property is made possible by Frenetic's novel run-time system which manages all of the details related to installing, uninstalling, and querying low-level packet-processing rules on physical switches.

Overall, this paper makes three main contributions: (1) We analyze the state-of-the art in languages for programming networks and identify the key limitations; (2) We present a language design that addresses these limitations, using a series of examples to motivate and validate our choices; (3) We describe our implementation and evaluate its performance on several benchmarks.

1. Introduction

Today's networks consist of hardware and software components that are closed and proprietary. The difficulty of changing these components has had a chilling effect on innovation, and forced network administrators to express policies through complicated and frustratingly brittle interfaces. As discussed in recent a *New York Times* article [31], the rise of data centers and cloud computing have brought these problems into sharp relief and led a number of networks researchers to reconsider the fundamental assumptions that underpin today's network architectures.

In particular, significant momentum has gathered behind OpenFlow, a new platform that opens up the software that controls the network but still allows packets to be processed using fast, commodity switching hardware [32]. OpenFlow defines a standard interface for installing flexible packet-forwarding rules on physical network switches. These rules are installed by a programmable *controller* that runs separately on a stock machine. The most well-known controller platform is NOX [21], though there are several others [1, 9, 26, 40]. OpenFlow is supported by a number of commercial Ethernet switch vendors, and has been deployed in several campus and backbone networks. Already, researchers have created a variety of controller applications that introduce new network functionality, like flexible access control [10, 34], Web server load balancing [22, 41], energy-efficient networking [23], and seamless virtual-machine migration [19].

Unfortunately, while OpenFlow and NOX now make it *possible* to implement exciting new network services, they do not make it *easy*. Programmers constantly grapple with several challenges.

First, networks often perform multiple tasks, like routing, access control, and traffic monitoring. Unfortunately, decoupling these tasks from each other and implementing them independently in separate modules is effectively impossible, since packet-handling rules (un)installed by one module often interfere with overlapping rules (un)installed by other modules.

Second, the OpenFlow/NOX interface is defined at a very low level of abstraction. For example, the OpenFlow rule algebra directly reflects the capabilities of the switch hardware (*e.g.*, bit patterns and integer priorities). Simple high-level concepts such as set difference require multiple rules and priorities to implement correctly. In addition, more powerful "wildcard" rules are a limited hardware resource that the programmer must manage by hand.

Third, controller programs only receive events for packets the switches do not know how to handle. Hence, code that installs a forwarding rule might prevent another, different event-driven callback from being triggered. Consequently, OpenFlow/NOX programming quickly becomes a difficult exercise in *two-tiered* programming in which the programmer must simultaneously reason about packets processed on switches and on the controller.

Fourth, because a network of switches is a distributed system, it is susceptible to various different kinds of race conditions. For example, a common NOX programming idiom is to handle the first packet of each network flow on the controller and install switch-level rules to handle the remaining packets. However, such programs can be susceptible to errors if the second, third, or fourth packets in a flow arrive before the appropriate switch-level rule is computed and installed on the switches in the network.

To address these challenges, we present Frenetic, a new programming language for 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) and are carefully constructed to support the following key principles:

Declarative Design. Where possible, we consider *what the programmer* might want, rather than *how the hardware* implements it. Hence, in many cases, we provide intuitive, high-level primitives, even though they are not directly supported by the hardware.

Modular Design. We have designed Frenetic's primitives to have *limited network-wide effects* and semantics that can be stated *independently of the context in which they are used*. This facilitates building modular programs with reuseable parts.

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	$a ::=$ `output`$(op) \mid$ `modify`(h, n)
Header Fields	$h ::=$ `in_port` $\mid$ `vlan` $\mid$ `dl_src` $\mid$ `dl_dst` $\mid$ `dl_type` $\mid$ `nw_src` $\mid$ `nw_dst` $\mid$ `nw_proto` $\mid$ `tp_src` $\mid$ `tp_dst`
Output Port	$op ::= n \mid$ `flood` $\mid$ `controller`

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

Single-tier Programming. Frenetic programmers do not have to worry that installing packet-handling rules may prevent the controller from analyzing other traffic. On the contrary, Frenetic supports a *see-every-packet abstraction* which guarantees every packet is available for analysis, thereby side-stepping the many complexities of today's two-tiered programming model.

Race-free Semantics. Because Frenetic queries supply the run-time system with information about *what programmers want*, the run-time can suppress superfluous packets that arrive at the controller due to network race conditions. Automatic race detection and packet suppression simplifies the programming model.

Cost Control. In general, a danger of adopting high-level, declarative features is that it may difficult for users to understand or control the computational costs of the abstractions they use. To avoid this pitfall, Frenetic gives programmers guidance concerning the costs of programming constructs. In particular, the query language is carefully defined so that the core query logic can be executed on network switches, thereby keeping most packets in the fast path.

The above principles make Frenetic programs robust, compact, easy to write, easy to understand and easy to modify. Hence, to summarize, this paper makes the following contributions:

- Analysis of OpenFlow/NOX difficulties (Section 3): Using our combined expertise in programming languages and networks, we identify weaknesses of the current model that modern programming language principles can overcome.
- Frenetic language design (Section 4): Applying ideas from the disparate fields of database query languages and functional reactive programming, we present and analyze our design choices for Frenetic, a language for programming OpenFlow networks.
- Frenetic implementation (Section 5): We describe Frenetic's implementation architecture, paying particular attention to the run-time system—the enabling technology that allows us to lift the level of abstraction without sacrificing performance.
- Evaluation (Section 6): We implement several applications in Frenetic and NOX and compare them on several metrics: lines of code, controller load, and total traffic. The results demonstrate that Frenetic programs are more concise than their NOX counterparts and yet achieve comparable performance.

2. Background on OpenFlow and NOX

This section presents the main features of OpenFlow and NOX. 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 specification document [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.

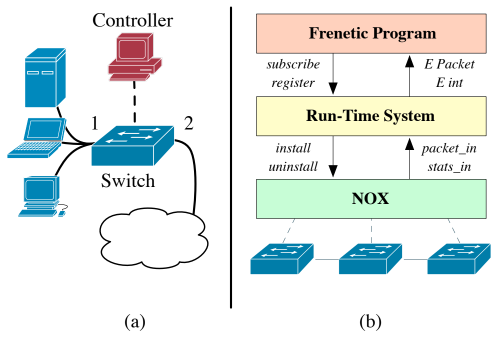


Figure 2. (a) Simple network topology (b) Frenetic architecture

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 $\{$`nw_src`$: 10.0.0.1,$ `tp_dst`$: 80\}$ matches packets from source IP address `10.0.0.1` going to destination port `80`. We use standard notation for values in common header fields—*e.g.*, writing "`10.0.0.1`" instead of "`167772161`." Any header fields not appearing in a pattern are unconstrained. We call rules with unconstrained fields *wildcard rules*.

OpenFlow Switches. When a packet arrives at a switch, the switch processes the packet in three steps:

1. 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 processing. Otherwise, if there are multiple matching rules, it picks the *exact-match* rule (*i.e.*, the rule whose pattern matches every header field in the packet) if one exists, or a wildcard rule with highest priority if not.
2. It updates the byte and packet counters associated with the rule.
3. It applies each of the actions listed in the rule to the packet, or drops the packet if the list is empty.

The action `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 it out on all physical ports (except the ingress port) and `controller` sends it to the controller. The action `modify`(h, n) instructs the switch to rewrite the header field h to n. The list of actions may contain both output and modify actions—*e.g.*, $[$`output`$(2),$ `output`$($`controller`$),$ `modify`$($`nw_src`$, 10.0.0.1)]$ outputs packets on switch port 2 and to the controller, and also rewrites their source IP address fields 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 [21]. NOX defines a number of events:

- *packet_in*(*switch*, *port*, *packet*), triggered when *switch* forwards a *packet* received on physical *port* to the controller;
- *stats_in*(*switch*, *xid*, *pattern*, *packets*, *bytes*), triggered when *switch* returns the *packets* and *bytes* counters in response to a request for statistics about rules contained in *pattern*. The *xid* parameter represents an identifier for the request.
- *flow_removed*(*switch*, *pattern*, *packets*, *bytes*), triggered when a rule with *pattern* exceeds its timeout and is removed from *switch*'s flow table. The *packets* and *bytes* parameters contain the values of the counters for the evicted rule.
- *switch_join*(*switch*), triggered when *switch* joins the network.
- *switch_exit*(*switch*), triggered when *switch* exits the network.
- *port_change*(*switch*, *port*, *up*), triggered when the link attached to a given physical *port* on *switch* goes up or down. The *up* parameter represents the new status of the link.

NOX also provides functions for sending messages to switches:

- *install*(*switch*, *pattern*, *priority*, *timeout*, *actions*), installs a rule with the given *pattern*, *priority*, *timeout*, and *actions* in the flow table of *switch*.
- *uninstall*(*switch*, *pattern*), removes all rules contained in *pattern* from the flow table of *switch*.
- *send*(*switch*, *packet*, *action*), sends the given *packet* to *switch* and applies *action* to it there.
- *query_stats*(*switch*, *pattern*), issues a request for statistics from all rules contained in *pattern* on *switch* and returns a request identifier *xid* that can be used to match up the asynchronous response from the switch.

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

Example. To illustrate the use of OpenFlow, consider a controller program written in Python that implements a simple repeater hub. 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 `switch_join` handler below invokes the `repeater` when the switch joins the network. The `repeater` function installs rules on switch `s` that instruct the switch to forward packets from port 1 to port 2 and vice versa.

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

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. This section presents four of the most substantial difficulties that arise when writing programs for OpenFlow/NOX. For concreteness, we focus on the NOX controller but other OpenFlow controllers such as Onix [26], Beacon [1], and Nettle [40] suffer from similar issues.

3.1 Interactions Between Concurrent Modules

The first issue is that NOX programs do not compose. Suppose that we want to extend the repeater hub 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(switch):
  pat = {in_port:2,tp_src:80}
  install(switch, pat, DEFAULT, None, [])
  query_stats(switch, pat)
def stats_in(switch, xid, pattern, packets, bytes):
  print bytes
  sleep(30)
  query_stats(switch, pattern)
```

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(switch):
  repeater(switch)
  monitor(switch)
```

Unfortunately, naively composing the two programs in this way will *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 will not update the counters needed for monitoring, while using the `monitor` rule will break the repeater program because its list of actions is empty (*i.e.*, packets 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(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  pat2web = {in_port:2, tp_src:80}
  install(switch, pat1, [output(2)], DEFAULT)
  install(switch, pat2web, [output(1)], HIGH)
  install(switch, pat2, [output(1)], DEFAULT)
  query_stats(switch, 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 NOX 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 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 express a logical "difference" of patterns, but OpenFlow patterns can only directly express positive constraints. Thus, 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(switch):
  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(switch, pat1, DEFAULT, None, [output(2)])
  install(switch, pat2srv, HIGH, None, [output(1)])
  install(switch, pat2web, MEDIUM, None, [output(1)])
  install(switch, pat2, DEFAULT, None, [output(1)])
  query_stats(switch, 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.

Describing packets using the low-level patterns supported by OpenFlow switches is cumbersome and error-prone. It forces programmers to use multiple rules and priorities to encode patterns that could be easily expressed using natural operations such as negation, difference, and union. It adds unnecessary clutter to programs 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. 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.

```
def repeater_monitor_hosts(switch):
  pat = {in_port:1}
  install(switch, pat, DEFAULT, None, [output(2)])
def packet_in(switch, inport, packet):
  if inport == 2:
     mac = dstmac(packet)
     pat = {in_port:2, dl_dst:mac}
     install(switch, pat, DEFAULT, None, [output(1)])
     query_stats(switch, pat)
```

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 port 2 to the controller. This causes the `packet_in` handler to be invoked; it processes each packet by installing a rule that handles future packets. Note that the controller only sees one incoming packet per host—the rule processes 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.

Queries	q ::=	`Select`(a) `*`
		`Where`(fp) `*`
		`GroupBy`($[qh_1, \ldots, qh_k]$) `*`
		`SplitWhen`($[qh_1, \ldots, qh_k]$) `*`
		`Every`(n) `*`
		`Limit`(n)
Aggregates	a ::=	`packets` | `sizes` | `counts`
Headers	qh ::=	`inport` | `srcmac` | `dstmac` | `ethtype` | `vlan` | `srcip` |
		`dstip` | `protocol` | `srcport` | `dstport` | `switch`
Patterns	fp ::=	`true_fp`() | qh`_fp`(n) | `and_fp`($[fp_1, \ldots, fp_n]$) |
		`or_fp`($[fp_1, \ldots, fp_n]$) | `diff_fp`(fp_1, fp_2) |
		`not_fp`(fp)

Figure 3. Frenetic query syntax

Although NOX 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.

3.4 Network Race Conditions

One of the corollaries of NOX's explicit two-tier programming model is that programs are susceptible to subtle network race conditions. For example, a common NOX programming idiom is to analyze the first packet of every flow and calculate an action to apply to all future packets in the same network flow. In fact, this is how the `repeater_monitor_hosts` example described in the previous subsection worked. Unfortunately, our statement that the `packet_in` handler "processes each packet by installing a rule that handles *all future packets* to the same host" was a simplification. The installed rule usually handles all future packets—but not always! If a new packet in the same flow arrives before the switch has been able to install the new rule, that new packet will also be sent up to the controller. Consequently, if the controller routines are not carefully crafted to be idempotent when receiving multiple unexpected packets in the same flow, they will fail.

4. Frenetic Language Design

Frenetic is a new, domain-specific language for programming OpenFlow networks. It is embedded in Python and comprises two integrated sublanguages: (1) a limited, but high-level and declarative *network query language*, and (2) a general-purpose, functional and reactive *network policy management library*. The language offers a number of features that make programming more convenient including a single-tier, "see-every-packet" abstraction; strong compositionality properties; a clear cost model; and a simple, race-free semantics. In the following subsections, we present the main features of the language and explain its semantic properties in detail.

4.1 The Network Query Language

The network query sublanguage allows Frenetic programs to *read* the state of network. To implement these reads efficiently, the Frenetic run-time system changes the state of the network by installing a variety of low-level rules on switches. However, from the high-level, abstract viewpoint of the Frenetic programmer, these reads and their implementation have no observable effect on network state. As a result, queries compose perfectly—both with each other and with the operations in the policy management library.

The key challenge in the design of Frenetic's query sublanguage involves finding a balance between expressiveness, simplicity, and control over cost. For example, the cost of evaluating a query can be defined as the number of packets that must be diverted from the fast path in the network and processed on the controller. Managing this cost is important because the latency of processing a diverted packet is orders of magnitude worse than processing it in hardware. Moreover, if many packets are diverted, the link between the switches and controller can become a bottleneck. Consequently, we deliberately limit the expressiveness of Frenetic query language to ensure that it has a simple, easy-to-understand cost model programmers can depend on.

Basic Concepts. Frenetic queries include constructs for *filtering* the set of all packets in the network using high-level patterns, subdividing this set by *grouping* on the basis of one or more header fields, further *splitting* these sets by arrival time or whenever a header field changes value, *limiting* the number of values returned, and *aggregating* by number or size of packets. The result produced by a query is an *event stream*—a data structure that represents an infinite, discrete, time-indexed stream of *values*. Though Frenetic is embedded in Python, an untyped language, it is useful to understand the types of events and event-driven programs.[1] The type α E denotes events carrying values of type α. For example, packet E is an event of packets and (switch $\times$ int) E is an event of pairs of switch identifiers and integers.

The syntax of Frenetic queries is given in Figure 3. Each top-level clause is optional, except for the Select, which identifies the type of event returned by the query—an event carrying packets, byte counts, or packet counts. In Python code, we use the infix operator * to combine clauses. We briefly explain the main syntactic elements below and follow up with illustrative examples.

A Select(a) clause aggregates the results returned by the rest of the query using method a, where a may be one of packets (return the packets themselves), counts (return the number of packets) or bytes (return the sum of the sizes of the packets).

A Where(*fp*) clause filters the results, retaining only those packets satisfying the *filter pattern fp*. Simple query patterns define sets of packets on the basis of packet header fields such as switch (switch), port (inport), source MAC address (srcmac), destination IP address (destip) and others. More complicated filter patterns can be constructed using natural set-theoretic operations such as intersection (and_fp), union (or_fp), difference (diff_fp), and complement (not_fp). These high-level patterns are compiled to OpenFlow-representable patterns by Frenetic.

A GroupBy([$qh_1, \ldots, qh_n$]) clause subdivides the set of queried packets into subsets based on header fields qh_1 through qh_n. For example, grouping by srcip and srcport results in one subset for all packets with source IP 10.0.0.1 and TCP source port 80, a second subset for all packets with source IP 10.0.0.2 and TCP source port 80, a third subset for all packets with source IP 10.0.0.1 and source port 21, *etc.*

A SplitWhen([$qh_1, \ldots, qh_n$]) clause, like a GroupBy, subdivides the set of selected packets into subsets. However, whereas GroupBy produces *one* subset for *all* packets with particular values for the given header fields, SplitWhen does not—it generates a new subset each time the value of one of the given fields *changes*. For example, suppose a query splits on source IP address, and packets with source IPs 10.0.0.1, 10.0.0.2 and 10.0.0.1 arrive in sequence. In this case, SplitWhen generates three subsets (the first and third packets are put in separate sets, because their IP addresses differ from the address of the preceding packet). If the arrival order was different, perhaps 10.0.0.1, 10.0.0.1, 10.0.0.2, then only two subsets would be generated.

An Every(n) clause partitions packets by time, grouping packets that arrive within the same n-second window together.

Finally, a Limit(n) clause limits the number of packets in each subset to n. The most common limit is 1.

Example Query. To get a taste of the Frenetic query language, consider the following web monitoring query, designed for the single-switch repeater network presented in the previous section.

```
def web_query():
  return \
    (Select(sizes) *
     Where (and_fp([inport_fp(2),srcport_fp(80)])) *
     Every(30))
```

When installed in the run-time system, this query selects all packets arriving on physical port 2 and from TCP source port 80. It sums the sizes of all such packets every 30 seconds and returns an event of type Int as a result.

The results of such queries may be used in a variety of ways in Frenetic programs—for traffic analysis, for security monitoring and for decisions about the forwarding policy. For now, all we will do is pipe the results to a printer:

```
def web_stats():
  web_query() >> Print()
```

Query Composition. To illustrate the modularity properties of Frenetic programs, let us carry the example a step further and extend it to monitor incoming traffic by host. As shown in Section 3.1, implementing this program in NOX is 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. Extending the Frenetic program, however, is simple. The following query summarizes the total volume of traffic arriving on physical port 2, grouped by destination host, every 60 seconds.

```
def host_query():
  return (Select(sizes) *
          Where(inport_fp(2)) *
          GroupBy([dstmac]) *
          Every(60))
```

This query may be composed with the web query using the Merge operator, a generic combinator that transforms a pair of events into an event of pairs.

```
def all_stats():
  Merge(host_query(), web_query()) >> Print()
```

The programmer who writes this program need not know the details of the individual query routines, as neither query can interfere with the results produced by the other. Why is that? Unlike NOX, Frenetic supports the abstraction that queries merely read network state and do not modify it (even though the underlying run-time system will, in fact, modify the state of the network by installing rules on switches). Moreover, by design, Frenetic supports a programming model in which every query can "see every packet" in the network. Thus, installing one query in the run-time does not silently inhibit any other queries from seeing certain packets. Lastly, note that the host query and the web queries operate at different frequencies—60 seconds vs. 30 seconds. Implementing this functionality in Frenetic is as easy as declaring the desired intervals. Implementing it in NOX, on the other hand, would be difficult, as the programmer would have to code tedious bookkeeping routines in event handlers to keep track of which statistics to collect at which times. Frenetic's run-time system does this bookkeeping automatically. Hence, our

[1] Though it is not central to this paper, we have implemented a dynamic typechecker for Frenetic that checks the types of operators dynamically.

design has changed query composition from a challenging, error-prone enterprise to a completely trivial one.

Race-Free Semantics. One of the most basic network programs is a *learning switch*, which discovers the identity of the hosts connected to each of its ports by recording the source MAC addresses contained in incoming packets. The following query could be used to implement the core functionality of a simple learning switch:

```
def learning_query():
  return (Select(packets) *
          Where(true_fp()) *
          GroupBy([srcmac]) *
          SplitWhen([inport]) *
          Limit(1))
def connection_printer():
  learning_query() >> Print()
```

When `learning_query` is executed, it generates an event that includes one packet for each distinct source MAC, unless the port associated with that source MAC changes (which might happen if a host, such as a laptop, were to move). This program is unremarkable except that it prints each new connection that it discovers exactly once because the query is limited to return *one* packet. Achieving the same effect in NOX is surprisingly tricky because of network race conditions. In the time it takes for a NOX program to generate and install a rule to suppress packets 2, 3, 4 with the same source MAC, those packets might already have arrived at the switch, be en route to the controller and be about to be processed by the handler. Consequently, the NOX programmer will have to remember to implement complex, error-prone bookkeeping if she wants to get it right. Such races affect the implementation of the Frenetic run-time system as well, but they are handled invisibly (and once-and-for-all) at that level, and are not exposed to the programmer. Unfortunately, the NOX implementation cannot mimic Frenetic here as it does not have access to the same high-level, semantic information expressed in the queries that allows Frenetic to squash superfluous packets.

The Query Cost Model. The cost of executing a Frenetic query can be understood in terms of *microflows*—*i.e.*, sets of related packets that share OpenFlow attributes (*e.g.*, having identical header fields, arriving at the same switch, and occurring within the same time window). Given a stable forwarding policy, the cost of a query is proportional to the sum of the number of microflows that appear in the network[2] plus the number of packets that contribute to the result. Moreover, if multiple, overlapping queries are are installed in the system, they share costs—only one packet per microflow is required independently of the number of queries. These costs follows from the microflow-based implementation strategy used by the Frenetic run-time system, which is described in Section 5.

Deep Packet Inspection. To implement deep packet inspection in Frenetic, one only needs to write a query that returns the packets to inspect—*e.g.*, the following query returns all web traffic:

```
def web_packets_query():
  return (Select(packets) *
          Where(srcport_fp(80)))
def dpi():
  web_packets_query() >> analyze_packet()
```

Of course, diverting a large number of packets to the controller is likely to overwhelm the system. However, this is not a limitation of the Frenetic design, it is a limitation of the popular OpenFlow platform on which Frenetic sits. In the future, OpenFlow switches may well be extended to allow efficient querying of additional bits

[2] Typically, there will be exactly one packet per microflow, but due to network race conditions, there may be a few more.

Events		
`Seconds`	$\in$	$\mathsf{int\ E}$
`SwitchJoin`	$\in$	$\mathsf{switch\ E}$
`SwitchExit`	$\in$	$\mathsf{switch\ E}$
`PortChange`	$\in$	$(\mathsf{switch} \times \mathsf{int} \times \mathsf{bool})\ \mathsf{E}$
`Once`	$\in$	$\alpha \to \alpha\ \mathsf{E}$
Basic Event Functions		
`>>`	$\in$	$\alpha\ \mathsf{E} \to \alpha\ \beta\ \mathsf{EF} \to \beta\ \mathsf{E}$
`Lift`	$\in$	$(\alpha \to \beta) \to \alpha\ \beta\ \mathsf{EF}$
`>>`	$\in$	$\alpha\ \beta\ \mathsf{EF} \to \beta\ \gamma\ \mathsf{EF} \to \alpha\ \gamma\ \mathsf{EF}$
`ApplyFst`	$\in$	$\alpha\ \beta\ \mathsf{EF} \to (\alpha \times \gamma)\ (\beta \times \gamma)\ \mathsf{EF}$
`ApplySnd`	$\in$	$\alpha\ \beta\ \mathsf{EF} \to (\gamma \times \alpha)\ (\gamma \times \beta)\ \mathsf{EF}$
`Merge`	$\in$	$(\alpha\ \mathsf{E} \times \beta\ \mathsf{E}) \to (\alpha\ \mathsf{option} \times \beta\ \mathsf{option})\ \mathsf{E}$
`BlendLeft`	$\in$	$\alpha \times \alpha\ \mathsf{E} \times \beta\ \mathsf{E} \to (\alpha \times \beta)\ \mathsf{E}$
`BlendRight`	$\in$	$\beta \times \alpha\ \mathsf{E} \times \beta\ \mathsf{E} \to (\alpha \times \beta)\ \mathsf{E}$
`Accum`	$\in$	$(\gamma \times (\alpha \times \gamma \to \gamma) \to \alpha\ \gamma\ \mathsf{EF}$
`Filter`	$\in$	$(\alpha \to \mathsf{bool}) \to \alpha\ \alpha\ \mathsf{EF}$
Listeners		
`>>`	$\in$	$\alpha\ \mathsf{E} \to \alpha\ \mathsf{L} \to \mathsf{unit}$
`Print`	$\in$	$\alpha\ \mathsf{L}$
`Register`	$\in$	$\mathsf{policy\ L}$
`Send`	$\in$	$(\mathsf{switch} \times \mathsf{packet} \times \mathsf{action})\ \mathsf{L}$
Rules and Policies		
`Rule`	$\in$	$\mathsf{pattern} \times \mathsf{action\ list} \to \mathsf{rule}$
`MakeForwardRules`	$\in$	$(\mathsf{switch} \times \mathsf{port} \times \mathsf{packet})\ \mathsf{policy\ EF}$
`AddRules`	$\in$	$\mathsf{policy\ policy\ EF}$

Figure 4. Selected Frenetic Operators.

of every packet in hardware. When such extensions are available, we anticipate it will be straightforward to extend the Frenetic query language to support deep packet inspection efficiently. For now, to maintain a clear cost model for Frenetic queries—*i.e.*, one where cost depends on the number of microflows, not the number of packets in a microflow (except for packets returned by the query)—we do not support deep packet inspection in queries themselves.

Summary. The Frenetic query language supports a collection of orthogonal, high-level query operators. The Frenetic run-time system supports the abstraction that these operators read, but do not modify network state. The key consequence of this abstraction is that queries compose seamlessly with one another. The Frenetic run-time system also suppresses superfluous packets that occur due to race conditions in the underlying network, giving queries a simple race-free semantics. Finally, Frenetic queries have a simple, clear cost model that depends primarily on the number of flows, not the number of packets within a flow.

4.2 The Network Policy Management Library

Frenetic programmers manage the policy that governs the forwarding of packets through the network using a combinator library for functional, reactive programming (FRP). The library design is inspired by Yampa [13] (a language for programming robots) and its implementation is based on the strategy used in FlapJax [33] (a library for web programming). However, there is still significant novelty in applying these old ideas to a new domain. In addition, the interaction between Frenetic's query language, its representation of network state in the run-time system, and its library of FRP combinators, all required careful design.

Basic Concepts. One of the basic operations performed by a Frenetic program is to construct packet-forwarding *rules* for installation on switches. These rules are created using the `Rule` constructor, which takes a *pattern* and a list of *actions* as arguments. Patterns are similar to the filter patterns used in the query language—the only difference is that rule patterns do not mention switches. Actions include *forwarding* through a particular port p (`forward(`p`)`), flooding through all ports (`flood()`), sending the

```
# query one packet per distinct source IP
def src_ips() =
  return (Select(packets) *
          Where(inport_fp(1)) *
          GroupBy([srcip]) *
          Limit(1))

# add switch to pair
def add_switch(port,packet):
  return (switch(header(packet)),port,packet)

# parameterized load balancer
def balance(balancer):
  return \
    (src_ips()          >> # ip × packet E
     ApplyFst(balancer) >> # port × packet E
     Lift(add_switch)   >> # switch × port × packet E
     MakeForwardRules() >> # incremental policy per ip
     AddRules())           # complete policy
```

Figure 5. A Parameterized Load Balancer

packet to the controller (`controller()`), and modifying header field f to a new value v (`modify(`f`,`v`)`). There is no explicit drop action. The empty list is interpreted as a directive to drop packets.

To associate rules with switches, Frenetic programs must create *network policies*. We represent policies in Python as dictionaries mapping switches to lists of rules.

Frenetic programs control the installation of policies in a network *over time* by generating *policy events*. Policy events are infinite, time-indexed streams of values, just like the events generated from queries that we saw in the previous subsection. In addition to policy events and query-generated events, Frenetic also contains the primitive events `Seconds`, which contains the number of seconds since the epoch, `SwitchJoin` and `SwitchExit`, which contains the identifiers of switches joining or leaving the network, and `PortChange`, which contains triples comprising a switch, a port number, and a boolean value. In this last event, the boolean value indicates whether the given port on the switch is enabled.

Frenetic also contains *Listeners*, which represent event consumers. One example of a listener is the primitive `Print` listener, which consumes string events by printing them to the console. Another example is the `Send` listener, which consumes (switch, packet, action list) events by sending each packet to the switch and applying the actions to it there. The `Register` listener applies a network policy to a network. The type of listeners of events α E is written α L.

Frenetic programs analyze or transform events using *event functions*. The type of event functions from α E to β E is written α β EF. Many such event functions are based on standard operators found in previous work on FRP. For example, `Merge`, which we saw in previous sections, transforms a pair of events into an event of pairs. `Lift(`f`)` transforms an ordinary function f of type $(\alpha \rightarrow \beta)$ into an event function of type α β EF that applies f to each value in its input event. Frenetic also supplies a derived library of event functions useful specifically in a networking context. For example, `MakeForwardRules` converts an event of (switch, port number, packet) triples into a forwarding policy that forwards packets with the same header out the given port. `AddRules` folds over the values in its incoming policy event by repeatedly merging the policies it receives and returning, at each time step, the total accumulated policy so far.

In the following paragraphs, we will further explain these concepts using examples. For reference, Figure 4 lists a selected set of the most important Frenetic operators and their types. Note that the composition operator `>>` is conveniently overloaded to work with events, event functions, and listeners.

A First Example. The simplest forwarding program just installs static packet-forwarding rules. The Frenetic program below mimics the NOX repeater hub presented in Section 3:

```
rules = [Rule(inport_fp(1), [forward(2)]),
         Rule(inport_fp(2), [forward(1)])]
def repeater():
  (SwitchJoin() >>
   Lift(lambda switch:{switch:rules}) >>
   Register())
```

The network policy in this program contains two rules. The first matches all packets arriving at port 1 and forwards them out port 2. Conversely, the second matches packets arriving on port 2 and forwards them out port 1. The `repeater` function passes the `SwitchJoin` event stream to a lifted function that builds an event carrying dictionaries with switches as keys and the list of rules as the corresponding value. It then pipes this policy to the `Register` listener, which installs it in the run-time system.

One of the first things to notice about this example is that it composes effortlessly with the network monitoring programs developed in the previous subsection:

```
def repeater_web_monitor():
  repeater()
  all_stats()
```

Unlike the NOX code we saw before, in Frenetic there is no need to rewrite and interleave overlapping monitoring code and forwarding policy code. Because Frenetic presents the abstraction that queries read, but do not modify the network, these reads do not interfere with the forwarding policy. Conversely, because queries "see every packet", forwarding does not interfere with the *semantics* of a query (though, of course, sending packets along a monitored link does affect the *results* of a query). Under the hood, the Frenetic run-time system manages the interactions between the OpenFlow rules generated by queries implementation and the rules generated by the network policy.

A Simple Load Balancer. A load balancer is a switch that receives traffic on its incoming ports and multiplexes that traffic out its outgoing ports. Our load balancer will multiplex traffic based on source IPs: traffic from the same source IP will be forwarded through the same output port; traffic from different source IPs may be forwarded through different output ports.

Figure 5 presents the code for the core load balancing algorithm. The code uses a query (defined by `src_ips`) to generate an event with one value for each new source IP in a packet arriving on port 1. The main routine, `balance`, takes an argument `balancer`, which is an event function that transforms IP addresses into ports (we assume traffic will be multiplexed through ports 2 through `OUTPORTS`). The `balance` function itself runs the `src_ips` query to generate an event for each new IP address seen, runs the `balancer` to determine the appropriate port through which to forward those packets, and uses library functions to construct the network policy as the result.

The `balancer` can be instantiated in many different ways. For example, the programmer might assume a uniform distribution of traffic across IP addresses and hash each source IP to a port,

```
def hash_balancer():
  return Lift(lambda ip,port: hash_ip_to_port(ip))
```

or they might implement round-robin load balancing:

```
def rr_balancer():
  next = lambda ip,port: (port % (OUTPORTS - 1)) + 2
  return (Accum(1,next))
```

```
# Delete rules referring to given IPs from policy
def filter_ips(ip_list, policy):
  secure_policy = policy
  for ip in ip_list:
    secure_policy = delete_ip_from_policy(secure_policy)
  return secure_policy

# Delete IPs generated from the bad_ips() event stream
def secure(policyE):
  return (BlendLeft({}, bad_ips(), policyE) >>
          Lift(filter_ips))

# Use the balancer and then apply security filtering
def secure_balance():
  (secure(balance(weighted_balancer())) >>
   Register())
```

Figure 6. Securing the weighted balancer. The `bad_ips` event and `delete_ip_from_policy` function are elided.

Yet another possibility is to monitor the load on the switch and implement the load balancer using dynamic traffic levels. The `ip_monitor` program below queries the packet counts by IP address every `INTERVAL` seconds. Then `weighted_balancer` pipes the result of the query into an event function `weighted_choice` (whose definition is elided), that selects the next port to forward through based on current traffic levels.

```
def ip_monitor():
  return (Select(counts) *
          Where(inport_fp(1)) *
          GroupBy([srcip]) *
          Every(INTERVAL))

def weighted_balancer():
  return (ip_monitor() >>
          weighted_choice())
```

Any of the above balancing functions can be used in conjunction with the generic balancer as follows.

```
def balance_switch():
  balance(weighted_balancer()) >>
  Register()
```

Interestingly, while creating this parameterized load balancer with Frenetic is a relatively straight-forward exercise in functional programming, simulating it in NOX is substantially more difficult. The crux of the problem is that the parameterized balancing algorithm (the function `balance`) cannot be defined in NOX without risking interference from the monitoring rules needed by components such as `weighted_balance`. The simplest NOX solution is likely to make multiple copies of the code—one for each separate balancing function—and handle interfering rules manually. Frenetic's run-time system handles all such interference automatically.

Composing Forwarding Decisions. The previous examples illustrate composition of queries with each other and with a single policy module. It is also possible to compose a routine that computes a forwarding policy with separate routines that transform or alter the policy. A typical example is a security module that prevents known bad source IPs from sending traffic, as shown in Figure 6. Frenetic's functional style makes such examples easy to code. It is typically much more difficult to compose the forwarding policies computed by different NOX modules, unless those modules act on completely disjoint sets of packets.

Summary. Most network programs involve a combination of monitoring and forwarding. Because queries can always "see every network packet" independently of the forwarding policies expressed by other modules, monitoring and policy components compose seamlessly in Frenetic. Moreover, because of Frenetic's functional style, post-facto application of policy modifiers, such as our security module, is trivial. Overall, it is far easier to write simple, modular, reuseable programs in Frenetic than it is in NOX.

```
function packet_in(packet, inport)
  isSubscribed := false
  actions := []
  for (query, event, counters, requests) ∈ subscribers 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, packets, bytes)
  for (query, event, counters, requests) ∈ subscribers do
    if requests.contains(xid) then
      counters.add(packets, bytes)
      requests.remove(xid)
      if requests.is_empty() then
        event.push(counters)

function stats_loop()
  while true do
    query := next_stats()
    counters.reset()
    for pattern ∈ flows do
      if query.matches(pattern) then
        xid := stats_request(pattern)
        requests.add(xid)
    sleep(next_stats_window())
```

Figure 7. Frenetic run-time system handlers

5. Frenetic Implementation

Frenetic provides high-level abstractions that free programmers from having to reason about a host of low-level details involving the underlying switch hardware. However, the need to deal with these low-level details does not just disappear because programs operate at a higher level. 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 is organized around three main pieces: the language itself, the run-time system, and NOX. The use of NOX is convenient but not essential—we could also use any other controller as a back-end.

The central piece of the implementation is the run-time system, which sits between the high-level program and NOX. It manages all of the bookkeeping related to installing and uninstalling rules on switches and also generates the necessary communication patterns between switches and the controller. To do all of this, the run-time maintains several global data structures:

- *policy*, a set of switch-indexed high-level rules describing the current packet-forwarding policy,
- *flows*, a set of low-level rules currently installed on the switches in the network, and

		Connectivity			Heavy Hitters			Web Stats		
		HUB	*LSW*	*LFLSW*	*HUB*	*LSW*	*LFLSW*	*HUB*	*LSW*	*LFLSW*
NOX	Lines of Code	20	55	75	110	198		104	135	
	Controller Traffic (kB)	8.8	9.7	22.2	8.4	10.7	⋆	7.5	7.1	⋆
	Aggregate Traffic (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 Traffic (kB)	8.8	11.8	12.4	10.8	11.8	12.3	5.8	6.8	7.4
	Aggregate Traffic (kB)	65.3	40.6	41.2	149.3	80.4	86.3	32.4	18.3	18.8

Table 1. Experimental results.

- *subscribers*, a set of tuples containing a defining query, an event for that subscriber, byte and packet counts, and a list of outstanding statistics requests.

To translate the high-level forwarding policy registered in the run-time into switch-level rules, the run-time uses 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 every packet is sent to the controller and passed to the `packet_in` handler. When it receives a packet, this function first iterates through the set of subscribers and propagates the packet to each subscriber whose defining query includes the packet in its result. Next, it traverses the policy and collects up the list of actions specified in all rules. Finally, it processes the packet in one of two ways: If there are no subscribers for the packet, then it installs a switch-level rule that processes future packets with the same header fields without involving the controller. Or, if there are 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 slightly different strategy to implement aggregate statistics subscribers, making use of the byte and packet counters maintained by the switches. The run-time system executes a loop that waits until the window for a statistics subscriber expires. 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.

Figure 7 gives pseudo-code 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[3] with which the actual implementation has to deal with 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.

[3] For example, when the forwarding policy changes, some of the rules installed on switches may be stale and must be uninstalled. But when the run-time uninstalls a rule on a switch, the byte and packet counters associated with the switch-level rule must not be lost. Thus, the Frenetic run-time defines a `flow_removed` handler that receives the counters for uninstalled rules and adds them to the counters maintained on the controller.

The other 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. Unlike classic FRP implementations, which support continuous streams called *behaviors* as well as 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. Accordingly, our FRP library uses a push-based strategy to propagate values from inputs to outputs.

The run-time system's use of exact-match rules follows the approach used in Ethane [10] and many OpenFlow-based applications [19, 22], and is well-suited for dynamic settings. Moreover, exact-match rules use the plentiful conventional memory (*e.g.*, SRAM) many switches provide, as opposed to the small, expensive, power-hungry Ternary Content Addressable Memories (TCAMs) needed to support wildcards. Still, wildcard rules are more concise and well-suited for static settings. We plan to develop a proactive, priority-based wildcard approach as part of Frenetic's run-time in the near future. Longer term, we plan to extend the run-time to *adaptively* select between exact-match and wildcard rules, depending on the capabilities of the switches in the network.

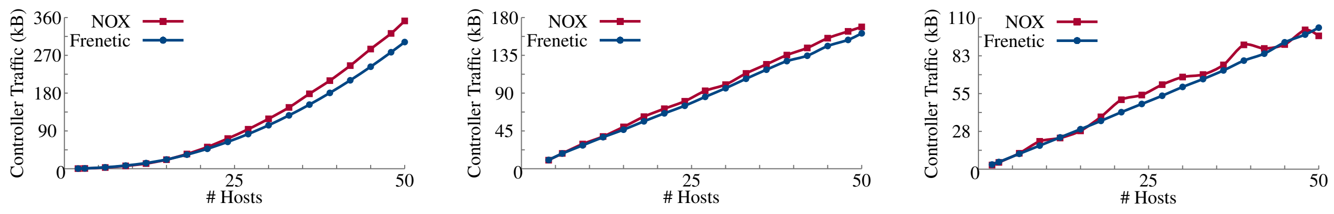
6. Evaluation

To evaluate our design for Frenetic, we implemented several simple applications in Frenetic and compared them against equivalent NOX programs on three metrics: lines of code, traffic to controller, and total 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 and Methodology. We ran our experiments using the Mininet virtualization environment [27] on a Linux host with a 2.4GHz Intel Core2 Duo processor and 4GB of RAM. Mininet does not provide performance fidelity but does give accurate traffic measurements. For the lines of code metric, we counted up to 80 characters of properly-indented Python excluding whitespace. We used Wireshark to tally controller and total traffic.

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

- **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.
- **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.



(a) All-Pairs Connectivity (c) Heavy Hitters (b) Web Statistics

Figure 8. Scalability results.

- **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 (LFLSW) learns the host-port mapping and also monitors the network topology and calculates a minimum spanning tree. This avoids forwarding loops when flooding packets.

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 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.[4] Finally, the aggregate traffic statistics for LFLSW 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 LFLSW 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.

[4] 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.

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 the learning switch forwarding policy, but scaled the number of hosts up from 1 to 50. The results in Figure 8 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 timeouts.

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 measured 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.

Further Experience. In addition to the quantitative benchmarks discussed so far, we have implemented a collection of network utilities in Frenetic to validate our language design. This list of programs ranges from essential network functions to novel applications that implement new functionality. Frenetic's modular design makes it easy to build new tools out of simpler, reuseable parts. Code for these examples is hosted on the Frenetic web site [2].

- **Discovery.** Discovers the network topology.
- **Spanning Tree.** Computes a spanning tree from the topology.
- **All-Pairs Shortest-Path Routing.** Uses the topology to compute a forwarding policy based on shortest paths.
- **Load Balancer.** Connects incoming traffic to one of several replica servers. Can be instantiated with many heuristics to balance incoming traffic across back-end servers.
- **Fault-tolerant Routing.** Connects incoming traffic to one of several replica switches, organized into several layers. When a

switch goes down, traffic is routed through the other switches in the same layer.

- **Address Resolution Protocol (ARP) Server.** Implements ARP *in the network*, by maintaining a global view of the IP-MAC address mapping.
- **Dynamic Host Configuration (DHCP) Server.** Implements DHCP to bootstrap network hosts with logical (IP) addressing information.
- **Memcached Query Router.** Connects clients to virtual servers implementing a key-value store. The switch translates between the virtual addresses assigned to servers and the servers' physical addresses. When servers fail, it reassigns its virtual addresses to another server; when new servers becomes available, virtual addresses from other servers are remapped to it.
- **Scan-Free Learning Switch.** Generalized learning switch. Detects and blocks malicious hosts that scan the network.
- **DDoS Defense.** Detects anomalies in the amount of traffic sent over the network and drops packets from the offending hosts.

7. Related Work

This paper extends preliminary work by Foster *et al.* [20]. Unlike the present paper, that work did not describe a run-time system, query language, or any significant applications, and it did not provide an evaluation of the language design or implementation.

The OpenFlow platform provides a uniform interface for programming physical network switches [3, 31, 32]. Many other platforms for programming network devices have also been proposed. The Click modular router [25] shares the general goal of making network devices programmable and, like Frenetic, emphasizes modularity as an organizing design principle. But Click exclusively targets software switches (implemented as a Linux kernel module) while Frenetic can be used with physical switches (implemented using special-purpose hardware). RouteBricks [16] attempts to obtain better performance from software switches implemented using stock machines. Bro [36] and Snortran [17] allow programmers to express rich packet-filtering and monitoring policies for securing networks and detecting intrusions while Shangri-La [11] and FPL-3E [15] compile high-level packet-processing programs down to special packet-processing hardware and FPGAs. The key difference between Frenetic and all of these systems is that they are limited to a single device. Thus, they do not address the issue of how to program a collection of interconnected switches.

The Frenetic implementation is hosted on the NOX controller [21], which provides convenient C++ and Python APIs for handling raw events and communicating with switches. Several other OpenFlow controllers have also been proposed. Beacon [1] is similar to NOX but provides a Java API. Maestro [9] provides a modular mechanism for managing network state using programmer-defined views. It is also multi-threaded, which increases throughput dramatically. Onix [26] provides abstractions for partitioning and distributing network state onto multiple distributed controllers, which addresses the scalability and fault-tolerance issues that arise when using a centralized controller. SNAC [4] provides high-level patterns (similar to Frenetic's filter patterns) for specifying access control policies as well as a graphical monitoring tool but is not a general programming environment. FML [24] also provides a high-level pattern language for specifying security policies in OpenFlow networks [24].

Frenetic's event functions are modeled after functional reactive languages such as Yampa and others [18, 33, 35, 37]. Its push-based implementation is based on FrTime [12] and is similar to adaptive functional programming [6]. The Flask [30] language applies functional reactive programming to sensor networks in a staged language. The key differences between Frenetic and these 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. At a high level, Frenetic is also similar to streaming languages such as StreamIt [39], CQL [7], Esterel [8], Brooklet [38], etc. The FRP operators used in Frenetic are more to our taste, but one could easily build a system that retained the main elements of our design (e.g., the query language and the run-time system) but used different constructs for processing streams of network events.

The Nettle [40] language also uses FRP combinators to program OpenFlow switches. A Nettle program takes a stream of raw OpenFlow events as input (e.g., *switch_join*, *port_change*, *packet_in*, etc.) and produces a stream of raw OpenFlow messages as output (e.g., *install*, *uninstall*, *query_stats*, etc.). Although Nettle and Frenetic appear superficially similar—both use FRP for OpenFlow networks—a closer inspection reveals substantial differences. The most important difference is that Nettle operates at a lower level of abstraction than Frenetic: it is an effective *substitute* for NOX while Frenetic *sits on top of* NOX (and, in the future, could potentially sit on top of Nettle). Nettle does not offer any analog of Frenetic's query language or its run-time system and so Nettle programs work in terms of low-level OpenFlow concepts such as switch-level rules, priorities, and timeouts. As such it suffers from all of the limitations of NOX discussed in Section 3—e.g., Nettle programs cannot be easily composed and are susceptible to network race conditions.

NDLog, an extension of Datalog developed by Loo, Hellerstein, et al. has been used to specify and implement routing protocols, overlay networks, and services such as distributed hash tables [28, 29]. Both Frenetic and NDLog use high-level languages to program networks, but there are some important differences. One is NDLog's focus on routing protocols and overlay networks, whereas Frenetic programs can be used to implement finer-grained packet-processing including rewriting header fields. Another difference is that NDLog programs are written in an explicitly distributed style while Frenetic offers the programmer the abstraction of a centralized view of the network. This dramatically changes the way that programs must be written: an NDLog programmer crafts a single query that is evaluated on every router in the network while a Frenetic programmer writes a program from the omniscient perspective of the controller and run-time system distributes low-level rules to the switches in the network. Finally, deploying NDLog in a production network would require deep changes to the way that switches are built, as it requires each switch to run a custom Datalog engine. Frenetic targets OpenFlow, which is already supported by several vendors, and so can be deployed immediately.

One of the main challenges in the implementation of Frenetic is splitting work between the (powerful but slow) controller and the (fast but limited) switches. Gigascope [14], a stream database for monitoring networks, addresses the same problem but, unlike Frenetic, only supports querying traffic and cannot be used to control the processing of packets in the network.

8. 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 programming model by introducing a collection of high-level and compositional operators for querying and transforming streams of network traffic. A run-time system handles all of the details related to installing and uninstalling low-level rules. An experimental evaluation demonstrates that the performance of Frenetic's run-time system is competitive with hand-written OpenFlow/NOX programs.

We are working to extend Frenetic in several directions. We are developing security applications for performing authentication and access control, and for ensuring isolation. We are also designing a new run-time system 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 Matthew Meola, Mark Reitblatt, and Minlan Yu for helpful discussions. Our work is supported in part by ONR grants N00014-09-1-0770 *Networks Opposing Botnets* and N00014-09-1-0652 *Fabric: A Higher-Level Abstraction for Building Secure Distributed Applications*. 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] The Frenetic language. See `http://www.frenetic-lang.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] Arvind Arasu, Shivanth Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15:121–142, Jun 2006.

[8] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, (2):87–152, 1992.

[9] Zheng Cai, Alan L. Cox, and T. S. Eugene Ng. Maestro: A system for scalable OpenFlow control. Technical Report TR10-08, Rice University, Dec 2010.

[10] 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.

[11] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *PLDI*, pages 224–236, Jun 2005.

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

[13] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Haskell Workshop*, pages 7–18, Aug 2003.

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

[15] Mihai Lucian Cristea, Claudiu Zissulescu, Ed Deprettere, and Herbert Bos. FPL-3E: Towards language support for reconfigurable packet processing. In *SAMOS*, pages 201–212. Jul 2005.

[16] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *SOSP*, Oct 2009.

[17] Sergei Egorov and Gene Savchuk. *SNORTRAN: An Optimizing Compiler for Snort Rules*. Fidelis Security Systems, 2002.

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

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

[20] Nate Foster, Rob Harrison, Matthew L. Meola, Michael J. Freedman, Jennifer Rexford, and David Walker. Frenetic: A high-level langauge for OpenFlow networks. In *PRESTO*, Nov 2010.

[21] 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.

[22] 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*.

[23] 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.

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

[25] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug 2000.

[26] 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.

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

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

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

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

[31] John Markoff. Open networking foundation pursues new standards. *The New York Times*, Mar 2011. See `http://nyti.ms/eK3CCK`.

[32] 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.

[33] 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, 2009.

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

[35] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell Workshop*, pages 51–64, Oct 2002.

[36] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, Dec 1999.

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

[38] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. A universal calculus for stream processing languages. In *ESOP*, pages 507–528, 2010.

[39] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196, Apr 2002.

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

[41] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE*, Mar 2011.