Composing Software-Defined Networks

Christopher Monsanto[*], *Joshua Reich*[*], *Nate Foster*[†], *Jennifer Rexford*[*], *David Walker*[*]
[*]Princeton [†]Cornell

Abstract

Managing a network requires support for multiple concurrent tasks, from routing and traffic monitoring, to access control and server load balancing. Software-Defined Networking (SDN) allows applications to realize these tasks directly, by installing packet-processing rules on switches. However, today's SDN platforms provide limited support for creating modular applications. This paper introduces new abstractions for building applications out of multiple, independent modules that jointly manage network traffic. First, we define composition operators and a library of policies for forwarding and querying traffic. Our parallel composition operator allows multiple policies to operate on the same set of packets, while a novel *sequential composition operator* allows one policy to process packets after another. Second, we enable each policy to operate on an *abstract topology* that implicitly constrains what the module can see and do. Finally, we define a new *abstract packet model* that allows programmers to extend packets with virtual fields that may be used to associate packets with high-level meta-data. We realize these abstractions in Pyretic, an imperative, domain-specific language embedded in Python.

1 Introduction

Software-Defined Networking (SDN) can greatly simplify network management by offering programmers network-wide visibility and direct control over the underlying switches from a logically-centralized controller. However, existing controller platforms [7, 12, 19, 2, 3, 24, 21] offer a "northbound" API that forces programmers to reason manually, in unstructured and ad hoc ways, about low-level dependencies between different parts of their code. An application that performs multiple tasks (e.g., routing, monitoring, access control, and server load balancing) must ensure that packet-processing rules installed to perform one task do not override the functionality of another. This results in monolithic applications where the logic for different tasks is inexorably intertwined, making the software difficult to write, test, debug, and reuse.

Modularity is the key to managing complexity in any software system, and SDNs are no exception. Previous research has tackled an important special case, where each application controls its own *slice*—a *disjoint* portion of traffic, over which the tenant or application module has (the illusion of) complete visibility and control [21, 8]. In addition to traffic isolation, such a platform may also support subdivision of network resources (e.g., link bandwidth, rule-table space, and controller CPU and memory) to prevent one module from affecting the performance of another. However, previous work does not address how to build a *single* application out of multiple, independent, reusable network policies that affect the processing of the *same* traffic.

Composition operators. Many applications require the same traffic to be processed in multiple ways. For instance, an application may route traffic based on the destination IP address, while monitoring the traffic by source address. Or, the application may apply an access-control policy to drop unwanted traffic, before routing the remaining traffic by destination address. Ideally, the programmer would construct a sophisticated application out of multiple modules that each *partially* specify the handling of the traffic. Conceptually, modules that need to process the same traffic could run in parallel or in series. In our previous work on Frenetic [6, 14], we introduced *parallel* composition, which gives each module (e.g., routing and monitoring) the illusion of operating on its own copy of each packet. This paper introduces a new kind of composition—*sequential* composition—that allows one module to act on the packets already processed by another module (e.g., routing after access control).

Topology abstraction. Programmers also need ways to limit each module's sphere of influence. Rather than have a programming platform with one (implicit) global network, we introduce *network objects*, which allow

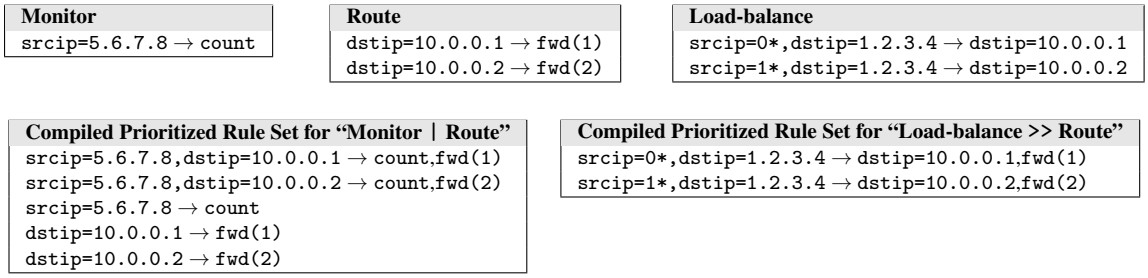

Figure 1: Parallel and Sequential Composition.

each module to operate on its own abstract view of the network. Programmers can define network objects that naturally constrain what a module can see (information hiding) and do (protection), extending previous work on topology abstraction techniques [4, 17, 10, 25].

The Pyretic Language and System. Pyretic is a new language and system that enables programmers to specify network policies at a high level of abstraction, compose them together in a variety of ways, and execute them on abstract network topologies. Running Pyretic programs efficiently relies on having a run-time system that performs composition and topology mapping to generate rules to install in the switches. Our initial prototype, built on top of POX [19], is a simple interpreter that handles each packet at the controller. While sufficient to execute and test Pyretic programs, it does not provide realistic performance. In our ongoing work, we are extending our run-time system to proactively generate and install OpenFlow rules, building on our previous research [14].

The next section presents a top-down overview of Pyretic's composition operators and topology abstraction mechanisms. The following two sections then explain each in detail, building a complete picture of Pyretic from the bottom up. Section 3 presents the Pyretic language, including an abstract packet model that conveys information between modules, and a library for defining and composing policies. Section 4 introduces network objects, which allow each module to apply a policy over its own abstract topology, and describes how our run-time system executes Pyretic programs. To evaluate the language, Section 5 presents example applications running on our Pyretic prototype. After reviewing related work in Section 6, we conclude in Section 7.

2 Abstractions for Modular Programming

Building modular SDN applications requires support for composition of multiple independent modules that each partially specify how traffic should be handled. The parallel and sequential composition operators (Section 2.1) offer simple, yet powerful, ways to combine policies generated by different modules. Network objects (Section 2.2) allow policies to operate on abstract locations that map—through one or more levels of indirection—to ones in the physical network.

2.1 Parallel and Sequential Composition Operators

Parallel and sequential composition are two central mechanisms for specifying the relationship between packet-processing policies. Figure 1 illustrates these abstractions through two examples in which policies are specified via prioritized lists of OpenFlow-like rules. Each rule includes a *pattern* (`field=value`) that matches on bits in the packet header (e.g., source and destination MAC addresses, IP addresses, and TCP/UDP port numbers), and simple *actions* the switch should perform (e.g., drop, flood, forward out a port, rewrite a header field, or count[1] matching packets). When a packet arrives, the switch (call it s) identifies the first matching rule and performs the associated actions. Note that one may easily think of such a list of rules as a function: The function input is a packet at a particular inport on s and the function output is a multiset of zero or more packets on various outports of s (zero output packets if the matching rule drops the input packet; one output if it forwards the input; and one or more if it floods). We call a packet together with its location a *located packet*.

Parallel Composition (|): Parallel composition gives the illusion of multiple policies operating concurrently on separate copies of the same packets [6, 14]. Given two policy functions f and g operating on a located packet p, parallel composition computes the multiset *union* of $f(p)$ and $g(p)$—that is, every located packet produced by either policy. For example, suppose a programmer writes one module to monitor traffic by source IP address, and another to route traffic by destination IP address. The monitoring module (Figure 1, top-left) comprises a simple policy that consists of a single rule applying the count action to packets matching source IP address `5.6.7.8`. The routing module (Figure 1, top-middle) consists of two rules, each matching on a destination IP address and forwarding packets out the specified port. Each module

[1]The OpenFlow API does not have an explicit `count` action; instead, every rule includes byte and packet counters. We consider `count` as an explicit action for ease of exposition.

generates its policy independently, with the programmer using the "|" operator to specify that both the route and monitor functions should be performed simultaneously. These can be mechanically compiled into a single joint ruleset (Figure 1, bottom-left) [14].

Sequential Composition (>>): Sequential composition gives the illusion of one module operating on the packets produced by another. Given two policy functions f and g operating on a located packet p, sequential composition applies g to each of the located packets produced by $f(p)$, to produce a new set of located packets. For example, suppose a programmer writes one module to load balance traffic destined to a public IP address `1.2.3.4`, over multiple server replicas at private addresses `10.0.0.1` and `10.0.0.2`, respectively, and another to route traffic based on the chosen destination server. The load-balancing module splits traffic destined to the public IP between the replicas based on the client IP address. Traffic sent by clients with an IP address whose highest-order bit is `0` go to the first server, while remaining traffic goes to the second server. As shown in Figure 1 (top-right), the load balancer performs a rewriting action to modify the destination address to correspond to the chosen server replica, without actually changing the packet's location. This load balancer can be composed sequentially with the routing policy introduced earlier. Here the programmer uses the ">>" operator to specify that load balancing should be performed first, followed by routing. Again, these may be mechanically compiled into a single joint ruleset (Figure 1, bottom-right).

2.2 Topology Abstraction With Network Objects

Modular programming requires a way to constrain what each module can *see* (information hiding) and *do* (protection). Network objects offer both information hiding and protection, while offering the familiar abstraction of a network topology to each module. A network object consists of an abstract topology, as well as a policy function applied to the abstract topology. For example, the abstract topology could be a subgraph of the real topology, one big virtual switch spanning the entire physical network, or anything in between. The abstract topology may consist of a mix of physical and virtual switches, and may have multiple levels of nesting on top of the one real network. To illustrate how topology abstraction may help in creating modular SDN applications we look at two examples: a "many-to-one" mapping in which several physical switches are made to appear as one virtual switch and a "one-to-many" mapping in which one physical switch is presented as several virtual switches.

Many-to-one. While MAC-learning is an effective way to learn the locations of hosts in a network, the need to compute spanning trees makes Ethernet protocols unattractive in large networks. Instead, a programmer could combine MAC-learning at the edge of the network with shortest-path routing (for unicast traffic) and multicast trees (for broadcast and flooding traffic) in the network interior [18, 23]. Topology abstraction provides a simple way to realize this functionality, as shown in Figure 2. The MAC-learning module sees the network as one big switch `V`, with one port for each edge link in the underlying physical network (dotted lines). The module can run the conventional MAC-learning program to learn where hosts are located. When a previously-unknown host sends a packet, the module associates the source address with the input port, allowing the module to direct future traffic destined to this address out that port. When switch `V` receives a packet destined to an unknown address, the module floods the packet; otherwise, the switch forwards the traffic to the known output port.

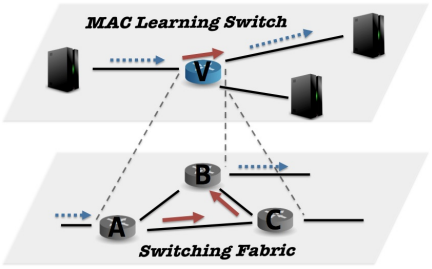

Figure 2: Many physical switches to one virtual.

The "switching fabric" of switch `V` is implemented by the switching-fabric module, which sees the entire physical network. the switching-fabric module performs routing from *one edge link to another* (e.g., from the ingress port at switch `A` to the egress port at switch `B`), This requires some coordination between the two modules, so the MAC-learner can specify the chosen output port(s), and the switching-fabric module can direct traffic on a path to the egress port(s).

As a general way to support coordination, we introduce an *abstract packet model*, incorporating the concept of *virtual packet headers* that a module can push, pop, and inspect, just like the actions OpenFlow supports on real packet-header fields like VLAN tags and MPLS labels. When the MAC-learning module directs traffic from an input port to an output port, the switching-fabric module sees traffic with a virtual packet header indicating the corresponding ingress and egress ports in its view of the network. A run-time system can perform the necessary mappings between the two abstract topologies, and generate the appropriate rules to forward traffic from the ingress port to the appropriate egress port(s). In practice, a run-time system may represent virtual packet-header fields using VLAN tags or MPLS labels, and in-

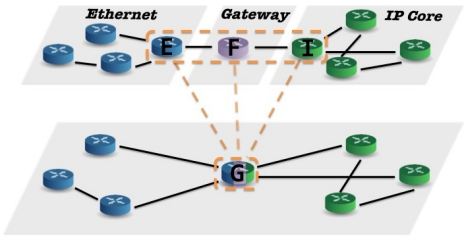

Figure 3: One physical switch to many virtual.

stall rules that push, pop, and inspect these fields.

One-to-many. Enterprise networks often consist of several Ethernet islands interconnected by gateway routers to an IP core, as shown in Figure 3. To implement this behavior, an SDN programmer would have to write a single, monolithic program that handles network events differently depending on the role the switch is playing in the network. This program would implement MAC-learning and flooding to unknown destinations for switches within Ethernet islands, shortest-path routing on IP prefixes for switches in the IP core, and gateway logic for devices connecting an island to the core. The gateway logic would be complicated, as the switch would need to act simultaneously as MAC-learner, IP router, and MAC-rewriting repeater and ARP server.

A better alternative would be to implement the Ethernet islands, IP core, and gateway routers using separate modules operating on a subset of the topology, as shown in Figure 3. This design would allow the gateway router to be decomposed into three virtual devices: one in the Ethernet island (`E`), another in the IP core (`I`), and a third interconnecting the other two (`F`). Likewise, its logic could be decomposed into three orthogonal pieces: a MAC-rewriting repeater that responds to ARP queries for its gateway address (on `F`), an Ethernet switch (on `E`), and an IP router (on `I`). The programmer would write these modules separately and rely on a run-time system to combine them into a single program.

For example, suppose a host in the Ethernet LAN sends a packet to a destination reachable via the IP core. In the Ethernet LAN, this packet has a destination MAC address of the gateway. The Ethernet module would generate a rule in switch `E` that matches traffic destined to the gateway's MAC address and forwards out `E`'s right port. The gateway module would generate a rule in switch `F` that matches packets from `F`'s left port destined to the gateway's MAC address and, after rewriting MAC headers appropriately, forwards out `F`'s right port. The IP core module would generate rules in switch `I` that match packets based on the destination IP address to forward traffic to the next hop along a path to the destination. A run-time system can combine these three sets of rules to generate the rules for the physical gateway switch `G`. Switch `G` would match traffic entering on its left two ports based on the gateway's destination MAC address and the destination IP address to forward via one of the two right ports, as chosen by the IP core module.

	Conventional SDN	Pyretic
Packet	Fixed OF fields	Extensible stacks of values
Policy	Prioritized OF rules	Functions of located packets
Network	One concrete network	Network object hierarchies

Table 1: Pyretic abstraction in three dimensions

3 The Pyretic Programming Language

Any SDN platform needs a model of data packets, forwarding policies, and the network that applies these policies—as summarized in Table 1. Compared to conventional platforms [7, 12, 2, 3, 19], our Pyretic language raises the level of abstraction by introducing an abstract packet model (Section 3.1), an algebra of high-level policies (Section 3.2), and network objects (Section 4).

3.1 Abstract Packet Model

The heart of the Pyretic programming model is a new, extensible packet model. Conceptually, each packet flowing through the network is a *dictionary* that maps field names to values. These fields include entries for (1) the packet location (either physical or virtual), (2) standard OpenFlow headers (e.g., source IP, destination IP, source port, etc.), and (3) custom data. The custom data is housed in *virtual fields* and is not limited to simple bit strings—a virtual field can represent an arbitrary data structure. Consequently, this representation provides a general way to associate high-level information with packets and enable coordination between modules.

In addition to extending the *width* of a packet by including virtual fields, we also extend its *height* by allowing every field (including non-virtual ones) to hold a *stack* of values instead of a single bitstring. These stacks allow Pyretic to present the illusion of a packet travelling through *multiple* levels of abstract networks. For example, to "lift" a packet onto a virtual switch, the run-time system *pushes* the location of the virtual switch onto the packet. Having done so, that virtual switch name sits on top of the concrete switch name. When a packet leaves a virtual switch, the run-time system *pops* a field off the appropriate stack. In the example in Figure 2, this enables the MAC-learning module to select an egress port on virtual switch `V` without knowing about the existence of switches `A`, `B`, and `C` underneath.

Expanding on the example in Figure 2, consider a packet `p` entering the network at physical switch `A` and physical input port `3`. We can represent `p` as:

```
{switch: A, inport: 3, vswitch: V, ... }
```

Pushing virtual switch name `V` on to the switch field of `p` produces a new packet with `V` on top of `A`:

```
{switch: [V, A], inport: 3, ... }
```

The push above hides the identity of the physical switch `A` and reveals only the identity of the virtual switch `V` to observers that only examine top-most stack values. This mechanism allows Pyretic applications to hide the concrete network and replace it with a new abstract one.

Thus far, we have experimented primarily with the abstraction of location information: switches and ports. However, there is nothing special about those fields. We can virtualize IP addresses, MAC addresses or any other information in a packet, if an application demands it. Programs must maintain the invariant that the standard OpenFlow header fields do not contain the empty stack; packets with an empty stack in such a field will not be properly realized as a standard OpenFlow packet.

Ideally, OpenFlow switches would support our extended packet model directly, but they do not. Our runtime system is responsible for bridging the gap between the abstract model and the OpenFlow-supported packets that traverse the network. It does so by generating a unique identifier that corresponds to a unique set of non-OpenFlow-compliant portions of the packet (i.e., all virtual fields and everything but the top of the stack in an OpenFlow-compliant field). This identifier is stored in spare bits in the packet.[2] Our run-time system manages a table that stores the mapping between unique ids and extended data. Hence, programmers do not need to manage this mapping themselves and can instead work in terms of high-level abstractions.

3.2 High-Level Policy Functions

Pyretic contains a sublanguage for specifying *static* (i.e., unchanging) policies. A static policy is a "snapshot" of a network's global forwarding behavior, represented as an abstract function from a located packet to a multiset of located packets. The output multiset may be empty; if so, the policy has effectively dropped the input packet. The output multiset may contain a single packet at a new location (e.g., unicast)—typically, though not always, an output port on the other side of the switch. Finally, the output multiset may contain several packets (e.g., multicast or broadcast). Of course, one cannot build many useful network applications with just a single static, unchanging policy. To do so, one must use a *series* of static policies (i.e., a *dynamic* policy).

3.2.1 Static Policy Functions

We first describe the details of the static policy language, which we call NetCore.[3] NetCore contains several distinct elements including *actions* (the basic packet-processing primitives), *predicates* (which are used to select certain subsets of packets), *query policies* (which are used to observe packets traversing the network), and finally *policy combinators*, which are used to mix primitive actions, predicates, and queries together to craft sophisticated policies from simple components. Figure 4 summarizes the syntax of the key elements of NetCore.

Primitive Actions:

A ::= `drop` | `passthrough` | `fwd(port)` | `flood` | `push(h=v)` | `pop(h)` | `move(h1=h2)`

Predicates:

P ::= `all_packets` | `no_packets` | `match(h=v)` | `ingress` | `egress` | P `&` P | (P `|` P) | `~`P

Query Policies:

Q ::= `packets(limit,[h])` | `counts(every,[h])`

Policies:

C ::= A | Q | P[C] | (C `|` C) | C `>>` C | `if_`(P,C,C)

Figure 4: Summary of static NetCore syntax.

Primitive actions. Primitive actions are the central building blocks of Pyretic policies; an action receives a located packet as input and returns a set of located packets as a result. The simplest is the `drop` action, which produces the empty set. The `passthrough` action produces the singleton set `{p}` where `p` is the input packet. Hence `passthrough` acts much like an identity function—it does not even move the packet from its input port. Perhaps surprisingly, `passthrough` is quite useful in conjunction with other policies and policy combinators. On input packet `p`, the `fwd(port)` action produces the singleton set containing the packet relocated to outport `port` on the same switch as a result. The `flood` action sends packets along a minimum spanning tree, excepting the incoming interface[4]. When viewed as a function, `flood` receives any packet located at an inport on switch `s` and produces an output set with one copy of the packet at each outport on `s` that belongs to a minimum spanning tree for the network (maintained by the run-time system). The last three actions, `push`, `pop`, and `move`, each yield a singleton set as their output: `push(h=v)` pushes value `v` on to field `h`; `pop(h)` pops a value off of field `h`; and `move(h1=h2)` pops the top value on field `h2` and pushes it on to `h1`.

Predicates. Predicates are essential for defining policies (or parts of policies) that act only on a *subset* of packets traversing the network. More specifically, given an input packet `p`, the policy `P[C]`, applies the policy function `C` to `p` if `p` satisfies the predicate `P`. If `p` does not satisfy `P` then the empty set is returned. (In other words, the packet is dropped.) Predicates include `all_packets` and `no_packets`, which match all or no packets, respectively; `ingress` and `egress` which, respectively, match any packets entering or exiting the net-

[2] Any source of spare bits (e.g., MPLS labels) could be used. Our current implementation uses the VLAN field.

[3] This variant of NetCore is an extension and generalization of a language with the same name, described in our earlier work [14].

[4] The same definition used by Openflow for its `flood` action.

work; and `match(h=v)`, matching all packets for which value v is the top value on the stack at field `h`. Complex predicates are constructed using basic conjunction (`&`), disjunction (`|`), and negation (`~`) operators. The form `match(h1=v1,h2=v2)` is an abbreviation for `match(h1=v1) & match(h2=v2)`.

As an example, the policy `flood`, on its own, will broadcast every single packet that reaches *any* inport of *any* switch *anywhere* in the network. On the other hand, the policy

```
match(switch=s1,inport=2,srcip='1.2.3.4') [flood]
```

only broadcasts those packets reaching `switch s1`, `inport 2` with source IP address `1.2.3.4`. All other packets are dropped.

Policies. Primitive actions `A` are policies, as are restricted policies `P[C]`. NetCore also contains several additional ways of constructing policies.

As discussed in Section 2, sequential composition is used to build packet-processing pipelines from simpler components. Semantically, we define sequential composition `C1 >> C2` as the function `C3` such that:

```
C3(packet) = C2(p1) ∪ ... ∪ C2(pn)
  when {p1,...,pn} = C1(packet)
```

In other words, we apply `C1` to the input, generating a set of packets (`p1`,..., `pn`) and then apply `C2` to each of those results, taking their union as the final result.

As an example, consider the following policy, which modifies the destination IP of any incoming packet to 10.0.0.1 and forwards the modified packet out port 3.

```
pop(dstip) >> push(dstip='10.0.0.1') >> fwd(3)
```

Indeed, the modification idiom is common enough that we define an abbreviation for it:

```
modify(h=v) = pop(h) >> push(h=v)
```

As a more elaborate example, consider a complex policy `P2`, designed for forwarding traffic using a set of tags (`staff`, `student`, `guest`) stored in a virtual field named `USERCLASS`. Now, suppose we would like to apply the policy for `staff` to a particular subset of the traffic arriving on network. To do so, we may write a policy `P1` to select and tag the relevant traffic. To use `P1` and `P2` in combination, we exploit sequential composition: `P1 >> P2`. Such a program is quite modular: if a programmer wanted to change the forwarding component, she would change `P2`, while if she wanted to change the set of packets labeled `staff`, she would change `P1`.

Parallel composition is an alternative and orthogonal form of composition to sequential composition. The parallel composition `P3 | P4` behaves as if `P3` and `P4` were executed on every packet simultaneously. In other words, given an input packet `p`, `(P3 | P4)(p)` returns the set of packets `P3(p) U P4(p)`.

Continuing our example, if a programmer wanted to apply the policy `P3 | P4` to packets arriving at switch `s1` and a different policy `P6` to packets arriving at `s2`, she could construct the following composite policy `P7`.

```
P5 = P3 | P4
P6 = ...
P7 = match(switch=s1)[P5] | match(switch=s2)[P6]
```

After recognizing a security threat from source IP address, say address `1.2.3.4`, the programmer might go one step further creating policy `P8` that restricts `P7` to applying only to traffic from other addresses (implicitly dropping traffic from `1.2.3.4`).

```
P8 = ~match(srcip='1.2.3.4')[P7]
```

The policy `if_` is a convenience conditional policy. For example, if the current packet satisfies `P`, then

```
if_(P, drop, passthrough)
```

drops that packet while leaving all others untouched (allowing a subsequent policy in a pipeline of sequential compositions to process it). Conditional policies are a derived form that can be encoded using parallel composition, restriction, and negation.

Queries. The last kind of policy we support is a *query policy* (`Q`). Intuitively, a query is an abstract policy that directs information from the physical network to the controller platform. When viewed as a function, a query receives located packets as arguments and produces new located packets as results like any other policy. However, the resulting located packets do not find themselves at some physical port on some physical switch in the network. Instead, these packets are diverted to a data structure resident on the controller called a "bucket".

NetCore contains two queries: `counts` and `packets`, which, abstractly, direct packets to two different types of buckets, a `packet_bucket` and a `counting_bucket`, respectively. Applications register listeners (i.e., callbacks) with buckets; these callbacks are invoked to process the information contained in the bucket. Semantically, the two query policies differ only in terms of the information each bucket reveals to its listeners.

The `packet_bucket`, as its name suggests, passes entire packets to its listeners. For example, `packets(limit,['srcip'])` invokes its listeners on up to `limit` packets for each source IP address. The two most common values for `limit` are `None` and 1, with `None` indicating the bucket should process an unlimited number of packets. More generally, a list of headers is allowed: the bucket associated with `packets(1,[h1,...,hk])` invokes each listener on at most 1 packet for each distinct record of values in fields `h1` through `hk`.

The `counting_bucket` supplies its listeners with aggregate packet statistics, not the packets themselves. Hence, it may be implemented using OpenFlow counters in switch hardware. The policy

```
from pyretic.lib import *

def main():
  return flood
```

Figure 5: A complete program: `hub.py`.

`counts(every,['srcip'])` creates a bucket that calls its listeners every `every` seconds and provides each listener with a dictionary mapping source IP addresses to the cumulative number of packets containing that source IP address received by the bucket. As above, counting policies may be generalized to discriminate between packets on the basis of multiple headers: `counts(every,[h1,...,hk])`.

A query policy `Q` may be used in conjunction with any other policy we define. For example, if we wish to analyze traffic generated from IP address `1.2.3.4` using `Q`, we may simply construct the following.

```
match(srcip='1.2.3.4') [Q]
```

If we wanted to both forward and monitor a certain subset of packets, we use parallel composition as before:

```
match(srcip='1.2.3.4') [Q | fwd(3)]
```

3.2.2 From Static Policies to Dynamic Applications

After defining a static policy, the programmer may use that policy within a Pyretic application. Figure 5 presents the simplest possible, yet complete, Pyretic application. It imports the Pyretic library, which includes definitions of all primitive actions (such as `flood`, `drop`, etc.), policy operators and query functions, as well as the run-time system. The program itself is trivial: `main` does nothing but return the `flood` policy. A Pyretic program such as this one is executed by starting up a modified version of the POX run-time system [19]. POX reads the Pyretic script and executes `main`. Although we use POX for low-level message processing, our use of POX is not essential. A Pyretic-like language could be built on top of any low-level controller.

Monitoring. Figure 6 presents a second simple application, designed to monitor and print packets from source IP `1.2.3.4` to the terminal. In this figure, the `dpi` function first creates a new packet-monitoring policy named `q`. Next, it registers the `printer` listener with the query `q` using `q`'s `when` method. This listener will be called each time a packet arrives at the `packet_bucket` to which `q` forwards. Finally, the `dpi` function constructs and returns a policy that embeds the query within it. The `main` function uses `dpi` and further composes it with a routing policy (the simple `flood`).

MAC-learning. Figure 7 presents an MAC-learning module that illustrates how to construct a dynamic policy. It is designed assuming that network hosts do not move. It initially operates by flooding all packets it receives. For each switch, when a packet with a new source MAC address (say, MAC address `M`) appears at one of its input ports (say, inport `I`), it concludes that `M` must live off `I`. Consequently, it refines its forwarding behavior so that packets with destination MAC address `M` are no longer flooded, but instead forwarded only out of port `I`.

```
def printer(pkt):
  print pkt

def dpi():
  q = packets(None,[])
  q.when(printer)
  return match(srcip='1.2.3.4')[q]

def main():
  return dpi() | flood
```

Figure 6: Deep packet inspection.

```
def learn(self):

  def update(pkt):
    self.P =
      if_(match(dstmac=pkt['srcmac'],
                switch=pkt['switch']),
          fwd(pkt['inport']),
          self.P)

  q = packets(1,['srcmac','switch'])
  q.when(update)

  self.P = flood | q

def main():
  return dynamic(learn)()
```

Figure 7: MAC-learning switch.

Examining a few of the details of Figure 7, we see that the last line of the `learn` function is the line that initializes the policy—it starts out flooding all packets and using `q` to listen for packets with new source MAC addresses. The listener for query is the function `update`, which receives packets with new source MACs as an argument. That listener updates the dynamic policy with a conditional policy that tests future packets to see if their destination MAC is equal to the current packet's source MAC. If so, it forwards the packet out the inport on which the current packet resides. If not, it invokes the existing policy `self.P`. In this way, over time, the policy is extended again and again until the locations of all hosts have been learned.

The last line of Figure 7 uses the function `dynamic` to wrap up `learn` and produce a new dynamic policy class, whose constructor it then calls to produce a operational dynamic policy instance.

Load balancer. As a final example in this section, we show how to construct a simple dynamic server load bal-

ancer. Doing so illustrates a common Pyretic programming paradigm: One may develop dynamic applications by constructing parameterized static policies, listening for network events, and then repeatedly recomputing the static policy using different parameters as the network environment changes. The example also illustrates the process of building up a somewhat more complex policy by defining a collection of ordinary Python functions that compute simple, independent components of the policy, which are subsequently composed together.

Figure 8 presents the code, which spreads requests for a single public-facing IP address over multiple back-end servers. The first three functions in the figure collaborate to construct a static load balancer. In the first function (`subs`), variable `c` is the client IP address prefix, `p` is the service's public-facing IP address, and `r` is the specific replica chosen. This function rewrites any packet whose source IP address matches `c` and destination IP address is `p` so the destination IP address is `r`, and vice versa. All other packets are left unmodified. The next function, `rewrite`, iterates `subs` over a dictionary `d` mapping IP prefixes to server replicas. To simplify this code, we have overloaded sequential composition (`>>`) so that it can be applied to a list of policies (placing each element of the list in sequence). Hence, intuitively, `rewrite` sends each packet through a pipeline of tests and when the test succeeds, the packet is transformed. The third function, `static_lb` invokes `rewrite` with a function `balance` (definition omitted from the figure) that partitions possible clients and assigns them to replicas, using a lists of server replicas `R` and a dictionary `H` containing a history of traffic statistics.

Now, to build a dynamic load balancer that changes the mapping from clients to server replicas over time, consider `lb`. This dynamic balancer issues a query `q` that computes a dictionary mapping source IP addresses to packet counts every minute (60 seconds). Each time the query returns a new `stats` value, the policy invokes `rebalance`, which updates the history `H` and recomputes a new load balancing policy using `static_lb`.

```
def subs(c,r,p):
  c_to_p = match(srcip=c,dstip=p)
  r_to_c = match(srcip=r,dstip=c)
  return c_to_p[modify(dstip=r)] |
         r_to_c[modify(srcip=p)] |
         (~r_to_c & ~c_to_p)[passthrough]

def rewrite(d,p):
  return (>>)([subs(c,r,p) for c,r in d])

def static_lb(p,R,H):
  return rewrite(balance(R,H),p)

def lb(self,p,R,H):

  def rebalance(stats):
    H = H.update(stats)
    self.P = static_lb(p,R,H)

  q = counts(60,['srcip'])
  q.when(rebalance)
  self.P = static_lb(p,R,H) | match(dstip=p)[q])
```

Figure 8: Dynamic load balancer (excerpts).

4 Network Objects

The policy language described in the preceding section provides programmers with flexible constructs that make it easy to build sophisticated network applications out of simple, independent components. However, it suffers from a significant limitation: programmers must specify policies in terms of the underlying physical topology. This hinders code reuse since policies written for one topology typically cannot be used with other topologies.

To address this limitation, Pyretic also provides *network objects* (or simply *networks*), which allow programmers to abstract away details of the physical topology and write policies in terms of abstracted views of that network—providing an important form of modularity. We do this by allowing a new *derived* network to be built on top of an already existing *underlying* network (and, as implied by this terminology, Pyretic programmers may layer one derived network atop another).

Each network object has three key elements: a topology, a policy, and, for derived networks, a *mapping*. The topology object is simply a graph with switches as nodes and links as edges. The policy specifies the intended behavior of the network object with respect to that topology. The mapping comprises functions establishing an association between elements of the derived topology and those of its underlying topology.

The *base* network object represents the physical network. The Pyretic run-time system implements a discovery protocol that learns the physical topology using a combination of OpenFlow events and simple packet probes. The run-time system ultimately resolves all derived network policies into a single policy that can be applied to the base network.

A derived network object's mapping comprises the following functions:

- A function to map changes to the underlying topology up to changes on the derived topology, and
- A function to map policies written against the derived topology down to a semantically equivalent policy expressed only in terms of the underlying topology.

Pyretic provides several constructs for implementing these functions automatically. In most situations, the programmer simply specifies the mapping between elements of the topologies, along with a function for calculating forwarding paths through the underlying topology, and Pyretic calculates correct implementations automati-

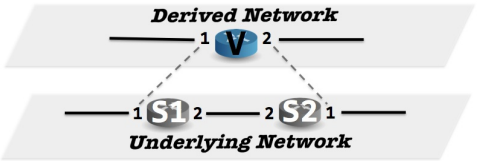

Figure 9: Derived "big switch" topology.

cally. The next few paragraphs describe these features in detail. For concreteness, we consider a running example where the derived network contains a single "big switch" as shown in Figure 9.

Transforming Topologies. The first step involved in implementing the "big switch" is to specify the relationship between elements of the underlying topology and elements of the derived topology. In this case, because the derived topology only contains a single switch, the association between underlying and derived topology elements can be computed automatically. To represent this association, we will use a dictionary that maps switch-port pairs `(vswitch,vport)` in the derived network to switch-port pairs `(switch,port)` in the underlying network. The following Python function computes a dictionary that relates ports on the switch in the derived network to ports at the perimeter of the underlying network:

```
def bfs_vmap(topo):
  vswitch = 1
  vport = 1
  for (switch, port) in topo.egress_locations:
      vmap[(vswitch, vport)] = (switch, port)
      vport += 1
  return vmap
```

Using this dictionary, it is straightforward to build the graph representing the derived topology—it consists of a single switch with the ports specified in the domain of the dictionary. Likewise, it is straightforward to map changes to the underlying topology up to changes for the derived topology. Pyretic provides code to implement these functions automatically from the `vmap` dictionary.

Transforming Policies. The next step involved in implementing the "big switch" is to transform policies written for the derived network into policies for the underlying network. This turns out to be significantly more challenging because it involves implementing a policy written against one topology, using the switches and links provided in a completely different topology. However, Pyretic's abstract packet model and support for sequential composition allow the transformation to be expressed in a clean and elegant way.

The transformation uses three auxiliary policies:[5]

- *Ingress Policy:* "lifts" packets in the underlying network up into the derived network by pushing appropriate switch and port identifiers onto the stack of values maintained in Pyretic's abstract packet model.
- *Egress policy:* "lowers" packets from the derived network to the underlying network by popping the switch and port identifier from the stack of values maintained in Pyretic's abstract packet model.
- *Fabric policy:* implements forwarding between adjacent ports in the derived network using the switches and links in the underlying topology. In general, calculating this policy involves computing a graph algorithm on the underlying topology.

[5] As before, Pyretic can automatically generate these from a vmap.

```
def virtualize(ingress_policy,
               egress_policy,
               fabric_policy,
               derived_policy):
  return if_(~match(vswitch=None),
    (ingress_policy >>
     move(switch=vswitch,
          inport=vinport) >>
     derived_policy >>
     move(vswitch=switch,
          vinport=inport,
          voutport=outport)),
    passthrough) >>
  fabric_policy >>
  egress_policy
```

Figure 10: Virtualization transformation.

With these auxiliary policies, the policy transformation can be expressed by composing several policies in sequence: ingress policy, derived policy, fabric policy, and egress policy. Figure 10 defines a general function `virtualize` that implements this transformation.

Example. To illustrate ingress, egress, and fabric policies, consider a specific physical topology consisting of two switches `S1` and `S2`, each with an outward-facing port and connected to each other by a link as shown in Figure 9. The policy running on the derived switch encodes the behavior of a repeater hub, as shown in Figure 5. The ingress policy is as follows:

```
ingress_policy =
( match(switch=S1, inport=1)
   [push(vswitch=V, vinport=1)]
| match(switch=S2, inport=1)
   [push(vswitch=V, vinport=2)])
```

It simply pushes the derived switch `V` and inport onto the corresponding "virtual" header stacks. The egress policy is symmetric:

```
egress_policy = match(vswitch=V)
  [if_(match(switch=S1, voutport=1)
    | match(switch=S2, voutport=2),
    pop(vswitch, vinport, voutport),
    passthrough)]
```

It pops the derived switch, inport, and outport from the appropriate virtual header stacks if the switch is labeled

with derived switch V, and otherwise passes the packet through unmodified. The fabric policy forwards packets labeled with derived switch V along the (unique) path between S1 and S2:

```
fabric_policy = match(vswitch=V)[
  ( match(switch=S1, voutport=1)[fwd(1)]
  | match(switch=S1, voutport=2)[fwd(2)]
  | match(switch=S2, voutport=1)[fwd(2)]
  | match(switch=S2, voutport=2)[fwd(1)])]
```

To illustrate these definitions, consider processing of a packet with the following headers.

```
{ switch:S1, inport:1, ... }
```

Recall that Pyretic's packet model treats the location of the packet as a header field. The first step of processing checks whether the packet has already entered the derived network, by testing for the presence of the vswitch header field. In this case, the packet does not contain this field so we treat it as being located at the ingress port of the derived switch and take the first branch of the conditional in Figure 10. Next, we evaluate ingress_policy which, by the first disjunct, pushes the headers vswitch=V and vinport=1 onto the packet, yielding a packet with the following headers:

```
{ switch:S1, inport:1,
  vswitch:V, vinport:1, ... }
```

Next we move the vswitch and vinport headers to switch and inport, and evaluate the policy written against the derived network (here simply flood). Flooding the packet on the derived network, generates a packet on outport 2 in this case:

```
{ switch:[V, S1], inport:[1, 1],
  outport:2, ...}
```

We then move the switch, inport, and outport headers to the corresponding virtual header stacks, which has the effect of restoring the original switch and inport headers,

```
{ switch:S1, inport:1,
  vswitch:V, vinport:1, voutport:2 }
```

and evaluate the fabric policy, which forwards the packet out port 2 of switch S1. Finally, the egress policy passes the packet through unmodified and the underlying topology transfers the packet to port 2 on switch S2:

```
{ switch:S2, inport:2,
  vswitch:V, vinport:1, voutport:2 }
```

This completes the first step of processing on the physical network. In the second step of processing, the packet already has virtual switch, inport, and outport labels. Hence, we do not calculate virtual headers as before and instead skip straight to the fabric policy, which forwards the packet out port 1 of S2. Now the packet does satisfy the condition stated in the egress policy, so it pops the virtual headers and forwards the packet out to its actual destination.

5 Example Pyretic Applications

To experiment with the Pyretic design, we have implemented a collection of applications. Table 2 lists a selection of these examples, highlighting the key features of Pyretic utilized, where the examples are discussed in the paper, and corresponding file names in the reference implementation [1]. Most terms in the features column should be familiar from prior sections. The term "novel primitives" simply refers to basic, but novel, features of Pyretic such as passthrough policies. Due to space constraints, we have omitted discussion of certain advanced Pyretic features that are needed to implement some applications including *traffic generation*, *topology-aware predicates*, *dynamic nesting*, and *recursion*. Section 5.1 elaborates on some of the additional applications found in our reference implementation and the key features they use. Section 5.2 concludes by presenting a "kitchen sink" example that utilizes all of Pyretic's features to write a truly modular application in just a few lines of code.

5.1 Pyretic Example Suite

ARP. The ARP application demonstrates how a Pyretic program can inject new packets into the network, and thereby respond to ARP traffic on behalf of hosts.

Firewalls. The firewall applications construct stateless (static) and stateful (dynamic) firewalls. These applications are similar in nature to the load balancer described in Section 3.2.2, but go a step further by demonstrating an advanced technique we call *dynamic nesting* in which one dynamic policy includes another dynamic policy within it. These firewalls also exploit topology-aware predicates such as ingress (which identifies packets at the network ingress) and egress (which identifies packets at the network egress).

Gateway. The gateway example implements the picture in Figure 3. The physical topology consists of three parts: an Ethernet island (switches 2, 3, and 4), a gateway router (switch 1), and an IP core (switches 5, 6, and 7). The gateway router is responsible for running several different pieces of logic. It is difficult to reuse standard components when all modules must share the same physical switch, so we abstract switch 1 to three switches (1000, 1001, 1002)—running MAC-learning, gateway logic, and IP routing, respectively. Unlike previous examples, the ports and links connecting these three switches are *completely virtual*—that is they map to no physical port, even indirectly. We encapsulate these components into a network object named GatewayVirt that performs the mechanical work of copying the base object and modifying it accordingly.

To a first approximation, here is how each of the virtu-

Examples	Pyretic Features Used	Section	File
Hub	static policy	3.2.2	`hub.py`
MAC-Learning	dynamic policy; queries; parallel comp.	3.2.2	`mac_learner.py`
Monitoring	static policy; queries; parallel comp.	3.2.2	`monitor.py`
Load Balancers	static policy; queries; parallel & sequential comp.; novel primitives	3.2.2	`load_balancer.py`
Firewalls	dynamic policy; queries; parallel & sequential comp.; novel primitives; topology-aware predicates; dynamic nesting	-	`firewall.py`
ARP	static policy; queries; parallel comp.; traffic generation	-	`arp.py`
Big Switch	static policy; topology abstraction; virtual headers; parallel comp.; novel primitives	4	`bfs.py`
Spanning Tree	static policy; topology abstraction; virtual headers; parallel comp.; novel primitives	-	`spanning_tree.py`
Gateway	static policy; topology abstraction; virtual headers; recursion; parallel & sequential comp.; novel primitives	-	`gateway.py`
Kitchen Sink	dynamic policy; topology abstraction; virtual headers; parallel comp.; novel primitives; topology-aware predicates; dynamic nesting	5.2	`kitchen_sink.py`

Table 2: Selected Pyretic examples.

alization components are implemented:[6]

- *Ingress policy*: Incoming packets to the physical gateway switch from the Ethernet side are tagged with `vswitch=1000` (and the appropriate `vinport`), and incoming packets to the physical gateway switch from the IP side are tagged with `vswitch=1002` (and the appropriate `vinport`).
- *Fabric policy*: For switches `1000-1002`, the fabric policy modifies the packet's virtual headers, effectively "moving" the packet one-step through the chain of switches. When moving the packet to a virtual port, the fabric policy recursively applies the entire policy (including ingress, fabric, and egress policies). The recursion halts when the packet is moved to a non-completely virtual port, at which time the packet is forwarded out of the corresponding physical port.
- *Egress policy*: As virtual links span at most one physical link, we strip the virtual headers after each forwarding action on the base network.

5.2 Putting it All Together

We conclude with an example application addressing the motivating "many-to-one" scenario discussed in Section 2.2 and shown in Figure 3. We implement the functionality of the Ethernet island by handling ARP traffic using the corresponding module from Table 2 and all other traffic with the familiar MAC-learning module.

```
eth = if_(ARP,dynamic(arp)(),dynamic(learn)())
```

We take the load balancer from Section 3.2.2 and combine it with a dynamic firewall from the examples table. This firewall is written in terms of white-listed traffic from client IPs to public addresses, creating another interesting twist— easily solved using Pyretic's operators. Specifically, the correct processing order of load balancer and firewall turns out to be *direction-dependent*. The firewall must be applied before load balancing for incoming traffic from clients—as the firewall must consider the original IP addresses, which are no longer be available after the load balancer rewrites the destination address that of a replica. In the other direction, the load balancer must first restore the original IP address before the firewall is applied to packets returning to the client.

```
fwlb = if_(from_client, afw >> alb, alb >> afw)
```

Finally, we complete our IP core by taking this combined load balancer/firewall and sequentially composing it with a module that implements shortest-path routing to the appropriate egress port—by running MAC-learning on a shortest path big switch!

```
ip = fwlb >> virtualize(dynamic(learn)(),
                        BFS(ip_core) )
```

The final component is our gateway logic itself. The gateway handles ARP traffic, rewrites source and destination MAC addresses (since these change on subnet transitions), and forwards out the appropriate port.

```
gw = if_(ARP,dynamic(arp)(),
             rewrite_macs(all_macs) >>
             ( eth_to_ip[fwd(2)] |
               ip_to_eth[fwd(1)] ))
```

We can then combine each of these policies, restricted to the appropriate set of switches, in parallel and run on the virtualized gateway topology discussed previously.

```
virtualize(in_(ethernet)[ eth ] |
           in_(gateway)[  gw  ] |
           in_(ip_core)[  ip  ],
           GatewayVirt(Recurse(self))
```

The `virtualize` transformation from Section 4 generates the ultimate policy that is executed on the base network.

6 Related Work

In recent years, SDN has emerged as an active area of research. There are now a number of innovative controller platforms based on the OpenFlow API [13] that make it possible to manage the behavior of a network using general-purpose programs [7, 12, 19, 2, 3, 24, 21]. Early

[6] See the reference implementation [1] for further details.

controller platforms such as NOX [7] offered a low-level programming interface based on the OpenFlow API itself; recent controllers provide additional features such as composition, isolation, and virtualization. We briefly review related work in each of these areas.

Composition. Pyretic's parallel and sequential composition operators resemble earlier work on the Click modular router [11]. However, rather than targeting multiple packet-processing modules within a single software router, Pyretic focuses on composing the control logic that affects the handling of traffic on an entire network of OpenFlow switches. Other controller platforms with some support for composition include Maestro [3], which allows programmers to write applications in terms of user-defined views of network state, and FRESCO [22], which provides a high-level language for defining security policies. Pyretic is distinguished from these systems in modeling policies as mathematical functions on packets, and in providing direct support for policy composition. In particular, unlike previous systems, Pyretic's composition operators do not require programmers to resolve conflicts by hand.

Isolation. To support multiple applications executing simultaneously in a single network, several controllers now support network "slices". Each slice may execute a different program while the controller provides isolation. One popular such controller is FlowVisor [21], a hypervisor that enforces strict traffic isolation between the controllers running on top of it, and also manages provisioning of shared resources such as bandwidth and the controller itself. Another recent proposal uses an extension to the NetCore compiler to provide traffic isolation by construction [8]. Finally, controllers that support the creation of virtual networks typically provide a form of isolation [16]. Pyretic's composition operators—in particular, sequential composition—make it straightforward to implement network slices. In addition, unlike controllers that only provide strict slicing, Pyretic can also be used to decompose a single application into different modules that affect the same traffic.

Network Objects. Pyretic's network objects generalize the global network views provided in other SDN controllers, such as NOX [7], ONIX [12], and POX [19]. In these systems, the network view (or network information base) represents the global topology as an annotated graph that can be configured and queried. Some systems go a step further and allow programmers to define a mapping between a representation of the physical topology and a simplified representation of the network. For example, the "big switch" abstraction [16, 4, 5, 20] can greatly simplify the logic of applications such as access control and virtual machine migration. Pyretic's network objects can be used to implement a wide range of abstract topologies. Moreover, as described in Section 4, sequential composition and virtual headers provide a simple and elegant mechanism for building derived network objects that implement a variety of abstract topologies. This mechanism is inspired by a technique for implementing virtual networks originally proposed by Casado et al. [4].

Programming Languages. This paper is part of a growing line of research on applying programming-language techniques to SDN [9, 24, 15, 6, 14]. Our early work on Frenetic [6, 14] introduced a functional language supporting parallel composition and SQL-like queries. This work goes further, by introducing an abstract packet model, sequential composition operator, and topology abstraction using network objects, as well as a new imperative implementation in Python. Taken together, these features facilitate "programming in the large" by enabling programmers to develop SDN applications in a modular way.

7 Conclusion

We believe the right level of abstraction for programmers is not a low-level interface to the data-plane hardware, but instead a higher-level language for writing and composing modules. Pyretic is a new language that allows SDN programmers to build large, sophisticated controller applications out of small, self-contained modules. It provides the programmatic tools that enable network programmers, operators, and administrators to master the complexities of their domain.

Acknowledgments. The authors wish to thank the NSDI reviewers, our shepherd Aditya Akella, Theophilus Benson, Marco Canini, Laurent Vanbever, and members of the Frenetic project for valuable feedback on earlier versions of this paper. This work was supported in part by the NSF under grants 0964409, 1111520, 1111698, 1117696 and TRUST; the ONR under award N00014-12-1-0757; the NSF and CRA under a Computing Innovations Fellowship; and a Google Research Award.

References

[1] Pyretic Reference Implementation. `http://www.frenetic-lang.org/pyretic`.

[2] Beacon: A Java-based OpenFlow Control Platform., Nov. 2010. See `http://www.beaconcontroller.net`.

[3] CAI, Z., COX, A. L., AND NG, T. S. E. Maestro: A System for Scalable OpenFlow Control. Tech. Rep. TR10-08, Rice University, Dec. 2010.

[4] CASADO, M., KOPONEN, T., RAMANATHAN, R., AND SHENKER, S. Virtualizing the Network Forwarding Plane. In *ACM PRESTO* (Nov. 2010).

[5] CORIN, R., GEROLA, M., RIGGIO, R., DE PELLEGRINI, F., AND SALVADORI, E. VeRTIGO: Network Virtualization and Beyond. In *EWSDN* (Oct. 2012), pp. 24–29.

[6] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A Network Programming Language. In *ACM ICFP* (Sep. 2011), pp. 279–291.

[7] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. *ACM SIGCOMM CCR 38*, 3 (2008).

[8] GUTZ, S., STORY, A., SCHLESINGER, C., AND FOSTER, N. Splendid Isolation: A Slice Abstraction for Software-Defined Networks. In *ACM SIGCOMM HotSDN* (Aug. 2012), pp. 79–84.

[9] HINRICHS, T. L., GUDE, N. S., CASADO, M., MITCHELL, J. C., AND SHENKER, S. Practical Declarative Network Management. In *ACM SIGCOMM WREN* (Aug. 2009), pp. 1–10.

[10] KELLER, E., AND REXFORD, J. The 'Platform as a Service' Model for Networking. In *IMN/WREN* (Apr. 2010).

[11] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK., M. F. The Click Modular Router. *ACM TOCS 18*, 3 (Aug. 2000), 263–297.

[12] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., HAMA, T., AND SHENKER, S. Onix: A Distributed Control Platform for Large-scale Production Networks. In *USENIX OSDI* (Oct. 2010), pp. 351–364.

[13] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR 38*, 2 (2008), 69–74.

[14] MONSANTO, C., FOSTER, N., HARRISON, R., AND WALKER, D. A compiler and run-time system for network programming languages. In *ACM POPL* (Jan. 2012), pp. 217–230.

[15] NAYAK, A., REIMERS, A., FEAMSTER, N., AND CLARK, R. Resonance: Dynamic Access Control in Enterprise Networks. In *ACM SIGCOMM WREN* (Aug. 2009), pp. 11–18.

[16] NICIRA. It's time to virtualize the network, 2012. `http://nicira.com/en/network-virtualization-platform`.

[17] NICIRA. Networking in the era of virtualization. `http://nicira.com/sites/default/files/docs/Networking%20in%20the%20Era%20of%20Virtualization.pdf`, 2012.

[18] PERLMAN, R. Rbridges: Transparent Routing. In *IEEE INFOCOM* (Mar. 2004), pp. 1211–1218.

[19] POX. `http://www.noxrepo.org/pox/about-pox/`.

[20] SHENKER, S. The Future of Networking and the Past of Protocols, Oct. 2011. Invited talk at Open Networking Summit.

[21] SHERWOOD, R., GIBB, G., YAP, K.-K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the Production Network Be the Testbed? In *USENIX OSDI* (Oct. 2010), pp. 365–378.

[22] SHIN, S., PORRAS, P., YEGNESWARAN, V., FONG, M., GU, G., AND TYSON, M. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Internet Society NDSS* (Feb. 2013). To appear.

[23] TOUCH, J., AND PERLMAN, R. Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement, May 2009. IETF RFC 5556.

[24] VOELLMY, A., AND HUDAK, P. Nettle: Functional Reactive Programming of OpenFlow Networks. In *ACM PADL* (Jan. 2011), pp. 235–249.

[25] WEBB, K. C., SNOEREN, A. C., AND YOCUM, K. Topology switching for data center networks. In *USENIX HotICE* (Mar. 2011).