Modular SDN Programming with Pyretic

JOSHUA REICH, CHRISTOPHER MONSANTO, NATE FOSTER,
JENNIFER REXFORD, AND DAVID WALKER

Joshua Reich is an NSF/CRA Computing Innovation Fellow at Princeton University's Department of Computer Science. He designs and builds systems to utilize networks more effectively—currently focusing on SDNs. His work on Pyretic received the NSDI Community Award (shared with this article's co-authors). jreich@cs.princeton.edu

Christopher Monsanto is a Ph.D. candidate at Princeton University, advised by David Walker. His research interests include programming languages and distributed computing. chris@monsan.to

Nate Foster is an Assistant Professor of Computer Science at Cornell University. His research focuses on abstractions and tools for building reliable systems. jnfoster@cs.cornell.edu

Jennifer Rexford is the Gordon Y.S. Wu Professor of Engineering in the Computer Science Department at Princeton University. She previously worked at AT&T Research, where she designed network-management techniques that were deployed in AT&T's backbone network. jrex@cs.princeton.edu

David Walker is a Professor of Computer Science at Princeton University. His research focuses on the theory, design, and implementation of programming languages. dpw@cs.princeton.edu

Software-Defined Networking (SDN) enables innovation in network management by giving a programmable controller direct control over the underlying switches through an open, standard API, like OpenFlow. However, existing SDN controllers offer programmers a low-level programming interface akin to assembly language. In this article, we present Pyretic, a programming platform that raises the level of abstraction and enables the creation of modular software, allowing programmers to create sophisticated SDN applications.

Managing today's computer networks is a complex and error-prone task. These networks consist of a wide variety of devices, from routers and switches to firewalls, network-address translators, load balancers, and intrusion-detection systems. Network administrators must express policies through tedious box-by-box configuration, while grappling with a multitude of protocols and baroque, vendor-specific interfaces.

In contrast, Software-Defined Networking (SDN) is redefining the way we manage networks. In SDN, a controller application uses a standard, open interface, such as OpenFlow [1], to specify how network elements or switches should handle incoming packets. Programmers develop their own new controller applications on top of a controller platform, which provides a programming API built on top of OpenFlow. Separating the controller platform and applications from the network elements allows anyone—not just the equipment vendors—to program new network control software.

In just a few years, SDN has enabled a wealth of innovation, including prominent commercial successes such as Nicira's network virtualization platform and Google's wide-area traffic-engineering system. Most of the major switch vendors support the OpenFlow API, and many large information-technology companies are involved in SDN consortia, such as the Open Networking Foundation and the Open Daylight initiative.

SDN is creating exciting new opportunities for network-savvy software developers and software-savvy network practitioners alike. But how should programmers write these controller applications? The first generation of SDN controller platforms offer programmers a low-level API closely resembling the interface to the switches. This forces programmers to program in "assembly language," by manipulating bit patterns in packets and carefully managing the shared rule-table space.

In the Frenetic Project [2], we are designing simple, reusable, high level abstractions for programming SDNs; and efficient runtime systems that automatically generate and install the corresponding low-level rules on switches [3–7]. Our abstractions cover the main facets of managing a network-specifying packet-forwarding policy, monitoring network conditions, and dynamically updating policy to respond to network events. In this article, we describe Pyretic, our Python-based platform that embodies many of these concepts, and enables systems programmers to create sophisticated SDN applications.

Pyretic is open-source software that offers a BSD-style license compatible with the needs of both commercial and research developers. Both the source code for, and a pre-packaged VM

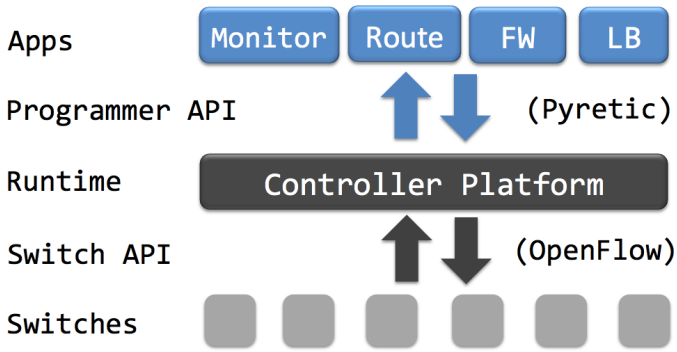

Figure 1: Software Defined Network (SDN)

containing, Pyretic's core policy language, libraries, and runtime are available on the Pyretic home page [8], along with documentation, video tutorials, links to our email discussion list, and more. Feel free to download and run any of the Pyretic examples covered in the article.

OpenFlow

Pyretic is both a response to the shortcomings of OpenFlow as a programmer API, and a client of OpenFlow in its role as an API to network switches. As such, we begin with a brief review of OpenFlow.

OpenFlow Switches

An OpenFlow switch has a rule table, where each rule includes:

- a bit pattern: including wildcards, for matching header fields—for example, MAC and IP addresses, protocol, TCP/UDP port numbers, physical input port, etc.;
- a priority: to break ties between overlapping patterns;
- a list of actions: for example, forward out a port, flood, drop, send to controller, assign a new value to a header field, etc.;
- optional hard and soft timeouts to evict stale rules;
- byte and packet counters that collect information about how much traffic is flowing through each rule.

Upon receiving a packet, the switch finds the highest-priority matching rule, applies each action, and updates the counters. Newer versions of OpenFlow support additional header fields and multiple stages of tables.

OpenFlow Controllers

The OpenFlow protocol defines how the controller and switches interact. The controller maintains a connection to each switch over which OpenFlow messages are sent. The controller uses these OpenFlow messages to (un)install rules, query the traffic counters, learn the network topology, and receive packets when the switch applies the "send to controller" action. Most existing controller platforms offer programmers an API that is a thin "wrapper" around these operations. Applications are expressed as event handlers that respond to events such as packet arrivals, topology changes, and new traffic statistics.

Controller Applications

OpenFlow has enabled a wealth of controller applications, including flexible access control, Web server load balancing, energy-efficient networking, billing, intrusion detection, seamless mobility and virtual-machine migration, and network virtualization. As an example, consider "MAC learning"—an application designed to detect the arrival of new hosts, discover their MAC addresses, and route packets to them. To begin, the application starts by installing a default rule in each edge switch that matches all packets and sends them to the controller. Upon receiving a packet, the application learns the location (i.e., the switch and input port) of the sender. If the receiver's location is already known, the application installs rules that direct traffic in both directions over a shortest path from one to the other; otherwise, the application instructs the switch to flood—broadcasting the packet to all possible receivers. If a host moves to a new location, the default rule at the new switch sends the next packet to the controller, allowing the application to learn the host's new location and update the paths that carry traffic to and from the host. Consequently, hosts can continue communicating without disruption, even when one or both hosts move.

Pyretic Language

Pyretic encourages programmers to focus on how to specify a network policy at a high level of abstraction, rather than how to implement it using low-level OpenFlow mechanisms. In particular, instead of implementing a policy by incrementally installing physical rule after physical rule on switch after switch, a Pyretic policy is specified for the entire network at once, via a function from an input located packet (i.e., a packet and its location) to an output set of located packets. The output packets can have modified fields and usually end up at new locations—this is how packet forwarding occurs. The programmer does not need to worry about which OpenFlow rules are used to move packets from place to place.

One of the primary advantages of Pyretic's policies-as-abstract-functions approach to SDN programming is that it helps support modular programming. In traditional OpenFlow programming, the programmer cannot write application modules independently

without worrying that they might interfere with one another. Rather than forcing programmers to carefully merge multiple pieces of application logic by hand, a Pyretic program can combine multiple policies together using one of several *policy composition operators*, including *parallel composition* and *sequential composition*.

On existing SDN controller platforms, monitoring is merely a side-effect of installing rules that send packets to the controller, or accumulate byte and packet counters. Programmers must painstakingly create rules that simultaneously monitor network conditions and perform the right forwarding actions. Instead, Pyretic integrates monitoring into the policy function and supports a high level query API. The programmer can easily combine monitoring and forwarding using parallel composition. Pyretic also provides facilities for creating a dynamic policy whose behavior will change over time, as specified by the programmer. Composition operators can be applied to these dynamic policies just as easily as fixed static ones.

Finally, Pyretic offers a rich topology-abstraction facility that allow programmers to apply policy functions to an abstract view of the underlying network. This facility is particularly noteworthy in that it is actually an application built on top of Pyretic using the other abstractions in the language.

In this section, we illustrate the features of the language using examples. Along the way, we build toward a single-switch Pyretic application that dynamically splits incoming traffic across several server instances. We conclude by using topology abstraction to distribute this single-switch application across a network of many switches.

Network Policy as a Function

A controller application determines the policy for the network at any moment in time. A conventional OpenFlow program includes explicit logic that creates and sends rule-installation messages to switches (logic that includes defining the low-level bit-match patterns, priorities, and actions for these rules) and that registers callbacks that poll traffic counters and handle packets sent to the controller.

In contrast, Pyretic hides these low-level details by allowing programmers to express policies as compact, abstract functions that take a packet (at a given location) as input, and return a set of new packets (at potentially different locations). Returning the empty set corresponds to dropping the packet. Returning a single packet corresponds to forwarding the packet to a new location. Returning multiple packets corresponds to multicast.

The simplest possible Pyretic policy is one where every switch floods each packet out all ports on the network spanning tree. In conventional OpenFlow programming, the controller application would, for each switch, install the rule whose pattern is "don't care" on all bits, with a single action "flood" (if that action is even supported by the switch). In contrast, in Pyretic, the programmer simply writes one line:

```
flood()
```

where flood() is interpreted as a function that takes a packet located at any port on any switch in the network as an input and outputs zero, one, or more copies of the same packet at the output ports of the switch it arrived at—one packet for each port on the network's spanning tree. Hence, this simple policy will allow any collection of hosts to broadcast information to one another over a network. Moreover, the policy no longer depends upon specific switch features. The switches used need not implement a "flood" primitive themselves as the runtime system can choose to implement flooding behavior using other OpenFlow actions—a good thing because the "flood" action is an optional feature in OpenFlow 1.0.

Of course, Pyretic programmers will typically write much more sophisticated policies. Here's a fragment of a policy that uses several more Pyretic features to route a packet with destination IP 10.0.0.1 across switches A and B.

```
(match(switch=A) & match(dstip='10.0.0.1') >> fwd(6)) +
(match(switch=B) & match(dstip='10.0.0.1') >> fwd(7))
```

Here, we use *predicate policies* (including match and conjunction) to disambiguate between packets based on their location in the network as well as their contents; we use *modification policies* (such as `fwd`) to change the header content or location of packets; and we use *composition operators* (such as +, parallel composition and >>, sequential composition) to put together policy components. Each of these features, as well as others, will be explained in the upcoming sections; Table 1 lists several of the most common basic Pyretic policies.

In this slightly more elaborate policy, there are components that look somewhat like OpenFlow rules—they match different kinds of packets and perform different actions; however, as the simpler flood example shows, these policies do not necessarily map to OpenFlow rules in a one-to-one fashion. Consequently, Pyretic programmers must discard the rule-based mental programming model and adopt the functional one. We believe doing so encourages programmers to focus their minds entirely on the essential problem: determining the fundamental, high-level logic required to implement the application properly, not the low-level encoding of that logic in terms of hardware abstractions and a series of controller-level event handlers. This also leads to much more concise code, avoids replicating related functionality, and reduces the risk of accidental inconsistencies between different parts of the application.

Syntax	Summary
`identity`	returns original packet
`drop`	returns empty set
`match(f=v)`	`identity` if field `f` matches `v`, `drop` otherwise
`modify(f=v)`	returns packet with field `f` set to `v`
`fwd(a)`	`modify (port=a)`
`flood()`	returns one packet for each local port on the network spanning tree

Table 1: Selected policies

From Bit Patterns to Boolean Predicates

An OpenFlow rule matches packets based on a bit pattern in the header fields, where each bit is a 0, 1, or "don't care"; however, expressing a policy in terms of bit patterns is tedious. For example, matching all packets except those with a destination IP address of 10.0.0.1 requires two rules. The first, higher-priority rule matches all packets destined to 10.0.0.1, so that all remaining packets "fall through" to the second, lower-priority rule that has a wildcard in each bit position. Similarly, matching either 10.0.0.3 or 10.0.0.4 requires two rules, one for each IP address (as there is no single bit-pattern that matches both).

Instead of bit patterns in packet-header fields, Pyretic allows programmers to write basic predicates of the form `match(f=v)`, demanding that a field `f` match an abstract value `v` (such as an IP address). They can then construct more complicated predicates using standard Boolean operators such as and (&), or (|), and not (~). Intuitively, all these predicates act as filters: If the incoming packet satisfies the predicate, the packet passes through the filter untouched, presumably to be processed in some interesting way by some subsequent part of the policy. If the incoming packet does not satisfy the predicate, it is dropped (i.e., the empty set of packets is generated as a result). For example, the Pyretic programmer simply writes

```
~match(dstip='10.0.0.1')
```

or

```
match(switch=A) &
 (match(dstip='10.0.0.3') | match(dstip='10.0.0.4'))
```

and the runtime system ensures that packets are filtered accordingly.

Virtual Packet Header Fields

A policy function in Pyretic can match on a packet-header field (using `match(f=v)`), and can assign a new value to a header field (using `modify(f=v)`). As we have seen, the fields available to the programmer include the standard physical OpenFlow packet header fields, such as source and destination IP; however, unlike OpenFlow packets, Pyretic packets provide a single unified abstraction for both the packet and its associated metadata. To this end, Pyretic packets also include standard virtual fields switch and port that together specify a packet's location in the network. In fact, the `fwd` policy we saw previously is actually just a special case of `modify`! Reassigning the value of port simply "moves" the packet from the port on which it arrived to the port on which it will be sent. The burden of managing all the details needed to ensure that each packet is forwarded out the correct hardware port is left to the Pyretic runtime.

Finally, Pyretic programmers are free to define their own, new virtual fields and use them however they choose, treating each Pyretic packet as if it were a Python dictionary. For example, a programmer may want to assign a packet to one of several paths through a network. Tagging the packet with the chosen path makes it easier to direct the packet over each of the hops in the path. In Pyretic, the programmer could create a new path field and assign it a particular path identifier. Here again, the burden of realizing this falls to the Pyretic runtime, which might, under the hood, represent the appropriate information using a conventional packet tagging mechanism such as VLANs or MPLS labels.

Parallel and Sequential Composition

A controller application often needs to perform multiple tasks (e.g., routing, server load balancing, monitoring, and access control) that affect handling of the same traffic. Rather than writing one monolithic program, programmers should be able to combine multiple independently written modules together. In traditional OpenFlow programming, different modules could easily interfere with each other. One module might overwrite the rules installed by another, or drop packets another module expects to see at the controller. Instead, Pyretic offers two simple composition operators that allow programmers to combine policies in series or in parallel.

SEQUENTIAL COMPOSITION

Sequential composition (>>) treats the output of one policy as the input to another. Consider a simple routing policy:

```
match(dstip='2.2.2.8') >> fwd(1)
```

In this policy, the match predicate filters out all packets that do not have destination 2.2.2.8. The >> operator places this filter in sequence with the forwarding policy `fwd(1)`. Hence any packets that pass through the filter are forwarded out port 1. Likewise, the programmer may write

```
match(switch=1) >> match(dstip='2.2.2.8') >> fwd(1)
```

to specify that packets located at switch 1 and destined to IP address 2.2.2.8 should be forwarded out port 1. This code uses

sequential composition to compose three independent policies. The first two policies happen to be filters (though they may be arbitrary policies). Of course, filtering packets first by one condition and then by a second condition is equivalent to filtering packets by the conjunction (&) of the two conditions.

PARALLEL COMPOSITION

Parallel composition (+) applies two policy functions on the same packet and combines the results. For example, a routing policy R could be expressed as

```
R = (match(dstip='2.2.2.8') >> fwd(1)) +
    (match(dstip='2.2.2.9') >> fwd(2))
```

Those packets destined to 2.2.2.8 will be forwarded out port 1, while those destined to 2.2.2.9 will be forwarded out port 2.

As another example, consider a server load-balancing policy that splits request traffic directed to destination 1.2.3.4 over two backend servers (2.2.2.8 and 2.2.2.9), depending on the first bit of the source IP address (packets with sources starting with 0 fall under IP prefix 0.0.0.0/1 and are routed to 2.2.2.8). This results in the policy:

```
L = match(dstip='1.2.3.4') >>
     ((match(srcip='0.0.0.0/1') >> modify(dstip='2.2.2.8')) +
      (~match(srcip='0.0.0.0/1') >> modify(dstip='2.2.2.9')))
```

This policy happens to adhere to a particularly common pattern: a clause matching one predicate is immediately followed by a clause matching its negation. Of course, in conventional programming languages, such patterns are just if statements. In Pyretic, if_ is an abbreviation that makes policies easier to read:

```
L = match(dstip='1.2.3.4') >>
    if_(match(srcip='0.0.0.0/1'),
        modify(dstip='2.2.2.8'),
        modify(dstip='2.2.2.9'))
```

CODE REUSE

One final example highlights the power of Pyretic's composition operators to enable modular programming. In just one line, the programmer can write

```
L >> R
```

producing a new policy that first selects a server replica and then forwards the traffic to that chosen replica. As simple as it seems, this kind of composition is impossible to achieve when programming directly against the OpenFlow API.

Traffic Monitoring

In traditional OpenFlow programs, collecting traffic statistics involves installing rules (so that byte and packet counters are available), issuing queries to poll these counters, parsing the responses when they arrive, and combining counter values across multiple rules.

Syntax	Summary

`packets(`
`  limit=n,`
`  group_by=[f1,f2,...])`
	callback on every packet received for up to n packets identical on fields f1,f2,...

`count_packets(`
`  interval=t,`
`  group_by= [f1,f2,...])`
	count every packet received callback every t seconds providing count for each group

`count_bytes(`
`  interval=t,`
`  group_by=[ f1,f2,...])`
	count every byte received callback every t seconds providing count for each group

Table 2: Query policies

In Pyretic, network monitors are just another simple type of policy that may be conjoined to any of the other policies seen so far. Table 2 shows several different kinds of monitoring policies available in Pyretic, including policies that monitor raw packets, packet counts, and byte counts. The forwarding behavior of these policies is the same as a policy that drops all packets.

For example, a programmer may create a new query for the first packet arriving from each unique source IP

```
Q = packets(limit=1,group_by=['srcip'])
```

and restrict it to Web-traffic requests (i.e., packets destined to TCP port 80):

```
match(dstport=80) >> Q
```

To print each packet that arrives at Q, the programmer registers a callback routine to handle Q's callback,

```
def printer(pkt):
  print pkt


Q.register_callback(printer)
```

The runtime system handles all of the low-level details of supporting queries—installing rules, polling the counters, receiving the responses, combining the results as needed, and composing query implementation with the implementation of other policies. For example, suppose the programmer composes the example monitoring query with a routing policy that forwards packets based on the destination IP address. The runtime system ensures that the first TCP port 80 packet from each source IP address reaches the application's printer routine, while guaranteeing that this packet (and all subsequent packets from this source) is forwarded to the output port indicated by the routing policy.

Writing Dynamic Policies

Query policies are often used to drive changes to other dynamic policies. These dynamic policies have behavior (defined by `self.policy`) that changes over time, according to the programmer's specification.

For example, the routine `round_robin` takes the first packet from a new client (source IP address) and updates the policy's behavior (by assigning `self.policy` to a new value), so all future packets from this source are assigned to the next server in the sequence (by rewriting the destination IP address); packets from all other clients are treated as before. After updating the policy, `round_robin` also moves the "currently up" server to the next server in the list.

```
def round_robin(self,pkt):
  self.policy = if_(match(srcip=pkt['srcip']),
                    modify(dstip=self.server),
                    self.policy)
  self.client += 1
  self.server = self.servers[self.client % m]
```

The programmer creates a new "round-robin load balancer" dynamic policy class `rrlb` by subclassing `DynamicPolicy` and providing an initialization method that registers `round_robin` as a callback routine:

```
class rrlb(DynamicPolicy):
  def __init__(self,s,servers):
    self.switch = s
    self.servers = servers
    ...
    Q.register_callback(self.round_robin)
    self.policy = match(dstport=80) >> Q

  def round_robin(self,pkt):
    ...
```

Note that here the query Q is defined as in the previous subsection; the only difference is that the programmer registers `round_robin` as the callback, instead of printer. The programmer then creates a new instance of `rrlb` (say, one running on switch 3 and sending requests to server replicas at 2.2.2.8 and 2.2.2.9) in the standard way

```
servers = ['2.2.2.8','2.2.2.9']
rrlb_on_switch3 = rrlb(3,servers)
```

producing a policy that can be used in exactly the same ways as any other. For example, to compose server load balancing with routing, we might write the following:

```
rrlb_on_switch3 >> route
```

Topology Abstraction

In traditional OpenFlow programming, a controller application written for one switch cannot easily be ported to run over a distributed collection of switches, or be made to share switch hardware with other packet-processing applications. In the case of our load balancer example, we may well want to use it to balance load coming in from many different hosts connected to many different switches in a complex network. And yet, we would prefer to avoid conflating the relatively simple functionality of the load balancer with the logic needed to route the traffic across the network. A good solution to this problem is to use topology abstraction to partition the application into two pieces: one that does the load balancing as before, as if the balancer was implemented on one big switch that could connect all hosts together, and one that decides on the lower level routes that implement it. This also serves a secondary purpose: the load balancer is reusable and can operate over any network of switches.

To develop this kind of modular program, Pyretic offers a library for topology abstraction that can represent multiple underlying switches as a single derived virtual switch, or, alternatively, one underlying switch as multiple derived virtual switches.

For example, to produce a policy that applies the client policy `rrlb_on_switch3` to a derived (i.e., virtual) switch 3 that abstracts switches 1, 2, and 3 as a single merged switch, the programmer simply uses Pyretic's virtualize function, inputting the desired policy function and the topology transformation:

```
virtualize(rrlb_on_switch3,
           merge(name=3,
                 from_switches=[1,2,3]))
```

Here, the `merge` topology transformation takes the name of a single virtual switch and a list of underlying switches that used to create it. Inside, the `merge` transformation applies shortest-path routing to direct packets from one edge link to another over the underlying switches. `merge` encodes this transformation in three auxiliary policies—one that handles incoming traffic, one that handles traffic passing through the derived switch, and one that handles traffic leaving the switch.

The virtualize policy then implements a transformation of the written policies (the client policy and three auxiliary policies) using virtual header fields and sequential composition to produce a single new policy written for the underlying network [6]. The resulting policy is exactly the same as any other Pyretic policy, and can be both composed with other policies, or used as the basis for yet another layer of virtualization.

Pyretic Runtime

Of course, high level programming abstractions are only useful if they can be implemented efficiently on the switches. This section provides a brief overview of the Pyretic runtime system, focusing on the backend interface to the OpenFlow switches and policy evaluation.

Backend Interface

Pyretic's runtime is designed to be used atop a variety of different OpenFlow controller backends. The Pyretic runtime connects via a standard socket to a simple OpenFlow client that could be written on top of any OpenFlow controller platform. The runtime manipulates the network by sending messages to the client (e.g., to inject packets, modify rules, and issue counter reads). Likewise messages from the client keep Pyretic updated regarding network events (e.g., packet ins, port status events, counter values read). This design enables Pyretic to take advantage of the best controller technology available, and allows the system to be entirely self-contained. The current Pyretic runtime comes packaged with an OpenFlow client written on the popular POX controller platform.

Policy Evaluation

The Pyretic runtime implements an interpreter that evaluates an input packet against the current policy. In its simplest mode of operation, all packets are initially evaluated by this interpreter. Concurrently, the runtime keeps track of currently active queries, updates to dynamic policies, and modifications to the network topology. On its general setting, when it is safe to do so, the runtime proactively installs rules on switches before they are needed, to avoid unnecessary switch-controller latency. For more information on the current runtime implementation, please see the Pyretic home page [8].

Conclusions

Pyretic lowers the barrier to creating sophisticated SDN applications and comes with several example of common enterprise and datacenter network applications (e.g., hub, MAC-learning switch, traffic monitor, firewall, ARP server, network virtualization, and gateway router). Since the initial release of Pyretic in April 2013, the community of developers has grown quickly. Some have built new applications from scratch, while others have ported systems originally written on other platforms.

In one case, the Resonance [9] system for event-driven control was rewritten in Pyretic, taking approximately one programmer-day and resulting in a six-fold reduction in code size over an earlier version written on the NOX controller platform. These savings were realized thanks to Pyretic's declarative design and powerful yet concise policy language. Short expressions involving basic policies, such as `match` and `fwd`, combined with composition operators to replace complex code specifying various packet handlers and the logic they contained: packet matching, modification and injection, as well as OpenFlow rule construction and installation. In fact, Pyretic's focus on modular design enabled the Resonance team to encode more sophisticated policies than had been available in the NOX version.

Pyretic has also been featured in Georgia Tech's SDN Coursera course [10] where it was used as the platform for one of the course's three programming assignments.

In addition to enhancing our runtime system with enhanced compilation support, in our ongoing work we are also making extensions to the language and runtime system to support new features, such as quality-of-service mechanisms and parsing of packet contents. Additionally, we are creating more sophisticated applications, including RADIUS and DHCP services (to authenticate end hosts and assign them IP addresses) and wide-area traffic-management solutions for Internet Service Providers at SDN-enabled Internet Exchange Points.

We welcome newcomers to our community, whether they are interested in using Pyretic or in contributing to its development. Please visit our Web site, join our discuss list, or email us.

Acknowledgments

Our work is supported in part by ONR grant N00014-09-1-0770 and NSF grants 1111698, 1111520, 1016937, 1253165, and 0964409, a Sloan Research Fellowship, and a NSF/CRA Computing Innovation Fellowship. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the NSF, CRA, ONR, or the Sloan Foundation.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," SIGCOMM CCR, vol. 38, no. 2 (2008), pp. 69-74.

[2] The Frenetic Project: http://www.frenetic-lang.org.

[3] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," ACM ICFP, Sept. 2011.

[4] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A Compiler and Run-Time System for Network Programs," POPL, Jan. 2012.

[5] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," ACM SIGCOMM, Aug. 2012.

[6] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-Defined Networks," USENIX NSDI, 2013.

[7] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Katta, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story, and D. Walker, "Languages for Software-Defined Networks," IEEE Communications, vol. 51 (Feb 2013), pp. 128-134.

[8] Pyretic home page: http://www.frenetic-lang.org/pyretic.

[9] Resonance Project: http://resonance.noise.gatech.edu.

[10] Coursera course on SDN: https://www.coursera.org/course/sdn.