Simpler Network Configuration with State-Based Network Policies

Hyojoon Kim, Arpit Gupta, Muhammad Shahbaz, Joshua Reich*, Nick Feamster, Russ Clark
*Georgia Tech, *Princeton University*
`http://resonance.noise.gatech.edu/`

Abstract

Operators make hundreds of changes to a network's router and switch configurations every day—a painstaking, error-prone process. If the network configuration could instead encode different forwarding behavior for different network states *a priori*, a network controller could automatically alter forwarding behavior when conditions change. To enable this capability, we introduce *state-based network policies*, which describe how a network's forwarding behavior should change in response to arbitrary network events. A state-based network policy comprises many *tasks*, each of which encodes the forwarding behavior for a single network management operation (*e.g.*, intrusion detection) or part of the network (*e.g.*, a sub-organization), and how that behavior should change when network conditions change. Composing these policies produces a network-wide control program that adapts to different operating conditions. We implement state-based network policies in a system called PyResonance and demonstrate with real-world examples and use cases that PyResonance is expressive enough to specify a wide range of network policies and simple enough for many operators to use. Our evaluation based on event traces from the Georgia Tech campus network shows that PyResonance can achieve good performance in operational settings.

1 Introduction

Network management is complex and error-prone; this complexity and brittleness has several causes. First, much of today's network management process remains low-level and *manual*: operators must update router and switch configuration to alter the network's forwarding behavior (*e.g.*, quarantining, garden-walling, or rate limiting a host), because, aside from routing around failures, network forwarding behavior does not automatically change in response to most events. Operators must continually modify low-level network configuration in response to events ranging from traffic surges to intrusions, making hundreds of changes to the switch and router configurations in a network every day [19]. Additionally, network management is often *federated*, even in a single domain. Network configuration involves composing many independent tasks and integrating configurations written by multiple sub-organizations (*e.g.*, a security and traffic engineering team working on a backbone network, or individual departments working in an enterprise). Although some existing tools help network operators "automate" these low-level processes [4, 24], these tools are generally wrappers around command-line interfaces that still leave most of the low-level work in the hands of the network operator.

Until recently, it was difficult to change this state of affairs because network configuration was so tightly coupled with the devices themselves. Software Defined Networking (SDN) offers the opportunity to write network-wide control programs that are simpler and easier to manage than existing network configuration. Yet, while SDN makes it *possible* to write control programs, it does not specify *how* network control should be designed to simplify network management. In their design of 4D, Greenberg *et al.* state [16]: "*The decision plane must react in real time to network events...Identifying the right abstractions to support rapid reactions to unplanned events...is an important and challenging research problem.*" Unfortunately, the events that current SDN controllers can respond to are generally limited to packet arrival events and topological changes; these controllers cannot easily incorporate many of the network events that make network management so difficult (*e.g.*, security events, traffic shifts).

Network control should instead allow an operator to encode *a priori* how forwarding behavior should change when the state of the network changes, for arbitrary events. Enabling this level of automation would eradicate many manual aspects of network configuration. Of course, to avoid making problems worse, operators need a natural way to express how the network's forwarding behavior should change when different types of events occur. To facilitate this capability, we introduce *state-based network policies*, which describe how a network's forwarding behavior should change in response to arbitrary network events. Such a policy specifies one or more *tasks*, each of which corresponds to a specific network management operation (*e.g.*, intrusion detection) and specifies which network control program should execute when a state transition takes place. An operator can define tasks that encode different network states, the specific control program that should run to execute that task when the network is in a certain state, and how different network events should cause transitions between states. For example, a task could specify that a host should be quarantined when an intrusion detection system determines that it is compromised, or that it should be rate-limited if it the network monitoring system determines that it exceeds a certain usage quota.

A network-wide policy that incorporates a broad range of events involves many independent operations (*e.g.*, traffic engineering, security, resource management) and even more possible states. Requiring a single network operator to specify the network's forwarding behavior for every combination of states is intractable. Additionally, network configuration is federated, meaning that different operators and groups may write different parts of the configuration, so there needs to be an easy way to compose independent tasks (and corresponding programs) into more complex ones. To solve this problem, we use existing SDN composition operators [20] to decompose a large monolithic state-based policy into smaller and simpler state-based tasks; this process improves readability, makes it easier to analyze a configuration for correctness, and enables reuse. It reduces the state complexity of specifying a state-based network policy from exponential in the number of tasks to linear. The composition operators also enable *federated* configuration, the composition of distinct control programs from different sub-organizations.

We implement state-based network policies in a system called *PyResonance*. We implemented PyResonance in Pyretic, a Python-embedded domain specific language (DSL) for specifying and composing different network programs. Pyretic [20] provides compositional constructs for building modular control programs that can be combined to produce more complex ones. PyResonance uses Pyretic's composition operators to allow operators to specify state-based network policies by composing one or more tasks, each of which has a finite state machine and a corresponding network program that determines forwarding behavior for that state. PyResonance allows operators to specify tasks, event streams for each task, and how each task should modify its corresponding network control program when different events occur. It then uses Pyretic to compose the resulting programs from each task into a single network-wide control program.

We evaluate the expressiveness, complexity, and performance of PyResonance. To demonstrate expressiveness, we show how PyResonance can enable easy implementation of many network tasks. To demonstrate how PyResonance reduces complexity, we implemented several state-based tasks in PyResonance that incorporate authentication, intrusion, and resource usage events; each required no more than about 60 lines of Python. We also describe some experiences of programmers who have used PyResonance to date; for example, in a recent Coursera SDN course [8], more than 95% of about 800 people who attempted to implement a load balancer in PyResonance [9] successfully completed the assignment. Our evaluation of PyResonance's performance based on authentication events from

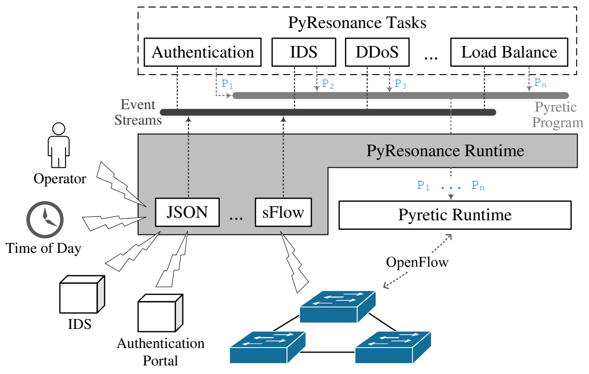


Figure 1: *PyResonance architecture.*

the Georgia Tech network (which sees about 1.7 million authentication events per day) shows that PyResonance can perform well in operational environments.

This paper offers four contributions. First, we develop a new primitive—a state-based network task (§3)—that allows operators to implement policies that specify how network behavior should change in response to a wide variety of events. Second, we demonstrate that the composition operators provided in Pyretic can help operators compose these tasks to implement complex state-based network policies with a linear number of states and with fewer lines of code than in existing control languages (§4). Third, we implement support for a much broader range of network events than today's network SDN control programs (§5). Fourth, we have implemented PyResonance, a framework for implementing state-based network policies, in Pyretic (§6), and evaluated the expressiveness and complexity of PyResonance, as well as its performance overhead of event processing in realistic network settings (§7). Some operators (*e.g.*, Peter Phaal, the creator of sFlow [22], as well as others in the SDN Coursera course) have already used PyResonance to simplify existing network management tasks [14]. We have released the PyResonance source code and tutorials for writing control applications on GitHub [26] to allow others to do the same.

2 Overview

We describe the PyResonance architecture, introduce terminology used, describe PyResonance's basic features, and provide background on Pyretic.

2.1 Architecture and Terminology

Figure 1 shows the PyResonance architecture. Either an operator or the third party can write a PyResonance *controller*, which is invoked at the command line as a Pyretic program. A controller may itself have multiple PyResonance *tasks*, each of which implements a different state-based network policy (*e.g.*, authentication, intrusion detection). Every task has exactly one *finite state machine*. Each *state* in a task's FSM may correspond to a different *program*; PyResonance expresses these programs in the Pyretic language. The simplest programs are "drop" and "passthrough". A program can also be a composition of simpler programs (within a single task), or a composition of PyResonance tasks. Each PyResonance task outputs a program, and the PyResonance runtime composes them using Pyretic's composition operators and pass it to the Pyretic runtime. This final program dictates how traffic should be forwarded in the underlying network. An *event* is anything that can cause any task in the controller to change control programs. When an event arrives at a PyResonance controller, the corresponding task updates its view of the network state and the program that corresponds to that state. Eventually the controller dynamically re-evaluates the overall program. As with any "vanilla" OpenFlow controller, events may be standard OpenFlow events (*e.g.*, packet arrivals). A significant contribution of PyResonance, however, is that it allows tasks to apply different programs in response to a wider range of events.

Each task may process *events* from any network device that knows how to send an event message that PyResonance understands. Event stream *drivers* translate events from network devices into events that PyResonance tasks can process. Our release of PyResonance provides a driver that directly incorporates sFlow [22] event streams; we have also implemented a JSON event stream driver that allows an application to incorporate any appropriately formatted JSON event. A PyResonance controller can incorporate event streams from any network device either with a custom driver (*e.g.*, the sFlow driver) or through the more generic JSON event driver.

2.2 Features

We briefly describe the features of PyResonance. We will discuss these features in more detail in subsequent sections.

Incorporating generic event streams. In today's networks, operators must often manually change configurations in response to many network events, which may arrive from heterogeneous sources (*e.g.*, security alerts, system errors, network failures). PyResonance automates this process; it allows a network operator to specify *state-based network tasks*, whereby changes in network state may cause one or more controller tasks to execute different programs. PyResonance tasks can update their view of the network state based on either generic JSON-formatted event messages or custom *drivers* that process specific types of events (*e.g.*, from sFlow).

Simplifying state-based control programs. Any task might switch the program that it is executing based on events from load balance, authentication, or intrusion detection, as well as other types of events, such as the time of day. Network operators typically must configure the network to address each of these tasks; the configuration can then interact in non-deterministic and unexpected ways. PyResonance's support for state-based network tasks alone is not sufficient, since expressing all network states resulting from combinations of tasks quickly results in a combinatorial explosion of states (*e.g.*, "if the network link is overloaded, and host authentication succeeds, and it is the peak time of day, and ..."). PyResonance cleanly decomposes policy specifications to avoid state explosion. PyResonance allows an operator to specify independent tasks for various subtasks and *compose* them to produce a single control program.

Supporting federated configurations. Most networks do not have a single operations team, but are in fact managed by groups of operators with different concerns (*e.g.*, the security group and the traffic engineering group might each configure aspects of the network). When different sub-organizations configure different parts of the network, PyResonance must compose these programs. The biggest challenge for such networks is to enable each group to independently express policy while avoiding possible policy conflicts and overlaps. The ability to compose programs is again not enough; *how* to compose programs in a way that prevents conflicts is an important unanswered question. PyResonance offers various solutions to this problem.

2.3 Pyretic

Pyretic [20] is a Python-embedded domain-specific programming language for SDN. Pyretic encodes network data-plane behavior in terms of functions (which we call *programs*[1]) that map an incoming packet to an outgoing set of packets. Basic programs provided by Pyretic include: `drop` which takes a packet and returns the empty set; `fwd(p)` which takes a packet and returns the set containing just that packet, but now located at outport `p`; `match(f=v)` which takes a packet and returns the set containing just that packet if its header field `f` matches value `v` or the empty set otherwise. More complex programs such as `flood` might produce sets containing multiple packets. And it turns out this function-based abstraction of program allows Pyretic to define combinators on programs, such as parallel and sequential composition, that enable such complex programs to be quickly constructed from the basic programs described above.

Parallel composition takes multiple programs and produces a new program whose output is the combined output of the original ones—the output of running each of the programs simultaneously. Likewise, sequential composition takes multiple programs and produces a new program whose output is that of running each program on the output of the previous program.

Pyretic's runtime system efficiently compiles programs

[1]In the original Pyretic paper, these functions are called "policies" instead of "programs". We use the term "program" to distinguish Pyretic's functions from higher-level network policies, which are often encoded as written documents that express operator intent (*e.g.*, [7]).

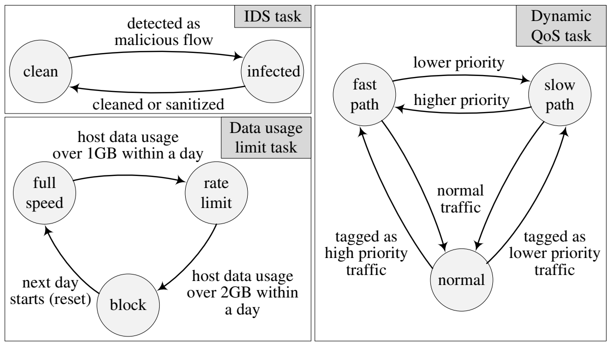


Figure 2: *FSMs for three different PyResonance tasks. Each state has a corresponding Pyretic program.*

to OpenFlow-based switches and provides support for incrementally updating these programs. PyResonance uses these facilities, providing infrastructure built on top of Pyretic for encoding FSM-based tasks and associating different tasks with external event sources. As events arrive at FSM-based tasks, these tasks will update their states, thereby triggering the Pyretic runtime to re-compile the updated program to network switches.

3 State-Based Network Tasks

To allow a controller to switch programs that it applies to traffic flows in response to changing network conditions, we introduce the notion of *state-based network tasks*. A *task* is a portion of a network-wide policy that is responsible for performing only one specific task. In this section, we describe how an operator defines these tasks; the subsequent section describes how tasks can be composed to implement more complex network behavior.

Controller applications and tasks. A PyResonance controller application comprises one or more *tasks*, which are composed using Pyretic composition operators. Each task represents some set of programs that might together achieve a particular network management task, such as load balancing, intrusion detection, or authentication. A task is thus: (1) a collection of (Pyretic) programs; (2) a finite state machine (FSM) that specifies which program should be executed for a particular state. As shown in Figure 2, each state has a (simple) associated Pyretic program; in practice, these Pyretic programs might be compositions of simpler Pyretic programs themselves. An operator thus must define a Pyretic program that corresponds to each state. Representing network tasks with FSMs and corresponding programs helps us address one of the fundamental challenges of network management (*i.e.*, continually changing network conditions).

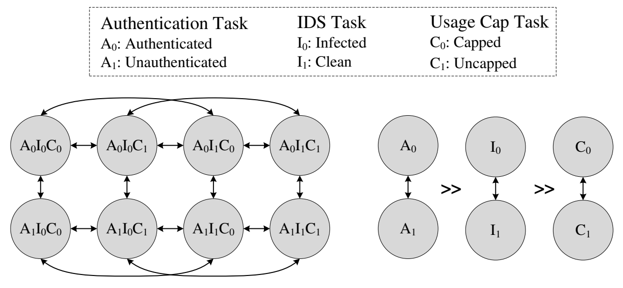
States and Pyretic programs. When a packet enters the network, PyResonance determines the state corresponding to that portion of flow space (*e.g.*, to the host that sent it). In this paper, packet is assumed to be represented by a 12-tuple *flow*, as defined in the OpenFlow specification, which provides more fine-grained control than conventional methods. For example, with VLANs, each VLAN is a separate policy group, but VLANs are assigned based on Ethernet MAC addresses. Thus, all traffic from a single host machine, whether HTTP or SSH, will be subject to the same network policy. Each task always maps a flow to exactly one state. A flow, can, of course, be represented in terms of multiple states, since each task (*e.g.*, authentication, IDS) has its own state machine, and a controller application that processes a traffic flow may be composed of many tasks.

```
1   class IDSTask(BaseTask):
2     ...
3     def infected(self):
4       return drop
5
6     def clean(self):
7       return passthrough
8
9     def action(self):
10      clean_flows=self.fsm.get_action('clean')
11      p1 = if_(clean_flows,
12               self.clean(),
13               self.infected())
14      return p1
```

Figure 3: *PyResonance code for implementing an IDS task in PyResonance. Lines 3–4 define the infected state and the corresponding Pyretic program (*`drop`*); lines 6–7 define a program for the clean state. Lines 9–13 define the main Pyretic program the IDS task ultimately returns.*

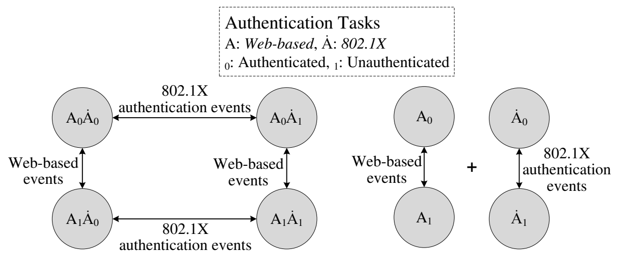
Events and transitions. Events that arrive at the PyResonance controller are dispatched to the appropriate task based on the type of event. The appropriate task then processes the event and executes a state transition and updates the corresponding program running for that task at the controller. One significant contribution of PyResonance is to provide support for processing a much broader set of events than today's OpenFlow controllers. Some examples of richer events include security events (*e.g.*, infections) and data usage events (*e.g.*, exceeding usage quotas). Section 5 describes event processing in detail.

Implementing state-based network tasks. Figure 3 shows a code sample from a simple state-based IDS task. Every application must implement the `action()` function which is called by PyResonance underlying runtime system when composing the applications together. The programmer needs to provide such a Pyretic program for each state. Packets belonging to flows in the "clean" state are allowed; those in the "infected" state are blocked. PyResonance provides a built-in function for getting a state's flow space (line 10), and matching it with incoming flows against a state (lines 11–13). Based on the outcome of this match, the task applies the appropriate Pyretic program.



(a) *Without composition* **(b)** *With composition*

Figure 4: *Composing independent tasks in sequence.*


Authentication Tasks
A: Web-based, Ȧ: 802.1X
0: Authenticated, 1: Unauthenticated
A0Ȧ0
A0Ȧ1
A1Ȧ0
A1Ȧ1
802.1X authentication events
Web-based events
Web-based events
802.1X authentication events
A0
A1
Ȧ0
Ȧ1
Web-based events
+
802.1X authentication events

(a) *Without composition* **(b)** *With composition*

Figure 5: *Composing multiple authentication tasks in parallel, so that any successful authentication would result in "passthrough" to subsequent states (as in Figure 4). As with sequential composition, the number of states is exponential in the number of tasks, but we show only two tasks to simplify our exposition.*

4 Composing State-Based Tasks

PyResonance makes it possible to express network policies in terms of states and events, but expressing a network policy in terms of all of the possible network states quickly leads to a combinatorial explosion of network states. Thus, PyResonance needs a way to decompose a complex representation of network policy into a simpler set of states and transitions. Pyretic provides *composition* operators that allow an operator to compose simpler network control programs to create more complex ones. Although Pyretic's composition operators were designed to allow the expression of policies in parallel or sequence, they can also allow a policy to be written in terms of simpler *tasks*, where each task has simpler state machines and corresponding programs. This decomposition makes it possible to express a network-wide policy by composing smaller programs which are easier to write and check for correctness. In this section, we explain how Pyretic's composition operators simplify state-based network policy and describe how some of these compositions can be implemented through topological associations.

4.1 Decomposition: State-Based Tasks

Network policies express how the forwarding should take place under many different conditions and for a large number of tasks, from traffic engineering to security. If a network operator had to precisely encode the network's desired behavior according for every possible network state, the policy would quickly become cumbersome to write and to validate.

Consider a simple control application that puts the host into a walled-garden until it has authenticated, rate limits a host if it has exceeded a usage cap, and quarantines the host if an infection has been detected. Each of these tasks has two possible states: authenticated or not, capped or not, and quarantined or not; depending on various events (*e.g.*, a message from an authentication module, intrusion detection system, etc.), a host may transition between any one of these two states for each of these tasks, thus resulting in a total of $2^3 = 8$ possible states (and eight corresponding programs for each of these states). In contrast, with PyResonance, an operator could express:

```
auth >> IDS >> cap
```

indicating that, first, the task should check whether the host is authenticated and apply the program corresponding to the appropriate state. The result of that program should be sequentially composed with the result of the intrusion detection task, and so forth. If any component of the network state changes, the program associated with the corresponding task also changes. The same network control program can be expressed in $2 \cdot 3 = 6$ programs and states. Figure 4 illustrates this comparison.

Thus, Pyretic's composition operators enable an operator to *independently* express the states and programs associated with different tasks. The resulting decomposition makes it easier to both express and verify the resulting network control programs, since an operator can independently express the tasks associated with each aspect of the network. In general, without composition, a network operator must define programs for $\Pi_{i=1}^{N} a_i$ possible states, where a_i represents the number of possible states for task i, where N is the total number of tasks. Pyretic's composition thus reduces state complexity for parallel task composition from exponential to linear. This savings is significant, even for applications with relatively few tasks and states. If a control program has 10 tasks, each with two possible states, a monolithic state-based control program would require nearly 1,000 states, as opposed to just 20 states with PyResonance. (If the order in which tasks are sequentially composed doesn't matter, the state explosion becomes even worse, requiring $N! \cdot \Pi_{i=1}^{N} a_i$ programs.)

As shown in Figure 5, parallel composition also reduces state complexity. For example, a program might specify that either a network flow should be authenticated by a web authentication module or by an 802.1X authentication module. If *either* of these tasks places the host in an authenticated state, the host should be allowed to send traffic, and the resulting program should be composed with subsequent tasks (*e.g.*, the IDS task and the data cap task,

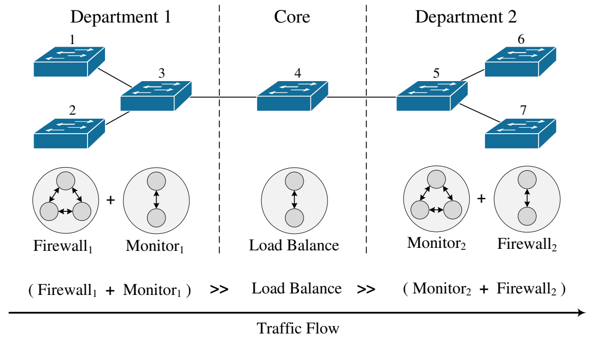


Figure 6: Federated configuration. *Because traffic travels through different parts of the network topology in sequence, sequential composition of federated policies can be applied on different switches as traffic traverses different network elements.*

as above). Without Pyretic's composition operators, the network state machine would need a second set of states to express an equivalent network control application. Specifically, if the global network state were represented as a bit, the network state machine would need to express the state tuple *(802.1X state, web state, ...)*, where `00*` (neither module authenticated the host) could represent one state and `01*`, `10*`, and `11*` (either module authenticated the host) could represent the other state. The total number of states required without decomposition would thus be 2^n, where n is the number of authentication tasks. The number of states explodes further if any module composed in parallel could assume more than two possible states. As before, a network operator must define programs for $\Pi_{i=1}^{N} a_i$ possible states, where a_i represents the number of possible states for task i, where N is the total number of tasks.

Implementing state-based policy composition. It is possible to combine and compose multiple PyResonance tasks using Pyretic's composition operators. Below is the configuration for PyResonance to compose authentication, DoS, and monitor tasks with sequential and parallel composition.

```
MODULES = {
  pyretic.pyresonance.tasks.auth,
  pyretic.pyresonance.tasks.ids,
  pyretic.pyresonance.tasks.monitor,
}

COMPOSITION = {
  auth >> (ids + monitor)
}
```

The results from whatever program the application task is running will be applied to the parallel combination of programs that the DoS and monitor tasks are running. With parallel composition, packets that are blocked by the DoS task will still pass through the monitor task.

4.2 Federated Configuration

Most network configurations are maintained by multiple sub-organizations within an organization. For example, in a backbone network, the traffic engineering team may manage traffic load balance, and the security team might manage access control and response to various security events. Similarly, in a campus or enterprise network, different departments might independently configure parts of the network. For example, the computer science department might manage and configure the switches for its part of the network independently of the switches and routers in the network core.

Today's networks have no formal notion of "composition", although it effectively happens in an *ad hoc* manner, since multiple network operators are effectively configuring different parts of the same configuration files (*e.g.*, access control and load balance) or different switches and network devices in the same network. The *ad hoc* nature of this composition can make it difficult to reason about the behavior and correctness of the configuration. Pyretic's composition operators allow each sub-organization to write independent PyResonance tasks; the resulting network-wide policy is a composition of tasks from different sub-organizations.

In a distributed network, certain aspects of composition can be implemented by exploiting the network topology and higher-level topological associations. For example, a network might indicate that outgoing traffic from a department should first be subject to authentication (*e.g.*, only permit the traffic flow if the host is authenticated) followed by load balance (*e.g.*, balance flows evenly across the links in the topology). Such a PyResonance program naturally decomposes across switches in the topology, since the authentication task can be applied as traffic leaves an edge network in the department and the load balance task can be applied as traffic traverses switches in the network core. The PyResonance configuration maintains an association between network devices and topological components (*e.g.*, specifying which switches and devices are associated with a particular sub-organization or portion of the network), thus making it easier to decompose network-wide policy across the devices in the network.

Implementing federated configuration. Suppose that *Department 1* and *Department 2* want their own monitoring and firewall modules applied to switches within their sub-domains. In the *core* network, which connects *Department 1* and *Department 2*, an IDS module is deployed to enforce a uniform security policy to communication between subnetworks and the Internet. Figure 6 shows the list of switches under *Department 1* and *Department 2*. As traffic moves from *Department 1* to *Department 2* via the *core* network, the programs corresponding to each task are applied to the network traffic in sequence.

To implement federated configuration, PyResonance extends the Pyretic `match` predicate, which matches all switches corresponding to particular department and compose policies for all the departments in parallel. The policies for the individual departments and network might be defined as follows:

```
DP1 = match(switch={D1}) >> (Firewall_1 +
     Monitor_1)
Core = match(switch={Core}) >> LB
DP2 = match(switch={D2}) >> (Firewall_2 +
     Monitor_2)

P = DP1 + Core + DP2
```

where `P` is the overall policy that says "match on the switches in department `D1` and apply the firewall and monitoring policies in parallel", and so forth. (Separately, `D1`, `Core`, and `D2` are defined as the set of switches corresponding to each department.) The resulting Pyretic program actually corresponds to the topological decomposition in Figure 6. In the (equivalent) policy shown in the figure, however, the parallel composition across departments becomes a sequential composition, because the program itself becomes physically distributed, so there is no longer a need to apply Pyretic's abstraction of matching on located packets. Of course, programs can apply to finer-grained subsets of traffic flows (as in Pyretic).

Support for hierarchy. Thus far, we have discussed only scenarios where a sub-organization defines tasks and programs for a single department only. Yet, in an actual enterprise network, a higher-level organization might define a network policy that it wants to apply to the entire campus (*e.g.*, the IT department might want to rate-limit all Bittorrent traffic). Let us call that global rate-limiting policy `R`. In this case, the IT department could express:

```
P_R = (match(switch={D1}) >> R) +
     (match(switch={D2}) >> R)
```

where `P` is the overall policy that says "match on the switches in department `D1` and apply policy `R`" and "match on the switches in department `D2` and apply policy `R`". The overall policy would then be:

```
P = P_R >> (dp1 + core + dp2)
```

5 Network Events

We describe event drivers, describe an example of a native event driver and a JSON-based event driver, and explain how PyResonance can incorporate event sources.

5.1 Event Drivers

To process the different types of events that devices in a network might generate, PyResonance provides a driver interface for each event type. PyResonance tasks register with one or more event drivers and update their states (and corresponding programs) in response to incoming events that may be processed by those drivers.

```
1  keys = 'ipsource'
2  value = 'frames'
3  rate = 1000 # packets per second
4  metric = 'ddos'
5  flows = {'keys':keys,'value':value}
6  threshold = {'metric':metric,'value':rate}
7
8  message_type = 'state'
9  message_value = 'denied'
10 message = {'event_type':metric, 'message_type':
         message_type, 'message_value':
        message_value}
```

Figure 7: *PyResonance code for defining and implementing sFlow rule for DoS detection task. Line 1–4 define the sFlow parameters. Line 5–6 specify the formatting. Line 8–10 specify the format and the action to perform based on the sFlow event.*

A PyResonance task can register for events by importing the relevant drivers. PyResonance provides two implementations: (1) a native sFlow driver; and (2) a generic driver for processing JSON events. Each driver takes a handler function as input. Usually, it is the "message_handler" function of a PyResonance task that is responsible for doing the state transitions whenever a new event is received. To incorporate events generated by a certain network device, an operator can either write a custom native driver (as we did for sFlow) or use the generic JSON driver and adapt the network device to sent appropriately formatted messages. Native drivers may be more useful when more accurate event handling and reaction time is desired. Yet, building native drivers is more involved and may be unnecessary, so we have provided the JSON driver to save programmers this trouble in some cases.

5.2 Example Drivers

We describe the implementation of an example native driver for sFlow, as well as a generic JSON driver.

5.2.1 Native Driver: sFlow

sFlow [22] is a widely used sampling technology used to monitor network traffic for collecting, sorting, and analyzing traffic data. It is low cost, scalable, and provides the ability to monitor thousands of network interfaces from a single vantage point (*i.e.*, controller). Providing a native driver for sFlow in PyResonance enables application developers to write policies based on different network events (*e.g.*, data rate, packet loss and link events).

Figure 7 shows the sFlow rule (line 1–6) and the corresponding state transition (line 8–10), which the programmer might specify when writing a DoS application. The sFlow rule specifies that an event should be generated whenever a flow belonging to a particular `ipsource` exceeds the threshold limit of 1,000 packets per second. Based on that event, the DoS detection task will update and switch to executing a program that denies traffic belonging to one or more set of source IP addresses for which the threshold value has been exceeded.

```
"event":
{
  "event_type":  # Type of event source
                 # (e.g., authentication, IDS
                     )
  "sender":
  {
    "id":,          # Unique ID number
    "description":, # Optional description
    ...
  }
  "message":
  {
    "type" :,  # query, data, state, module
    "flow":,   # 12-tuple flow specification
    "value" :  # Message value
  },
...
}
```

Figure 8: *JSON event. Each event has an event type. The sender encapsulates the message, which contains the message type (e.g., querying, providing data value or state value, module on/off function, etc.), flow specification, and message.*

A native sFlow driver makes it easier to write control programs based on a richer set of traffic monitoring events. An operator could also write native drivers for other protocols (*e.g.*, Netflow, IPFIX).

5.2.2 Generic Driver: JSON

Figure 8 shows a JSON event message that PyResonance's generic event driver can understand and parse. We include a Python-based script named `json_sender.py` that can generate and send JSON-formatted event messages. We use this script to demonstrate examples throughout this section. There are two ways to send JSON events:

Sending the next state value. In this mode, the event message includes the *next-state* value for a given flow. This event will cause the state of the flow to change to a particular *next-state* value, causing PyResonance to update and apply a new program to the flow that corresponds to the new state. In this case, it is the event source's responsibility to determine the next state for a given flow, and the PyResonance controller merely extracts this state value and updates the mapping between the flow and state value, which in turn will influence the state policy applied to that specific flow. Note that it is necessary to explicitly specify the flow. A flow can be specified using as many as 12 fields in the OpenFlow specification (where unspecified fields are interpreted as wildcards). Below is an example of this mode using our Python-based event sending script.

```
$ python json_sender.py --flow='{scrip=10.0.0.1}'
    -e auth -s authenicated
```

Sending the general data value. In this mode, the event message contains an arbitrary value that may represent an event. This case is likely more realistic for network devices, which would not necessarily know the FSM structure corresponding to a task. In this case, the event source is not responsible for determining the next state value for the given flow. Instead, the PyResonance controller receives and parses a generic data value, and determines what to do with the given information. The following shows an example of this mode using our Python-based event sending script.

```
$ python json_sender.py --flow='{scrip=10.0.0.1}'
    -e auth -i ./data_info
```

The above script opens and reads a given file, `data_info`, and sends its content to the controller, which is also JSON-formatted. This facility provides flexibility for specifying arbitrary events. In the next section, we explain how PyResonance incorporates these arbitrary events.

5.3 Incorporating Arbitrary Event Sources

The JSON event driver understands and parses JSON-formatted event messages, which makes it easy to build a variety of custom event sources. Operators can add small scripts to existing middleboxes or network devices (*e.g.*, intrusion detection boxes, monitoring systems, or authentication portals) that send JSON messages, which in turn induce a task to change its state and corresponding program (*e.g.*, block malicious traffic, allow authenticated host's traffic). It is also possible to implement custom small-scale scripts that generate events periodically. For example, `time-of-day.py` in our public repository is a small Python code that generates JSON events at 7 am (morning) and 7 pm (night) everyday, and when weekend starts (Friday 11:55 pm) and ends (Sunday 11:55 pm).

PyResonance's JSON driver makes it easy to build higher-level abstractions to incorporate PyResonance into Web development frameworks like Google App Engine [15] or Django [10]. For example, management systems could provide intuitive interfaces (*e.g.*, a Web GUI) that generate and send event streams based on user's click in a Web browser.

6 PyResonance Programming

We describe the process of writing, running, and debugging a PyResonance controller.

6.1 Writing a PyResonance Controller

The PyResonance GitHub wiki contains a detailed, step-by-step tutorial on how to write a PyResonance task and run the PyResonance controller [25]. We provide a brief summary of the process.

1. *Express state-based network policy in terms of tasks.* First, an operator must express dynamic network policies in terms of one or more tasks, where each task has one FSM.
2. *Implement and return a Pyretic program for each state in every task's FSM.* In each task's FSM, every state must return a Pyretic program (*e.g.*, drop, passthrough, some composition of simpler Pyretic programs, *etc.*).

3. *Define a unique event type and handler.* Define a unique event type for each task, and implement an event handler for each task that can process events.
4. *Implement the* `action()` *method.* The `action()` method applies the correct program on incoming flows based on the current state.
5. *Specify how tasks (and, hence, corresponding programs) should be composed.* Apply Pyretic's composition operators to compose one or more tasks into a more complex network program that represents the overall network-wide policy.
6. *Store each task's file, and write a configuration file.* Write a configuration file that lists the task names. Optionally, specify the composition of tasks.
7. *Run PyResonance.* Invoke Pyretic with the PyResonance control program as an argument, along with PyResonance's specific arguments.

6.2 Debugging Control Programs

PyResonance also has debugging features.

Querying state information for flows. PyResonance provides a separate handler that answers queries about the state information in all FSMs loaded in a control application for a particular flow. The programmer can use the JSON proxy driver along with the *json_sender.py* to send such queries. For example, running the command below will return the set of states PyResonance currently has for each application's FSM for a flow from 10.0.0.1 with source port 80.

```
$ python json_sender.py --flow='{scrip=10.0.0.1,
    srcport=80}' -q all
```

This information is useful when operators need to know the current states that tasks associate with a particular flow, which can help determine the resulting programs that each task is currently running (and, hence, the behavior of the overall control program).

Enabling and disabling tasks. It is possible to dynamically turn on or off task by sending event. Disabling a task results in the module reverting to a `passthrough` program in the case of sequential composition or a `drop` program if the tasks composed in parallel with other tasks. This feature can be useful when an operator wants to manually disable a task (*e.g.*, for maintenance or debugging).

7 Evaluation

Evaluating PyResonance is a challenge in and of itself, as its most significant benefits are simpler network management. To guide our evaluation of PyResonance, we again draw from the 4D work [16]: "*The proposed configuration language should be evaluated along two dimensions: complexity and expressiveness. It should have a lower complexity than that of configuring individual routers today. In addition, it should be able to express the network-level objectives that arise in existing networks.*" Accordingly, we evaluate PyResonance in terms of both expressiveness (Section 7.1) and complexity (Section 7.2) using a combination of qualitative and quantitative metrics. We also evaluate the effect of event processing on the PyResonance controller performance (Section 7.3).

7.1 Expressiveness

We present several realistic network policies implemented with PyResonance. The source code for these examples is available on GitHub [26].

Authentication system. An authentication system is essential in many networks. The system we implemented checks the state of the host corresponding to the flow for an incoming packet. If the host is not in an *authenticated* state, it redirects the traffic to the authentication portal so that the user can authenticate using username and password combination. Upon authentication, the portal generates an event to PyResonance to change the state of user's device to *authenticated* state, so that host's traffic is allowed to traverse the network and access the Internet. We compose the resulting task with an intrusion detection task, so that authenticated hosts are only allowed network access if they are determined to be *clean*; if an IDS event indicates that a host is *infected*, its network access is blocked.

DoS detection with automatic quarantine. The DoS task allows traffic to pass through by default, but blocks traffic that is generated from a host that is suspected to be a DoS origin. To detect DoS activity, we simply use the sFlow package, which is included in OpenVSwitch, to monitor and detect abnormal traffic increases from a host. The sFlow server daemon detects such abnormal behavior and raises an event, which is sent to the DoS task in PyResonance via the native sFlow driver. The DoS task in turn updates its state to run a Pyretic program that quarantines the host. If the host is considered to be clean again, either a manual update (via JSON) or an update via the sFlow driver can shift the host back to a normal state.

Bandwidth limit based on data usage. We implement the *data usage limit task* shown in Figure 2. sFlow tracks data usage; when the host downloads data over 1 GB within a day, PyResonance shifts the host's traffic to use a link that adds additional delay, which rate-limits the traffic. If the data usage goes over 2 GB within a day, the module starts to block traffic from that host. This application module is inspired by a real network policy concerning data usage at Carnegie Mellon University [7]; many enterprise and campus networks have similar policies.

Inbound server load balancing. The server load balancing task splits traffic that is destined to a public IP and redirects to two different servers. For example, if traffic is destined to a certain IP (10.0.0.100), the policy rewrites the destination IP and Ethernet address so that it is redirected

#Lines	File	Task
36	auth.py	Checks if a flow is in authenticated state, passthrough action if authenticated else redirection to authentication server
22	ids.py	Checks if a flow is in quarantine/clean state, passthrough action if clean else drop
60	ratelimit.py	Monitors data usage for a user and applies the data cap actions if required
53	server_lb.py	Splits traffic between set of servers to balance the server load

Table 1: *The lines of code for different PyResonance tasks.*

to either 10.0.0.3 or 10.0.0.4 based on the state associated with the incoming flow. In addition to the task itself, we implemented an automated event source that raises and sends JSON events to the PyResonance controller based on the time-of-day value to allow for different load-balancing behavior during peak hours.

7.2 Complexity

Complexity is subjective; we approximate reductions in complexity with both reduction in lines of code and qualitative programmer experiences.

Reduction in lines of code (LoC). PyResonance use of Pyretic's APIs and libraries reduces the lines of code (LoC) required to write a controller application. PyResonance's own APIs and generic design further reduce the lines of code. PyResonance, which includes its runtime, baseFSM, eventProcessor and all the drivers has 314 lines of Pyretic and Python code. Table 1 shows the LoC for various PyResonance tasks.

Qualitative programmer experiences. We introduced PyResonance to students through an assignment in a Coursera SDN course [8] after first teaching them Pox and Pyretic. Students had to complete an assignment using each of the three frameworks (*e.g.*, Pox, Pyretic, and PyResonance) [9]. Overall, 80% of the students who completed the Pyretic assignment *also* completed the PyResonance assignment, suggesting that writing a PyResonance program given that the programmer already has knowledge of Pyretic is relatively straightforward. In fact, since students who successfully completed the Pyretic assignment and all previous assignments with high scores did not have to complete the PyResonance assignment, the success rate is likely even higher than 80%. Another useful statistic is the completion rate, among students who attempted the assignment: 96.5% students who attempted the PyResonance assignment successfully completed it, which is comparable to the completion rate for the Pyretic assignment (97.4%).

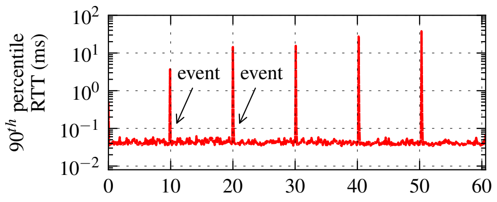
Inspired by PyResonance's capabilities from taking the course, Peter Phaal, the creator of sFlow, implemented a DDoS application using PyResonance and noted that it made his original script simpler because it no longer needs logic to convert measurements into OpenFlow flow table modifications. His script now simply needs to send appropriate high-level event message to the PyResonance controller [14].

7.3 Performance

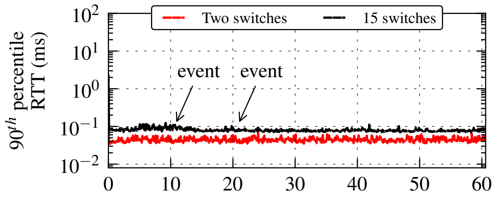
We evaluate PyResonance performance in terms of the effect of events on forwarding rate and processing time at the controller. To evaluate packet-in processing performance with respect to Pyretic, we used Cbench [3], an OpenFlow controller benchmarking tool. Packet-in processing performance is directly tied to data-plane forwarding performance, as every first packet has to come to the controller in OpenFlow. PyResonance introduces negligible overhead on packet-in processing performance in addition to Pyretic when it is not processing dynamic events. We do not include the details of this experiment due to space constraints; instead, we focus on how event processing at the controller affects forwarding performance.

Effect of events on forwarding performance. Figure 9 demonstrates how processing dynamic events affects data-plane forwarding performance. We measure the change in round trip time (RTT) as a result of event arrivals at different rates. We emulate a testbed with two switches and two hosts, where each switch has a single host attached. Each host sends 10 ICMP ping requests per second to the other host. We then send an event to the PyResonance controller every 10 seconds. Figure 9a shows RTT impulses whenever PyResonance receives an event; these spikes occur as the PyResonance controller processes events and expunges the relevant flow table entries to ensure that incoming traffic is subject to the program corresponding to the new state. This increase in RTT results from flow setup time (which is a function of the underlying Pyretic runtime). However, flows whose forwarding behavior is not changed by the incoming event (*i.e.*, those flows that are still subject to the same PyResonance program) do not experience any increase in latency, because the flow table entries can remain cached. Figure 9b shows the RTT value for such flows with events incoming every 10 seconds; the 90th percentile RTT does remain roughly the same, even when an event arrives. The base RTT value slightly increases with more switches attached to the PyResonance controller.

Events arriving at PyResonance controller increase the latency of flows whose forwarding behavior must be updated. This increase only applies to the first packet of the flow. Therefore, the forwarding performance depends on the *ratio of traffic amount to number of events*. We now study the effects of event arrival rates on forwarding performance. Naturally, if the amount of data traffic traversing the data plane is significant and the number of events happening is negligible, the forwarding performance would not



(a) *Flows that are affected by program update.*



(b) *Flows that are not affected by program update.*

Figure 9: *Effect of program updates in response to events on network latency.*

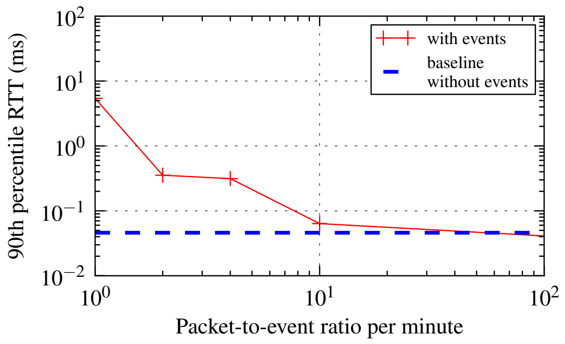


Figure 10: *Packet vs. event ratio and median RTT.*

suffer. To explore forwarding performance overhead for a range of event arrival rates, we vary the *packet-to-event ratio*, where packet-to-event ratio is the ratio of number of pings per minute to the number of events per minute. For example, if there are 60 pings per minute and 60 events per minute, the packet-to-event ratio is 1. Below one packet-to-event ratio means there are more events happening than the actual data traffic, while over one packet-to-event ratio means there are more ICMP traffic then number of events per fixed interval. As Figure 10 shows, the median RTT decreases as the event arrival rate decreases.

Event processing time. Next, we measure how much time a PyResonance controller spends processing incoming events. Event processing time is measured by subtracting the timestamp value when an event was received from the timestamp value when PyResonance actually sends a policy update to the runtime. We repeated the measurement 10 times to account for any abnormalities from a single experiment run, with varying number of PyResonance tasks

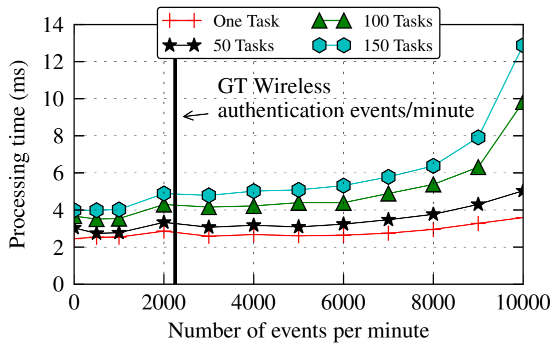


Figure 11: *Event processing time vs. event rate.*

loaded and sequentially composed in PyResonance. Figure 11 shows the average processing time with varying number of events per minute. The event processing time increases to 2.8 milliseconds at around 2,000 events per minute and gradually increases until around 7,000 events per minute. Still, the actual processing time remains small. The black vertical line shows how many wireless authentication events occur in the Georgia Tech campus network, which is one of the most frequent network events that occur in a large campus network. The network experiences about 1.7 million authentication events in its wireless network in a typical day, or an average of 2,360 events per minute, if we assume most authentication events happen in 12 hours per day (*e.g.*, 8 a.m.–8 p.m.). Figure 11 shows that PyResonance can handle this event arrival rate.

8 Related Work

Network management. PyResonance simplifies the expression of network policies. Previous works have proposed various management methods to achieve a similar goal; most previous systems focus on building an abstraction layer on top of vendor-specific low-level commands; most of these approaches pre-date software defined networking. Ballani *et al.* built CONMan, which uses higher-level modular building blocks to achieve same functions [1]. PACMAN [6] and COOLAID [5] both implement a higher-level construct than device-specified configuration languages to automate certain networking tasks. Puppet [24] and Chef [4] can help automate network device configuration tasks; certain vendors even have device-specific plugins to help automate router configuration. Still, these tools typically operate on low-level, device-specific commands, involve disjoint configuration of individual network devices, and provide no correctness guarantees.

SDN, OpenFlow, and controller platforms. Software defined networking (SDN) has roots in RCP [11], 4D [16], and Ethane [2]. The 4D architecture describes the motivation for separating the control and data planes. Ethane introduced a method for managing a network with a software program that populates flow-table entries in network switches. Controllers such as NOX [17], POX [23], Floodlight [12], and OpenDaylight [21] make it possible to write controller applications but do not specify *how* to design and build them.

Languages for SDN. Recent work has developed languages for software defined networks. FML [18] allows network operators to write and maintain policies efficiently in a declarative manner. Nettle [27,28] is a domain specific language implemented in Haskell that is used to configure BGP policies with more comprehensive abstraction calculation constructs. Procera [29] is a more recent work that attempts to express reactive policies with functional reactive programming. Frenetic [13] is a family of domain specific languages (DSLs) for specifying and composing different network policies. Languages in this family, such as Pyretic [20] a Python-embedded language with support for constructing policies that can be automatically re-evaluated on change, share a common set of fundamental constructs such as basic policies and combinators, as well as a shared toolbox of techniques for efficiently compiling these to OpenFlow switches.

Procera and Pyretic provide some support for encoding programs that change in response to events, yet neither provides PyResonance's functionality. Procera re-evaluates programs when input values change, but it does not provide mechanisms for incorporating states or events. Pyretic provides syntactic constructs for designating dynamic programs and runtime support for updating them, but lacks a framework for structuring *how* these policies should evolve. PyResonance fills this gap by providing a principled framework on top of Pyretic for encoding dynamics.

9 Future Work

PyResonance opens up several avenues for enhancements and follow-on work, both to the network management domain and to the underlying Pyretic language.

New Pyretic operators: Subsumption and Distribution. Pyretic is a new, evolving language, and PyResonance is perhaps the first complete system that has been implemented in Pyretic. Our use of Pyretic to implement common network management tasks has highlighted a few missing features in the language. We have felt the lack of a *subsumption* operation, which would allow one program to subsume another "base" program under certain circumstances. Such an operation could allow network operators to specify some base level of operations (*e.g.*, routing), which might then be enhanced or subsumed with additional operations (*e.g.*, access control, load balance). PyResonance's application to federated configuration also presents the need for logic that supports placing rules for a particular program on the appropriate network switches, depending on the network topology and the flow of traffic. In Section 4.2, we illustrated that by distributing sequential composition in a natural way across switches, the composition operation effectively happens "for free" as traffic traverses the network. This phenomenon may be generalizable, and we are investigating algorithms for optimizing rule placement across topologies for future work.

Coupling tasks and finite state machines. Currently, PyResonance achieves FSM decomposition by exploiting either the independence of tasks (*e.g.*, security tasks being written independently of resource management tasks) or the common dependence on input. In the future, PyResonance could support state-based tasks whereby the state (and corresponding program) of one task depends on the states of others (effectively allowing direct communication between tasks). For example, a planned maintenance task might enter a state that "drains" a portion of the network; the resulting "maintenance" state might be coupled to multiple other tasks, such as load balancing, which could automatically determine that certain subsets of application traffic should be prioritized during such a state. Similar functions can be achieved by sending the same event to different tasks, but coupling tasks may ultimately simplify certain operations. PyResonance already has most of the plumbing to make this type of coupling possible.

Automated verification. Composition allows a network operator to express a complex network-wide policy in terms of simpler tasks. In addition to making it possible to specify tasks and programs that are easier to manually inspect, PyResonance could ultimately make automated verification more feasible, since finite state machines have structure that can be applied to verification methods, such as a statement written in temporal logic.

10 Conclusion

We have introduced *state-based network policies*, which help a network operator describe how a network's forwarding behavior should change in response to arbitrary network events. Such a policy is implemented in a control program that comprises one or more state-based network tasks, each of which encodes the forwarding behavior for a single network management task (*e.g.*, intrusion detection) or part of the network (*e.g.*, a sub-organization). Composing these tasks (and their corresponding programs) produces a network-wide control program that adapts to different operating conditions. We used Pyretic to implement state-based network policies in a system called PyResonance, and we showed that PyResonance is expressive enough to specify a wide range of network policies, simple enough for many operators to use, and efficient enough to process events in real-time for operational networks. Some network operators have already used PyResonance to simplify common network management tasks, and we have released the code to encourage others to do the same.

References

[1] H. Ballani and P. Francis. Conman: a step towards network manageability. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 205–216, New York, NY, USA, 2007. ACM. (Cited on page 11.)

[2] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking packet forwarding hardware. In *Proc. Seventh ACM SIGCOMM HotNets Workshop*, Nov. 2008. (Cited on page 11.)

[3] Cbench. `http://www.sdncentral.com/projects/cbench`. (Cited on page 10.)

[4] Chef. `http://www.opscode.com/chef/`. (Cited on pages 1 and 11.)

[5] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. Declarative configuration management for complex and dynamic networks. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 6:1–6:12, New York, NY, USA, 2010. ACM. (Cited on page 11.)

[6] X. Chen, Z. M. Mao, and J. Van der Merwe. Pacman: a platform for automated and controlled network operations and configuration management. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, CoNEXT '09, pages 277–288, New York, NY, USA, 2009. ACM. (Cited on page 11.)

[7] CMU network bandwidth usage guildline. `http://www.cmu.edu/computing/network/connect/guidelines/bandwidth.html`. (Cited on pages 3 and 9.)

[8] Coursera SDN MOOC, May 2013. `http://coursera.org/course/sdn/`. (Cited on pages 2 and 10.)

[9] Coursera SDN MOOC Programming Assignments, May 2013. `http://goo.gl/sAsqGV`. (Cited on pages 2 and 10.)

[10] Django. `https://www.djangoproject.com/`. (Cited on page 8.)

[11] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and K. van der Merwe. The case for separating routing from routers. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture*, Portland, OR, Sept. 2004. (Cited on page 11.)

[12] Floodlight OpenFlow Controller. `http://floodlight.openflowhub.org/`. (Cited on page 12.)

[13] 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*, 51(2):128–134, Feb. 2013. (Cited on page 12.)

[14] Frenetic, Pyretic and Resonance. `http://blog.sflow.com/2013/08/frenetic-pyretic-and-resonance.html`. (Cited on pages 2 and 10.)

[15] Google AppEngine. `https://developers.google.com/appengine/`. (Cited on page 8.)

[16] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *ACM Computer Communications Review*, 35(5):41–54, 2005. (Cited on pages 1, 9 and 11.)

[17] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, July 2008. (Cited on page 12.)

[18] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN '09, pages 1–10, New York, NY, USA, 2009. ACM. (Cited on page 12.)

[19] H. Kim, T. Benson, A. Akella, and N. Feamster. The evolution of network configuration: a tale of two campuses. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 499–514, New York, NY, USA, 2011. ACM. (Cited on page 1.)

[20] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *USENIX NSDI*, 2013. (Cited on pages 2, 3 and 12.)

[21] OpenDaylight. `http://www.opendaylight.org/`. (Cited on page 12.)

[22] P. Phaal, S. Panchen, and N. McKee. *InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. Internet Engineering Task Force, Nov. 2001. RFC 3176. (Cited on pages 2, 3 and 7.)

[23] POX OpenFlow controller. `http://www.noxrepo.org/pox/about-pox/`. (Cited on page 12.)

[24] Puppet. `http://puppetlabs.com/solutions/juniper-networks`. (Cited on pages 1 and 11.)

[25] PyResonance Wiki: How to write a PyResonance application. `https://github.com/Resonance-SDN/pyresonance/wiki/How-to-write-a-PyResonance-Task:-Step-by-step-tutorial`. (Cited on page 8.)

[26] PyResonance Github repository. `https://github.com/Resonance-SDN/pyresonance/`. (Cited on pages 2 and 9.)

[27] R. Rocha and J. Launchbury, editors. *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*, volume 6539 of *Lecture Notes in Computer Science*. Springer, 2011. (Cited on pages 12 and 13.)

[28] A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In Rocha and Launchbury [27], pages 235–249. (Cited on page 12.)

[29] A. Voellmy, H. Kim, and N. Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM. (Cited on page 12.)