Lithium: Event-Driven Network Control

Hyojoon Kim†, Andreas Voellmy‡, Sam Burnett†
Nick Feamster†, Russ Clark†
†Georgia Institute of Technology ‡Yale University

ABSTRACT

This paper introduces *event-driven network control*, a network control framework that makes networks easier to manage by automating many tasks that must currently be performed by manually modifying low-level, distributed, and complex device configuration. We identify four policy domains that inherently capture many events: time, user, history, and traffic flow. We then present Lithium, an event-driven network control framework that can implement policies expressed using these domains. Lithium can support policies that automatically react to a wide range of events, from fluctuations in traffic volumes to changes in the time of day. Lithium allows network operators to specify network-wide policies in terms of a high-level, event-driven policy model, as opposed to configuring individual network devices with low-level commands. To show that Lithium is practical, general, and applicable in different types of network scenarios, we have deployed Lithium in both a campus network and a home network and used it to implement more flexible and dynamic network policies. We also perform evaluations to show that Lithium introduces negligible overhead beyond a conventional OpenFlow-based control framework.

1. Introduction

Network management is incredibly difficult and remains one of the most important unsolved problems in communications networks; this problem is becoming increasingly acute as networks become bigger and more complex. Network management potentially entails implementing a variety of policies and actions, ranging from provisioning network topologies to implementing traffic load balance and access control. Unfortunately, despite the increasing variety and complexity of network management tasks, the mechanisms for configuring and managing networks remain frustratingly primitive: to perform even simple tasks, operators must grapple with low-level configuration of individual devices, as well as complicated network-wide dependencies. Manual configuration is complex and error-prone [3, 14, 23, 24, 26].

Network operators cope with the continual transformation of network states caused by various events ranging from intrusions to excessive data usage to network congestion by either manually reconfiguring individual network devices or relying on unwieldy collections of scripts. Different modules, set of devices and scripts are responsible of solving different dynamics of the network, making it harder to maintain the network in a manageable state. The problem is exacerbated by the fact that current configuration languages are low-level do not accommodate frequent changes.

We posit that the complexity of network management stems from the following two shortcomings of the current configuration model:

- *Network conditions are dynamic.* The state of the network is continually changing due to a variety of network events. Hence, network configuration is continually in flux [22].
- *Configuration languages are low-level and distributed.* The languages that network operators use for configuring network devices are rudimentary. A network device's configuration can have hundreds to thousands of device-specific configuration lines that poorly represent the intended policy or high-level intended behavior [4, 8, 12].

One approach to coping with the difficulty of network configuration is to design tools that automate and check existing network configurations [14, 16, 25]. Despite the significant amount of previous work in this area, however, testing network configurations remains extremely difficult. Another approach is to devise a higher-level language that could make network management and configuration significantly easier and less prone to errors. Although recent developments in network programming languages based on functional reactive programming show promise [17, 21], these languages still typically operate on a packet level, rather than on the level of higher-level policies. Raising the level of abstraction requires developing a control model that can incorporate and process more sophisticated, high-level events.

This paper describes the design, implementation, and deployment of Lithium, a new *event-driven network control runtime* that supports more realistic network policies. Unlike existing paradigms for network configuration, Lithium naturally handles changes to network conditions that arise due to temporal conditions and changes in network state. The model supports reactive constructs, which may ultimately enable a wider classes of applications to be programmed in a high-level declarative language. Lithium allows packet processing

rules to change over time and in reaction to internal or external network events.

Network operators can use Lithium to enforce policies that are written as event-driven programs, using the state machine model and event listener built in the Lithium controller. Through these components, it is possible to specify a complex network policy in a simple, high-level declarative manner, which normally requires intensive planning and use of variety of scripts to achieve the same goal if current configuration methods based on low-level language is used, as we later demonstrated in Section 5.

Software-defined networking separates the "data plane" (*i.e.*, the devices that forward packets) from the "control plane" (*i.e.*, the logic that makes decisions about how traffic should be forwarded). Lithium draws inspiration from a large body of previous work on software defined networking, including Ethane [6], RCP [15], 4D [19], and proposals in the IETF FORCES working group [11]. This decoupling makes it possible for a network's forwarding behavior to be dictated by a single, logically centralized software control program, rather than complex, low-level network configurations. However, the central question in software defined networking—how should network control be specified at the controller—remains as an unanswered question of utmost importance. For example, OpenFlow, which is one of protocols built upon this paradigm, provides the environment to control forwarding of traffic based on the incoming flow. However, OpenFlow does not specify how software defined networking could simplify network management, nor does it incorporate control mechanisms for processing dynamic network events. Lithium is a software defined networking framework that supports a richer set of policies than conventional OpenFlow-based control, which only operates on flow characteristics.

We deployed Lithium in two unique settings to demonstrate the power and flexibility of its constructs. First, we used Lithium to re-implement a network access control framework (which is currently implemented with a complicated VLAN-based configuration) across three buildings on the campus; the deployment has five switches, about thirty active network ports, and a wireless network. Our research group uses the deployment for general network access. We also deployed Lithium in a home network to address an interesting management problem in such networks: usage cap management. Using Lithium's configuration model, we were able to easily set up network policies for home networks that dynamically react to meaningful network events such as monthly data usage capacity reached. Our deployment and evaluation show that Lithium can simplify configuration management for real-world configuration scenarios in campus and enterprise network settings as well as in solving home network management problems.

This paper presents three contributions. First, we introduce the concept of event-driven network control and identify four different domains for expressing event-based network policies: time, history, user, and flow. Second, we design and Lithium, a new event-driven control framework that can support policies that incorporate these domains. Third, to demonstrate that Lithium can simplify network management tasks in different types of networks, we deploy Lithium in an enterprise network and a home network, two real-world settings where network configuration is challenging today. We evaluate Lithium in both of these settings to show that it is both usable and feasible.

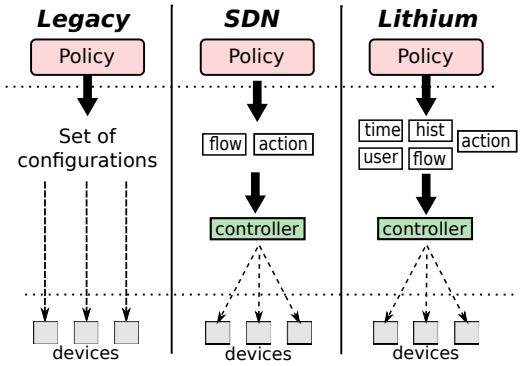


Figure 1: *Comparing Lithium to existing approaches. The legacy approach to network configuration uses a variety of protocols from multiple layers to convert high-level network policy to a set of configuration files. SDN advocates a centralized control with a software program, but is still limited to flow-action pairs. Lithium uses a much richer set of control domains.*

2. Lithium: Event-Driven Network Control

Although the research community has seen numerous calls for developing higher-level network configuration languages, few such high-level languages have emerged. The lack of a higher-level language is not for lack of trying; rather, we believe that the underlying network "runtime" remains too complicated to directly support higher-level programming domains. *Software defined networking* advocates network control from a centralized software program, creating the possibility for a range of improvements for expressing higher-level network policies. Although *OpenFlow* is one mode for central control, specifying actions based only on properties of traffic flows inherently limits the expressiveness of network policies. Lithium attempts to fill this gap by allowing operators to use additional event-driven control domains for more declarative and expressive network policies. Figure 1 shows how Lithium extends the control model that current software-defined networking systems provide.

2.1 Event-Driven Control Domains

Domains for expressing event-driven control. Table 1 shows possible domains along with example policies where each condition might be useful. The first three rows are ex-

domains	*Examples*
Time	peak traffic hours, academic semester start date
History	amount of data usage, traffic rate, traffic delay, loss rate
User	identity of the user, assignment to distinct policy group
Flow	ingress port, ether src, ether dst, ether type, vlan id, vlan priority, IP src, IP dst, IP dst, IP ToS bits, src port, dst port

Table 1: *Control domains, and examples of how a higher-level policy might use them. First three rows are event-driven control domains, meaning these domains can invoke an* event*, which in turn can change the network state. In contrast, flow is not a event-driven control domain.*

amples of event-driven control domains; the last row, *flow*, is not. Flow is a 12-tuple control domain that already exists in the OpenFlow specification: action is determined for the incoming packet based on the flow domain specification. For the purposes of this paper, we consider only the set of actions permitted by OpenFlow (*e.g.*, forward, drop, redirect); instead, we focus on how the network might support a richer set of conditions on which to perform action on a certain traffic flow. We focus on four domains for which the controller might take different actions:

- *Time.* Network operators often need to implement policies where network behavior depends on the date or time of day. For example, a campus network would want to manage traffic differently in semester breaks when traffic loads are less than they are during the academic year; similarly, operators may want to rate-limit non-essential traffic during peak hours. In a home network, users might want to use time as the basis for parental control. Operators normally rely on time-triggered scripts to implement these policies, but they are still low-level and error-prone.
- *History.* Operators sometimes specify policies whereby the behavior of the network depends on history (*i.e.*, past conditions or patterns in the network). Examples of historical information include amount of data usage (download/upload), data transfer rate over a particular time interval, or load on a certain port.
- *User.* An operator may wish to specify privileges for different users or groups of users. Moreover, user's privilege or status often changes due to various reasons. Legacy configuration systems do attempt to distinguish users through various methods, including VLAN tag, MAC address, IP address, port numbers, or any combination of these fields.
- *Flow.* A network flow contains fields such as the source/destination MAC address, source/destination IP address, TCP/UDP port numbers, etc. Network operators want to specify different network behaviors based on various field values in multiple layers, specified in a packet or flow.

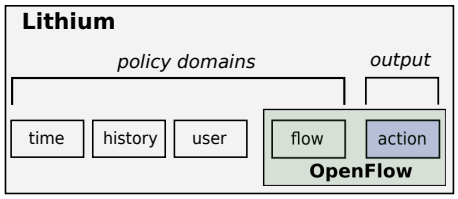


Figure 2: *Conceptual representation of Lithium. Lithium extends OpenFlow, which only uses the flow as a condition for taking an action, by supporting additional domains to be used as condition when deciding traffic behavior.*

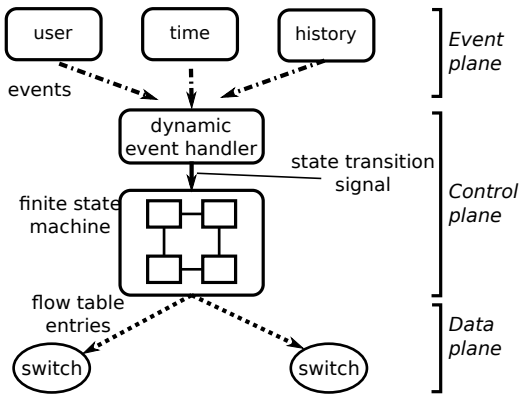


Figure 3: *High-level design of Lithium.*

Figure 2 shows the conceptual illustration of how Lithium extends OpenFlow's expressiveness, since OpenFlow supports actions based on flow properties only. We do not claim that the set of domains that we have outlined is complete, but we demonstrate in later sections that this set is expressive enough to support a wide range of network policies in various types of network deployments.

2.2 Designing Lithium

We now explain how we design Lithium to support the four control domains. Figure 3 shows the high-level design of Lithium. Lithium has two main components: (1) a *finite state machine*, and (2) a *dynamic event handler*.

2.2.1 Finite state machine

Lithium uses a finite state machine model to express and enforce network policies. Event-driven control domains essentially invoke network events, which in turn can trigger state transitions in the finite state machine. Incoming traffic is subjected to different actions depending on the current network state.

States. A *state* in Lithium's control model maps to a particular network state that correspond to a set of static network policies. A set of domain values represents a state. For example, a state may be a certain time of day, the presence

of a certain user on the network, the traffic that an application has sent over the past day, or even some combination of these types of values. In Lithium, each end-system device is mapped to a single state, which is identified by its MAC address. The state of a device at any time determines how a network switch will forward its traffic. In this paper, we assume an end host device can only be assigned to a single state, for simplicity.

Events. Networks may experience a variety of events, ranging from intrusions to changes in traffic volumes to the arrival or departure of new hosts; the Lithium controller responds automatically to these events. We define an *event* as any change in state (*i.e.*, a change in time, history, user, or flow). Any event can induce the controller to execute a different set of actions for a particular end host by updating flow table entries. In summary, a change in a domain value (*i.e.*, a state change) is an event that triggers an ultimate change in the flow table entries of one or more network switches; the network policy determines exactly how these flow-table entries should change for different events.

2.2.2 Dynamic event handler

The Lithium *dynamic event handler* processes network events that arrive at the controller. It waits for events on a TCP socket and determines where incoming events are coming from and whether they were sent by a known event dispatcher. If the handler recognizes an event's source and message type, it processes the event and executes a state transition for the associated device based on accompanying event parameters. The event handler also manages flow table state by deleting old flow table entries on switches, which will cause the switches to receive the correct set of entries the next time they query the controller.

Lithium reacts to domain events instead of proactively fetching domain values. We do this for practicality: if the controller proactively queried the network domain, then every packet would need to visit the controller to determine its action; this approach scales poorly in operational OpenFlow networks because of flow setup overhead. Recent work in declarative configuration languages suggest methods for mitigating this scalability limitation [29].

3. Towards "High-Level" Configuration

Although Lithium presents a new programming model for specifying network policies, operators still need a high-level *language* for easily specifying these policies. Lithium does not currently have a high-level language for expressing network policies. The policy itself is embedded in the controller as a program expressed in C++, which offers the convenience of a widely used language but does not offer the potential benefits of other high-level languages. This paper focuses on designing the back-end framework that can more readily support a high-level language. We envision a high-level language with the following features:

- *Declarative Reactivity:* Operators should be able to define reactive device permissions in a declarative way, by describing when events happen, what changes they trigger, and how permissions change over time.
- *Expressive and Compositional Operators:* A powerful, expressive set of language constructs should allow operators to build reactive permissions out of smaller reactive components.
- *Well-defined Semantics:* The language should have simple semantics, simplifying policy specification when compared with current methods of configuration. In addition, the semantics should provide a basis for building a variety of related tools, such as static analysis tools for policy analysis prior to deployment.
- *Error Checking & Conflict Resolution:* Leveraging well-defined, mathematical semantics, the language should be able to automatically inspect its own configuration and detect errors or conflicting statements or operational goals after reconfiguration attempts.

Embedded domain-specific languages (DSLs) allow network operators to specify event-based policies at a higher level of abstraction. Currently all domains in Lithium are represented with a single datatype so Lithium's policies are technically not an embedded domain-specific language, but we are currently extending Lithium to provide support for embedded DSLs. DSLs that facilitate functional reactive programming (FRP) are a natural fit for event-driven programming models like Lithium since they make it easy to specify programs that operate on streams of events. Possible future work might involve extending a DSL like Nettle [33, 37], Frenetic [17], or NetCore [29] to express policies in Lithium at a higher level than is currently possible with any of these languages.

4. Implementation

We now describe the implementation of Lithium and how various components support policies that build on the domains we outlined in Section 2.

4.1 Lithium Controller Overview

Our current implementation uses NOX [20] version 0.6.0, which is compatible with OpenFlow version 1.0.0. The controller manipulates the flow table entries for all connected OpenFlow switches, as a conventional NOX controller does. Through the predefined API in NOX, the controller adds, deletes, and modifies flow table entries. As with a conventional OpenFlow controller, the Lithium controller makes decisions about distinct flows and enforces these decisions by installing flow table entries in individual switches.

Figure 4 shows an overview of Lithium's implementation. As in conventional OpenFlow-based network, when a new flow arrives, it is directed to the controller, at which point the controller determines the state that corresponds to the flow,

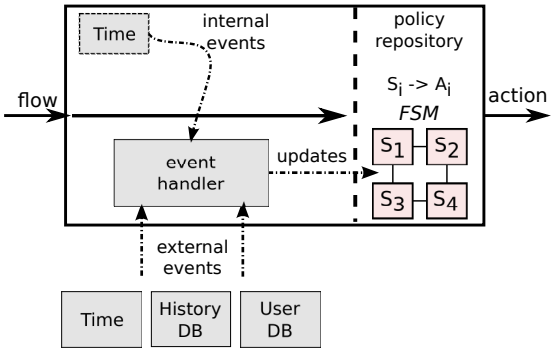


Figure 4: *How Lithium processes flows using the extended set of domains from Section 2.*

based on the flow's MAC address. Each state, S_i, has a default associated action, A_i, that determines the action that a switch should take for that flow and state (*e.g.*, forward, drop). The controller's event handler processes dynamic network events invoked by internal or external modules. Based on the identity of the event, the state to which the flow is mapped to can change, thereby altering how switches in the network process the flow.

The processing machinery for Lithium is written in C++. Excluding the code to implement policy specifications, the central controller is implemented with 1,266 lines of C++. The ultimate size of the code running at the controller of course depends on the network policy itself, since currently the policy is programmed and encoded in the controller (also in C++). As we explained in Section 3, a separate high-level language for specifying policy could also be run in conjunction with the existing controller.

4.2 Event-Driven Control Domains

To support the three additional domains in addition to to the *flow* domain that already exists in OpenFlow, Lithium has several additional components that we describe below: a user database, a history database, and a time module. Some components can be implemented within the controller itself, while some components, such as the history database, are implemented as a separate piece of functionality.

User database. Lithium allows a user's identity to be mapped to network identifiers, as stated above. Lithium also allows users or devices to be assigned to different policy groups in a declarative way. Moreover, different network events can change a user's privilege level; for example, a successfu login at a Web authentication portal could change a user's privilege level. These external events can update the internal *user DB* shown in Figure 4. The runtime system should allow different network policies to be applied to different users. For the case studies Section 5, we use MAC address to distinguish between users and user devices.

History database. The history domain could conceivably be used to express a wide range of values ranging from historical data usage for past five minutes to the average throughput for a day. We have currently set up a history database especially for tracking the Internet usage of end-host devices, differentiated by their MAC address. The database is updated periodically with the new data usage information. Operators can configure the history database so that it invokes events when specified conditions are met: in our case study, data usage of particular hosts reaching their capacity threshold value. Based on the usage value as well as the capacity set for the host device, forwarding policy for each device's traffic is determined. We demonstrate Lithium's support for historical queries in Section 5, in more detail.

Timing module. Lithium supports policies that use time of day as a condition through a separate module. As in Figure 4, Lithium has an internal time module which an incoming flow uses to fetch the time at the moment the flow enters the controller. A separate external time module could be used, as well. The time module is often used in conjunction with the history database many historical policies naturally incorporate time ranges (*e.g.*, data usage over one gigabyte from 2 to 3 p.m.).

4.3 Processing External Events

As shown in Figure 4, Lithium contains a dynamic event handler, which is responsible of processing external events and signaling the internal finite state machine in turn to make relevant changes to the underlying network. The event dispatcher sends a message to the central controller via a TCP connection over port 9999. We have implemented a client program in both C and Python; therefore, to send events to the Lithium controller, the network entity must only run the program we provide or generate a message that conforms to the specification in Table 2.

The event message has two header fields: *payload length* and *message type*; and three body parameters: *sender ID*, *device ID*, and *event ID*. Table 2 shows the format of an event message packet. The header contains the total length of the packet and the message type. In the body, *sender ID* is the identification number of event sender or the dispatcher. The controller uses this ID to determine the origin of the event; the current implementation uses pre-defined numbers, but the sender ID might also be the MAC address associated with the device that generated the event. The *device ID* is the MAC address of the host that may be affected as a result of the event. The *event ID* specifies the type of event that has occurred; the event handler uses the event ID, plus the host device's current state, to determine the subsequent state for the device.

If the event triggers a state transition, Lithium deletes existing flow table entries in all switches. This is to make sure

payload length	msg type	sender ID	device ID	event ID

Table 2: *Event message payload format.*

new incoming packets do not match the outdated flow table entries, because different states have different set of forwarding rules. Correct flow table entries will be populated onwards.

5. Real-World Deployments

In this section, we describe the deployment of Lithium in two real-world scenarios where network configuration can be difficult: enterprise networks and home networks. In Section 5.1, we study how Lithium and our new configuration model can be used to solve a common access control problem that arises in computer networks, furthermore how to extend and simplify the task; we describe the deployment of Lithium in an operational campus network. In Section 5.2, we explain how Lithium can be applied to home networks to make certain network management tasks easier, and to facilitate the outsourcing of some aspects of home network management.

Each of the deployment scenarios below currently exercises different domains that Lithium supports: Our campus network deployment focuses on Lithium's support for handling different types of external events, while our home network deployment focuses on Lithium's support for time and history. In practice, both of these scenarios could be extended to use more domains to support a richer set of policies. In addition to the policies that we have implemented in current deployments, we describe how our current deployments can be easily extended to support a richer set of policies.

5.1 Campus Network

We describe how we have deployed a network access control system within a single network using Lithium. The function of the system we have deployed mirrors and extends an access control system in our campus network, which is currently based on a complicated VLAN-based setup. The Lithium state machine implements basic states that are associated with network access control and were inspired by the policies that the VLAN-based system implements. The resulting deployment has several distinct advantages over the existing deployment, which has several shortcomings: The current framework is *hard to manage*, due to the distributed configurations; it is *too coarse-grained*, due to the limited number of VLAN groups defined; and it is *too static*, due to the fact that it is difficult to remap clients from one VLAN to another when various network events occur.

We begin by describing these policies; we then explain how Lithium processes events from various other network components, as well as how the system might be extended to process other event streams. Finally, we describe the deployment and operation of this access control system in practice.

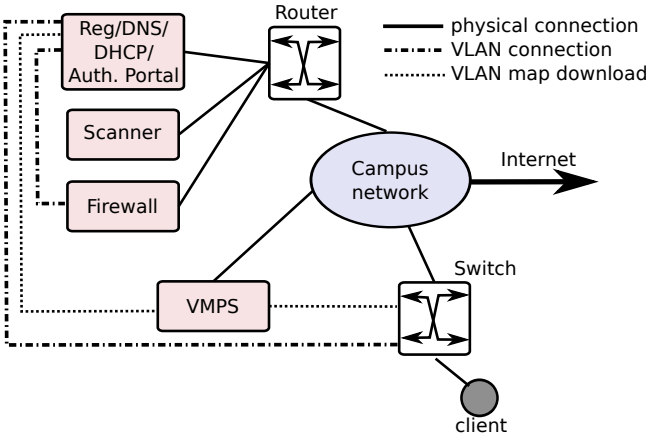


Figure 5: *The legacy system architecture.*

5.1.1 Current approach

Figure 5 shows the current access control system, which we will call the *legacy system* for now on. The system is based on VLAN technology and dynamic manipulation of firewall rules. Network operators must enable VLAN functions in all switches by adding VLAN assignment commands to switch configuration files. Every port that has a connection in a switch needs a set of VLAN commands. Although the setup we describe is specific to our network, many campus network operators we have talked to describe similar setups and configurations on their campus networks. Essentially, there is no "out of the box" solution to for implementing dynamic network access control.

The network has two separate VLAN groups: for registered users (*reg-vlan*) and the other for unregistered users (*unreg-vlan*). There are also two DHCP servers, one for each VLAN group. If an unknown device appears in the network, it is assigned to the *unreg-vlan* VLAN group; known devices stay in the *reg-vlan* group. Based on this VLAN assignment, a device's DHCP lease request reaches a different DHCP server and receives IP addresses for different address ranges. The DNS server returns different DNS records based on the host's IP address. If request comes from the unregistered group, all requests are resolved to the authentication web portal, where the user must input credentials; registered users' DNS requests resolve to their designated destination. The scanner is responsible for scanning the host devices for known vulnerabilities when during the registration process. Switches periodically download the VLAN group-host mapping from VLAN Management Policy Server, or VMPS [1] so that they have the most updated mapping between hosts and VLANs.

Access control lists in firewalls ensure that devices in the *unreg-vlan* group are only allowed to connect to the authentication Web portal through port 80 or 443; all other traffic is blocked. Traffic from host devices in the *reg-vlan* group is basically allowed with few exceptions due to security issues. Specific source ports can be allowed, allowing host devices

user	time	history	flow	action
un-auth.	*	*	dst. port == 80/443	redirect to auth. portal
scan	*	*	ip addr == scanner's IP	allow
guest	*	*	*	allow
student	*	*	*	allow
guest	**2pm-6pm**	**data usage>1 GB**	*	**block**
infected	*	*	*	**block**

Table 3: *Policy for each state in campus*

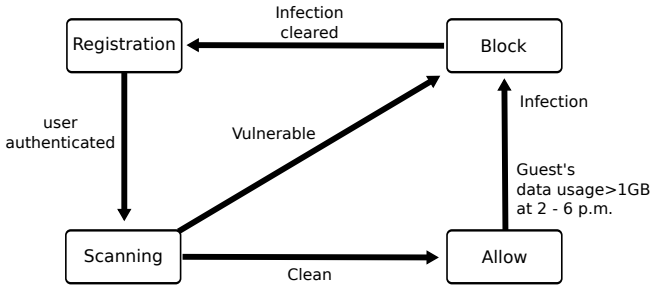


Figure 6: *Transitions and events in campus.*

to host services like Web server, based on the options selected when registering the device. This is done by inserting additional firewall rules dynamically in the firewall device.

5.1.2 Lithium approach

We start by enumerating the actual required network policies currently enforced in the network by the legacy system. Additionally, we add more useful policies that can be achieved easily through Lithium, but far more harder to accomplish with the legacy system. Figure 3 summarizes the network policy. Additional network policies achieved by Lithium are shown in bold.

States. In this case study, we define four different network states that that are represented by combinations of values in the underlying network domains: *Registration*, *Scanning*, *Allow*, and *Block*. Hosts in the *Registration* state are unregistered devices that have not yet authenticated via the authentication Web portal. In this state, HTTP traffic is redirected to the authentication Web portal; all other traffic is dropped. Host devices in *Scanning* state are only allowed to interact with the vulnerability scanner in the network, which checks the host device for known vulnerabilities. All other traffic will be dropped. The *Allow* state allows all traffic (subject to normal operational firewall rules), and *Block* state blocks all communication to the rest of the network. *Peak guest* blocks guest traffic which hits the data usage limit of 1GB during the peak hour 2–6 p.m. everyday. Although not the actions are not shown in the table, both DHCP traffic and DNS traffic are always allowed by default.

Events and transitions. After defining the states, an operator must specify the transitions between them, as well as the events that trigger each transition. Figure 6 describes the transitions and events for this operational deployment. The host's traffic behavior solely depends on its current policy state. To ensure that switches forward traffic according to policy, flow table entries that have the host's MAC address are deleted and re-populated in all connected switches when transitions happen between states.

Event dispatch. Our deployment of Lithium in a campus network has several types of devices and systems that send events to the Lithium controller. We have deployed a set of devices that could trigger state changes that alter a host's access rights; other types of network policies might take input from different event dispatchers and require policies that build on a different set of domains. Events are dispatched according to the format presented in Section 4: Each event is accompanied with a *sender ID*, which identifies the event dispatcher; and *event ID*, which differentiates different events from a single event dispatcher; and a *device ID*, which represents the host device that may change its state according to the event.

- **Authentication portal.** This component authenticates users who attempt to use to the network. Any HTTP request from an unauthenticated user is redirected to the authentication portal, which requests the user's login credentials. Only when the user provides the correct credentials (in this case, a combination of username and password) does the authentication portal invoke the notifier executable to send a notification to Lithium. A successful authentication moves user's device to the *Scanning* state.

- **Scanner.** There are a variety of proprietary and free scanners; our deployment uses a port scanner as a proof of concept. We use nmap [36] to scan for open ports of the host machine. However, any scanner would work as long as it is able to dispatch events to the Lithium controller. After the scanning is done, the scanner notifies Lithium so that the host's state can change to *Block* (if vulnerabilities are found) or *Allow* (if host turns out to be clean) based on the result and depending on the group the scanned host belongs to.

- **Intrusion detection system.** Lithium uses Snort, an open source network intrusion and detection system [35]. The IDS inspects the payload of traversing traffic and generates an alert if suspicious packets are detected. This actually causes the state of the hosts to change to *Block* state, as shown in Figure 6. As the result, traffic of the host is dropped. It is possible for the

operator to revert back to the original state dynamically without restarting the controller by sending an appropriate remedy notification to the dynamic listener.

- **History database.** We have built a separate history database, which continuously tracks the data usage of each host device in the network and stores it for future use. A graphical user interface is available via a web browser to set the limit, or cap, data usage value for each device. If the actual data usage hits the limit value in the specified block of time, (*e.g.*, 2–6 p.m.), a trigger invokes a dynamic event towards the controller to enable the transition.

5.1.3 Deployment experience

Figure 7 shows the current deployment of system in the our network. There are five OpenFlow-enabled switches (two HP switches, two NEC switches, and one Toroki switch) deployed across the campus, forming our system network that spans three buildings. The NEC switch's OpenFlow switching performance is 136 Gbps, 101.2 Mpps. The controller runs on a Dell PowerEdge 1950 machine running Ubuntu Linux 9.04 with 8 GB RAM and Intel Xeon Quad-core 2.5 GHz processors. It has 1 Gbps network interfaces attached to it. Two access points are deployed around the lab and hallway in building #3 to enable users to connect via a wireless network. Two /27 IP subnets are reserved for the Lithium network: one for management and the other for data traffic; thus, the network is separate from the existing access control network managed by the legacy system.

A user who is within range of the access points in the network can connect to the Lithium network by establishing a wireless connection to our predefined SSID with a shared key. Immediately after successful connection establishment, the DHCP server assigns the host a public address from the IP pool we have reserved for the Lithium network. If the controller has never previously seen the host's MAC address or the host has never been authenticated, the traffic will be redirected to our authentication portal. The user must input valid credentials to the portal so that the host machine's state can transfer from the *Registration* to *Scanning* state. At this point, the vulnerability scanner initiates a scanning on the host machine. If the host machine is considered to be clean, the state will finally change to *Allow*.

The deployment was successful and reliable enough to be used as a daily access point for many users, especially when the pre-existing VLAN-based network access control system experienced configuration or connectivity problems.

5.2 Home Network

Internet service providers around the world are beginning to deploy monthly "usage caps", which limit the amount of traffic that any particular subscriber can transfer within a billing cycle. For example, in the United States, Comcast has deployed a monthly usage cap of 250 gigabytes,

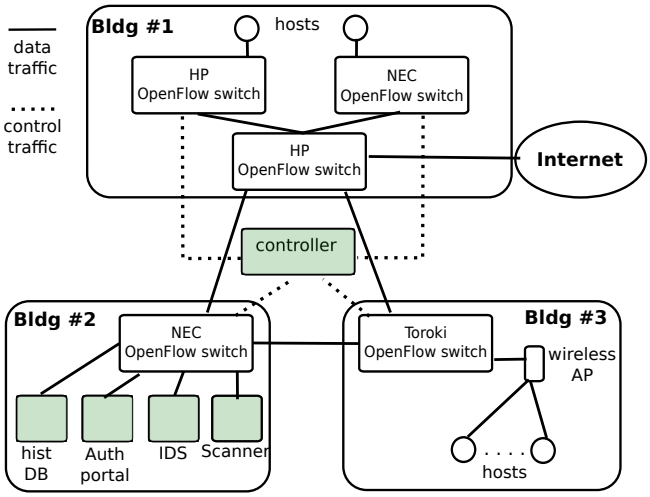


Figure 7: *Campus network deployment.*

and AT&T DSL and U-Verse users are subject to a 150-gigabyte usage cap. Given the growing diversity of applications, the increasing demands of certain applications (*e.g.*, high-definition NetFlix movies), and the relative opacity of usage information to individual users [9], consumers need better ways of managing these caps.

Unfortunately, intuitive tools for helping users monitor and manage their usage caps effectively do not exist today. To fill the need for such a tool, we are developing a system to help consumers perform flexible and fine-grained monitoring and management of their usage caps. Previous user studies have identified that consumers want information to help them manage the usage cap to ensure all household members enjoy good performance for their activities, while also avoiding overage charges or service interruptions if the "cap" is exceeded [9].

Lithium empowers unskilled home users to be semi-operators in their own household, allowing them to perform general resource management of data usage. Through our system, home users can set usage *caps* or *limits* on each device, user and household and even facilitates other more sophisticated policies, such as parental control.

5.2.1 Current approach

Some ISPs provide a Web interface where a user can view and monitor the aggregate usage at a household level, but we are not aware of any management system that allows users to specify any type of control, in addition to plain usage monitoring, even though most existing home networking infrastructure supports such functions.

One possible approach is manual configuration of the home router by a skilled home user. This will likely involve manual manipulation of firewall-like rules in the home router or modem, which is difficult to land a correct configuration, even harder to maintain. Another approach is the ISP performing host device level access control. However, this in-

user	time	history	flow	action
home device	monthly	data usage>=cap value	*	block
home device	monthly	data usage<cap value	*	allow

Table 4: *Network policy for home network usage management.*

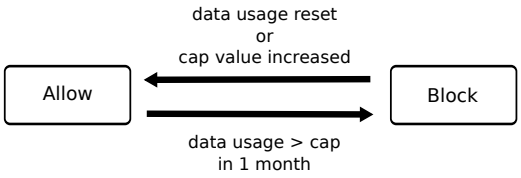


Figure 8: *States and transitions for a simple example of usage cap management in home networks.*

corporates dynamic manipulation of access control lists in firewalls or routers managed by the ISP.

5.2.2 system approach

Assuming the ISP is willing to provide this service to home users, we use Lithium to build a working solution.

States. Table 4 details the network policy for the home network usage management system. This policy requires two states states here: *Allow* and *Block*. Host devices in *Block* state are not allowed to have internet connection while host traffic in *Allow* state is all allowed without exception. Host devices are assigned to the *Allow* state by default whenever a new monthly cycle starts.

Events & transitions. There are two transitions in the home deployment which are driven by two events: cap reached and cap released. The transitions are shown in Figure 8. *Cap reached* event is triggered when the data usage exceeds the predefined cap value. The databased detects the situation and raises an event to notify the central controller. This event triggers a host to move to the *Block*state, *i.e.*, *Capped* state, thus blocking all traffic between the host and the Internet. *Cap released* event is generated by the database if it detects that the cap value is again over the actual data usage, or if the capping functionality is disabled. The event makes the host's state change to *Allow*, *i.e.*, *Uncapped* state where all traffic is allowed.

Event dispatch. Lithium's home network solution uses the same history database used in Section 5.1. As described earlier, the history database sends an event to the central controller to invoke transitions between states. For this particular task, the time value is set to monthly billing cycle of each household.

5.2.3 Deployment experience

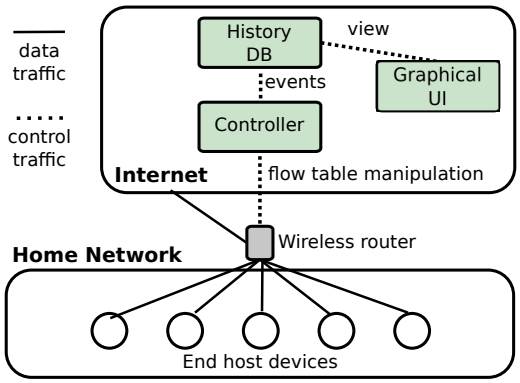


Figure 9: *Home network deployment.*

Figure 9 how Lithium can be used for home data usage management system. We use a NetGear WNDR 3700v2 gateway as the wireless home router, which runs a custom made firmware based on OpenWrt. We use PostgreSQL as the back end database and an Apache web server is set up to provide a graphical interface for monitoring as well as managing usage in the home network. Home users interact with the graphical interface through a Web browser to view and set usage caps on devices and users. When a cap is reached, relevant devices will be cut off from the network and will be informed by an automatically generated email.

We have deployed these wireless routers in two households and are planning a more extensive deployment in a dozen households to support a broader user study on home network management interfaces. More generally, the deployment shown in Figure 8 can allow for a variety of home network management tasks to be "outsourced". Several projects have advocated slicing the home network infrastructure to allow multiple service providers to use and control the home network infrastructure simultaneously [38]; others have even proposed outsourcing some aspects of network control entirely to third parties [13]. This deployment of Lithium could easily support such a broader range of management tasks and scenarios.

6. Evaluation

In this section, we evaluate the usability and feasibility of Lithium. Evaluation of network management systems is difficult since there are no quantitative metrics that allow us to compare Lithium to existing approaches. The state of the art in complexity evaluation is syntactic: the methods only apply to networks where one can analyze the syntax of low-level network configurations [5, 27]. Because Lithium takes an new approach to configuration that does not rely on low-level configuration, these approaches do not apply. We concede that there is definitely a need for more comprehensive metrics for measuring the complexity of various network man-

agement systems. Given that no such framework exists, we focus our evaluation on two aspects:

- *Usability (Section 6.1).* We perform a qualitative evaluation of how Lithium can make network management tasks easier for network operators and users. Our aim is to argue informally that Lithium is more manageable than existing network configuration frameworks.

- *Feasibility (Section 6.2).* We perform a quantitative evaluation to demonstrate that the performance overhead imposed by Lithium is negligible compared to existing OpenFlow-based systems with published performance numbers (such as the deployment in the Gates building at Stanford University).

6.1 Qualitative Evaluation: Usability

Measuring whether a certain network management or configuration system is more effective or usable than existing solutions is difficult, due to the lack of standard usability metrics. Here, we discuss several criteria that can possibly reflect the usability of a management system. We do not claim that Lithium can achieve networking tasks that operators cannot in some way perform with legacy methods today. In fact, with the variety of configuration commands used in network devices combined with additional customized scripts and tools, legacy systems are remarkably flexible and capable. Rather than focus on what is possible (*capability*), we focus on the ease of solving certain problems (*usability*).

More expressive with fewer touches, fewer scripts. When configuring and managing a network, fewer touches to network configuration may reduce configuration errors and overall workload for network operators. For example, consider adding or deleting a VLAN group in the previous campus access control case study through legacy methods. Essentially, operators have to touch multiple switches that are responsible of forwarding traffic for the group. In each switch, ports that are related to that VLAN group should be configured in detail. In Lithium, the same task can be achieved by adding a new policy state in the controller if necessary without touching other network devices, and additional scripts are not required as well.

Fewer distinct scripts are desirable as well because they become harder to manage and maintain as the number grows. Lithium maintains to be very expressing, leveraging the event-driven control domains while refraining from using separate scripts to do the job.

More general. Lithium is general enough to solve a variety of network management tasks. The case studies in the previous section demonstrate this generality. Legacy configuration techniques would require two different approaches for each task; campus network access control requires VLAN configuration, and the home network management case requires a separate script is required to update *iptable* rules in the local wireless router.

More portable. In legacy configuration methods, policy is enforced by configuring individual devices in the network, so applying an identical or even similar network policy to another network with different physical infrastructure may be cumbersome. Because network policy depends on on individual configuration files from network devices, configurations may be specific to network topology, switch vendors and software versions, and other specifics of a particular network. In Lithium, policies might be more easily ported, since the control program could be written in a more device-independent manner.

6.2 Quantitative Evaluation: Feasibility

As the size of network grows, the key factors that affect the feasibility of Lithium are (1) forwarding performance; (2) the size of the flow table in switches; and (3) the load on controller and switches. However, as OpenFlow is used for materializing Lithium, the performance and feasibility also heavily relies on OpenFlow's capability. Although this paper's main focus is about presenting a better *network control model*, we do performed some basic quantitative evaluation to prove two main points: First, it is indeed feasible to run Lithium on real operational networks, *e.g.*, campus or home; Second, Lithium does not incur additional processing overhead compared to the basic NOX controller.

6.2.1 Forwarding performance & latency

We first evaluate the throughput and latency of Lithium for both case studies from Section 5. We emulate synthetic traffic with `netperf` [34]. For throughput measurement, we send a stream of TCP packets between two hosts for at least 120 seconds until it achieves a confidence level of 95%. For latency, we measure the round trip time, or *RTT*, between two hosts, collecting around 500 round trip time measurements.

Campus network measurements. In our campus access control case study, presented in Section 5.1, we use production quality OpenFlow-enabled switches from professional network switch vendors, NEC, HP, and Toroki. These switches implement OpenFlow as a kernel module and performs packet forwarding and flow matching in the *hardware*. Measurement testbed is set up as following: Two end hosts are directly connected to one NEC switch, and the controller is one-hop away from the NEC switch.

Table 5 shows the throughput and latency measurements in the campus network between three systems on synthetic traffic: (1) baseline (without system or NOX), (2) with NOX switch implementation, and (3) with Lithium. The baseline is when only basic VLAN configuration is used to connect the two end hosts, and we assume the performance of baseline is same as the existing configuration without OpenFlow. NOX switch is an existing implementation which comes with the NOX suite, and simply makes OpenFlow switches to forward

every packet it receives to the correct output port based on the destination MAC address. The comparison shows negligible performance difference between baseline, NOX switch implementation, and Lithium for both throughput and latency. The table also suggests that Lithium does not introduce additional delay for policy lookup when compared to the NOX switch implementation.[1]

Home network measurements. Table 6 shows the measurement results in the home network (wired and wireless) between three systems on synthetic traffic. The use of the NOX switch significantly hinders performance because the OpenFlow module for OpenWrt is currently written in user space; there is negligible difference between the NOX switch implementation and Lithium.

Further investigation revealed that the home gateway is the bottleneck cause for the performance degradation in the case of home network. The NetGear WNDR 3700v2 gateway we use has a 680 MHz MIPS 32-bit processor, and running an OpenFlow module causes the CPU cycle to hit close to 90% constantly during the test. This overhead is caused by the fact that OpenFlow instances runs in the user-space instead of as a kernel module in the gateway. More optimization iterations can boost the performance further, but is limited. Open vSwitch [2] is a production quality virtual switch, which can be installed in linux-based wireless routers (*e.g.*, OpenWrt). Open vSwitch has OpenFlow as a built-in kernel module, hence, we expect the performance to be comparable to the baseline.

We expect that if the OpenFlow module were implemented in the OpenWrt kernel, then both the NOX switch and Lithium setups would forward traffic much more quickly. This trend is apparent in the latency measurement as well. These results show that, while enabling tue current OpenFlow module for traffic forwarding in OpenWrt significantly slows forwarding performance, Lithium itself does not incur additional performance degradation compared to the base NOX implementation, despite the additional modules and network policies embedded in Lithium.

6.2.2 Flow table size

Next, we study the scalability of Lithium in terms of flow table size. There is a limit on the number of flow table entries a single switch can maintain: 131,072 for exact match entries and 100 for wildcard entries, according to the OpenFlow specification [31]. An exact-match entry has all 12-tuples specified with some value, while a wildcard entry contains one or more tuples as wildcards.

To help us understand how Lithium might scale on larger networks and in a home network, we performed a trace-based analysis using data from a large campus network, and a typical home network, as shown in Figure 10. For the campus network, we captured an hour of traffic at the campus network gateway on a typical weekday from 2 p.m. to 3 p.m. For the home network, we captured an hour of traffic on the home gateway router between 9 p.m to 10 p.m. We analyzed these traces to infer the number of unique flows in specific time intervals. With this information, we measure how many flow-table entries are needed to manage the traffic.

In the campus network, as shown in Figure 10a, the number of flows is always below 25,000 along the measured period, which is far below the 131,072 threshold for exact match entries. Thus, in a typical campus network, Lithium will likely scale in terms of flow table entries that each switch must store. Moreover, it is likely that more than a single switch would be responsible to handle a /16 subnet, thus considering these flows will be spread out to multiple switches, Lithium is scalable in terms of flow table size. Figure 10b also suggests the number of flow table entries needed for a typical home network traffic would not exceed the limit.

Additional way to mitigate the growth of flow table entries is to use wildcard entries. It is possible to dramatically reduce the amount of communication between the controller and switches by using several wildcard entries wisely. For example, Figure 10 shows that DNS packets are responsible for much of real production traffic. ARP packets and DHCP packets will account for a notable portion of the traffic as well, though not depicted in our figure.

6.2.3 Scalability: Load on controller and switches

The process of installing flow table entries in switches by a central controller affects the scalability of both the controller and switch. Basically, if a packet arrives at a switch and cannot find a matching flow table entry, the switch buffers that packet and consults the central controller, where the network policy is stored. Then, the controller sends the decision to the switch, and after a flow table entry is inserted, the packet is processed based on that entry.

This overhead, which is commonly known as *flow setup overhead*, is inherent to the current design of OpenFlow and the NOX controller. This control plane interaction not only incurs additional delay in packet forwarding, but also increases the load on the controller as well as switches as they sacrifice certain amount of CPU cycles and memory to perform the task. In other words, poor network topology and system design can lead to overwhelming *new flow setup requests*, overloading the switch and/or the central controller.

In fact, in our lab test, continuous stream of 2,000 unique flows was able to overwhelm the production quality OpenFlow-enabled switch we have in several seconds, after hitting 95-100% CPU utilization. Figure 11 shows the *processing time* of the Lithium controller with different traffic loads. *Processing time* is defined as the time the controller uses to process a new flow request and send out a decision to the switch, after the request arrives at the controller. The number of unique flows, and, thus, the number of requests ar-

[1] NOX has sometimes performed worse than Lithium; we suspected that the default wildcard entries we insert into Lithium might be responsible for improved performance. We added the wildcard entries in NOX and saw a performance improvement.

System	Throughput (Mbps)	Latency (RTT *avg.*)	Latency (RTT *95th %*)
Baseline	941.34	0.1248	0.1258
NOX	940.14	0.1215	0.1246
Lithium	939.94	0.1239	0.1250

Table 5: *Performance comparison in campus network. Baseline is the system without OpenFlow enabled. NOX represents a system with OpenFlow running controlled by the basic NOX switch implementation. Lithium is our own controller implementation with the finite state machine and dynamic event handler module.*

System	Throughput (Mbps)		Latency (RTT *avg.*)		Latency (RTT *95th %*)	
	wired	wireless	wired	wireless	wired	wireless
Baseline	93.76	35.16	1.739	2.584	1.779	2.903
NOX	21.13	16.29	2.309	3.323	2.428	3.976
Lithium	22.52	16.93	2.327	3.317	2.427	3.872

Table 6: *Performance comparison in home network. System definition is same as Table 5.*

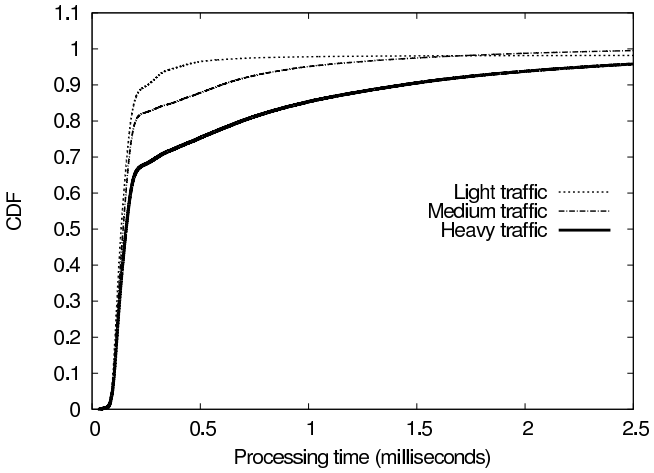


Figure 11: *Processing time difference at the controller with different traffic load. Light traffic is a continuous TCP stream traffic with 500 unique flows. Medium traffic identical but with 1,000 unique flows, and heavy traffic with 1,500 unique flows.*

riving at the controller, clearly affect the performance of the central controller.

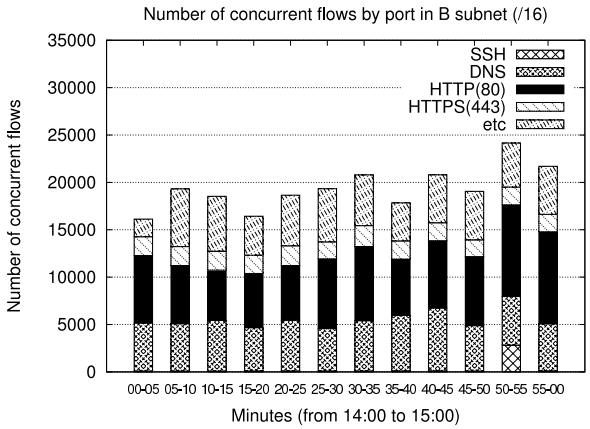
Our test results suggest that the number of interactions between the controller and switches in the control plane is the key factor that determines the scalability of Lithium. Fortunately, several recent studies tackle this overhead issue using clever techniques. For example, DIFANE suggests to lessen the controller load by deploying authority switches that cache authority rules and respond to normal switch's requests instead of the central controller [39]. DevoFlow uses several techniques simultaneously, (*e.g.*, rule-cloning, minimizing statistic requests), to minimize the control plane interaction [10]. Continuous advancement in improving the scalability of OpenFlow, or software defined networking in general, can help foster and enable the deployment of many network control systems that rely on or assume logically centralized control, such as Lithium.

7. Related Work

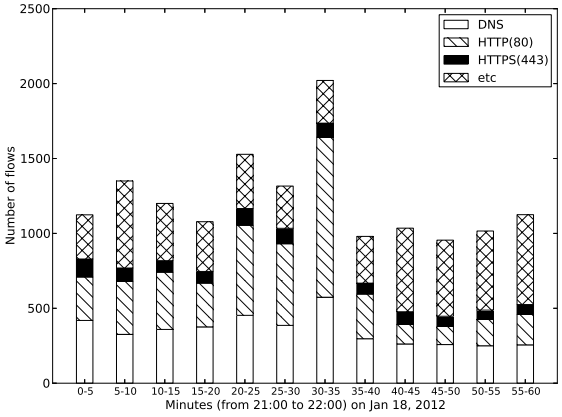
In this section, we survey previous work on software defined networking, languages for programming Openflow networks, and other network management frameworks.

The field of software defined networking (SDN) has roots in Ethane [6], RCP [15], and 4D [19]. The 4D architecture generally describes the separation of the control and data planes. Ethane embodies this architecture; it introduced a new method for configuring and managing a network through central command with a software software program that populates flow-table entries in network switches. Ethane's policy model and its policy language, *POL-ETH*, only allow limited configuration of static policies. Lithium complements Ethane's work by introducing a new policy model with a right abstraction, an attempt to reduce network complexity and ease the process of network configuration and making changes to it. The OpenFlow protocol standard is currently the *lingua franca* of software-defined networking [28, 32], and it is the standard on which Lithium builds. While OpenFlow is often used synonymously with software-defined networks, we note that the protocol standard is just one instantiation of software defined networking. Richer control frameworks such as Lithium also fall within the broader paradigm of SDN. In an early workshop paper, Nayak *et al.* proposed an SDN-control framework that processes certain network events for access control systems [30]; Lithium extends the design of that system to incorporate a much broader set of control domains. We also present a complete implementation and deployment of such a system.

There has been significant recent attention into developing languages for programming OpenFlow networks. FML [21] is a policy language for NOX that allows network opera-



(a) *Number of flows in campus network*



(b) *Number of flows in home network*

Figure 10: *Number of unique flows in (a) a /16 subnet from a large campus network, and (b) a home network. Time periods are selected based on the peak usage hours on a typcial weekday.*

tors to write and maintain policies efficiently in a declarative manner. Frenetic is a domain specific language for programming OpenFlow networks that provides programmers an abstraction of seeing every packet, even though the program is operating from a central controller [17, 18]. Nettle is a domain specific language implemented in Haskell that is used to configure BGP policies with more comprehensive abstraction calculation constructs [33,37]. Both Frenetic and Nettle use *functional reactive programming*, a declarative programming paradigm that allows programmers to express event processing very naturally; in principle, Lithium could serve as the underlying control system for either language. Frenetic's programming model is at a packet-level granularity, so it might be too low-level for the types of policies that we aim to implement with Lithium. NetCore is a declarative language for expressing packet forwarding policy on SDNs which improves the performance of Frenetic's see-every-packet abstraction by automating and optimizing the installation of flow table entries [29]. Nettle's and NetCore's focus on higher-level policies may make them more appropriate languages to run on top of Lithium.

Researchers have proposed various management frameworks and systems to make network configuration easier and less error-prone; none of these previous systems have used software defined networking as a platform for doing so. Ballani *et al.*, identified the problem of low-level configuration language used in network devices, and built CONMan, which uses higher-level modular building blocks to achieve same functions [4]. PACMAN [8] and COOLAID [7] by Chen *et al.*, both implement a higher-level construct than device-specified configuration languages to achieve certain networking tasks. However, the above studies focus on building a language that ultimately translates into low-level device-specified commands, while Lithium advocates a new configuration model as a stand-alone, event-driven program that captures the dynamics of a network well.

8. Conclusion

Configuring networks is difficult and error-prone, and operators must configure individual network devices using low-level, vendor-specific configuration commands. Ensuring that these devices achieve some higher level network-wide property is difficult, and certain types of policies cannot be expressed at all. Although the networking community has long agreed that networks need better network configuration languages, most network configuration languages remain low-level and are unable to allow operators to express high-level events and policies. We believe that enabling configuration languages that allow the expression of higher-level policies first requires a network control model that can process higher-level events in the first place. Towards this goal, we have designed, implemented, and evaluated Lithium, an event-driven network control framework that can implement policies based on four different domains: time, user, history, and traffic flow. To demonstrate the power and flexibility of these domains for expressing higher-level network policies, as well as the ability of Lithium to implement them, we deployed Lithium in two different network settings—a campus network and a home network. Our evaluations show that Lithium introduces negligible overhead beyond a conventional OpenFlow-based network.

Significant work remains in developing higher-level configuration languages for networks, and we believe that Lithium can potentially enable this next step by providing an event-based control framework that could act as a runtime for event-based network configuration languages. The advent of software-defined networking also potentially enables a wide range of network control frameworks that support an

even broader range of domains than the ones that we have explored in this paper. In this sense, Lithium serves only as the first step towards exploring how software defined networking can help simplify network configuration and management.

REFERENCES

[1] Configuring Dynamic Port VLAN Membership with VMPS, Jan. 2003. http://www.cisco.com/univercd/cc/td/doc/product/lan/cat5000/rel_4_2/config/vmps.htm.

[2] Open VSwitch, 2012. http://openvswitch.org/.

[3] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 111–122, New York, NY, USA, 2008. ACM.

[4] 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.

[5] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association.

[6] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking packet forwarding hardware. In *Proc. Seventh ACM SIGCOMM HotNets Workshop*, Nov. 2008.

[7] 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.

[8] 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.

[9] M. Chetty, R. Banks, A. J. Bernheim Brush, J. Donner, and R. E. Grinter. Under development: While the meter is running: computing in a capped world. *interactions*, 18:72–75, March 2011.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[11] A. Doria, R. Haas, J. Salim, H. Khosravi, and W. M. Wang. *ForCES Protocol Specification*. Internet Engineering Task Force, Mar. 2007. Internet Draft (http://www.ietf.org/internet-drafts/draft-ietf-forces-protocol-09.txt). Work in progress, expires August 2007.

[12] W. Enck, T. Moyer, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, Y. Sung, S. Rao, and W. Aiello. Configuration management at massive scale: system design and experience. *IEEE Journal on Selected Areas in Communications*, 27(3):323–335, 2009.

[13] N. Feamster. Outsourcing home network security. In *Proceedings of the 2010 ACM SIGCOMM workshop on Home networks*, pages 37–42. ACM, 2010.

[14] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proc. 2nd Symposium on Networked Systems Design and Implementation*, Boston, MA, May 2005.

[15] 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.

[16] A. Feldmann and J. Rexford. IP network configuration for intradomain traffic engineering. *IEEE Network*, Sept. 2001.

[17] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *International Conference on Functional Programming*, Sept. 2011.

[18] N. Foster, R. Harrison, M. Meola, M. Freedman, J. Rexford, and D. Walker. Frenetic: A high-level language for openflow networks. In *Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, Dec. 2010.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb. Detecting network-wide and router-specific misconfigurations through data mining. *IEEE/ACM Trans. Netw.*, 17:66–79, February 2009.

[24] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proc. ACM SIGCOMM*, pages 3–17, Pittsburgh, PA, Aug. 2002.

[25] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. King. Debugging the data plane with anteater. In *Proc. ACM SIGCOMM*, Toronto, Ontario, Canada, Aug. 2011.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[27] D. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjálmtỳsson, and A. Greenberg. Routing Design in Operational Networks: A Look from the Inside. In *Proc. ACM SIGCOMM*, Portland, OR, Aug. 2004.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM Computer Communications Review*, Apr. 2008.

[29] C. Monsanto, N. Foster, R. Harrison, , and D. Walker. A compiler and run-time system for network programming languages. In *Proc. ACM SIGPLAN-SIGACT POPL*, Jan. 2012.

[30] A. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic access control in enterprise networks. In *Proc. Workshop: Research on Enterprise Networking*, Barcelona, Spain, Aug. 2009.

[31] OpenFlow Specification v1.0. http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf.

[32] OpenFlow Switch Consortium. http://www.openflowswitch.org/, 2008.

[33] 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.

[34] S. Seshan and H. Balakrishnan. netperf: Network Performance Utility. ftp://daedalus.cs.berkeley.edu/pub/netperf, 1996.

[35] open source network intrusion prevention and detection system (ids/ips). http://www.snort.org/, Sept. 2010.

[36] F. Vaskovich. Nmap stealth port scanner. http://www.insecure.org/nmap/index.html, 2002.

[37] A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In Rocha and Launchbury [33], pages 235–249.

[38] Y. Yiakoumis and N. McKeown. Slicing Home Networks. In *ACM SIGCOMM Workshop on Home Networking (Homenets)*, Toronto, Ontario, Canada, May 2011.

[39] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *Proc. ACM SIGCOMM*, August/September 2010.