Don't Configure the Network, Program It!
Domain-Specific Programming Languages for Network Systems

Andreas Voellmy[1] Ashish Agarwal[1] Paul Hudak[1] Nick Feamster[2]
Sam Burnett[2] John Launchbury[3]

July 10, 2010

[1] Department of Computer Science, Yale University, New Haven, CT 06520, USA
[2] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA
[3] Galois Inc., Portland, OR 97204, USA

Research Report YALEU/DCS/RR-1432[1]

Abstract

Network operators must configure networks to accomplish critical, complex, and often conflicting requirements: they must ensure good performance while maintaining security, and satisfy contractual obligations while ensuring profitable use of interdomain connections. Unfortunately, today they have no choice but to implement these high-level goals by configuring hundreds of individual network devices. These interact in complex and unexpected ways, often resulting in misconfigurations or downtime. We propose a new approach: rather than configure individual network devices, operators should *program* the network holistically, according to high-level policies.

Towards this goal, we present Nettle, a system for clearly and concisely expressing network requirements, together with mechanisms to control the network accordingly. At the lowest level, we rely on OpenFlow switches for programmable network hardware. On top of this layer, we build an extensible family of embedded domain-specific languages (EDSLs), each aimed at different operational concerns, and provide convenient ways to sensibly combine expressions in these languages. We present a case study demonstrating a DSL for networks that provides fine-grained, dynamic access control policies.

[1]A version of this report was submitted to the Ninth ACM Workshop on Hot Topics in Networks (HotNets-IX).

1 Introduction

The behavior of a communications network depends on the configuration of hundreds to thousands of switches, routers, firewalls, and other devices. For example, a campus network may have as many as 2,000 inter-operating network devices and about one million lines of configuration; whether the network operates correctly, and according to the network operator's policy, depends for the most part on the configuration of these devices. Operators configure these devices to perform complex tasks ranging from provisioning and access control to rate limiting and load balancing by independently adjusting the network configurations of these devices. The complexity of these tasks and the low-level at which operators interact with network devices make network configuration time-consuming, and easily one of the most significant costs of running a network today.

Despite its importance, network configuration remains primitive and error prone [4, 9]. According to a recent study, human error accounts for as much as 80% of outages [8]. Even when an operator finally manages to configure a collection of devices to achieve some high-level task or implement some policy, the configuration itself remains extremely brittle: because the configuration rests on the devices themselves and also depends on various low-level details (e.g., where the device is located in the network topology, software versions or vendor models of switches), a small change in configuration can result in an overall network configuration that fails to achieve the desired policy and is difficult to fix. Much previous work has attempted to help network operators detect configuration errors through static configuration analysis (*e.g.*, [4, 5]) or network-wide simulation (*e.g.*, [11, 12, 17]). Unfortunately, these approaches do not make it easier for network operators to configure network policies in the first place, nor do they guarantee that the network's behavior is correct.

The configurations of hundreds of individual network devices collectively determine the behavior of the network. In other words, network configuration is effectively a large distributed program, with today's network configuration languages effectively operating at the level of assembly language. This observation leads us to the following question, which we explore in this report: *Can a communications network be "programmed" with a high-level programming language?* We believe the answer is yes; in this report, we argue that advanced, high-level programming languages and tools allow one to express the overall network behavior as a single program expressed in a declarative style.

Of course, previous research has suggested that a high-level language for configuring networks could eradicate many configuration errors and problems in today's networks [2,4,9]; despite widespread agreement on the need for such a language, however, a solution as not materialized. Fortunately, there is a recent emergence of network switches that expose a unified, flexible, dynamic, remotely programmable interface that allow network switches to be controlled from a logically centralized location (*e.g.*, OpenFlow [1]). We leverage this interface to help us incorporate advanced programming language ideas to ensure that our programming model is expressive, natural, concise, and designed precisely for networking applications. In particular, we borrow ideas from functional reactive programming and adopt the design methodology of domain-specific language (DSL) research.

Our framework, which we call *Nettle*, radically refactors network configuration. Rather than configure individual network devices, operators can now *program* the network as a whole. This subtle shift offers significant benefits. First, operators can easily define complex network policies and behaviors, such as more complicated business relationships, traffic load balance goals, and security policies. They can also define the

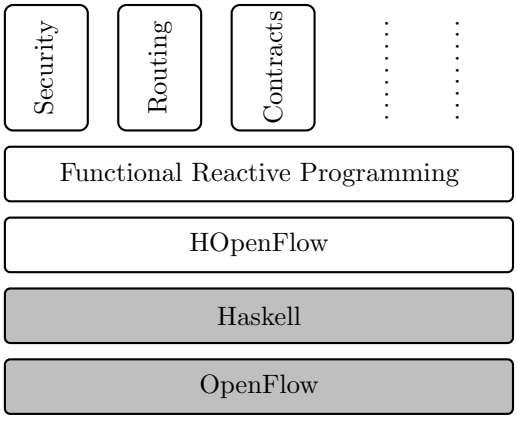

Figure 1: Nettle's layered design.

network behavior such that it automatically adjusts to changing network conditions such as traffic surges and failures. With this perspective, we provide a more direct approach to specifying and implementing operator intentions than is provided by static, distributed configurations.

This report presents the case for programming networks in a high-level language and illustrates our approach by applying this framework to an example task in an enterprise network. We describe the Nettle approach in more detail and demonstrate our ideas through a DSL in the context of fine-grained, dynamic access control for an enterprise network. The Nettle programming framework is broad and can apply to many network operations tasks, ranging from specifying security policies or performance requirements to business relationships. We explore how Nettle may apply to these settings.

2 Approach

This section describes Nettle, our framework for programming networks. We present an overview first and then discuss two components—domain specific languages and functional reactive programming—in detail.

2.1 Overview

Figure 1 illustrates our layered architecture. At the bottom are OpenFlow-enabled switches, which make programmable networks possible. One level up is Haskell, the language we have chosen to implement Nettle in, for reasons explained shortly. Above that is the first layer we provide, HOpenFlow, a library for constructing OpenFlow messages and marshalling between the required binary protocol.

The next layer in our stack is an instantiation of a language in the Functional Reactive Programming (FRP) paradigm. FRP is a family of languages that provide an expressive and mathematically sound approach to programming real-time interactive systems in a declarative manner. It enables us to treat the network as a dynamical system and network management as a feedback controller. Specifically, we define FRP-based controllers that take a stream of OpenFlow messages as input and generate a stream of OpenFlow messages as output.

The FRP layer is very flexible, but it does not contain any of the terms specific to networking. Nettle's highest-level consists of a family of domain specific languages (DSLs) that operators can use to encode constraints that relate to specific networking tasks (*e.g.*, security, performance). These DSLs are implemented in terms of the FRP and OpenFlow constructs of the lower layers.

2.2 Domain Specific Languages

Because there are many types of computer networks, it would be a mistake to try designing a single one-size-fits-all language. Our approach is to instead design an *extensible family of DSLs*, each capturing an important network abstraction. For example, we may have one DSL for access control policies, another for traffic engineering strategies, and another for expressing interdomain contracts. Because the family is extensible, operators can easily add new abstractions and functions.

To avoid creating small, isolated DSLs, we rely on the technique of *embedding* each DSL into a host language. We choose Haskell [15] as our host because of its remarkable flexibility in supporting embedded DSLs [6]. This approach allows our DSLs to share a common "look and feel" through the adoption of the same language infrastructure, such as variable naming conventions, function definitions, primitive data types, a powerful type system, and so on. This not only relieves us from the burden of implementing these general features, allowing us to focus on domain-specific concepts, but more importantly allows the DSLs to interoperate with one another.

2.3 Functional Reactive Programming

FRP-based languages have been used successfully in computer animation, robotics, control systems, GUIs, interactive multimedia, and other areas in which there is a combination of both continuous and discrete entities [3,13,14,16]. We now briefly introduce the key ideas of functional reactive programming (FRP) [7]. The simplest way to understand FRP is to think of it as a language for expressing electrical circuits. We refer to the wires in a typical circuit diagram as *signals*, and the boxes (that convert one signal into another) as *signal functions*. For example, this very simple circuit has two signals, x and y, and one signal function, $sigfun$:

This is written as a code fragment in FRP simply as:

$y \leftarrow sigfun \prec x$

The wires have values on them continuously so really x and y are functions of time. Unlike normal circuit diagrams, FRP seamlessly also handles streams of discrete events. For instance, x could be a stream of host-join events, and $sigfun$ could output an event that modifies a flow table on each such input.

FRP has many built-in signal functions, including all of the obvious numeric functions, as well as ones for integration and differentiation of signals. Of course one can also define new signal functions. For example, here is a definion for $sigfun$ above that simply returns a signal that always takes the sine of one greater than its input:

$$
\begin{aligned}
&sigfun :: SF\ Float\ Float \\
&sigfun = \textbf{proc}\ x \rightarrow \textbf{do} \\
&\quad y \leftarrow sin \prec\!\!- x + 1 \\
&\quad returnA \prec\!\!- y
\end{aligned}
$$

The first line in this program is a type signature that declares that $sigfun$ is a signal function that converts continuous values of type $Float$ to continuous values of type $Float$.

We can use signals and signal functions to program controllers that alter traffic flow based on signals that measure the traffic volume on particular links. In this report, however, we emphasize a different use: we will use signals to represent *streams of messages* flowing to and from the networks OpenFlow switches; each signal (*i.e.*, wire) is effectively a stream of messages.

In FRP, signal processing is declarative (as opposed to callback-based): the programmer thinks of, and programs with, message streams as a whole. For example, the merger of two message streams $ms1$ and $ms2$ is simply $ms1 .|. ms2$. A message, of course, carries data, and sometimes we need to manipulate the data in each message of a message stream. We can apply a function fn to each message in a message stream ms with the expression $ms \ggg fn$. Sometimes our chore is even simpler: we may want to simply replace each message with a different one, say m, which can be written $ms \gg\!- m$. We will use both of these operators in later examples.

3 Fine-grained Dynamic Access Control Language

We demonstrate our ideas by presenting an overview of a domain-specific language for fine-grained dynamic access control, intended to securely control enterprise networks. Our language and implementation are based directly on a prototype system Resonance [10], and are intended to generalize this prototype to apply to a whole class of similar systems. Before presenting our DSL, we briefly review the Resonance system.

3.1 Enterprise Access Control with Resonance

Resonance is an OpenFlow-based system, aimed at campus networks, in which the network infrastructure is actively involved in enforcing the security policies of the system, removing responsibility from end hosts or higher network layers for the enforcement of security policies. Resonance is designed to apply policies dynamically, incorporating input from network monitoring devices, such as virus scanners and intrusion detection systems, and actively and dynamically control the network equipment in response. For example, a Resonance system might quarantine a host when a compromise or other security breach is detected.

A key notion of Resonance is to implement *fine-grained, dynamic access control*. This means that a Resonance system allows or prevents users from gaining access to the network, and that it provides operators fine-grained control over this functionality. The control is fine-grained in two ways: (1) users can be given access to different resources on the network independently and (2) that these priveleges can be different for different users. It is dynamic in the sense that a user's priveleges may change over time. Dynamic access control is critical to security in order to, for example, quarantine a normally trusted user once he is known to have become infected with a virus.

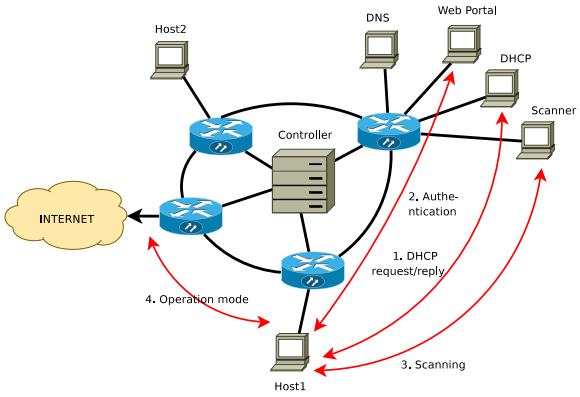

Figure 2: System architecture for example system

3.2 Resonance in Nettle

In this section, we present an EDSL for describing fine-grained, dynamic access control policies, and show how these can be used to control a network. This EDSL demonstrates many of the ideas of our approach; (1) we can successfully tailor our host language to our domain and allow users to clearly describe their high-level ideas, (2) it relies critically on Haskell, our host language, to provide general features, such as variable definitions, functions and datatypes, and (3) FRP plays a crucial role in in providing a declarative approach to describing the reactive component of a Resonance system.

We will take a top-down approach, and demonstrate how we use the constructs of the EDSL to describe and implement a secure Enterprise network, that is very closely related to the one given in Ankur et al [10]. In the process, we will introduce—by example—the underlying components of the EDSL. Of course, we will omit many details. The intent is to provide a flavor of the approach, and perhaps convey a sense of the ability to describe a wide variety of fine-grained dynamic access controlled network controllers.

As shown in Figure 2, the network includes several basic components, such as DHCP and DNS servers. A web portal provides a web-based authentication server. Upon successful authentication, the web portal sends a message to the controller via its Messenger interface. A vulnerability scanner monitors hosts for know vulnerabilities and contacts the controller, again via its Messenger interface about the status of host scans.

The security model for the infrastructure devices on the network, such as the web portal and dhcp server is very simple. For brevity, we focus here on the most interesting part of the policy, namely the end host configuration.

3.2.1 End Host Policy

To begin with we define a state space—which is simply a pair of booleans modeling whether the host has been authenticated and whether it is not deemed to have vulnerabilities—and the event set—which captures information about the hosts:

```
data HostState
  = HostState { authenticated,
                notVulnerable :: Bool }
data HostEvent
  = Authenticated
  | ScanOK
  | ScanFailed
```

We then define a state machine for an end host. Initially our host should not be authenticated and should be considered vulnerable, and our transition function maps is entirely straightforward mapping of events to state modifiers:

```
initialState
  = HostState { authenticated = False,
                notVulnerable = False }
nextState Authenticated hs
  = hs { authenticated = True }
nextState ScanOK        hs
  = hs { notVulnerable = True }
nextState ScanFailed    hs
  = hs { notVulnerable = False }
```

All this is plain Haskell so far. Now we use Nettle and define the event signal for hosts:

```
eventSignal ethAddr
  = proc hi -> do
      authE <- authEventSF ethAddr  -< hi
      ae    <- scanAcceptSF ethAddr -< hi
      re    <- scanRejectSF ethAddr -< hi
      returnA -< (authE >>> Authenticated .|.
                  ae    >>> ScanOK        .|.
                  re    >>> ScanFailed)
```

This simple signal function converts the various messages received from the various devices in the network into the events in our state machine model of a host. *need to explain what is happening to the return values*. The function makes use of signal functions $authEventSF$, $scanAcceptSF$ and $scanRejectSF$. The first of these describes whether an authorization has been received for a particular host. The latter two are event sources for successful and unsuccessful vulnerability scans for a particular host.

We also have to define the dynamic security policy for our end hosts:

```
dynSecurityPolicy state pktIn
  | auth ∧ notVuln state
```

$$
\begin{aligned}
&\quad = allow \\
&\mid auth \wedge pktIn \vdash arp \vee dns \vee dhcp \\
&\quad = flood \\
&\mid \neg\ auth \wedge pktIn \vdash arp \vee dns \vee dhcp \\
&\quad = flood \\
&\mid \neg\ auth \wedge pktIn \vdash http \\
&\quad = forwardTo\ webPortalMacAddr \\
&\mid otherwise \\
&\quad = deny \\
&\quad \textbf{where}\ auth \quad = authenticated\ state \\
&\qquad\quad notVuln = notVulnerable\ state
\end{aligned}
$$

The dynamic security policy uses predicates for packet types to distinguish between DNS packets and DHCP packets, for example.

We have a function $secureHost$ which builds a host controller from a state machine, a signal containing relevant events, and an appropriate dynamic security property. Thus, we can specify a host controller for an end host by defining:

$$
\begin{aligned}
&endHost\ ethAddr = \\
&\quad secureHost \\
&\qquad (initialState, nextState) \\
&\qquad (eventSignal\ ethAddr) \\
&\qquad dynSecurityPolicy
\end{aligned}
$$

3.2.2 Dynamic Security Policy

Let's look at the definition of the dynamic security policy in more detail. As the Resonance paper states, "a policy effectively dictates what actions a switch should take on traffic to and from a host that is of a particular security class and state." We therefore, represent a dynamic security policy as a mapping from some state space to a static security policy:

$$
\begin{aligned}
&\textbf{type}\ DynamicSecurityPolicy\ state \\
&\quad = state \rightarrow SecurityPolicy
\end{aligned}
$$

Values of type $DynamicSecurityPolicy\ state$ define, at each host state, the security policy applied while the host is in that state.

There may be many choices for how to represent security policies; here we choose a simple representation, as follows:

$$\begin{aligned}&\textbf{type } SecurityPolicy\\&\quad = PacketIn \rightarrow ForwardingDecision\end{aligned}$$

The semantics of the language implementation are that for each packet sent by a host, the security policy for that host at that time - based on the state of the host then - is applied to determine how that packet will be treated.

Note that this allows for non-symmetrical security policies, where one host can send packets to another host, but not vice versa. One could argue that this should be disallowed, but we have chosen not to do so for this example.

We provide four forms of forwarding decisions:

$$\begin{aligned}&deny, allow, flood :: ForwardingDecision\\&forwardTo \quad :: EthernetAddress \rightarrow\\&\qquad\qquad ForwardingDecision\end{aligned}$$

The semantics are going to be specified as follows: for a $packetIn$ value, $deny$ drops the packet, $allow$ forwards the packet towards the destination as specified in the $packetIn$ value, $flood$ floods the packet, and $forwardTo\ ethAddr$ forwards the packet towards the host $ethAddr$. For both $allow$ and $forwardTo$ decisions, the forwarding path is determined by the routing algorithm in use by our language implementation.

3.2.3 Executing the Nettle Controller

At the highest-level, a Resonance system is specified by associating a $HostController$ with each host, i.e. by a mapping from hosts to $HostController$s. In the current system, we identify hosts with the Ethernet addresses of their device, and thus, we define:

$$\begin{aligned}&\textbf{type } NetworkController\\&\quad = EthernetAddress \rightarrow HostController\end{aligned}$$

This representation allows for different hosts to be controlled by different $HostController$s, allowing, for example, a different security policy to be applied to a trusted dhcp server than to an untrusted end host.

We provide a function for running a network controller Resonance system:

$$\begin{aligned}&runNetwork ::\\&\quad ControlParameters \rightarrow\\&\quad NetworkController \rightarrow\\&\quad IO\ ()\end{aligned}$$

The function $runNetwork\ params\ ncontroller$ does the following:

- Start an OpenFlow controller, connecting to switches;
- React to packet-in events from OpenFlow switches, turning on host controllers as hosts become active in the network, and controlling switches to satisfy host control policies;

- Start a TCP server running a simple text-based protocol that can be used by components on the network, such as an authentication server, to communicate with the controller.

The $ControlParameters$ argument includes various configurable parameters which we omit here.

4 Conclusion

Communication networks have become increasingly large and complex, and serve an expanding range of purposes. At the same time, they have become more difficut to configure, manage, and troubleshoot. Despite the fact that network configuration largely dictates the network behavior—and hence, its correctness—network configuration languages remain disconcertingly low-level.

Drawing on the observation that a network configuration essentially specifies the behavior of a large distributed program, we propose developing technology to *program* networks using higher-level programming languages that are more tailored to the tasks that network operators are trying to perform.

We propose to develop a family of embedded domain-specific languages, called *Nettle*. In Nettle, each DSL captures a network abstraction, such as performance, traffic load balance, security, and business relationships. The underlying substrate, based on functional reactive programming, provides a flexible and powerful language for programming real-time, interactive systems and provides an appropriate base language for the development of higher-level DSLs.

This report has shown an example of describing a network behavior in a DSL in the Nettle family, namely one for fine-grained, dynamic access control.

We are still in the early days with the design of Nettle and are discovering the right abstractions for programming networks—both data and control abstractions. Our approach of embedding languages within Haskell provides an excellent setting for lightweight experimentation with languages, and enables our exploration.

From the perspective of network configuration, this report has merely scratched the surface for what may be possible in this new programming paradigm for networks. We envision that Nettle will allow operators to configure networks to perform a much wider variety of tasks. In addition to an enterprise deployment of Nettle on which we are testing the suitability of the access control DSL we have outlined in this report, we are working on deploying Nettle across GENI's multi-campus OpenFlow testbed to test its suitability for expressing interdomain routing policies. We hope Nettle may ultimately serve as the foundation for expressing a variety of network configuration policies and tasks.

Finally, we are also in the early phases of discovering where the performance bottlenecks are. Clearly, a key factor affecting performance in an OpenFlow system will be the amount of packets transmitted to the controller for processing. In effect, Nettle has a two-level execution/compilation strategy, where some decisions are cached at the switch, and others are made in the controller system. Ultimately, we may need to compile Nettle into a three-layer target, where we introduce a pro- grammable control component to sit right next to the switch, and automatically compile and distribute Nettle code to the right layer of the control hierarchy.

5 Acknowledgements

This research was supported in part by an STTR grant from the Defense Advanced Research Projects Agency. We wish to thank our STTR industrial partner, Galois, Inc. for its support as well. Vijay Ramachandran motivated our initial foray into language design for networking.

References

[1] `http://www.openflowswitch.org/`.

[2] M. Caesar and J. Rexford. BGP routing policies in ISP networks. *Network, IEEE*, 19(6):5 – 11, nov.-dec. 2005.

[3] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273, June 1997.

[4] 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.

[5] N. Feamster and J. Rexford. Network-Wide Prediction of BGP Routes. pages 253–266, Apr. 2007.

[6] P. Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), Dec. 1996.

[7] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Robots, arrows, and functional reactive programming. In *Summer School on Advanced Functional Programming, Oxford University*. Springer Verlag, LNCS 2638, 2003.

[8] What's Behind Network Downtime? `www.juniper.net/solutions/literature/white_papers/200249.pdf`, 2008. Juniper White Paper.

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

[10] 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.

[11] Opnet NetDoctor. `http://opnet.com/products/modules/netdoctor.htm`.

[12] Opnet Modeler. `http://opnet.com/products/modeler/opnet_modeler.pdf`, 2003.

[13] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, 1999.

[14] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *First International Workshop on Practical Aspects of Declarative Languages*. SIGPLAN, Jan 1999.

[15] S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[16] A. Reid, J. Peterson, G. Hager, and P. Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *Proc. Int'l Conference on Software Engineering*, May 1999.

[17] WANDL IPAT. `http://wandl.com/html/ipat/IPAT_new.cfm`, 2003.