Nettle: Taking the Sting Out of Programming Network Routers

Andreas Voellmy and Paul Hudak

Yale University
andreas.voellmy@yale.edu, paul.hudak@yale.edu

Abstract. We describe a language-centric approach to solving the complex, low-level, and error-prone problem of *network control*. Specifically, we have designed a domain-specific language called *Nettle*, embedded in Haskell, that allows programming *OpenFlow* networks in an elegant, declarative style. Nettle is based on the principles of *functional reactive programming* (FRP), and as such has both continuous and discrete abstractions, each of which is leveraged in the design. We have implemented Nettle and tested it on real OpenFlow switches. We demonstrate our methodology by writing several non-trivial OpenFlow controllers.

1 Introduction

Networks continue to increase in importance and complexity, yet the means to configure them remain primitive and error prone. There is no precise language for describing what a network should do, nor how it should behave. At best, network operators document their complex requirements informally, but then are faced with the daunting and unreliable task of translating their specifications by hand into the low-level, device-specific, often arcane scripts used to control today's commercial switches and routers. This low-level programming model often results in devices and protocols interacting in unexpected ways [6], and gives little hope in validating high-level protocols and policies such as traffic engineering, business relationships, and security policies [14,3].

We believe that these problems can be overcome through the use of advanced high-level programming languages and tools that allow one to express overall network behavior as a single program expressed in a declarative style. Although this idea has been suggested by several researchers [3,11], the development of an actual solution has been elusive. There are two aspects of our approach that we believe will result in a successful outcome: First, we abandon conventional switches in favor of flexible, dynamically adaptable *programmable switches.* In particular, we have focused our efforts on *OpenFlow switches* [1], which present a flexible, dynamic, remotely programmable interface that allows them to be controlled from a logically centralized location.

Second, we use advanced programming language ideas to ensure that our programming model is expressive, natural, concise, and designed precisely for networking applications. Specifically, we borrow ideas from *functional reactive*

R. Rocha and J. Launchbury (Eds.): PADL 2011, LNCS 6539, pp. 235–249, 2011.

programming (FRP) and adopt the design methodology of *domain-specific language* (DSL) research.

Our overall approach, which we call *Nettle*, allows us to radically rethink the problem of network configuration. Indeed, we like the mantra, "Don't *configure* the network, *program it*!" [15]. In doing this at a high level, we enable the development of new, powerful, and natural network policies, protocols, and control algorithms.

2 Overall Approach

In this paper, we focus on the problem of configuring a network of OpenFlow switches, varying in size from a single router to several hundred. Such a network may belong to a commercial entity, an Internet service provider (ISP), a university, etc. Typically, certain border routers of such a network interface to the Internet, but our focus is on the internal interactions and coordination between local switches. Unlike most conventional networks, all of the OpenFlow switches communicate with a centralized *controller*. It is here that a Nettle program runs, thus implementing a global control policy for the entire local network. Although a centralized controller will ultimately present problems as the network is scaled upward in size, it is adequate to handle most moderately-sized networks.

Figure 1 illustrates our software architecture. At the bottom are OpenFlow switches themselves. One level up is Haskell, our host language. Above that is a library, Nettle/OpenFlow, that abstractly captures the OpenFlow protocol.

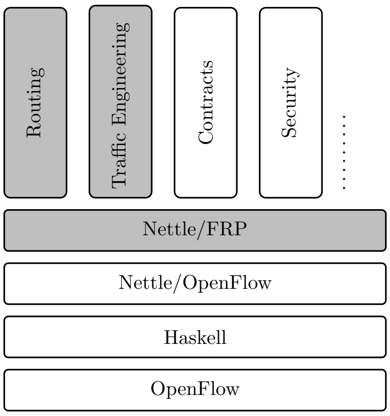


Fig. 1. Nettle layered system architecture

The next layer in our stack is an instantiation of the Functional Reactive Programming (FRP) paradigm. FRP is a family of languages that provide an expressive and mathematically sound approach to programming reactive systems in a declarative manner. FRP has 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 [13,4].

Above the FRP layer, we plan to implement an extensible family of DSLs, each member capturing a different network abstraction. For example, we may have one DSL for access control, another for traffic engineering, and another for interdomain contracts. As a concrete example, in [15] we describe a DSL for expressing a class of dynamic security policies for campus networks and its implementation on Nettle's FRP layer.

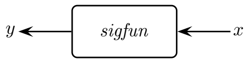
In this paper we focus on the Nettle components that are shaded in Figure 1. Our contributions, as outlined in this paper, include:

1. A *core* Nettle/FRP language that supports the development of higher-level control languages by providing two key abstractions:
   (a) A discrete, event-based abstraction that declaratively captures communication patterns to and from OpenFlow switches.
   (b) A notion of continous, time-varying quantities that capture higher-level abstractions such as traffic volume on individual network links.
2. A simple, declarative approach to dynamic routing, based on Nettle/FRP's discrete communication abstraction.
3. A simple, declarative approach to traffic engineering, based on Nettle/FRP's abstraction of continuous quantities.
4. An implementation of Nettle/FRP in the context of the software architecture of Figure 1. We have tested our system on reference implementations of the OpenFlow switches, as well as on real OpenFlow switches.

3 Functional Reactive Programming

In this section we briefly introduce the key ideas and constructs of Nettle/FRP, whose design is strongly influenced by *Yampa* [8], an FRP-based DSL that we previously designed for robotics and animation.

The simplest way to understand Nettle/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 Nettle simply as: $y \leftarrow \mathit{sigfun} \prec x$, which uses Haskell's *arrow syntax* [9,12]. It is beyond the scope of this paper to describe arrows in detail, but note that signal functions such as *sigfun* will have a type of the form $SF\ T_1\ T_2$, for some types T_1 and T_2, in which case x will have type T_1, and y will have type T_2. Although signal functions act on signals, the arrow notation allows one to manipulate the instantaneous values of the signals, such as x and y.

Nettle/FRP has many built-in signal functions, including ones for integration and differentiation. Of course one can also define new signal functions. For example, here is a definition for *sigfun* that integrates a signal that is one greater than its input:

```
sigfun :: SF Double Double
sigfun = proc x → do
  y ← integral −≺ x + 1
  returnA −≺ y
```

The first line is a type signature that declares *sigfun* to be a signal function that converts time-varying values of type *Double* into time-varying values of type *Double*. The notation **proc** x → **do**... introduces a signal function, giving the name x for the instantaneous values of the input. The third line adds one to this instantaneous value, and sends the resulting signal to the integrator. Finally, we specify the output of the signal function by feeding y into *returnA*, a special signal function that returns the final result.

We can also create and use signal functions that operate on tuples of signals. For example, a signal function *exp* :: *SF* (*Double*, *Double*) *Double* that raises its first argument to the power of its second, at every point in time, could be used as follows:

```
z ← exp −≺ (x, y)
```

In Section 7 we will see how continuous signals can be used to program controllers that alter traffic flow based on signals representing message volume on a link. However, we first wish to focus on a different use of signals, namely to represent *streams of control messages* flowing to and from OpenFlow switches. Nettle/FRP represents message streams as continuous signals that are only defined at discrete points in time. A discrete signal that periodically carries information of some type T has type *Event* T, whose values are either *NoEvent* or *Event* T (the word *Event* is overloaded). Therefore, for example, a signal function that converts a stream carrying messages of type M_1 into a stream carrying messages of type M_2 has type *SF* (*Event* M_1) (*Event* M_2).

In a conventional language, an event-based system might be implemented by some kind of call-back mechanism and a loop that handles messages as they arise, one by one. But in Nettle, it is done much more declaratively, where we think of, and program with, message streams as a whole.

Nettle/FRP provides a powerful collection of signal functions and event operators, most of which we introduce as we encounter them in this paper. For reference, they are summarized in Figure 2.

4 FRP for OpenFlow Control

OpenFlow switches maintain a *flow table* containing *flow entries* consisting of a *match* condition, a list of *forwarding action*s, expiration settings, and flow statistics. The match condition can optionally match on most Ethernet, IP, or transport protocol header fields. The forwarding actions include forwarding to

```
never          :: SF a (Event b)
hold           :: a → SF (Event a) a
accum          :: a → SF (Event (a → a)) (Event a)
integral       :: SF Double Double
tag            :: Event a → b → Event b
liftE          :: (a → b) → Event a → Event b
mapFilterE     :: (a → Maybe b) → Event a → Event b
mergeEventsBy  :: (a → a → a) → [Event a] → Event a
```

Fig. 2. Signal function and event combinators

specific ports on the switch, flooding the packet, dropping the packet and many other options. When a packet is received by a switch, it searches for a matching entry. If matches are found, the highest priority one is chosen, its forwarding actions are executed and its statistics are updated. If the list of actions is empty, the packet is dropped. If no match is found, the packet is encapsulated and sent to the controller in an OpenFlow message. Optional expiration settings cause a flow entry to expire after some prescribed time.

OpenFlow switches attempt to establish a TCP connection with a controller at a pre-configured IP address. This connection typically takes place over a control network that is separate from the main data network. The OpenFlow protocol defines a variety of messages, including messages to query a switch for information, to command a switch to send a packet and to modify the flow table of a switch. Other messages allow a switch to inform the controller of relevant events, such as the arrival of a packet not matching any flow rule.

We can think of a controller abstractly as a black box which transforms a stream of messages from switches into a stream of commands for switches. We therefore define a Nettle controller as any value having the type:

```
SF (Event (SwitchID, SwitchMessage)) (Event SwitchCommand)
```

Here we model messages from switches with the *SwitchMessage* data type and commands to switches with the *SwitchCommand* data type. The *SwitchMessage* data type is a sum of several message types. Typically a user is interested in a particular kind of event, and we provide projection functions for each variant in the *SwitchMessage* sum type. For example, the function *switchJoinE* extracts just the messages that occur when a switch connects with the controller. This particular message carries information about the joining switch in the form of a *SwitchFeatures* record. Figure 3 lists the particular projections we will use in this paper. We will explain their meaning as we encounter them in the examples.

*SwitchCommand*s are commands that send packets, modify flow tables, and request information. Among the most important of these is *insertRule*, which can

```
switchJoinE :: Event (SwitchID, SwitchMessage) → Event (SwitchID, SwitchFeatures)
packetInE   :: Event (SwitchID, SwitchMessage) → Event (SwitchID, PacketIn)
```

Fig. 3. Nettle event projections used in this paper

be used to insert a *FlowRule* into a switch's flow table. We write a *FlowRule* as $predicate \Longrightarrow actions$, where *predicate* has type *PacketPredicate* and *actions* has type *ForwardingAction*. For example, the following is a flow rule that forwards packets with source Ethernet address *addr* to port *port*:

$$ethSourceIs\ addr \Longrightarrow sendOnPort\ port$$

Commands can be combined to create compound commands with the command sequencing operator $\oplus$. Figure 4 summarizes the commands used in this paper. We will explain their meaning as we encounter them in the examples.

$clearTable$	$:: SwitchID \to SwitchCommand$
$sendPacketIn$	$:: ForwardingAction \to (SwitchID, PacketIn) \to SwitchCommand$
$insertRule$	$:: FlowRule \to SwitchID \to SwitchCommand$
$deleteRules$	$:: PacketPredicate \to SwitchID \to SwitchCommand$
$(\oplus)$	$:: SwitchCommand \to SwitchCommand \to SwitchCommand$
$(\Longrightarrow)$	$:: PacketPredicate \to ForwardingAction \to FlowRule$

Fig. 4. Nettle commands used in this paper

4.1 Basic Event Handling and Switch Commands

The simplest possible controller is one that does nothing at all:

$$\begin{array}{l}
controller_0 = \mathbf{proc}\ msgE \to \mathbf{do} \\
\quad cmdE \leftarrow never \prec msgE \\
\quad returnA \prec cmdE
\end{array}$$

We use the *never* signal function which never outputs any events.

It is a good idea to clear the flow table of every switch as soon as it connects with the controller, so that our switches start in a known state. We can do this by executing a *clearTable* command whenever a *SwitchJoin* event occurs:

$$\begin{array}{l}
clearOnJoin = \mathbf{proc}\ msgE \to \mathbf{do} \\
\quad returnA \prec liftE\ f\ (switchJoinE\ msgE) \\
\quad \mathbf{where}\ f\ (sid, \_) = clearTable\ sid
\end{array}$$

Here we use *switchJoinE* to extract the switch join events from the input message stream. For each such event, we apply the function f to the event, which in turn applies *clearTable* to the *SwitchID* of the joining switch, giving a command that will delete all entries from the flow table of the joining switch. In order to apply a function to each event in an event stream, we use $liftE :: (a \to b) \to Event\ a \to Event\ b$.

Having cleared the table of all connected switches, the switches will send any incoming packets to the controller. In a network that doesn't contain any cycles among its switches, it is safe to simply flood packets, and we can accomplish this in Nettle by writing:

$$\begin{array}{l}
floodPackets_1 = \mathbf{proc}\ msgE \to \mathbf{do} \\
\quad returnA \prec liftE\ f\ (packetInE\ msgE) \\
\quad \mathbf{where}\ f = sendPacketIn\ flood
\end{array}$$

Here we use *packetInE* to extract only the *PacketIn* messages from the incoming message stream. For each such event, we apply *sendPacketIn flood*, instructing the switch to send the referenced packet using the action *flood* (of type *ForwardingAction*), which results in the switch forwarding the packet on every port except the incoming port (i.e. the port on which the packet was received).

We can now create a single controller that combines both the table clearing and packet flooding controllers, as follows:

$$\begin{array}{l}
\mathit{controller}_1 = \mathbf{proc}\ \mathit{msgE} \rightarrow \mathbf{do} \\
\quad \mathit{clearE} \leftarrow \mathit{clearOnJoin} \ \prec \mathit{msgE} \\
\quad \mathit{floodE} \leftarrow \mathit{floodPackets}_1 \prec \mathit{msgE} \\
\quad \mathit{returnA} \prec \mathit{mergeEventsBy}\ (\oplus)\ [\mathit{clearE}, \mathit{floodE}]
\end{array}$$

In this signal function we feed the incoming message stream to *both* signal functions, naming events in the resulting message streams *clearE* and *floodE*. We then merge these two command streams, resolving the simultaneous occurrence of commands with ⊕, and output the merged command stream.

4.2 Programming the Flow Table

In the previous controller, the switches sent a *PacketIn* message to the controller for *every* incoming packet, and the controller responded with an explicit command for the switch to flood the packet. We can dramatically improve the performance of the network by installing a flow rule at the switch to flood every packet, thereby avoiding the need for the switch to communicate with the controller for every packet and taking advantage of specialized packet forwarding hardware at the switch. We install the flow rule, whenever a switch joins the network:

$$\begin{array}{l}
\mathit{floodPackets}_2 = \mathbf{proc}\ \mathit{msgE} \rightarrow \mathbf{do} \\
\quad \mathit{returnA} \prec \mathit{liftE}\ f\ (\mathit{switchJoinE}\ \mathit{msgE}) \\
\quad \mathbf{where}\ f\ (\mathit{sid}, \_) = \mathit{insertRule}\ (\mathit{anyPacket} \Longrightarrow \mathit{flood})\ \mathit{sid}
\end{array}$$

insertRule rule sid is a command that installs rule *rule* on switch *sid* and *anyPacket* is a packet predicate that matches every packet. Again, we can combine this in parallel with *clearOnJoin* to form a complete controller:

$$\begin{array}{l}
\mathit{controller}_2 = \mathbf{proc}\ \mathit{msgE} \rightarrow \mathbf{do} \\
\quad \mathit{clearE} \qquad\ \leftarrow \mathit{clearOnJoin} \ \prec \mathit{msgE} \\
\quad \mathit{tableModE} \leftarrow \mathit{floodPackets}_2 \prec \mathit{msgE} \\
\quad \mathit{returnA} \prec \mathit{mergeEventsBy}\ (\oplus)\ [\mathit{clearE}, \mathit{tableModE}]
\end{array}$$

5 Learning Switch

In this section, we will program a so-called *learning switch*. Traditionally, a learning switch is an Ethernet switch which initially acts much like an Ethernet hub, flooding frames received on one port to all other ports. However, a learning switch also maintains a table of Ethernet addresses and ports, such that if (a, p) is in the table, then p is the port at which the switch most recently received a frame *from* the host with address a. Since the switch received a packet *from a* on port p, port p must be on the path *to a* (assuming our network is loop-free). Consequently, when a switch receives a frame addressed to a, it forwards the frame on port p if (a, p) is in its table at that time, or else floods it on all ports other than the incoming one. In addition, a learning switch typically expires entries in the flow table after some period of inactivity.

As a first step to building our learning switch controller, we will program a component which performs the "learning" part; that is, it builds the table described above for each switch, inferring the direction of each host from every switch in the network. We implement this table using the *Map* data type from Haskell's standard library, which implements maps from keys to values (dictionaries). We will use that data type's *insert* function to add or update the value associated with a key. We will build the table by transforming each packet-in event into a table update and accumulating these updates with *accum*:

```
nextHopsSF = proc msgE → do
  hostMapE ← accum empty −≺ liftE updateMap (packetInE msgE)
  returnA −≺ hostMapE
```

accum empy takes as input an event stream carrying state-modifying functions. At each event in its input stream, it applies the state-modifying function carried by the event to the current state, updates the current state with that new value, and outputs an event carrying the updated value. As a result, the output signal will start out as the empty map, and will output an updated map whenever a packet in event occurs. The function *updateMap* is straightforward: it updates the table for key (*sid*, *addr*) to be the port ID of the port on which the packet was received:

```
updateMap (sid, PacketIn {receivedOnPort, enclosedFrame}) =
  insert (sid, sourceAddress enclosedFrame) receivedOnPort
```

We can now use *nextHopsSF* to program our controller. The controller will monitor the packet in events, and for each such event, if it has learned the direction the packet should travel, it will install appropriate flow rules at the switches to forward similar packets in the learned direction. If it has not learned the direction the packet should travel, it will simply flood the packet on all ports, without installing flow rules in any switches:

```
controller3 = proc msgE → do
  nextHopsE ← nextHopsSF   −≺ msgE
  nextHops  ← hold empty   −≺ nextHopsE
  let tableModE = mapFilterE (packetToCmd nextHops) (packetInE msgE)
  clearE    ← clearOnJoin  −≺ msgE
  floodE    ← floodPackets1 −≺ msgE
  returnA −≺ mergeEventsBy (⊕) [clearE, tableModE, floodE]
```

Here we pass the output stream of *nextHopsSF* through *hold empty*, which turns the event stream into a signal defined at all times by starting off as *empty* and then holding the value of the last event in its input signal. We evaluate *packetToCmd nextHops* on every incoming packet, which results in a

Maybe SwitchCommand value. Applying *mapFilterE* (*packetToCmd nextHops*) filters out those packet events for which *packetToCmd nextHops* evaluates to *Nothing*, and evaluates to an event carrying x whenever *packetToCmd nextHops* evaluates to *Just x*. The function *packetToCmd* looks up the source and destination ports in the *nextHops* and if these are both present, returns a command, and otherwise returns nothing:

```
packetToCmd nextHops (sid, PacketIn {enclosedFrame}) =
   case lookup (sid, s) nextHops of
      Just ps  → case lookup (sid, r) nextHops of
                      Just pr  → Just (makeCommand sid s ps r pr)
                      Nothing → Nothing
      Nothing → Nothing
   where (s, r) = (sourceAddress enclosedFrame, destAddress enclosedFrame)
```

In turn, the function *makeCommand* outputs a command consisting of three commands in sequence:

```
makeCommand sid s ps r pr =
   deleteRules (ethSourceDestAre s r ∨ ethSourceDestAre r s) sid ⊕
   insertRule (flowFromTo s ps r pr 30) sid ⊕
   insertRule (flowFromTo r pr s ps 30) sid
```

The first deletes any existing rules matching packets from source s to destination r or vice versa. The second command inserts a rule that forwards any incoming traffic on port *ps* from s with destination r on outgoing port *pr*. The third rule is similar. Both inserted flows are set to expire after 30 seconds of inactivity. We omit the straightforward definition of *flowFromTo* here.

The inserted rules match on *both* the destination and the *source* address of a packet. Matching on the source is in fact crucial to the correctness of the controller. If we omit matching on the source address, then the switch will forward traffic from any sources — including sources whose location is unknown to the controller — toward the destination, bypassing the controller. This may result in the controller not learning the location of some hosts and consequently flooding packets unnecessarily.

6 Declarative Routing

The learning switch router in the previous section is arguably too low-level: the overall goal of the program is lost in the details of stream transformers. It would be preferrable to express our program by simply describing the forwarding table of each switch in our network at every moment in time. To do this, we will describe the forwarding tables in terms of quantities which themselves vary over time. We illustrate this idea by rewriting our learning switch in this fashion.

The essential feature of the learning switch controller is that it inserts flow rules in switches so that, at any time, the packets for any pair of hosts whose location is known at that time are forwarded directly between them with no flooding. Thus, the collection of rules present in a switch at any time depends on which host locations are known and where those hosts are at that time. If we name the current values of these quantities *knownHosts* and *nextHops*, we can express the desired collection of rules for switch *sid* as the following set:

$$\{\, \mathit{inPortIs}\ sp \wedge \mathit{ethSourceIs}\ s \wedge \mathit{ethDestIs}\ d \Longrightarrow \mathit{sendOnPort}\ dp \\ \mid s, d \in \mathit{knownHosts}, s \neq d, sp \in \mathit{nextHops}\ \mathit{sid}\ s, dp \in \mathit{nextHops}\ \mathit{sid}\ d \,\}$$

Motivated by this, we define a *SwitchProgram* to be a signal function that periodically outputs updated flow rules for each switch:

```
type SwitchProgram =
  SF (Event (SwitchID, SwitchMessage)) (Event (SwitchID → [FlowRule]))
```

A collection of switches governed by a *SwitchProgram* should forward traffic at any moment according to the flow rules of the most recent event of the output stream of the program. For example, we can implement the program for the learning switch as follows:

```
program₁ = proc msgE → do
  knownHosts ← knownHostsSF −≺ msgE
  nextHopsE  ← nextHopsSF   −≺ msgE
  nextHops   ← hold empty   −≺ nextHopsE
  let rules₁ sid =
    [inPortIs sp ∧ ethSourceIs s ∧ ethDestIs d ⟹ sendOnPort dp
     | s ← knownHosts, d ← knownHosts, s ≢ d,
       Just sp ← lookup (nextHops (sid, s)),
       Just dp ← lookup (nextHops (sid, d))
    ]
  returnA −≺ tag nextHopE rules₁
```

Here we use a Haskell *list comprehension* to simulate the set we wrote previously. Generators such as $s \leftarrow \mathit{knownHosts}$ introduce a variable to range over a given list. The left hand sides of generators can be patterns, such as in the final generator in the example above. In this case, the results include only those elements for which the pattern match succeeds. Guards, such as $s \not\equiv d$ filter elements from the resulting list.

We use *nextHopsSF* to output an event stream carrying updated next hop maps for the network. *knownHostsSF* outputs a list of hosts whose location is known, and is easily implementable in terms of *nextHopsSF*. Since both the *knownHosts* and *nextHops* values change precisely when the output stream of *nextHopSF* has an event, we output an updated list of flow rules at exactly that moment, using *tag* to output the value of rules_1 at exactly the moments when *nextHopE* carries a value.[1]

This approach allows us to easily extend our program in various ways. For example, we can modify the previous controller to have switches simply flood all ARP (Address Resolution Protocol) Ethernet frames as follows: (unchanged parts are elided):

[1] In addition to the rules that a switch should follow, the user must also specify how the controller should process packets for which no rule applies. Specifying this is straightforward, and we omit the description here due to space constraints.

```
program2 = proc msgE → do
  ...
  let rules1 sid = ...
  let rules2 sid = [arp ⟹ flood]
  let rules  sid = rules1 sid ++ rules2 sid
  returnA ─≺ tag nextHopE rules
```

Although in this case no packet will match more than one rule, we adopt the convention that the first rule in the list matching a packet applies. This convention allows us to compose rule sets with one taking precedence over another.

There are many feasible ways to implement a *SwitchProgram*. Due to space constraints we omit discussion of our simple implementation. However, we note that our implementation converts a *SwitchProgram* into a signal function:

```
runSP :: SwitchProgram →
           SF (Event (SwitchID, SwitchMessage)) (Event SwitchCommand)
```

Thus, we are able to implement the run-time envionment for our higher-level abstraction within Nettle/FRP. In this way, Nettle/FRP provides a convenient and powerful tool for exploring and implementing high level networking languages and abstractions.

7 Time-Varying Quantities

In this section we show how we can use continuous values in programming a dynamic load-balancing controller. This is a feature that other controller frameworks do not provide, but which we expect to be very useful in programming dynamic network controllers.

Consider a load balancing problem in which a single switch S has three links l_1, l_2 and l_3, as shown in Fig. 5. We assume that there are many traffic sessions in the network so that we can approximately model the network traffic using *traffic flow rates*. We will name the flow rates for links l_2 and l_3 as f_2 and f_3, respectively. In this highly simplified scenario, we imagine that traffic enters the system on link l_1 and that the switch can reach all destinations of this traffic by forwarding on either l_2 or l_3. Let r_{ideal} be the desired ratio of traffic that should flow over link l_2, r_{actual} be the actual ratio, and e be the error. That is:

$$\begin{aligned} r_{actual} &= \tfrac{f_2}{f_2+f_3} \\ e &= r_{ideal} - r_{actual} \end{aligned}$$

We would like our controller to maintain a balance of traffic such that $e \approx 0$. Note that the sign of e indicates whether the flow on port 2 should be increased or decreased: when $e < 0$, the flow should be decreased, otherwise it should be increased.

We implement a simple "dial" to control our switch. We will have a "dial", named a, that ranges over IP addresses, viewed as 32 bit integers. At any moment in time, we will forward all traffic destined to addresses less than or equal to a via link 2 and any traffic destined toward address greater than a via link 3. This

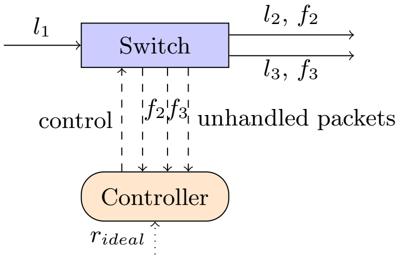


Fig. 5. Control system for Sec. 7. Solid lines correspond to physical links used by the network to send data traffic. Dashed lines indicate switch-controller communication.

dial is depicted in Fig. 6. Note that the ratio of addresses less than a to the total addresses does not indicate how much traffic will flow over links 2 and 3, since traffic could be unevenly distributed over the address space. Still, under a given traffic distribution, the dial's setting will determine how much traffic flows on each link: increasing a increases traffic on link 2 while decreasing it increases traffic on link 3, and it makes some sense to change the dial setting in proportion to the size of the current error. Furthermore, since traffic patterns change over time, there is no single right setting, and we will have to adjust it as our system evolves.

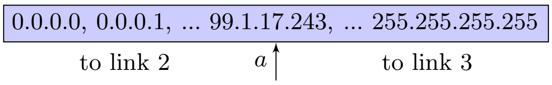


Fig. 6. The "dial" of Sec. 7

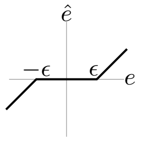


Fig. 7. Graph of the *dead zone* in $\hat{e}$

In order to use traditional control theory techniques, we first turn the problem into a continuous one. We define a new, real-valued version of our dial, u, ranging over $[0, 1]$. We can translate from u to a as follows:

$$a = \lfloor 2^{32} * u \rfloor \ .$$

As we argued above, we would like to change u in proportion to the error. We can write this as a simple differential equation:

$$\dot{u} = ke$$

Integrating, we find that:

$$u(t) = k \int_0^t e(\tau)\mathrm{d}\tau + u(0)$$

We have now arrived at a familiar integral control. However, due to the discrete nature of our system, our control will be unstable. At any time there are a finite number of flows, and it will not be possible to split these flows in two groups so that the error is zero. We can mitigate this problem by introducing a *dead zone* into the error signal. We define a new error signal $\hat{e}(t)$ as follows:

$$\hat{e}(t) = \begin{cases} e(t) - \epsilon & \text{if } e(t) > \epsilon \\ e(t) + \epsilon & \text{if } e(t) < -\epsilon \\ 0 & \text{otherwise} \end{cases}$$

Fig. 7 graphs the relationship between e and $\hat{e}$. Introducing the dead zone into the error term effectively turns the error signal off, once the controller is sufficiently close to reducing the error to 0. The size ϵ we need will depend on our assumptions about the minimum number and size of flows in our network.

We can now directly translate this mathematical model into Nettle code. The following defines a signal function implementing u, assuming the initial value u_0, gain k, dead zone size *eps*, and a suitable definition of *deaden* are defined elsewhere:

```
$uSF = \textbf{proc}\ (f_2, f_3, r_{ideal}) \rightarrow \textbf{do}$
  $\textbf{let}\ r_{actual} = f_2\ /\ (f_2 + f_3)$
  $\textbf{let}\ error\ \ = r_{ideal} - r_{actual}$
  $i \leftarrow integral \prec deaden\ eps\ error$
  $returnA \prec k * i + u_0$
```

We omit the overall controller, which includes signal functions providing the flow rates f_2, f_3 and the user determined r_{ideal}. Although the simplistic control algorithm presented here does not perform well in practice (because of a delay in observing port flow rates), it nevertheless illustrates how continuous quantities can be used to implement network control algorithms; a more sophisticated controller would have the same structure.

8 Discussion and Related Work

We have implemented all of the ideas in this paper (and more) using standard Haskell using the GHC Haskell compiler. We have tested our controllers using a reference implementation of OpenFlow version 1.0. We have also done preliminary tests of some of our controllers on real OpenFlow switches, which demonstrate that our controllers perform comparably with existing control frameworks, such as NOX [2]. All of our code is available on `hackage.haskell.org`

Future work includes refinements to Nettle/FRP and improvements to our dynamic continuous controller design. We also plan to develop more DSLs at the top-most layer of our software architecture, to capture more domain specific features, such as security and business relationships. We are also interested in studying the problem of protocol and policy verification techniques.

We previously used the name "Nettle" for an embedded DSL in Haskell for describing BGP router configurations [16]. At the risk of creating some confusion (since the languages are entirely distinct and have different purposes), we have reused that name for the language we present in this paper.

NOX [2] is an open-source library for writing controllers for OpenFlow switches in C++ and Python. Both NOX and Nettle provide a framework for writing controllers that hide low-level details from the user, allow fine-grained control over switch behavior, and provide an event-based programming model with extensible collections of events.

Nettle provides a more declarative approach to event-based programming by handling entire message streams, instead of individual messages. Nettle has a more expressive language for composing controllers – in parallel as in NOX, but also in sequence, and in many other combinations. The interactions between controllers is made explicit through lightweight input and output types of the components. In contrast, the interaction of NOX components requires investigating the internals of each component, since modules may interact imperatively by method invocation. Nettle also provides an elegant, declarative mechanism for describing time-sensitive and time-varying behaviors, whereas in NOX these must be simulated by delays and timers. Finally, Nettle has continuous quantities that reflect abstract properties of a network, such as the volume of messages on a network link. We are not aware of any other language that has this capability.

Flow-based Management Language (FML) [7] is a declarative policy language for configuring networks. An FML program is a Datalog-like set of rules that ultimately describe which forwarding actions should hold of flows. Although FML provides a higher-level abstraction than Nettle, many applications cannot be expressed in FML. In particular, FML has no way of expressing dynamic policies, where forwarding decisions change over time. Nettle provides a more concrete abstraction that exposes the message-passing interface to OpenFlow switches, but within a strongly typed language, Haskell, and within an expressive FRP layer. This allows Nettle users to extend Nettle and program in ways very similar to FML, as seen in Section 6.

Frenetic [5] is an FRP-based language for controlling OpenFlow networks, embedded in Python. Frenetic presents a “program like you see every packet” abstraction, thus providing a higher-level abstraction than Nettle. Nettle on the other hand, introduces FRP at the message stream level, while leveraging the embedding in Haskell to enable the development of higher level abstractions.

The Declarative Networking [10] approach uses a Datalog-like language to express routing protocols as recursive queries executing over a distributed collection of routers. Declarative Networking thus targets a different type of system than Nettle, since Nettle is aimed at OpenFlow-based systems in which switches have no query-processing capabailities.

Acknowledgements. This research was supported in part by STTR grant number ST061-002 from the Defense Advanced Research Projects Agency. We thank our STTR industrial partner, Galois, Inc. for its support, Ashish Agarwal for help with the Haskell/OpenFlow layer, and Nick Feamster and Sam Burnett with feedback and help testing Nettle on their OpenFlow network.

References

1. http://www.openflowswitch.org/
2. http://noxrepo.org/wp/
3. Caesar, M., Rexford, J.: BGP routing policies in ISP networks. IEEE Network 19(6), 5–11 (2005)
4. Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming, pp. 263–273 (June 1997)
5. Foster, N., Harrison, R., Meola, M.L., Freedman, M.J., Rexford, J., Walker, D.: Frenetic: A high-level langauge for openflow networks. In: ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO) (November 2010)
6. Griffin, T.G., Jaggard, A.D., Ramachandran, V.: Design principles of policy languages for path vector protocols. In: SIGCOMM 2003: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. pp. 61–72. ACM, New York(2003)
7. Hinrichs, T.L., Gude, N.S., Casado, M., Mitchell, J.C., Shenker, S.: Practical declarative network management. In: WREN 2009: Proceedings of the 1st ACM Workshop on Research on Enterprise Networking, pp. 1–10. ACM, New York (2009)
8. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Robots, arrows, and functional reactive programming. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638. Springer, Heidelberg (2003)
9. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. 37(1-3), 67–111 (2000)
10. Loo, B.T., Hellerstein, J.M., Stoica, I., Ramakrishnan, R.: Declarative routing: extensible routing with declarative queries. In: SIGCOMM 2005: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 289–300. ACM, New York (2005)
11. Mahajan, R., Wetherall, D., Anderson, T.: Understanding BGP misconfiguration. In: SIGCOMM, Pittsburgh, PA, pp. 3–17 (August 2002)
12. Paterson, R.: A new notation for arrows. In: ICFP 2001: Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming, pp. 229–240. ACM, New York (2001)
13. Peterson, J., Hager, G., Hudak, P.: A language for declarative robotic programming. In: International Conference on Robotics and Automation (1999)
14. Ramachandran, V.: Foundations of Inter-Domain Routing. Ph.D. thesis, Yale University (May 2005)
15. Voellmy, A., Agarwal, A., Hudak, P., Feamster, N., Burnett, S., Launchbury, J.: Don't configure the network, program it! domain-specific programming languages for network systems. Tech. Rep. YALEU/DCS/RR-1432, Yale University (July 2010)
16. Voellmy, A., Hudak, P.: Nettle: A language for configuring routing networks. In: Taha, W.M. (ed.) DSL 2009. LNCS, vol. 5658, pp. 211–235. Springer, Heidelberg (2009)