# Nettle: Functional Reactive Programming of OpenFlow Networks

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 of Nettle: using the *discrete* nature of FRP we are able to elegantly capture control messages to and from OpenFlow switches as streams of Nettle events; using the *continuous* nature of FRP we are able to express dynamic load balancing algorithms in a concise manner, reflecting directly the mathematics of the underlying control system. 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 and unintended ways [5], and gives little hope in validating high-level protocols and policies such as those related to traffic engineering, business relationships, security [11, 3].

Part of the problem is that most conventional routers are not only low-level, they are also decidedly impoverished in expressive power, and inflexible in their configuration capabilities – it is sometimes difficult to get even the most basic configurations to work correctly.

We believe that these problems can be overcome through the use of advanced high-level programming languages and tools that allow one to express the overall network behavior as a single program expressed in a declarative style. Although this idea has been suggested by several researchers [3, 9], 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 commercial routers in favor of *OpenFlow switches* [1]. OpenFlow presents a unified, flexible, dynamic, remotely programmable interface that allows network switches 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 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*!" 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

We are interested in the problem of configuring a local network of OpenFlow switches, varying in size from a single router to several hundred, or even thousands. Such a network may belong to a commercial entity, an 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 to a centralized *controller*. It is here that a Nettle program runs, thus forming a global control policy for the entire local network.

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, HOpenFlow, that abstractly captures the OpenFlow protocol.

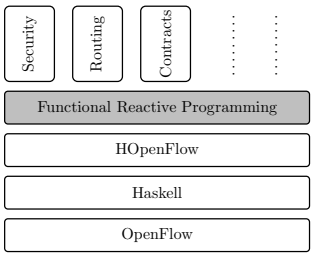



**Fig. 1.** Nettle layered system architecture.

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 interactive 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 [10, 4]. This is the layer that is the focus of this paper.

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 [12] we describe a DSL for expressing a class of dynamic security policies for campus networks and its implementation on Nettle's FRP layer.

Our contributions in the design of Nettle/FRP may be summarized as follows:

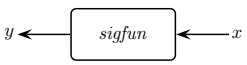
1. We have designed a discrete, event-based abstraction that captures communication patterns to and from OpenFlow switches. With this abstraction we are able to develop complete controllers in an elegant style that treat each message stream as a whole, rather than as individual messages.
2. We have designed a notion of continous, time-varying quantities that capture higher-level abstractions such as traffic volume on individual network links. With this abstraction we are able to think of certain kinds of network control, such as traffic engineering, as a continous control problem, much like in conventional control theory.
3. We have implemented all 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.
4. We have used Nettle/FRP to design several non-trivial controllers that solve realistic networking objectives, as well as exotic ones that show off the power of our approach. A selection of these controllers is described in this paper.

The remainder of the paper is organized as follows: In the next section we introduce the basic concepts of Nettle/FRP, and its specific use in controlling OpenFlow switches. In Section 4 we describe a series of increasingly complex discrete controllers, culminating in Section 5 in a larger example: a "learning switch." Then in Section 6 we introduce the notion of continuous quantities, and use it in the design of dynamic controllers for traffic enginnering. Finally, in Section 7 we compare our approach to related work.

## 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* [7], an FRP-based DSL that we previously designed for use in 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: $y \leftarrow sigfun \prec\!\!- x$

Nettle 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:

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

The first line in this program is a type signature that declares *sigfun* to be a signal function that converts continuous values of type *Float* into continuous values of type *Float*. The notation **proc** $x \rightarrow$ **do**... in the body of *sigfun* introduces a signal function, giving the name $x$ to the input signal. We then use the specialized arrow syntax $var \leftarrow sf \prec\!\!- exp$ introduced above to feed signals through signal functions and to name intermediate signals, in this case $y$. Finally, we denote the output of the signal function by feeding $y$ into *returnA*, a special signal function that returns the final result.

In Nettle we can use signals and signal functions in this way to program, for example, controllers that alter traffic flow based on signals that measure the volume of traffic on particular links. Indeed, we present an example of such a controller in Sec. 6. However, in this paper we place more emphasis on a different use: we will use signals to represent *streams of control messages* flowing to and from our OpenFlow switches – you can think of each signal (i.e. wire) as being a stream of messages.

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 simply and elegantly unifies message streams with continuous signals by representing message streams as continuous signals that are only defined at discrete points in time. More concretely, Nettle represents a discrete signal that periodically carries information of some type $\alpha$ as a signal that takes on values of the *Event* $\alpha$ datatype, whose values are either *NoEvent* or *Event* $a$ for some $a :: \alpha$. Note that *Event* $\alpha$ is isomorphic to Haskell's *Maybe* $\alpha$ data type. For example, a signal function that converts a message stream carrying messages of type $M_1$ into a message stream carrying messages of type $M_2$ has type $SF\ (Event\ M_1)\ (Event\ M_2)$.

Nettle provides several constructs that convert between discrete and continuous signals. One we will use later is $hold :: a \rightarrow SF\ (Event\ a)\ a$, that converts a stream of events carrying values of type $a$ into a continuous, piece-wise constant signal of $a$ values. The output signal of $hold\ a_0$ "holds" the last $a$ value received on its input line, and starts out as $a_0$.

Another useful stateful signal function is $accum :: a \rightarrow SF\ (Event\ (a \rightarrow a))\ (Event\ a)$. $accum\ a_0$ 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.

## 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 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 a format defined by the OpenFlow protocol. Optional expiration settings cause a flow entry to expire after some prescribed time.

OpenFlow switches attempt to establish a connection with a controller at a pre-configured IP address, using TCP. Typically, the switch-controller communication takes place over a control network that is separate from the main data network. The switches and controller communicate using the OpenFlow protocol, which defines message formats that allows controllers to query for information and command switches to perform actions or change state and allows switches to inform controllers of events.

We model messages from switches with a data type *SwitchMessage* and commands to switches (including queries) with a data type *SwitchCommand*. For example, one variant in the *SwitchMessage* data type is *SwitchJoin SwitchID SwitchFeatures*. Values of this form will signal that a switch has connected with the controller and hence "joined" the network, and such values carry information about the joining switch. Commands include commands to send packets, modify flow tables, and request information.

A Nettle program is a signal function having an input type carrying switch messages from all switches in the network and output type carrying switch commands to any switches in the network, i.e. it has a type:

$$SF\ (Event\ SwitchMessage)\ (Event\ SwitchCommand)$$

### 4.1 Basic Event Handling and Switch Commands

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

$$\begin{aligned} &controller_0 = \mathbf{proc}\ evt \rightarrow \mathbf{do} \\ &\quad cmdEvent \leftarrow never \multimap evt \\ &\quad returnA \multimap cmdEvent \end{aligned}$$

We use the library signal function *never* 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:

```
clearOnJoin = proc evt → do
  switchJoinEvt ← switchJoins —≺ evt
  returnA —≺ (clearTable switchJoinEvt)
```

Here we use *switchJoins* to extract the stream of switch join events from the switch message stream. We name the output of this signal function *switchJoinEvt* and for each such event, we apply *clearTable* to the event, deleting all entries from the flow table of the joining switch.

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:

```
floodPackets1 = proc evt → do
  packetInEvt ← packetIns —≺ evt
  returnA —≺ sendReceivedPacket flood packetInEvt
```

Here we use *packetIns* to extract only the *PacketIn* messages from the incoming message stream. For each such event, we apply *sendReceivedPacket flood*, instructing the switch to send the port using the action *flood*, 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:

```
controller1 = proc evt → do
  clearCmd ← clearOnJoin  —≺ evt
  floodCmd ← floodPackets1 —≺ evt
  returnA —≺ clearCmd ⊕ floodCmd
```

In this signal function we feed the incoming message stream to *both* signal functions. We then combine their outputs with the expression *clearCmd* ⊕ *floodCmd* which combines the two commands in sequence, i.e. the effect of the combined command is the effect of the first command followed by the effect of the second command.

### 4.2 Programming the Flow Table

In the previous controller, our 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 this controller by taking advantage of the special-purpose switching hardware present in the switches. We can do this by installing an appropriate flow rule whenever a switch joins the network:

$$
\begin{aligned}
&\mathit{floodPackets}_2 = \mathbf{proc}\ \mathit{evt} \to \mathbf{do} \\
&\quad \mathit{joinEvt} \leftarrow \mathit{switchJoins} \prec\!\!- \mathit{evt} \\
&\quad \mathit{returnA} \prec\!\!- \mathit{liftE}\ f\ \mathit{joinEvt} \\
&\quad \mathbf{where}\ f\ (\mathit{sw}, \_) = \mathit{insertRule}\ (\mathit{anyPacket} \Longrightarrow \mathit{flood})\ \mathit{sw}
\end{aligned}
$$

Here we use the function $\mathit{liftE} :: (a \to b) \to \mathit{Event}\ a \to \mathit{Event}\ b$ to lift an ordinary function to operate over events. Again, we can combine this in parallel with *clearOnJoin* to form a complete controller:

$$
\begin{aligned}
&\mathit{controller}_2 = \mathbf{proc}\ \mathit{evt} \to \mathbf{do} \\
&\quad \mathit{clearCmd} \leftarrow \mathit{clearOnJoin}\ \ \prec\!\!- \mathit{evt} \\
&\quad \mathit{floodCmd} \leftarrow \mathit{floodPackets}_2 \prec\!\!- \mathit{evt} \\
&\quad \mathit{returnA} \prec\!\!- \mathit{clearCmd} \oplus \mathit{floodCmd}
\end{aligned}
$$

In this case, *clearCmd* and *floodCmd* will occur simultaneously, and hence the order in which the commands are combined is crucial.

## 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 bridge, 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 and will use that data type's *insert* function. We will build the table by transforming each packet-in event into a table update, accumulating these updates with *accum*, and finally using *hold* to hold the most recent table between events:

$$
\begin{aligned}
&\mathit{hostDirectionTracker} = \mathbf{proc}\ \mathit{evt} \to \mathbf{do} \\
&\quad \mathit{packetEvt} \leftarrow \mathit{packetIns}\ \ \ \ \prec\!\!- \mathit{evt}
\end{aligned}
$$

```
tblEvent  ← accum empty —≺ liftE updateMap packetEvt
hold empty —≺ tblEvent
```

The function *updateMap* is straightforward: it updates the table for key $(swid, addr)$ to be the port ID of the port on which the packet was received:

```
updateMap (swid, pktInfo) tbl = insert (swid, ethSrcAddr) inPort tbl
  where inPort     = packetInPort pktInfo
        ethSrcAddr = sourceAddress (enclosedFrame pktInfo)
```

In this code we make use of a Nettle library function *enclosedFrame* which parses the enclosed packet data into a prefix of an Ethernet frame.

We can use *hostDirectionTracker* to write a signal function *tableManager* that outputs forwarding rules incrementally:

```
tableManager = proc evt → do
  clearCmd     ← clearOnJoin —≺ evt
  hostDirTable ← hostDirectionTracker —≺ evt
  packet       ← packetIns —≺ evt
  returnA —≺ clearCmd ⊕ mapFilterE (packetToCmd hostDirTable) packet
```

Here we simply evaluate *packetToCmd hostDirTable* on every incoming packet, which results in a *Maybe SwitchCommand* value. Applying *mapFilterE*, results in outputting no events if the value is *Nothing*, and outputting an event carrying *cmd* whenever the value is *Just cmd*. The function *packetToCmd* looks up the source and destination ports in the *hostDirTable* and if these are both present, returns a command, and otherwise returns nothing:[1]

```
packetToCmd hostDirTable (swid, pktInfo) =
  do ps ← lookup (swid, s) hostDirTable
     pr ← lookup (swid, r) hostDirTable
     return (makeCommand swid s ps r pr)
  where ethFrame = packetInFrame pktInfo
        (s, r)   = (sourceAddress ethFrame, destAddress ethFrame)
```

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

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

[1] This code uses the *Maybe* monad. If any computation in a *Maybe* monad evaluates to *Nothing* then the entire computation returns *Nothing*. Only if every computation succeeds with then the entire computation returns *Just result*.

```
insertRule (flowFromTo r pr s ps) swid
where flowFromTo s ps r pr =
        ((inPortIs ps ∧ ethSourceDestAre s r) ⟹ sendOnPort pr)
        `expireAfterInactiveFor` 30
```

The first deletes any existing rules at the switch for this pair of hosts by applying *deleteRules* to the predicate *ethSourceDestAre s r* ∨ *ethSourceDestAre r s* deleting any rule that matches packets from source with Ethernet address $s$ to destination with Ethernet address $r$ 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 can wire all the pieces together to define our controller, as follows:

```
controller3 = proc evt → do
  tableCmd ← tableManager -≺ evt
  floodCmd ← floodPackets1 -≺ evt
  returnA -≺ tableCmds ⊕ floodCmd
```

## 6 Time-Varying Quantities

In this section, we show how we can use a seemingly exotic feature of FRP, namely quantities that vary over continuous time, i.e. defined at all points in time, to good effect in programming a load balancing controller. This is a feature that other controller frameworks do not provide, but which we expect will be very useful in programming control systems for networks.

In this example, we will 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. 2. 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 $l_3$. Further, an operator specifies the desired ratio, $r_{ideal}$, of traffic that should flow over link $l_2$. Let $r_{actual}$ be the actual ratio of flow on link 2 to total flow, i.e.

$$r_{actual} = \frac{f_2}{f_2 + f_3}$$

and let the *error*, $e(t)$ be

$$e(t) = r_{ideal}(t) - r_{actual}(t) \ .$$

We would like our controllers to maintain a balance of traffic such that $e(t) \approx 0$. Note that the sign of $e$ indicates whether the flow on port 2 should be increased or decreased: when $e < 0$, then $r_{actual} > r_{ideal}$ and the flow on port 2 should be decreased, while if $e > 0$ the flow on port 2 should be increased.

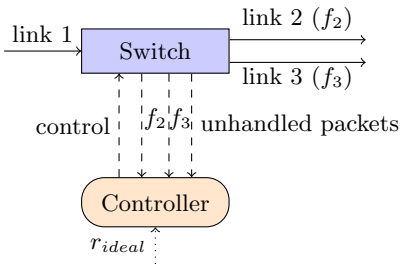



**Fig. 2.** Control system for Sec. 6. Solid lines correspond to physical links used by the network to send data traffic. Dashed lines indicate switch-controller communication.

We will present two controllers for accomplishing this control. The first controller is a simple feedback system, while the second is based on the proportional-integral-derivative (PID) controller used in many control systems. Before we do this, however, we will discuss how we can measure traffic flows in the network and react to operator input.

### 6.1 Measuring Traffic Flows

OpenFlow switches track various statistics, including packet and byte counts for packets received and sent for each port. Nettle includes commands and events that allow a controller to query the switch for port statistics and to handle the switches' responses to these requests. To monitor the port statistics of a switch, we periodically query the switch for this information, and process the responses. Nettle includes a library function

$$\mathit{portFlowMonitor} :: \mathit{Time} \to \mathit{SF}\ (\mathit{Event}\ \mathit{SwitchMessage})\ (\mathit{Event}\ \mathit{SwitchCommand})$$

that queries the connected switches in the network for port statistics periodically, with periodicity defined by the first *Time* argument. As a result, connected switches will send port statistics periodically, and these can be processed using the signal function:

$$\begin{aligned}\mathit{portStatistics} :: \mathit{SwitchID} &\to \mathit{PortID} \to \\ &\mathit{SF}\ (\mathit{Event}\ \mathit{SwitchEvent})\ (\mathit{Event}\ \mathit{PortStatsVector})\end{aligned}$$

which returns a record containing numerous statistics. In our case, the quantity of interest is the number of bytes sent, which can be accessed by $\mathit{sentBytes} :: \mathit{PortStatsVector} \to \mathit{Double}$. Assuming that we have a signal processor, $\mathit{sampledDerivative} :: \mathit{SF}\ (\mathit{Event}\ \mathit{Double})\ \mathit{Double}$, that returns the rate of change of a discretely sampled numeric signal, we can write a signal function that computes the byte send rate of a port:

$$\mathit{portBytesSentRate} :: \mathit{SwitchID} \to \mathit{PortID} \to \mathit{SF}\ (\mathit{Event}\ \mathit{SwitchEvent})\ \mathit{Double}$$

### 6.2 Operator Input

We can track the ideal rate $r_{ideal}$ as follows: we start with a rate of 0.5 (i.e. balanced); at any time an operator may input a new decimal number between 0 and 1, which becomes the new rate:

$$
\begin{aligned}
&\mathit{operatingRate} = \mathbf{proc}\ \mathit{msgIn} \rightarrow \mathbf{do} \\
&\quad \mathit{hold}\ 0.5 \multimap \mathit{mapFilterE}\ \mathit{parseRate}\ \mathit{msgIn}
\end{aligned}
$$

where $\mathit{parseRate} :: \mathit{String} \rightarrow \mathit{Maybe\ Double}$ parses a floating point value from a string input by the user. We omit the definition of *parseRate* here for brevity.

### 6.3 Discrete Feedback Controller

Using the components defined above, we can define a simple controller for approximately balancing the flows over links 2 and 3 according to the desired operating rate. We simply monitor the flow rates, $f_2$ and $f_3$ of links 2 and 3 and we let our preferred link be link 2 if more flow should be placed on link 2 to achieve our desired operating rate, and link 3 otherwise. That is, we define:

$$
\begin{aligned}
&\mathit{preferredLink}\ r_{ideal}\ f_2\ f_3 \\
&\quad \mid \mathit{err} > 0 \quad = 2 \\
&\quad \mid \mathit{otherwise} = 3 \\
&\quad \mathbf{where}\ r_{actual} = f_2\ /\ (f_2 + f_3) \\
&\qquad\qquad \mathit{err} \quad\ = r_{ideal} - r_{actual}
\end{aligned}
$$

We can write a signal function to track the *preferredLink* dynamically as:

$$
\begin{aligned}
&\mathit{preferredLinkSF} = \mathbf{proc}\ (\mathit{switchEvent}, \mathit{userEvent}) \rightarrow \mathbf{do} \\
&\quad f_2 \quad\ \leftarrow \mathit{portBytesSentRate}\ \mathit{switch1}\ \mathit{port2} \multimap \mathit{switchEvent} \\
&\quad f_3 \quad\ \leftarrow \mathit{portBytesSentRate}\ \mathit{switch1}\ \mathit{port3} \multimap \mathit{switchEvent} \\
&\quad r_{ideal} \leftarrow \mathit{operatingRate} \multimap \mathit{userEvent} \\
&\quad \mathit{returnA} \multimap \mathit{preferredLink}\ r_{ideal}\ f_2\ f_3
\end{aligned}
$$

We will use a simple strategy to control our traffic according to this dynamically changing preffered link quantity. We begin with an empty flow table at the switch. Whenever an unhandled packet arrives, we install a new flow rule at the switch with a narrowly defined match condition - that is, roughly matching only packets in the session being initiated by the packet - and forwarding on the link that is most preferred *at that moment*. Since, by assumption, there will be many sessions, this strategy will allow the controller to implement fine-grained load-balancing control. Furthermore, we will expire installed flows after some time period, to ensure that the controller has the opportunity to change its forwarding decisions for particular flows. Therefore, we write our packet handler as:

$$
\begin{aligned}
&\mathit{handlePacketIn} = \mathbf{proc}\ (\mathit{switchEvent}, \mathit{preferredLink}) \rightarrow \mathbf{do} \\
&\quad \mathit{packetEvt} \leftarrow \mathit{packetIns} \multimap \mathit{switchEvent}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{returnA} \prec (\mathit{liftE}\ (\mathit{flowForPacket}\ \mathit{preferredLink})\ \mathit{packetEvt}) \oplus \\
&\qquad (\mathit{sendReceivedPacket}\ (\mathit{sendOnPort}\ \mathit{preferredLink})\ \mathit{packetEvt}) \\
&\textbf{where}\ \mathit{flowForPacket}\ \mathit{preferredLink}\ (\mathit{swid}, \mathit{pktInfo}) = \\
&\quad \mathit{insertFlow}\ ((\mathit{exactPredicate}\ \mathit{pktInfo} \Longrightarrow \mathit{sendOnPort}\ \mathit{preferredPort}) \\
&\qquad\quad \text{`}\mathit{expireAfter}\text{`}\ 60)
\end{aligned}
$$

In this code we use the Nettle library function *exactPredicate* to compute a match condition that matches the packet as narrowly as possible with OpenFlow's match conditions.

Finally, we combine these signal functions to create our controller:

$$
\begin{aligned}
&\mathit{controller}_4 = \textbf{proc}\ (\mathit{switchEvent}, \mathit{userEvent}) \rightarrow \textbf{do} \\
&\quad \mathit{clearCmd} \leftarrow \mathit{clearOnJoin} \prec \mathit{switchEvent} \\
&\quad \mathit{preferredLink} \leftarrow \mathit{preferredLinkSF} \prec (\mathit{switchEvent}, \mathit{userEvent}) \\
&\quad \mathit{tableCmd} \leftarrow \mathit{handlePacketIn} \prec (\mathit{switchEvent}, \mathit{preferredLink}) \\
&\quad \mathit{monitorCmd} \leftarrow \mathit{portFlowMonitor}\ 0.1 \prec () \\
&\quad \mathit{returnA} \prec \mathit{clearCmd} \oplus \mathit{tableCmd} \oplus \mathit{monitorCmd}
\end{aligned}
$$

We have run this controller in a virtualized test bed, using a traffic generator that simulates traffic according to stochastic process models of the initiation and termination of traffic sessions and the traffic sent during each session. In the sample shown in Fig 3, we run the controller with an operating rate to 0.5, and with a traffic generator that creates a new traffic flows every 5 seconds, causing the total byte rate to increase over time. The sample run demonstrates how the controller assigns flows to ports to bring the total flow more in balance.

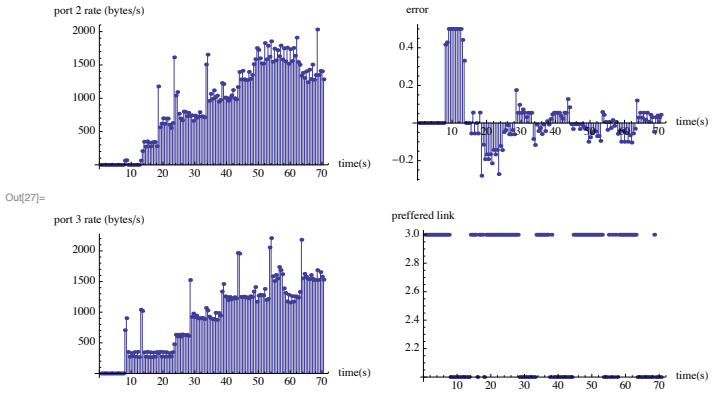



**Fig. 3.** A sample run of the controller of Sec. 6.3. The plots on the left are the byte rates on ports 2 and 3. The plot on the top right is the error while the bottom right is the preferred link. The values of these quantities were sampled every 0.5 seconds.

### 6.4 Continuous Control

Using OpenFlow we can implement a simple "dial" that we can use to control the 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 dial is depicted in Fig. 4. 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 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.

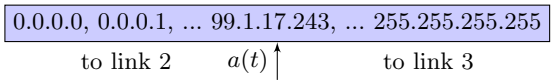



**Fig. 4.** The "dial": the manipulated variable for the control system of Sec. 6.4.

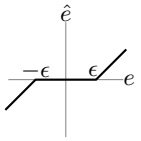



**Fig. 5.** Graph of the *dead zone* in $\hat{e}$.

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 $a(t)$ dial, $u(t)$, ranging over $[0, 1]$. We can translate from $u$ to $a$ as follows:

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

As we argued above, we would like to change $u(t)$ 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 control, 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. 5 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 $\epsilon$ 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 that tracks the continuous dial $u(t)$, assuming the initial value $u_0$, gain $k$, dead zone size *eps*, and a suitable definition of *deaden* are defined elsewhere:

$$
\begin{aligned}
&uSF = \mathbf{proc}\ (f_2, f_3, r_{ideal}) \rightarrow \mathbf{do} \\
&\quad \mathbf{let}\ r_{actual} = f_2\ /\ (f_2 + f_3) \\
&\quad \mathbf{let}\ error\ \ = r_{ideal} - r_{actual} \\
&\quad i \leftarrow integral \multimap deaden\ eps\ error \\
&\quad returnA \multimap k * i + u_0
\end{aligned}
$$

We omit the overall controller, which is very similar to previous controllers. While this controller does not yet perform well enough for a practical network, it illustrates the possibility of programming with continuous quantities for network control.

## 7 Related Work

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 in which users can extend the types of events in the system.

But Nettle provides a more declarative approach to event-based programming by handlng entire message streams, instead of individual messages. Nettle also 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) [6] is a declarative policy language for configuring and managing enterprise networks, built on top of NOX. An FML program is a Datalog-like set of rules that define when certain facts hold of a flow, and ultimately define which forwarding actions should hold of a given flow. Although FML is higher-level than Nettle, presenting a logic-based abstraction that is convenient in some applications, many applications cannot be

expressed in FML. For example, 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.

The Declarative Networking [8] 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.

## References

1. `http://www.openflowswitch.org/`
2. `http://noxrepo.org/wp/`
3. Caesar, M., Rexford, J.: BGP routing policies in ISP networks. Network, IEEE 19(6), 5 – 11 (nov-dec 2005)
4. Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming. pp. 263–273 (Jun 1997)
5. Griffin, T.G., Jaggard, A.D., Ramachandran, V.: Design principles of policy languages for path vector protocols. In: SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 61–72. ACM, New York, NY, USA (2003)
6. Hinrichs, T.L., Gude, N.S., Casado, M., Mitchell, J.C., Shenker, S.: Practical declarative network management. In: WREN '09: Proceedings of the 1st ACM workshop on Research on enterprise networking. pp. 1–10. ACM, New York, NY, USA (2009)
7. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Robots, arrows, and functional reactive programming. In: Summer School on Advanced Functional Programming, Oxford University. Springer Verlag, LNCS 2638 (2003)
8. Loo, B.T., Hellerstein, J.M., Stoica, I., Ramakrishnan, R.: Declarative routing: extensible routing with declarative queries. In: SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 289–300. ACM, New York, NY, USA (2005)
9. Mahajan, R., Wetherall, D., Anderson, T.: Understanding BGP misconfiguration. In: SIGCOMM. pp. 3–17. Pittsburgh, PA (Aug 2002)
10. Peterson, J., Hager, G., Hudak, P.: A language for declarative robotic programming. In: International Conference on Robotics and Automation (1999)
11. Ramachandran, V.: Foundations of Inter-Domain Routing. Ph.D. thesis, Yale University (5 2005)
12. 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)