Nettle: Declarative Event-Driven Network Control

Andreas Voellmy

Yale University
Joint work with Paul Hudak

Computer Communications Workshop 2011

Software-Defined Networking

- Move functionality into software-defined programs which users can change easily.
- Separate basic packet processing from higher-level logic:
  - Simple, flexible forwarding plane;
  - Complex, user-defined control software.
- Applications: building new systems (data center networking, etc), re-engineering old systems to make them more manageable.
- There are a variety of approaches to introducing programmable network components:
  - OpenFlow
  - Software routers, e.g. Click, Quagga, Xorp
  - Declarative Networking (Loo et al)

Language-based Solution to the Configuration Problem

Configurations are brittle; Millions of lines of configuration files in many networks.

Provide a higher level language that allows admins to describe the network behavior in a way that is comprehensible to them

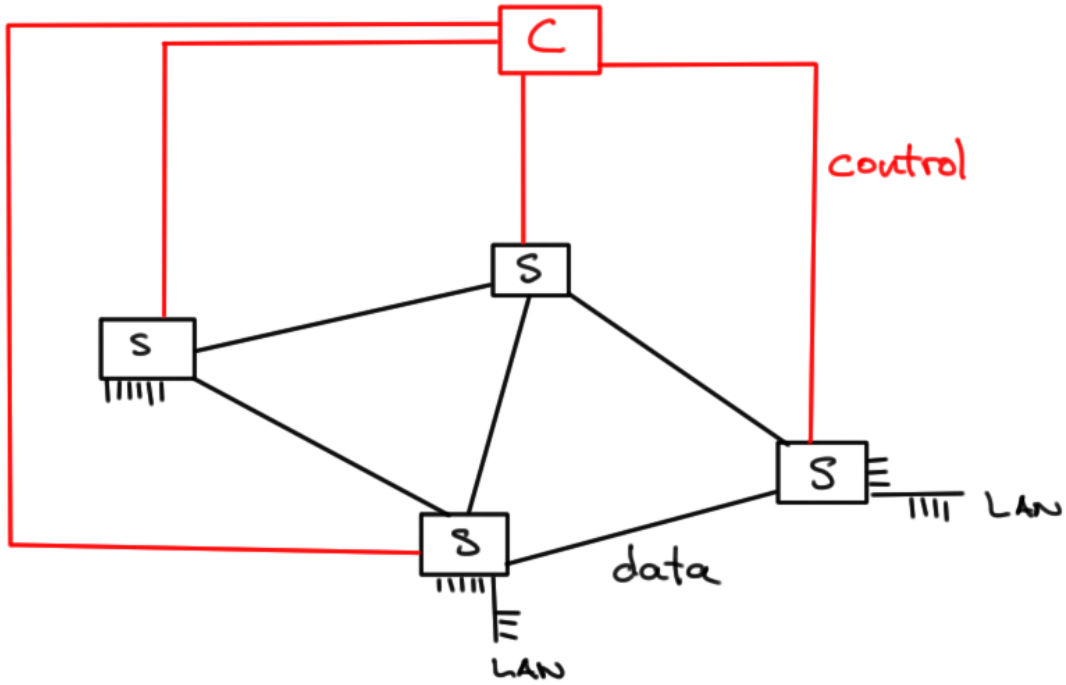
Implement a network controller that guarantees to correctly implement the user program.

Examples:

- Ethane/FSL; Casado et al.; Simple declarative security policies.
- Resonance; Feamster et al.; Dynamic security policies.

OpenFlow

- Standard API/Interface to L2/L3 packet processing functionality.
- Logically centralized controller, implemented on standard server.
- Programmable control with commercial-quality hardware; current implementations include NEC, HP.



OpenFlow - A Few Details

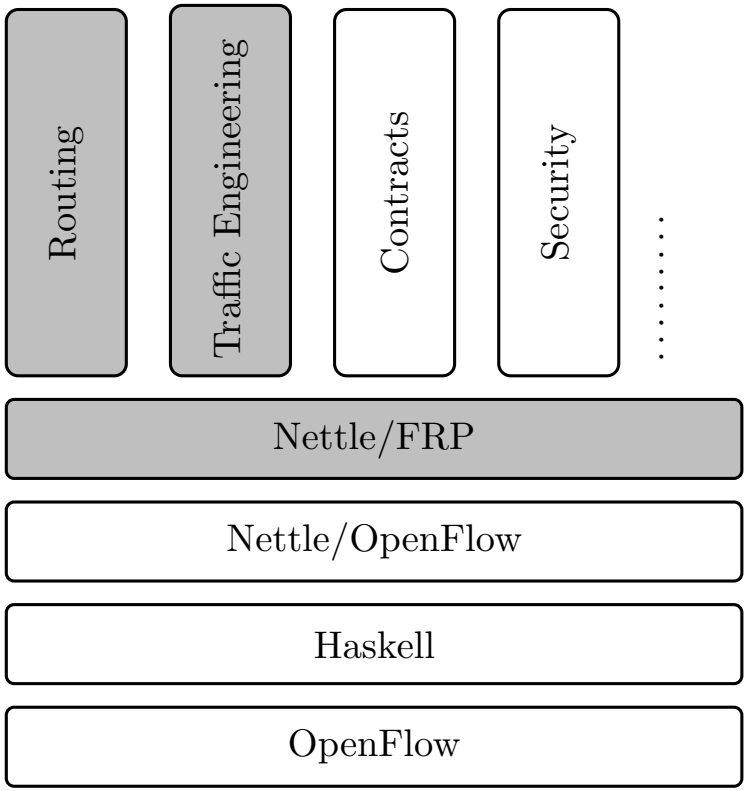
- Switch has forwarding rules consisting of (condition, action) pairs.
- Condition matches on packet headers
- Actions include forward on one or more ports, broadcast, modify headers, etc.
- Switch looks for matching condition; if match, update counters, perform action; otherwise send packet to controller.
- Controller installs forwarding rules at any time, but typically in reaction to packet...
- + API to get hardware config, traffic stats, link stats, etc.

Nettle Vision

Allow users to *describe network behavior* in a *single coherent program*, expressed in a *declarative style*.

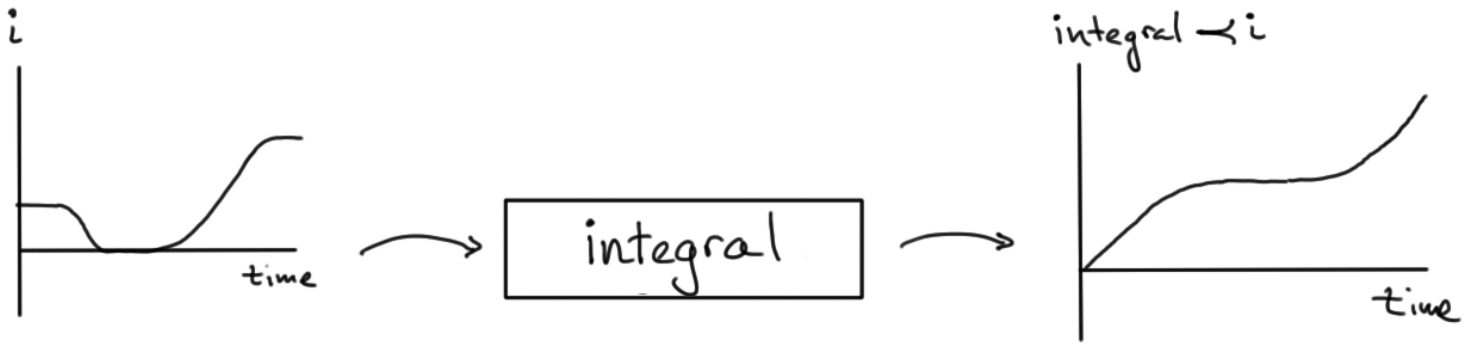
- Adopt methodology of *domain specific languages* (DSLs) to capture network abstractions.
- Use *functional reactive programming* (FRP) to provide a declarative method of programming dynamic, reactive behavior.

Nettle Software Architecture



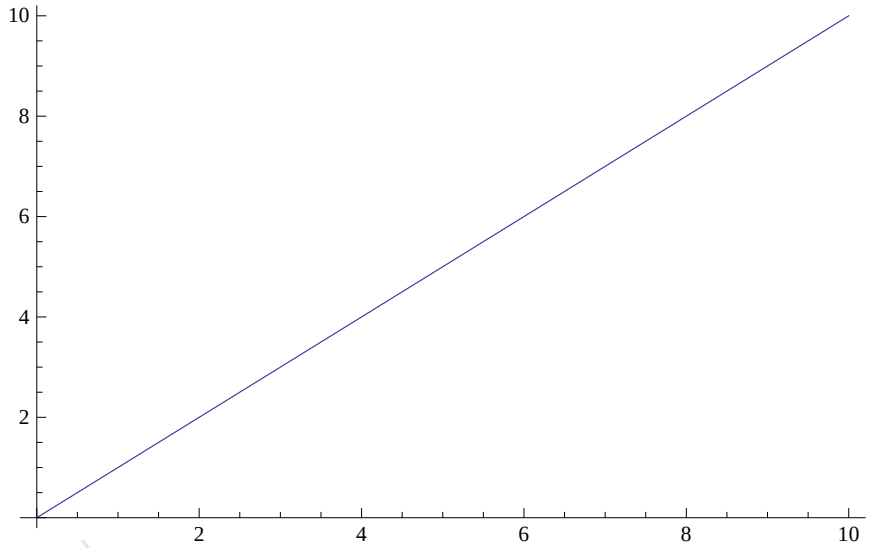
Functional Reactive Programming (FRP)

- A functional approach to programming reactive systems.
- An alternative to traditional callback-based imperative event-driven systems.
- An FRP program is a causal function that determines the output signal in terms of the input signal.
- Evaluate the function incrementally, interleaving input and output.



Nettle Examples: Signals

time



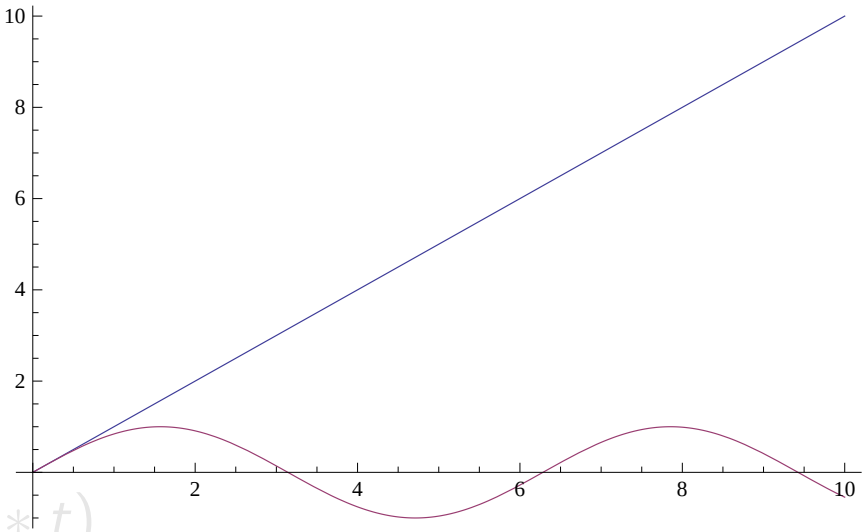

```
proc () → do
    t ← time ⤙ ()
    returnA ⤙ sin t
```

```
proc () → do
    t ← time ⤙ ()
    x ← integral ⤙ 1 + sin (2 ∗ t)
    returnA ⤙ sin t + x
```

Nettle Examples: Signals

time

```
proc () → do
    t ← time ⤙ ()
    returnA ⤙ sin t
```



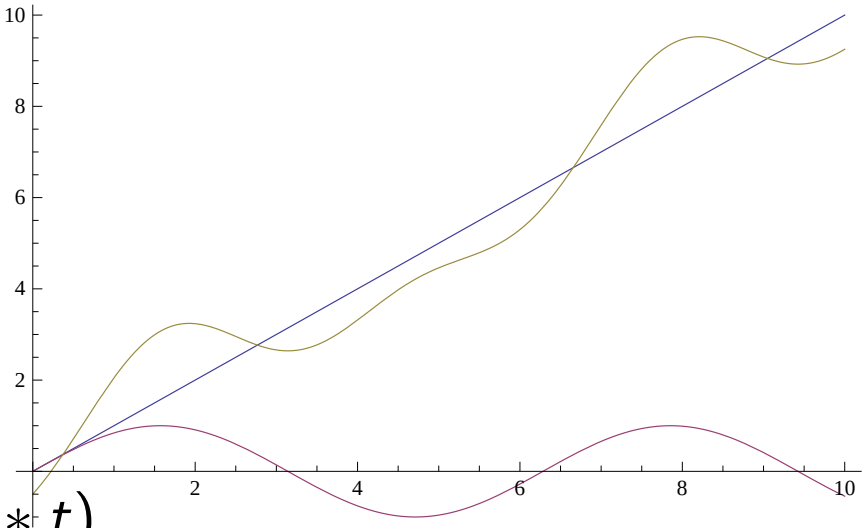

```
proc () → do
    t ← time ⤙ ()
    x ← integral ⤙ 1 + sin (2 ∗ t)
    returnA ⤙ sin t + x
```

Nettle Examples: Signals

$time$

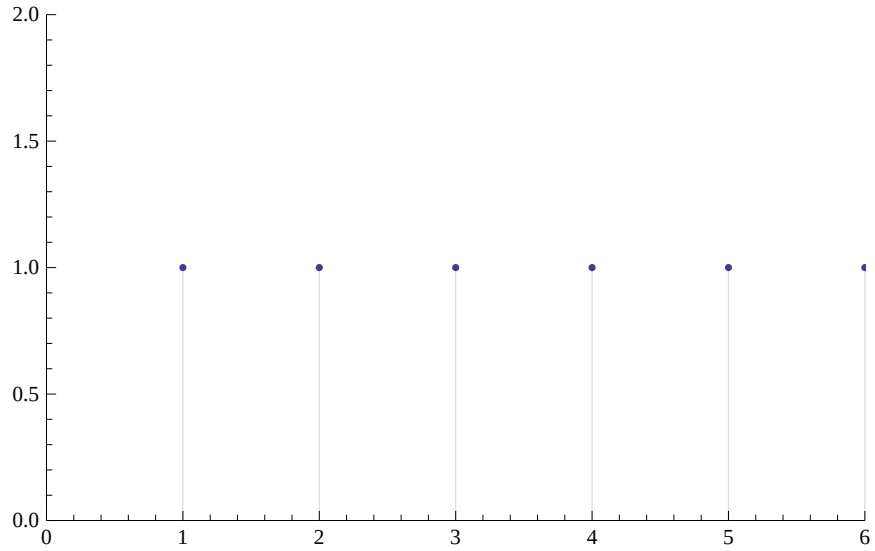
proc $() \rightarrow$ **do**
$t \leftarrow time \prec ()$
$returnA \prec sin\ t$

proc $() \rightarrow$ **do**
$t \leftarrow time \prec ()$
$x \leftarrow integral \prec 1 + sin\ (2 * t)$
$returnA \prec sin\ t + x$



Nettle: Events

repeatedly 1 1




```
proc () → do
  e ← repeatedly 1 1 ⪦ ()
  returnA ⪦ [n + 1 | n ← e]
```

```
proc () → do
  e1 ← repeatedly 1 1 ⪦ ()
  e2 ← repeatedly 1.5 1 ⪦ ()
  returnA ⪦ e1 `merge` e2
```

```
repeatedly 1 (λm → m + 2)
```

```
countBy2 = proc () → do
  e ← repeatedly 1 (λm → m + 2) ⪦ ()
  accum 0 ⪦ e
```

Nettle: Events

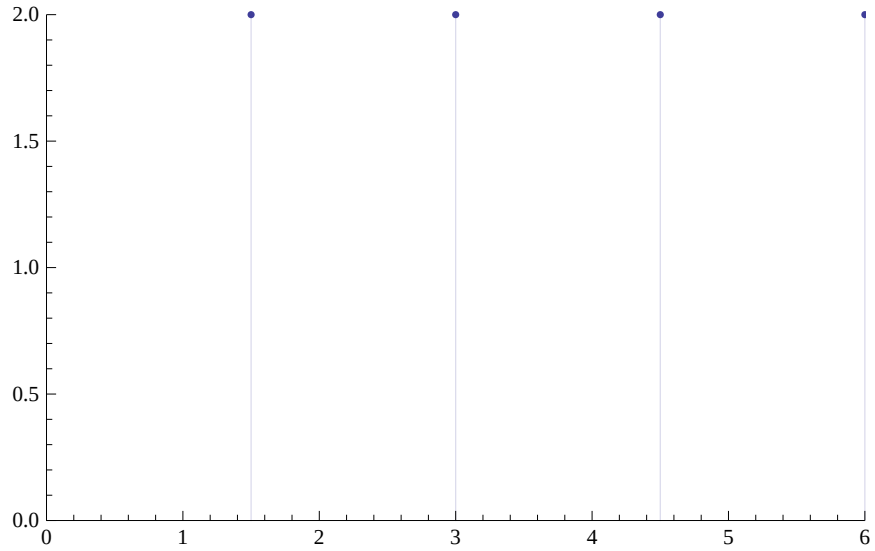
repeatedly $1\ 1$

proc $() \rightarrow$ **do**
  $e \leftarrow$ *repeatedly* $1\ 1 \prec ()$
  returnA $\prec [n+1 \mid n \leftarrow e]$

proc $() \rightarrow$ **do**
  $e_1 \leftarrow$ *repeatedly* $1\ 1 \prec ()$
  $e_2 \leftarrow$ *repeatedly* $1.5\ 1 \prec ()$
  returnA $\prec e_1$ ‘*merge*‘ e_2

repeatedly $1\ (\lambda m \rightarrow m+2)$

countBy2 = **proc** $() \rightarrow$ **do**
  $e \leftarrow$ *repeatedly* $1\ (\lambda m \rightarrow m+2) \prec ()$
  accum $0 \prec e$

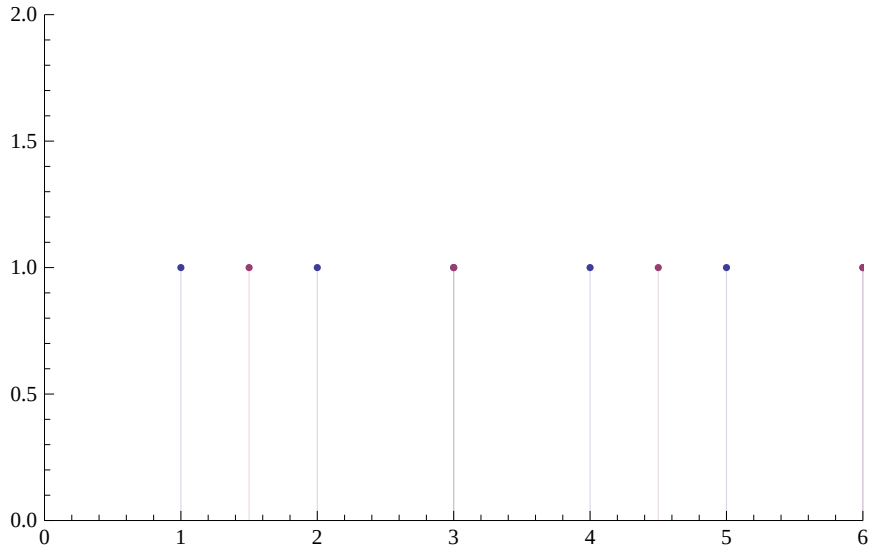


Nettle: Events

$$\mathit{repeatedly}\ 1\ 1$$

$$\begin{array}{l}\textbf{proc}\ () \to \textbf{do} \\ \quad e \leftarrow \mathit{repeatedly}\ 1\ 1 \prec () \\ \quad \mathit{returnA} \prec [n+1 \mid n \leftarrow e]\end{array}$$

$$\begin{array}{l}\textbf{proc}\ () \to \textbf{do} \\ \quad e_1 \leftarrow \mathit{repeatedly}\ 1\ 1 \prec () \\ \quad e_2 \leftarrow \mathit{repeatedly}\ 1.5\ 1 \prec () \\ \quad \mathit{returnA} \prec e_1\ \text{`}\mathit{merge}\text{`}\ e_2\end{array}$$



$$\mathit{repeatedly}\ 1\ (\lambda m \to m+2)$$

$$\begin{array}{l}\mathit{countBy2} = \textbf{proc}\ () \to \textbf{do} \\ \quad e \leftarrow \mathit{repeatedly}\ 1\ (\lambda m \to m+2) \prec () \\ \quad \mathit{accum}\ 0 \prec e\end{array}$$

Nettle: Events

$$\mathit{repeatedly}\ 1\ 1$$

$$\begin{aligned}&\textbf{proc}\ () \to \textbf{do}\\ &\quad e \leftarrow \mathit{repeatedly}\ 1\ 1 \prec ()\\ &\quad \mathit{returnA} \prec [n+1 \mid n \leftarrow e]\end{aligned}$$

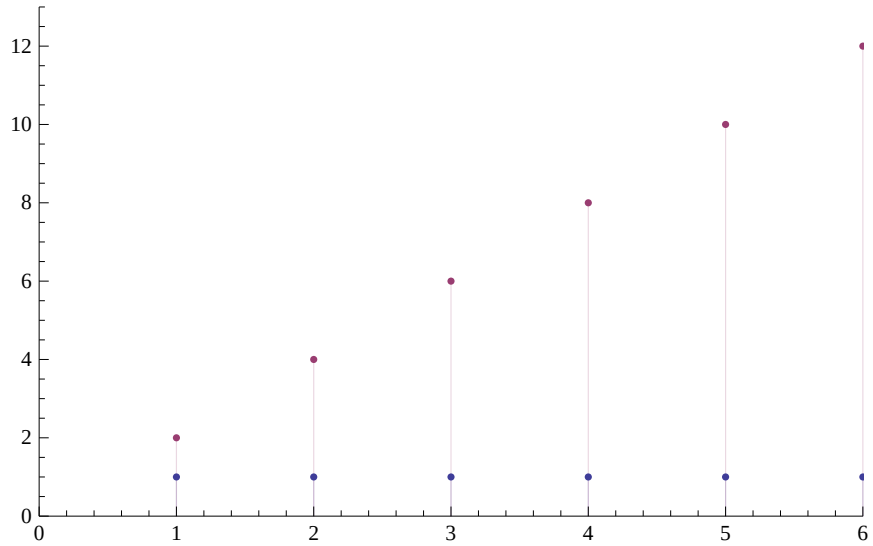
$$\begin{aligned}&\textbf{proc}\ () \to \textbf{do}\\ &\quad e_1 \leftarrow \mathit{repeatedly}\ 1\ 1 \prec ()\\ &\quad e_2 \leftarrow \mathit{repeatedly}\ 1.5\ 1 \prec ()\\ &\quad \mathit{returnA} \prec e_1\ \text{`}\mathit{merge}\text{`}\ e_2\end{aligned}$$

$$\mathit{repeatedly}\ 1\ (\lambda m \to m+2)$$

$$\begin{aligned}&\mathit{countBy2} = \textbf{proc}\ () \to \textbf{do}\\ &\quad e \leftarrow \mathit{repeatedly}\ 1\ (\lambda m \to m+2) \prec ()\\ &\quad \mathit{accum}\ 0 \prec e\end{aligned}$$

Nettle: Events

$repeatedly\ 1\ 1$



proc $() \rightarrow$ **do**
  $e \leftarrow repeatedly\ 1\ 1 \prec ()$
  $returnA \prec [n + 1 \mid n \leftarrow e]$

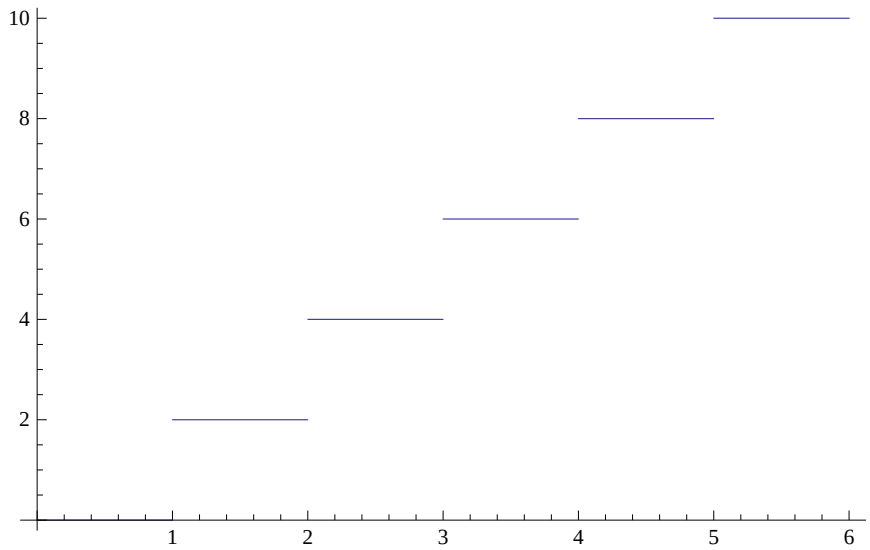
proc $() \rightarrow$ **do**
  $e_1 \leftarrow repeatedly\ 1\ 1 \prec ()$
  $e_2 \leftarrow repeatedly\ 1.5\ 1 \prec ()$
  $returnA \prec e_1\ `merge`\ e_2$

$repeatedly\ 1\ (\lambda m \rightarrow m + 2)$

$countBy2 =$ **proc** $() \rightarrow$ **do**
  $e \leftarrow repeatedly\ 1\ (\lambda m \rightarrow m + 2) \prec ()$
  $accum\ 0 \prec e$

Nettle: Signals to Events, Events to Signals

```
proc () → do
    c ← countBy2 ⤙ ()
    hold 0 ⤙ c
```



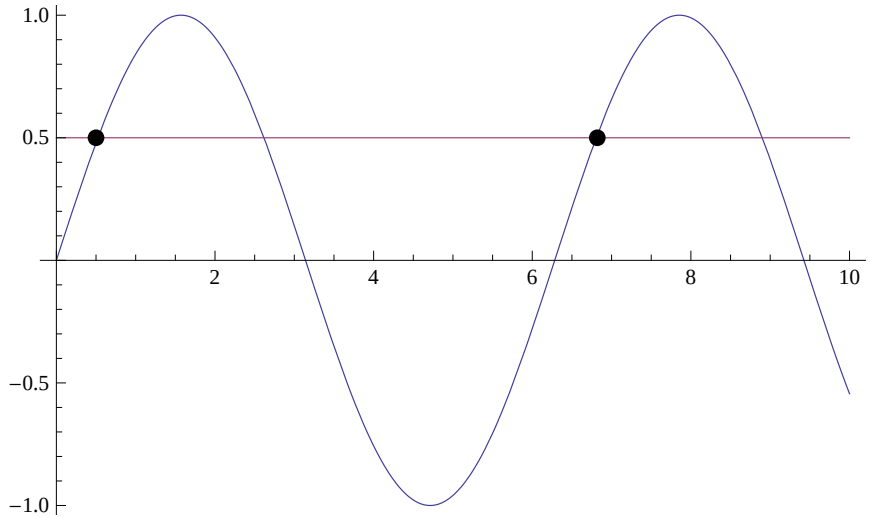

```
proc () → do
    t ← time ⤙ ()
    edge ⤙ (sin t > 0.5)
```

```
switch_ (proc () → do
            t ← time ⤙ ()
            e ← edge ⤙ t > 2
            returnA ⤙ (sin t, [constant 2 | _ ←- e])
```

Nettle: Signals to Events, Events to Signals

proc () → **do**
  $c \leftarrow countBy2 \prec ()$
  $hold\ 0 \prec c$

proc () → **do**
  $t \leftarrow time \prec ()$
  $edge \prec (sin\ t > 0.5)$

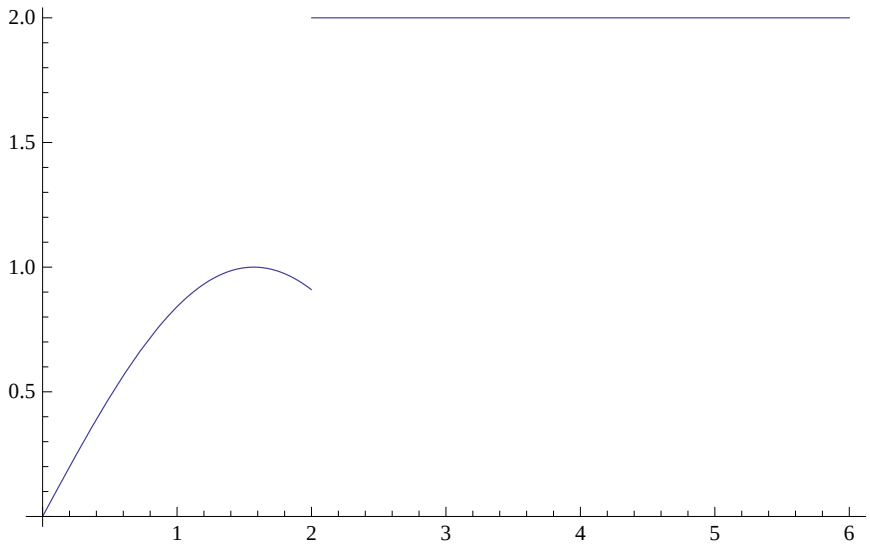


$switch\_$ (**proc** () → **do**
  $t \leftarrow time \prec ()$
  $e \leftarrow edge \prec t > 2$
  $returnA \prec (sin\ t, [constant\ 2 \mid \_ \leftarrow e])$

Nettle: Signals to Events, Events to Signals

```
proc () → do
  c ← countBy2 ⤙ ()
  hold 0 ⤙ c
```

```
proc () → do
  t ← time ⤙ ()
  edge ⤙ (sin t > 0.5)
```




```
switch_ (proc () → do
           t ← time ⤙ ()
           e ← edge ⤙ t > 2
           returnA ⤙ (sin t, [constant 2 | _ ← e])
```

Nettle: Network Control

A Nettle controller has signals and events as inputs and command events as output.

$$
\begin{aligned}
&\mathit{initSF} = \textbf{proc}\ \mathit{network} \rightarrow \textbf{do} \\
&\quad \mathit{returnA} \prec [\mathit{clearTables}\ \mathit{switchID} \mid \mathit{SwitchJoin}\ \mathit{switchID} \leftarrow \mathit{network}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{floodSF} = \textbf{proc}\ \mathit{network} \rightarrow \textbf{do} \\
&\quad \mathit{returnA} \prec [\mathit{send}\ p\ \mathit{flood} \mid \mathit{Packet}\ p \leftarrow \mathit{network}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{controller} = \textbf{proc}\ \mathit{network} \rightarrow \textbf{do} \\
&\quad \mathit{clear} \leftarrow \mathit{initSF} \prec \mathit{network} \\
&\quad \mathit{flood} \leftarrow \mathit{floodSF} \prec \mathit{network} \\
&\quad \mathit{returnA} \prec \mathit{merge}\ \mathit{clear}\ \mathit{flood}
\end{aligned}
$$

Nettle Network Control - Install Rules

$$
\begin{aligned}
&\mathit{rulesSF} = \textbf{proc}\ \mathit{network} \rightarrow \textbf{do} \\
&\quad \mathit{returnA} \prec [\mathit{insertRule}\ s\ (\mathit{anyPacket} \Longrightarrow \mathit{flood}) \\
&\qquad\qquad \mid \mathit{SwitchJoin}\ s \leftarrow \mathit{network}]
\end{aligned}
$$

Nettle Network Control - Learn Host Locations

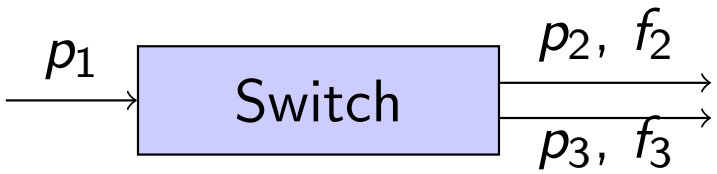
```
hostLocationsSF = proc network → do
  let keyVals = [((switch p, source p), port p) | Packet p ← network]
  accumMap ⤙ keyVals

rulesSF = proc network → do
  hostLocs ← hostLocationsSF ⤙ network
  returnA ⤙ [ rule p sp dp
            | Packet p ← network,
              sp ← maybeToEvent (lookup (source p) hostLocs)
              dp ← maybeToEvent (lookup (dest p) hostLocs)]
  where rule p sp dp
      = insertRule
            (switch p)
            ( inPortIs sp ∧ sourceIs (source p) ∧ destIs (dest p)
              ⟹ sendOnPort dp)
```

Nettle Network Control - Learn Host Locations

$$
\begin{aligned}
&\mathit{controller} = \textbf{proc}\ \mathit{network} \rightarrow \textbf{do} \\
&\quad \mathit{inits} \leftarrow \mathit{initSF} \prec\!\!- \mathit{network} \\
&\quad \mathit{rules} \leftarrow \mathit{rulesSF} \prec\!\!- \mathit{network} \\
&\quad \mathit{floods} \leftarrow \mathit{floodSF} \prec\!\!- \mathit{network} \\
&\quad \mathit{returnA} \prec\!\!- \mathit{merges}\ [\mathit{inits}, \mathit{rules}, \mathit{floods}]
\end{aligned}
$$

Nettle: Mathematical Network Control



- Address split: $0 \leqslant u(t) \leqslant 1$; low addresses on on port 2, high on port 3.
- Change u proportional to error: $\dot{u} = ke$, whence

$$u(t) = k \int_0^t e(\tau)\mathrm{d}\tau + u_0$$

Strategy Expressed in Nettle

$$
\begin{aligned}
u = \textbf{proc}\ (f_2, f_3) \rightarrow \textbf{do}\\
\quad \textbf{let}\ \mathit{error} = 0.5 - f_2\ /\ (f_2 + f_3)\\
\quad i \leftarrow \mathit{integral} \prec \mathit{error}\\
\quad \mathit{returnA} \prec k * i + u_0
\end{aligned}
$$

Nettle Implementation

- Nettle has been implemented and tested with real OpenFlow switches.
- Our single-threaded, sequential server serves 60,000 flows per second on a 2.5 GHz Intel Core 2 Duo with 4GB memory.
- According to estimates (Casado et al), this should be adequate for networks of over 10^5 nodes.

Ongoing & Future Work: High Level Abstractions

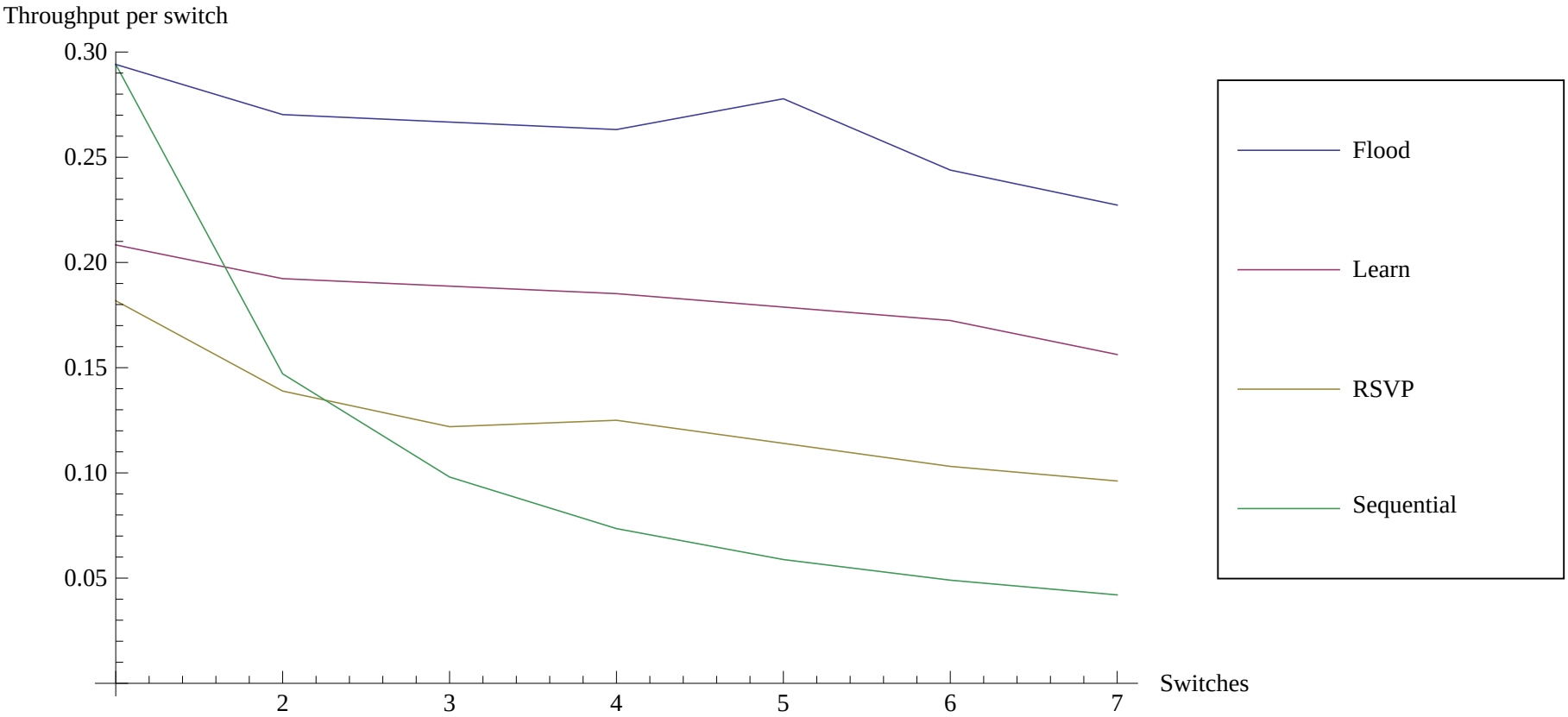
- Security
- Previous systems (Ethane, FSL) have provided a high-level, declarative language for static security policies, but don't allow dynamic security policies.
- Consider:
  "A user becomes banned when they exceed 5 day average bandwidth of 100Gb, at which point they may no longer use the network, until they are reinstated by an administrator, at which point the ban is lifted and their normal policy is applied."
- Goal: Integrate static policy rules with dynamic, reactive state in a single high-level declarative language.

Ongoing & Future Work: Scaling

- To scale to large networks, replicate controllers and distribute work.
- Fortunately, parallelism is plentiful: packets arriving from different hosts can generally be processed independently.
- But programming distributed systems is hard.
- Can we exploit this parallelism and keep simple programming model?
- Solution: use transactional memory; allows programmers to easily convert a sequential controller into a *correct, scaleable* concurrent controller by adding 'atomic' annotations.

Ongoing & Future Work: Scaling

Scaling results for some simple controllers:



Future Work: Interdomain Routing

Some problems with BGP:

- Policy language too limited.
- Bad interactions between IGP with BGP.

Solve this by designing a controller that:

- Speaks BGP to external peers (drop-in replacement).
- Allows a very flexible policy language (not restricted to BGP's decision process)
- Can be composed with IGP routing algorithms securely, i.e. impossible to misconfigure.

Build on the work on Routing Control Platform (RCP) (Caesar et al)

Conclusion

- SDN will help us build new systems, and reengineer old systems to make them more manageable.
- Nettle is a high-level declarative language for programming OpenFlow controllers.
- Future work will develop DSLs addressing high-level network concerns, and will provide tools for scaling controllers to large networks.

Thank you!