# Frenetic: Functional Reactive Programming for Networks

Nate Foster (Cornell)
Mike Freedman (Princeton)
Rob Harrison (Princeton)
Matthew Meola (Princeton)
Jennifer Rexford (Princeton)
David Walker (Princeton)

IBM PLDay 2010

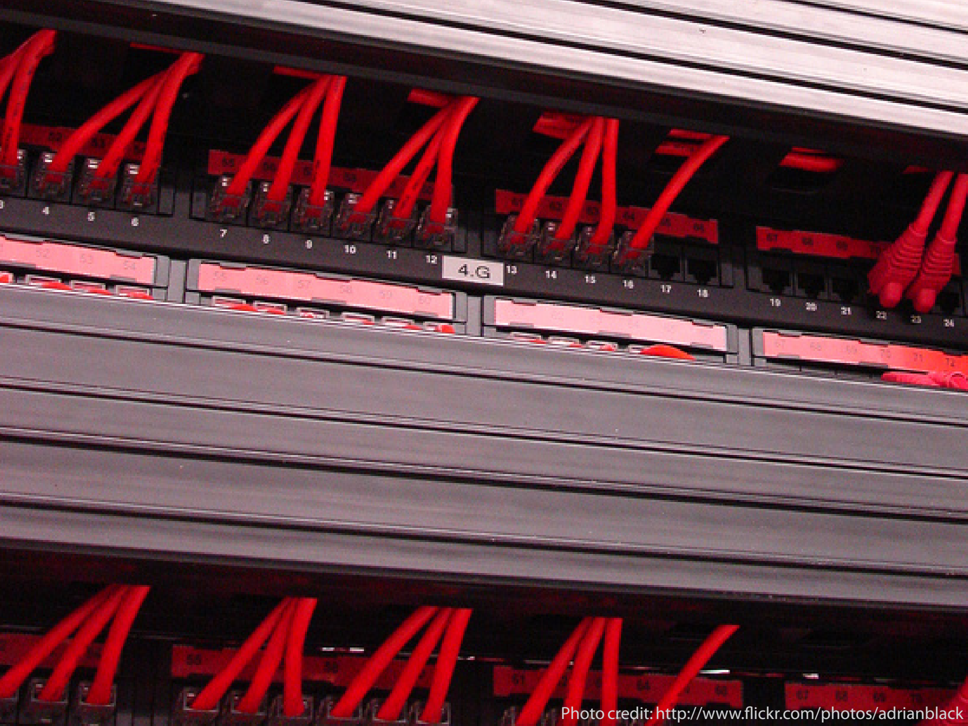



Photo credit: http://www.flickr.com/photos/adrianblack

# Why Programmable Networks?

## Security

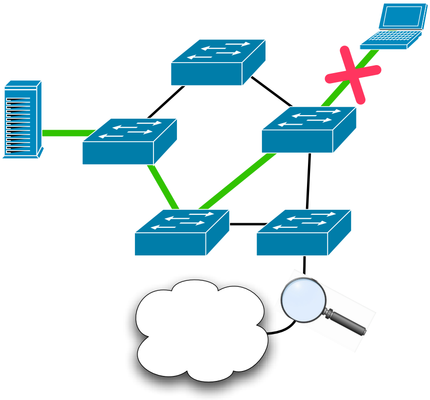
- Access control
- Traffic isolation

## Monitoring

- Usage / billing
- Anomaly detection

## Features

- Virtual Private Networks
- Content Distribution
- Resource Indirection
- Anycast



# Current State of Play

It's a mess! [Caldwell et al. '03, Oppenheimer et al. '03]

# Current State of Play

It's a mess! [Caldwell et al. '03, Oppenheimer et al. '03]

Configuration is vendor specific and complicated

Hodgepodge of mechanisms:

- OSPF / BGP for routing
- ACLs for security
- Netflow for monitoring

Operator errors common and costly

- Outages
- Degraded performance
- Security vulnerabilities

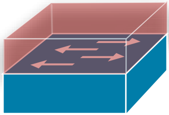
Configuration checkers and lint-like tools help a bit... but they are only a "band-aid", not a robust solution

# This Talk

1. OpenFlow
2. Examples
3. Frenetic
4. Implementation
5. Current and Ongoing work

# OpenFlow

# Traditional Switch



## Control Plane

- General-purpose hardware
- Runs (distributed) routing protocols
- Manipulates the forwarding table in the data plane

---

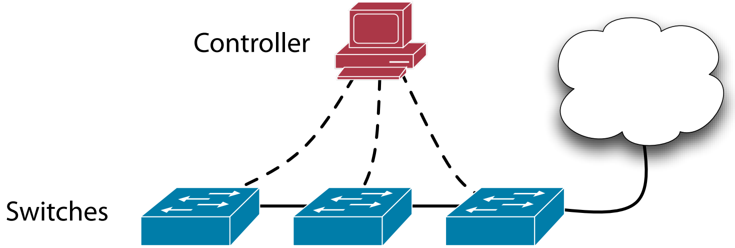
## Data Plane

- Special-purpose hardware
- Implements high-speed forwarding table
- Processes packets at line speed

# OpenFlow

## Key Ideas

- Move control from switch to a stock machine
- Standardize interface between switches and controller




http://www.openflowswitch.org/

# OpenFlow Switch

Switches process packets using rules described by:

- pattern – identify a set of packets
- priority – disambiguate rules with overlapping patterns
- actions – specify processing of packets
- counters – track number and size of packets processed

# OpenFlow Switch

Switches process packets using rules described by:

- pattern – identify a set of packets
- priority – disambiguate rules with overlapping patterns
- actions – specify processing of packets
- counters – track number and size of packets processed

## Example (OpenFlow Rules)

| Pattern | Priority | Actions | Counters |
|---|---|---|---|
| {in_port=2, trans_src=80} | HIGH | [ (OFPAT_OUTPUT, PORT_1) (OFPAT_OUTPUT, CONTROLLER) ] | (3,1455) |
| {in_port=2} | LOW | [ (OFPAT_OUTPUT, PORT_1) ] | (20,12480) |

# OpenFlow Controller

Controller runs a program that responds to events in the network by installing / uninstalling rules and collecting statistics from counters.

## Event Handlers

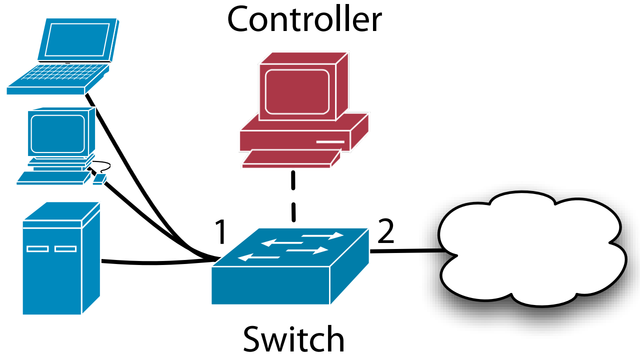
- switch_join(switch)
- switch_leave(switch)
- packet_in(switch, inport, packet)
- stats_in(switch, pattern, stats)

## Messages

- install(switch, pattern, priority, action)
- uninstall(switch, pattern)
- query_stats(switch, pattern)

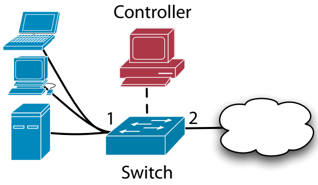
# Examples

# Topology



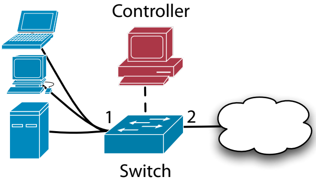
# Static Forwarding

```
def static_forwarding():
    # patterns
    p1 = {IN_PORT:1}
    p2 = {IN_PORT:2}
    # actions
    a1 = [(OFPAT_OUTPUT, PORT_2)]
    a2 = [(OFPAT_OUTPUT, PORT_1)]
    # install rules
    install(switch, p1, HIGH, a1)
    install(switch, p2, HIGH, a2)
```



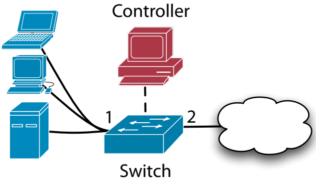
# Forwarding + Per-Host Monitoring

```
def static_forwarding_per_host_monitoring():
    # patterns
    p1 = {IN_PORT:1}
    p2 = {IN_PORT:2}
    # actions
    a1 = [(OFPAT_OUTPUT, PORT_2)]
    a2 = [(OFPAT_OUTPUT, CONTROLLER)]
    # install rules
    install(switch, p1, HIGH, a2)
    install(switch, p2, LOW, a2)
```



# Forwarding + Per-Host Monitoring

```
def packet_in(switch, inport, packet):
    # patterns
    p = {DL_DST:dstmac(packet)}
    pweb = {DL_DST:dstmac(packet), DL_TYPE:IP,
            NW_PROTO:TCP, TP_SRC:80}
    # action
    a = [(OFPAT_OUTPUT, PORT_1)]
    # install rules
    install(switch, pweb, HIGH, a)
    install(switch, p, MEDIUM, a)
    # query counters
    query_stats(switch, pweb)
```



# OpenFlow Limitations

## Low-level interface to switch hardware

- priorities used to disambiguate overlapping rules
- no support for negation
- wildcard vs. exact-match rules

## Two-tier programming model

- controller program manipulates rules
- asynchronous callbacks
- tricky race conditions

## Program pieces don't compose

- many programs decompose naturally into modules—e.g., forwarding + monitoring + access control
- but difficult to program in a compositional style because in general the rules manipulated by each module will overlap

# Frenetic

# Frenetic Ingredients

## High-level pattern algebra

- Hides details of how rules are implemented on switches
- Includes standard logical operators (e.g., negation)

## Unified programming model

- Programs “see every packet”
- Based on FRP $\rightarrow$ no asynchronous callbacks

## Fully compositional

- Programs can operate on overlapping subsets of the traffic
- Run-time system handles switch-level implementation details

# Frenetic Ingredients

## High-level pattern algebra

- Hides details of how rules are implemented on switches
- Includes standard logical operators (e.g., negation)

## Unified programming model

- Programs "see every packet"
- Based on FRP → no asynchronous callbacks

## Fully compositional

- Programs can operate on overlapping subsets of the traffic
- Run-time system handles switch-level implementation details

Main Challenge: having all these features without sacrificing performance.

# Frenetic Core

| | |
|---|---|
| E $\alpha$ | event stream carrying values of type $\alpha$ |
| EF $\alpha$ $\beta$ | operator that transforms an E $\alpha$ into an E $\beta$ |

| | | |
|---|---|---|
| Packets | $\in$ | E packet |
| Seconds | $\in$ | E int |
| Apply | $\in$ | (EF $a$ $b$ $\times$ E $a$) $\rightarrow$ E $b$ |
| Lift | $\in$ | ($a \rightarrow b$) $\rightarrow$ EF $a$ $b$ |
| \|O\| | $\in$ | EF $a$ $b$ $\rightarrow$ EF $b$ $c$ $\rightarrow$ EF $a$ $c$ |
| First | $\in$ | EF $a$ $b$ $\rightarrow$ EF ($a \times c$) ($b \times c$) |
| Merge | $\in$ | (E $a$ $\times$ E $b$) $\rightarrow$ E ($a$ option $\times$ $b$ option) |
| LoopPre | $\in$ | ($c \times$ EF ($a \times c$) ($b \times c$)) $\rightarrow$ EF $a$ $b$ |
| Calm | $\in$ | EF $a$ $a$ |
| Filter | $\in$ | ($a \rightarrow$ bool) $\rightarrow$ EF $a$ $a$ |
| Group | $\in$ | ($a \rightarrow b$) $\rightarrow$ EF $a$ ($b \times$ E $a$) |
| Regroup | $\in$ | (($a \times a$) $\rightarrow$ bool) $\rightarrow$ EF ($b \times$ E $a$) ($b \times$ E $a$) |
| Ungroup | $\in$ | int option $\times$ ($b \times a \rightarrow b$) $\rightarrow b \rightarrow$ EF ($c \times$ E $a$) ($c \times b$) |

# Forwarding + Per-Host Monitoring

```
# sum_sizes: (packet list) -> int
def sum_sizes(l):
    return (reduce(lambda n,p:n + size(p),l,0))

# per_host_monitoring_ef: EF packet (mac * int)
def per_host_monitoring_ef():
    return (Filter(inport_fp(2) & srcport_fp(80)) |O|        # E packet
            Group(dstmac_gp()) |O|                            # E (mac * E packet)
            ReGroupByTime(30) |O|                             # E (mac * packet list)
            Lift(lambda (m,l):(m,sum_sizes(l))))              # E (mac * int)

# rules: (rule list)
rules = [Rule(inport_fp(1), [output(2)]),
         Rule(inport_fp(2), [output(1)])]

# main function
def per_host_monitoring():
    register_static(rules)
    stats = Apply(Packets(), per_host_monitoring_ef())
    print_stream(stats)
```

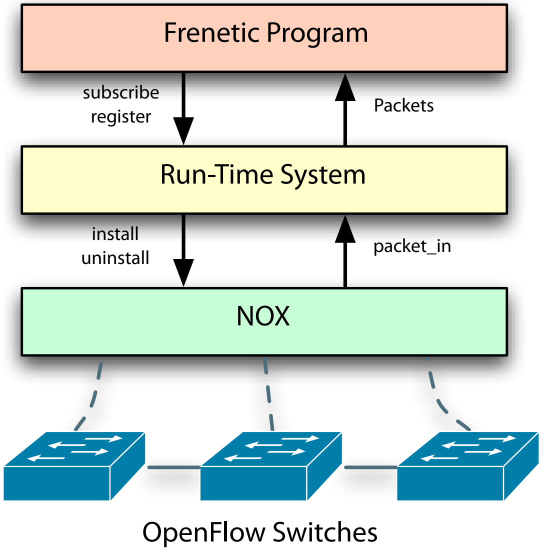
# Ethernet Learning

```
# add_rule: (mac * packet) * ((mac * rule) list) -> ((mac * rule) list) * ((mac * rule) list)
def add_rule(((m,p),t)): . . .

# complete_rules: ((mac * rule) list) -> (rule list)
def complete_rules(t): . . .

# learning_switch_ef: EF packet
def learning_switch_ef():
    return (Group(srcmac_gp()) |O|                      # E (mac * E packet)
            Regroup(inport_rf()) |O|                    # E (mac * E packet)
            Ungroup(1, lambda n,p:p, None) |O|          # E (mac * packet)
            LoopPre({}, Lift(add_rule)) |O|             # E ((mac * rule) list)
            Lift(complete_rules))                       # E (rule list)

# main function
def learning_switch():
    rules = Apply(Packets(), learning_switch_ef())
    register_stream(rules)
```

# Per-Host Monitoring + Learning

```
def per_host_monitoring_learning_switch():
    # ethernet learning
    rules = Apply(Packets(), learning_switch_ef())
    register_stream(rules)
    # per-host monitoring
    stats = Apply(Packets(), per_host_monitoring_ef())
    print_stream(stats)
```

# Implementation



# Implementation

## Push-based FRP implementation

- Classic pull-based strategy is not a good fit for networks
- Frenetic implementation based on strategy developed in FrTime [Cooper and Krishnamurthi '06]

## Subscribe / Register Library

- Programs can subscribe to streams of packets, headers, ints
- They can also register packet-forwarding policies
- Semantics is fully compositional
- Run-time system manages switch-level rules, event handlers, etc.
- Two strategies: proactive (eager) and reactive (lazy)

# Current and Ongoing Work

## Surface Language

- Current prototype is implemented as a Python library
- We want a front end with convenient syntax, typechecker, etc.

## Algebraic Optimizer

- Key optimization is moving processing from controller to switches
- Currently programmers must transform programs by hand
- We want an optimizer that rewrites programs automatically

## Formal Semantics

- Want a framework for modeling network behavior
- Use to prove optimizations correct
- And to develop new constructs for manipulating traffic atomically

## Applications

- Application-level load balancing
- Isolation in multi-tenant networks

# Questions?




## Collaborators

Mike Freedman, Rob Harrison, Matt Meola,
Jen Rexford, Dave Walker

`http://www.cs.cornell.edu/~jnfoster`