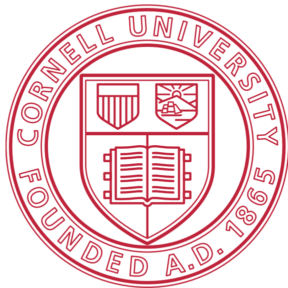
Frenetic: A Programming Language for OpenFlow Networks



Jennifer Rexford

Princeton University

http://www.frenetic-lang.org/

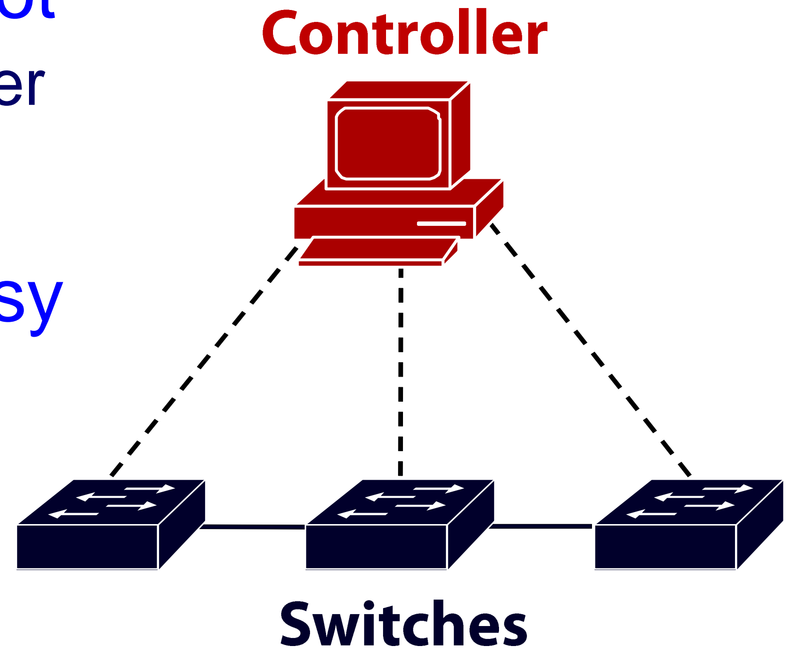
Joint work with Nate Foster, Dave Walker, Rob Harrison, Michael Freedman, Chris Monsanto, Mark Reitblatt, and Alec Story

Network Programming is Hard

- Programming network equipment is hard
  - Complex *software* by equipment vendors
  - Complex *configuration* by network administrators
- Expensive and error prone
  - Network outages and security vulnerabilities
  - Slow introduction of new features
- SDN gives us a chance to get this right!
  - Rethink abstractions for network programming

Programming OpenFlow Networks

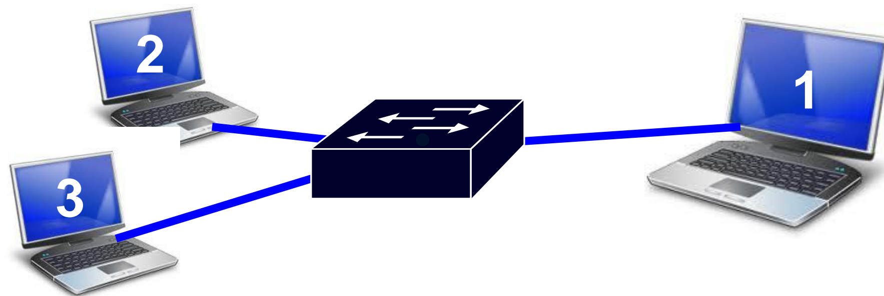
- OpenFlow already helps a lot
  - Network-wide view at controller
  - Direct control over data plane
- The APIs do *not* make it easy
  - Limited controller visibility
  - No support for composition
  - Asynchronous events



- Frenetic simplifies the programmer's life
  - A *language* that raises the level of abstraction
  - A *run-time system* that handles the gory details

Limited Controller Visibility

- Example: MAC-learning switch
  - Learn about new source MAC addresses
  - Forward to known destination MAC addresses
- Controller program is more complex than it seems
  - Cannot install *destination*-based forwarding rules
  - … without keeping controller from learning new *sources*



1 sends to 2 → learn 1, install
3 sends to 1 → never learn 3
1 sends to 3 → always floods

- Solution: rules on <inport, src MAC, dst MAC>

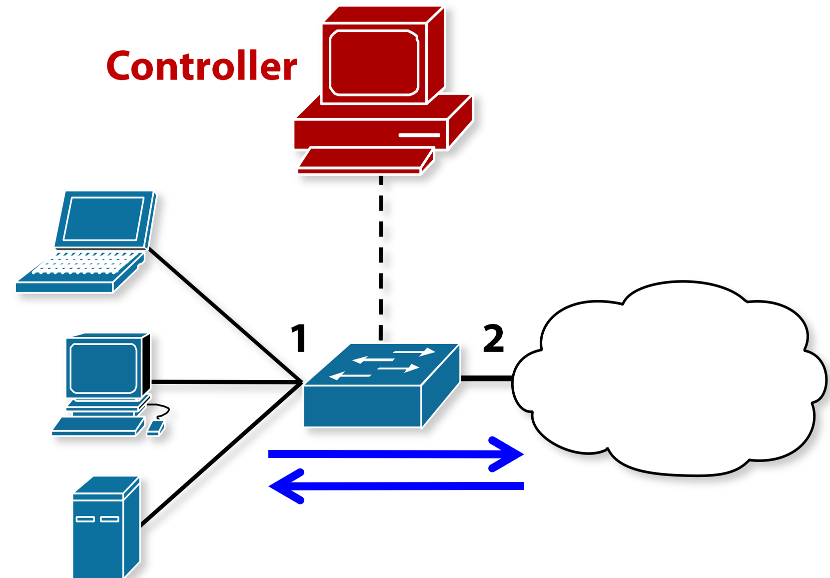
Must think about *reading* and *writing* at the same time.

Composition: Simple Repeater

Simple Repeater

```
def switch_join(switch):
  # Repeat Port 1 to Port 2
  p1 = {in_port:1}
  a1 = [forward(2)]
  install(switch, p1, DEFAULT, a1)

  # Repeat Port 2 to Port 1
  p2 = {in_port:2}
  a2 = [forward(1)]
  install(switch, p2, DEFAULT, a2)
```



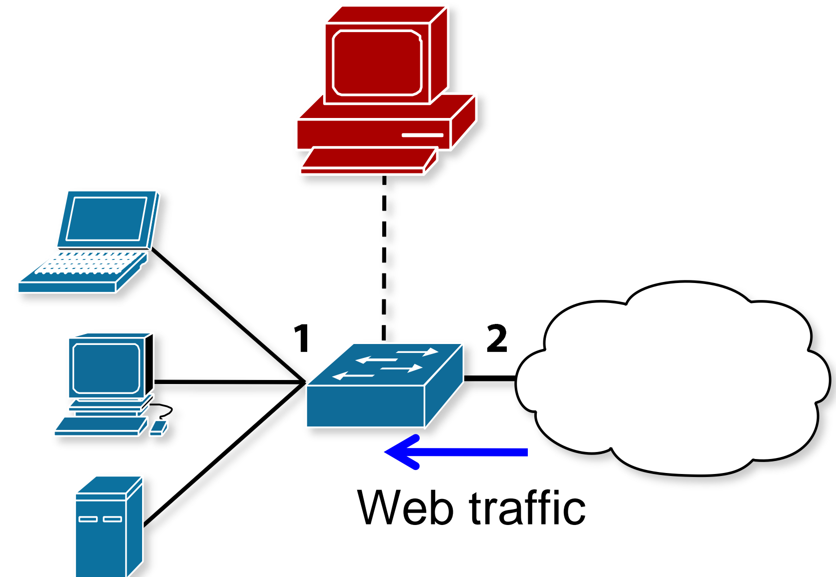
When a switch joins the network, install two forwarding rules.

Composition: Web Traffic Monitor

Monitor "port 80" traffic

```
def switch_join(switch)):
  # Web traffic from Internet
  p = {inport:2,tp_src:80}
  install(switch, p, DEFAULT, [])
  query_stats(switch, p)

def stats_in(switch, p, bytes, …)
  print bytes
  sleep(30)
  query_stats(switch, p)
```



When a switch joins the network, install one monitoring rule.

Composition: Repeater + Monitor

Repeater + Monitor

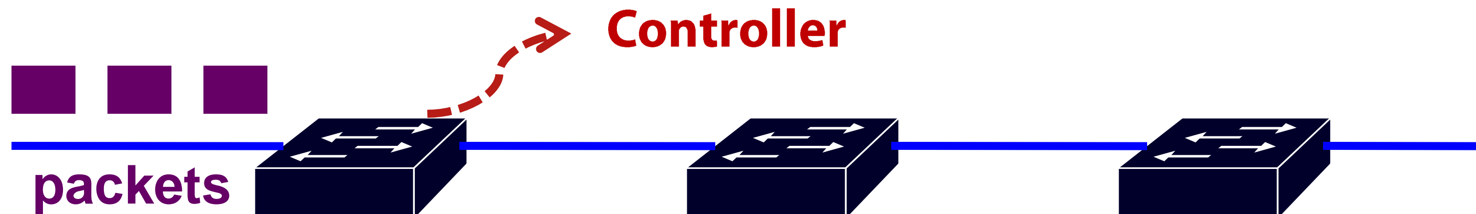
```
def switch_join(switch):
  pat1 = {inport:1}
  pat2 = {inport:2}
  pat2web = {in_port:2, tp_src:80}
  install(switch, pat1, DEFAULT, None, [forward(2)])
  install(switch, pat2web, HIGH, None, [forward(1)])
  install(switch, pat2, DEFAULT, None, [forward(1)])
  query_stats(switch, pat2web)

def stats_in(switch, xid, pattern, packets, bytes):
  print bytes
  sleep(30)
  query_stats(switch, pattern)
```

Must think about both tasks at the same time.

Asynchrony: Switch-Controller Delays

- Common OpenFlow programming idiom
  - First packet of a flow goes to the controller
  - Controller installs rules to handle remaining packets



- What if more packets arrive before rules installed?
  - Multiple packets of a flow reach the controller
- What if rules along a path installed out of order?
  - Packets reach intermediate switch before rules do

Must think about all possible event orderings.

Wouldn't It Be Nice if You Could…

- Separate reading from writing
  - Reading: specify queries on network state
  - Writing: specify forwarding policies
- Compose multiple tasks
  - Write each task once, and combine with others
- Prevent race conditions
  - Automatically apply forwarding policy to extra packets

This is what Frenetic does!

Our Solution: Frenetic Language

- **Reads**: query network state
  - Queries can see any packets
  - Queries do not affect forwarding
  - Language designed to keep packets in data plane
- **Writes**: specify a forwarding policy
  - Policy separate from mechanism for installing rules
  - Streams of packets, topology changes, statistics, etc.
  - Library to transform, split, merge, and filter streams
- Current implementation
  - A collection of Python libraries on top of NOX

Example: Repeater + Monitor

```
# Static repeating between ports 1 and 2
def repeater():
 rules=[Rule(inport:1, [forward(2)]),
        Rule(inport:2, [forward(1)])]
 register(rules)
```

Repeater

```
# Monitoring Web traffic
def web_monitor():
 q = (Select(bytes) *
      Where(inport:2 & tp_src:80) *
      Every(30))
 q >> Print()
```

Monitor

Repeater + Monitor

```
# Composition of two separate modules
def main():
 repeater()
 web_monitor()
```

Frenetic System Overview

- High-level language
  - Query language
  - Composition of forwarding policies
- Run-time system
  - Interprets queries and policies
  - Installs rules and tracks statistics
  - Handles asynchronous events

Frenetic Program

subscribe
register

E {pkts,hdrs,stats}

Run-Time System

install
uninstall

packet_in

NOX

OpenFlow Switches

Frenetic Run-Time System

- Rule granularity
  - Microflow: exact header match
  - Wildcard: allow "don't care" fields
- Rule installation
  - Reactive: first packet goes to controller
  - Proactive: rules pushed to the switches
- Frenetic run-time system
  - Version 1.0: reactive microflow rules [ICFP'11]
  - Version 2.0: proactive wildcard rules [POPL'12]
- Get it right once, and free the programmer!

Evaluation

- Example applications
  - **Routing**: spanning tree, shortest path
  - **Discovery**: DHCP and ARP servers
  - **Load balancing**: Web requests, memcached queries
  - **Security**: network scan detector, DDoS defenses
- Performance metrics
  - **Language**: lines of code in applications
    - Much shorter programs, especially when composing
  - **Run-time system**: overhead on the controller
    - Frenetic 1.0: competitive with programs running on NOX
    - Frenetic 2.0: fewer packets to the controller

Ongoing and Future Work

- Consistent writes [HotNets'11]
  - Transition from one forwarding policy to another
  - Without worrying about the intermediate steps
- Network virtualization
  - Multiple programs controlling multiple virtual networks
- Network-wide abstractions
  - Path sets, traffic matrices, reachability policy, etc.
- Joint host and network management
  - Controller managing the hosts and the switches
- Concurrent and distributed controllers

Conclusions

- Frenetic foundation
  - Separating reading from writing
  - Explicit query subscription and policy registration
  - Operators that transform heterogeneous streams
- Makes programming easier
  - Higher-level patterns
  - Policy decoupled from mechanism
  - Composition of modules
  - Prevention of race conditions
- And makes new abstractions easier to build!

The Frenetic Team

Nate Foster

Mike Freedman

Rob Harrison

Mark Reittblatt

Chris Monsanto

Jen Rexford

Alec Story

Dave Walker

Thanks!

frenetic >>

http://www.frenetic-lang.org/