Frenetic: A High-Level Language for OpenFlow Networks

Nate Foster, **Rob Harrison**,
Matthew L. Meola, Michael J. Freedman,
Jennifer Rexford, David Walker

PRESTO 2010,
Philadelphia, PA

11.28.2010

Background

OpenFlow/NOX allowed us to take back the network

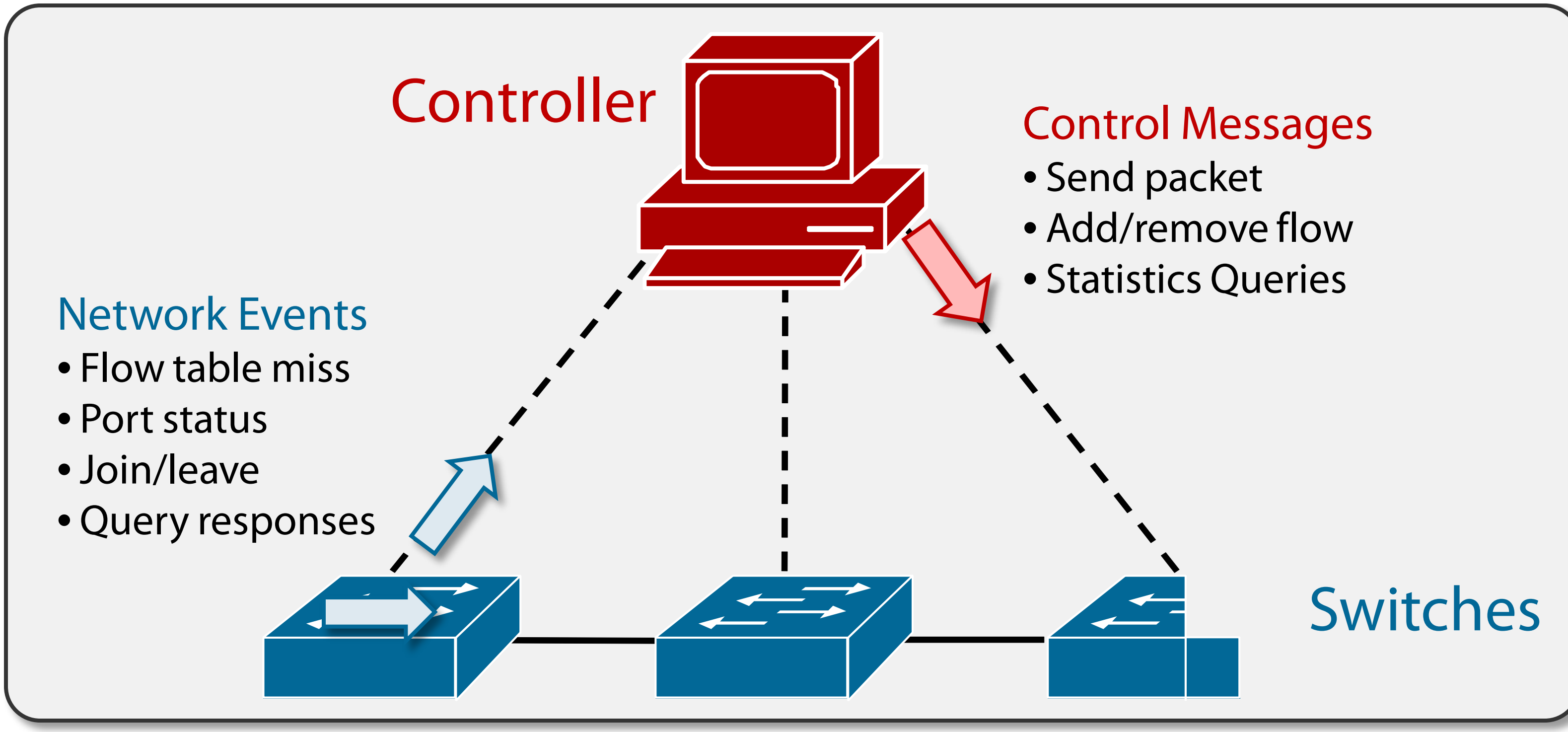
- Direct access to dataplane hardware
- Programmable control plane via open API

OpenFlow/NOX made innovation possible, not easy

- Low level interface mirrors hardware
- Thin layer of abstraction
- Few built-in features

So let's give the network programmer some help…

OpenFlow Architecture

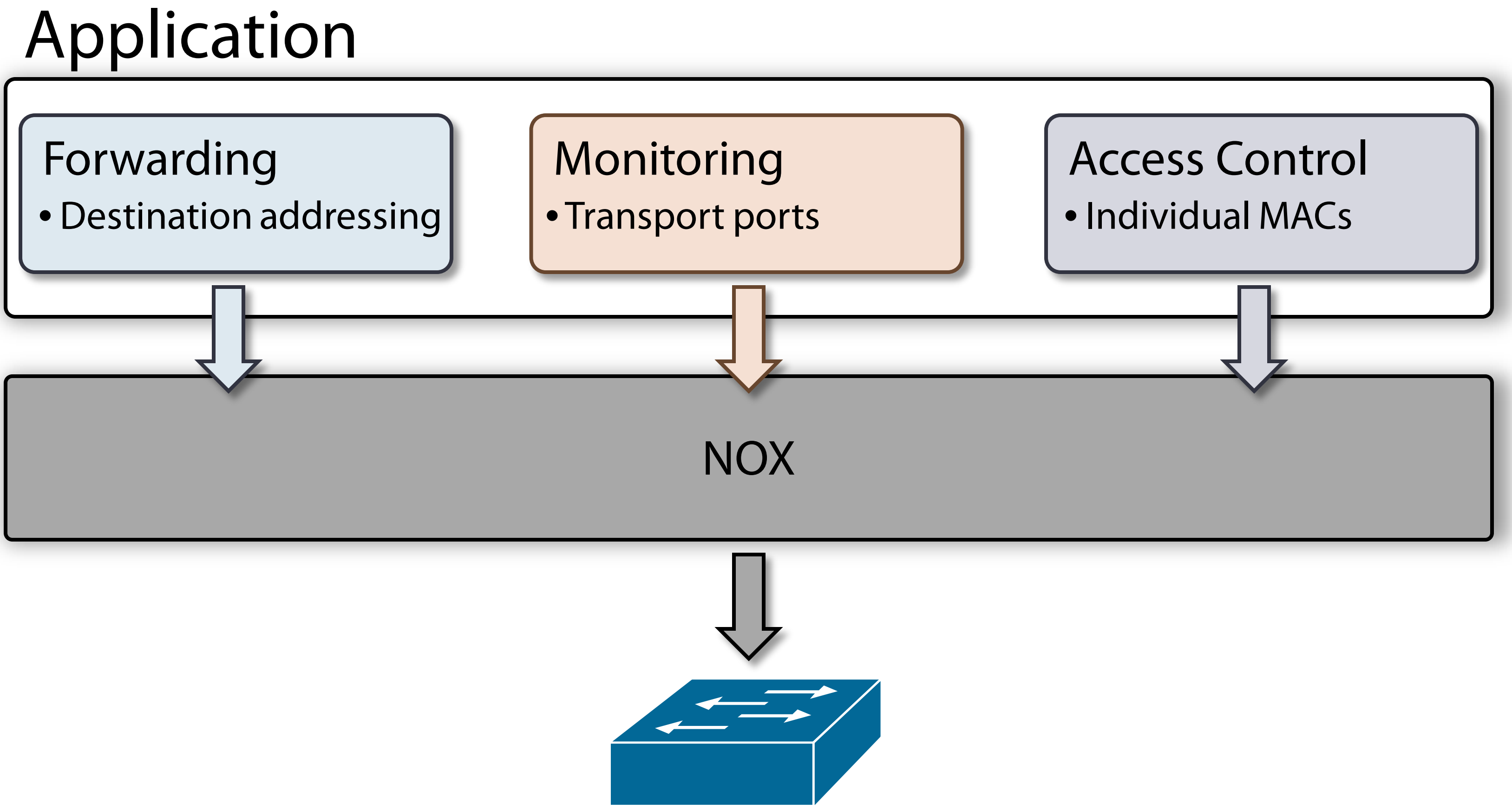


OpenFlow Switch Flow Table

Priority	Pattern	Action	Counters

`0-65535`	`Physical Port, Link Source/Destination/Type, VLAN, Network Source/Destination/Type, Transport Source/Destination`	`Forward`
		`Modify`
		`Drop`
			`Bytes, Count`

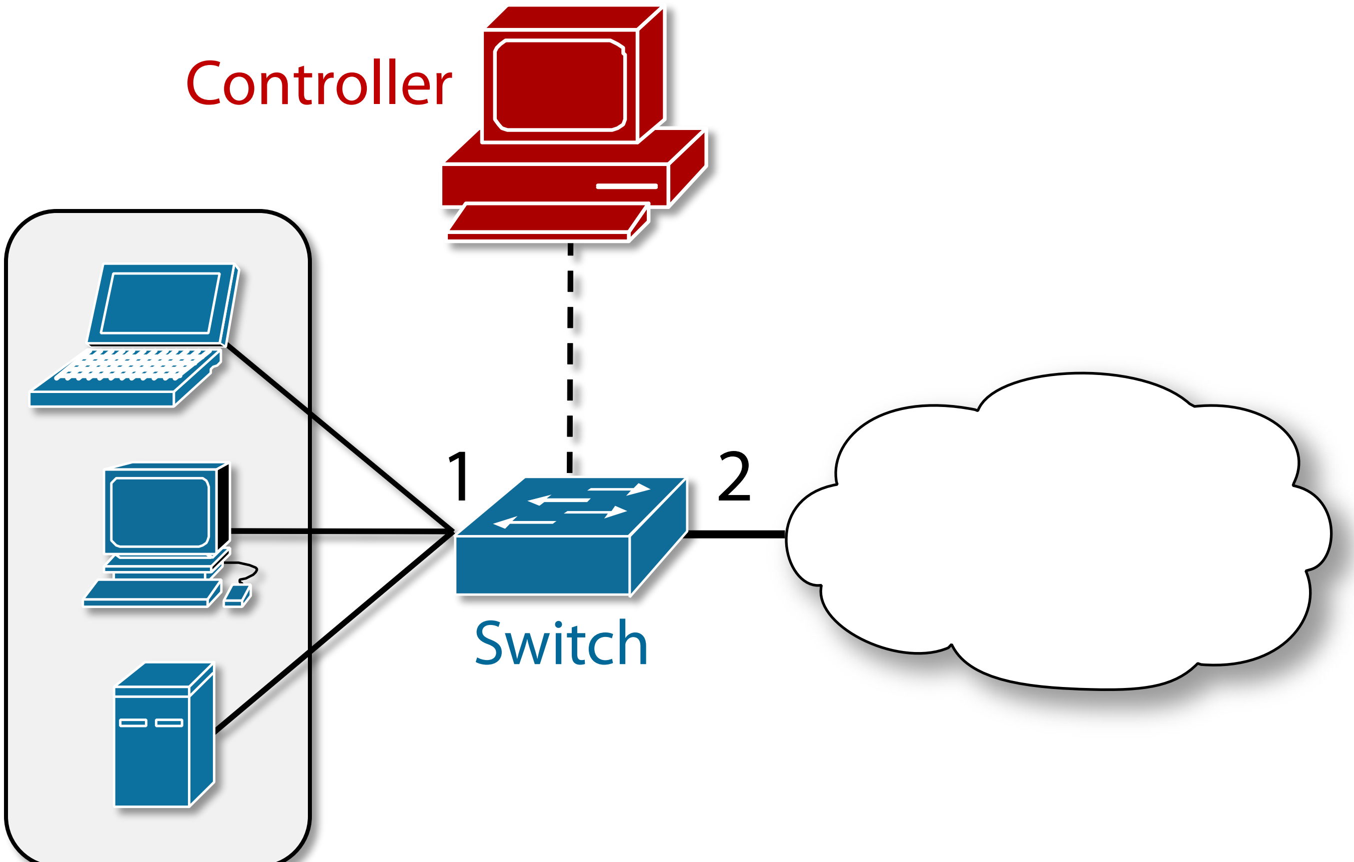
Programming Networks with NOX



In general, program modules do not *compose*

- If m yields r, and some m' yields r', then $(m \wedge m')$ does not yield $(r \wedge r')$

Example



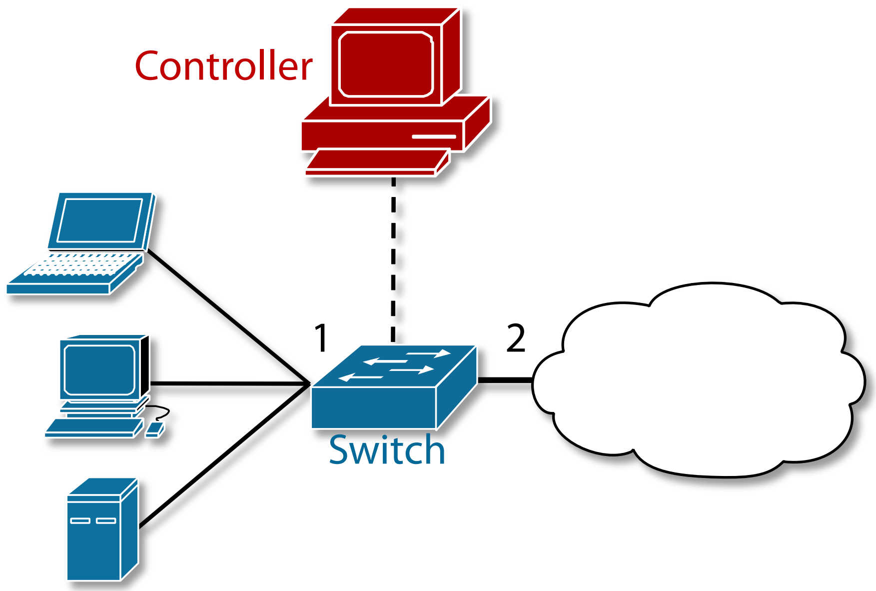
Simple Network Repeater

- Forward packets received on port 1 out 2; vice versa

Simple Repeater

NOX Program

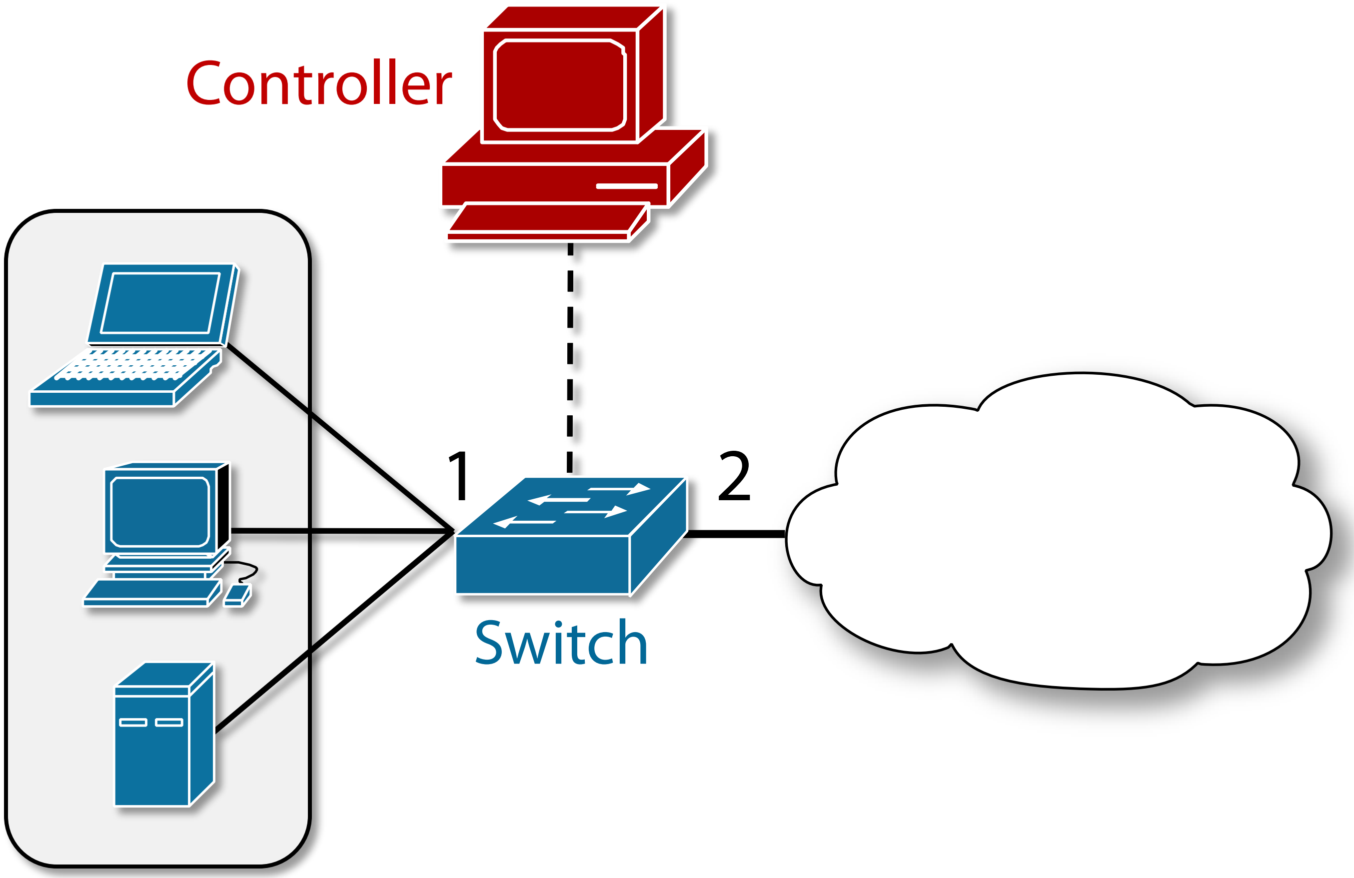
```
def simple_repeater():
  # Repeat Port 1 to Port 2
  p1 = {IN_PORT:1}
  a1 = [(OFPAT_OUTPUT, PORT_2)]
  install(switch, p1, HIGH, a1)

  # Repeat Port 2 to Port 1
  p2 = {IN_PORT:2}
  a2 = [(OFPAT_OUTPUT, PORT_1)]
  install(switch, p2, HIGH, a2)
```



Flow Table

Priority	Pattern	Action	Counters
HIGH	IN_PORT:1	OUTPUT:2	(0,0)
HIGH	IN_PORT:2	OUTPUT:1	(0,0)

Example



Simple Network Repeater with Host Monitoring

- Forward packets received on port 1 out 2; vice versa
- Monitor incoming HTTP traffic totals per host

Simple Repeater with Host Monitoring

```
# Repeat port 1 to 2
def port1_to_2():
  p1 = {IN_PORT:1}
  a1 = [(OFPAT_OUTPUT, PORT_2)]
  install(switch, p1, HIGH, a1)

# Callback to generate rules per host
def packet_in(switch, inport, pkt):
  p    = {DL_DST:dstmac(pkt)}
  pweb = {DL_DST:dstmac(pkt),
          DL_TYPE:IP,NW_PROTO:TCP,
          TP_SRC:80}
  a = [(OFPAT_OUTPUT, PORT_1)]
  install(switch, pweb, HIGH, a)
  install(switch, p, MEDIUM, a)

def main():
  register_callback(packet_in)
  port1_to_2()
```

```
def simple_repeater():
  # Port 1 to port 2
  p1 = {IN_PORT:1}
  a1 = [(OFPAT_OUTPUT, PORT_2)]
  install(switch, p1, HIGH, a1)

  # Port 2 to Port 1
  p2 = {IN_PORT:2}
  a2 = [(OFPAT_OUTPUT, PORT_1)]
  install(switch, p2, HIGH, a2)
```

Priority	Pattern	Action	Counters
HIGH	{IN_PORT:1}	OUTPUT:2	(0,0)
HIGH	{DL_DST:*mac*,DL_TYPE:IP_TYPE,NW_PROTO:TCP, TP_SRC:80}	OUTPUT:1	(0,0)
MEDIUM	{DL_DST:*mac*}	OUTPUT:1	(0,0)

OpenFlow/NOX Difficulties

Low-level, brittle rules

- No support for operations like union and intersection

Split architecture

- Between logic running on the switch and controller

No compositionality

- Manual refactoring of rules to compose subprograms

Asynchronous interactions

- Between switch and controller

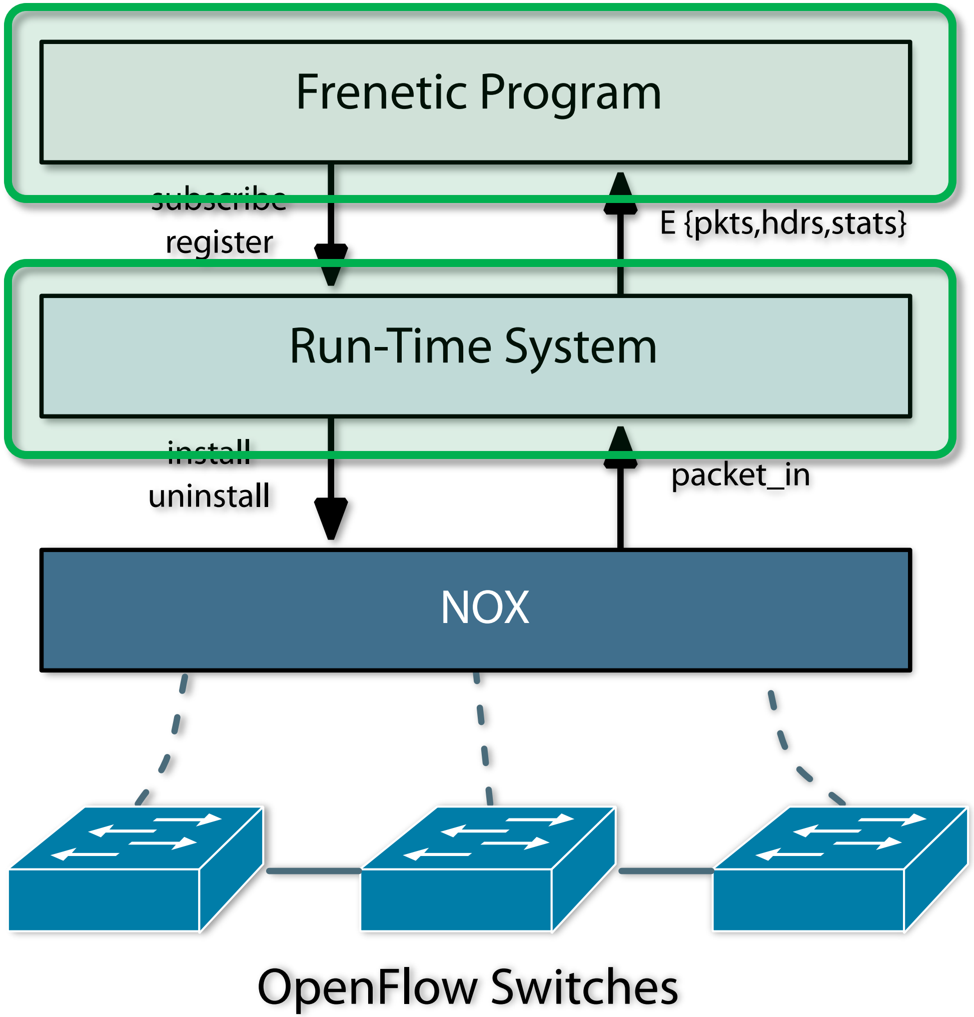
Our Solution: Frenetic

A High-level Language

- High-level patterns to describe flows
- Unified abstraction
- Composition

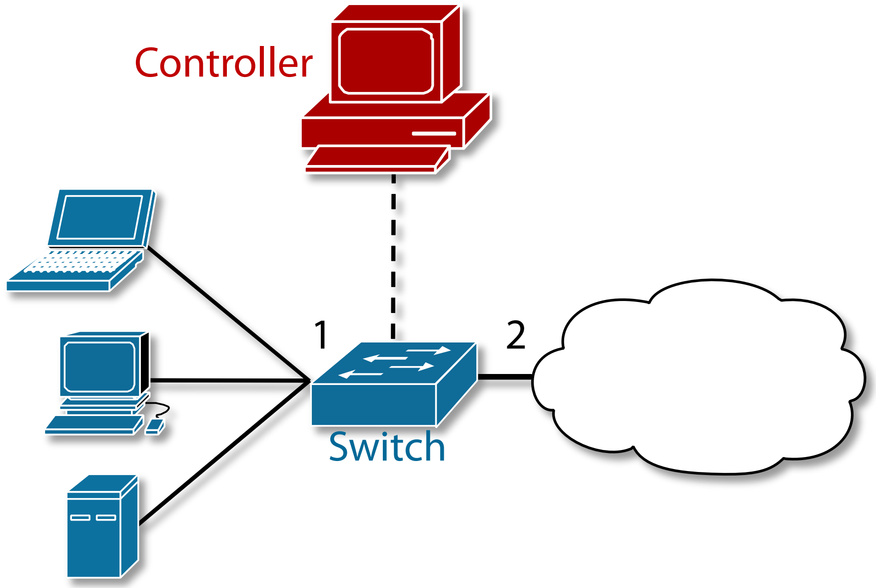
A Run-time System

- Handles module interactions
- Deals with asynchronous behavior



Frenetic Version

```
# Static repeating between ports 1 and 2
def simple_repeater():
 rules=[Rule(inport_fp(1), [output(2)]),
        Rule(inport_fp(2), [output(1)])]
 register_static(rules)

# per host monitoring es: E(int)
def per_host_monitoring():
 q = (Select(bytes) *
      Where(protocol(tcp) & srcport(80))*
      GroupBy([dstmac]) *
      Every(60))
 log = Print("HTTP Bytes:")
 q >> l

# Composition of two separate modules
def main():
 simple_repeater()
 per_host_monitoring()
```



Frenetic Version

```
# Static repeating between ports 1 and 2
def simple_repeater():
 rules=[Rule(inport_fp(1), [output(2)]),
        Rule(inport_fp(2), [output(1)])]
 register_static(rules)

# per host monitoring es: E(int)
def per_host_monitoring():
 q = (Select(bytes) *
      Where(protocol(tcp) & srcport(80))*
      GroupBy([dstmac]) *
      Every(60))
 log = Print("HTTP Bytes:")
 q >> l

# Composition of two separate modules
def main():
 simple_repeater()
 per_host_monitoring()
```

- No refactoring of rules
- Pure composition of modules
- Unified "see every packet" abstraction
- Run-time deals with the rest

Frenetic Language

Network as a stream of discrete, heterogenous events

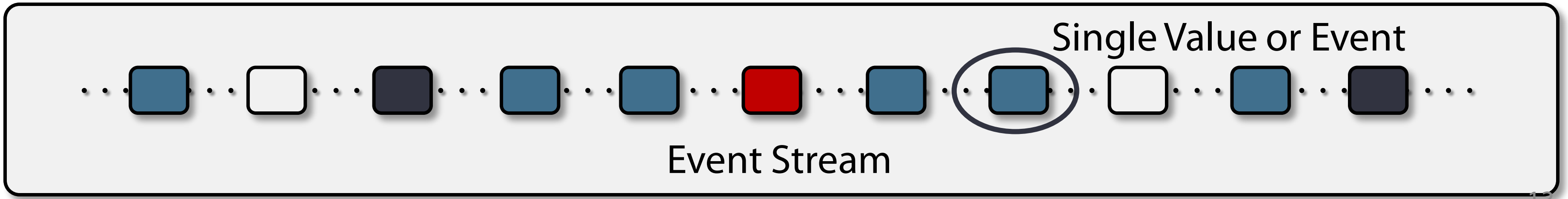
- Packets, node join, node leave, status change, time, etc…

Unified Abstraction

- "See every packet"
- Relieves programmer from reasoning about split architecture

Compositional Semantics

- Standard operators from Functional Reactive Programming (FRP)



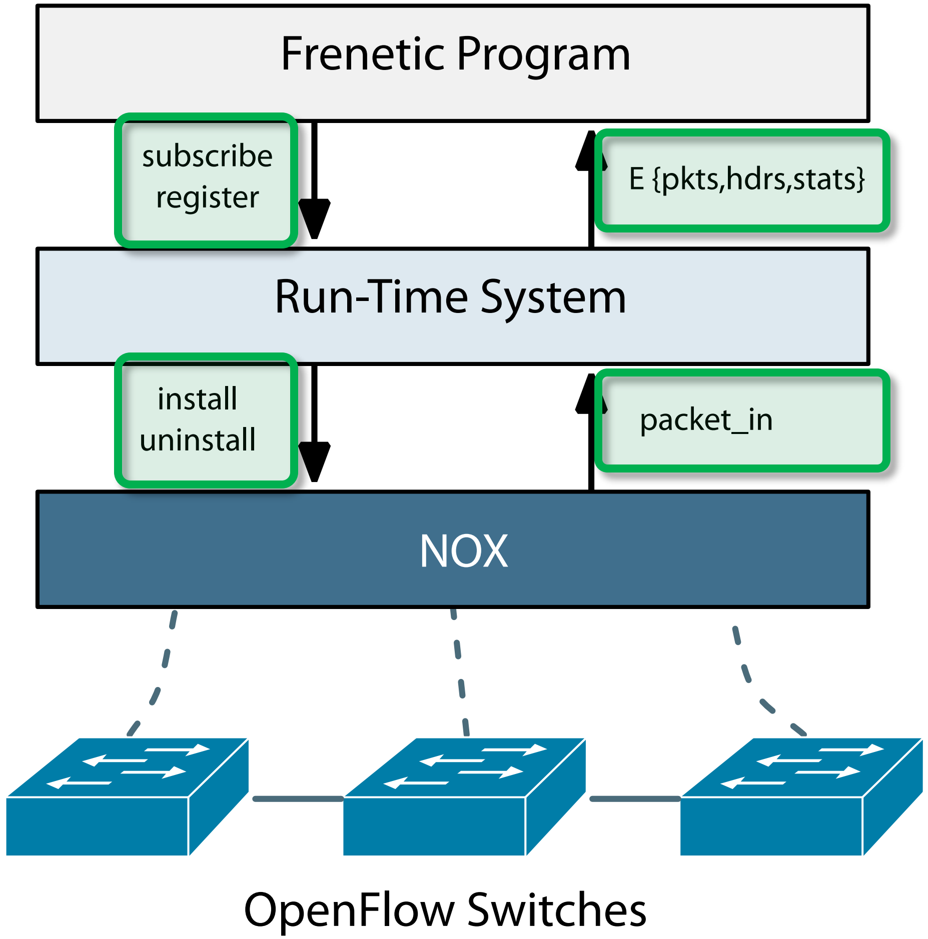
Frenetic Run-time System

Frenetic programs interact only with the run-time

- Programs create *subscribers*
- Programs *register* rules

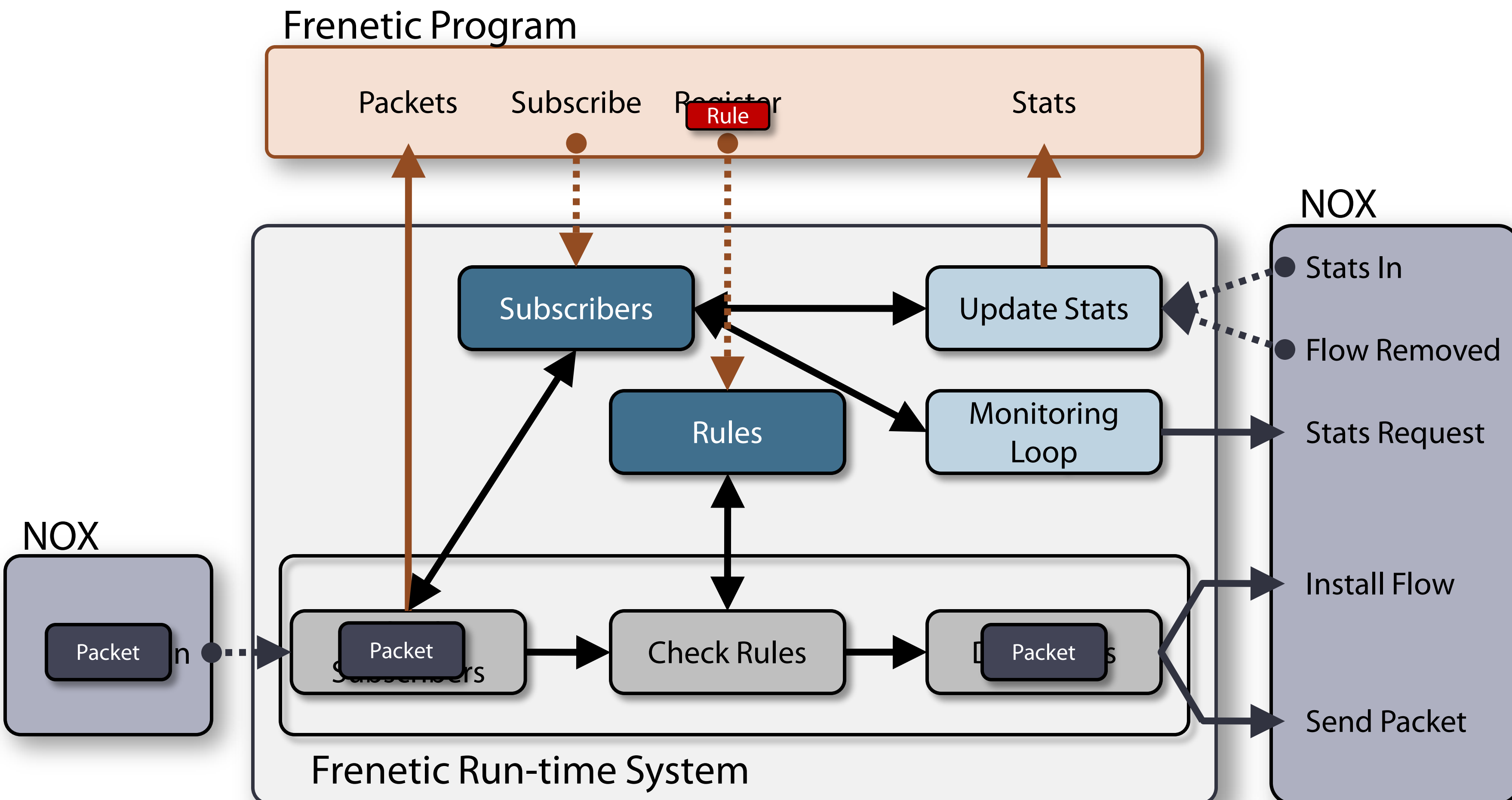
Run-time handles the details

- Manages switch-level rules
- Handles NOX events
- Pushes values onto the appropriate event streams



Run-time System Implementation

Reactive, microflow based run-time system



Optimizing Frenetic

"See every packet" abstraction can negatively affect performance in the worst case

- Naïve implementation strategy
- Application directed

Using an efficient combination of operators, we can keep packets in the dataplane

- Must match switch capabilities
  - Filtering, Grouping, Splitting, Aggregating, Limiting
- Expose this interface to the programmer explicitly

Does it Work in Practice?

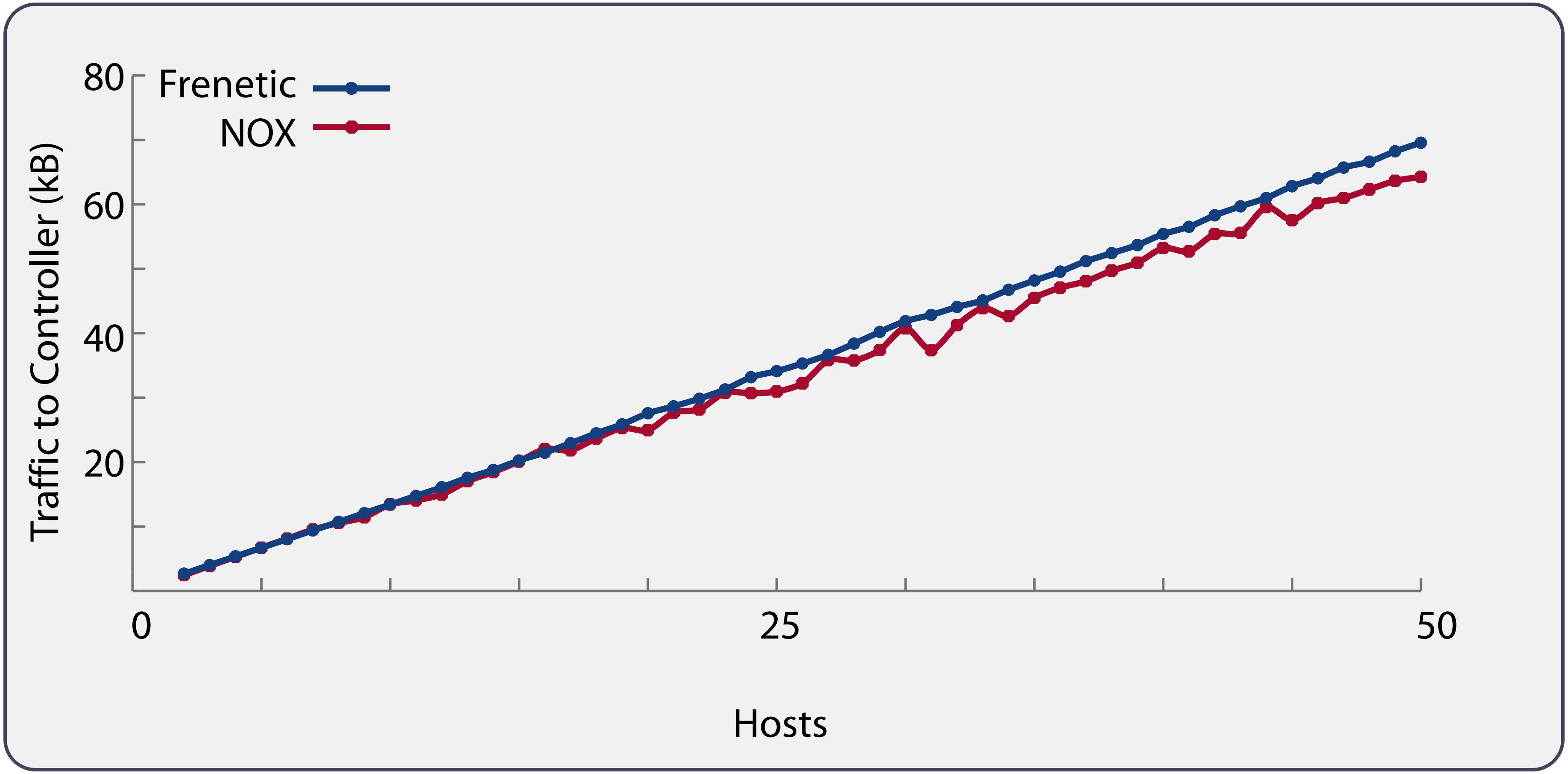
Frenetic programs perform comparably with pure NOX

- But we still have room for improvement

	Learning Switch	Web Stats Static	Web Stats Learning	Heavy Hitters Learning
Pure NOX				
Lines of Code	55	29	121	125
Traffic to Controller (Bytes)	71224	1932	5300	18010
Naïve Frenetic				
Lines of Code	15	7	19	36
Traffic to Controller (Bytes)	120104	6590	14075	95440
Optimized Frenetic				
Lines of Code	14	5	16	32
Traffic to Controller (Bytes)	70694	3912	5368	19360

Frenetic Scalability

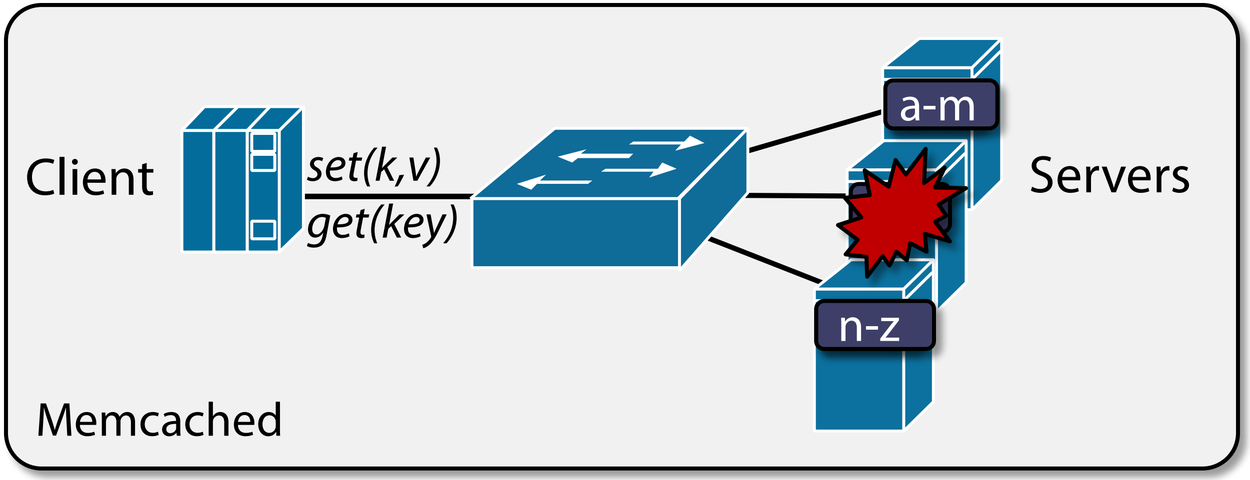
Frenetic scales to larger networks comparably with NOX



Larger Applications

Memcached with dynamic membership

- Forwards queries to a dynamic member set
- Works with unmodified memcached clients/servers



Defensive Network Switch

- Identifies hosts conducting network scanning
- Drops packets from suspected scanners

Ongoing and Future Work

Surface Language

- Current prototype is in Python – to ease transition
- Would like a standalone language

Optimizations

- More programs can also be implemented efficiently
- Would like a compiler to identify and rewrite optimizations

Proactive Strategy

- Current prototype is reactive, based on microflow rules
- Would like to enable proactive, wildcard rule installation

Network Wide Abstractions

- Current prototype focuses only on a single switch
- Need to expand to multiple switches

Questions?

See our recent submission for more details…

http://www.cs.cornell.edu/~jnfoster/papers/frenetic-draft.pdf