Frenetic: A High-Level Language for OpenFlow Networks

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

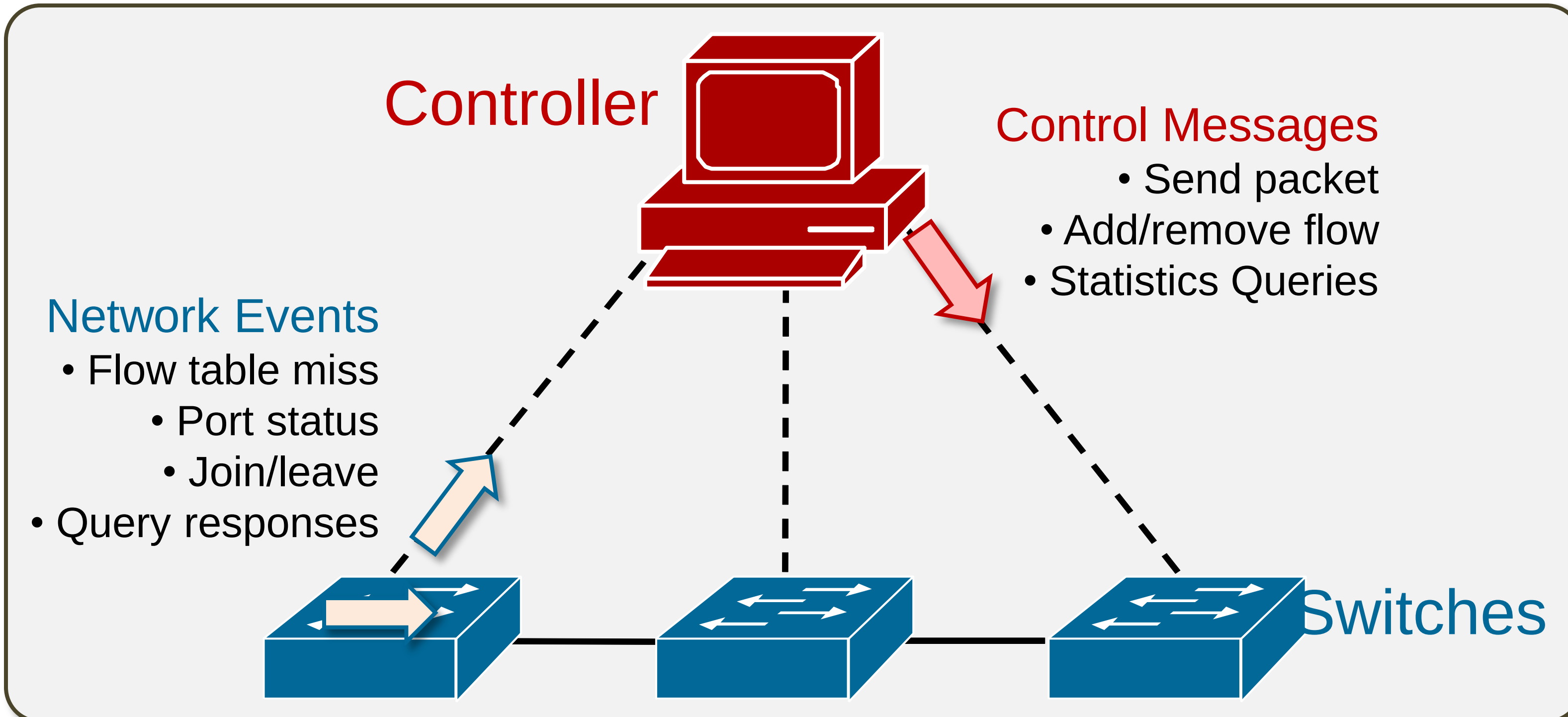
Present by Xiang Wang
Venus Team, NSLab
RIIT, Tsinghua Univ.

May. 08, 2012 @ NSLab Seminar

Background

- Chilling effect on innovation
  - Hardware and software are closed and proprietary
  - New requirements brought problems into sharp relief
- 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…

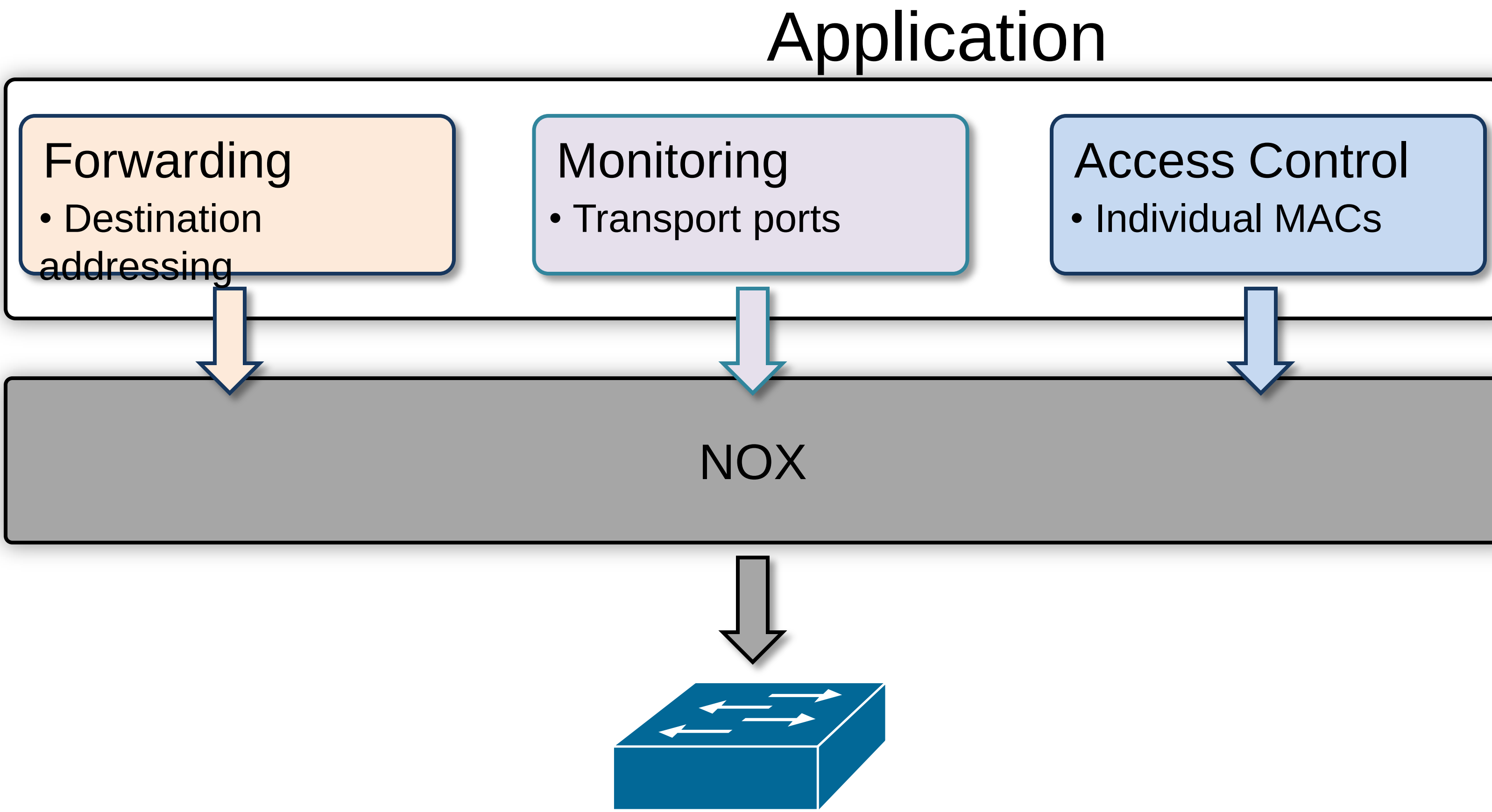
OpenFlow Architecture



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*

OpenFlow/NOX Difficulties

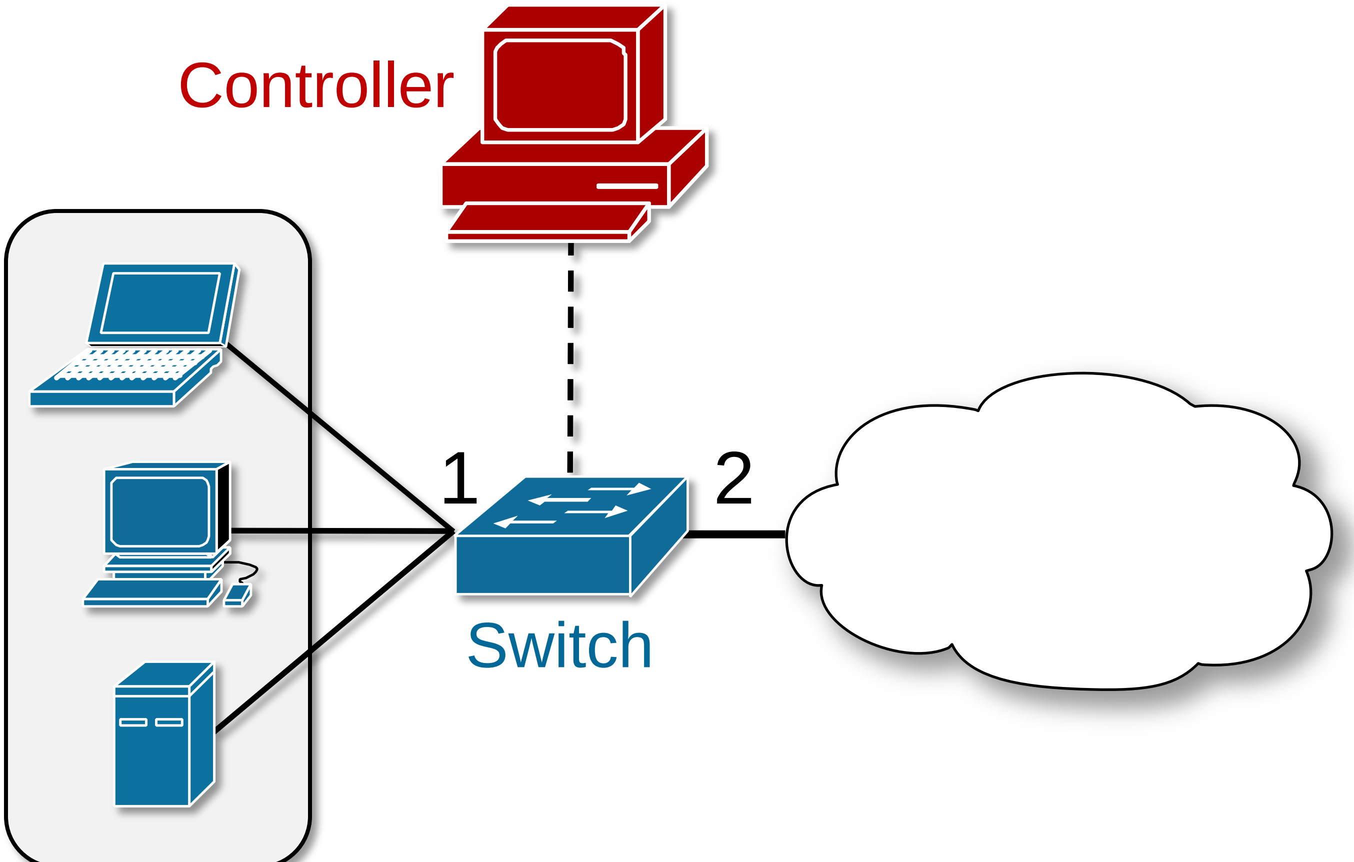
- Multiple task vs. independent modules
  - Routing, access control, traffic monitoring
  - Rules may be overlapping

- Low-level abstraction
  - No support for operations like union and intersection
  - Manual refactoring of rules to compose subprograms

- Split architecture
  - Between logic running on the switch and controller

- Asynchronous interactions
  - Between switch and controller
  - Various race conditions

Example

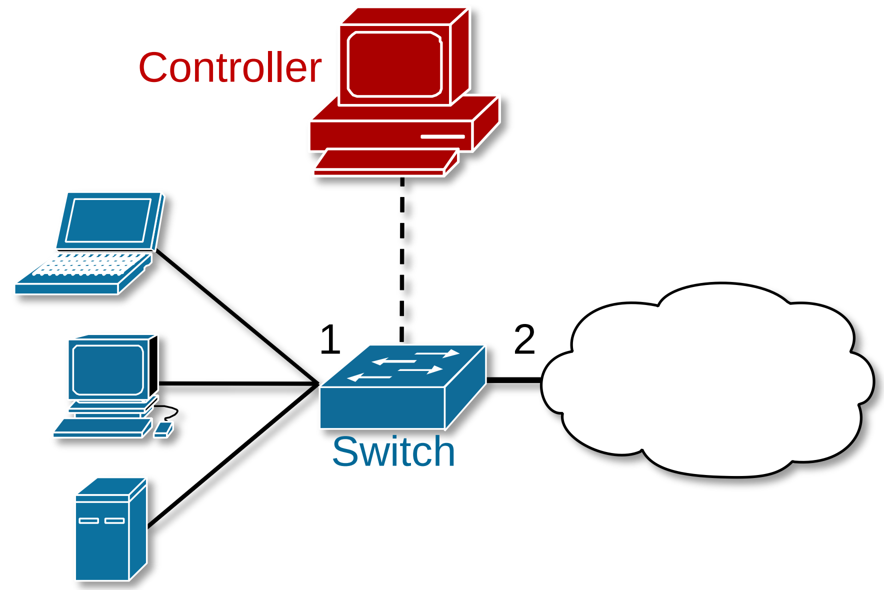


- Simple Network Repeater
  - Forward packets received on port 1 out 2; vice versa

Simple Repeater

NOX Program

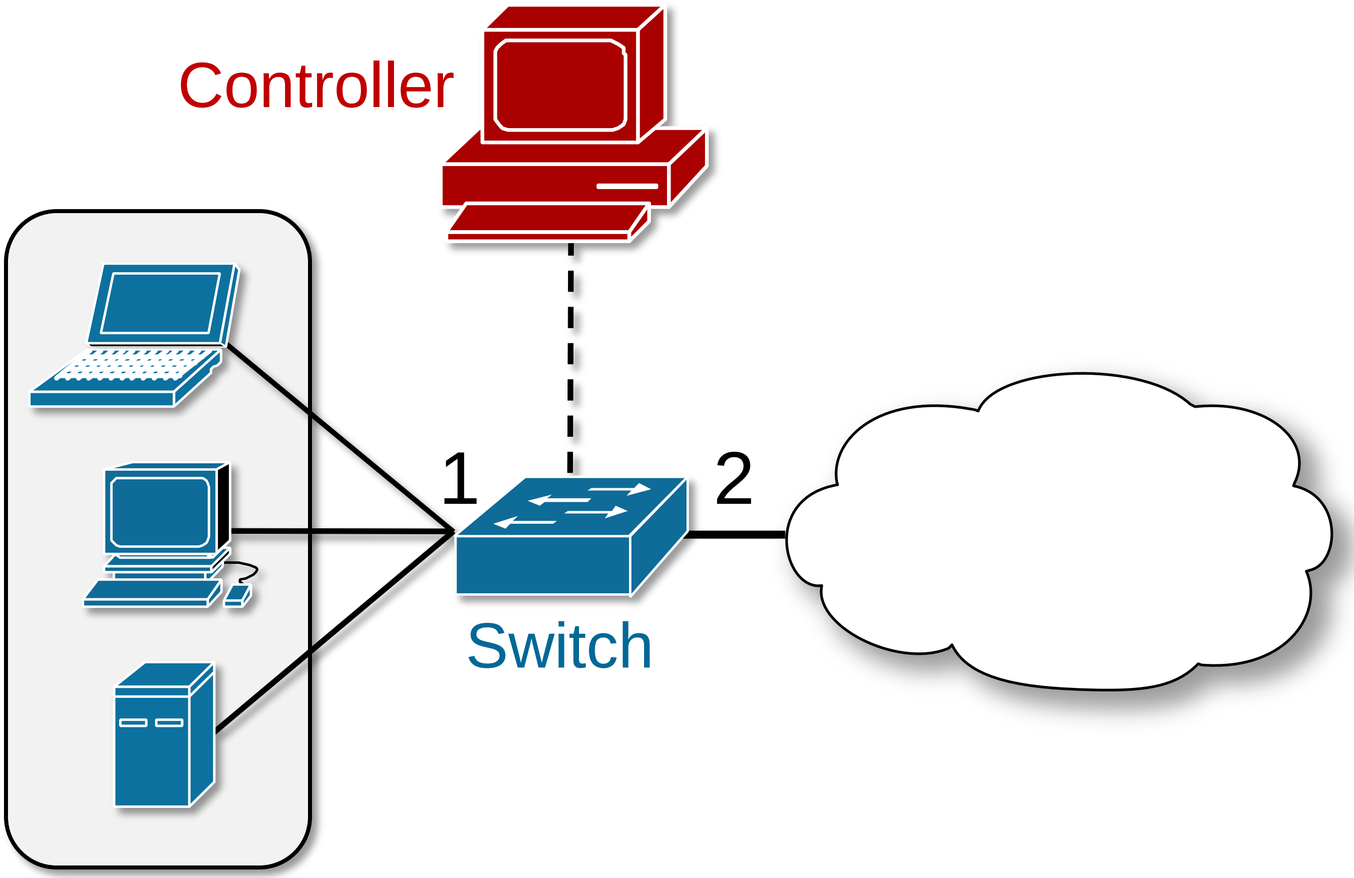
```
def simple_repeater():
  # Repeat Port 1 to Port 2
  p1 = {IN_PORT:1}
  a1 = [(OFPAT_OUTPUT, PORT_2)]
  install(switch, p1, HIGH, a1)

  # 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)

Interactions between modules



- Simple Network Repeater with Host Monitoring
  - Forward packets received on port 1 out 2; vice versa
  - Monitor incoming HTTP traffic totals per host

Interactions between modules

```
# 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)

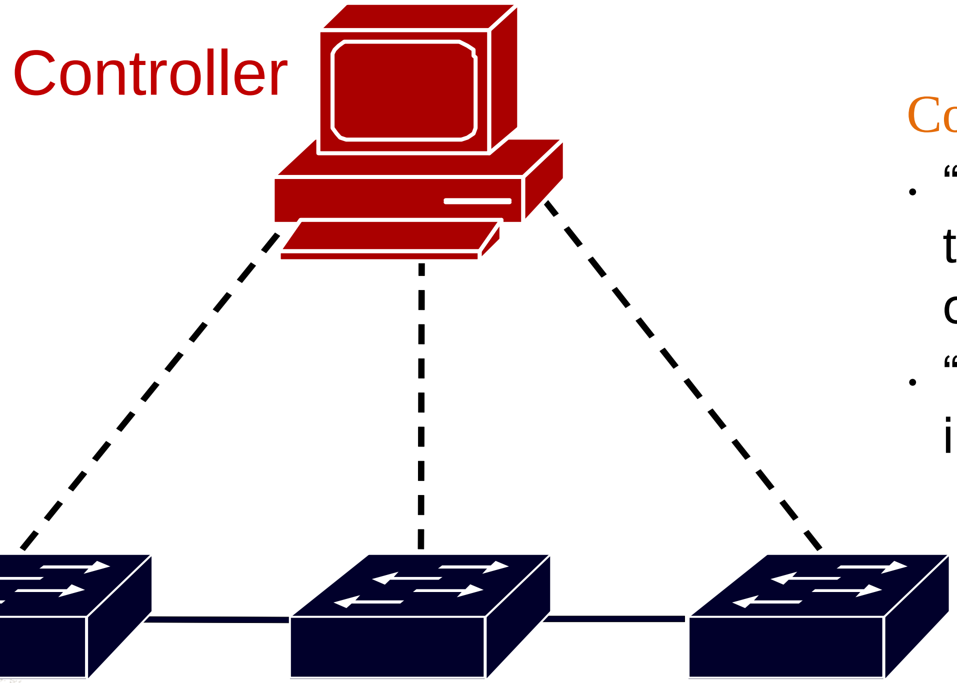
Low-level programming interface

Switch-level API:

- Derived from switch feature, rather than being designed for ease of use
- Multiple rules are manually adjusted with restrict priority

```
def repeater_monitor(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  pat2web = {in_port:2,tp_src:80}
  install(switch,pat1,[output(2)],DEFAULT)
  install(switch,pat2web,[output(1)],HIGH)
  install(switch,pat2,[output(1)],DEFAULT)
  query_stats(switch,pat2web)

def repeater_monitor_noserver(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  pat2web = {in_port:2,tp_src:80}
  pat2srv = {in_port:2,nw_dst:10.0.0.9,tp_src:80}
  install(switch,pat1,DEFAULT,None,[output(2)])
  install(switch,pat2srv,HIGH,None,[output(1)])
  install(switch,pat2web,MEDIUM,None,[output(1)])
  install(switch,pat2,DEFAULT,None,[output(1)])
  query_stats(switch,pat2web)
```

Two-tiered Programming Model



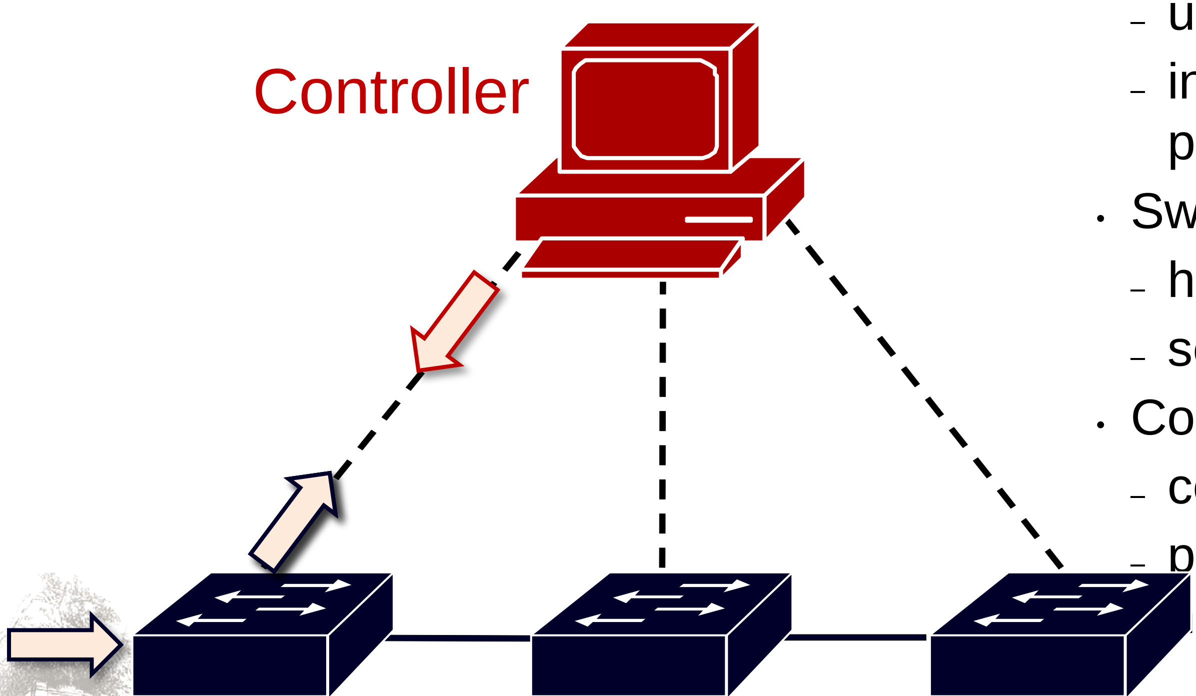
Tricky problem:

- Controller activity is driven by packets sent from switches
- Efficient applications install rules on switches to forward packets in hardware

Constant questions:

- "Is that packet going to come to the controller to trigger my computation?"
- "Or is it already being handled invisibly on the switch?"

Network Race Conditions



A challenging chain of events:

- Switch
  - sends packet to controller
- Controller
  - analyzes packet
  - updates its state
  - initiates installation of new packet-processing rules
- Switch
  - hasn’t received new rules
  - sends new packets to controller
- Controller
  - confused
  - packets in the same flow handled inconsistently

Problems – One Common Cause

- Problems:
  - Non-modular programming: Programs can't be divided into modules for monitoring and forwarding
  - Network race conditions: The controller sees more events (packets) than it anticipates
  - Two-tiered programming: Will the controller be able to see the appropriate events given the forward rules installed?

- One common cause:
  - No effective *abstractions* for *reading* network state

The Solution

- Separate network programming into two parts:
  - Abstractions for reading network state
    - Reads should have *no effect* on forwarding policy
    - Reads should be able to *see every packet*
  - Abstractions for specification of forwarding policy
    - Forwarding *policy* must be separated from *implementation mechanism*
- A natural decomposition that mirrors the two fundamental tasks of network management
  - Monitoring and forwarding

Key principles

- Declarative Design
  - Consider what the programmer might want
  - Rather than how the hardware implements it
- Modular Design
  - Limited network-wide effects primitives and semantics
  - Independently used in different contexts
- Single-tier Programming
  - Support see-every-packet abstration
  - Side-step many complexities of two-tiered model
- Race-free Semantics
  - Auto race detection
  - Packet suppression
- Cost Control
  - Gives programmers guidance concerning the costs
  - Query language is carefully defined

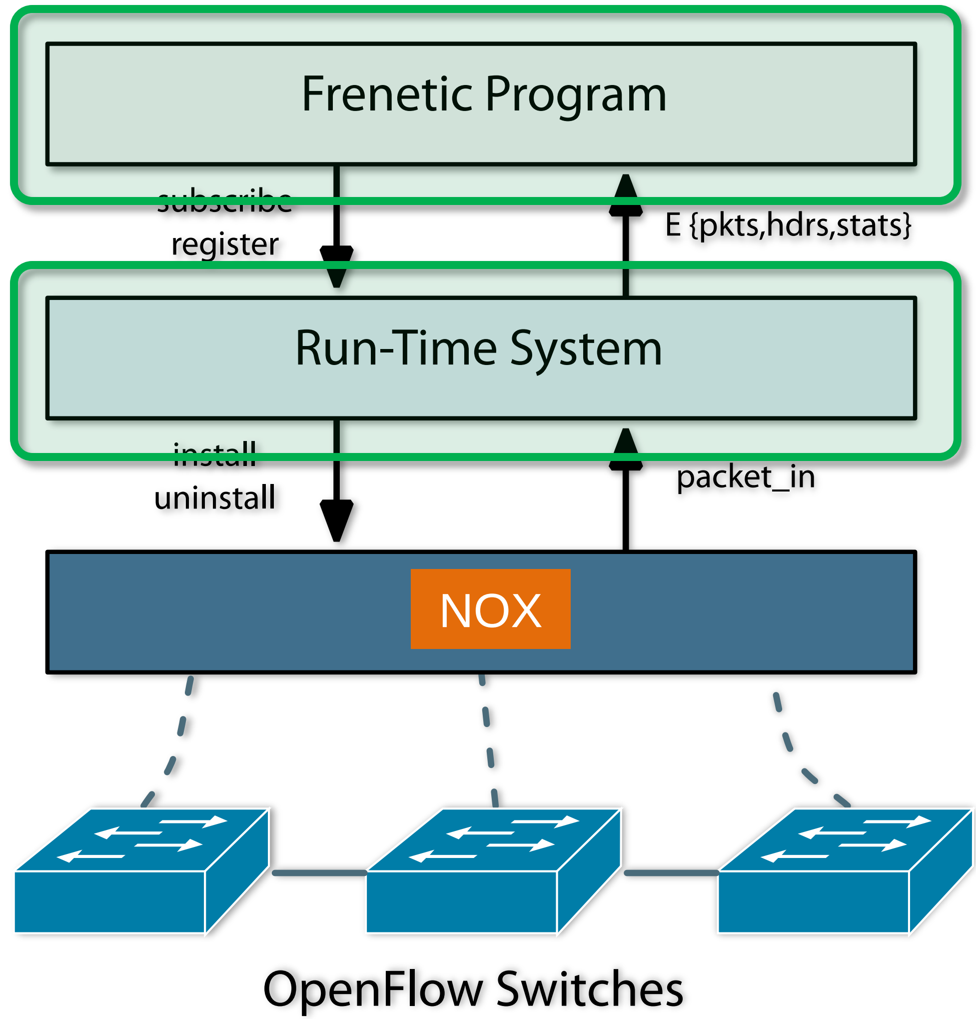
Frenetic

A High-level Language

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

A Run-time System

- Handles module interactions
- Deals with asynchronous behavior



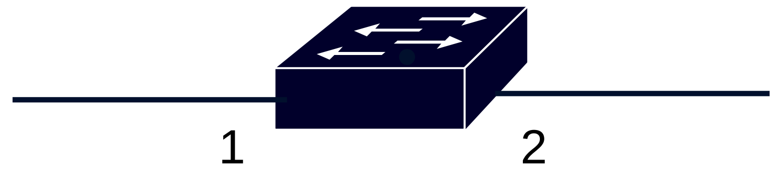
Frenetic Design

The Network Query Language

- Low-level rules on switches
- High-level abstraction for programmers
- Result is an event stream

$$
\begin{array}{lrl}
\textit{Queries} & q ::= & \texttt{Select}(a) \ \texttt{*} \\
 & & \texttt{Where}(fp) \ \texttt{*} \\
 & & \texttt{GroupBy}([qh_1, \ldots, qh_n]) \ \texttt{*} \\
 & & \texttt{SplitWhen}([qh_1, \ldots, qh_n]) \ \texttt{*} \\
 & & \texttt{Every}(n) \ \texttt{*} \\
 & & \texttt{Limit}(n) \\
\textit{Aggregates} & a ::= & \texttt{packets} \mid \texttt{sizes} \mid \texttt{counts} \\
\textit{Headers} & qh ::= & \texttt{inport} \mid \texttt{srcmac} \mid \texttt{dstmac} \mid \texttt{ethtype} \mid \\
 & & \texttt{vlan} \mid \texttt{srcip} \mid \texttt{dstip} \mid \texttt{protocol} \mid \\
 & & \texttt{srcport} \mid \texttt{dstport} \mid \texttt{switch} \\
\textit{Patterns} & fp ::= & \texttt{true\_fp}() \mid qh\texttt{\_fp}(n) \mid \\
 & & \texttt{and\_fp}([fp_1, \ldots, fp_n]) \mid \\
 & & \texttt{or\_fp}([fp_1, \ldots, fp_n]) \mid \\
 & & \texttt{diff\_fp}(fp_1, fp_2) \mid \texttt{not\_fp}(fp)
\end{array}
$$

Frenetic Queries



Goal: measure the total bytes of web traffic arriving on port 2, every 30 seconds

data to be returned from query (options: sizes, counts, packets)

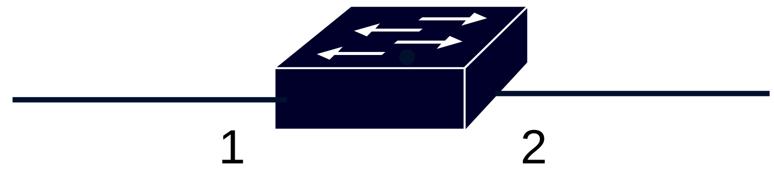
filter based on packet headers (web traffic in on port 2)

```
def web_query():
  return (Select (sizes) *
          Where  (inport_fp (2) & srcport_fp (80)) *
          Every  (30))
```

period: 30 seconds

Key Property: Query semantics independent of other program parts

Frenetic Queries

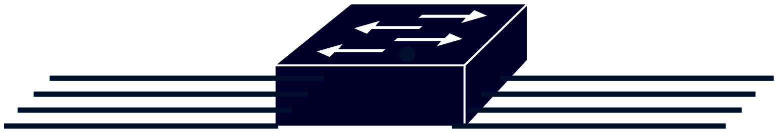


Goal: sum the number of packets, per host (ie: mac address), traveling through port 2, every minute

```
def host_query():
  return (Select  (counts)       *
          Where   (inport_fp(2)) *
          GroupBy ([srcmac])     *
          Every   (60))
```

categorize results by srcmac address

Frenetic Queries



Goal: report the hosts connected to each switch port; report a host each time it moves from one port to the next

get packets for analysis

```
def learning_query():
  return (Select    (packets)   *
          GroupBy   ([srcmac])  *
          SplitWhen ([inport])  *
          Limit     (1))
```

at most one packet per flow

categorize by srcmac

sub-categorize when the inport changes (the host moves)

Key Property: Query implementation handles network race conditions

Using Queries

- Query results, or other streams, are piped in to listeners

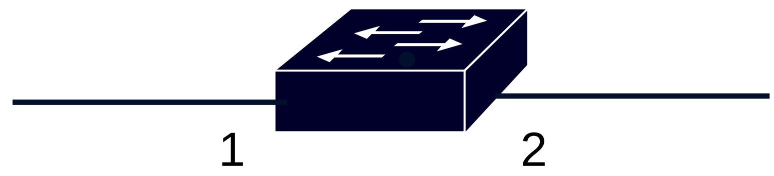
```
def web_query(): …
def host_query(): …
def learning_query(): …
```

```
def web_stats():
  web_query() >> Print()
```

```
def all_stats():
  Merge(web_query(), host_query()) >> Print()
```

Key Property: Queries compose

Frenetic Forwarding Policies



Goal: implement a repeater switch

rule actions

```
rules = [Rule(inport_fp(1), [forward(2)]),
         Rule(inport_fp(2), [forward(1)])]
```

```
def repeater():
  return (SwitchJoin() >>
          Lift(lambda switch: {switch:rules}))
```

listen for switch joining the network

packet pattern (defined over headers)

```
def main():
  repeater() >> register()
```

construct repeater policy for that switch

register policy with run time

Key Property: Policy semantics independent of other queries/policies

Program Composition

Goal: implement both the stats monitor and the repeater

```
def main():
  repeater() >> register()
  all_stats()
```

Key Property: Queries and policies compose

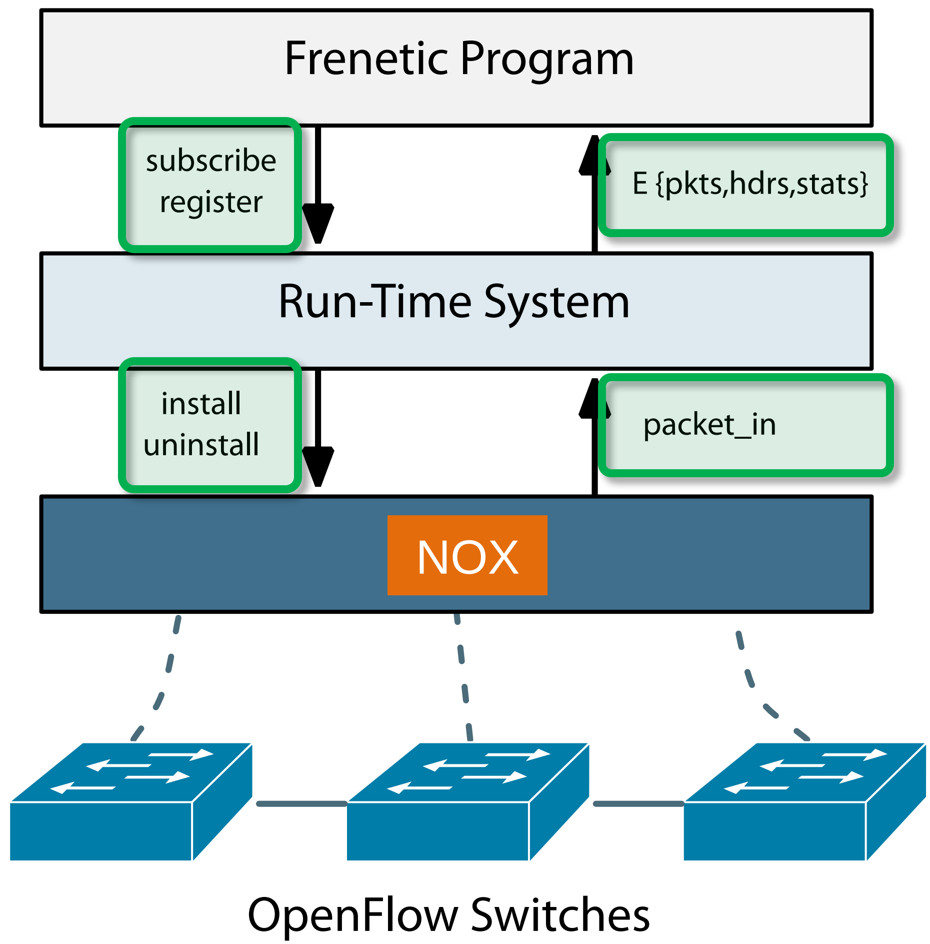
Frenetic Run-time System

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



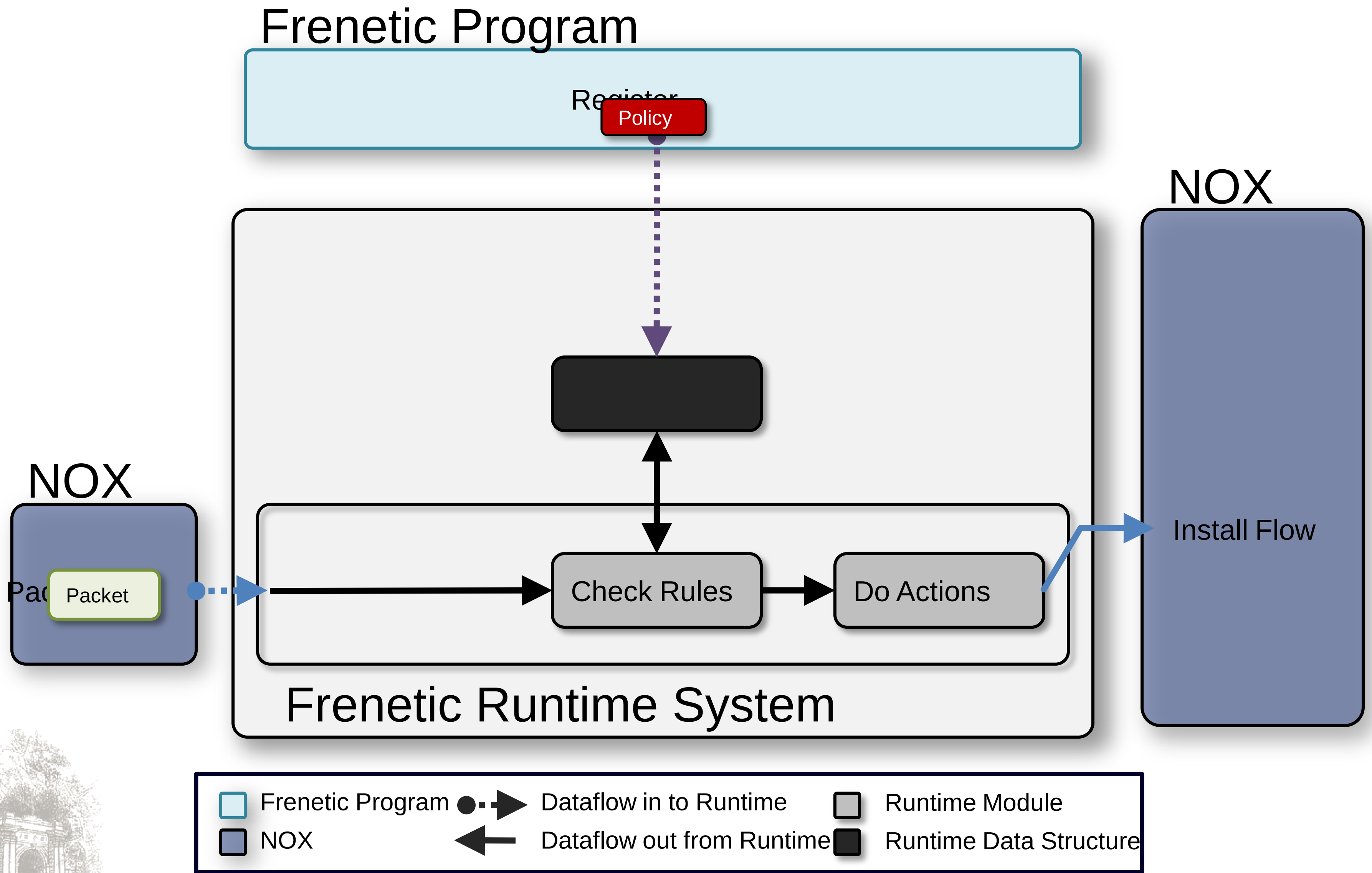
Implementation Options

- Rule Granularity
  - microflow (exact header match)
    - simpler; more rules generated
  - wildcard (multiple header match in single rule)
    - more complex; fewer rules (may be) generated
- Rule Installation
  - reactive (lazy)
    - first packet of each new flow goes to controller
  - proactive (eager)
    - new rules pushed to switches

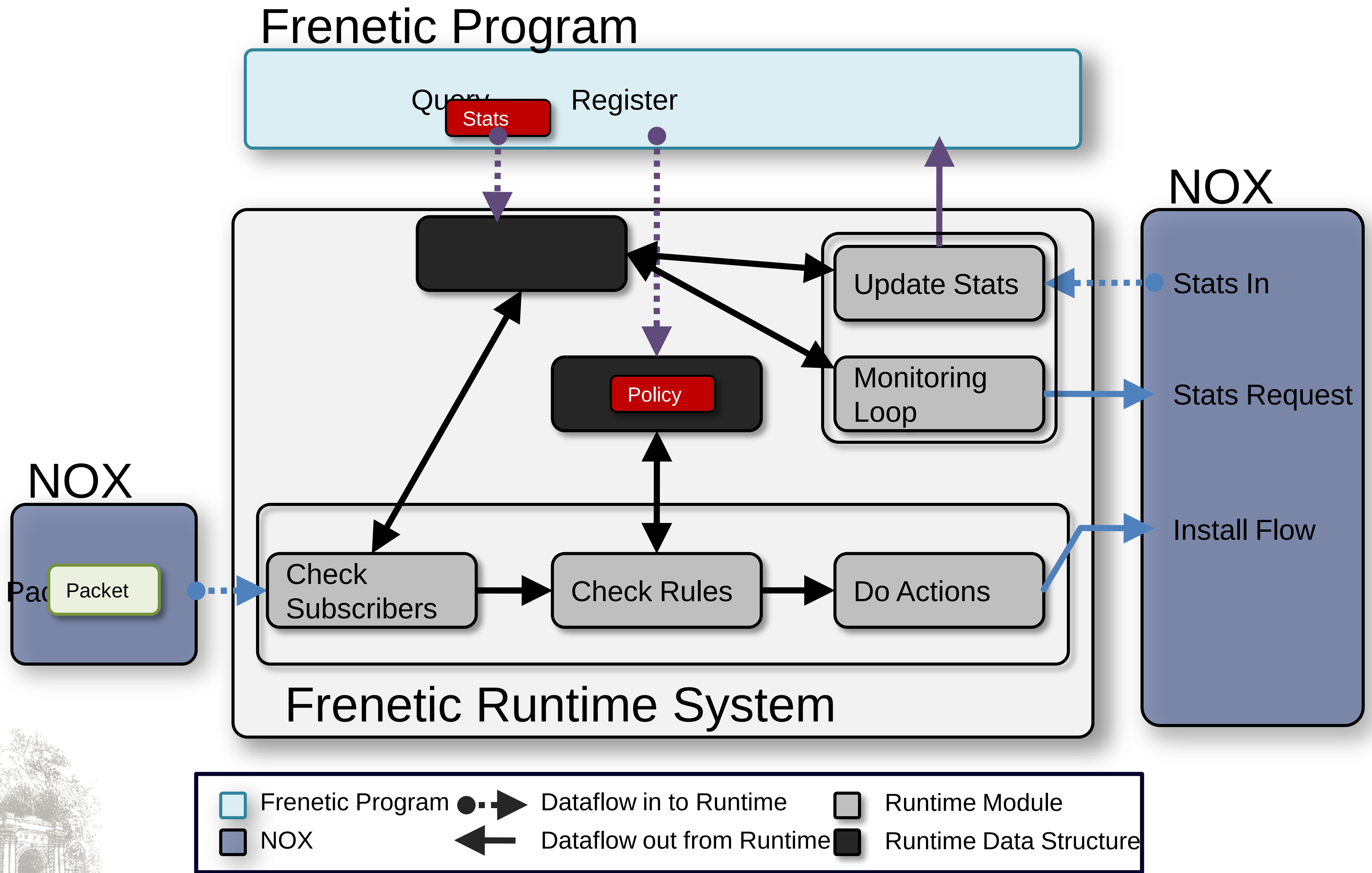
Frenetic 1.0

Frenetic 2.0

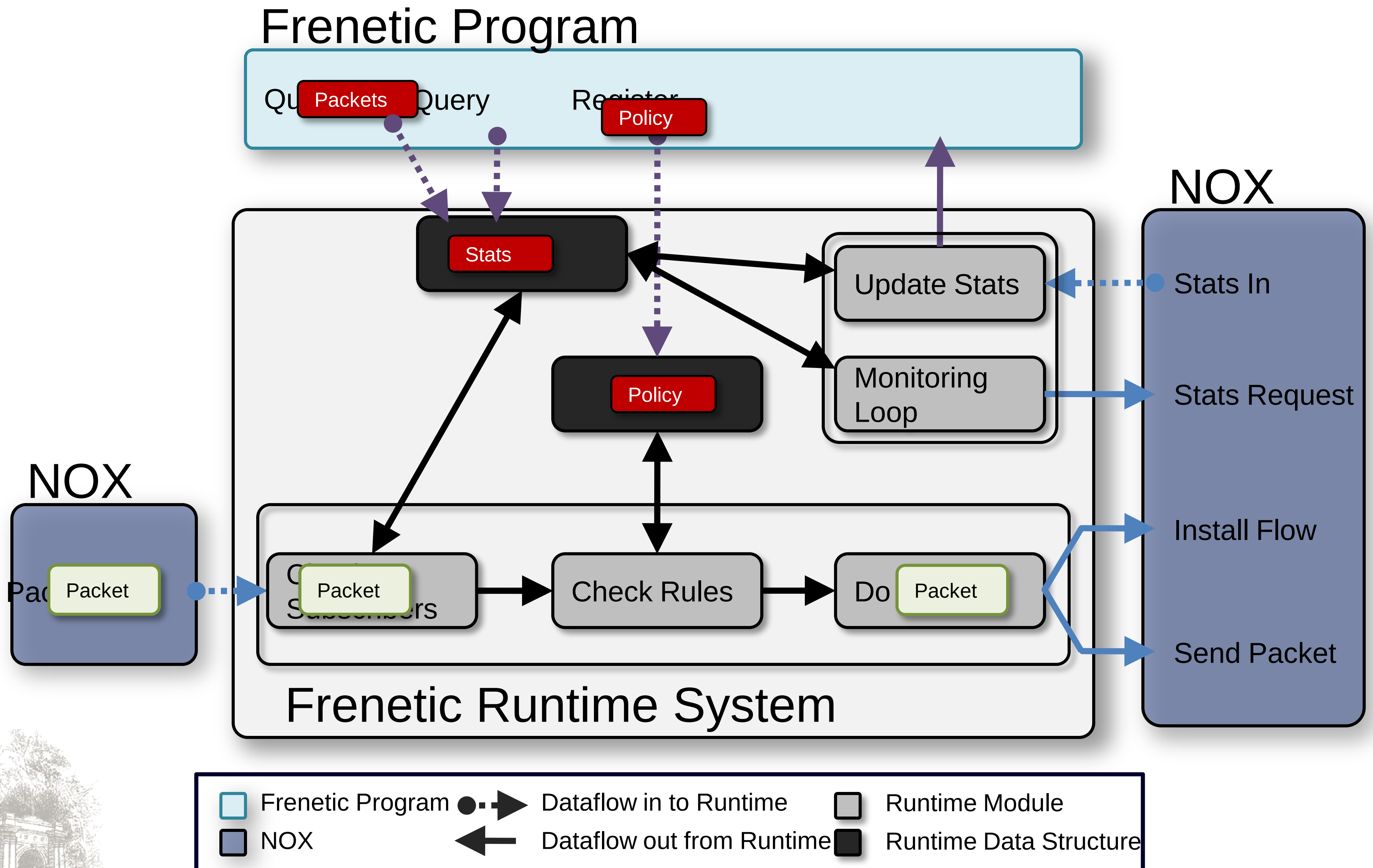
Run-time Activities



Run-time Activities



Run-time Activities



Evaluation

Metrics:

- Lines of code
- Traffic to controller
- Total trafic

Microbenchmarks:

- All-Pairs Connectivity
- Web Statistics
- Heavy Hitters

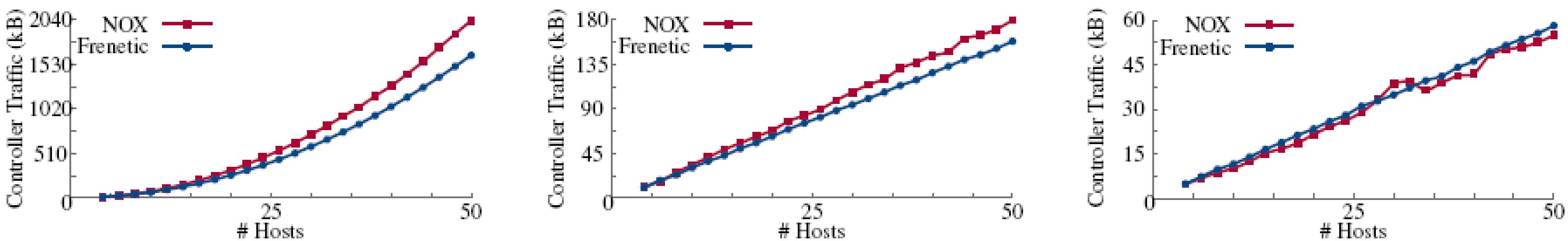
Forwarding Policy:

- Hub
- Learning Switch
- Loop-Free Learning Switch

Evaluation

		Connectivity			Heavy Hitters			Web Stats		
		HUB	*LSW*	*LFLSW*	*HUB*	*LSW*	*LFLSW*	*HUB*	*LSW*	*LFLSW*
NOX	Lines of Code	20	55	75	110	198		104	135	
	Controller Traffic (kB)	12.8	13.5	31.3	9.3	10.3	⋆	4.5	4811	⋆
	Aggregate Traffic (kB)	69.2	42.3	64.1	57.2	36.1		14.1	9.0	
Frenetic	Lines of Code	6	30	58	29	53	81	13	37	65
	Controller Traffic (kB)	9.1	12.0	12.4	11.1	10.6	10.9	4.5	5.1	5.8
	Aggregate Traffic (kB)	65.6	41.0	41.5	55.0	36.4	36.9	13.6	9.20	9.9

Table 1. Experimental results.



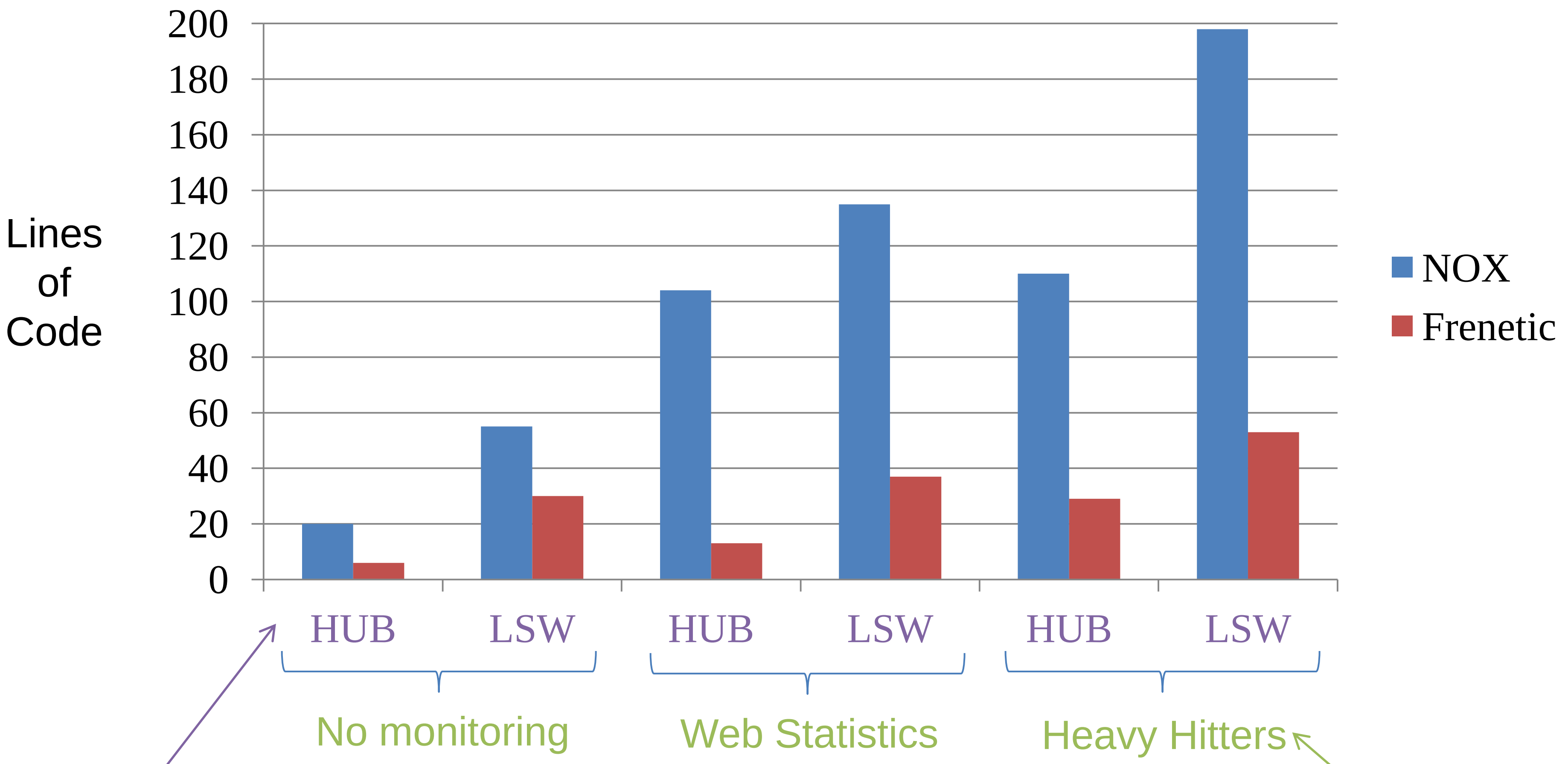
(a) All-Pairs Connectivity

(c) Heavy Hitters

(b) Web Statistics

Figure 9. Scalability experimental results.

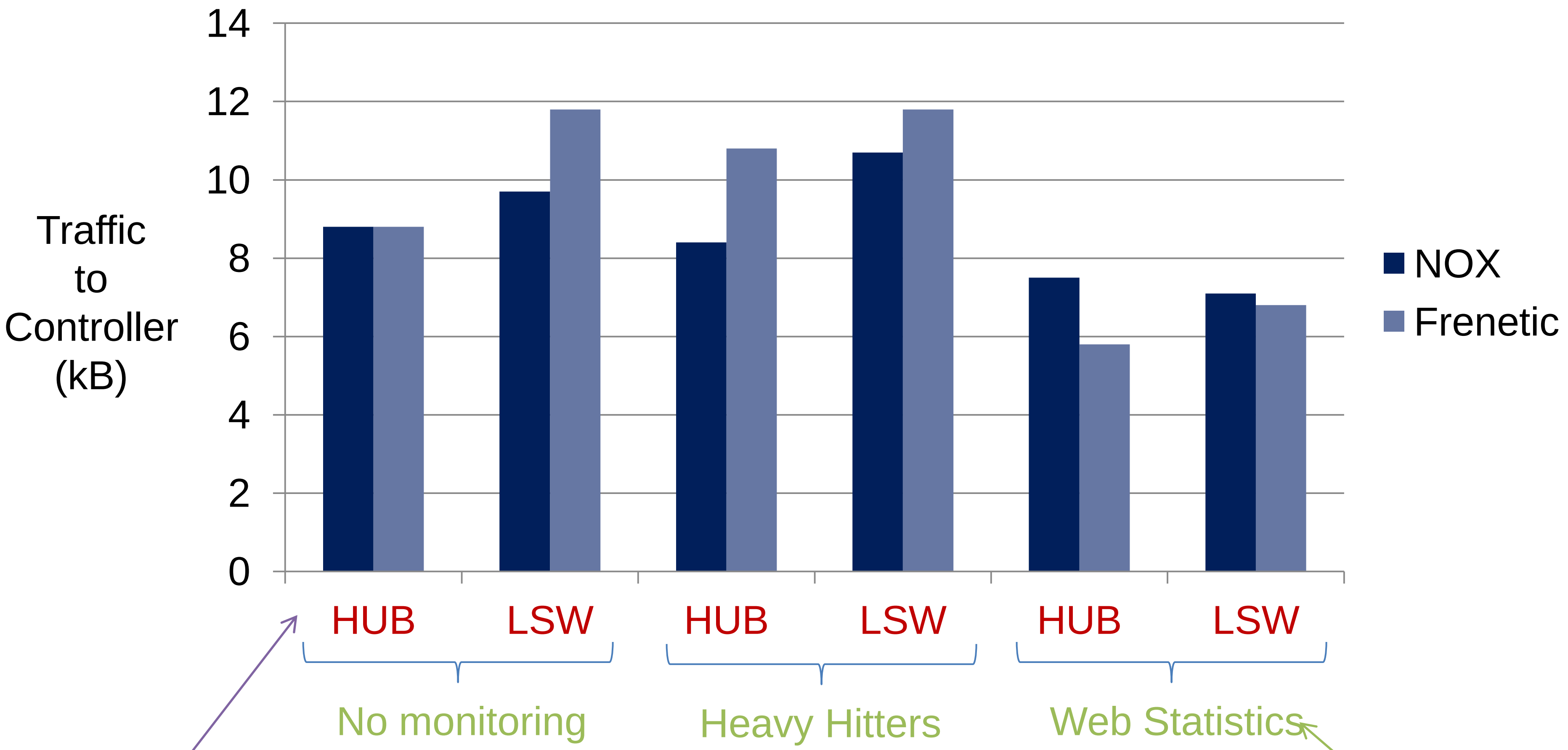
MicroBench: Lines of Code



Forwarding Policy:
HUB: Floods out other ports
LSW: Learning Switch

Monitoring Policy

MicroBench: Controller Traffic

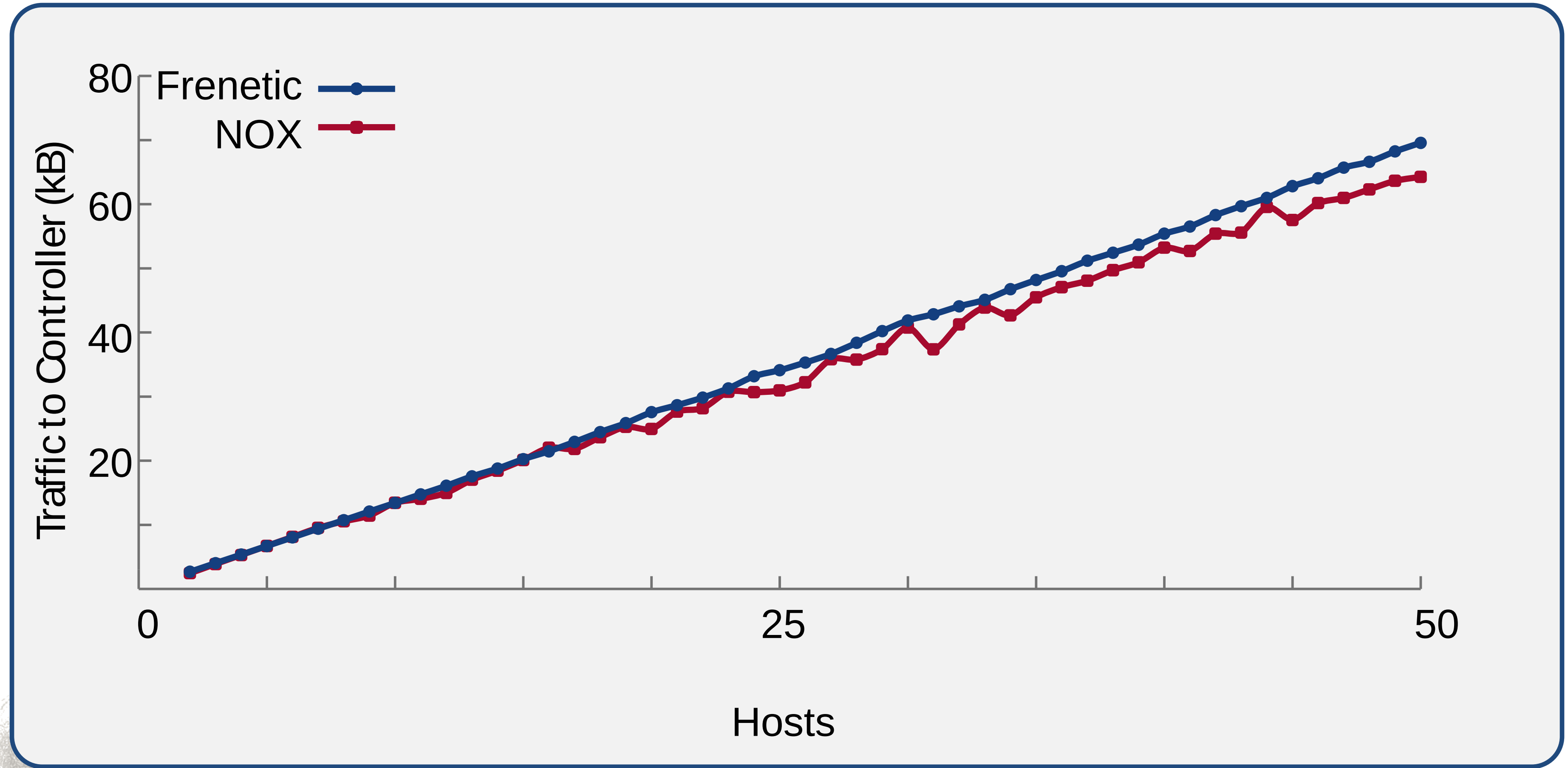


Forwarding Policy:
HUB: Floods out other ports
LSW: Learning Switch

Monitoring Policy

Frenetic Scalability

Frenetic scales to larger networks comparably with NOX



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

Acknowledgements

- Almost the whole content comes from authors' slides presented at PRESTO 2010, Microsoft Research 2011 and also their paper at ICFP 2011
- This slides is only for seminar use in NSLab
- For more information, please refer to the following links:
  - http://www.cs.princeton.edu/~jrex/papers/icfp11.pdf
  - http://www.cs.princeton.edu/~jrex/talks/presto11.pptx
  - http://www.cs.princeton.edu/~dpw/talks/frenetic-05-11.pptx

Discussion