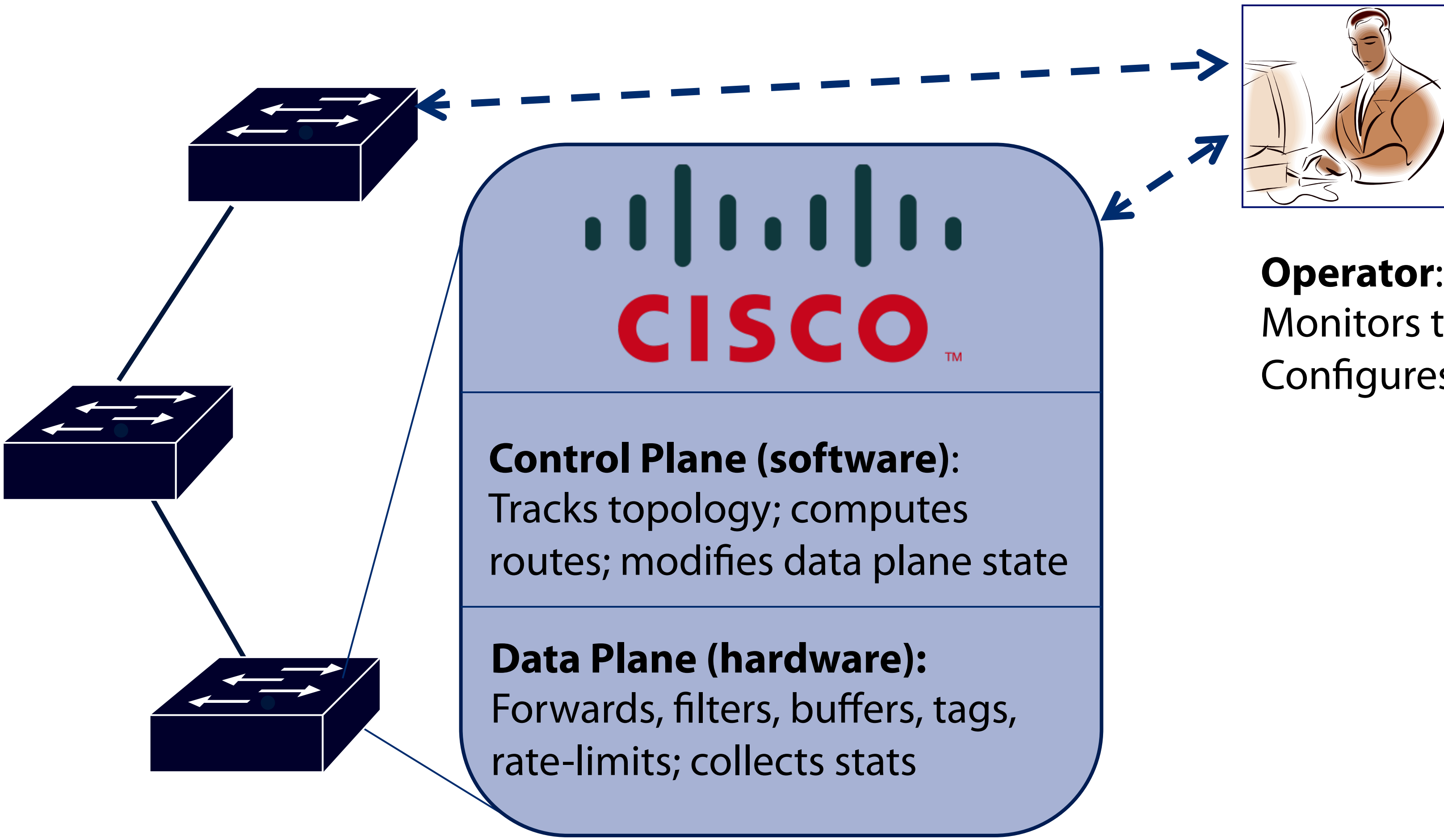
frenetic >>

A Network Programming Language

Nate Foster, Mike Freedman,
Rob Harrison, Chris Monsanto,
Jen Rexford, Alec Story, and Dave Walker

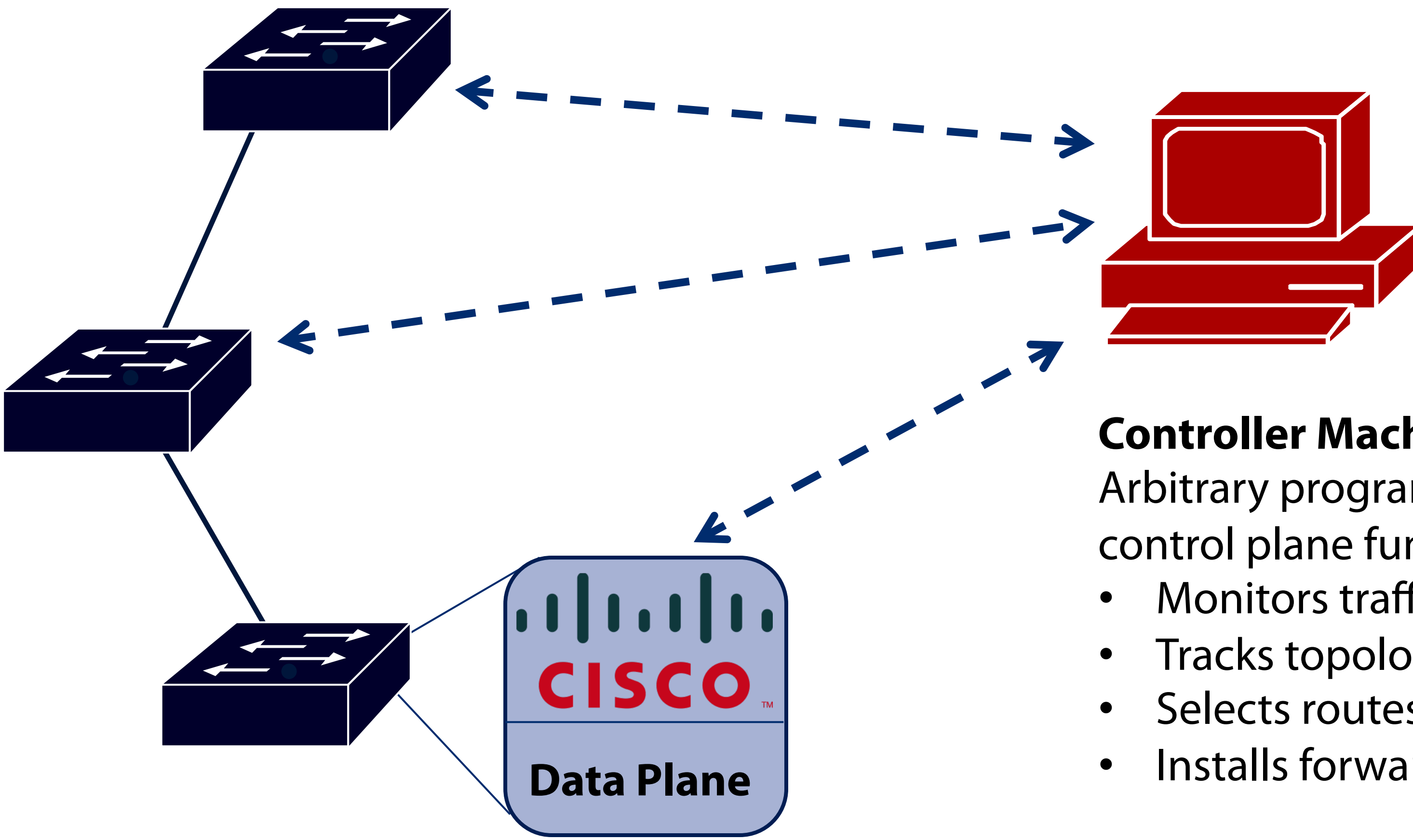
Traditional Networks



Operator:
Monitors traffic,
Configures policy

Software-Defined Networks

Idea: move control off of switches and onto a separate, general-purpose computer.



Controller Machine

Arbitrary program implements control plane functionality:

- Monitors traffic,
- Tracks topology,
- Selects routes,
- Installs forwarding rules.

OpenFlow Momentum

Everyone has signed on

Microsoft, Google, Cisco, Yahoo, Facebook, Deutch Telekom,…

New Applications

- Host mobility
- Virtualization
- Dynamic access control
- Energy-efficiency
- Load balancing

New Challenges

OpenFlow makes it *possible* to program the network, but it does not make it *easy!*

- Provides a thin veneer over switch hardware
- Like programming in assembly

Our goal

- Develop new abstractions for programming networks
  - More convenient
  - More modular
  - More reliable
  - More secure

This Talk

OpenFlow in more depth

- Existing programming model and problems

Frenetic Language

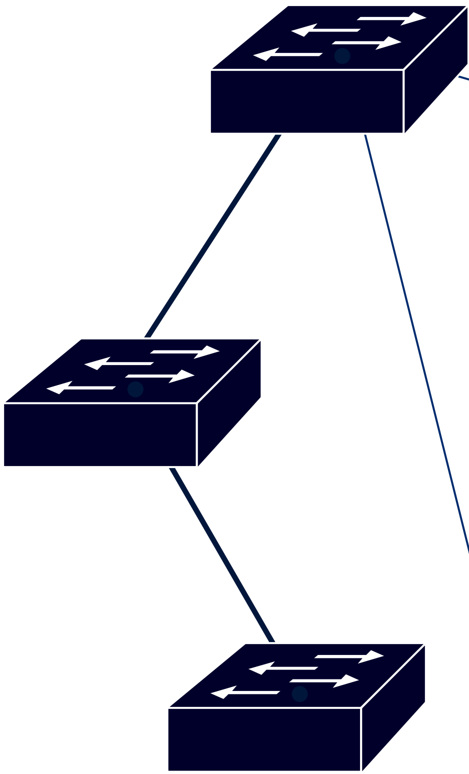
- New abstractions for network programming

Frenetic Run-time System

- Implementation strategy and experience

OpenFlow Switches

Switches



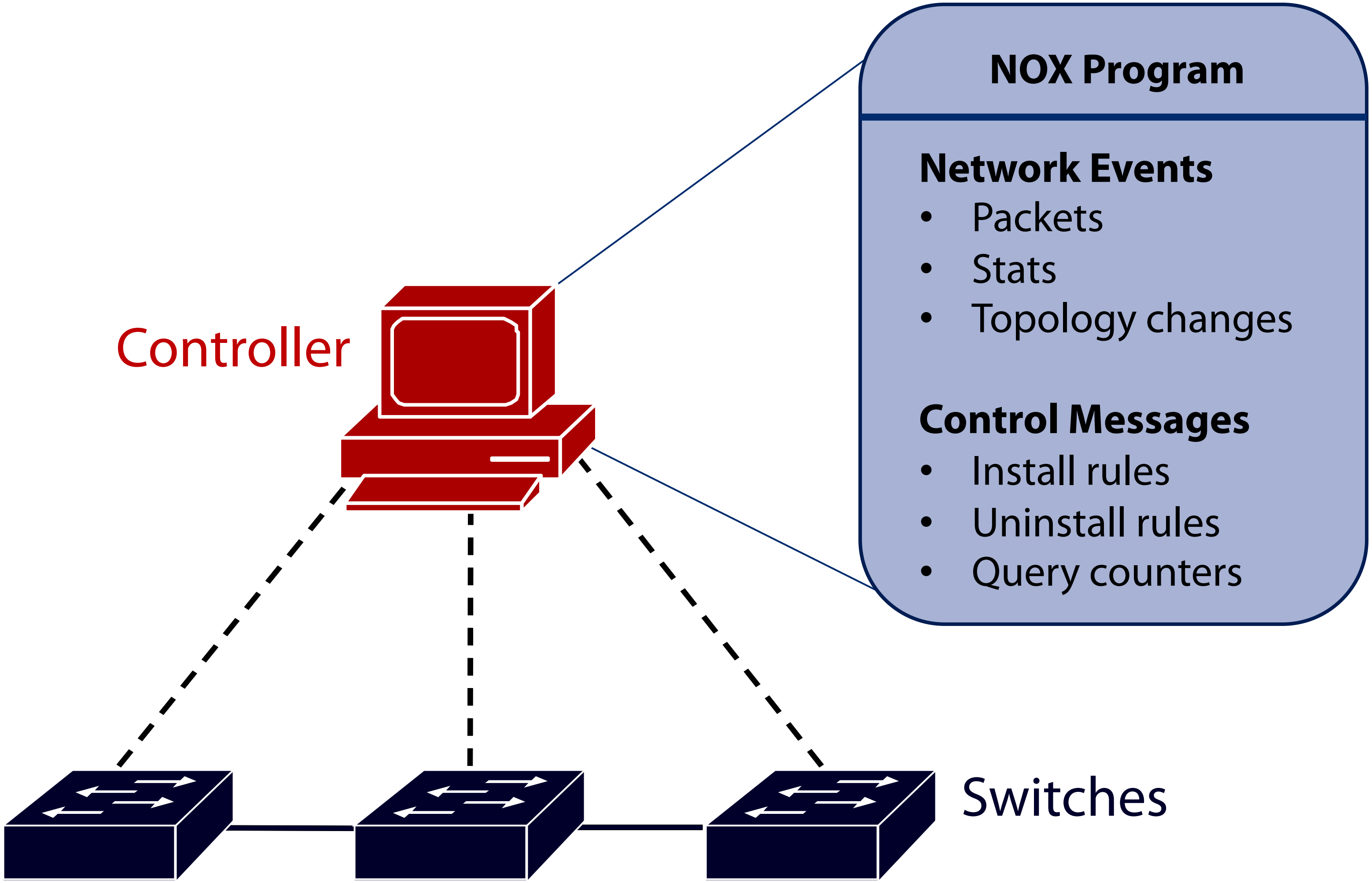
OpenFlow

Flow Table

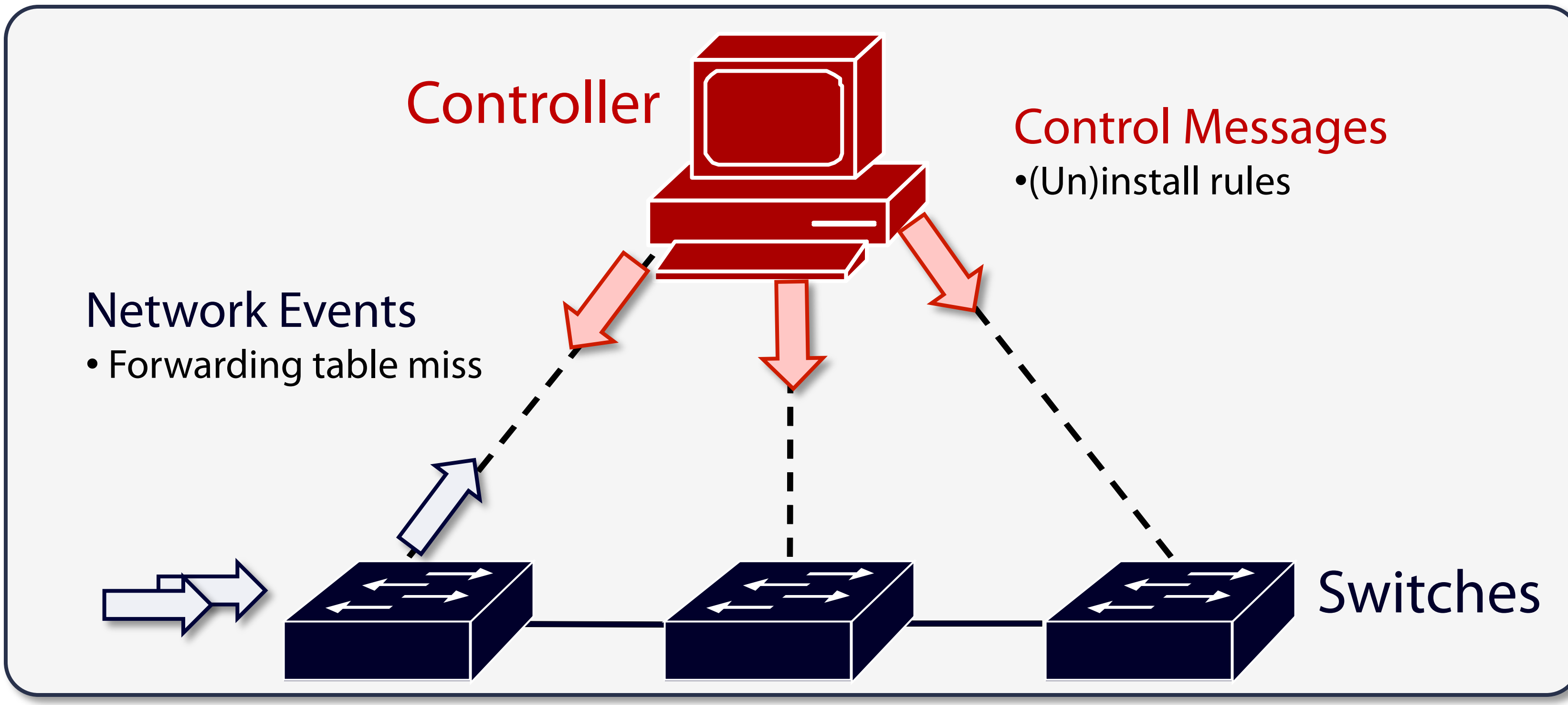
Pattern	Action	Bytes	Packets
01010	Drop	200	10
010*	Forward(n)	100	3
011*	Controller	0	0

priority ↓

OpenFlow Controllers (NOX)

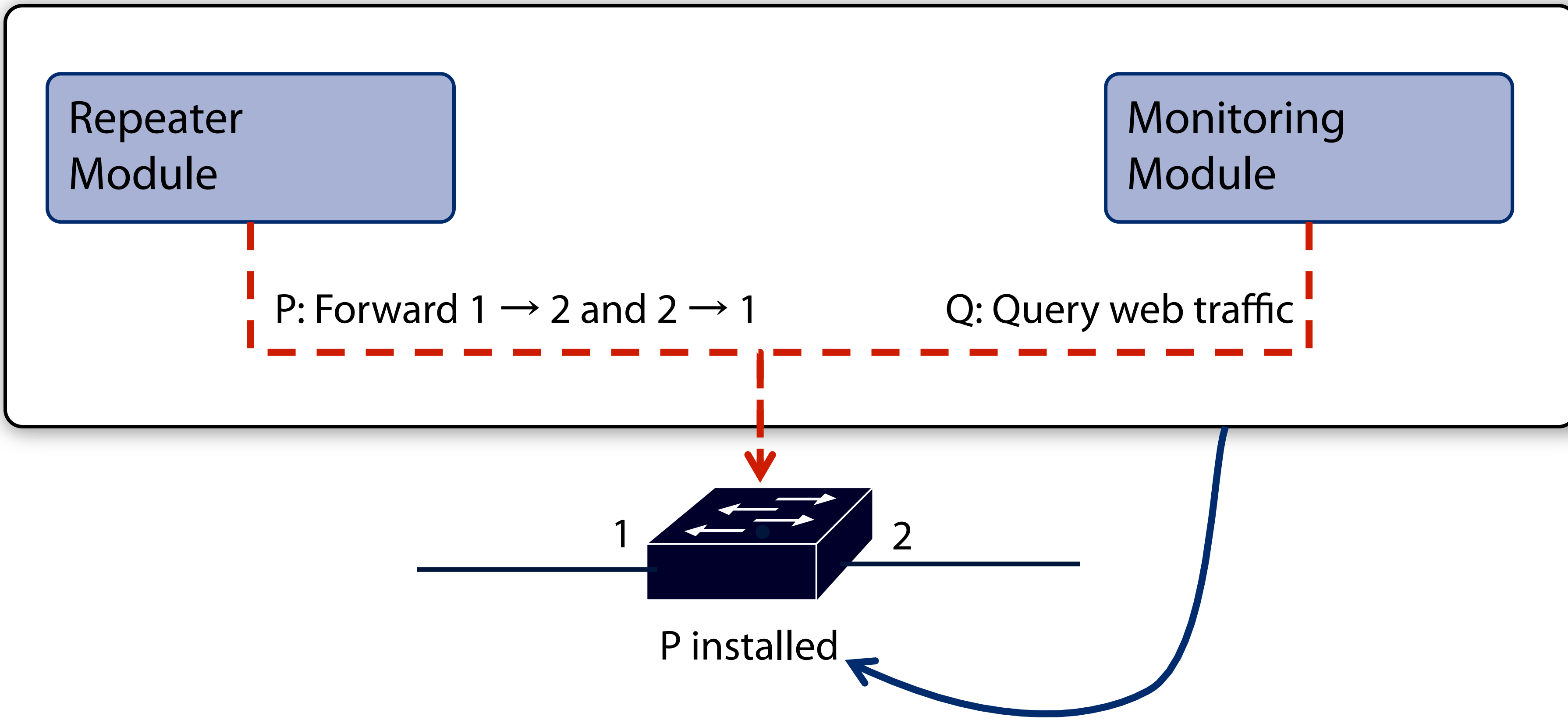


Typical OpenFlow Application



Problem I: Anti-Modular

Controller Application



Doesn't work because repeater rules too coarse-grained; monitoring rules don't forward

Anti-Modularity: A Closer Look

Repeater

```
def switch_join(switch):
  repeater(switch)

def repeater(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  install(switch,pat1,DEFAULT,None,[output(2)])
  install(switch,pat2,DEFAULT,None,[output(1)])
```

Web Monitor

```
def monitor(switch):
  pat = {in_port:2,tp_src:80}
  install(switch, pat, DEFAULT, None, [])
  query_stats(switch, pat)

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

Repeater/Monitor

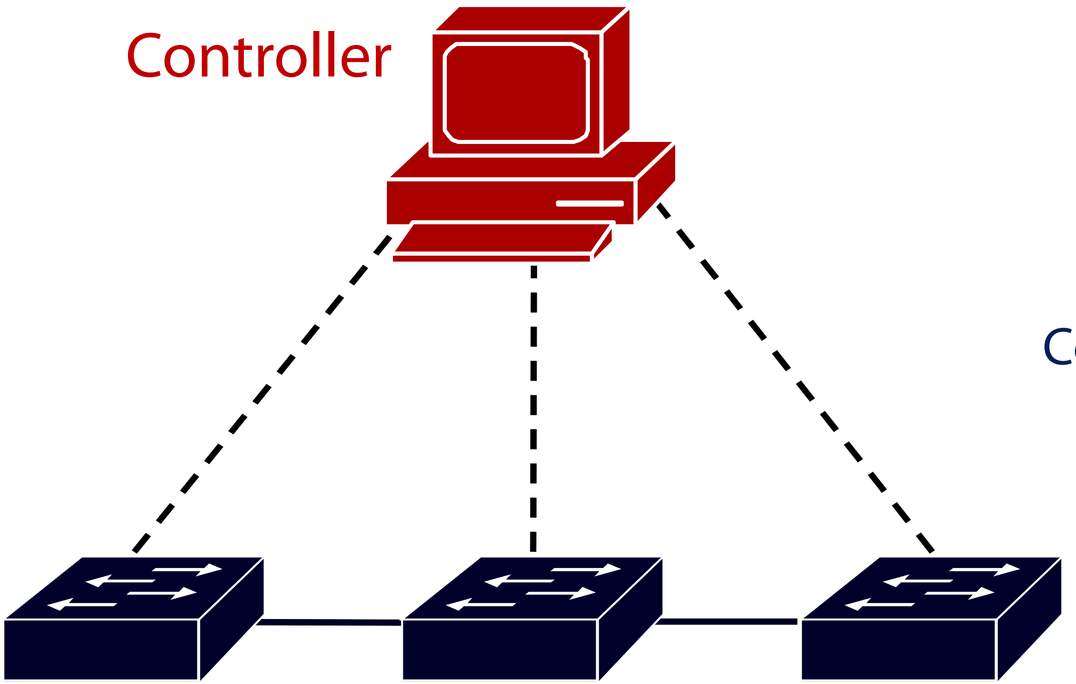
```
def switch_join(switch)
  repeater_monitor(switch)

def repeater_monitor(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  pat2web = {in_port:2, tp_src:80}
  Install(switch, pat1, DEFAULT, None, [output(2)])
  install(switch, pat2web, HIGH, None, [output(1)])
  install(switch, pat2, DEFAULT, None, [output(1)])
  query_stats(switch, pat2web)

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

blue = from repeater
red = from web monitor
green = from neither

Problem II: Two-tiered Model



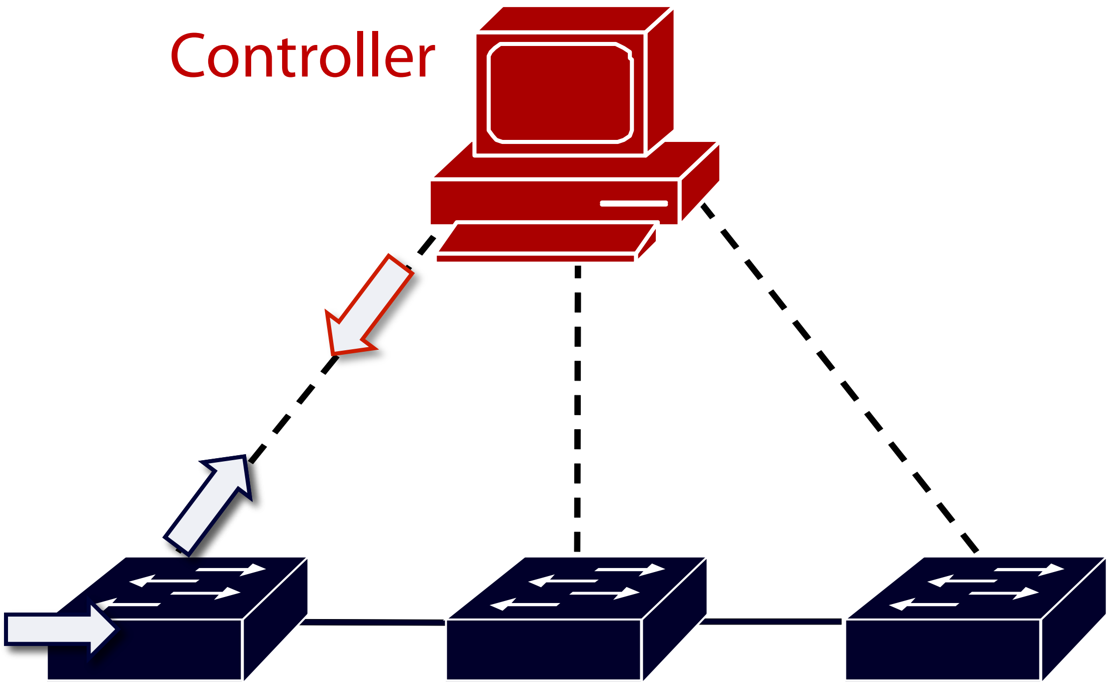
Tricky problem:

- Controller activity is driven by packets
- For efficiency, applications install rules to forward packets in hardware

Constant questions:

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

Problem III: Network Race Conditions



A challenging sequence 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

Three problems with a common cause

Three problems

- Anti-modular
- Two-tiered model
- Network race conditions

One cause

No effective *abstractions* for *reading* network state

The Solution

Separate network programming into two parts:

- Abstractions for querying network state
  - Reads have *no effect* on forwarding policy
  - Reads able to *see every packet*

- Abstractions for specifying a forwarding policy
  - Forwarding *policy* must be separated from *implementation mechanism*

A natural decomposition that mirrors two fundamental tasks: monitoring and forwarding

This Talk

OpenFlow & Nox in more depth

- Existing programming model and problems

Frenetic Language

- New abstractions for network programming

Frenetic Run-time System

- Implementation strategy and experience

Frenetic Language

Abstractions for querying network state

- An integrated query language
  - select, filter, group, sample sets of packets or statistics
  - designed so that computation can occur on data plane

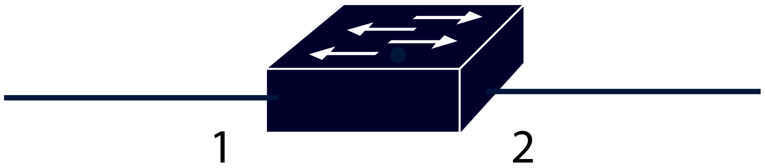
Abstractions for specifying a forwarding policy

- A functional stream processing library (based on FRP)
  - generate streams of network policies
  - transform, split, merge, filter policies and other streams

Implementation:

- A collection of Python libraries on top of NOX

Frenetic Queries

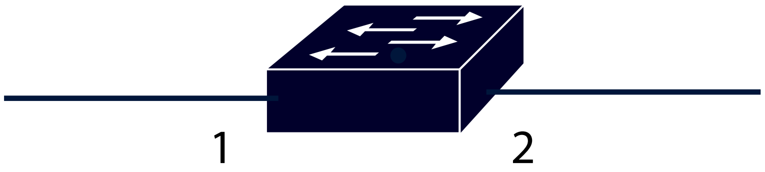


Goal: measure total web traffic on port 2, every 30 seconds

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

Key Property: query semantics is independent of other program parts

Frenetic Forwarding Policies



Goal: implement a repeater switch

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

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

Key Property: Policy semantics independent of other queries/policies

Program Composition

Goal: implement both web monitoring and repeater

```
def host_query():
  return (Select(counts) *
          Where(inport_fp(1) *
          GroupBy([srcmac]) *
          Every(60))

def secure(host_policy_stream): ...
```

```
def main():
  web_query() >> Print()
  secure(Merge(host_query(), repeater())) >> Register()
```

Key Property: queries and policies compose

This Talk

OpenFlow & Nox in more depth

- Existing programming model and problems

Frenetic Language

- New abstractions for network programming

Frenetic Run-time System

- Implementation strategy and experience

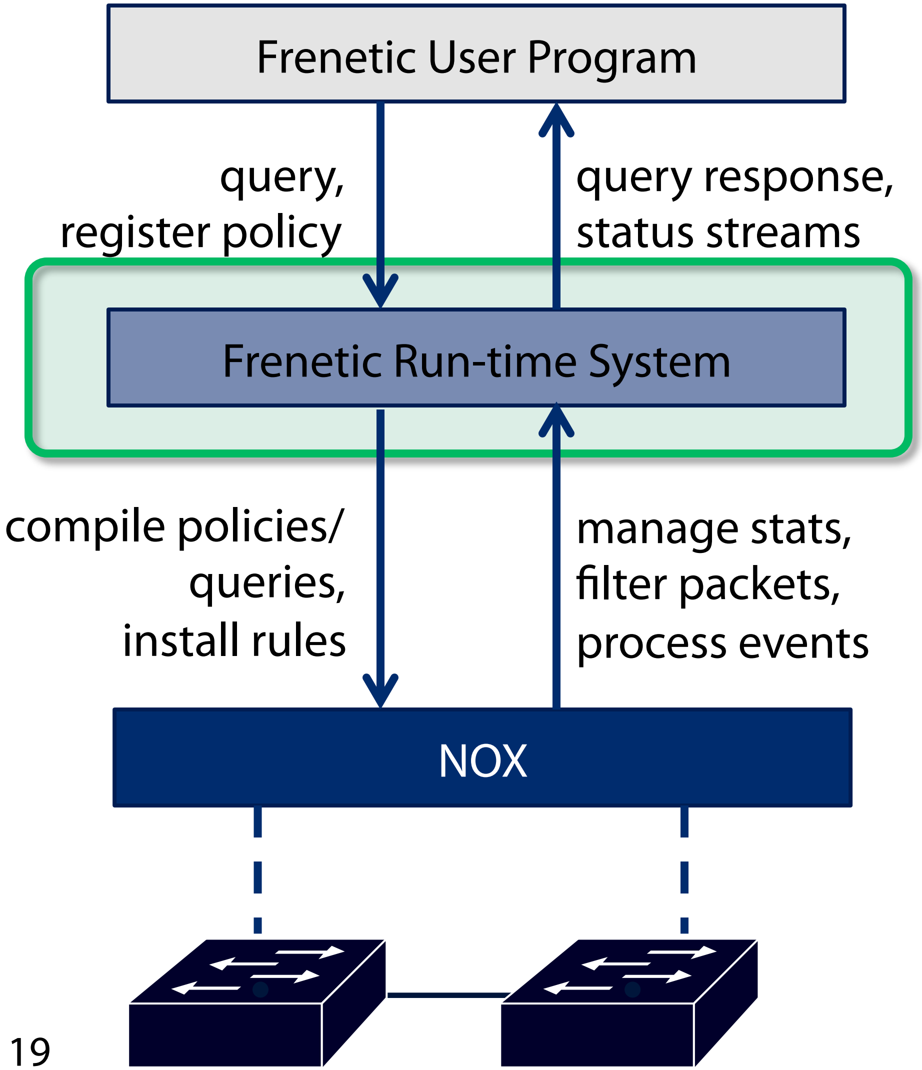
Frenetic System Overview

High-level Language

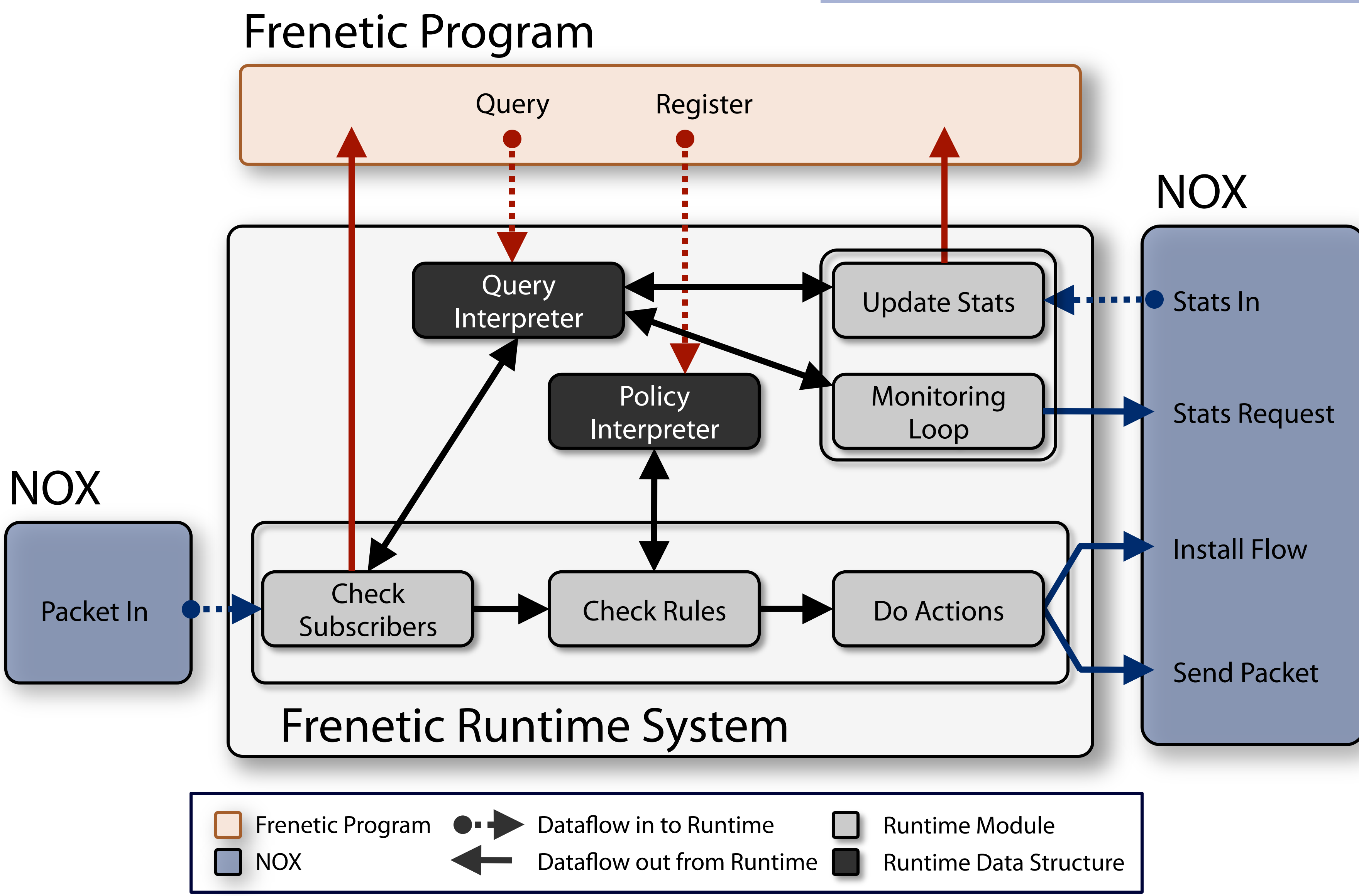
- Integrated query language
- Effective support for composition and reuse

Run-time System

- Interprets queries, policies
- Installs rules
- Tracks stats
- Handles asynchronous events



Run-time Activities



Preliminary Evaluation

Core Network Applications

- Learning Switch
- Spanning Tree
- Shortest path routing
- DHCP server
- Centralized ARP server
- Generic load balancer

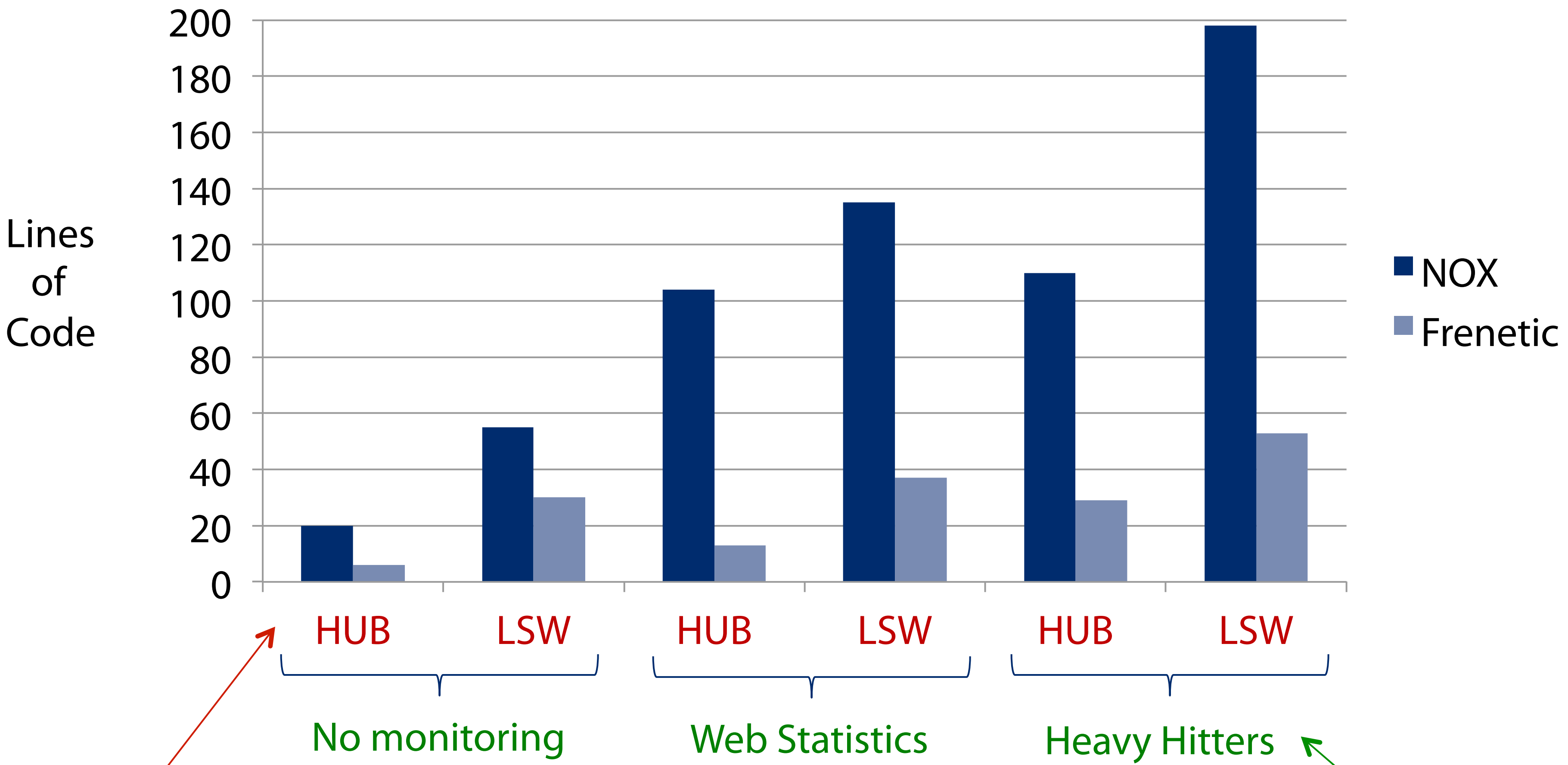
Additional Applications

- Memcached query router
- Network scan detector
- DDOS defensive switch

Micro Benchmarks

- Coded in Frenetic and NOX

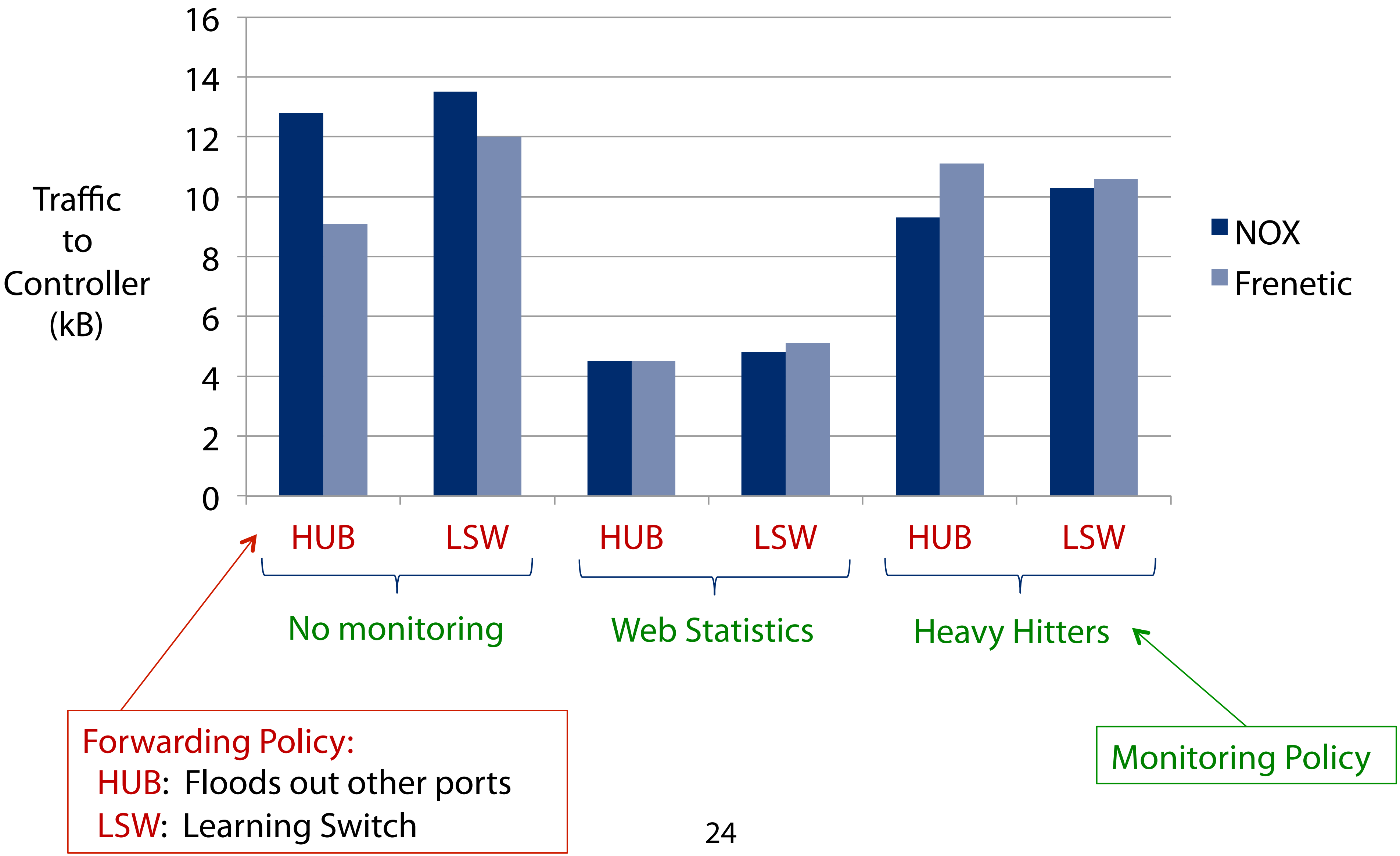
MicroBench: Lines of Code



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

Monitoring Policy

MicroBench: Controller Traffic



Ongoing and Future Work

Performance evaluation & optimization

- Measure controller response time and network throughput
- Wildcard rules and proactive rule installation
- Support for parallelism

Program Analysis

- Establish key invariants

Hosts and Services

- Extend queries & controls to end hosts

More abstractions

- Virtual network topologies
- Network updates with improved semantics

Conclusion: An Analogy

Concern	Assembly Languages		Programming Languages	
	x86	**NOX**	**Haskell/ML**	**Frenetic++**
Resource Allocation	Move values to/from registers	Install/ uninstall rules on switches	Declare/use program variables	Construct/ register policy
Resource Tracking	Have I spilled that value?	Will that packet arrive at the controller?	Program variables always accessible	Queries can read every packet
Coordination	Unregulated calling conventions	Unregulated rule installation	Function calls managed automatically	Policies managed automatically
Portability	Hardware Dependent	Hardware Dependent	Hardware Independent	Hardware Independent

The Team

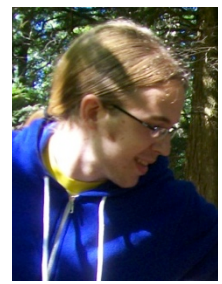
Nate Foster

Mike Freedman

Rob Harrison

Chris Monsanto

Jen Rexford



Alec Story

Dave Walker

frenetic >>

http://frenetic-lang.org

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

Frenetic 1.0

Rule Installation

- reactive (lazy)
  - first packet of each new flow goes to controller
- proactive (eager)
  - new rules pushed to switches

Frenetic 2.0