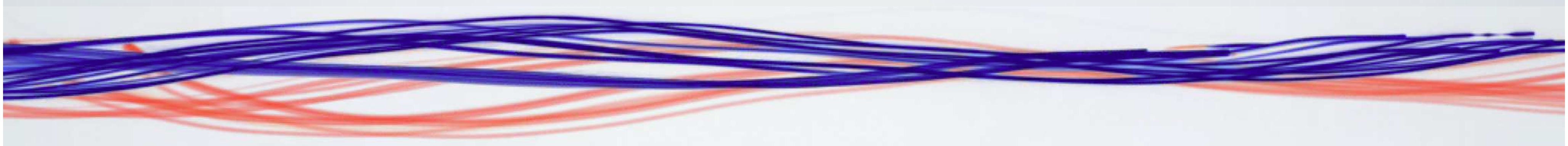
Abstractions For Software-Defined Networks



Nate Foster
Cornell

Jen Rexford & David Walker
Princeton

Software-Defined Networking

The Good

- Logically-centralized architecture
- Direct control over the network

Images by Billy Perkins

Software-Defined Networking

The Good

- Logically-centralized architecture
- Direct control over the network

The Bad

- Low-level programming interfaces
- Functionality derived from hardware

Images by Billy Perkins

Software-Defined Networking

The Good

- Logically-centralized architecture
- Direct control over the network

The Bad

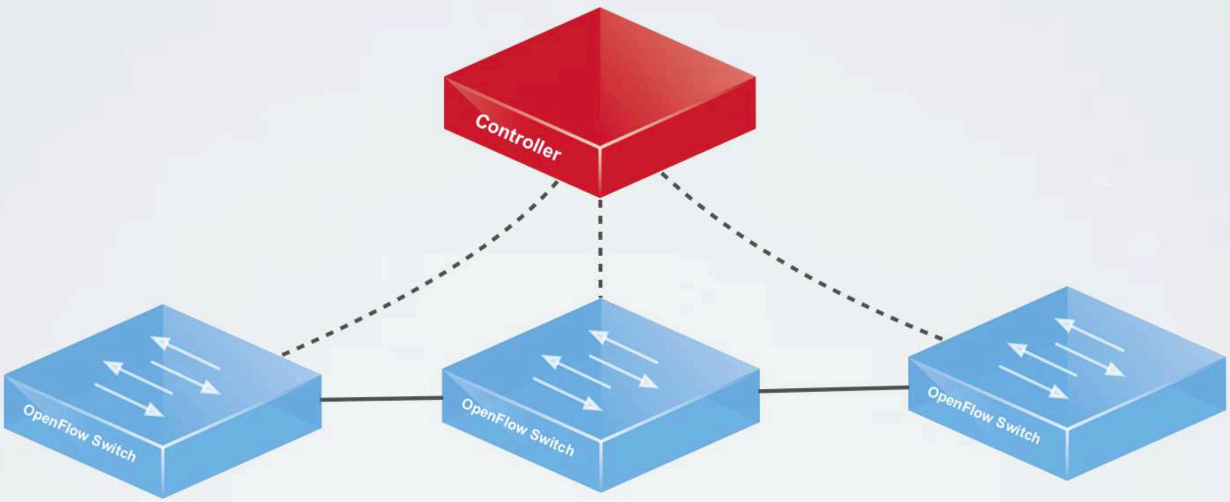
- Low-level programming interfaces
- Functionality derived from hardware

Images by Billy Perkins

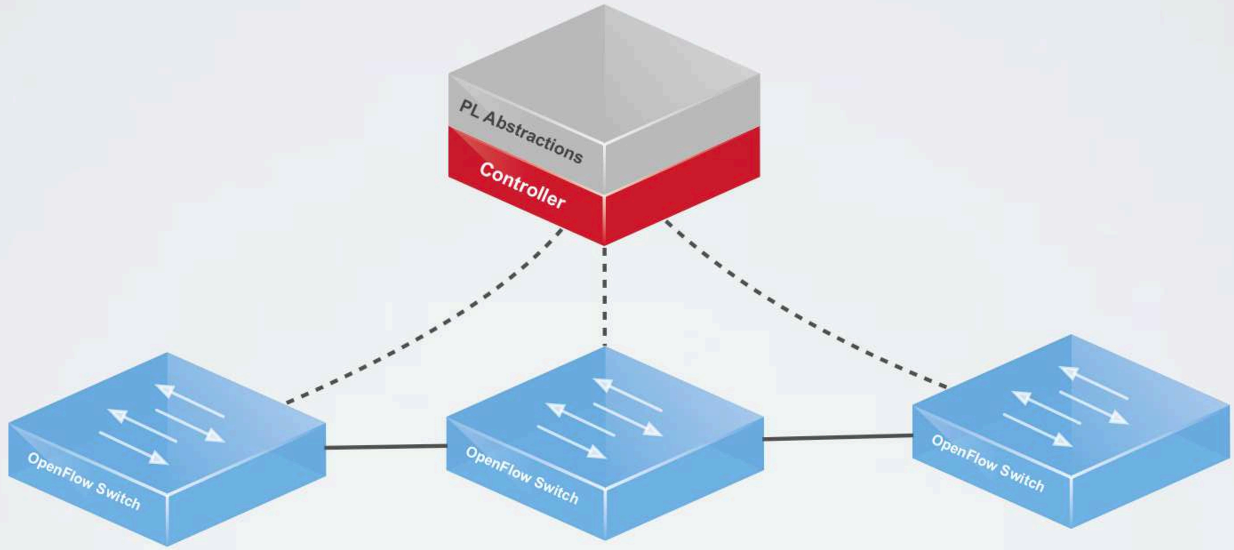
The Ugly

- Program pieces don't compose
- *Many* distributed systems challenges

Programming Abstractions



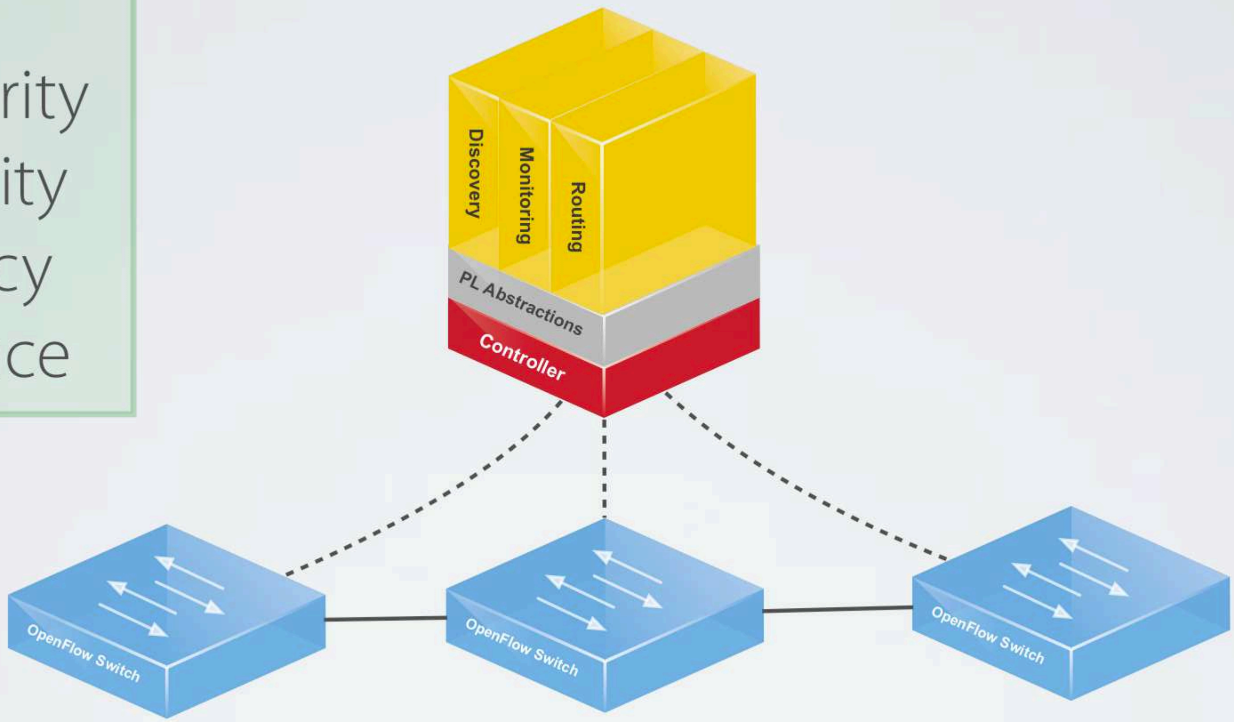
Programming Abstractions



Programming abstractions are crucial for achieving the vision of software-defined networking.

Programming Abstractions

Benefits
- Modularity
- Portability
- Efficiency
- Assurance



Programming abstractions are crucial for achieving the vision of software-defined networking.

This talk: Outline

SDN Basics

- Architecture
- Programming model

Network-Wide Abstractions

- Global network view
- Network updates

Modularity

- Composing programs
- Declarative policies and queries

Vision

- Challenges
- Opportunities

SDN Basics

Switches

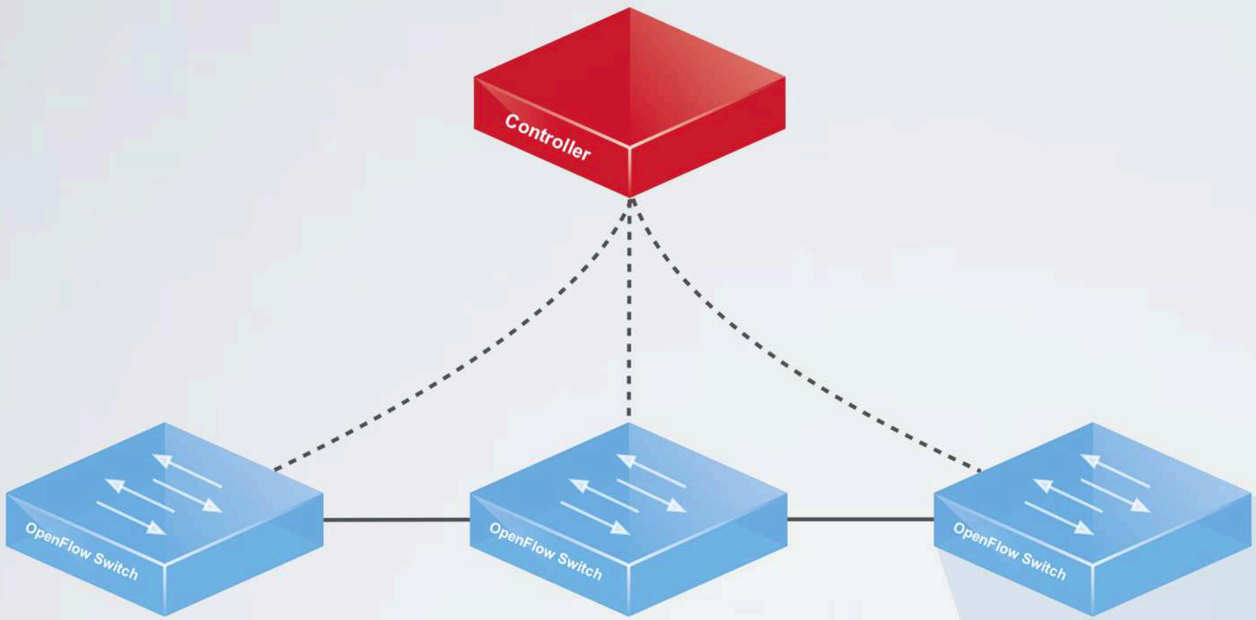


Table: prioritized list of rules

Rule: pattern, actions, and counters

Pattern: prefix match on headers

Action: forward or modify

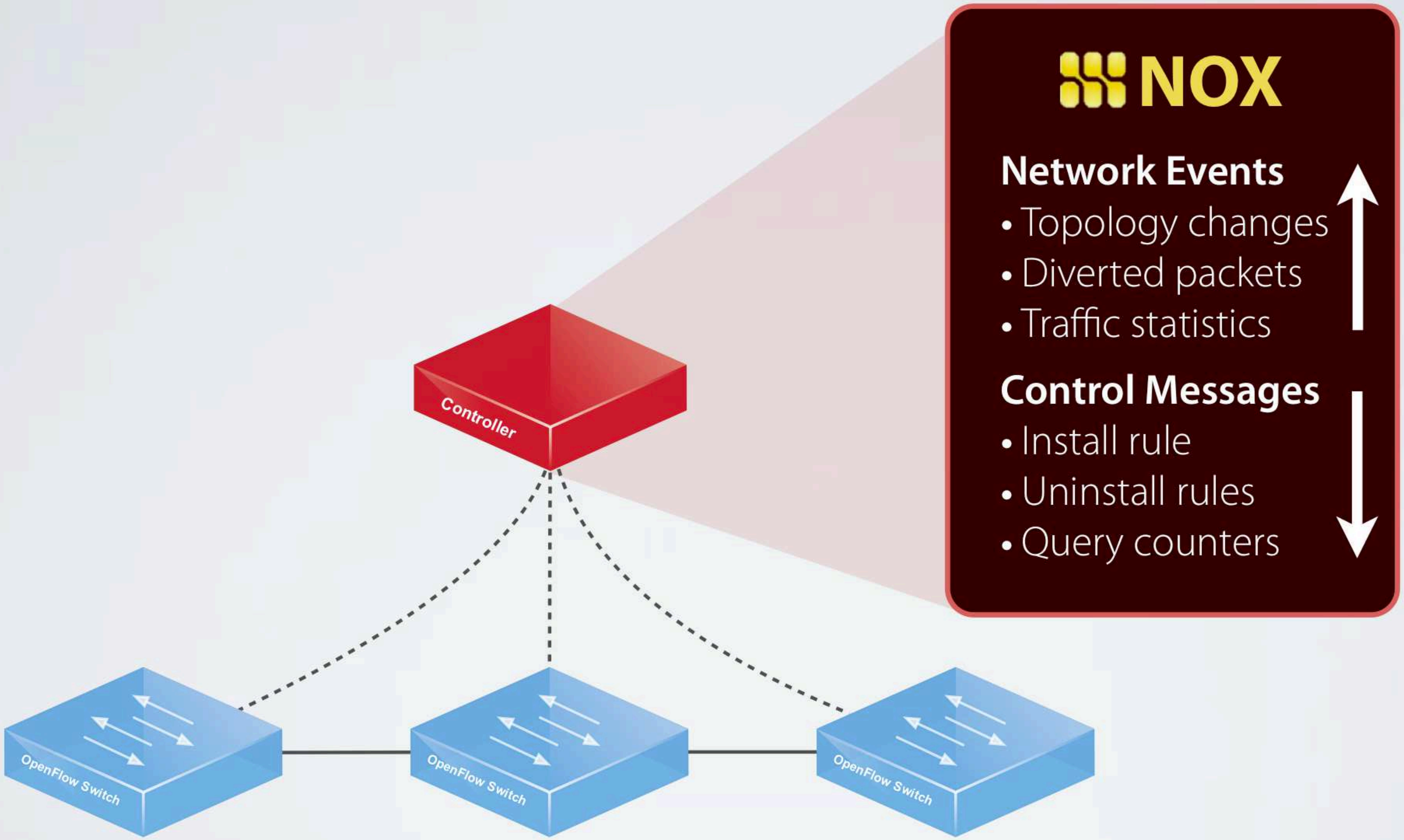
Counters: total bytes and packets processed

OpenFlow

Pattern	Action	Bytes	Packets
1010	Drop	200	10
010*	Forward(2)	100	4
011*	Controller	0	0

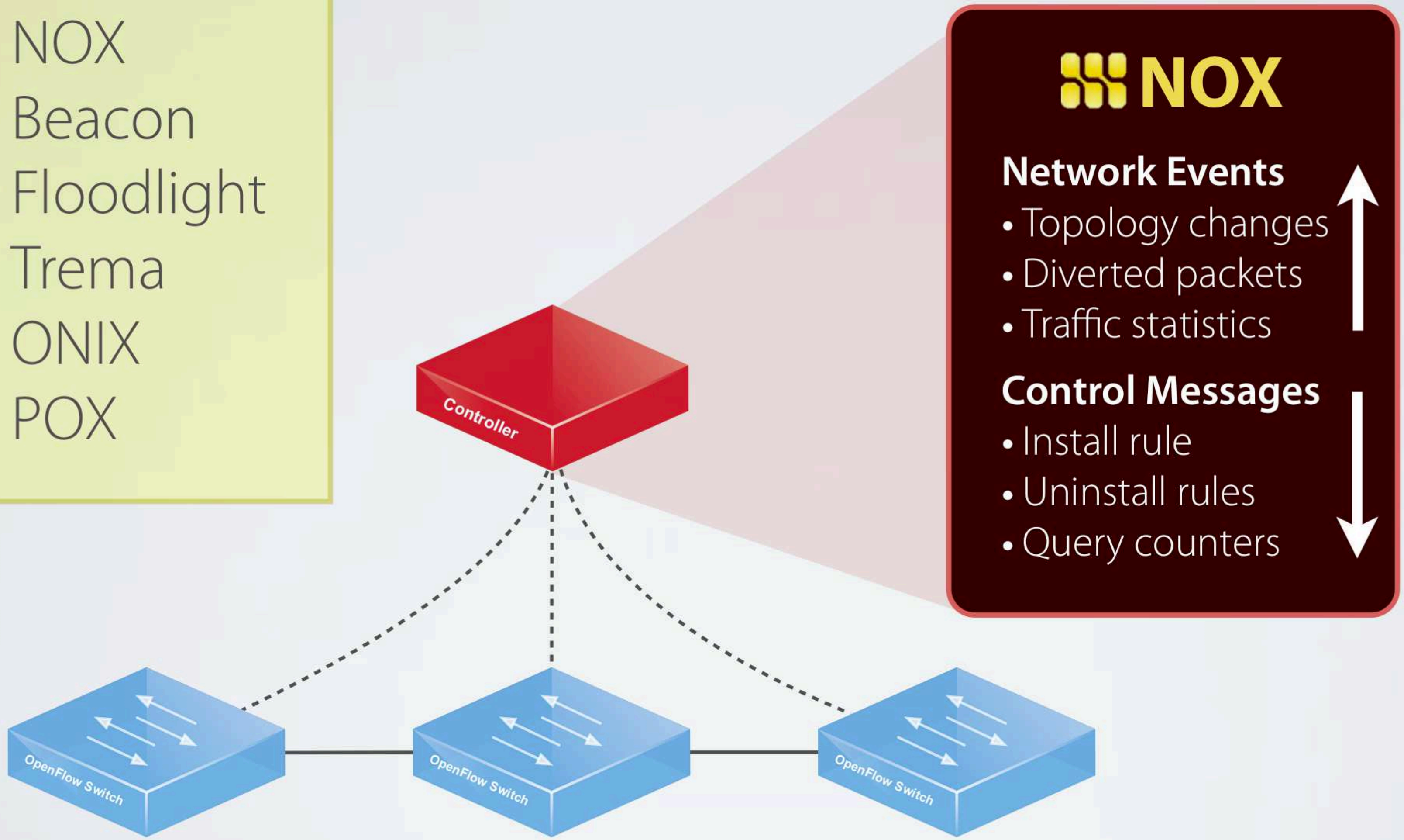
Priority ↓

Controllers

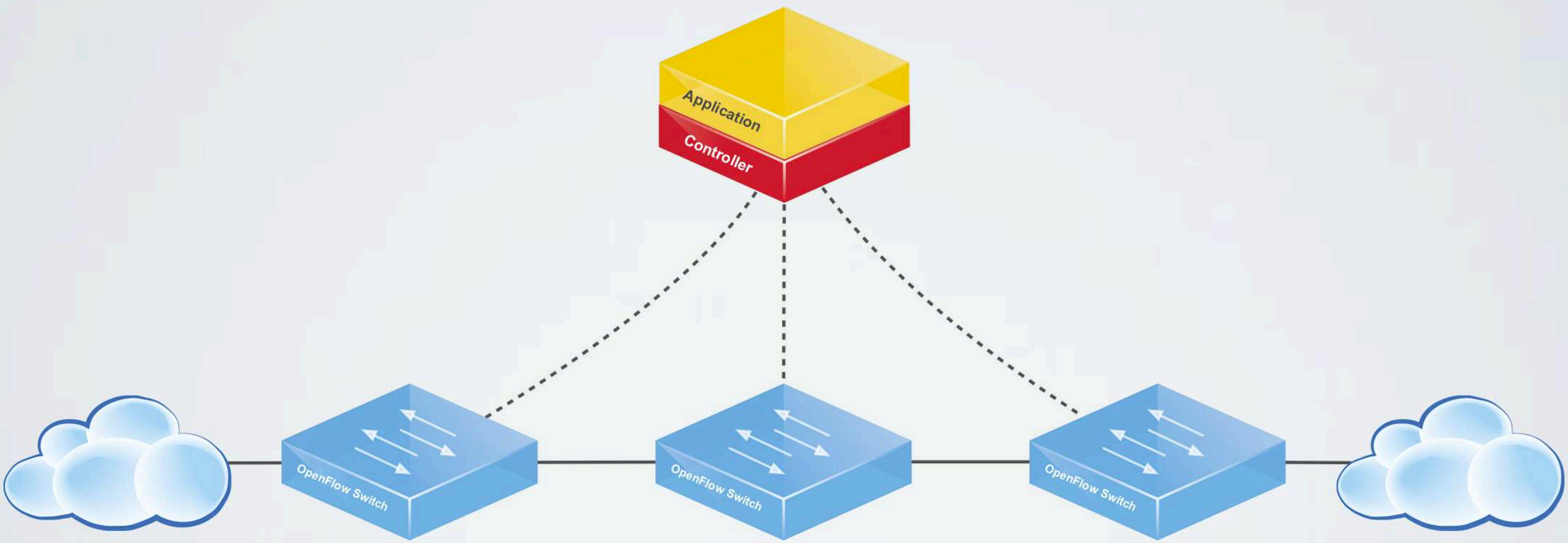


Controllers

- NOX
- Beacon
- Floodlight
- Trema
- ONIX
- POX

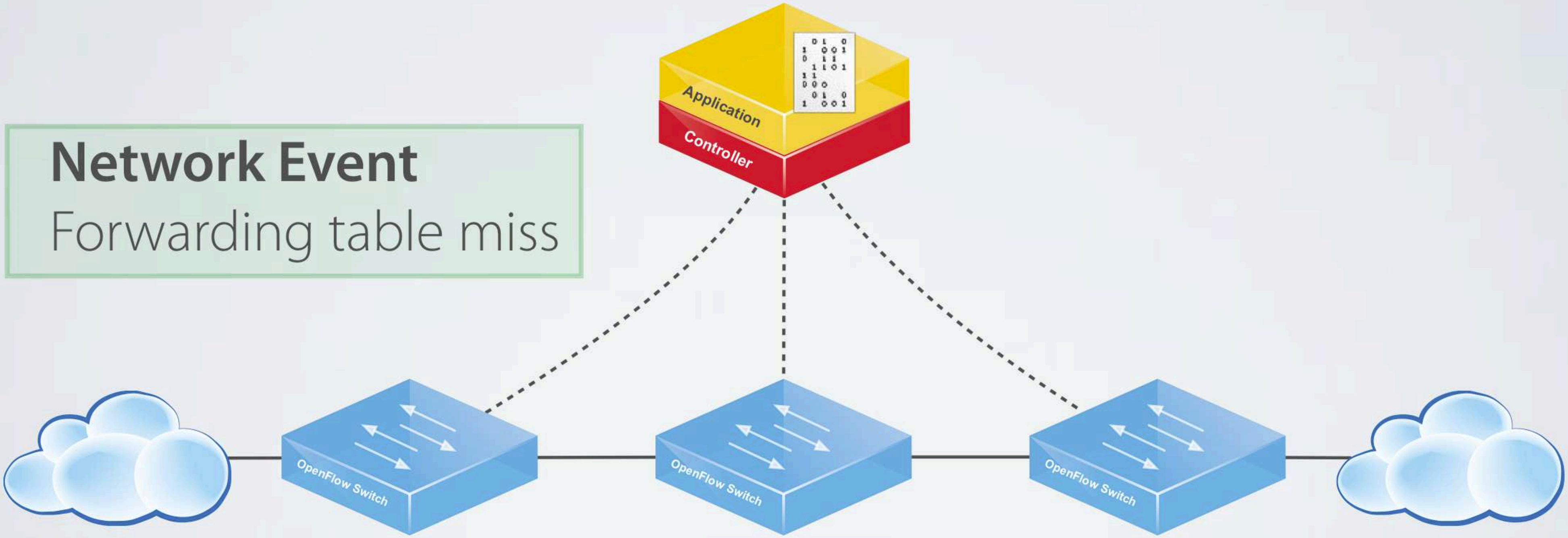


Example: Reactive Applications



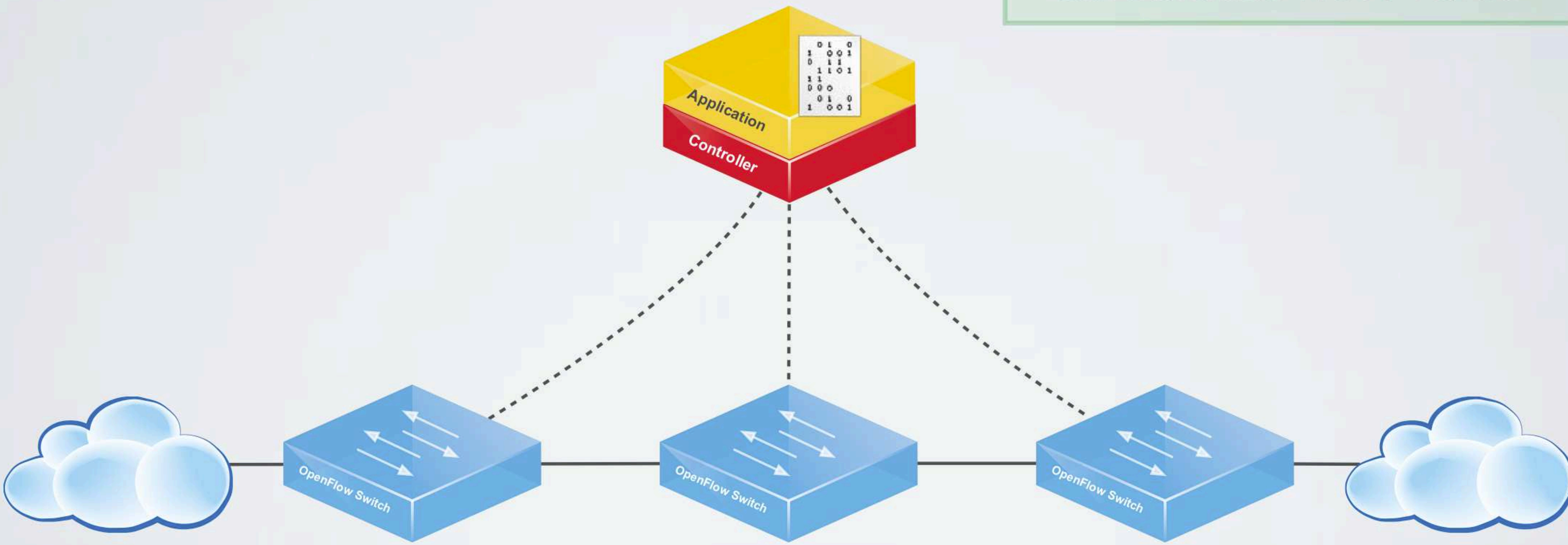
Example: Reactive Applications

Network Event
Forwarding table miss

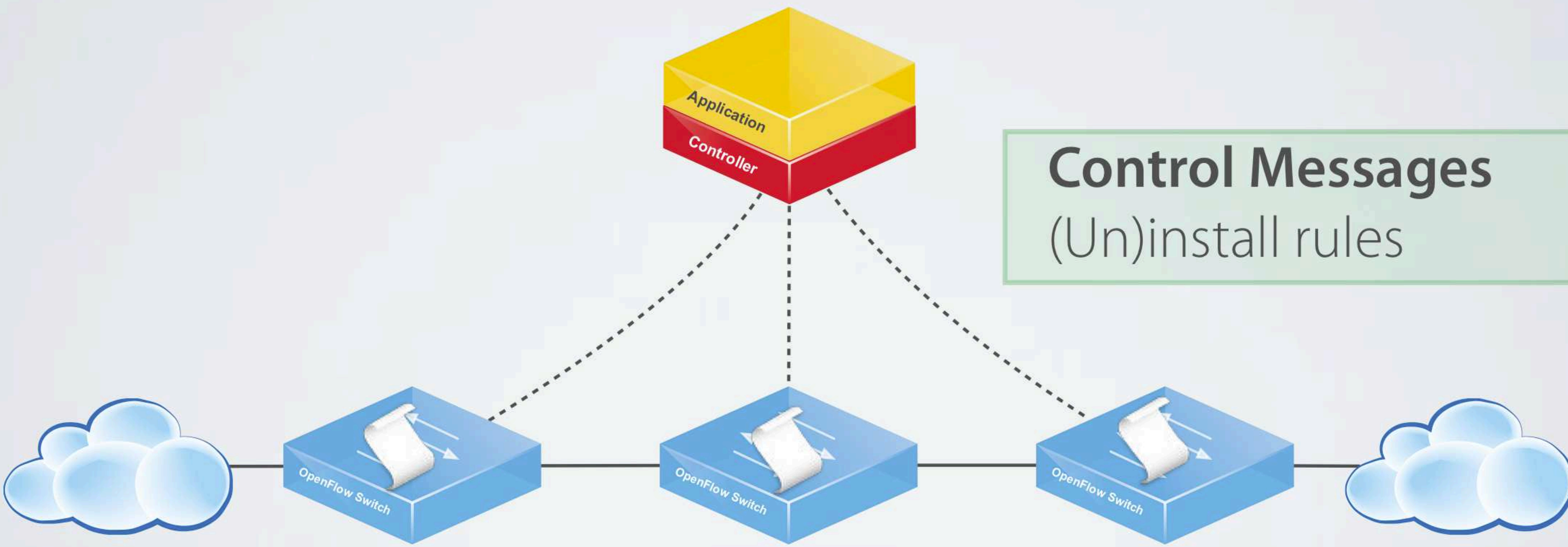


Example: Reactive Applications

Application
Calculates new rules



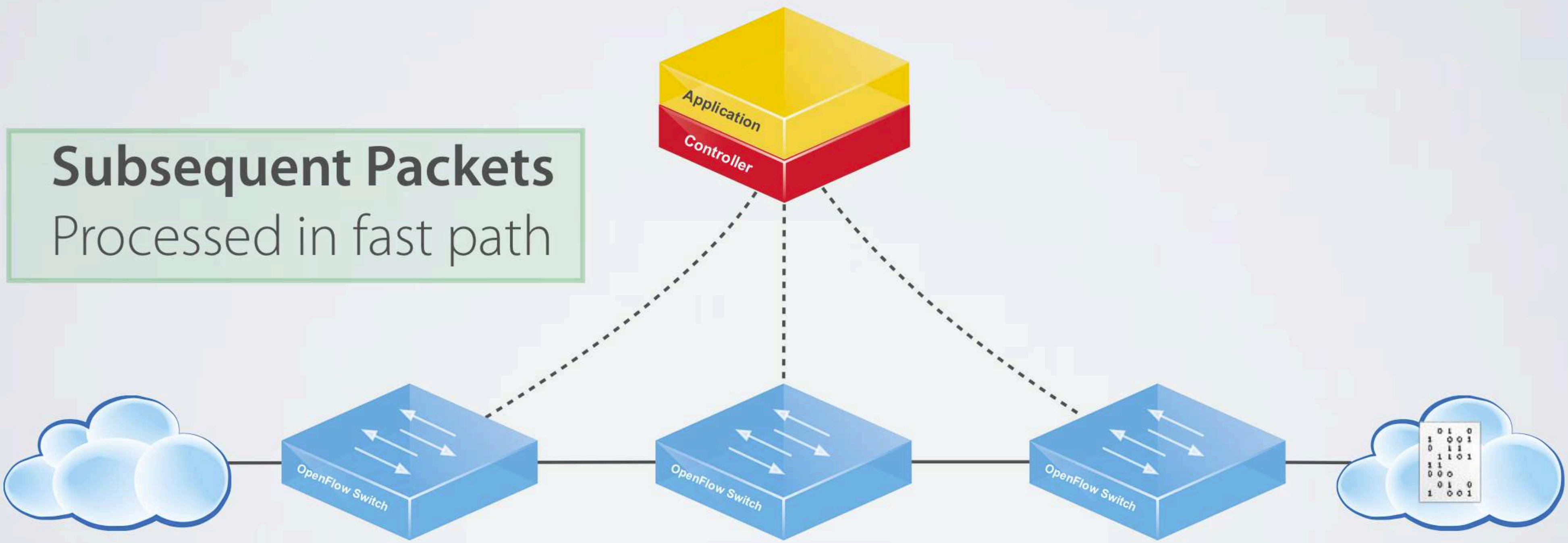
Example: Reactive Applications



Control Messages
(Un)install rules

Example: Reactive Applications

Subsequent Packets
Processed in fast path



Of course, purely proactive applications also possible

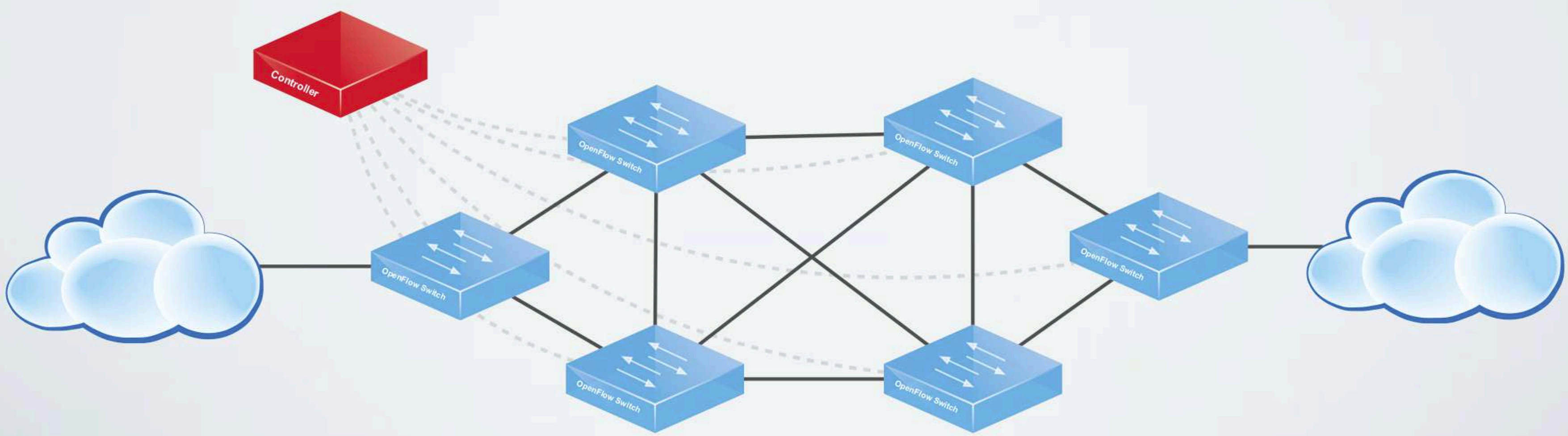
Network-Wide Abstractions

Network-Wide Abstractions

“Holy grail” of network management

Write one program that specifies the behavior of the whole network

- Packet forwarding
- Traffic monitoring
- Access control

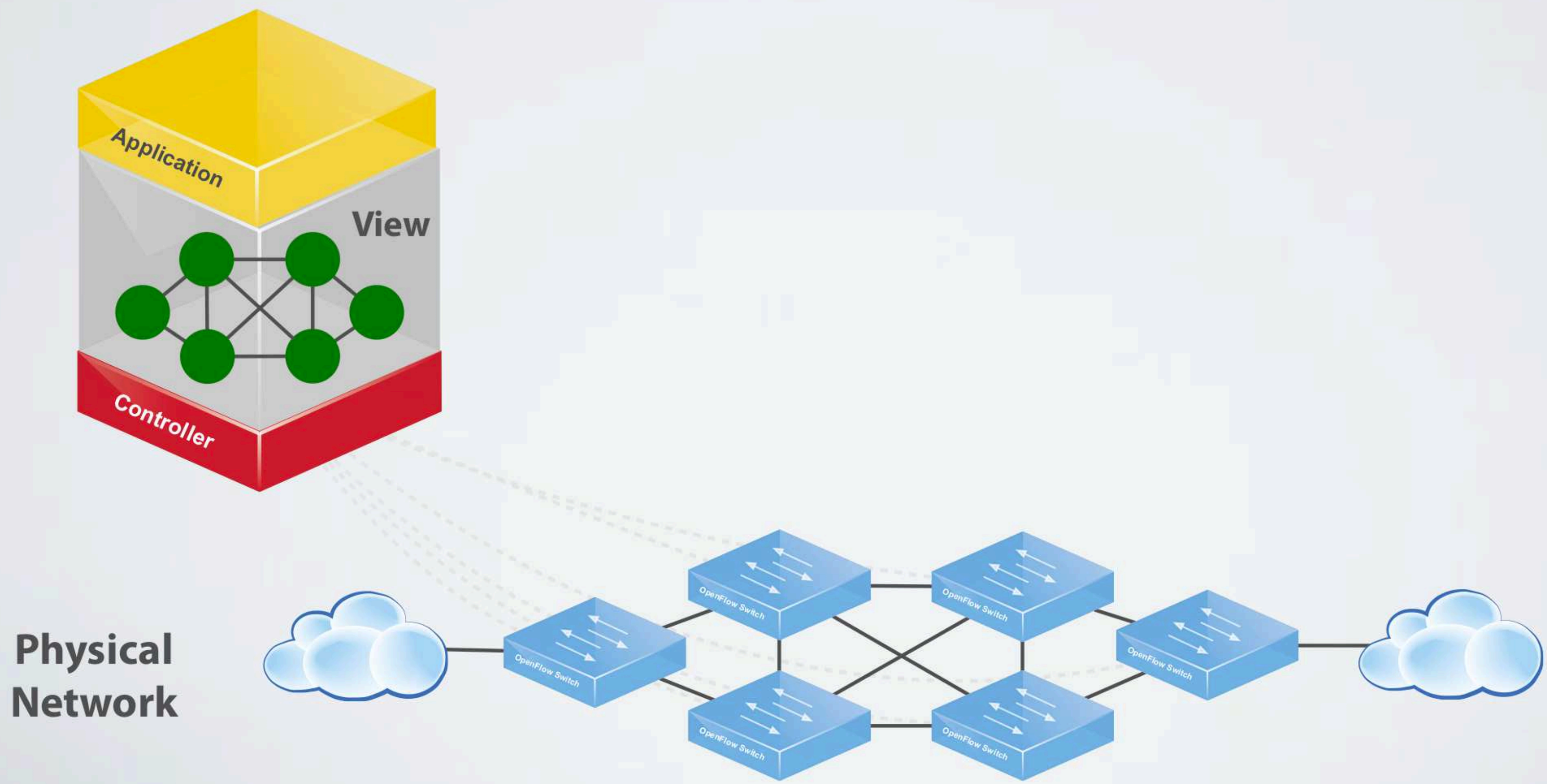


Network-Wide Abstractions

Slogan: configuration = function(view)

NOX

- Global network view
- Eventual consistency

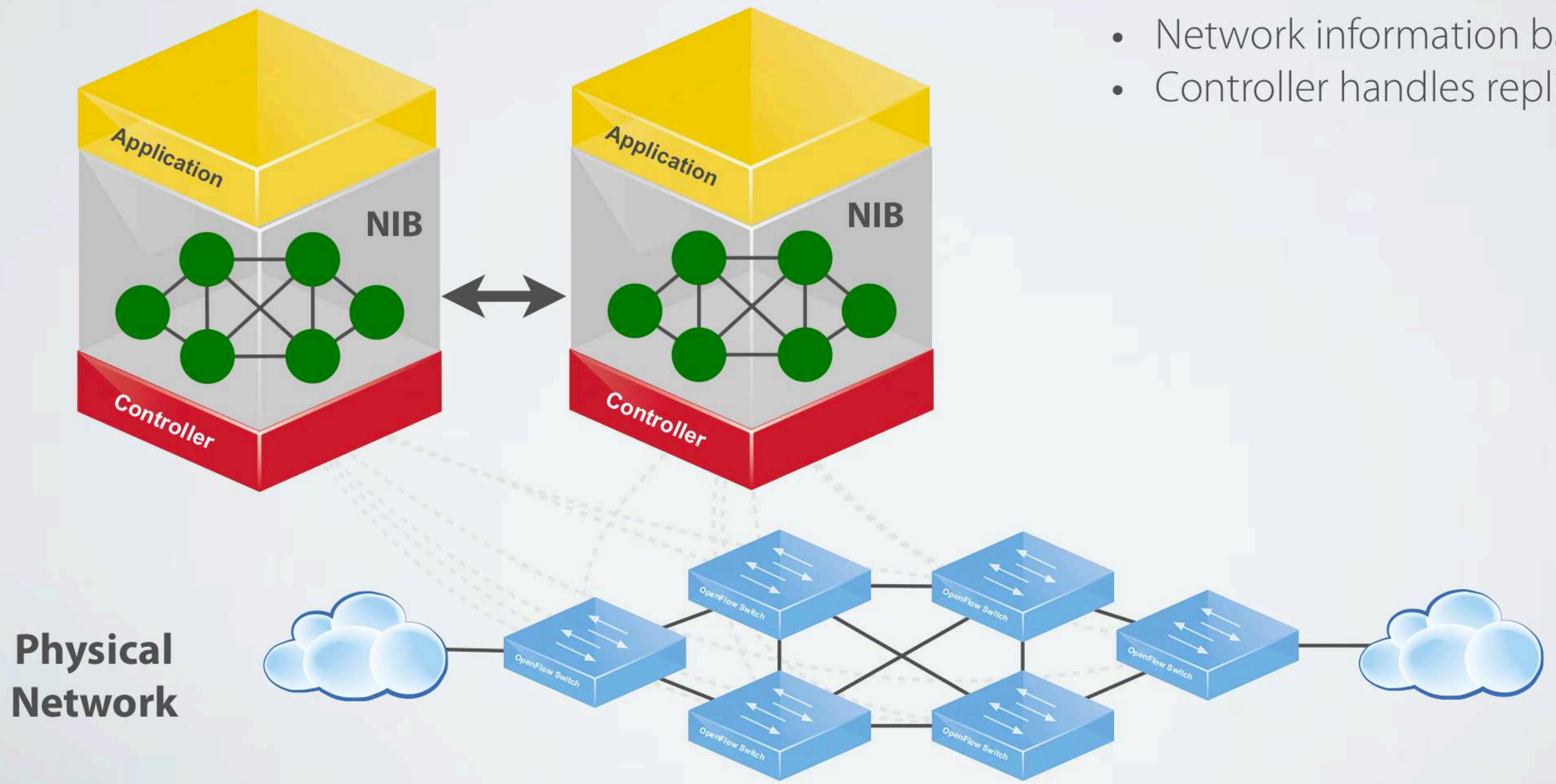


Network-Wide Abstractions

Slogan: configuration = function(view)

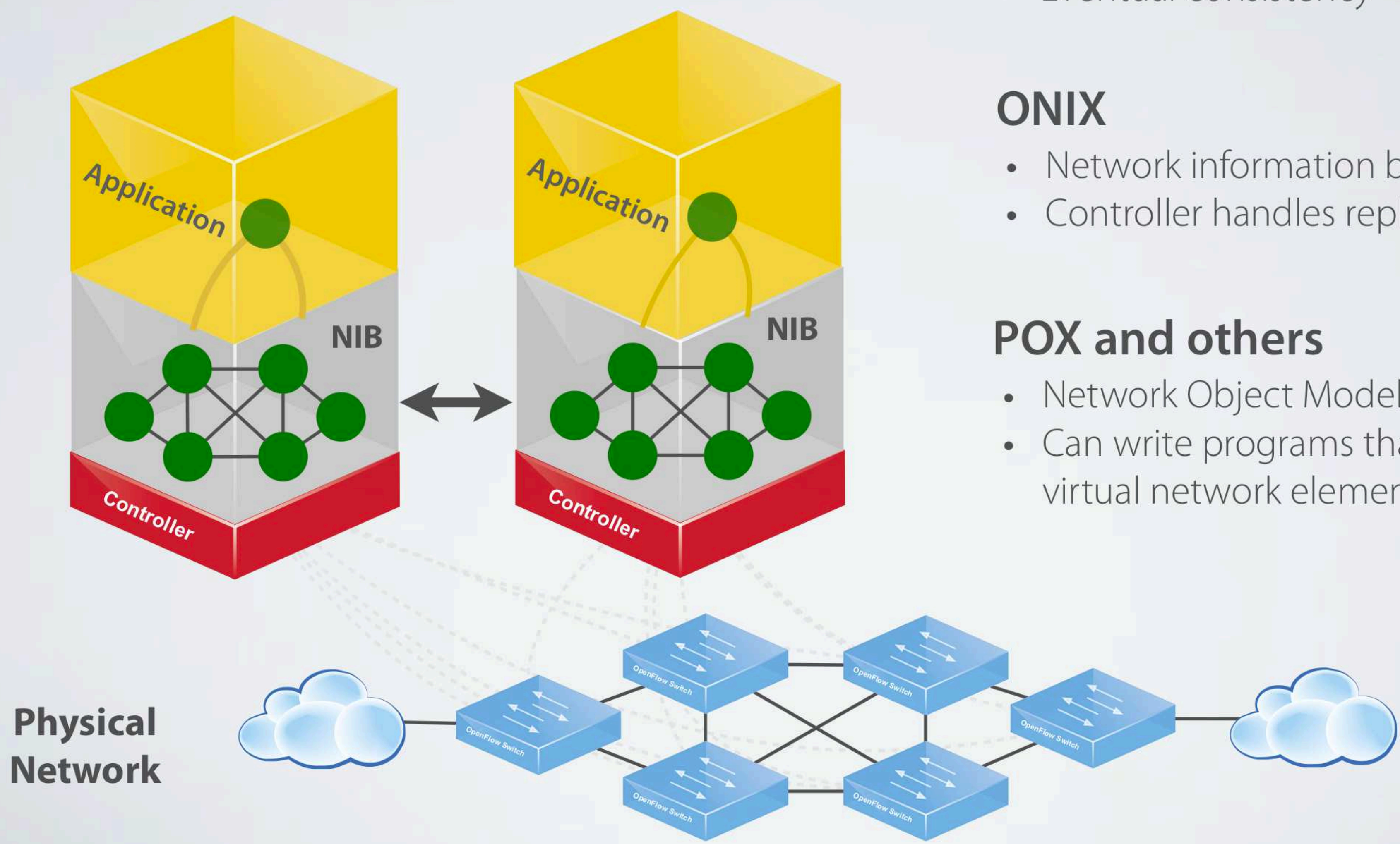
NOX
- Global network view
- Eventual consistency

ONIX
- Network information base (NIB)
- Controller handles replication



Network-Wide Abstractions

Slogan: configuration = function(view)



NOX

- Global network view
- Eventual consistency

ONIX

- Network information base (NIB)
- Controller handles replication

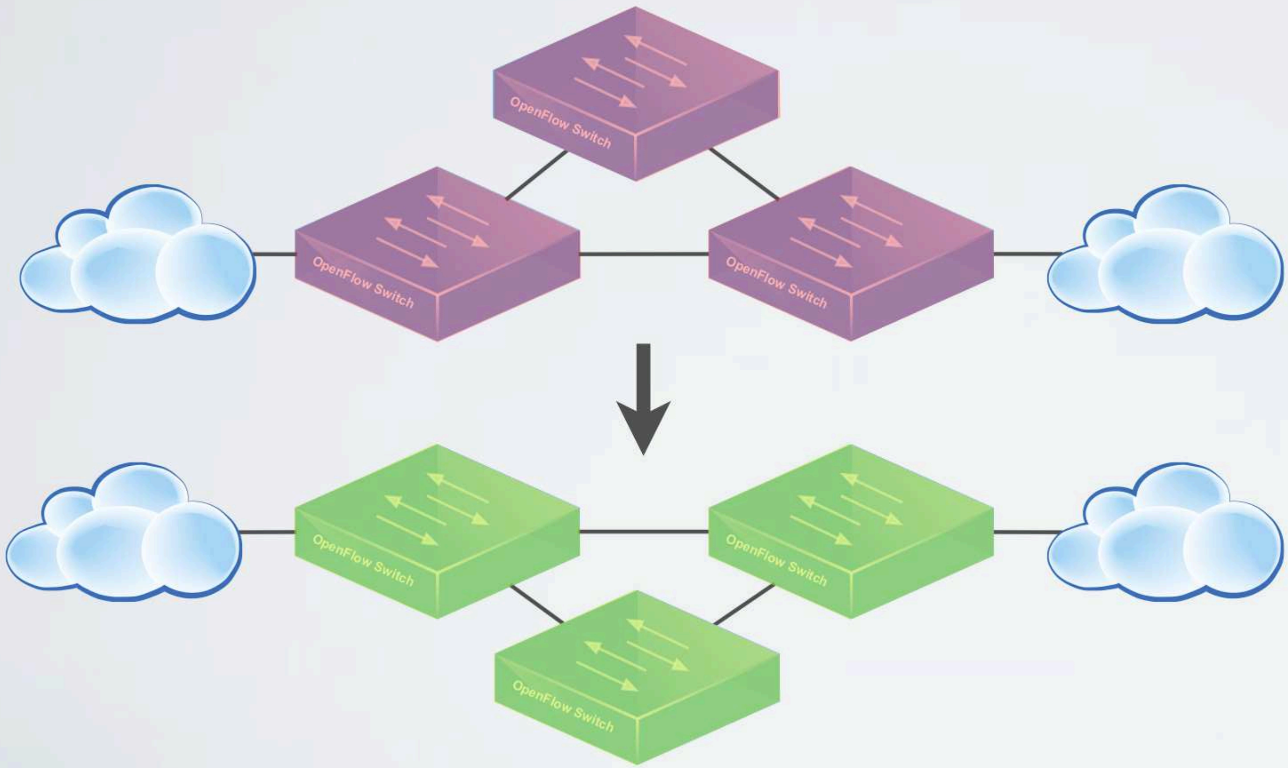
POX and others

- Network Object Model (NOM)
- Can write programs that create virtual network elements

Network Updates

We said configuration = function(view)...

...what happens when the view changes?

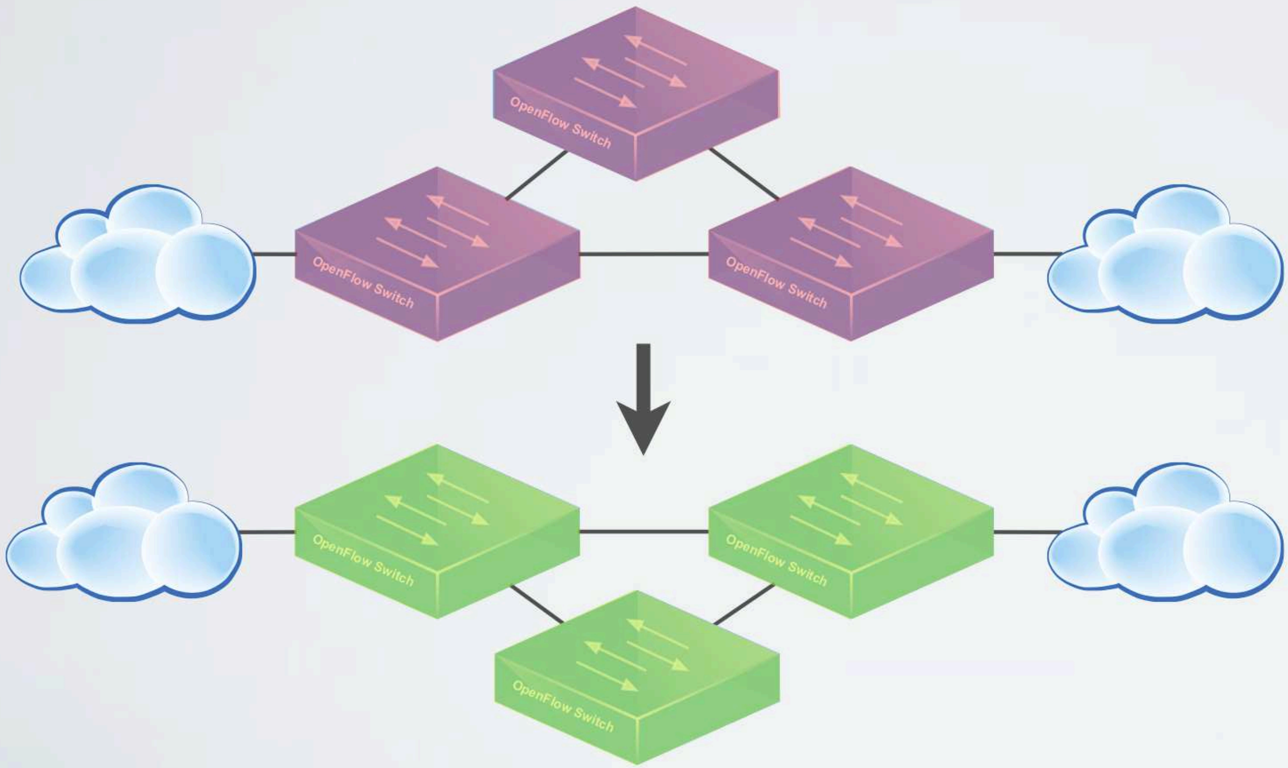


Network Updates
- Routine maintenance
- Unexpected failures
- Traffic engineering
- Changes to ACLs

Network Updates

We said configuration = function(view)...

...what happens when the view changes?



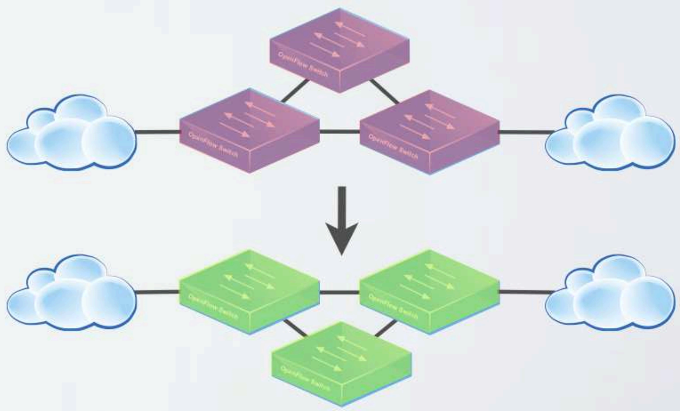
Network Updates
- Routine maintenance
- Unexpected failures
- Traffic engineering
- Changes to ACLs

Desired Invariants
- No lost packets
- No broken connections
- No forwarding loops
- No security holes

Abstractions for Network Update

Challenges

- The network is a distributed system
- Can only update one element at a time
- *Very* easy to make mistakes



Abstractions for Network Update

Challenges

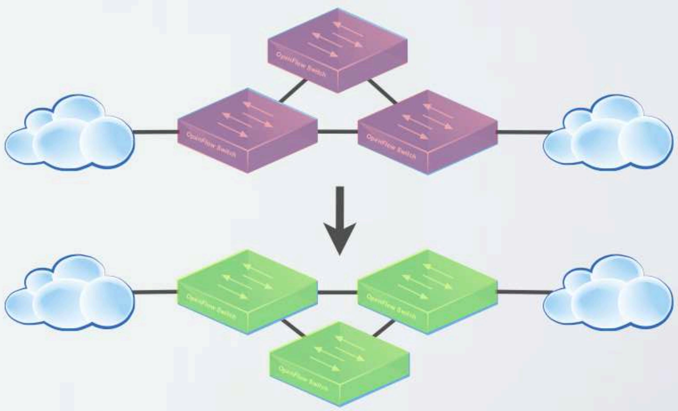
- The network is a distributed system
- Can only update one element at a time
- *Very* easy to make mistakes

amazon webservices™

At 12:47 AM PDT on April 21st, a network change was performed as part of our normal scaling activities...

The traffic shift was executed incorrectly and the traffic was routed onto the lower capacity redundant network. This led to a "re-mirroring storm"...

The trigger for this event was a **network configuration change**.



Abstractions for Network Update

Challenges

- The network is a distributed system
- Can only update one element at a time
- *Very* easy to make mistakes

Possible Approaches

1. Programmer specifies update protocol

2. Controller provides an abstraction

```
update(config)
```

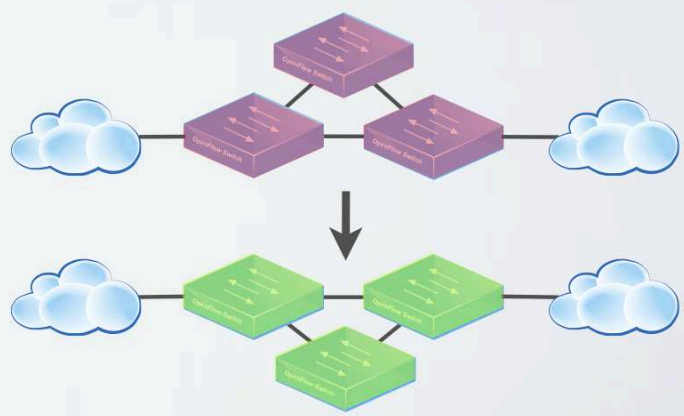
with "reasonable" semantics

amazon webservices™

At 12:47 AM PDT on April 21st, a network change was performed as part of our normal scaling activities...

The traffic shift was executed incorrectly and the traffic was routed onto the lower capacity redundant network. This led to a "re-mirroring storm"...

The trigger for this event was a **network configuration change**.

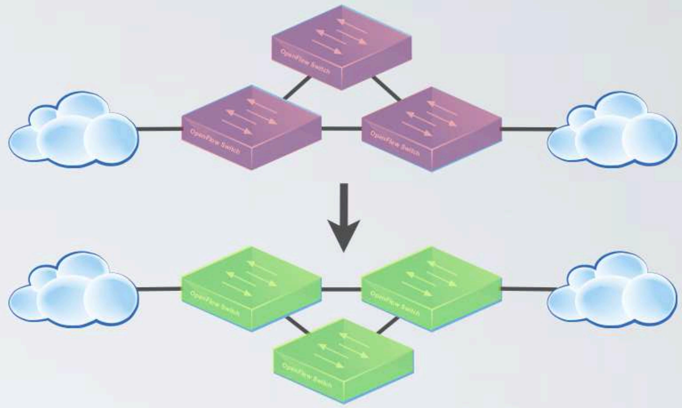


Abstractions for Network Update

[SIGCOMM '12]

Atomic Updates

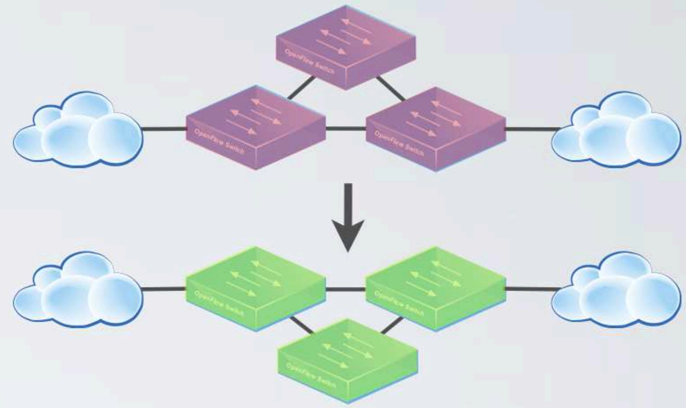
- Seem sensible...
- ...but are costly to implement...
- ...and reasoning about effects on in-flight packets is hard!



Abstractions for Network Update [SIGCOMM '12]

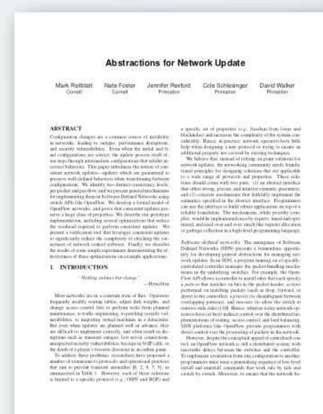
Atomic Updates

- Seem sensible...
- ...but are costly to implement...
- ...and reasoning about effects on in-flight packets is hard!



Per-Packet Consistent Updates

Every packet processed with the old configuration or the new configuration, but not a mixture of the two

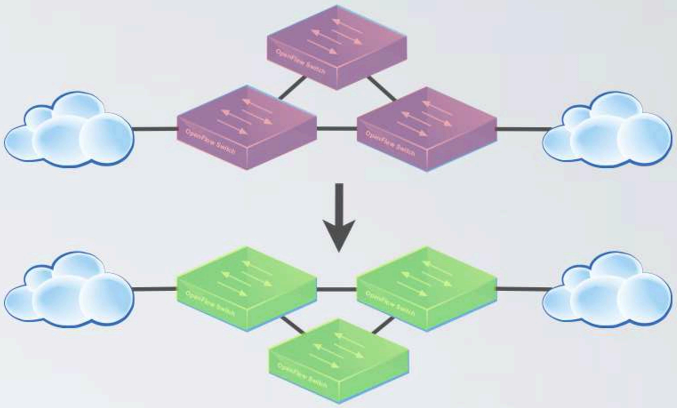

Abstractions for Network Update

Mark Reitblatt (Cornell), Nate Foster (Cornell), Jennifer Rexford (Princeton), Cole Schlesinger (Princeton), David Walker (Princeton)

Abstractions for Network Update [SIGCOMM '12]

Atomic Updates

- Seem sensible...
- ...but are costly to implement...
- ...and reasoning about effects on in-flight packets is hard!

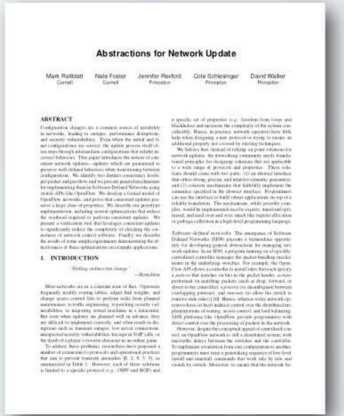


Per-Packet Consistent Updates

Every packet processed with the old configuration or the new configuration, but not a mixture of the two

Per-Flow Consistent Updates

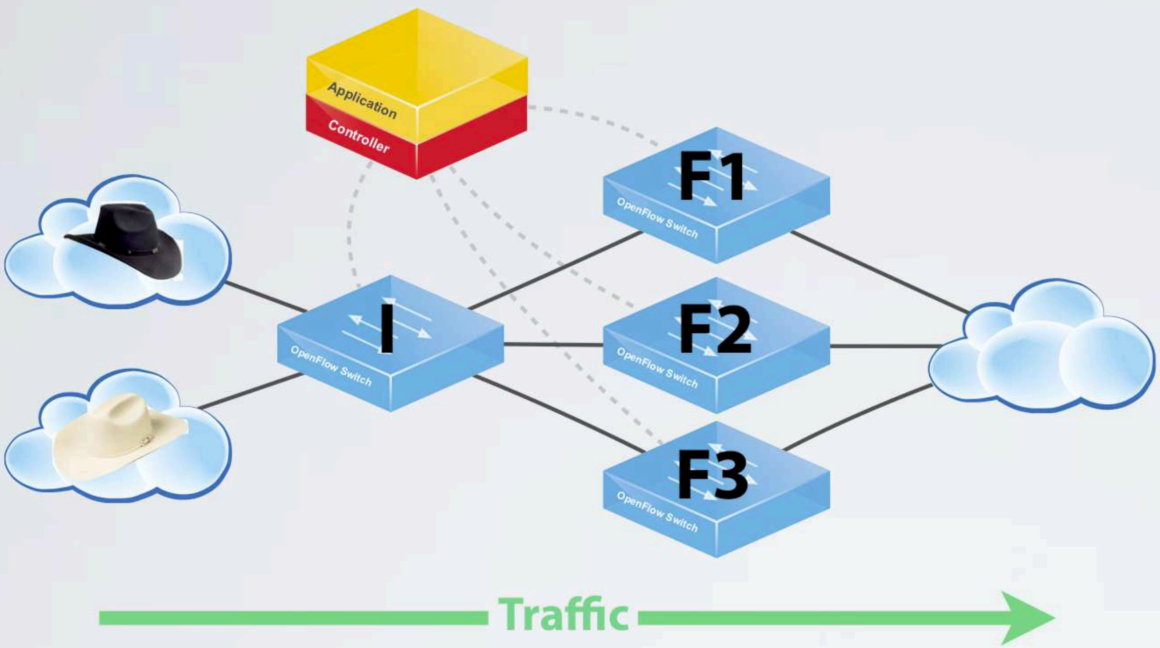
Every packet in the same flow processed with old or new configuration, but not a mixture of the two


Abstractions for Network Update

Mark Reitblatt, Cornell — Nate Foster, Cornell — Jennifer Rexford, Princeton — Cole Schlesinger, Princeton — David Walker, Princeton

ABSTRACT

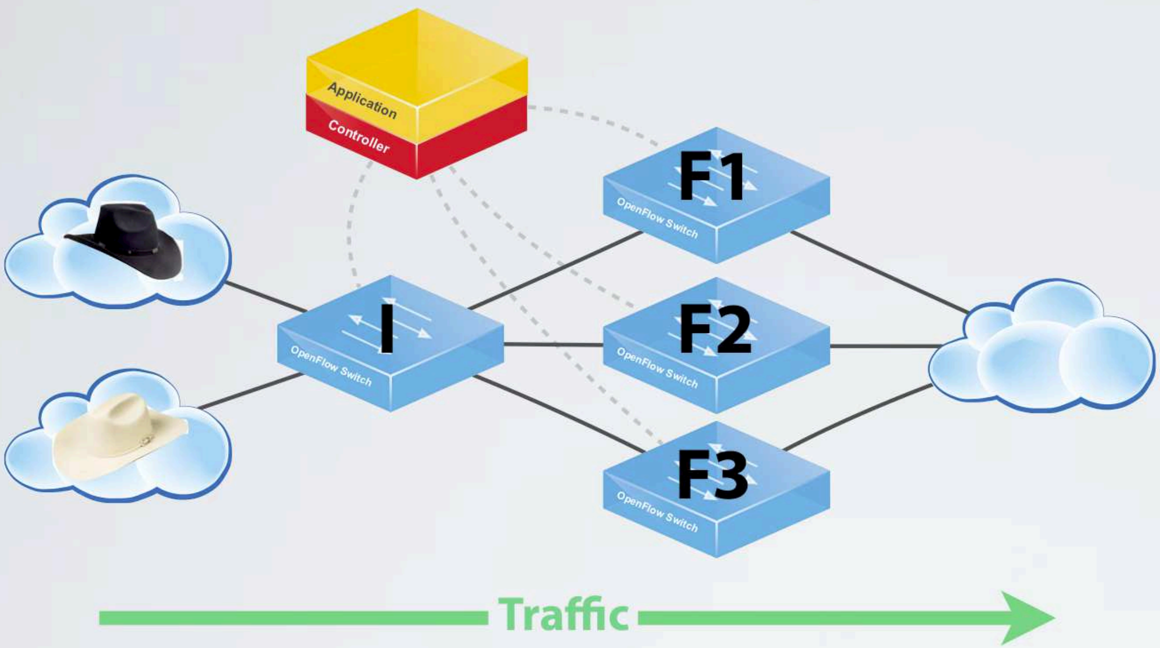
Consistent Updates in Action



Security Policy

Src	Traffic	Action
	Web	Allow
	Non-web	Drop
	Any	Allow

Consistent Updates in Action



Security Policy

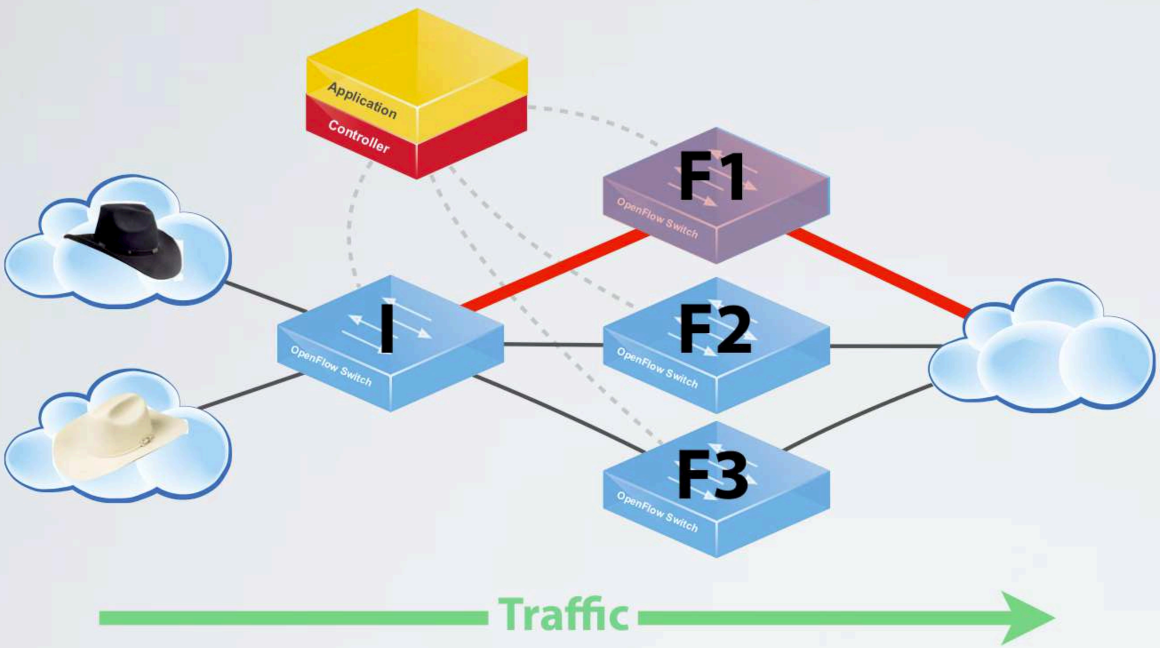
Src	Traffic	Action
	Web	Allow
	Non-web	Drop
	Any	Allow

Configuration A

Process black-hat traffic on F1

Process white-hat traffic on {F2,F3}

Consistent Updates in Action



Security Policy

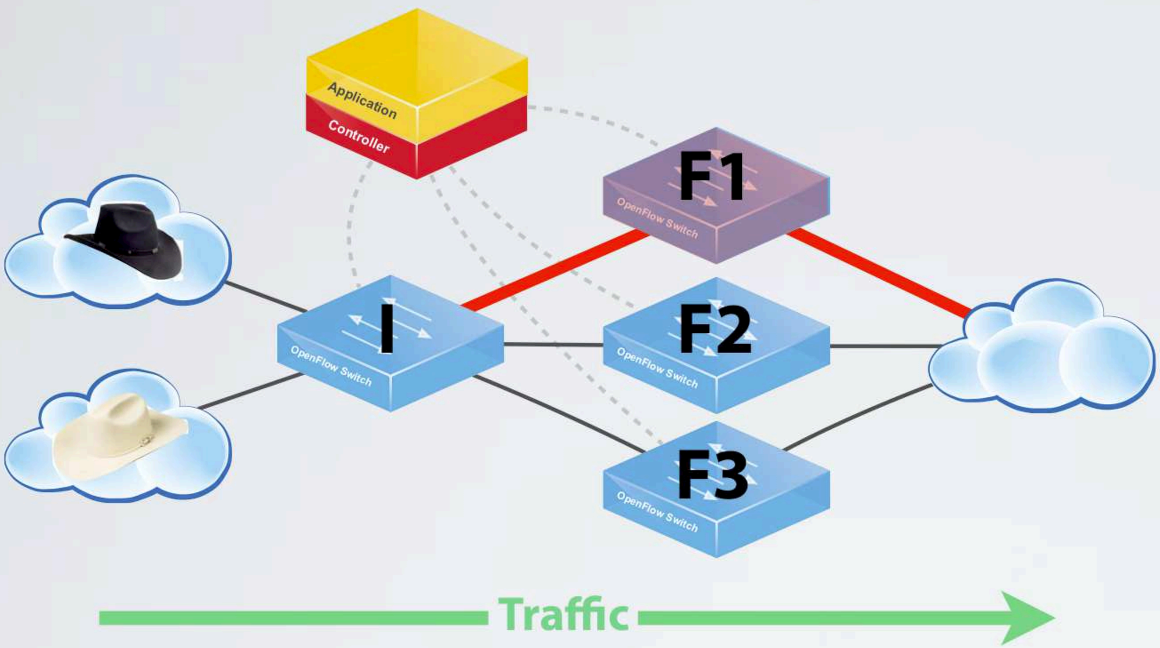
Src	Traffic	Action
	Web	Allow
	Non-web	Drop
	Any	Allow

Configuration A

Process black-hat traffic on F1

Process white-hat traffic on {F2,F3}

Consistent Updates in Action



Security Policy

Src	Traffic	Action
	Web	Allow
	Non-web	Drop
	Any	Allow

Configuration A

Process black-hat traffic on F1

Process white-hat traffic on {F2,F3}

?

Configuration B

Process black-hat traffic on {F1,F2}

Process white-hat traffic on F3

Consistent Updates in Action

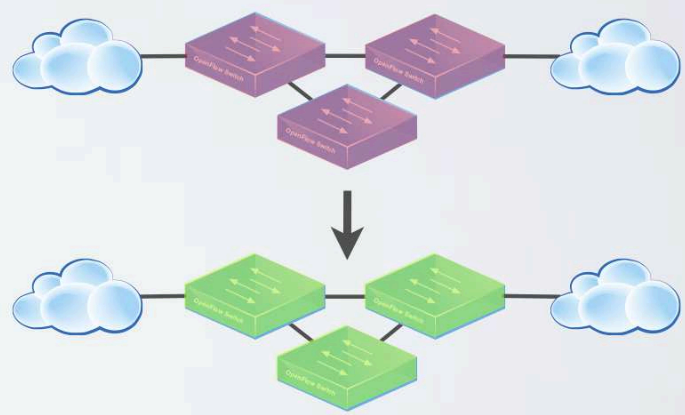
```
# Configuration A
I_
F1
F2
F3
co
```

```
# Configuration B
I_configB = [Rule({IN_PORT:1},[forward(5)]),
             Rule({IN_PORT:2},[forward(6)]),
             Rule({IN_PORT:3},[forward(7)]),
             Rule({IN_PORT:4},[forward(7)])])
F1_configB = [Rule({TP_DST:80}, [forward(2)]),
              Rule({TP_DST:22}, [])])
F2_configB = [Rule({TP_DST:80}, [forward(2)]),
              Rule({TP_DST:22}, [])])
F3_configB = [Rule({},[forward(2)])]
configB = {I:SwitchConfiguration(I_configB),
           F1:SwitchConfiguration(F1_configB),
           F2:SwitchConfiguration(F2_configB),
           F3:SwitchConfiguration(F3_configB)}
```

```
# Main Function
topo = Topo(...)
update(configA, topo)
...wait for traffic load to shift...
update(configB, topo)
```

Security Policy

Src	Traffic	Action
	Web	Allow
	Non-web	Drop
	Any	Allow



One abstraction, many implementations

Composition principles

- Combine updates, preserve consistency

Two-phase commit

- Construct versioned internal and edge configurations
- Phase 1: Install internal configuration
- Phase 2: Install edge configuration

Pure Extension

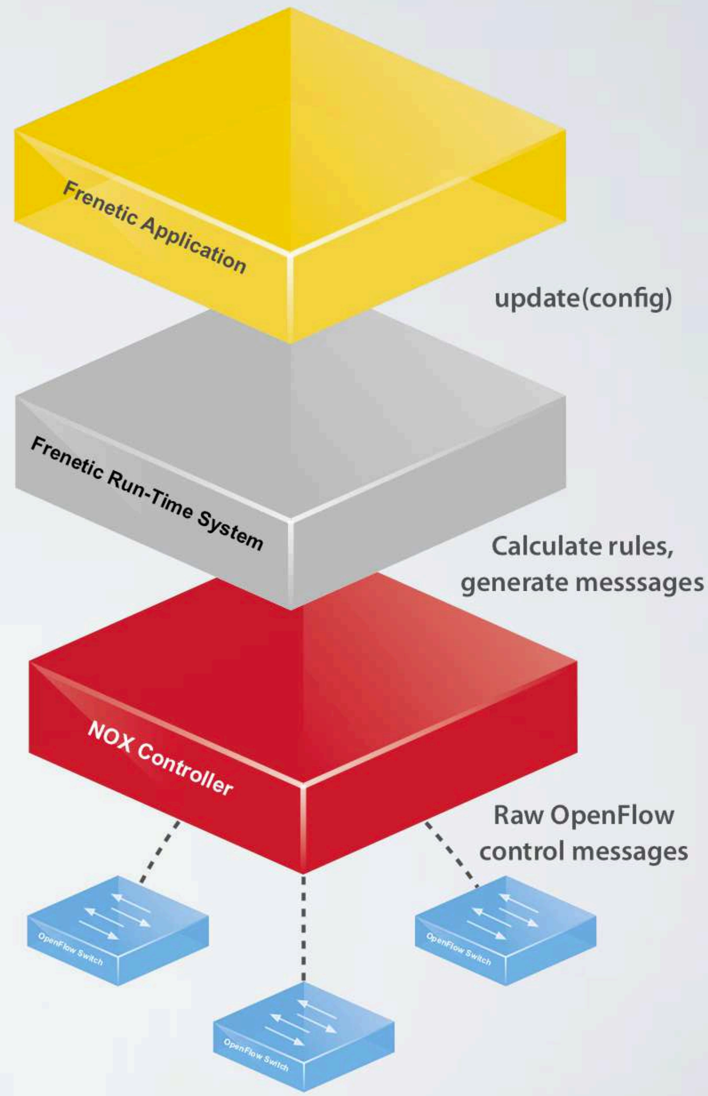
- Update strictly adds paths

Pure Retraction

- Update strictly removes paths

Slice Update

- Update only affects a few switches



Network Updates, Formally

Global Config **Update** **Packet Queue**

$$\langle C, Q \rangle \xrightarrow{u} \langle C', Q' \rangle$$

Network Updates, Formally

Global Config

Update

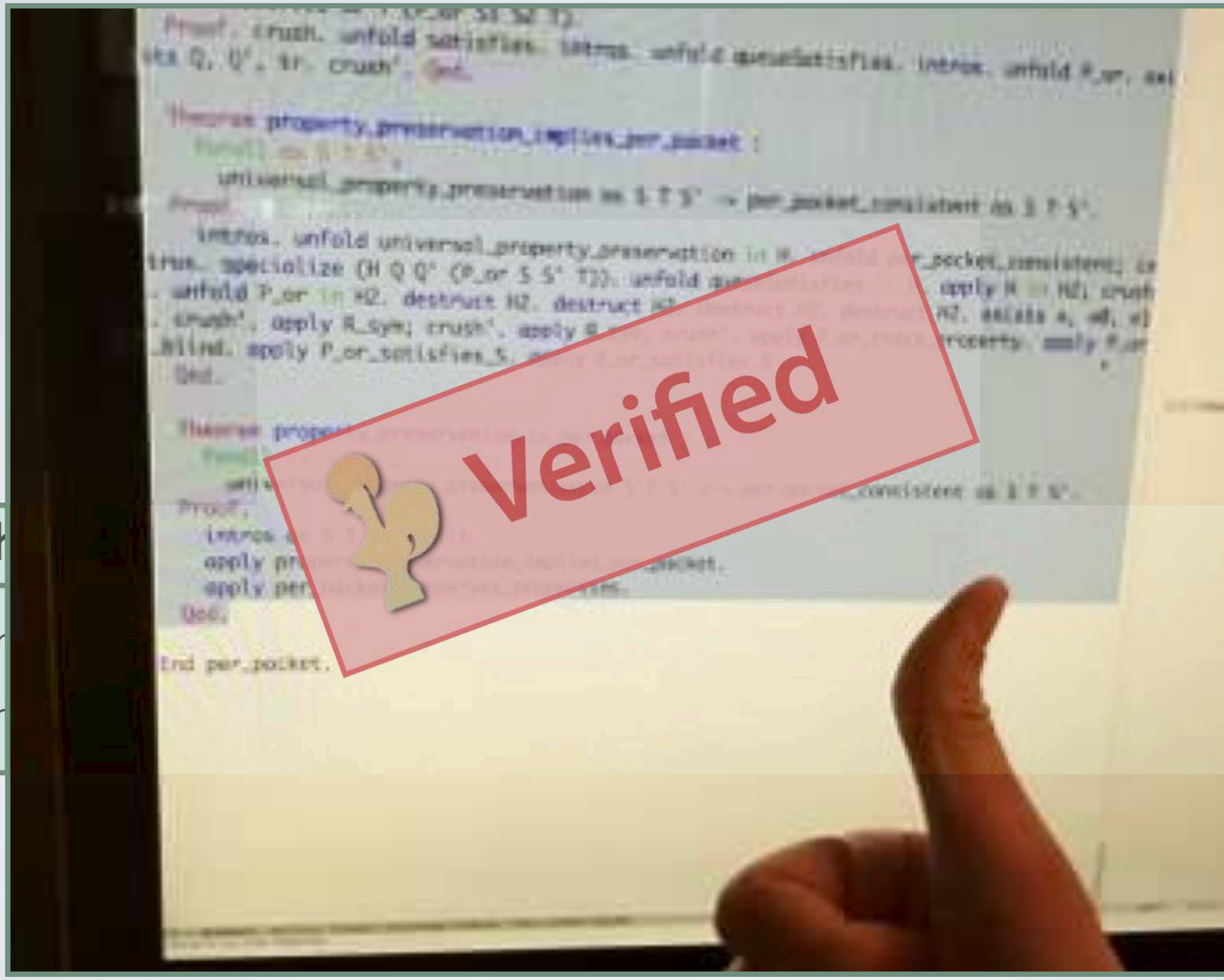
Packet Queue

$$\langle C, Q \rangle \xrightarrow{u} \langle C', Q' \rangle$$

Theorem

An update u from C_1 to C_2 is per-packet consistent if and only if it preserves all properties satisfied by C_1 and C_2.

Network Updates, Formally



Th

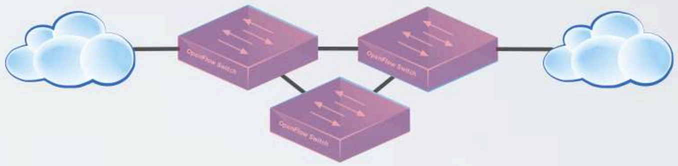
Ar

or

Verification

Corollary

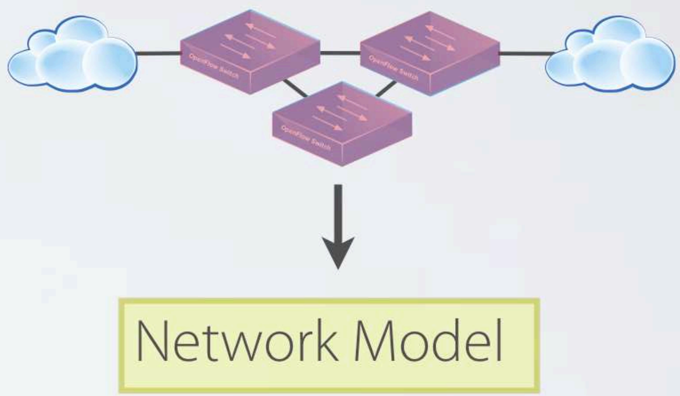
To verify that a property is invariant across an update, simply check that the old and new configurations both satisfy it



Verification

Corollary

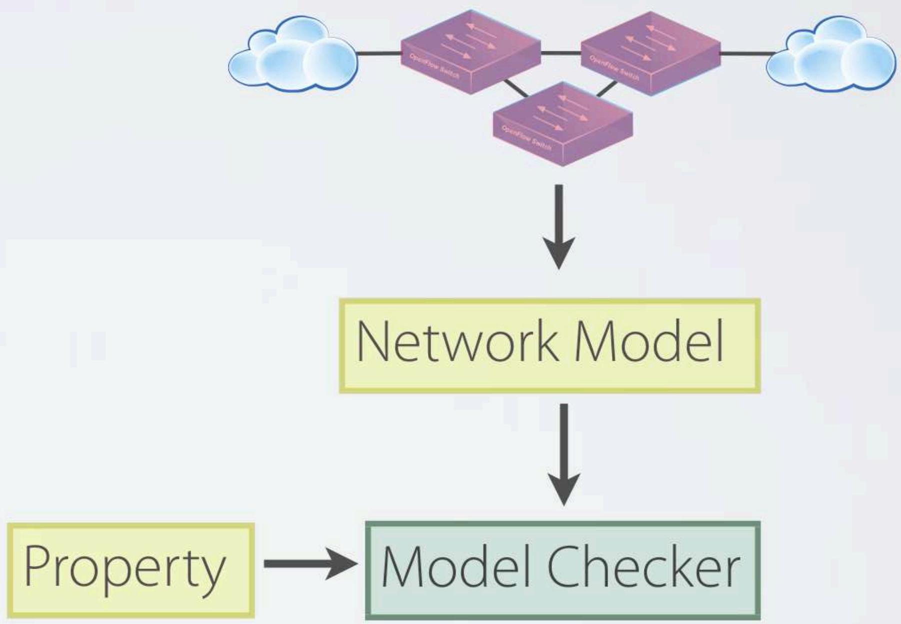
To verify that a property is invariant across an update, simply check that the old and new configurations both satisfy it



Verification

Corollary

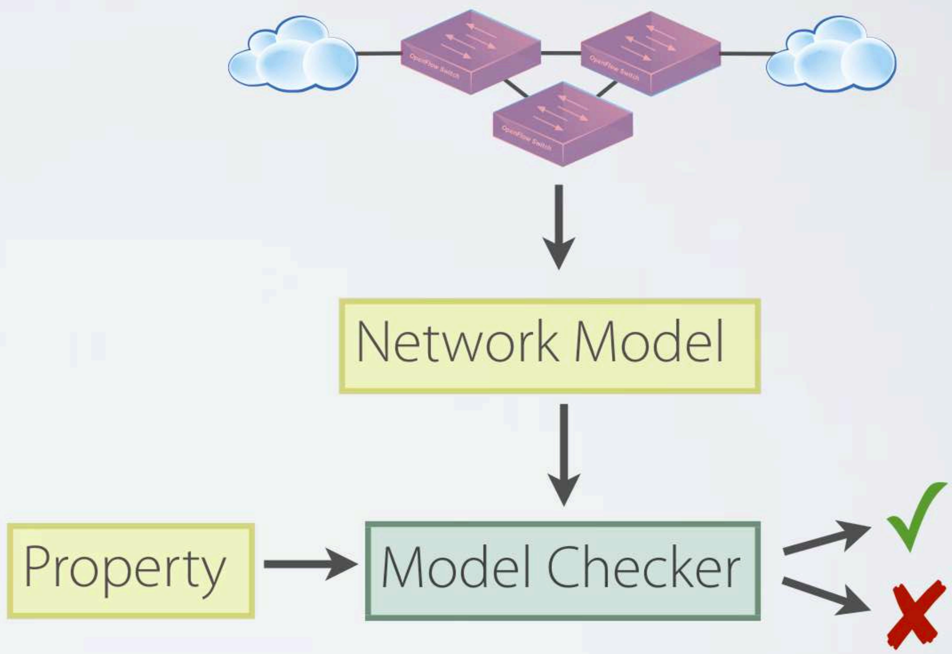
To verify that a property is invariant across an update, simply check that the old and new configurations both satisfy it



Verification

Corollary

To verify that a property is invariant across an update, simply check that the old and new configurations both satisfy it



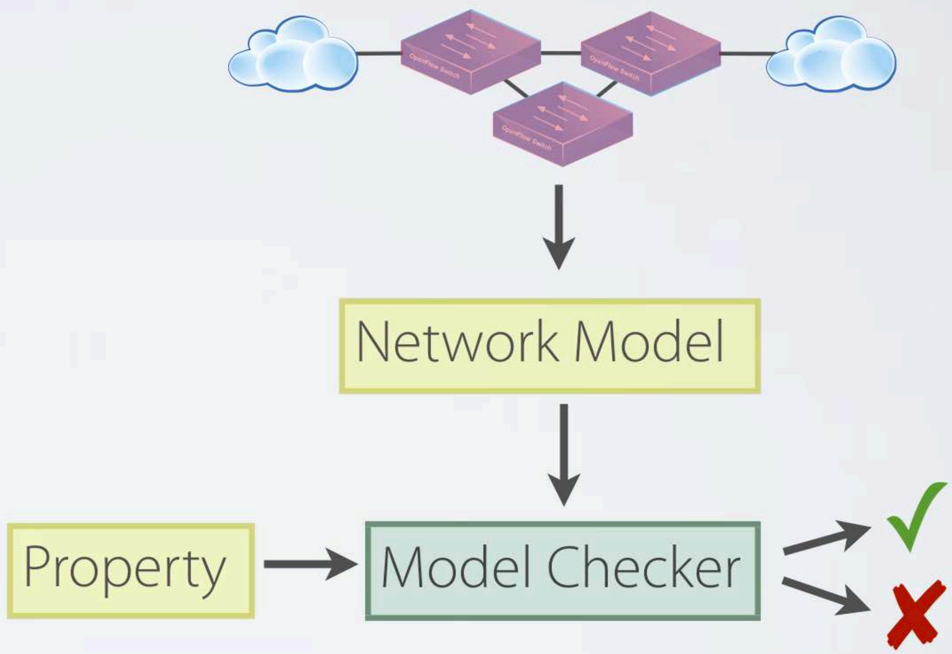
Verification

Corollary

To verify that a property is invariant across an update, simply check that the old and new configurations both satisfy it

Properties

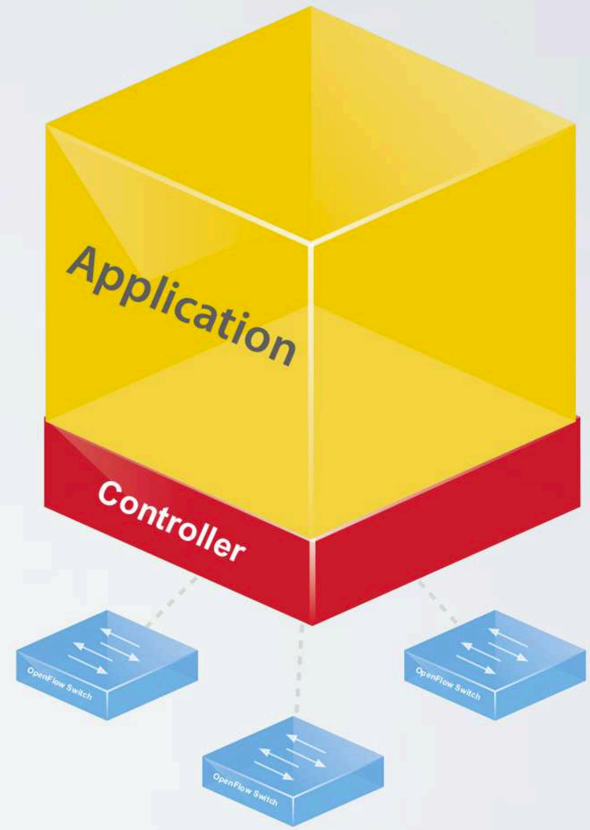
- Connectivity
- Loop freedom
- Blackhole freedom
- Access control
- Waypointing
- Totality



Modularity

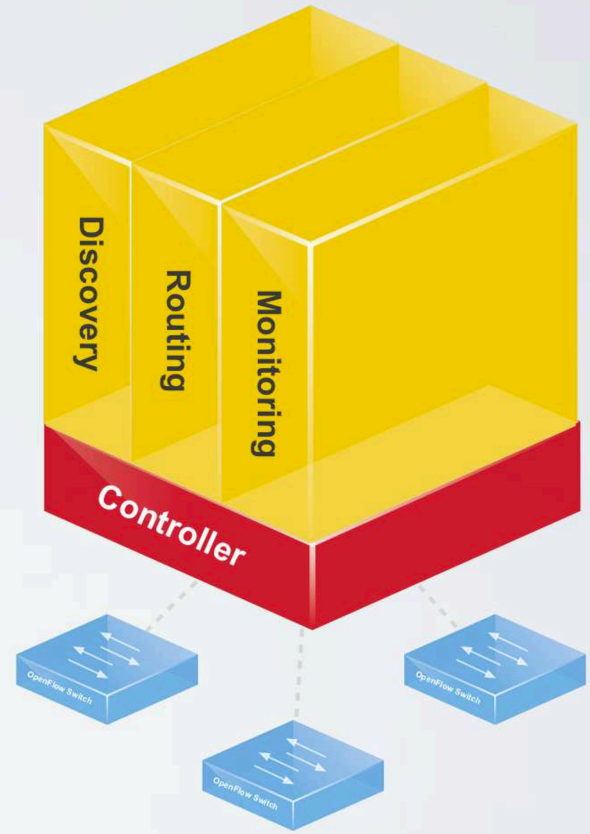
Composing Programs

Many applications decompose naturally into components



Composing Programs

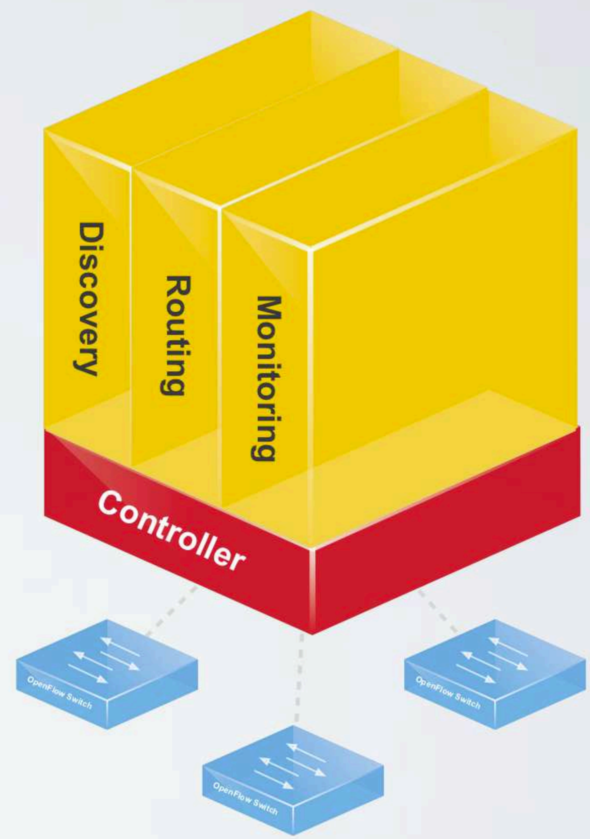
Many applications decompose naturally into components



Composing Programs

Many applications decompose naturally into components

Want to write these components once, and use them many times...



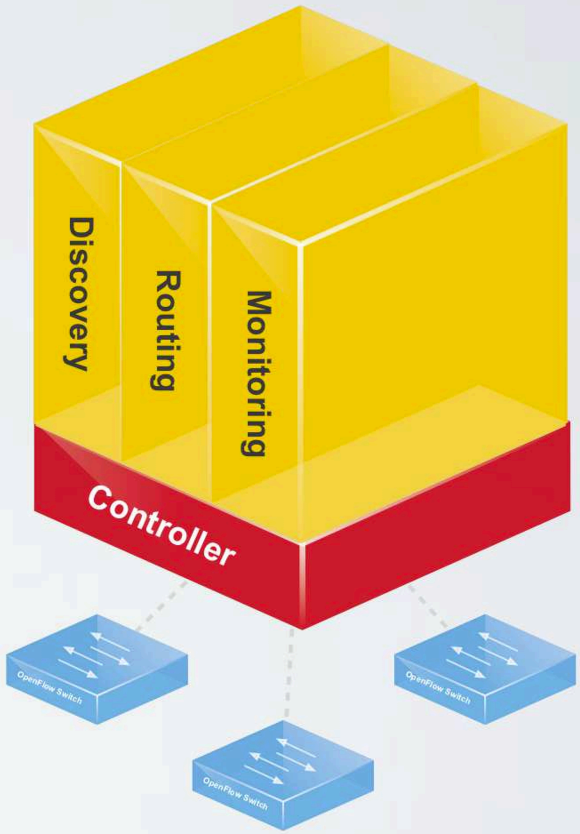
Composing Programs

Many applications decompose naturally into components

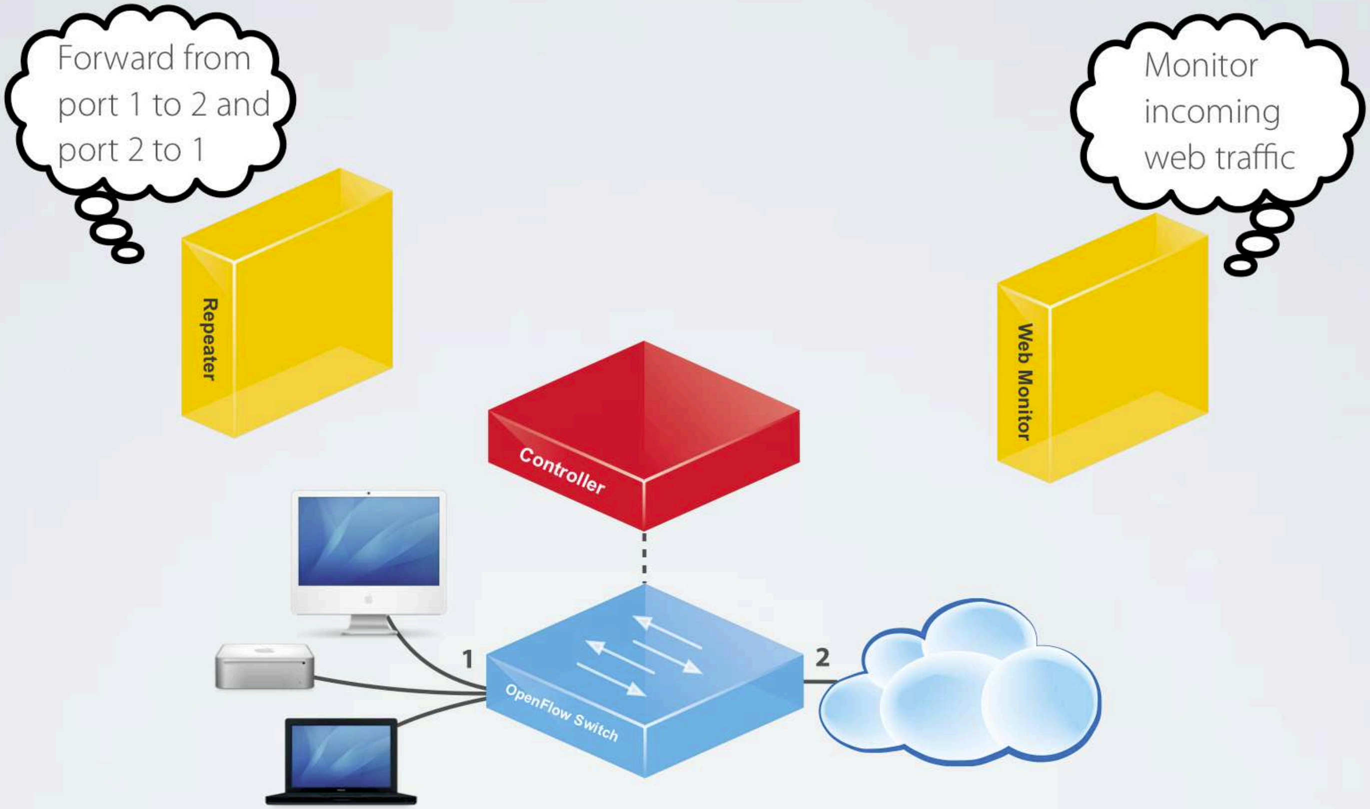
Want to write these components once, and use them many times...

...but this is difficult to achieve using current controllers

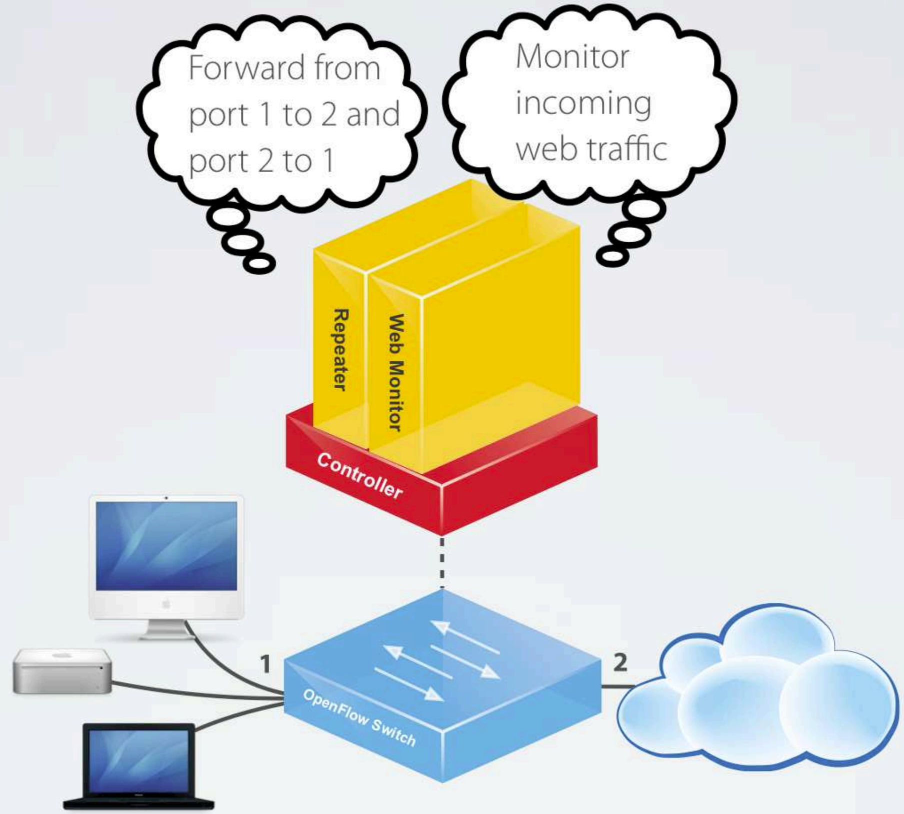
- Network events processed by each component (in some specified order)
- May either propagate or suppress each event
- Components manipulate switch state directly
- State generated by one component can be accessed by others



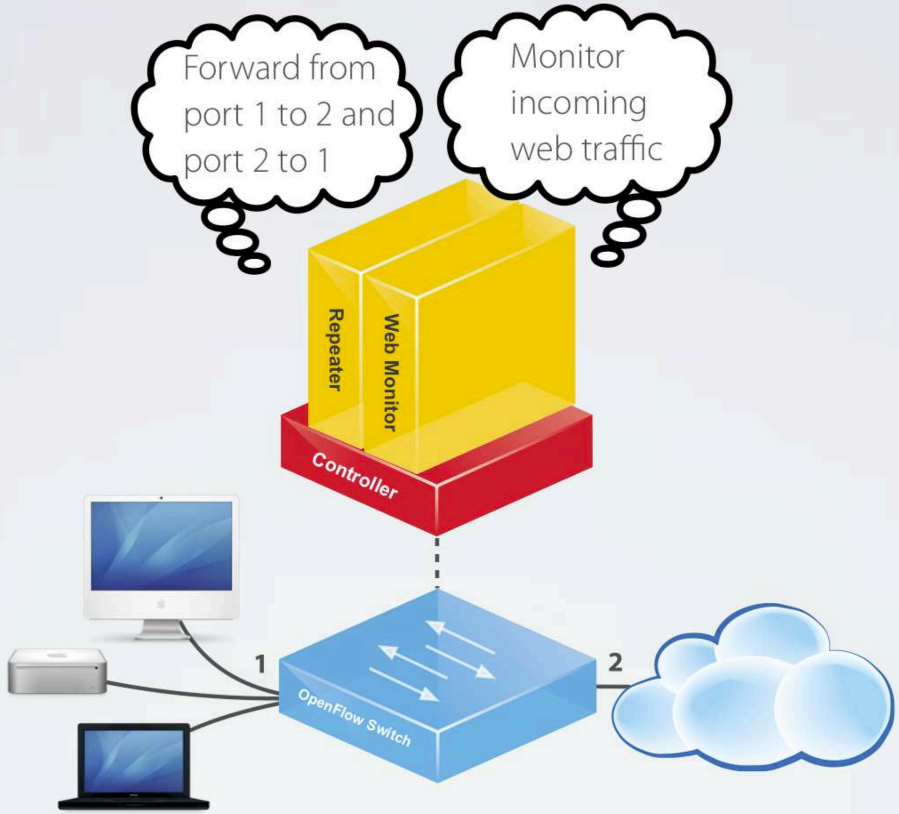
Modularity Problems



Modularity Problems



Modularity Problems



Problems

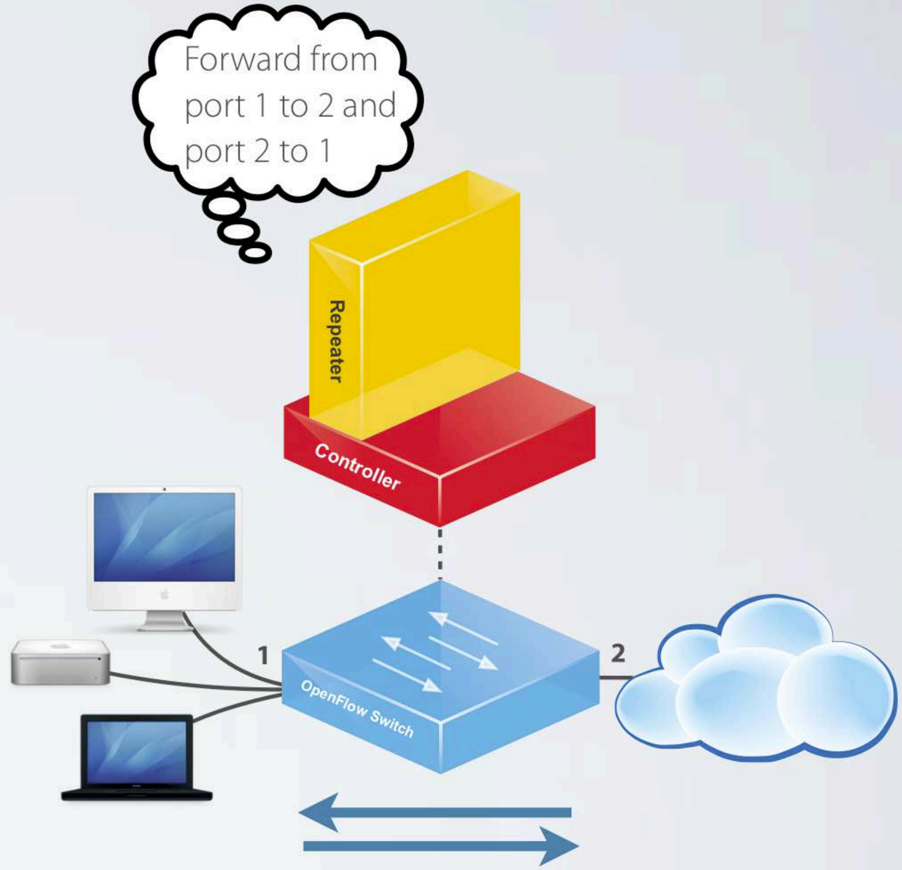
- Repeater rules too coarse grained
- Monitoring rules don't forward

Example: Repeater + monitor

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)
```



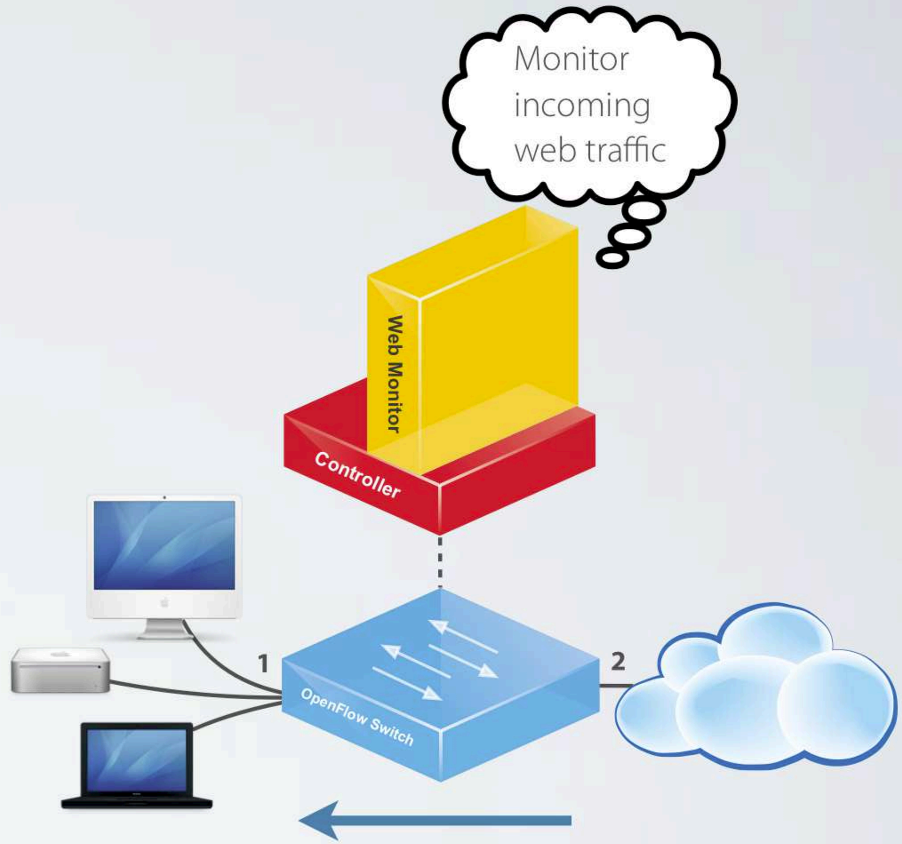
When a switch joins the network, install two rules

Example: Repeater + monitor

Web Monitor

```
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)
```



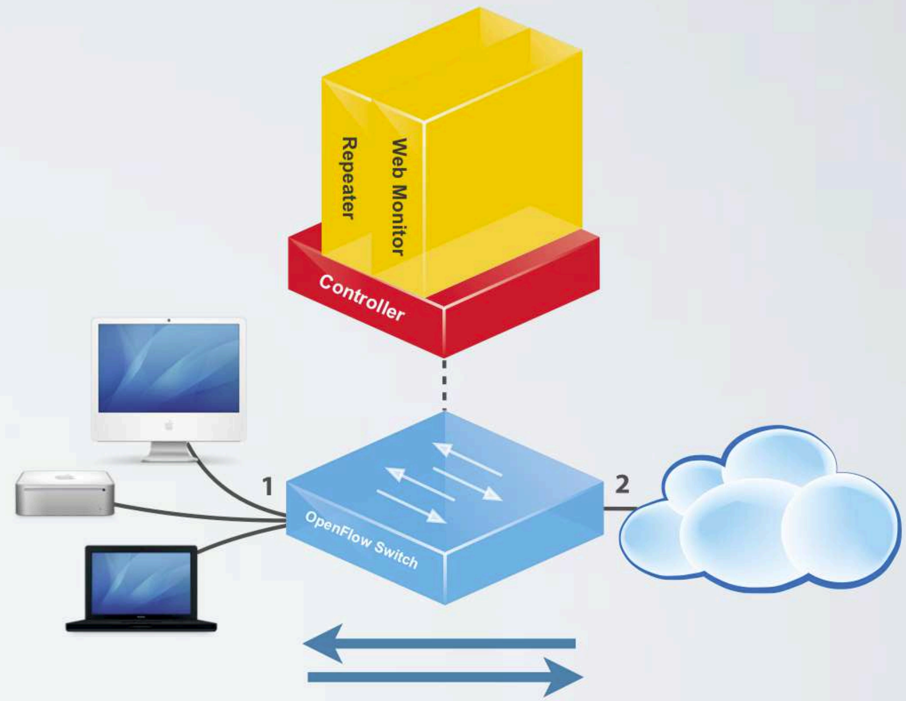
When a switch joins the network, install a monitoring rule

Example: Repeater + monitor

Repeater + Web Monitor

```
def switch_join(switch):
  p1 = {inport:1}
  a1 = [forward(2)]
  install(switch, pat1, DEFAULT, None, a1)
  p2 = {inport:2}
  pat2web = {in_port:2, tp_src:80}
  a2 = [forward(1)]
  install(switch, pat2web, HIGH, None, a2)
  install(switch, pat2, DEFAULT, None, a2)
  query_stats(switch, pat2web)

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



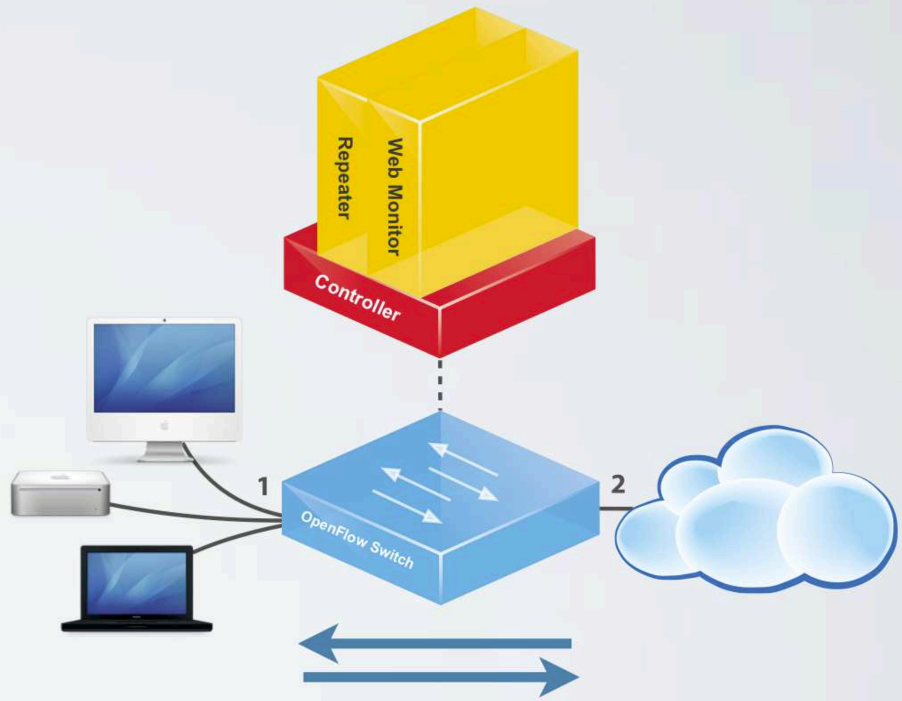
Must think about *both tasks* at the same time

Example: Repeater + monitor

Repeater + Web Monitor

```
def switch_join(switch):
  p1 = {inport:1}
  a1 = [forward(2)]
  install(switch, pat1, DEFAULT, None, a1)
  p2 = {inport:2}
  pat2web = {in_port:2, tp_src:80}
  a2 = [forward(1)]
  install(switch, pat2web, HIGH, None, a2)
  install(switch, pat2, DEFAULT, None, a2)
  query_stats(switch, pat2web)

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



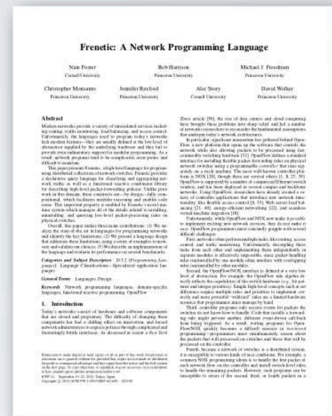
Must think about *both tasks* at the same time

Frenetic

[ICFP ’11, POPL ’12]

Network Programming Language

- Streaming functional language—no events!
- Declarative semantics
- Separates reads (queries) from writes (policy)



Frenetic: A Network Programming Language

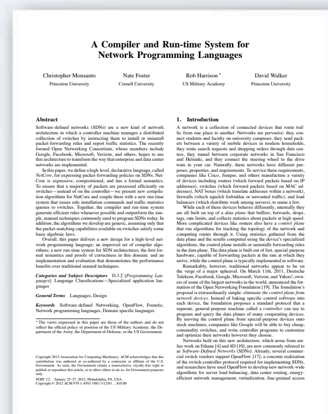
Nate Foster, Cornell University
Rob Harrison, Princeton University
Michael J. Freedman, Princeton University
Christopher Monsanto, Princeton University
Jennifer Rexford, Princeton University
Alec Story, Cornell University
David Walker, Princeton University

Abstract

1. Introduction

Compiler and Run-time System

- Translates high-level programs to switches
- Automatically manages low-level resources



A Compiler and Run-time System for Network Programming Languages

Christopher Monsanto, Princeton University
Nate Foster, Cornell University
Rob Harrison *, US Military Academy
David Walker, Princeton University

Abstract

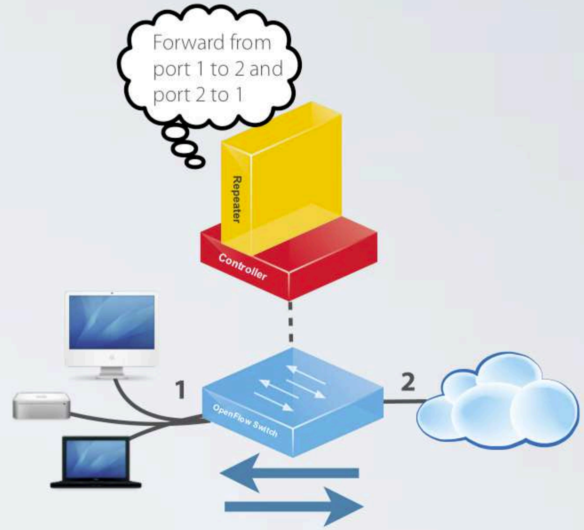
1. Introduction

Frenetic By Example

Repeater

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

def repeater():
 return \
  (SwitchJoin() >>
   Lift(lambda s:{s:policy}))
```



Policies have a declarative semantics
that is independent of other program pieces

Frenetic By Example

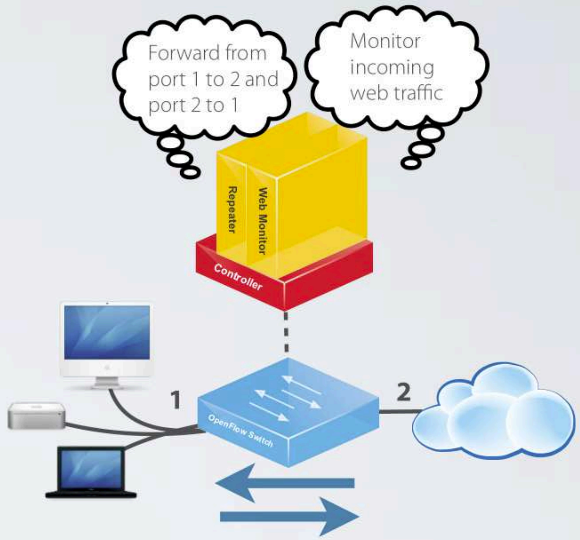
Repeater

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

def repeater():
 return \
  (SwitchJoin() >>
   Lift(lambda s:{s:policy}))
```

Web Monitor

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



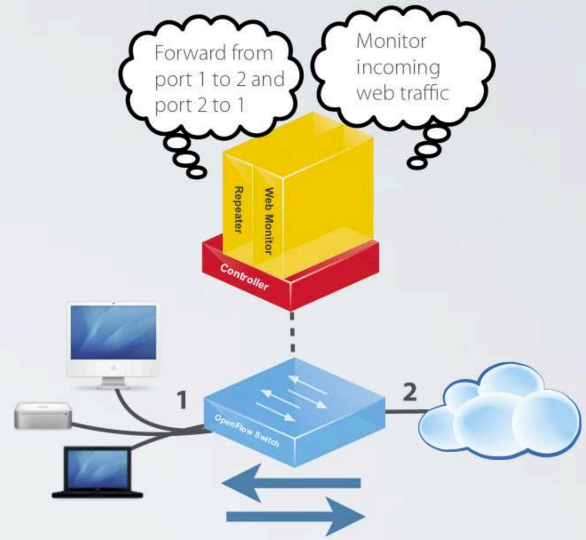
Queries have a declarative semantics
that is independent of other program pieces

Frenetic By Example

Repeater

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

def repeater():
 return \
  (SwitchJoin() >>
   Lift(lambda s:{s:policy}))
```



Web Monitor

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

Repeater + Web Monitor

```
def main():
 web_query() >> Print()
 repeater() >> Register()
```

Program pieces compose

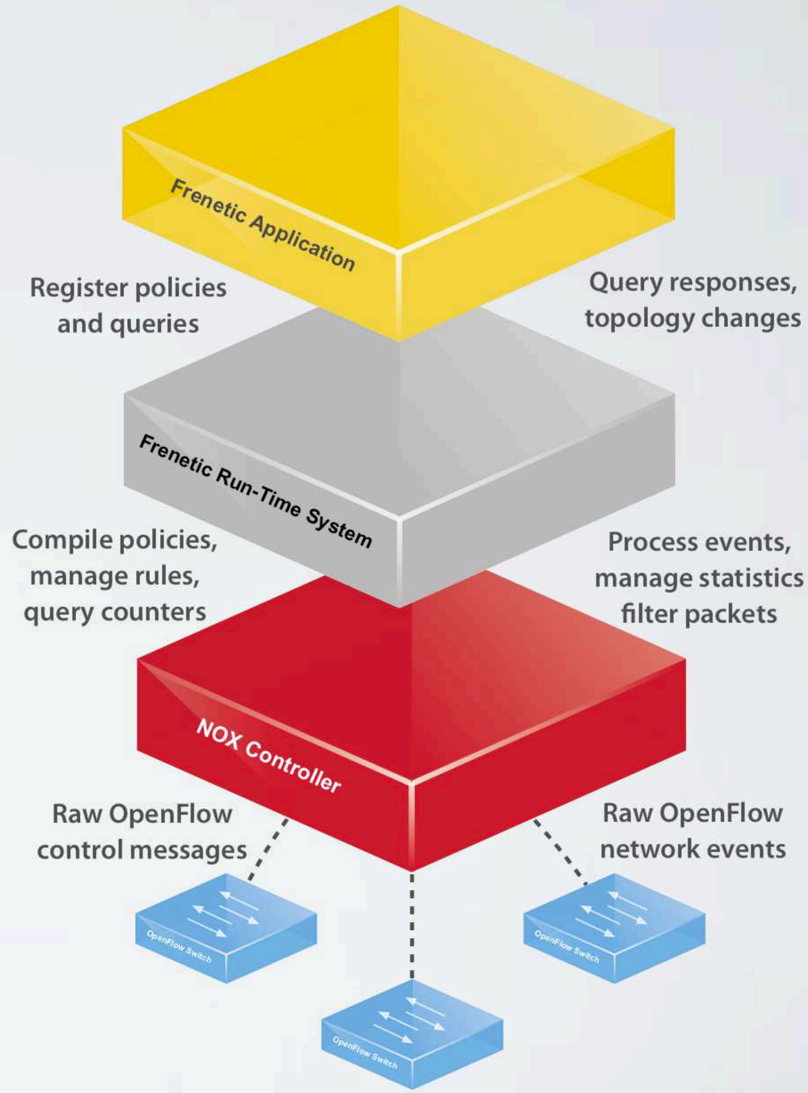
Frenetic System Overview

High-level Language

- Declarative policies
- Integrated queries
- Effective support for composition

Compiler and Run-time System

- Translates policies and queries
- Manages forwarding rules
- Tracks statistics
- Handles asynchronous events



Vision

(and Challenges)

Tony Hoare's "Mistake"

I call it my billion-dollar mistake.

It was the invention of the null reference in 1965.

My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.

This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Programming Language Abstractions

Many high-profile mistakes!

- Polymorphism + references
- Bounded quantification
- Pretty much every C compiler :-)


Finding and Understanding Bugs in C Compilers

Programming Language Abstractions

Many high-profile mistakes!

- Polymorphism + references
- Bounded quantification
- Pretty much every C compiler :-)

So language researchers have developed a body of techniques for modeling and reasoning precisely about language abstractions

- Operational semantics
- Denotational semantics
- Axiomatic semantics
- Bisimulations

$$\langle \sigma, c \rangle \models \phi$$

$$[\![e]\!] \qquad P \sim P'$$

$$e \rightarrow e' \qquad e \Downarrow v$$

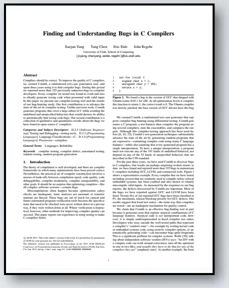
$$\Gamma \vdash e : \tau$$

Programming Language Abstractions

Many high-profile mistakes!

- Polymorphism + references
- Bounded quantification
- Pretty much every C compiler :-)

POPL mark


Finding and Understanding Bugs in C Compilers

So language researchers have developed a body of techniques for modeling and reasoning precisely about language abstractions

- Operational semantics
- Denotational semantics
- Axiomatic semantics
- Bisimulations

$$\langle \sigma, c \rangle \models \phi$$

$$[\![e]\!] \qquad P \sim P'$$

$$e \rightarrow e' \qquad e \Downarrow v$$

$$\Gamma \vdash e : \tau$$

Proving "obvious" theorems often reveals bugs

Writing down a semantics is an efficient way to communicate ideas

A lot of effort has gone into making these techniques scalable!

Opportunities and Challenges

Opportunities and Challenges

SDNs offer a unique opportunity to

- Define new abstractions for networks
- Develop their mathematical properties
- Design efficient implementations
- Deploy verification tools that provide assurance

and avoid (the analogues of) Hoare's mistake!

Opportunities and Challenges

SDNs offer a unique opportunity to

- Define new abstractions for networks
- Develop their mathematical properties
- Design efficient implementations
- Deploy verification tools that provide assurance

and avoid (the analogues of) Hoare's mistake!

Challenge #2

- Want to program virtual networks
- Slices? Logical forwarding plane?
- Want to validate implementations, prove isolation properties

Opportunities and Challenges

SDNs offer a unique opportunity to

- Define new abstractions for networks
- Develop their mathematical properties
- Design efficient implementations
- Deploy verification tools that provide assurance

and avoid (the analogues of) Hoare's mistake!

Challenge #1

- Combining conflicting policies
- Constraint-based policies?
- FML [Hinrichs+ '09] and Cologne [Liu+ '12]

Challenge #2

- Want to program virtual networks
- Slices? Logical forwarding plane?
- Want to validate implementations, prove isolation properties

Thank You!

Collaborators

Shrutarshi Basu (Cornell)
Mike Freedman (Princeton)
Stephen Gutz (Cornell)
Rob Harrison (West Point)
Chris Monsanto (Princeton)
Joshua Reich (Princeton)
Mark Reitblatt (Cornell)
Emin Gün Sirer (Cornell)
Cole Schlesinger (Princeton)
Alec Story (Cornell)
Jen Rexford (Princeton)
David Walker (Princeton)

Funding

http://frenetic-lang.org