

Machine-Verified Network Controllers

Nate Foster
Cornell University



Proof Assistants



Coq

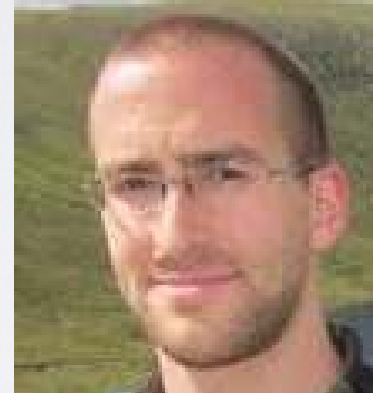
Proof Assistants



Coq



Arjun Guha
Postdoc → UMass



Mark Reitblatt
PhD student

Networks in Practice

Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region

April 29, 2011

Now that we have fully restored functionality to all affected services, we would like to share more details with our customers about the events that occurred with the Amazon Elastic Compute Cloud ("EC2") last week, our efforts to restore the services, and what we are doing to prevent this sort of issue from happening again. We are very aware that many of our customers were significantly impacted by this event, and as with any significant service issue, our intention is to share the details of what happened and how we will improve the service for our customers.

The issues affecting EC2 customers last week primarily involved a subset of the Amazon Elastic Block Store ("EBS") volumes in a single Availability Zone within the US East Region that became unable to service read and write operations. In this document, we will refer to these as "stuck" volumes. This caused instances trying to use these affected volumes to also get "stuck" when they attempted to read or write to them. In order to restore these volumes and stabilize the EBS cluster in that Availability Zone, we disabled all control APIs (e.g. Create Volume, Attach Volume, Detach Volume, and Create Snapshot) for EBS in the affected Availability Zone for much of the duration of the event. For two periods during the first day of the issue, the degraded EBS cluster affected the EBS APIs and caused high error rates and latencies for EBS calls to these APIs across the entire US East Region. As with any complicated operational issue, this one was caused by several root causes interacting with one another and therefore gives us many opportunities to protect the service against any similar event reoccurring.

Overview of EBS System

It is helpful to understand the EBS architecture so that we can better explain the event. EBS is a distributed, replicated block data store that is optimized for consistency and low latency read and write access from EC2 instances. There are two main components of the EBS service: (i) a set of EBS clusters (each of which runs entirely inside of an Availability Zone) that store user data and serve requests to EC2 instances; and (ii) a set of control plane services that are used to coordinate user requests and propagate them to the EBS clusters running in each of the Availability Zones in the Region.

An EBS cluster is comprised of a set of EBS nodes. These nodes store replicas of EBS volume data and serve read and write requests to EC2 instances. EBS volume data is replicated to multiple EBS nodes for durability and availability. Each EBS node employs a peer-to-peer based, fast failover strategy that aggressively provisions new replicas if one of the copies ever gets out of sync or becomes unavailable. The nodes in an EBS cluster are connected to each other via two networks. The primary network is a high bandwidth network used in normal operation for all necessary communication with other EBS nodes, with EC2 instances, and with the EBS control plane services. The secondary network, the replication network, is a lower capacity network used as a back-up network to allow EBS nodes to reliably communicate with other nodes in the EBS cluster and provide overflow capacity for data replication. This network is not designed to handle all traffic from the primary network but rather provide highly-reliable connectivity between EBS nodes inside of an EBS cluster.

When a node loses connectivity to a node to which it is replicating data to, it assumes the other node failed. To preserve durability, it must find a new node to which it can replicate its data (this is called re-mirroring). As part of the re-mirroring process, the EBS node searches its EBS cluster for another node with enough available server space, establishes connectivity with the server, and propagates the volume data. In a normally functioning cluster, finding a location for the new replica occurs in milliseconds. While data is being re-mirrored, all nodes that have copies of the data hold onto the data until they can confirm that another node has taken ownership of their portion. This provides an additional level of protection against customer data loss. Also, when data on a customer's volume is being re-mirrored, access to that data is blocked until the system has identified a new primary (or writable) replica. This is required for

“The trigger for this event was a network configuration change”
—Amazon

Networks in Practice

The screenshot shows a web browser window with two tabs. The top tab is titled "Summary of the Amazon EC2" and the address bar shows "aws.amazon.com/message/65648/". The bottom tab is titled "CEO Addresses Sept. 10 Serv" and the address bar shows "support.godaddy.com/godaddy/ceo-addresses-sept-...". The GoDaddy support page features a navigation menu with "Domain", "Websites", and "Hosting" categories. A search bar is present with the text "Enter your search terms here" and a "GO" button. The main content area is titled "CEO Addresses Sept. 10 Service Outage" and includes a date of "9-11-2012" and a "Go Daddy" logo. The text of the message reads: "We owe you a big apology for the intermittent service outages we experienced on September 10th that may have impacted your website and your interaction with GoDaddy.com. The service outage was due to a series of internal network events that corrupted router data tables. Once the issues were identified, we took corrective actions to restore services for our customers and GoDaddy.com. We have implemented measures to prevent this from occurring again. At no time was any sensitive customer information, such as credit card data, passwords or names and addresses, compromised. Throughout our history, we have provided 99.999% uptime in our DNS infrastructure. This is the level of performance our customers have come to expect from us and that we expect from ourselves. We pride ourselves on providing world-class service — through our products, our site experience and customer care. We have let our customers down and we know it. I cannot express how sorry I am to those of you who were inconvenienced. We will learn from this. I'd like to express my profound gratitude to all our customers. We are thankful for your straightforward feedback and the confidence you have shown in us. In appreciation, we will reach out to affected customers in the coming days with a good faith gesture that acknowledges the disruption. We are grateful for your continued loyalty and support. Sincerely, Scott Wagner, Go Daddy CEO." The page also includes a "220 People Found this Helpful" badge, a "Feedback" button, and a "24/7 Support" section with contact information and a "7 min expected wait time" indicator.

“The service outage was due to a series of internal network events that corrupted router data tables”

—GoDaddy

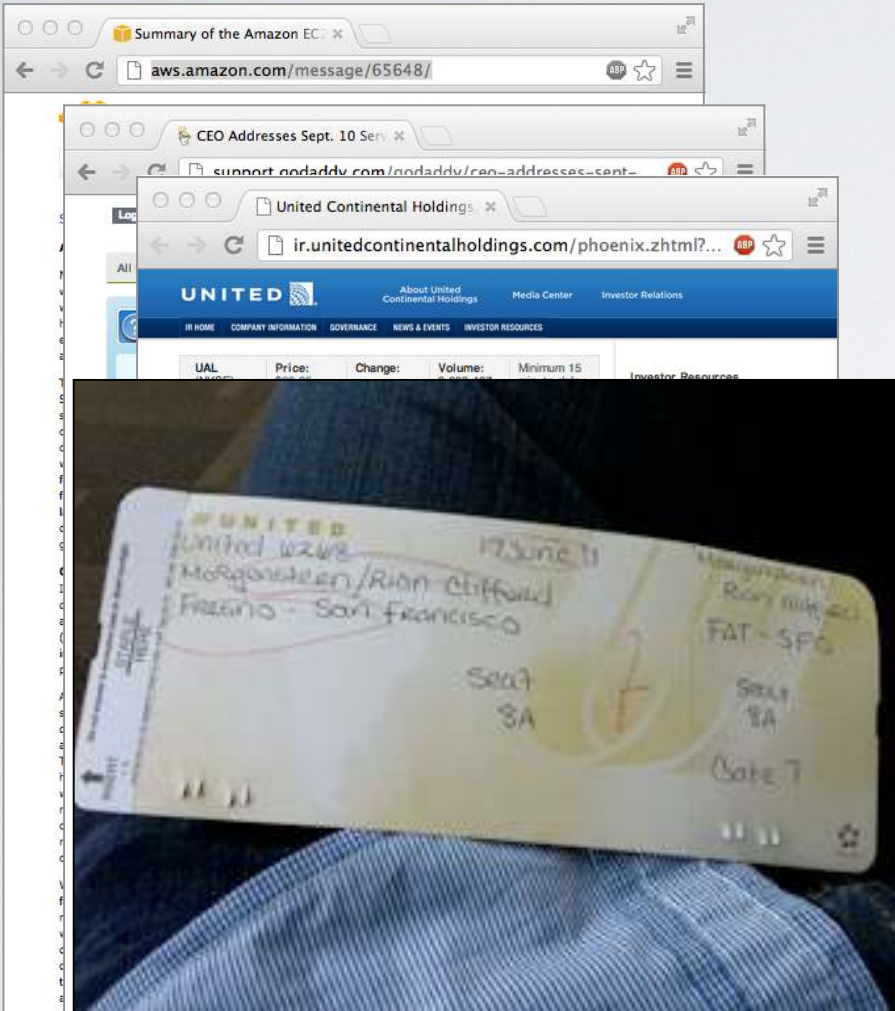
Networks in Practice

The screenshot shows a web browser window displaying the United Airlines website. The browser's address bar shows the URL `ir.unitedcontinentalholdings.com/phoenix.zhtml?...`. The website header includes the United logo and navigation links for 'About United Continental Holdings', 'Media Center', and 'Investor Relations'. A table displays stock information for UAL (NYSE) with a price of \$20.25 and a volume of 3,633,407. The main content area features a news article titled 'United Airlines Restoring Normal Flight Operations Following Friday Computer Outage'. The article text states: 'CHICAGO, June 18, 2011 /PRNewswire via COMTEX/ -- United Airlines, a subsidiary of United Continental Holdings, Inc. (NYSE: UAL), is in the process of resuming normal operations Saturday, June 18, following a temporary computer outage Friday. The airline experienced a network connectivity issue at about 7:15 p.m. CT Friday, which was resolved at midnight. United apologizes for the disruption caused to travelers at affected airports and is reaccommodating travelers where necessary. "While we will be experiencing some residual effect on our flight operations throughout the weekend, United is committed to restoring normal operations as soon as possible," said Alexandria Marnen, senior vice president System Operations Control. "We encourage customers to print their boarding pass prior to arrival at the airport and give themselves extra time. "We are reaching out through multiple channels to ask customers who were inconvenienced by this event to contact us." United has been providing regular updates for customers through Twitter and other channels. The computer problem interrupted the airline's flight departures, airport processing and reservations systems, including access to the united.com internet site. Waiver policy for United customers booked on June 17 and 18 United is allowing fee-waived exceptions for customers whose travel plans were impacted by the June 17 computer outage. Customers scheduled on United flights on June 17 and 18 may reschedule their itinerary with a one-time date or time change, and the change fees will be waived. For customers wishing to cancel their travel plans, a refund in the original form of payment may be requested. Complete details and eligible travel dates are available at united.com and continental.com. Customers should continue to manage their reservations on the respective company's website from which their ticket was purchased. Customers may also book a new reservation, change an existing reservation or check flight status by calling United Reservations at 800-UNITED-1 or Continental Reservations at 800-525-0280 or their travel agent. SOURCE United Continental Holdings, Inc.

At the bottom of the page, there are links for 'Print Page' and 'E-mail Page', and a footer note: 'Quotes delayed at least 15 minutes. Market data provided by Interactive'.

“The airline experienced a network connectivity issue...”
—United Airlines

Networks in Practice



“The airline experienced a network connectivity issue...”
—United Airlines

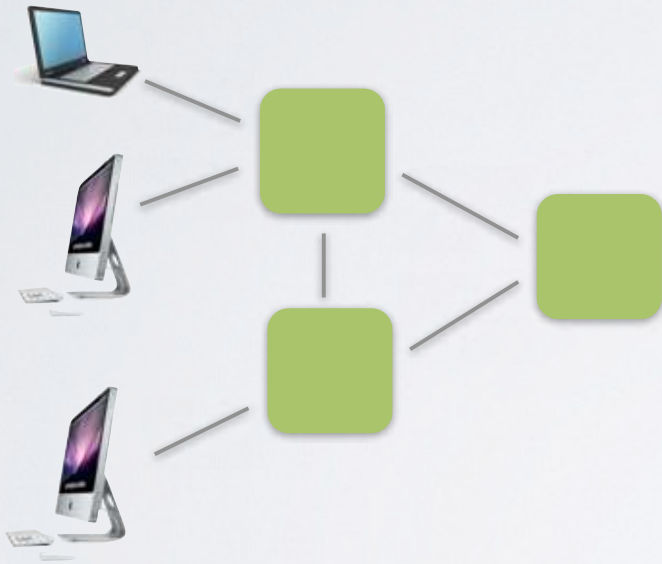
Networks in Practice

There are hosts...



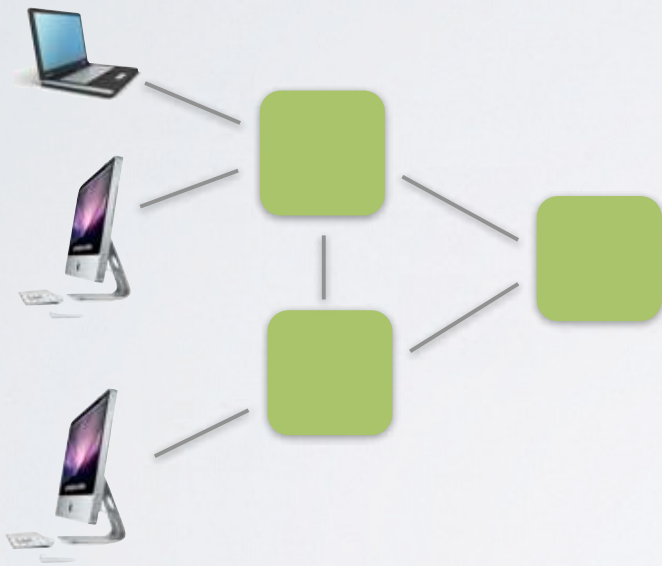
Networks in Practice

Connected by switches...



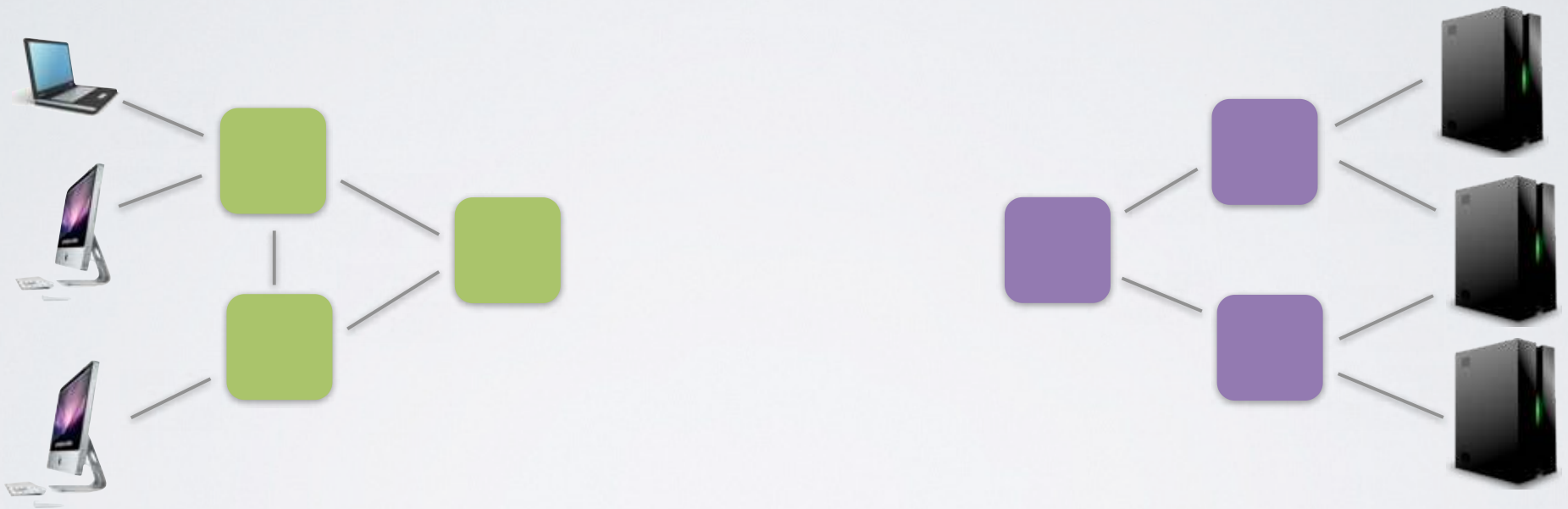
Networks in Practice

There are also servers...



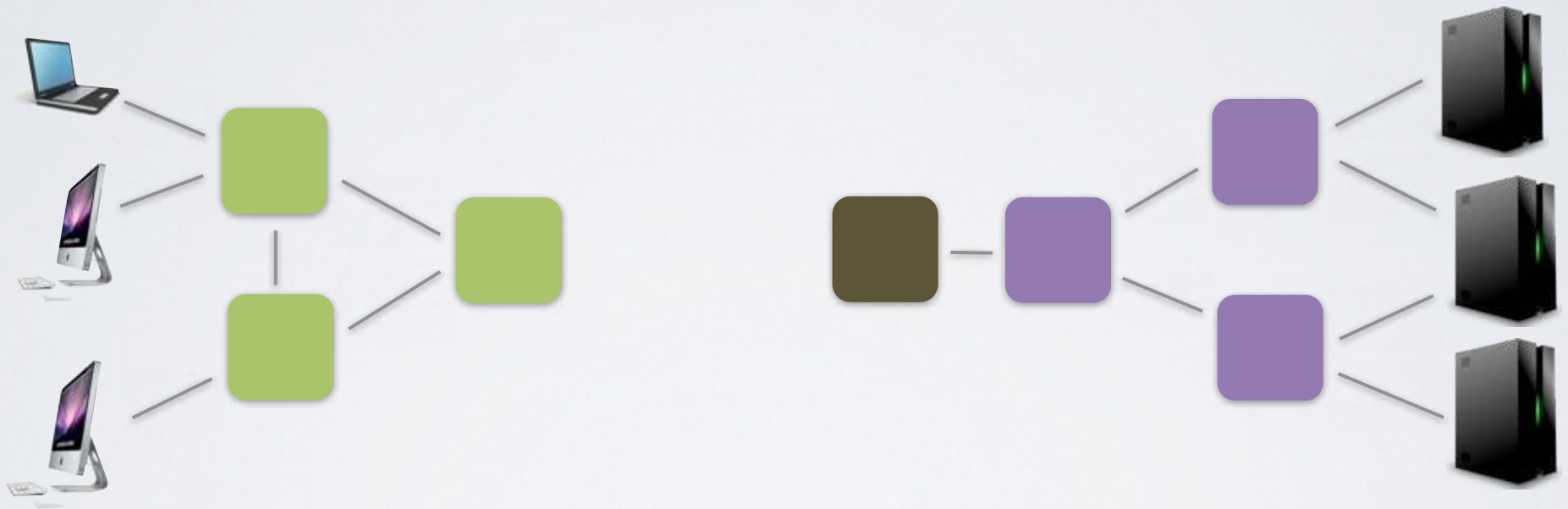
Networks in Practice

Connected by routers...



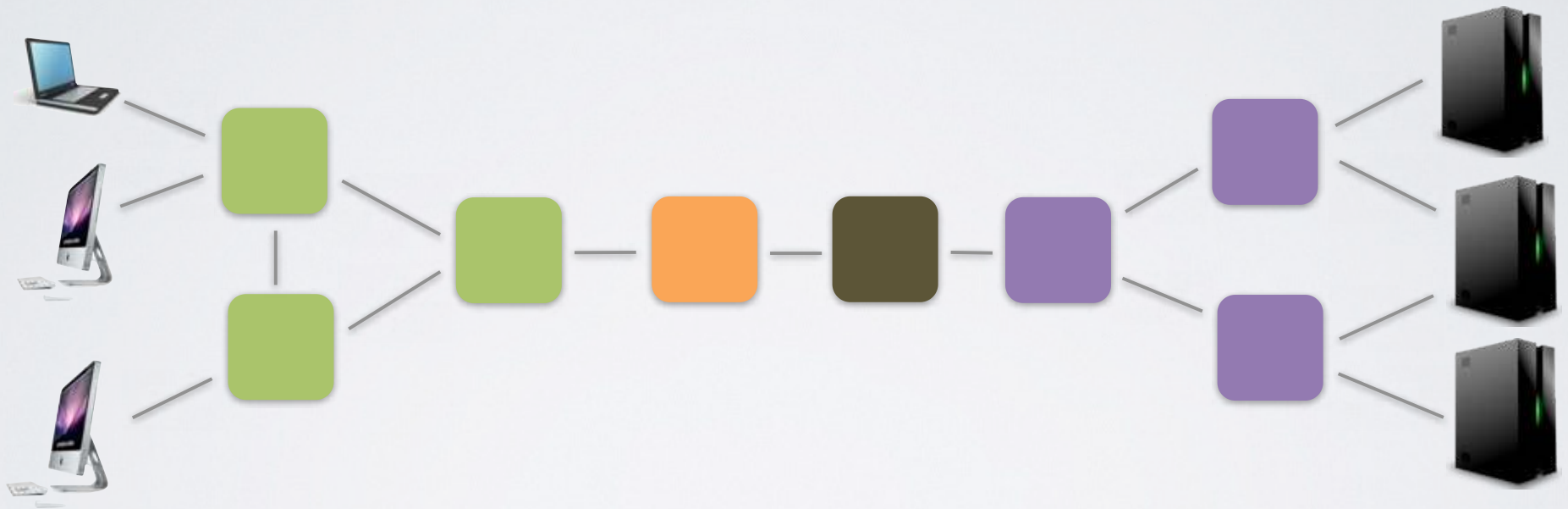
Networks in Practice

And a load balancer...



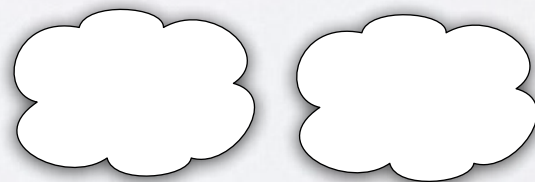
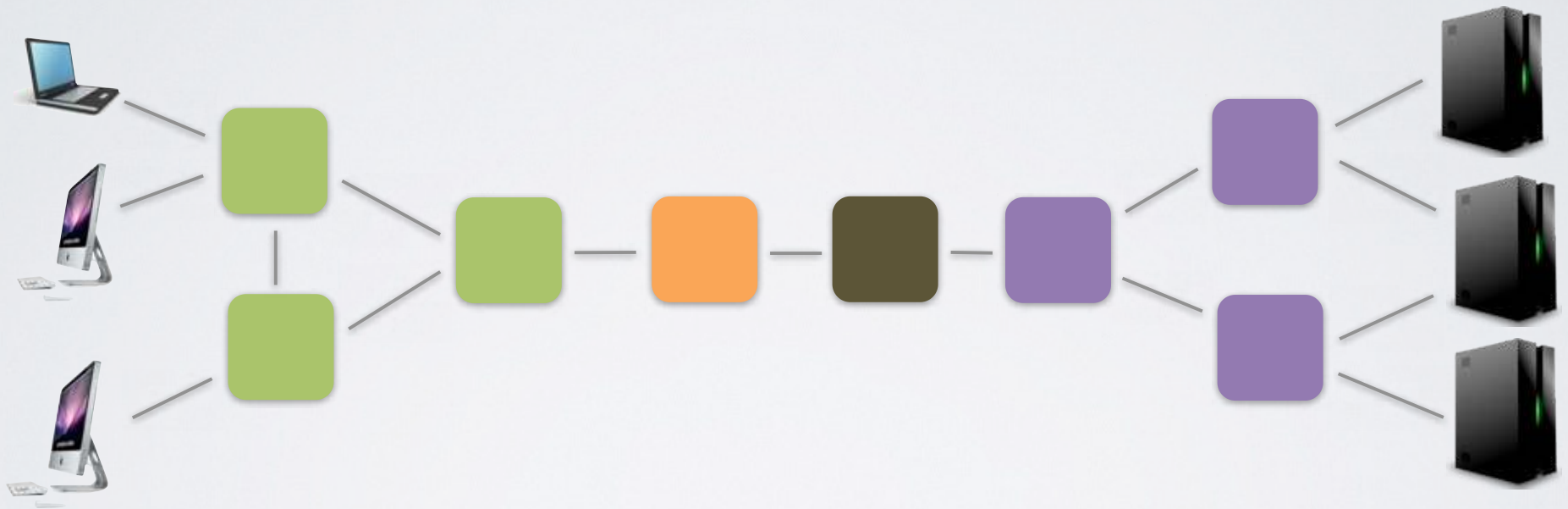
Networks in Practice

And a gateway router...



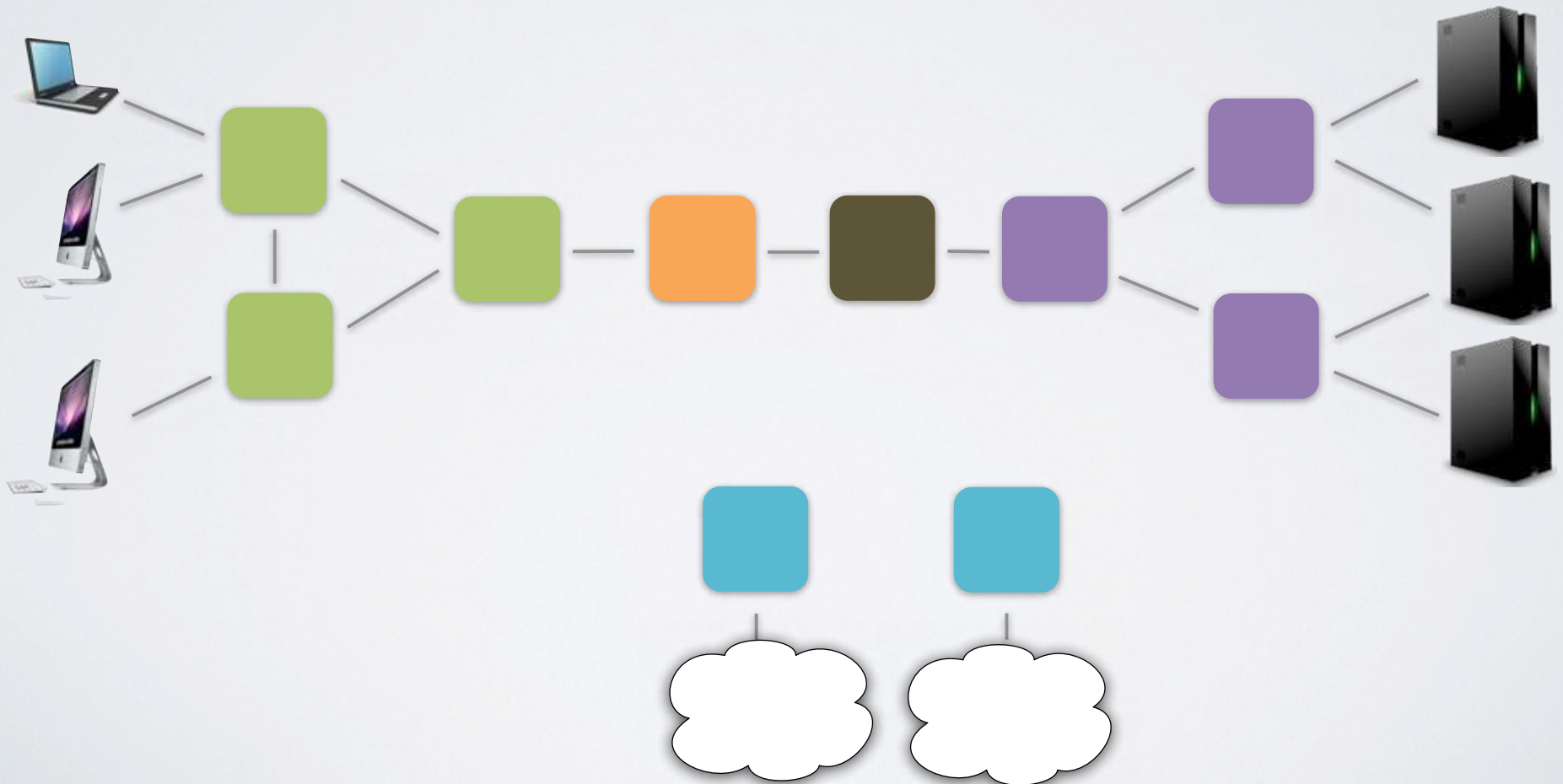
Networks in Practice

There are other ISPs...



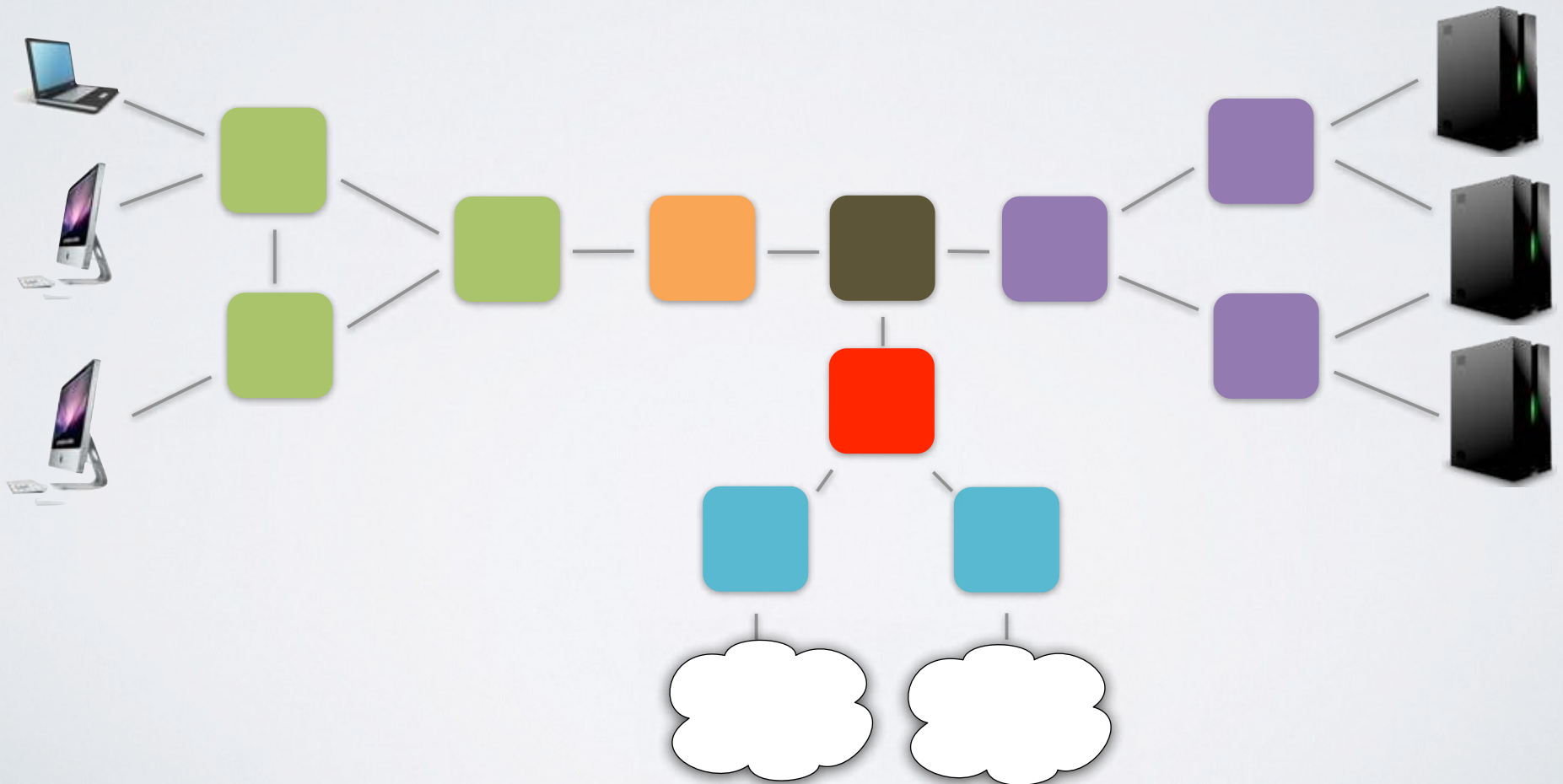
Networks in Practice

So we need to run BGP...



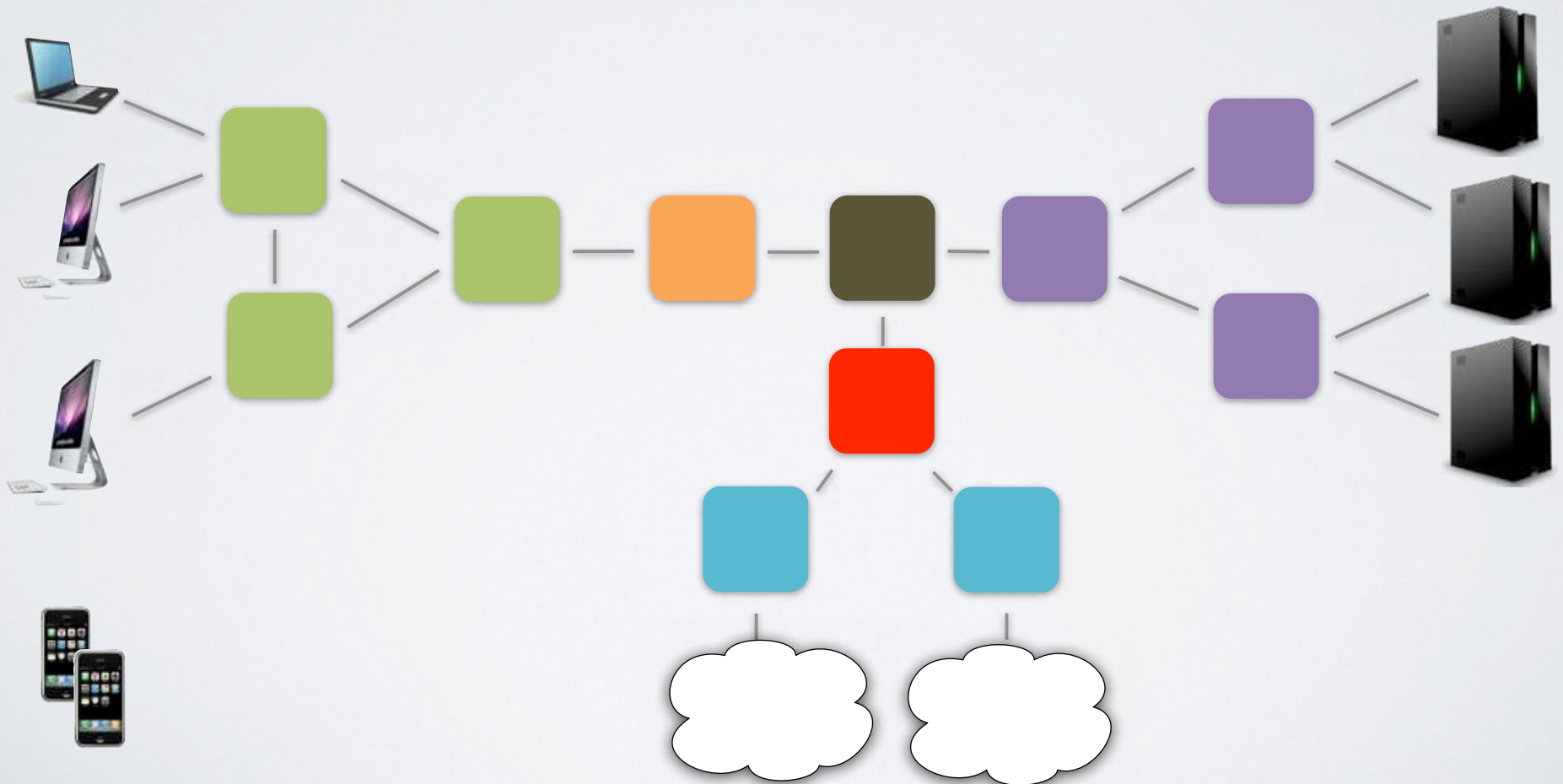
Networks in Practice

And we need a firewall to filter incoming traffic...



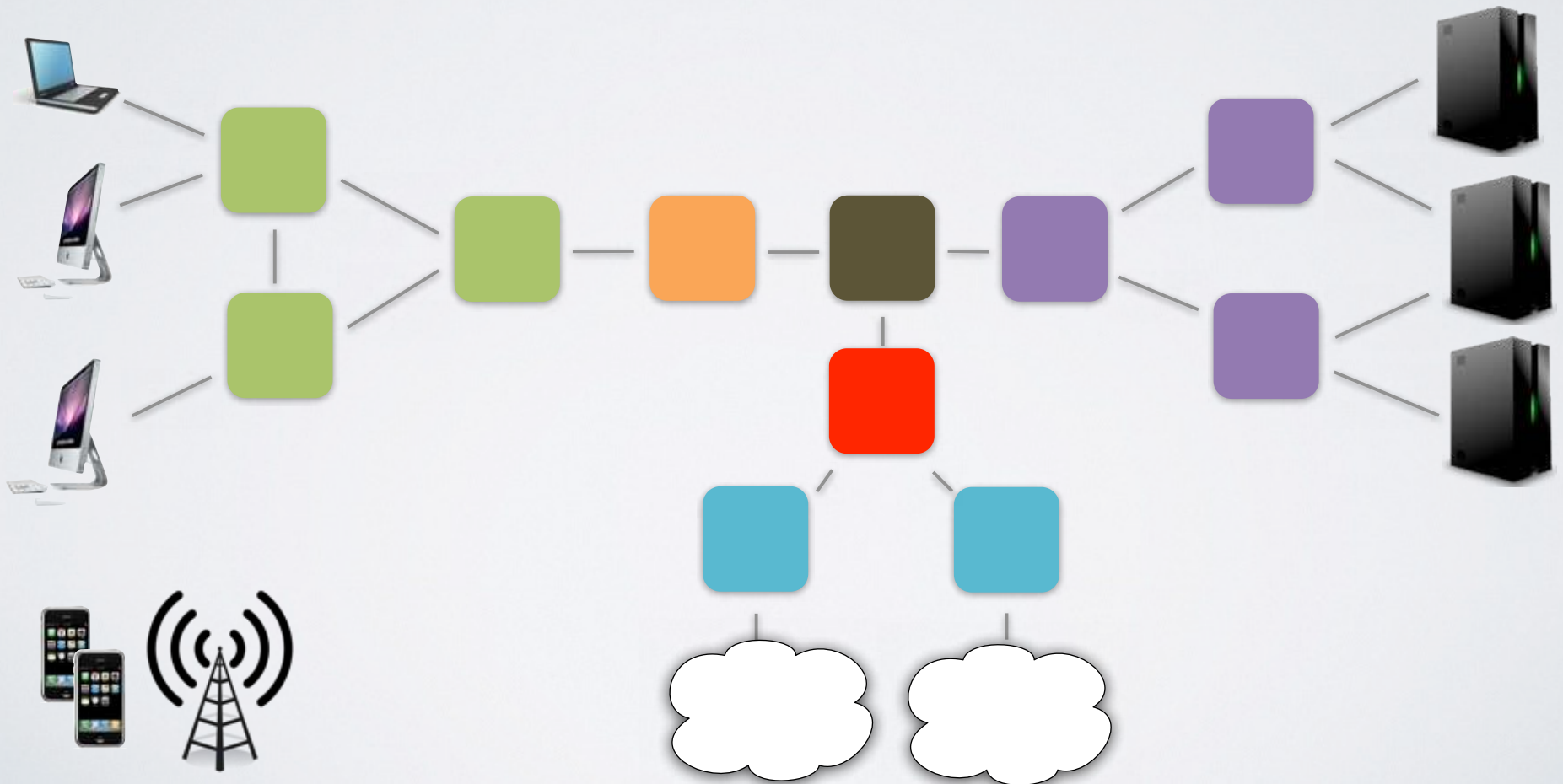
Networks in Practice

There are also wireless hosts...



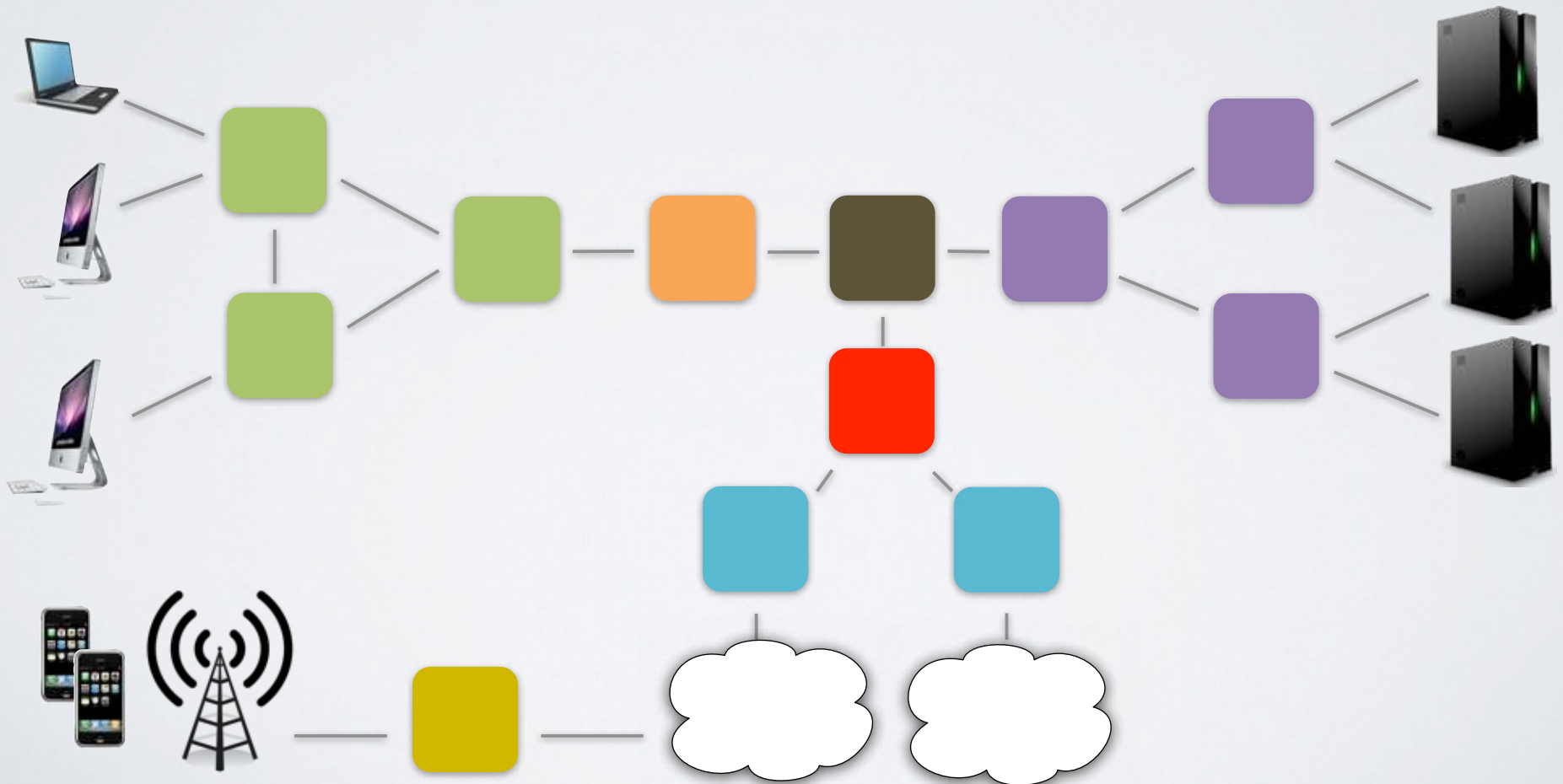
Networks in Practice

So we need wireless gateways...



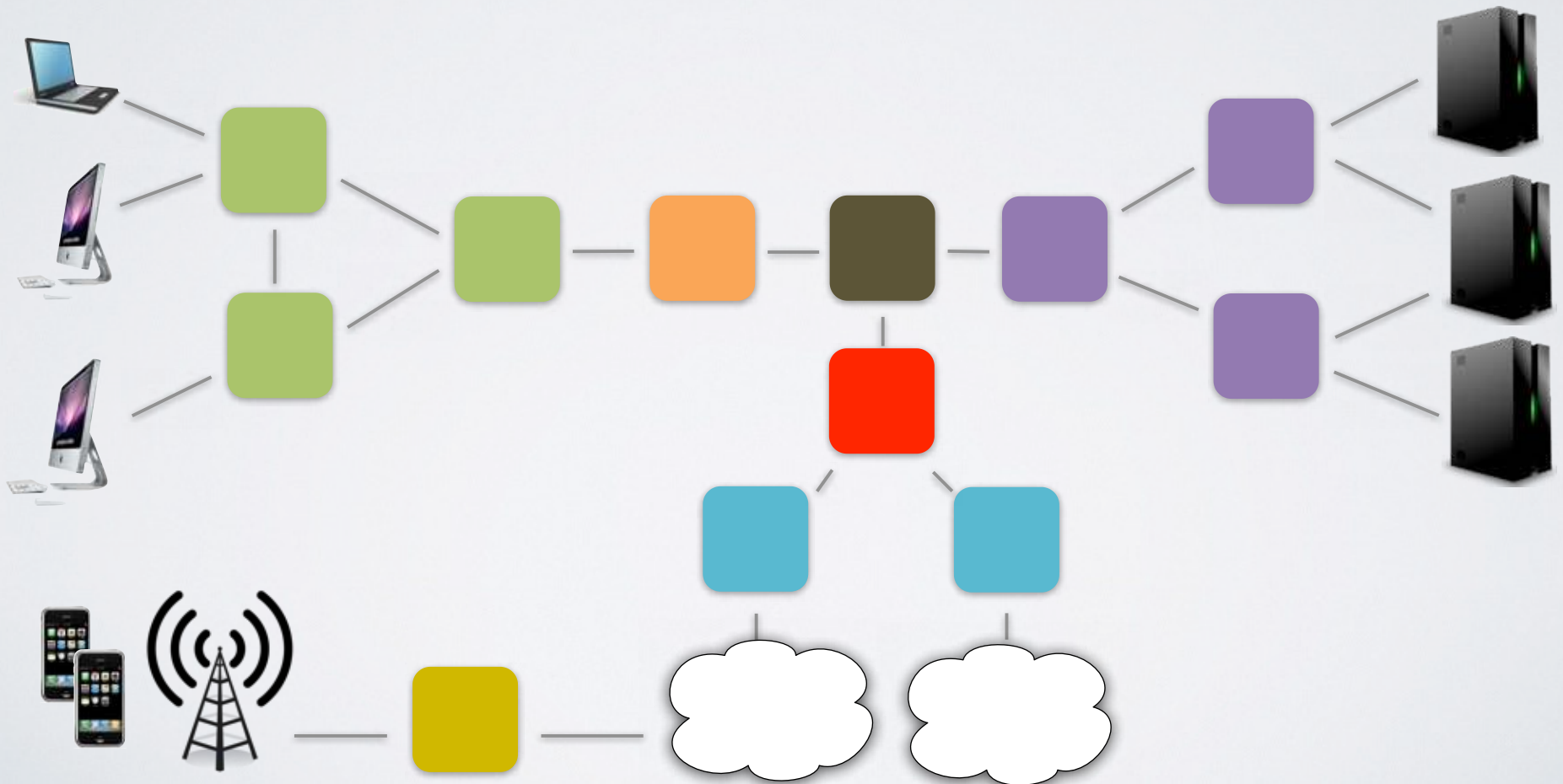
Networks in Practice

And yet more middleboxes for lawful intercept...



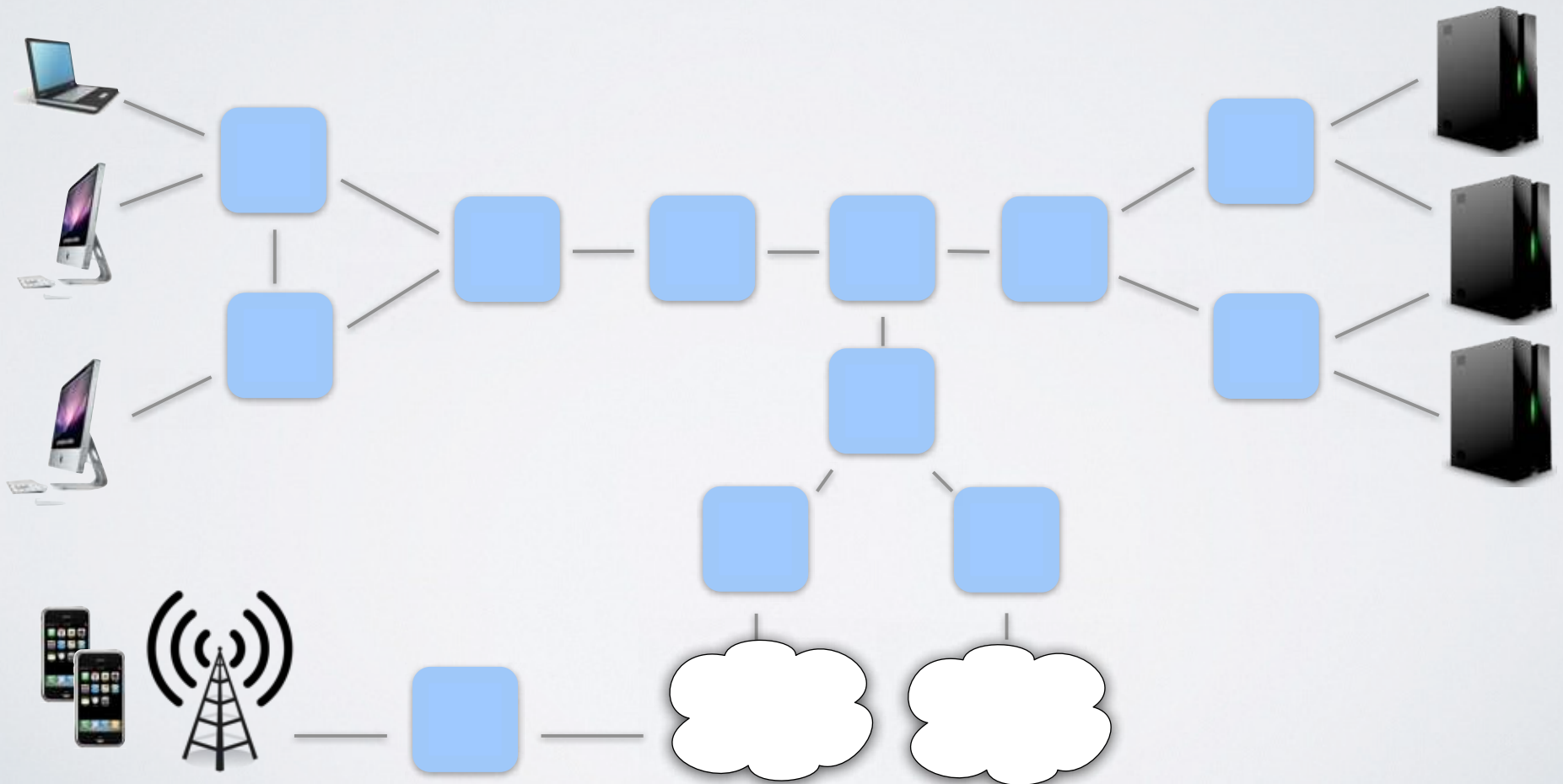
Networks in Practice

Each color represents a different set of control plane protocols and algorithms... this is



Software-Defined Networking

A clean-slate architecture that standardizes features and decouples forwarding from



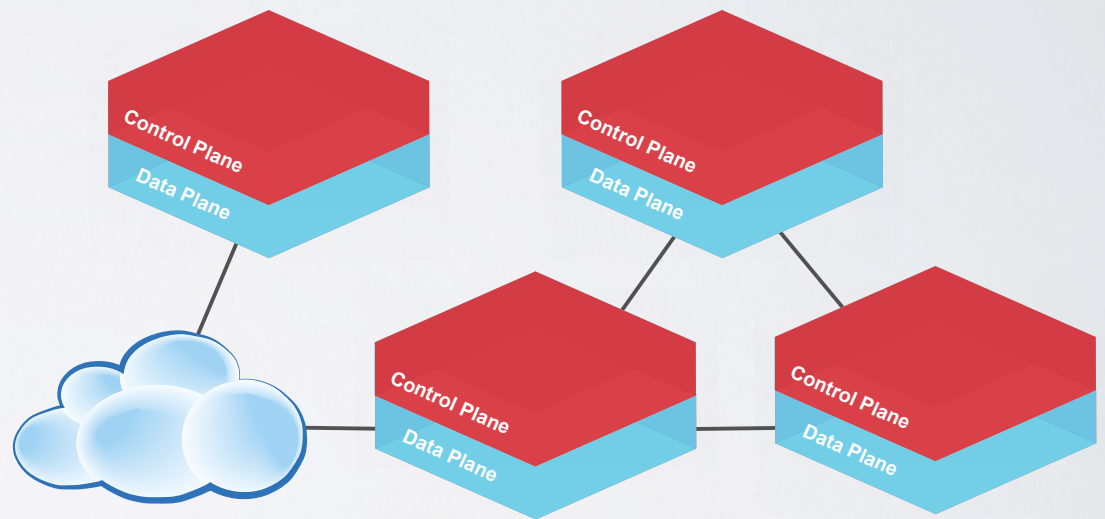
Software-Defined Networking

Essential ingredients

- Decouple control and data planes
- Logically-centralized control

Enables

- Novel functionality
- Formal reasoning



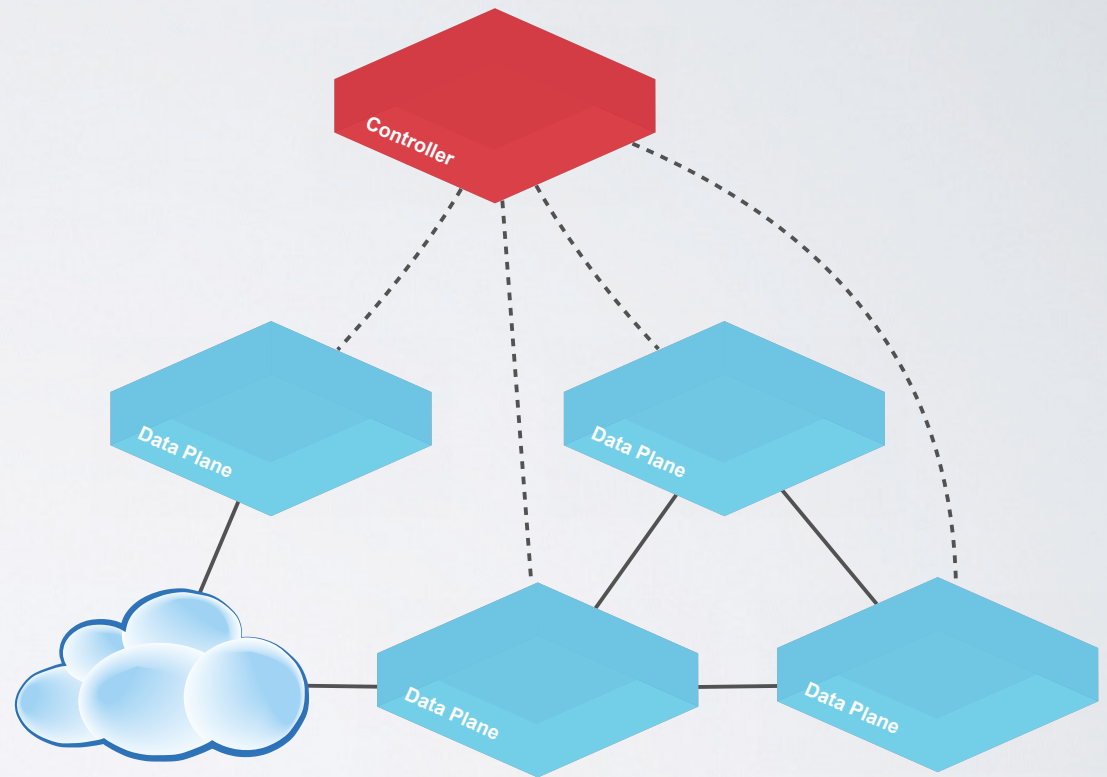
Software-Defined Networking

Essential ingredients

- Decouple control and data planes
- Logically-centralized control

Enables

- Novel functionality
- Formal reasoning



Existing Tools



There is a cottage industry in SDN configuration-checking tools...

Existing Tools

There is a cottage industry in SDN configuration-checking tools...

- FlowChecker [SafeConfig '10]

FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures

Ehab Al-Shaar and Saeed Al-Hajj
Department of Software and Information Systems
University of North Carolina at Charlotte
{ealshaar, sahai}@uncg.edu

ABSTRACT

It is difficult to build a real network to test novel experiments. OpenFlow enables us to use computers to run their own experiments by providing a virtual slice and configuration on real networks. Building a network can allow the user network by specifying a definition file for each one. Users are given the responsibility to maintain and use their own slice by writing rules in a Flow Table. Misconfiguration problems can arise when a user writes conflicting rules for single FlowTables or even within a path of multiple OpenFlow switches that need multiple FlowTables to be maintained at the same time.

In this work, we describe a tool, FlowChecker, to identify any inter-switch misconfiguration within a single FlowTable. We also describe the intra-switch or inter-federated inconsistency in a path of OpenFlow switches across the same or different OpenFlow infrastructures. FlowChecker models FlowTable configuration using Binary Decision Diagrams and then uses the model checker technique to model the interconnected network of OpenFlow switches.

Categories and Subject Descriptors: Network Operations—Network management

General Terms

Security, Verification

Keywords

OpenFlow, configuration verification, access control, automated analysis, binary decision diagrams

1. INTRODUCTION

OpenFlow is an innovative architecture that provides an open programmable platform for network access control [1]. By separating the data and control planes, users can use the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made for distribution for profit or commercial advertising and promotion, to be used only for personal or classroom use, to be copied, to be republished, to post on an internal or external network, to be included in any repository, to be used for teaching or research, or to be distributed to a list, require prior specific permission and/or fee.

SafeConfig '10, October 4, 2010, Chicago, Illinois, USA.
Copyright 2010 ACM 978-1-4503-0100-1/10/0000...\$5.00.

OpenFlow centralized controllers to install filters (match, count and actions) in the OpenFlow switches and control the global data processing in the network. Currently, OpenFlow control supports the following actions: forward, drop, encapsulate, encrypt, limit, and classify/inspect for QoS. The platform is also extensible to support more actions. The controller running new protocols or algorithms might insert, modify, or remove filters in the switches in order to enforce network-wide policies or properties (e.g. packets should access the internet only through a proxy) [7]. Thus, it is assumed that the integrated behavior of the installed filters will globally implement these policies. However, the following conflicts become apparent: (1) the semantic gap between the controller platform (e.g., SDN, [1]) and the filter tables in the data processing units; (2) the distribution of access control that supports aggregate flows (wild-cards) and many different actions; (3) the ability of sharing one controller by different users; and (4) the ability of using multiple controllers in the same domain. These conflicts together increase the potential of intra-federated (single domain) OpenFlow configuration conflicts. In addition, as two or more OpenFlow infrastructures communicate with each other, potential inter-federated conflicts may appear due to inconsistency in the controller or switch configuration. This may result in accumulation of non-optimal policy enforcement.

Due to these reasons, a correct enforcement of the controller policies might be guaranteed without the support of formal automated configuration verification tools. This work attempts to address these problems by (1) modeling OpenFlow configuration using Binary Decision Diagrams (BDDs) considering the priority-based matching semantics, various actions, the existence of multiple controllers and multiple users; (2) modeling the global behavior of the OpenFlow network based on FlowTables over single or multiple federated infrastructures in a single state machine; and (3) providing a generic property-based verification interface using BDD-based symbolic model checking and temporal logic. The presented system, called FlowChecker, can be used by administrators/users for (1) verifying the consistency of different switches and controllers across different OpenFlow federated infrastructures; (2) validating the correctness of the configuration syntheses generated by a new implemented protocols; and (3) debugging reachability and security problems. FlowChecker can also be used to conduct "what-if" analysis to study the impact of the new protocols or algorithms on the network by simply changing the state in the FlowTables and then analyzing the effects.

The development of FlowChecker leverages our previous

Existing Tools

There is a cottage industry in SDN configuration-checking tools...

- FlowChecker [SafeConfig '10]
- AntEater [SIGCOMM '11]

FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures

Debugging the Data Plane with Ant eater

Haoxui Mai, Ahmed Khurshid, Rachit Agarwal
Matthew Caesar, P. Brighten Godfrey, Samuel T. King
University of Illinois at Urbana-Champaign
{ma4, khursh1, agarwa16, caesar, pbg, kingst}@illinois.edu

ABSTRACT
It is difficult to verify OpenFlow configurations on real networks. Existing tools can only verify the syntax of configurations, but not the semantics. In this work, we describe a new tool, Ant eater, that can verify the semantics of OpenFlow configurations on real networks. Ant eater can verify the semantics of OpenFlow configurations on real networks by using a set of verification rules that are derived from the OpenFlow protocol. Ant eater can verify the semantics of OpenFlow configurations on real networks by using a set of verification rules that are derived from the OpenFlow protocol. Ant eater can verify the semantics of OpenFlow configurations on real networks by using a set of verification rules that are derived from the OpenFlow protocol.

Categories
C.2.3 (Computer-Communication Networks) Network Operations

General Terms
Security, Verification

Keywords
OpenFlow, network analysis, network verification

1. INTRODUCTION
OpenFlow is an open standard for network programmability. It allows network administrators to program the data plane of a network. However, OpenFlow configurations are often complex and error-prone. Existing tools can only verify the syntax of configurations, but not the semantics. In this work, we describe a new tool, Ant eater, that can verify the semantics of OpenFlow configurations on real networks. Ant eater can verify the semantics of OpenFlow configurations on real networks by using a set of verification rules that are derived from the OpenFlow protocol. Ant eater can verify the semantics of OpenFlow configurations on real networks by using a set of verification rules that are derived from the OpenFlow protocol.

Categories and Subject Descriptors
C.2.3 (Computer-Communication Networks) Network Operations; D.2.2 (Software Engineering) Testing and Debugging

General Terms
Algorithms, Reliability

Keywords
Data Plane Analysis, Network Troubleshooting, Business Scalability

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted by ACM, provided that the copies are not made or distributed for profit or commercial advantage and that copies bear the notice and full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and may be subject to payment of a fee. Copyright 2011 ACM 978-1-4558-0911-0/11/000000...\$10.00

290

Existing Tools

There is a cottage industry in SDN configuration-checking tools...

- FlowChecker [SafeConfig '10]
- AntEater [SIGCOMM '11]
- NICE [NSDI '12]

FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures

Debugging the Data Plane with Ant eater

A NICE Way to Test OpenFlow Applications

Marco Canina¹, Daniele Venzano², Peter Pesešini¹, Dejan Kostić¹, and Jennifer Rexford¹

¹EPFL ²Princeton University

Abstract

The emergence of OpenFlow-capable switches enables exciting new network functionality, at the risk of programming errors that make communication less reliable. The centralized programming model, where a single controller program manages the network, seems to reduce the likelihood of bugs. However, the system is inherently distributed and asynchronous, with events happening at different switches and end hosts, and inevitable delays affecting communication with the controller. In this paper, we present efficient, systematic techniques for testing unmodified controller programs. Our NICE tool applies model checking to explore the state space of the entire system—the controller, the switches, and the hosts. Scalability is the main challenge, given the diversity of data packets, the large system state, and the many possible event orderings. To address this, we propose a novel way to augment model checking with symbolic execution of event handlers via identity representative packets that exercise code paths on the controller. We also present a simplified OpenFlow switch model to reduce the state space, and effective strategies for generating event interleavings likely to uncover bugs. Our prototype tests OpenFlow applications on the popular NOX platform. In testing three real applications—a MAC-learning switch, in-network server load balancing, and energy-efficient traffic engineering—we uncover eleven bugs.

1 Introduction

While lowering the barrier for introducing new functionality into the network, Software Defined Networking (SDN) also raises the risk of software faults (or bugs). Even today's networking software—written and extensively tested by equipment vendors, and constrained (at least somewhat) by the protocol standardization process—can have bugs that trigger Internet-wide outages [1, 2]. In contrast, programmable networks will offer a much wider range of functionality, through software created by a diverse collection of network operators and third-party developers. The ultimate success of SDN, and enabling technologies like OpenFlow [3], depends on having effective ways to test applications in pursuit of achieving high reliability. In this paper, we present NICE, a tool that efficiently uncovers bugs in OpenFlow programs, through a combination of model checking and symbolic execution. Building on our previous paper [4] that argues for automating the testing of OpenFlow applications, we introduce several new contributions summarized in Section 1.1.

1.1 Bugs in OpenFlow Applications

An OpenFlow network consists of a distributed collection of switches managed by a program running on a logically-centralized controller, as illustrated in Figure 1. Each switch has a flow table that stores a list of rules for processing packets. Each rule consists of a pattern (matching on packet header fields) and actions (such as forwarding, dropping, flooding, or modifying the packet, or sending them to the controller). A pattern can require an “exact match” on all relevant header fields (i.e., a rule-matching rule), or have “don't care” bits in some fields (i.e., a wildcard rule). For each rule, the switch maintains traffic counters that measure the bytes and packets processed so far. When a packet arrives, a switch selects the highest-priority matching rule, updates the counters, and performs the specified actions. If no rule matches, the switch sends the packet header to the controller and awaits a response on what actions to take. Switches also send event messages, such as a “port” event joining the network, or “port change” when links go up or down.

The OpenFlow controller maintains rules in the switches, reads traffic statistics, and responds to events. For each event, the controller program defines a handler, which may install rules or issue requests for traffic statistics. Many OpenFlow applications¹ are written on the NOX controller platform [5], which offers an OpenFlow for a much wider range of functionality, through software created by a diverse collection of network operators and

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted by ACM for non-profit educational institutions registered with ACM Copyright 2010.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted by ACM for non-profit educational institutions registered with ACM Copyright 2010.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted by ACM for non-profit educational institutions registered with ACM Copyright 2010.

Existing Tools

There is a cottage industry in SDN configuration-checking tools...

- FlowChecker [SafeConfig '10]
- AntEater [SIGCOMM '11]
- NICE [NSDI '12]
- Header Space Analysis [NSDI '12]

FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures

Debugging the Data Plane with Ant eater

A NICE Way to Test OpenFlow Applications

Header Space Analysis: Static Checking For Networks

Marco Canini*, Daniele Venzano*, Peter Petrášil†, Dejan Kostić*, and Jennifer Rexford†

Payman Kazemian *Stanford University* George Varghese *UCSD and Babco Labs* Nick McKeown *Stanford University*
kazemian@stanford.edu varghese@cs.ucsd.edu mcckeown@cs.stanford.edu

Abstract

Today's networks typically carry or deploy dozens of protocols and mechanisms simultaneously such as MPLS, NAT, ACLs and route redistribution. Even when individual protocols function correctly, failures can arise from the complex interactions of their aggregate, requiring network administrators to be masters of detail. Our goal is to automatically find an important class of failures, regardless of the protocols running, for both operational and experimental networks.

To this end we developed a general and protocol-agnostic framework, called *Header Space Analysis (HSA)*. Our formalism allows us to statically check network specifications and configurations to identify an important class of failures such as *Reachability Failures*, *Forwarding Loops* and *Traffic Isolation and Leakage* problems. In HSA, protocol header fields are not first class entities, instead we look at the entire packet header as a concatenation of bits without any associated meaning. Each packet is a point in the $\{0,1\}^L$ space where L is the maximum length of a packet header, and networking boxes transform packets from one point in the space to another point or set of points (multicast).

We created a library of tools, called *Hazel*, to implement our framework, and used it to analyze a variety of networks and protocols. Hazel was used to analyze the Stanford University backbone network, and found all the forwarding loops in less than 10 minutes, and verified reachability constraints between two subnets in 15 seconds. It also found a large and complex loop in an experimental loose routing protocol in 4 minutes.

1 Introduction

"actions will occur in the best-equipped families" — Charles Dickens

In the beginning, a switch or router was breathtakingly simple. About all the device needed to do was write into a forwarding table using a destination address, and decide where to send the packet next. Over time, forwarding grew more complicated. Middleboxes (e.g., NAT and firewalls) and encapsulation mechanisms (e.g., VLANs and MPLS) appeared to escape from IP's limitations: e.g., NAT bypasses address limits and MPLS

allows flexible routing. Further, new protocols for specific domains, such as data centers, WANs and wireless, have greatly increased the complexity of packet forwarding. Today, there are over 6000 Internet BGCs and it is not unusual for a switch or router to handle ten or more encapsulation formats simultaneously.

This complexity makes it daunting to operate a large network today. Network operators require great sophistication to master the complexity of many interacting protocols and middleboxes. The future is not any more easy complexity today makes operators wary of trying new protocols, even if they are available, for fear of breaking their network. Complexity also makes network fragile, and susceptible to problems where hosts become isolated and unable to communicate. Debugging reachability problems is very time consuming. Even simple questions are hard to answer, such as "Can Host A talk to Host B?", or "Can packet X pass my network?", or "Can I see A's traffic as communication between Clients B and C?". These questions are especially hard to answer in networks carrying multiple encapsulation and containing boxes that filter packets.

Thus, our first goal is to help system administrators statically analyze production networks today. We describe new methods and tools to provide formal answers to these questions, and many other failure conditions, regardless of the protocols running in the network.

Our second goal is to make it easier for system administrators to guarantee isolation between sets of hosts, users or traffic. Partitioning networks this way is usually called "slicing". VLANs are a simple example used today. If configured correctly, we can be confident that traffic in one slice (e.g., a VLAN) cannot leak into another.

This is useful for security, and to help answer questions such as "Can I prevent Host A from talking to Host B?". For example, imagine two health-care providers using the same physical network. HIPAA [20] rules require that no information about a patient can be read by other providers. Thus a natural application of slicing is to place each provider in a separate slice and guarantee that no packet from one slice can be controlled by or read by the other slice. We call this *secure slicing*. Secure slicing may also be useful for banks as part of defense-in-depth, and for classified and unclassified users sharing the same physical network. Our tools can verify that slices have

Existing Tools

There is a cottage industry in SDN configuration-checking tools...

- FlowChecker [SafeConfig '10]
- AntEater [SIGCOMM '11]
- NICE [NSDI '12]
- Header Space Analysis [NSDI '12]
- VeriFlow [HotSDN '12]
- and many others...

FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures

Debugging the Data Plane with AntEater

A NICE Way to Test OpenFlow Applications

Header Space Analysis: Static Checking For Networks

VeriFlow: Verifying Network-Wide Invariants in Real Time

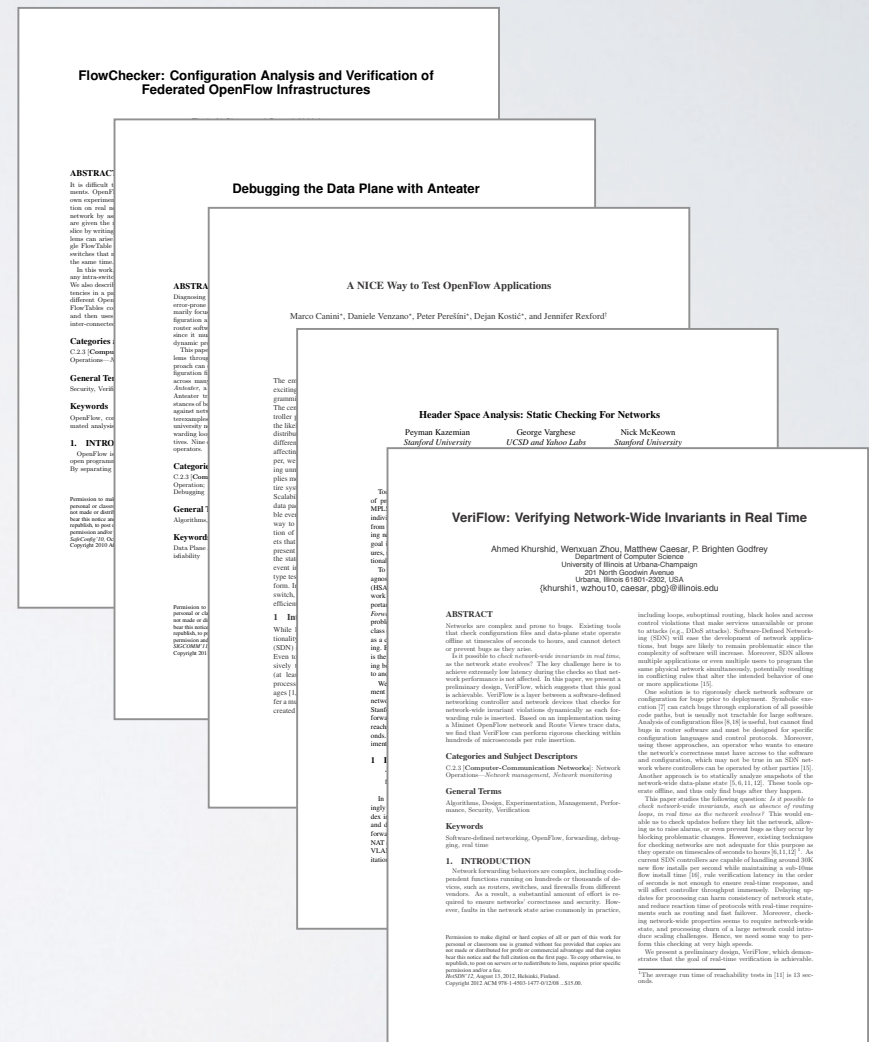
Existing Tools

There is a cottage industry in SDN configuration-checking tools...

- FlowChecker [SafeConfig '10]
- AntEater [SIGCOMM '11]
- NICE [NSDI '12]
- Header Space Analysis [NSDI '12]
- VeriFlow [HotSDN '12]
- and many others...

These are all great tools!

But they are expensive to run, and each builds on a custom (typically ad hoc) foundation



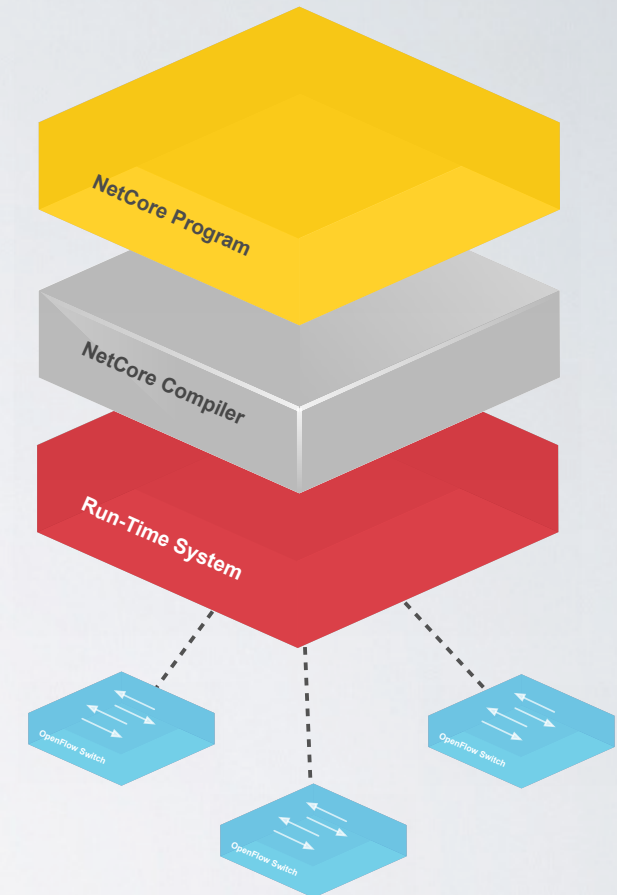
Machine-Verified Controllers

Vision

- Develop programs in a high-level language
- Reason at a high level of abstraction
- Use a compiler and run-time system to generate low-level control messages
- Machine-verified proofs of correctness

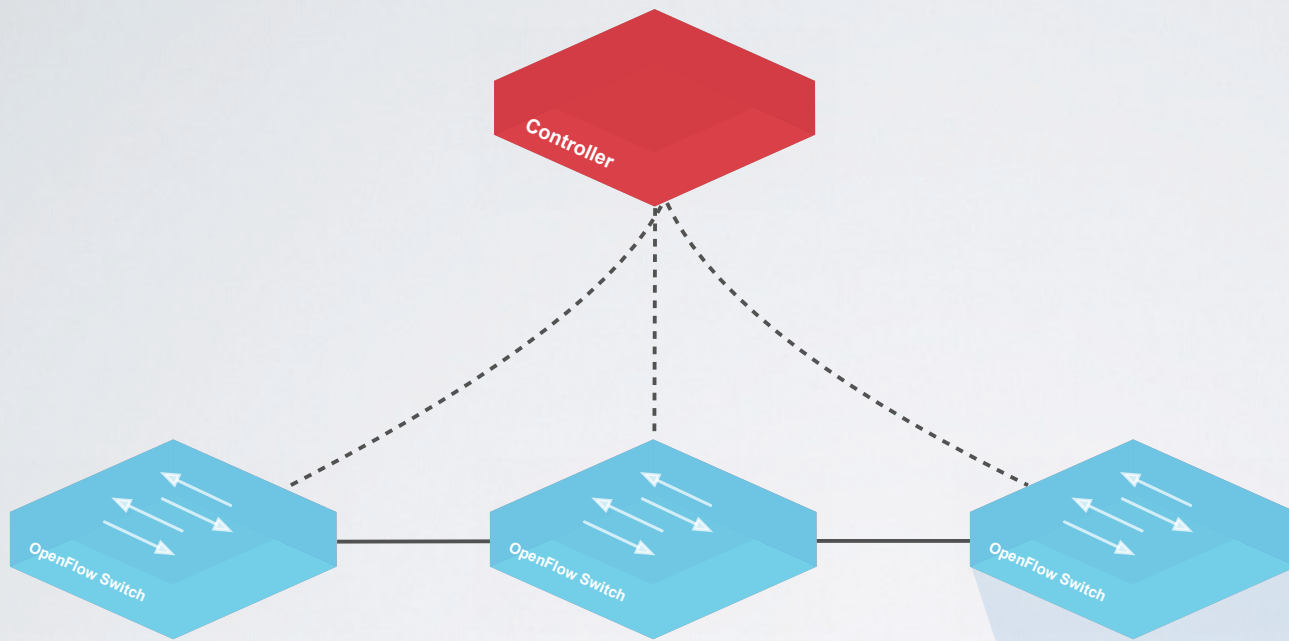
Contributions

- NetCore compiler + optimizer
- Featherweight OpenFlow model
- General framework for establishing run-time system correctness



OVERVIEW

OpenFlow Switches




Forwarding Table: prioritized list of rules

Rule: pattern, actions, and counters

Pattern: prefix match on headers

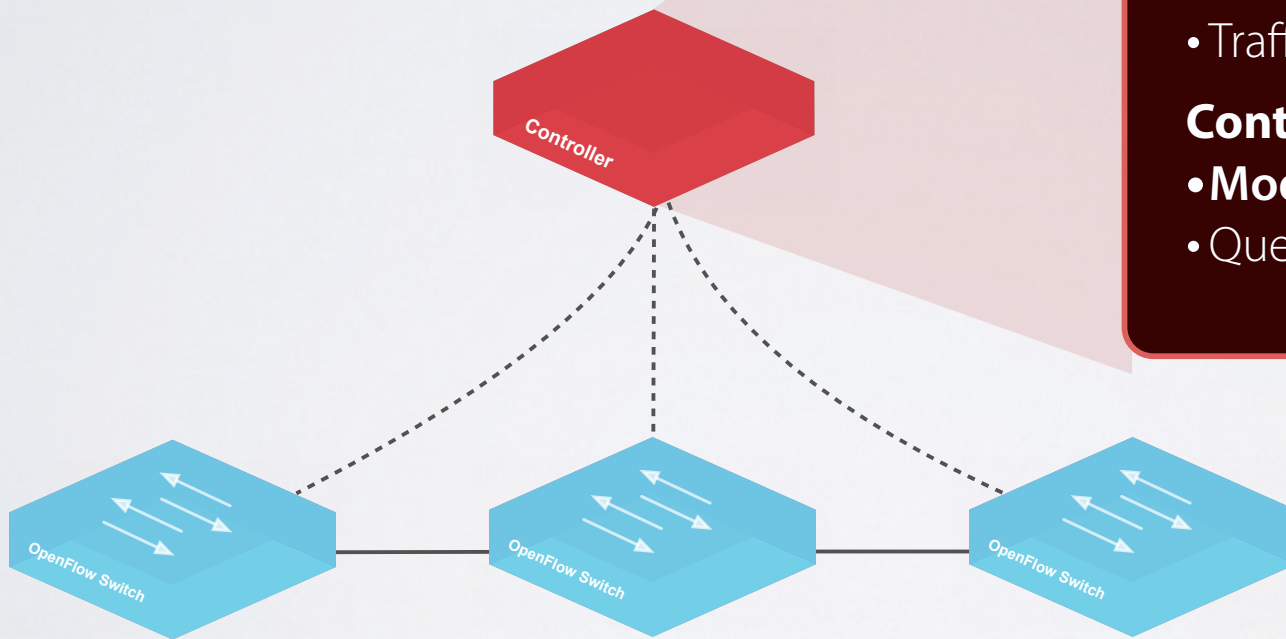
Action: forward or modify

Counters: total bytes and packets processed



OpenFlow

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



NOX

Network Events

- Topology changes
- **Diverted packets**
- Traffic statistics

Control Messages

- **Modify rules**
- Query counters

Issue #1: Switch-Level Errors



What happens if...

- The controller misses a keep-alive message?
- The controller sends a malformed message?
 - Bad output port
 - Too many actions
 - Inconsistent actions
 - Unsupported actions
- The switches runs out of space for rules?

Any of these can lead to essentially arbitrary behavior

Issue #2: Malformed Patterns

What happens if the controller sends the following message to a switch?

```
FlowMod AddFlow { match = { srcIPAddress = 10.0.1.*", ... },  
                  actions = [ flood ], ... }
```

Issue #2: Malformed Patterns

What happens if the controller sends the following message to a switch?

```
FlowMod AddFlow { match = { srcIPAddress = 10.0.1.*", ... },  
                  actions = [ flood ], ... }
```

We'd expect the switch to install a rule that broadcasts all traffic from a host the given subnet...

Issue #2: Malformed Patterns

What happens if the controller sends the following message to a switch?

```
FlowMod AddFlow { match = { srcIPAddress = 10.0.1.*", ... },  
                  actions = [ flood ], ... }
```

We'd expect the switch to install a rule that broadcasts all traffic from a host the given subnet...

...but it actually installs a rule that floods *all* traffic

Issue #2: Malformed Patterns

What happens if the controller sends the following message to a switch?

```
FlowMod AddFlow { match = { srcIPAddress = 10.0.1.*", ... },  
                  actions = [ flood ], ... }
```

We'd expect the switch to install a rule that broadcasts all traffic from a host the given subnet...

...but it actually installs a rule that floods *all* traffic

Why? Switches *silently* ignore IP fields unless the Ethernet frame type is IP!

Issue #3: Message Reordering

What happens if the controller sends the following pair of OpenFlow messages to a switch in sequence?

```
FlowMod AddFlow { match = { ethFrameType = ethTypeIP,  
                             srcIPAddress =  
"10.0.1.99", ... },  
                  priority = 1,  
                  actions = [ ] }  
  
FlowMod AddFlow { match = { ethFrameType = ethTypeIP,  
                             srcIPAddress = "10.0.1.*", ... },  
                  priority = 2,  
                  actions = [ flood ] }
```

The intention is to encode a negation...

Issue #3: Message Reordering

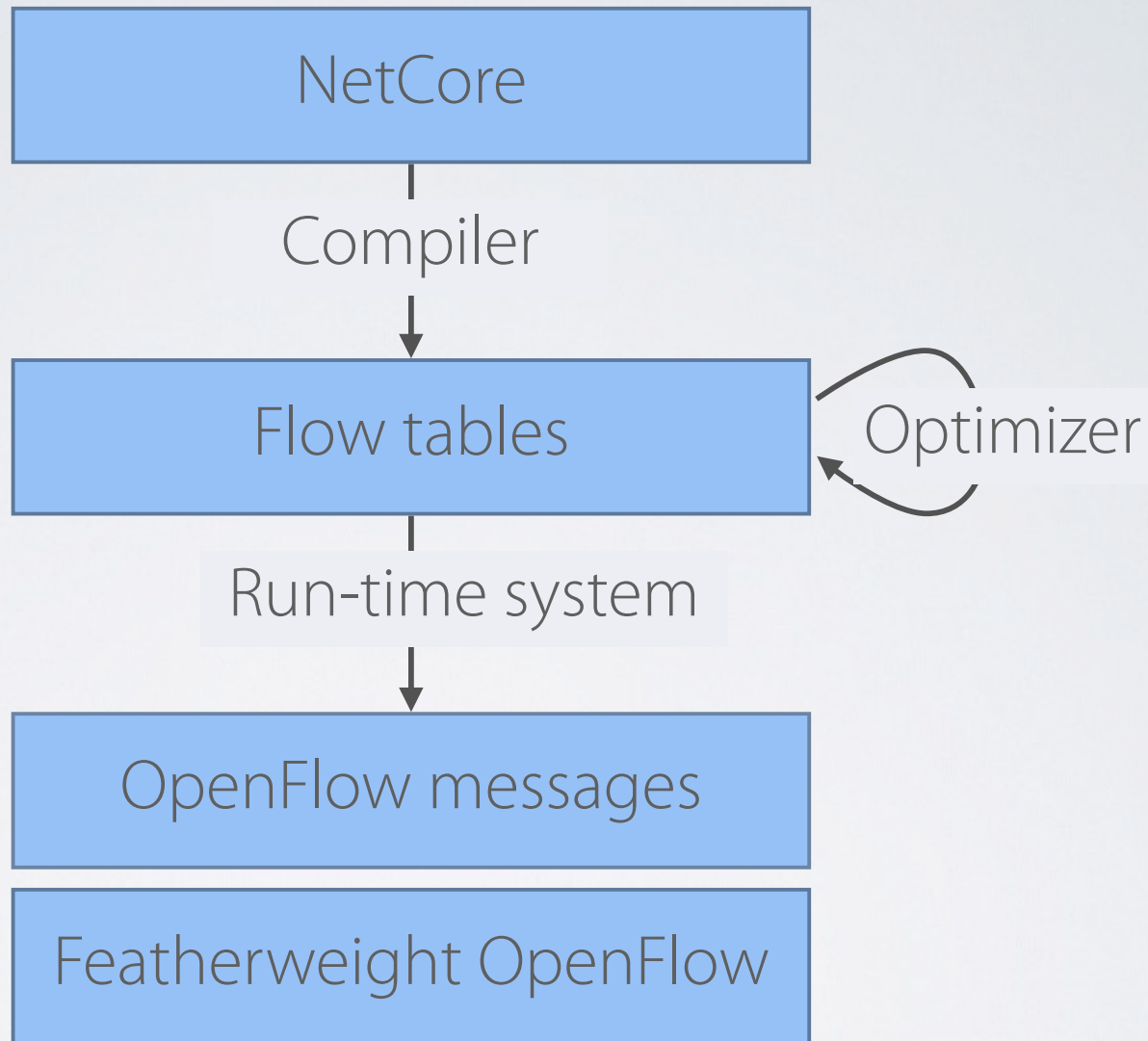
What happens if the controller sends the following pair of OpenFlow messages to a switch in sequence?

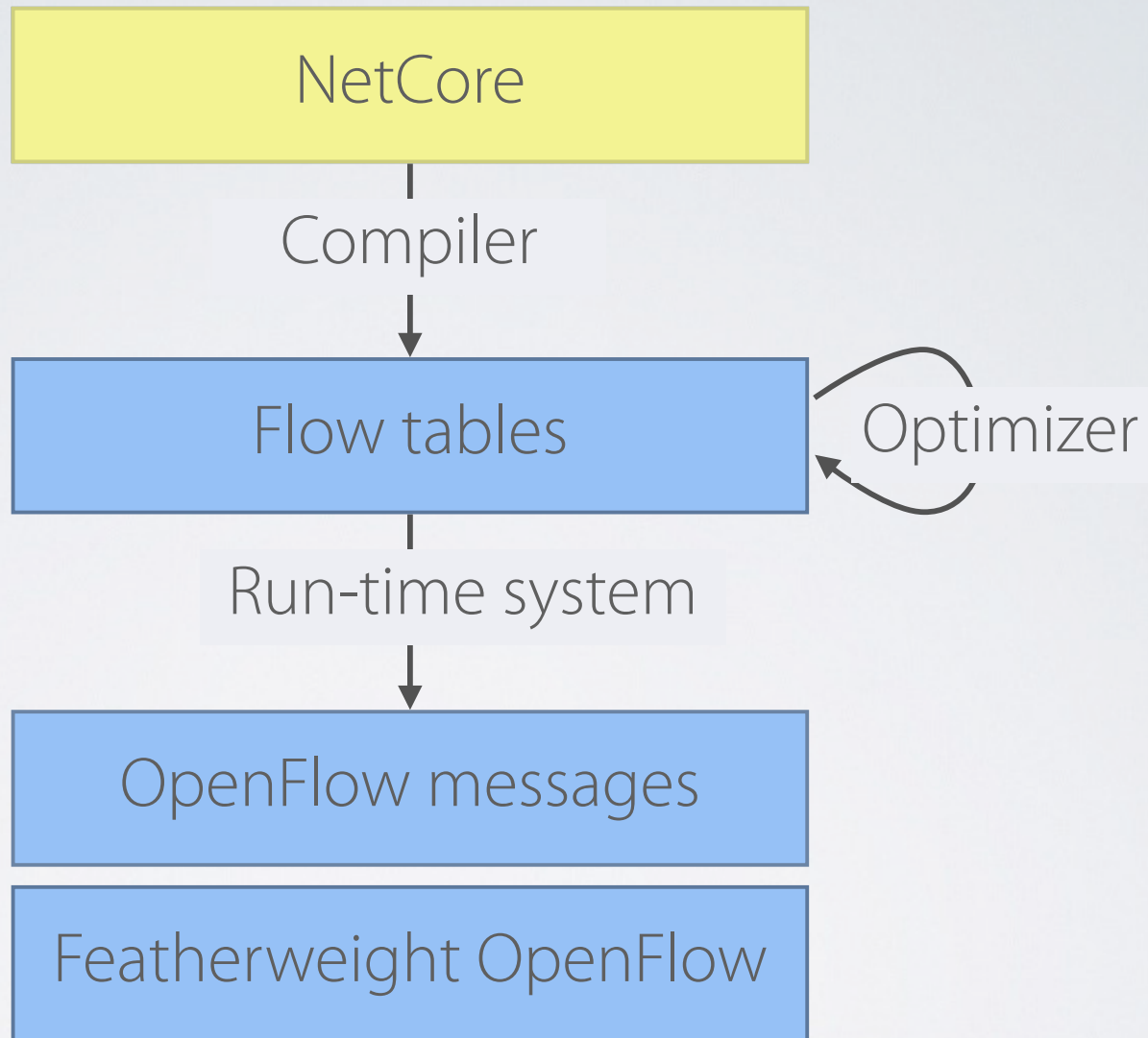
```
FlowMod AddFlow { match = { ethFrameType = ethTypeIP,  
                             srcIPAddress =  
"10.0.1.99", ... },  
                  priority = 1,  
                  actions = [ ] }  
  
FlowMod AddFlow { match = { ethFrameType = ethTypeIP,  
                             srcIPAddress = "10.0.1.*", ... },  
                  priority = 2,  
                  actions = [ flood ] }
```

The intention is to encode a negation...

...but the switch may process these in either order!

MACHINE-VERIFIED CONTROLLERS





Syntax

```

Inductive pred : Type :=
  | OnSwitch : Switch -> pred
  | IngressPort : Port -> pred
  | DlSrc : EthernetAddress -> pred
  | DlDst : EthernetAddress -> pred
  | DlVlan : option VLAN -> pred
  | ...
  | And : pred -> pred -> pred
  | Or : pred -> pred -> pred
  | Not : pred -> pred
  | All : pred
  | None : pred.
  
```

(* Predicates *)

```

Inductive PseudoPort : Type :=
  | PhysicalPort : Port -> PseudoPort
  | AllPorts : PseudoPort.
  
```

(* Psuedo Ports *)

```

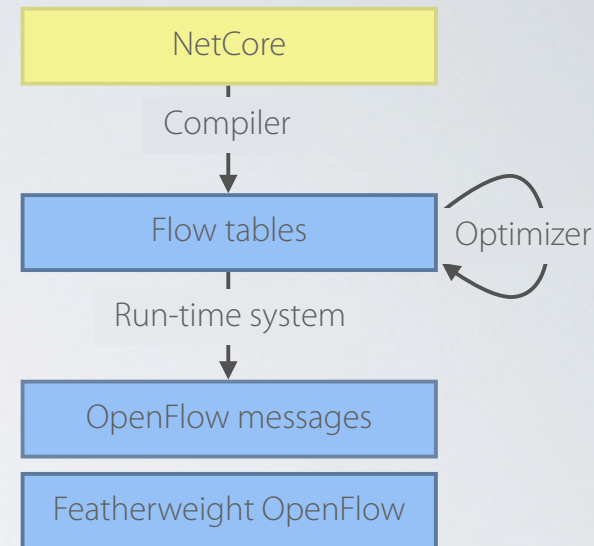
Inductive act : Type :=
  | FwdMod : Mod -> PseudoPort -> act
  
```

(* Actions *)

```

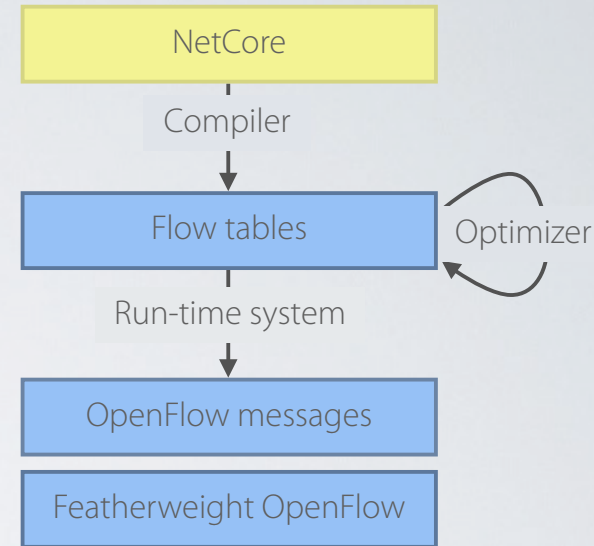
Inductive pol : Type :=
  | Policy : pred -> list act -> pol
  | Union : pol -> pol -> pol
  | Restrict : pol -> pred -> pol.
  
```

(* Policies *)

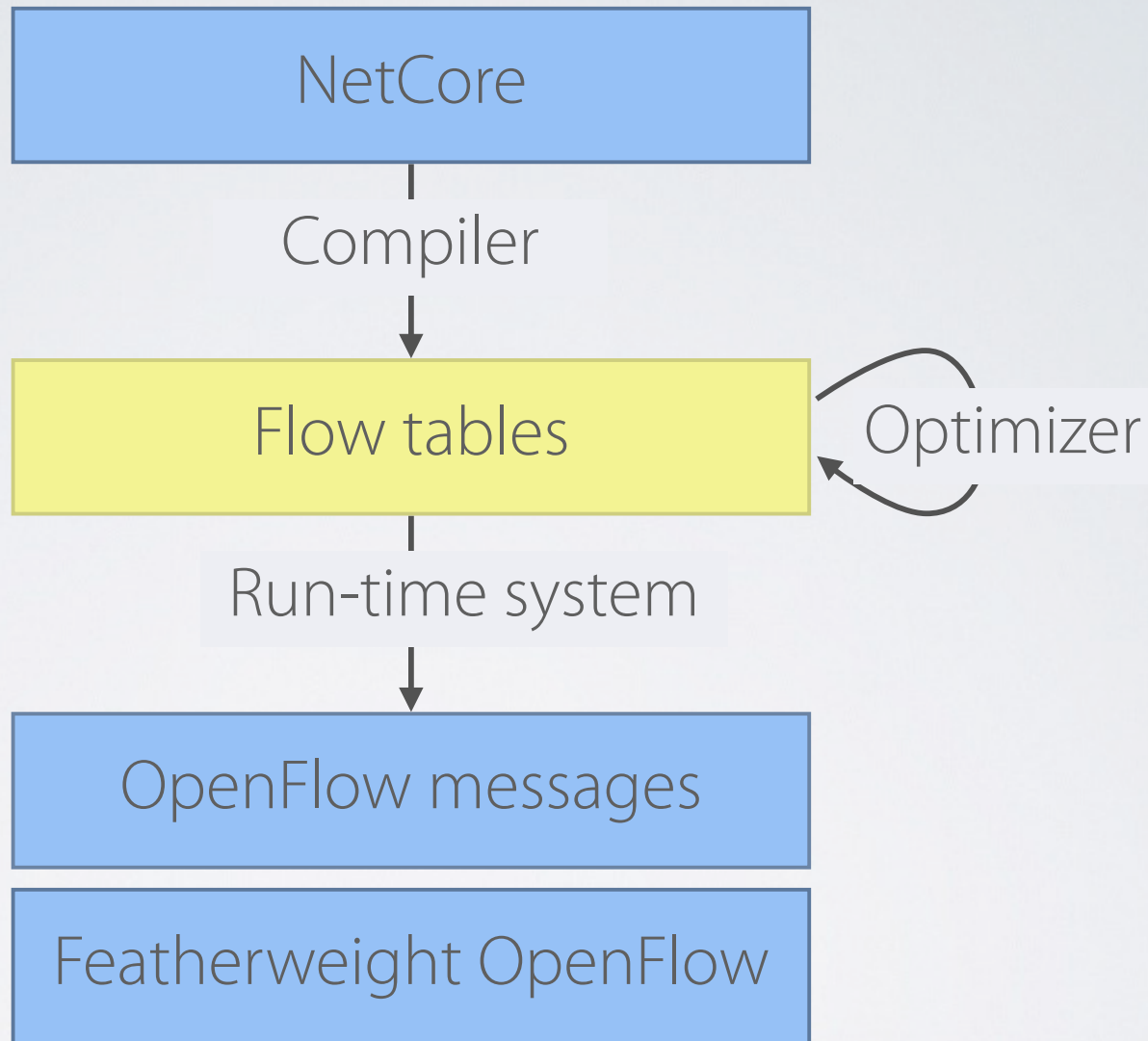


Semantics

$$\begin{aligned}
 lp &= (sw, pt, pk) \\
 lps_{out} &= pol(sw, pt, pk) \\
 S &= \{(T(sw, pt_{out}), pk) \mid (pt_{out}, pk) \in lps_{out}\} \\
 \hline
 \{lp\} \uplus \{lp_1 \cdots lp_n\} &\xrightarrow{lp} S \uplus \{lp_1 \cdots lp_n\}
 \end{aligned}$$



- Models hop-by-hop forwarding behavior of the network
- Abstracts away from the underlying distributed system
- Makes it easy to reason about network-wide properties



NetCore to Flow Tables

Example

Priority	Pattern	Action
65534	inPort = 2, dlSrc = dc:ba:65:43:21	Fwd 2
65533	inPort = 2	Fwd 3

NetCore compiler

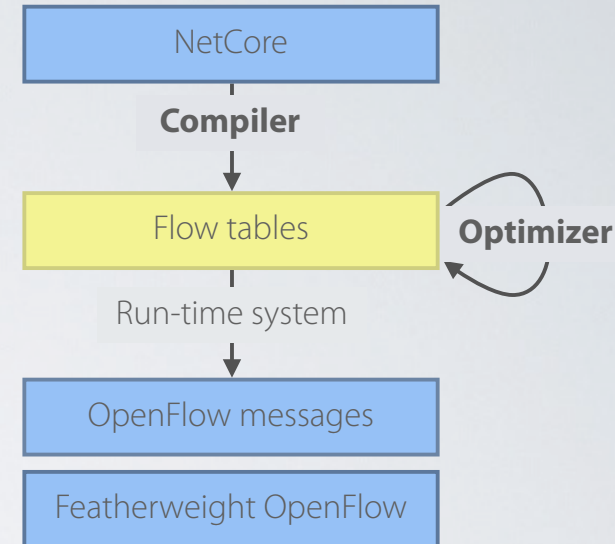
- Key operation: flow table intersection
- Must restrict to “valid” patterns

Optimizer

- Optimizer prunes (many) redundant rules
- Based on simple algebra of operations

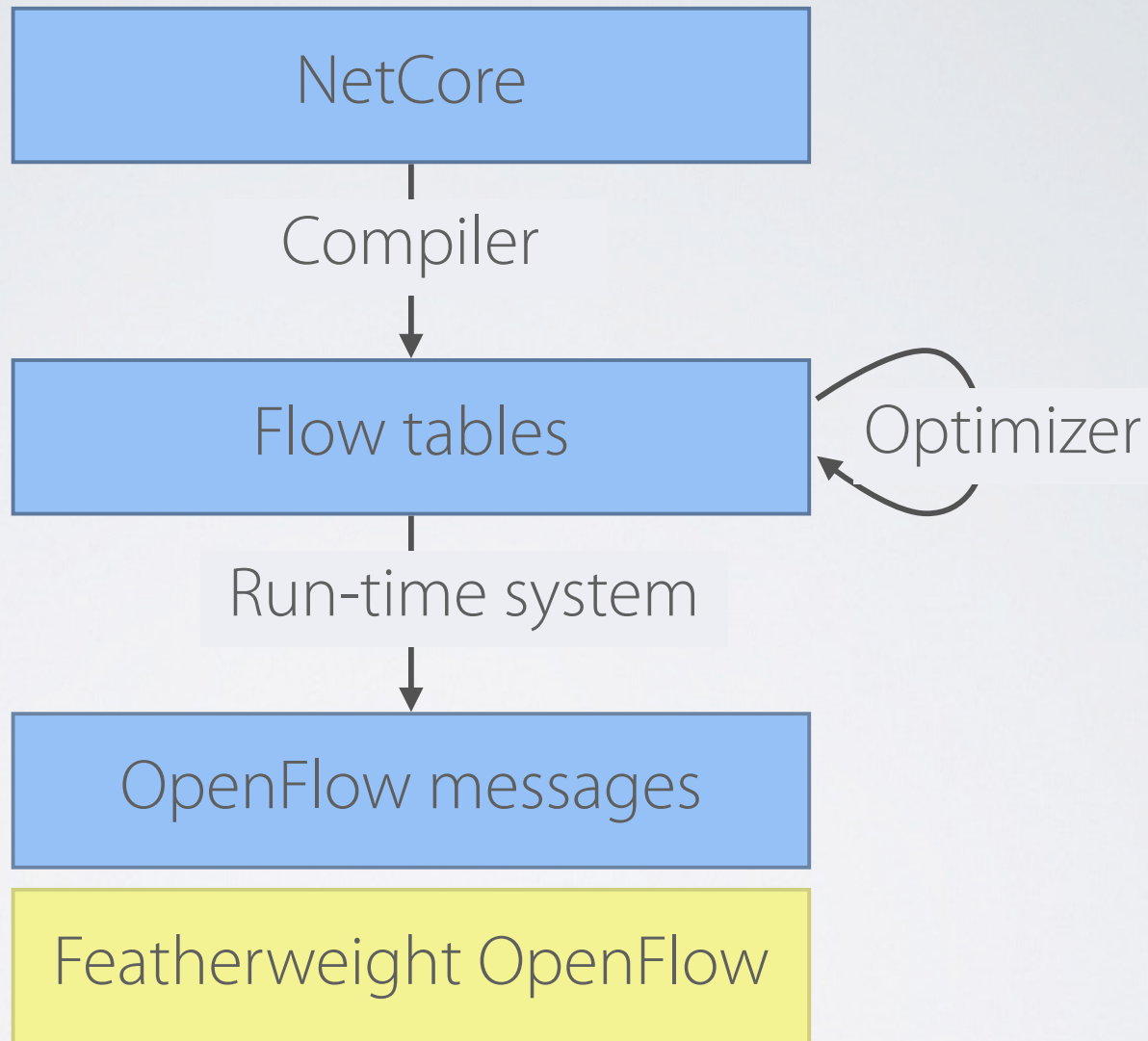
Correctness Theorem

NetCore \sim FlowTable



Valid Patterns

```
Inductive ValidPattern : Pattern -> Prop :=
| SupportedIPPatternValid : forall d1Src d1Dst d1Vlan d1VlanPcp nwSrc nwDst nwTos
    tpSrc tpDst inPort nwProto,
    In nwProto SupportedL4Protos ->
    ValidPattern (MkPattern d1Src d1Dst (WildcardExact Const_0x800)
        d1Vlan d1VlanPcp
        nwSrc nwDst (WildcardExact nwProto)
        nwTos tpSrc tpDst inPort)
| UnsupportedIPPatternValid : forall d1Src d1Dst d1Vlan d1VlanPcp nwSrc nwDst nwTos
    inPort nwProto,
    ~ In nwProto SupportedL4Protos ->
    ValidPattern (MkPattern d1Src d1Dst (WildcardExact Const_0x800)
        d1Vlan d1VlanPcp
        nwSrc nwDst (WildcardExact nwProto)
        nwTos WildcardAll WildcardAll inPort)
| ARPPacketValid : forall d1Src d1Dst d1Vlan d1VlanPcp nwSrc nwDst inPort,
    ValidPattern (MkPattern d1Src d1Dst (WildcardExact Const_0x806)
        d1Vlan d1VlanPcp
        nwSrc nwDst WildcardAll
        WildcardAll WildcardAll WildcardAll inPort)
| UnknownD1TypPatternValid : forall d1Src d1Dst d1Typ d1Vlan d1VlanPcp inPort,
    ValidPattern (MkPattern d1Src d1Dst d1Typ
        d1Vlan d1VlanPcp
        WildcardAll WildcardAll WildcardAll
        WildcardAll WildcardAll WildcardAll inPort)
| EmptyPatternValid :
    ValidPattern Pattern_empty.
```

OpenFlow Specification



42 pages...

...of informal English text

...and C struct definitions

Featherweight OpenFlow

Syntax

Devices	Switch	S	$::= \mathbb{S}(sw, pts, RT, inp, outp, inm, out)$
	Controller	C	$::= \mathbb{C}(\sigma, f_{in}, f_{out})$
	Link	L	$::= \mathbb{L}(loc_{src}, pks, loc_{dst})$
	OpenFlow Link to Controller	M	$::= \mathbb{M}(sw, SMS, CMS)$
Packets and Locations	Packet	pk	$::= abstract$
	Switch ID	sw	$\in \mathbb{N}$
	Port ID	pt	$\in \mathbb{N}$
	Location	loc	$\in sw \times pt$
	Located Packet	lp	$\in loc \times pk$
Controller Components	Controller state	σ	$::= abstract$
	Controller input relation	f_{in}	$\in sw \times CM \times \sigma \rightsquigarrow \sigma$
	Controller output relation	f_{out}	$\in \sigma \rightsquigarrow sw \times SM \times \sigma$
Switch Components	Rule table	RT	$::= abstract$
	Rule table Interpretation	$\llbracket RT \rrbracket$	$\in lp \rightarrow \{\{lp_1 \dots lp_n\} \times \{CM_1 \dots CM_n\}\}$
	Rule table modifier	ΔRT	$::= abstract$
	Rule table modifier interpretation	apply	$\in \Delta RT \rightarrow RT \rightarrow \Delta RT$
	Ports on switch	pts	$\in \{pt_1 \dots pt_n\}$
	Consumed packets	inp	$\in \{lp_1 \dots lp_n\}$
	Produced packets	$outp$	$\in \{lp_1 \dots lp_n\}$
	Messages from controller	inm	$\in \{SM_1 \dots SM_n\}$
	Messages to controller	$outm$	$\in \{CM_1 \dots CM_n\}$
Link Components	Endpoints	loc_{src}, loc_{dst}	$\in loc$ where $loc_{src} \neq loc_{dst}$
	Packets from loc_{src} to loc_{dst}	pks	$\in \{pk_1 \dots pk_n\}$
Controller Link	Message queue from controller	SMS	$\in \{SM_1 \dots SM_n\}$
	Message queue to controller	CMS	$\in \{CM_1 \dots CM_n\}$
Abstract OpenFlow Protocol	Message from controller	SM	$::= \mathbf{FlowMod} \Delta RT \mid \mathbf{PktOut} \ pt \ l$
	Message to controller	CM	$::= \mathbf{PktIn} \ pt \ pk \mid \mathbf{BarrierReply} \ n$

Key judgments:

- Controller in: $(sw, CM, \sigma) \rightsquigarrow \sigma'$
- Controller out: $\sigma \rightsquigarrow (sw, SM, \sigma')$
- Network step: $M \rightarrow M'$

Models *all* essential asynchrony

Semantics

$$\frac{(outp', outm') = \llbracket RT \rrbracket(lp)}{\mathbb{S}(sw, pts, RT, \{\{lp\}\} \uplus inp, outp, inm, outm) \xrightarrow{lp} \mathbb{S}(sw, pts, RT, inp, outp' \uplus outp, inm, outm' \uplus outm)} \text{ (PKT-PROCESS)}$$

$$\frac{\mathbb{S}(sw, pts, RT, inp, \{\{(sw, pt, pk)\}\} \uplus outp, inm, outm) \mid \mathbb{L}((sw, pt), pks, loc')}{\rightarrow \mathbb{S}(sw, pts, RT, inp, outp, inm, outm) \mid \mathbb{L}((sw, pt), [pk] \uplus pks, loc')} \text{ (SEND-WIRE)}$$

$$\frac{\mathbb{L}(loc, pks \uplus [pk], (sw, pt)) \mid \mathbb{S}(sw, pts, RT, inp, outp, inm, outm)}{(\text{sw}, pt, pk) \ \mathbb{L}(loc, pks, (sw, pt)) \mid \mathbb{S}(sw, pts, RT, \{\{(sw, pt, pk)\}\} \uplus inp, outp, inm, outm)} \text{ (RECV-WIRE)}$$

$$\frac{RT' = \text{apply}(\Delta RT, RT)}{\mathbb{S}(sw, pts, RT, inp, outp, \{\mathbf{FlowMod} \ \Delta RT\} \uplus inm, outm) \rightarrow \mathbb{S}(sw, pts, RT', inp, outp, inm, outm)} \text{ (SWITCH-FLOWMOD)}$$

$$\frac{pt \in pts}{\mathbb{S}(sw, pts, RT, inp, outp, \{\mathbf{PktOut} \ pt \ pk\} \uplus inm, outm) \rightarrow \mathbb{S}(sw, pts, RT, inp, \{\{(sw, pt, pk)\}\} \uplus outp, inm, outm)} \text{ (SWITCH-PKTOU)}$$

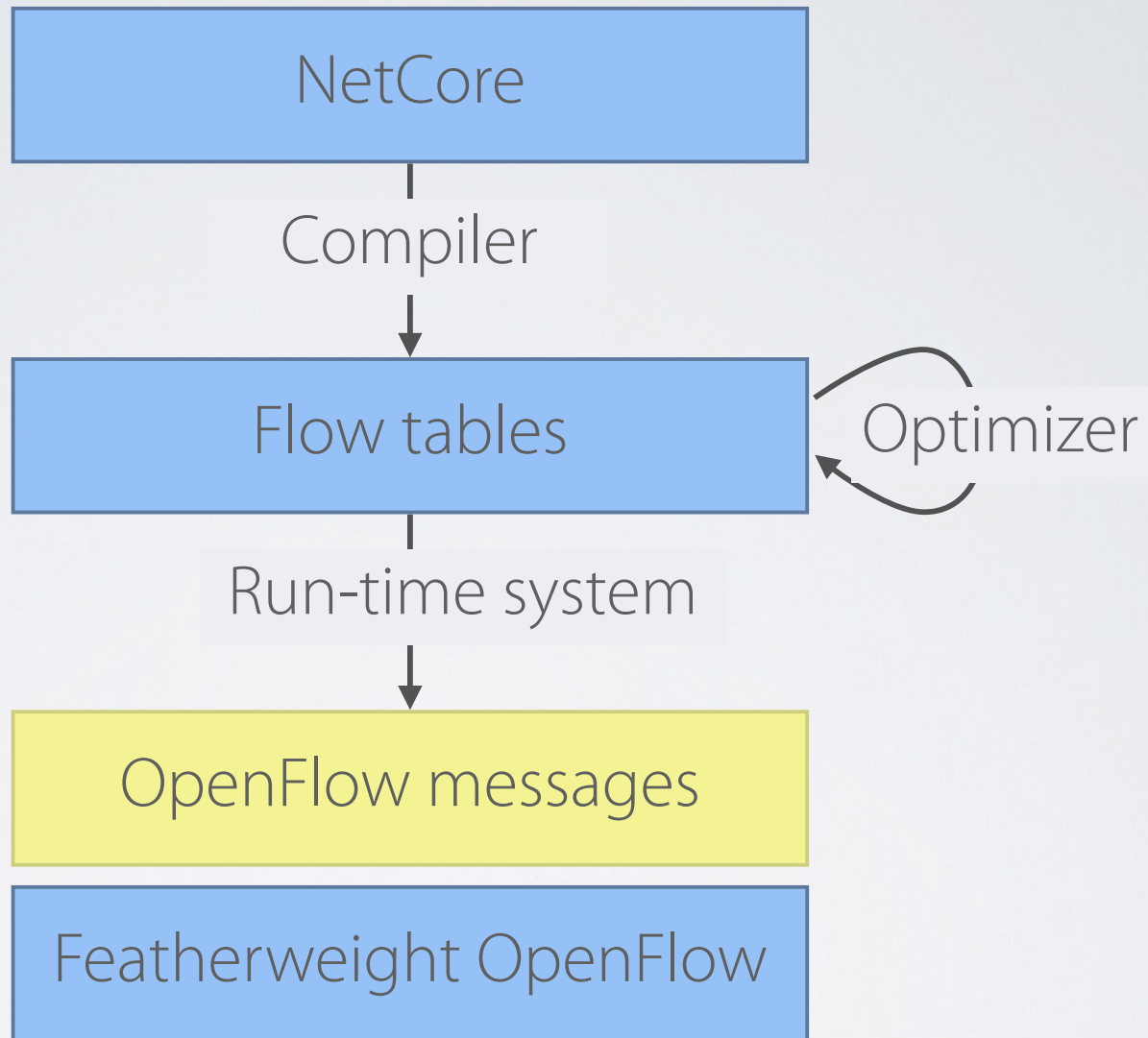
$$\frac{f_{out}(\sigma \rightsquigarrow (sw, SM, \sigma'))}{\mathbb{C}(\sigma, f_{in}, f_{out}) \mid \mathbb{M}(sw, SMS, CMS) \rightarrow \mathbb{C}(\sigma', f_{in}, f_{out}) \mid \mathbb{M}(sw, [SM] \uplus SMS, CMS)} \text{ (CTRL-SEND)}$$

$$\frac{f_{in}(sw, \sigma, CM) \rightsquigarrow \sigma'}{\mathbb{C}(\sigma, f_{in}, f_{out}) \mid \mathbb{M}(sw, SMS, CMS \uplus [CM]) \rightarrow \mathbb{C}(\sigma', f_{in}, f_{out}) \mid \mathbb{M}(sw, SMS, CMS)} \text{ (CTRL-RECV)}$$

$$\frac{SM \neq \mathbf{BarrierRequest} \ n}{\mathbb{M}(sw, SMS \uplus [SM], CMS) \mid \mathbb{S}(sw, pts, RT, inp, outp, inm, outm) \rightarrow \mathbb{M}(sw, SMS, CMS) \mid \mathbb{S}(sw, pts, RT, inp, outp, \{\{SM\}\} \uplus inm, outm)} \text{ (SWITCH-RECV-CTRL)}$$

$$\frac{\mathbb{M}(sw, SMS \uplus [\mathbf{BarrierRequest} \ n], CMS) \mid \mathbb{S}(sw, pts, RT, inp, outp, \emptyset, outm)}{\rightarrow \mathbb{M}(sw, SMS, CMS) \mid \mathbb{S}(sw, pts, RT, inp, outp, \emptyset, \{\mathbf{BarrierReply} \ n\} \uplus outm)} \text{ (SWITCH-RECV-BARRIER)}$$

$$\frac{\mathbb{S}(sw, pts, RT, inp, outp, inm, \{\{CM\}\} \uplus outm) \mid \mathbb{M}(sw, SMS, CMS)}{\rightarrow \mathbb{S}(sw, pts, RT, inp, outp, inm, outm) \mid \mathbb{M}(sw, SMS, [CM] \uplus CMS)} \text{ (SWITCH-SEND-CTRL)}$$



Run-Time System

Invariants

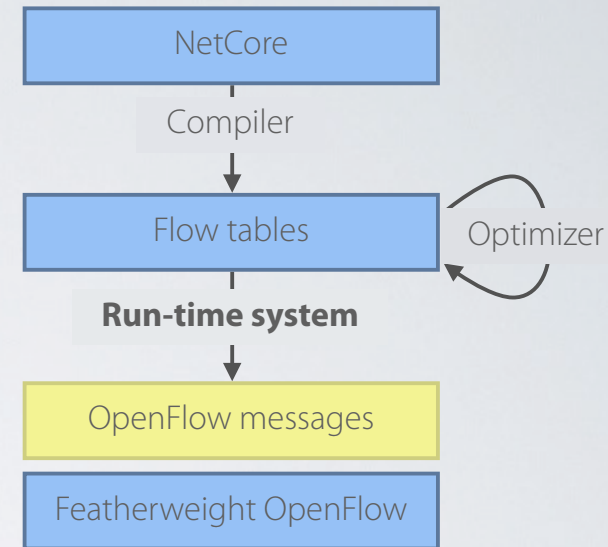
- Maintain a sound approximation of overall flow table each switch
- Eventually process all diverted packets

Theorem

FlowTable \approx Featherweight OpenFlow

Run-time instances

- Trivial: processes all packets on controller
- Proactive: installs rules, falls back to Trivial when out of space
- Full: like Proactive, but also installs exact-match rules



Safe Wires

```
Inductive SafeWire : SF -> SF -> SF -> list CM -> Prop :=
| SafeWire_nil : forall lb ub,
  extends ub lb ->
  SafeWire lb ub lb nil
| SafeWire_cons_FlowMod : forall lb ub sf sft lst,
  SafeWire lb ub sf lst ->
  extends ub (apply_SFT sft sf) ->
  SafeWire lb ub (apply_SFT sft sf) (FlowMod sft :: lst)
| SafeWire_cons_PktOut : forall lb ub sf pt pk lst,
  SafeWire lb ub sf lst ->
  SafeWire lb ub sf (PktOut pt pk :: lst)
| SafeWire_cons_BarrierRequest : forall lb ub sf n lst,
  SafeWire lb ub sf lst ->
  SafeWire lb ub sf (BarrierRequest n :: lst).
```

Implementation

Source

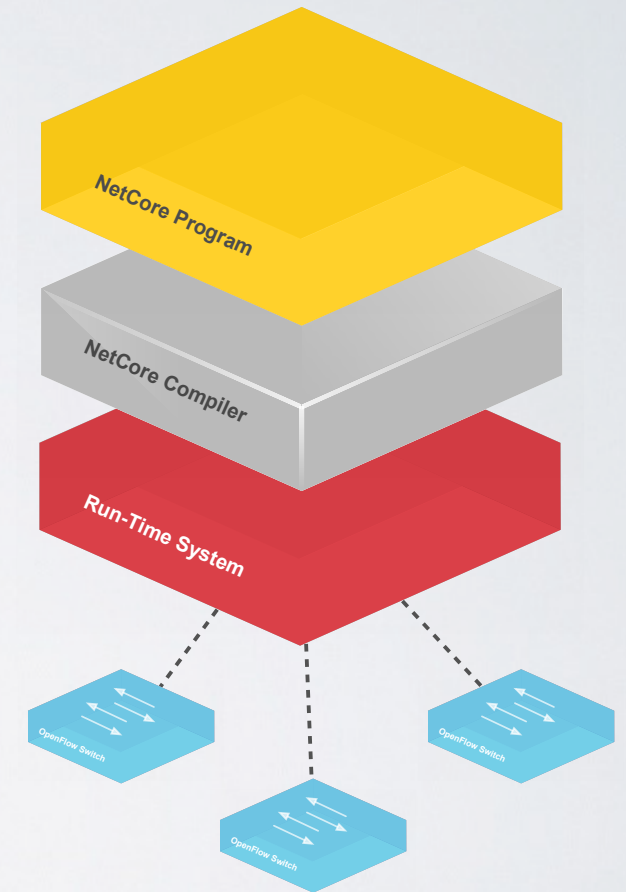
- ~8,000 lines of Coq
- ~1,500 lines of Haskell

Components

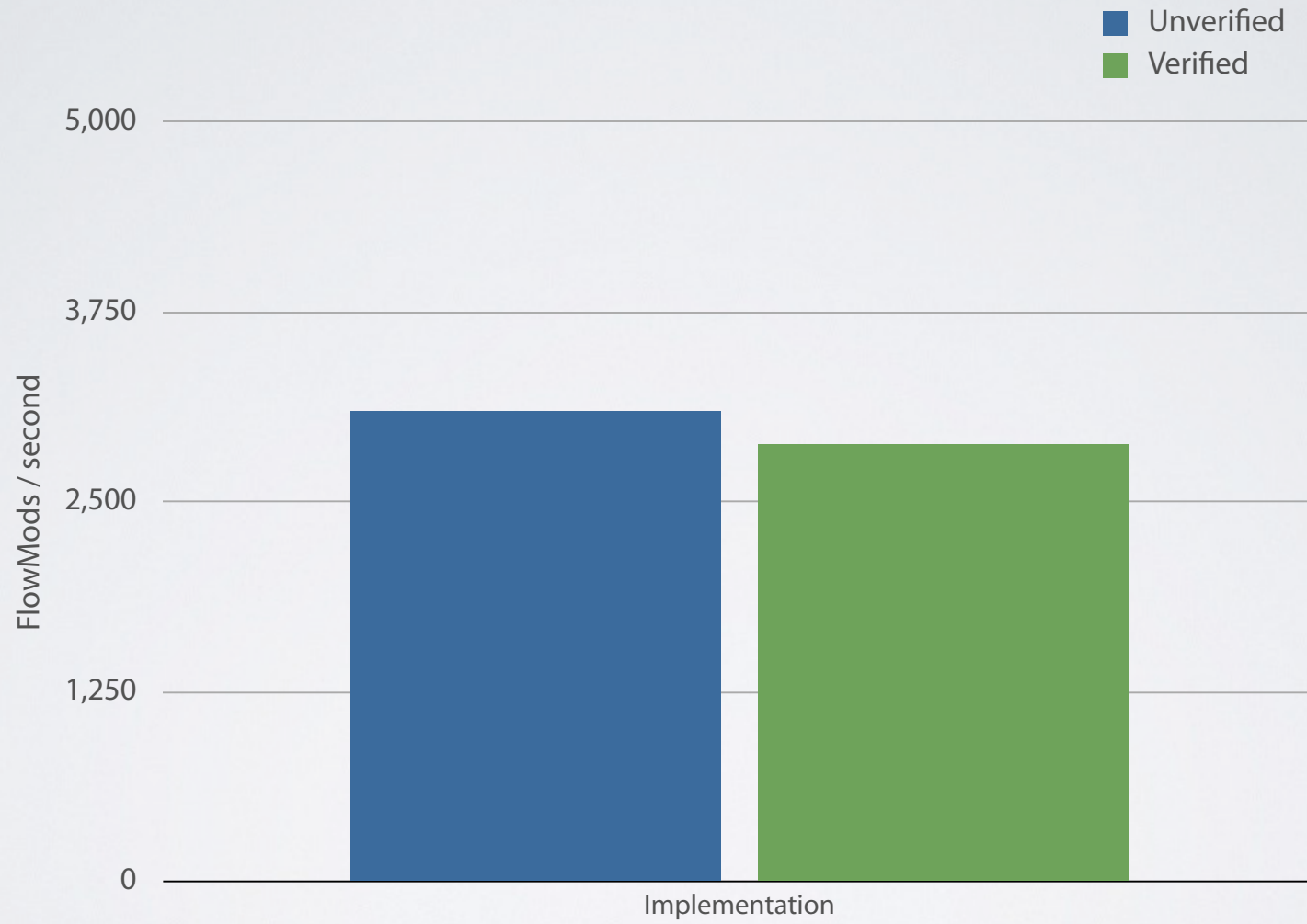
- NetCore compiler and optimizer
- Flow tables
- Featherweight OpenFlow
- Run-time system instances
- Proofs of correctness

Status

- Extracts to Haskell source code
- Compiles against Nettle libraries
- Running on “production” traffic in the lab



Performance



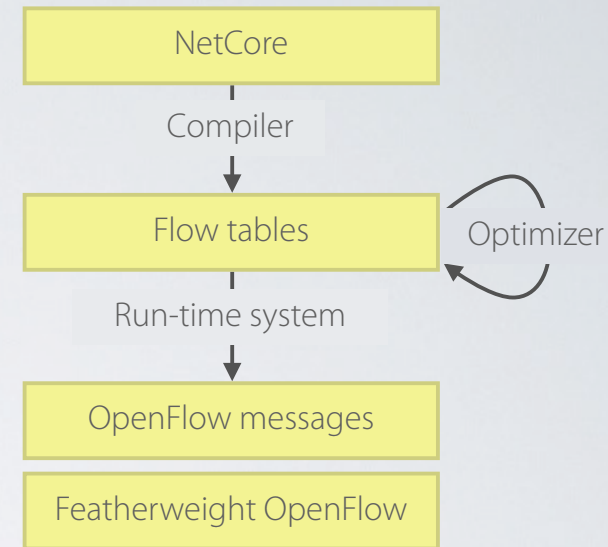
Conclusion

Networks are critical infrastructure...

...developed using 1970s-era techniques

Software-defined networks are an architecture that could be used to put networks on a solid foundation

Machine-verified controllers based on NetCore a first step in this direction



A Grand Collaboration: Languages + Networking

Frenetic Cornell



Shrutarshi Basu (PhD)

Nate Foster (Faculty)

Arjun Guha (Postdoc)

Stephen Gutz (Undergrad)

Mark Reitblatt (PhD)

Robert Soulé (Postdoc)

Alec Story (Undergrad)

Frenetic Princeton

Chris Monsanto (PhD)

Joshua Reich (Postdoc)

Jen Rexford (Faculty)

Cole Schlesinger (PhD)

Dave Walker (Faculty)

Naga Praveen Katta (PhD)



frenetic >>

<http://frenetic-lang.org>