# Managing the Network with Merlin


Robert Soulé
Shrutarshi Basu
Robert Kleinberg
Emin Gün Sirer
Nate Foster



flickr.com/photos/retiredinwasaga

# This talk is *not* about...

*Machine-Verified Network Controllers*

- Arjun Guha (Cornell)
- Mark Reitblatt (Cornell)
- Nate Foster (Cornell)

In PLDI '13.

## Machine-Verified Network Controllers


Arjun Guha
Cornell University
arjun@cs.cornell.edu

Mark Reitblatt
Cornell University
reitblatt@cs.cornell.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.edu



### Abstract

In many areas of computing, techniques ranging from testing to formal modeling to full-blown verification have been successfully used to help programmers build reliable systems. But although networks are critical infrastructure, they have largely resisted analysis using formal techniques. Software-defined networking (SDN) is a new network architecture that has the potential to provide a foundation for network reasoning, by standardizing the interfaces used to express network programs and giving them a precise semantics.

This paper describes the design and implementation of the first machine-verified SDN controller. Starting from the foundations, we develop a detailed operational model for OpenFlow (the most popular SDN platform) and formalize it in the Coq proof assistant. We then use this model to develop a verified compiler and run-time system for a high-level network programming language. We identify bugs in existing languages and tools built without formal foundations, and prove that these bugs are absent from our system. Finally, we describe our prototype implementation and our experiences using it to build practical applications.



***Categories and Subject Descriptors*** F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

***Keywords*** Software-defined networking, OpenFlow, formal verification, Coq, domain-specific languages, NetCore, Frenetic.


### 1. Introduction

Networks are some of the most critical infrastructure in modern society and also some of the most fragile! Networks fail with alarming frequency, often due to simple misconfigurations or software bugs [8, 19, 30]. The recent news headlines contain numerous examples of network failures leading to disruptions: a configuration error during routine maintenance at Amazon triggered a sequence of cascading failures that brought down a datacenter and the customer machines hosted there; a corrupted routing table at GoDaddy disconnected their domain name servers for a day and caused a widespread outage; and a network connectivity issue at United Airlines took down their reservation system, leading to thousands of flight cancellations and a "ground stop" at their San Francisco hub.

One way to make networks more reliable would be to develop tools for checking important network invariants automatically. These tools would allow administrators to answer questions such as: "does this configuration provide connectivity to every host in the network?" or "does this configuration correctly enforce the access control policy?" or "does this configuration have a forwarding loop?" or "does this configuration properly isolate trusted and untrusted traffic?" Unfortunately, until recently, building such tools has been effectively impossible due to the complexity of today's networks. A typical enterprise or datacenter network contains thousands of heterogeneous devices, from routers and switches, to web caches and load balancers, to monitoring middleboxes and firewalls. Moreover, each device executes a stack of complex protocols and is configured through a proprietary and idiosyncratic interface. To reason formally about such a network, an administrator (or tool) must reason about the proprietary programs running on each distributed device, as well as the asynchronous interactions between them. Although formal models of traditional networks exist, they have either been too complex to allow effective reasoning, or too abstract to be useful. Overall, the incidental complexity of networks has made reasoning about their behavior practically infeasible.

Fortunately, recent years have seen growing interest in a new kind of network architecture that could provide a foundation for network reasoning. In a *software-defined network* (SDN), a program on a logically-centralized *controller machine* defines the overall policy for the network, and a collection of *programmable switches* implement the policy using efficient packet-processing hardware. The controller and switches communicate via an open and standard interface. By carefully installing packet-processing rules in the hardware tables provided on switches, the controller can effectively manage the behavior of the entire network.

Compared to traditional networks, SDNs have two important simplifications that make them amenable to formal reasoning. First, they relocate control from distributed algorithms running on individual devices to a single program running on the controller. Second, they eliminate the heterogeneous devices used in traditional networks—switches, routers, load balancers, firewalls, etc.—and replace them with stock programmable switches that provide a standard set of features. Together, this means that the behavior of the network is determined solely by the sequence of configuration instructions issued by the controller. To verify that the network has some property, an administrator (or tool) simply has to reason about the states of the switches as they process instructions.

In the networking community, there is burgeoning interest in tools for checking network-wide properties automatically. Systems such as FlowChecker [1], Header Space Analysis [12], Anteater [17], VeriFlow [13], and others, work by generating a logical representation of switch configurations and using an automatic solver to check properties of those configurations. The configurations are obtained by "scraping" state off of the switches or inspecting the instructions issued by an SDN controller at run-time.

These tools represent a good first step toward making networks more reliable, but they have two important limitations. First, they are based on ad hoc foundations. Although SDN platforms such as OpenFlow [21] have precise (if informal) specifications, the tools make simplifying assumptions that are routinely violated by real



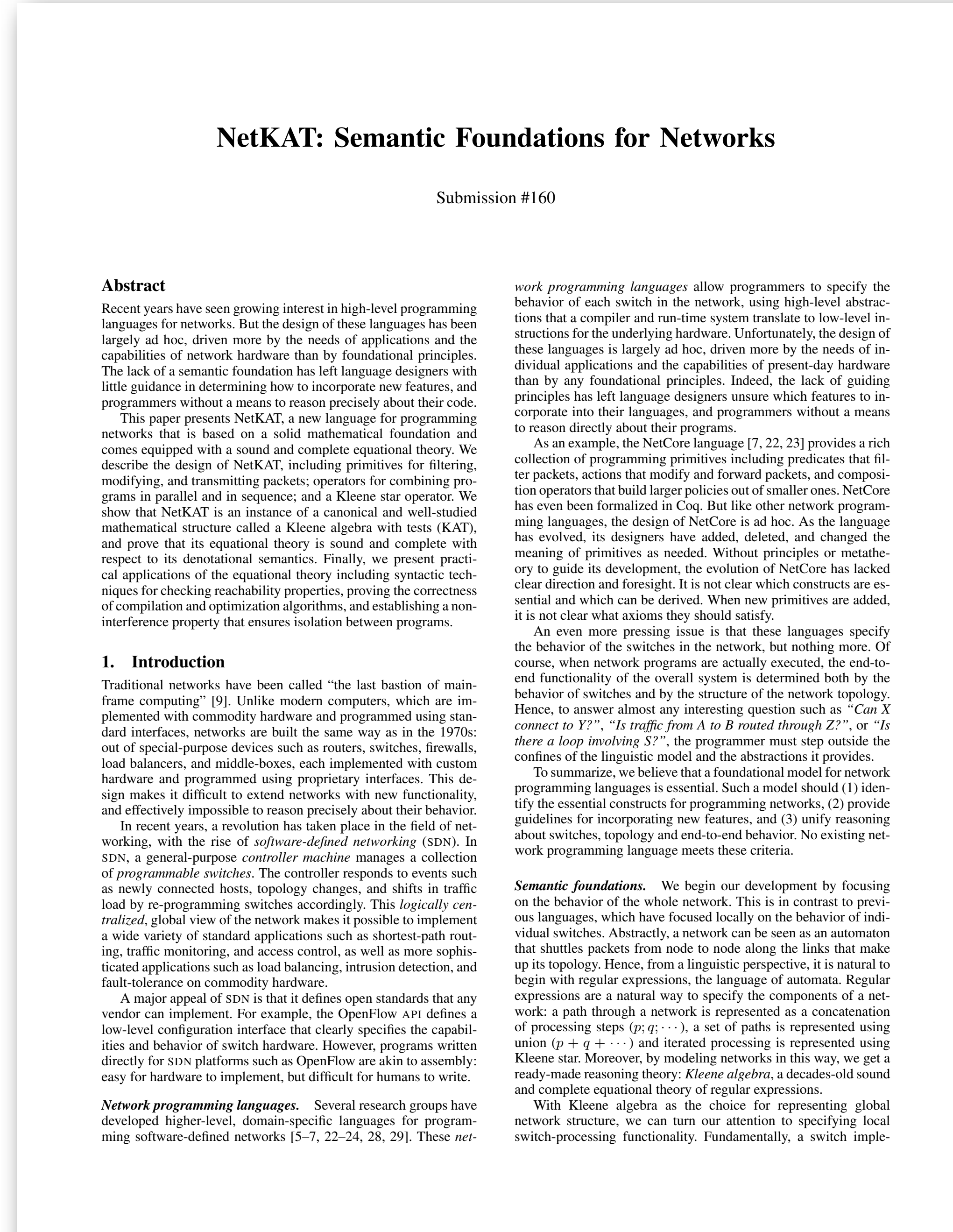
*PLDI'13,* June 16–19, 2013, Seattle, WA, USA.

# This talk is *not* about...

*NetKAT: Semantic Foundations for Networks*

- Carolyn Anderson (Swarthmore)
- Nate Foster (Cornell)
- Arjun Guha (UMass)
- Jean-Baptiste Jeannin (CMU)
- Dexter Kozen (Cornell)
- Cole Schlesinger (Princeton)
- David Walker (Princeton)

To appear in POPL '14.



**NetKAT: Semantic Foundations for Networks**

Submission #160

**Abstract**

Recent years have seen growing interest in high-level programming languages for networks. But the design of these languages has been largely ad hoc, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation has left language designers with little guidance in determining how to incorporate new features, and programmers without a means to reason precisely about their code.

This paper presents NetKAT, a new language for programming networks that is based on a solid mathematical foundation and comes equipped with a sound and complete equational theory. We describe the design of NetKAT, including primitives for filtering, modifying, and transmitting packets; operators for combining programs in parallel and in sequence; and a Kleene star operator. We show that NetKAT is an instance of a canonical and well-studied mathematical structure called a Kleene algebra with tests (KAT), and prove that its equational theory is sound and complete with respect to its denotational semantics. Finally, we present practical applications of the equational theory including syntactic techniques for checking reachability properties, proving the correctness of compilation and optimization algorithms, and establishing a non-interference property that ensures isolation between programs.

**1. Introduction**

Traditional networks have been called "the last bastion of mainframe computing" [9]. Unlike modern computers, which are implemented with commodity hardware and programmed using standard interfaces, networks are built the same way as in the 1970s: out of special-purpose devices such as routers, switches, firewalls, load balancers, and middle-boxes, each implemented with custom hardware and programmed using proprietary interfaces. This design makes it difficult to extend networks with new functionality, and effectively impossible to reason precisely about their behavior.

In recent years, a revolution has taken place in the field of networking, with the rise of *software-defined networking* (SDN). In SDN, a general-purpose *controller machine* manages a collection of *programmable switches*. The controller responds to events such as newly connected hosts, topology changes, and shifts in traffic load by re-programming switches accordingly. This *logically centralized*, global view of the network makes it possible to implement a wide variety of standard applications such as shortest-path routing, traffic monitoring, and access control, as well as more sophisticated applications such as load balancing, intrusion detection, and fault-tolerance on commodity hardware.

A major appeal of SDN is that it defines open standards that any vendor can implement. For example, the OpenFlow API defines a low-level configuration interface that clearly specifies the capabilities and behavior of switch hardware. However, programs written directly for SDN platforms such as OpenFlow are akin to assembly: easy for hardware to implement, but difficult for humans to write.

***Network programming languages.*** Several research groups have developed higher-level, domain-specific languages for programming software-defined networks [5–7, 22–24, 28, 29]. These *network programming languages* allow programmers to specify the behavior of each switch in the network, using high-level abstractions that a compiler and run-time system translate to low-level instructions for the underlying hardware. Unfortunately, the design of these languages is largely ad hoc, driven more by the needs of individual applications and the capabilities of present-day hardware than by any foundational principles. Indeed, the lack of guiding principles has left language designers unsure which features to incorporate into their languages, and programmers without a means to reason directly about their programs.

As an example, the NetCore language [7, 22, 23] provides a rich collection of programming primitives including predicates that filter packets, actions that modify and forward packets, and composition operators that build larger policies out of smaller ones. NetCore has even been formalized in Coq. But like other network programming languages, the design of NetCore is ad hoc. As the language has evolved, its designers have added, deleted, and changed the meaning of primitives as needed. Without principles or metatheory to guide its development, the evolution of NetCore has lacked clear direction and foresight. It is not clear which constructs are essential and which can be derived. When new primitives are added, it is not clear what axioms they should satisfy.

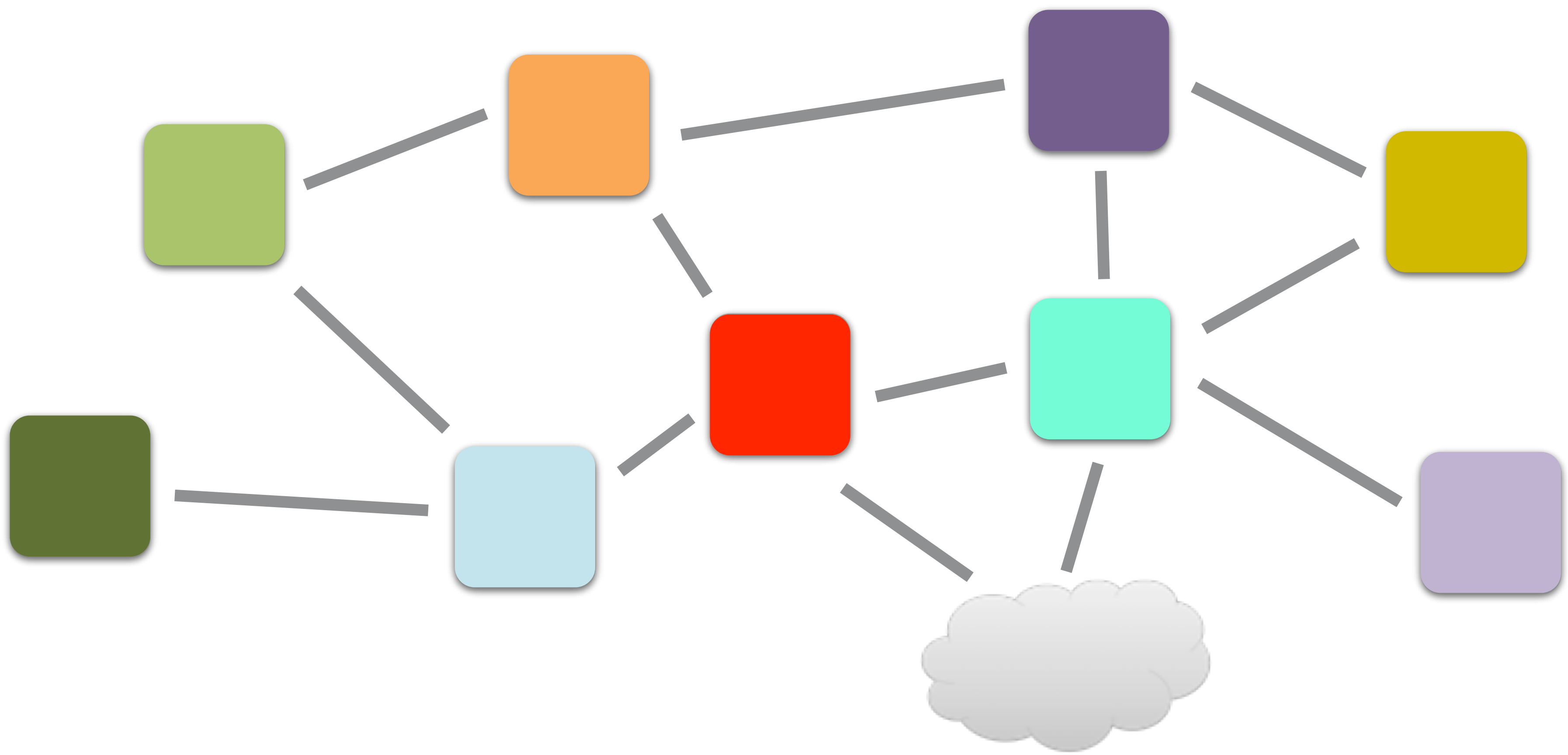
An even more pressing issue is that these languages specify the behavior of the switches in the network, but nothing more. Of course, when network programs are actually executed, the end-to-end functionality of the overall system is determined both by the behavior of switches and by the structure of the network topology. Hence, to answer almost any interesting question such as *"Can X connect to Y?"*, *"Is traffic from A to B routed through Z?"*, or *"Is there a loop involving S?"*, the programmer must step outside the confines of the linguistic model and the abstractions it provides.

To summarize, we believe that a foundational model for network programming languages is essential. Such a model should (1) identify the essential constructs for programming networks, (2) provide guidelines for incorporating new features, and (3) unify reasoning about switches, topology and end-to-end behavior. No existing network programming language meets these criteria.

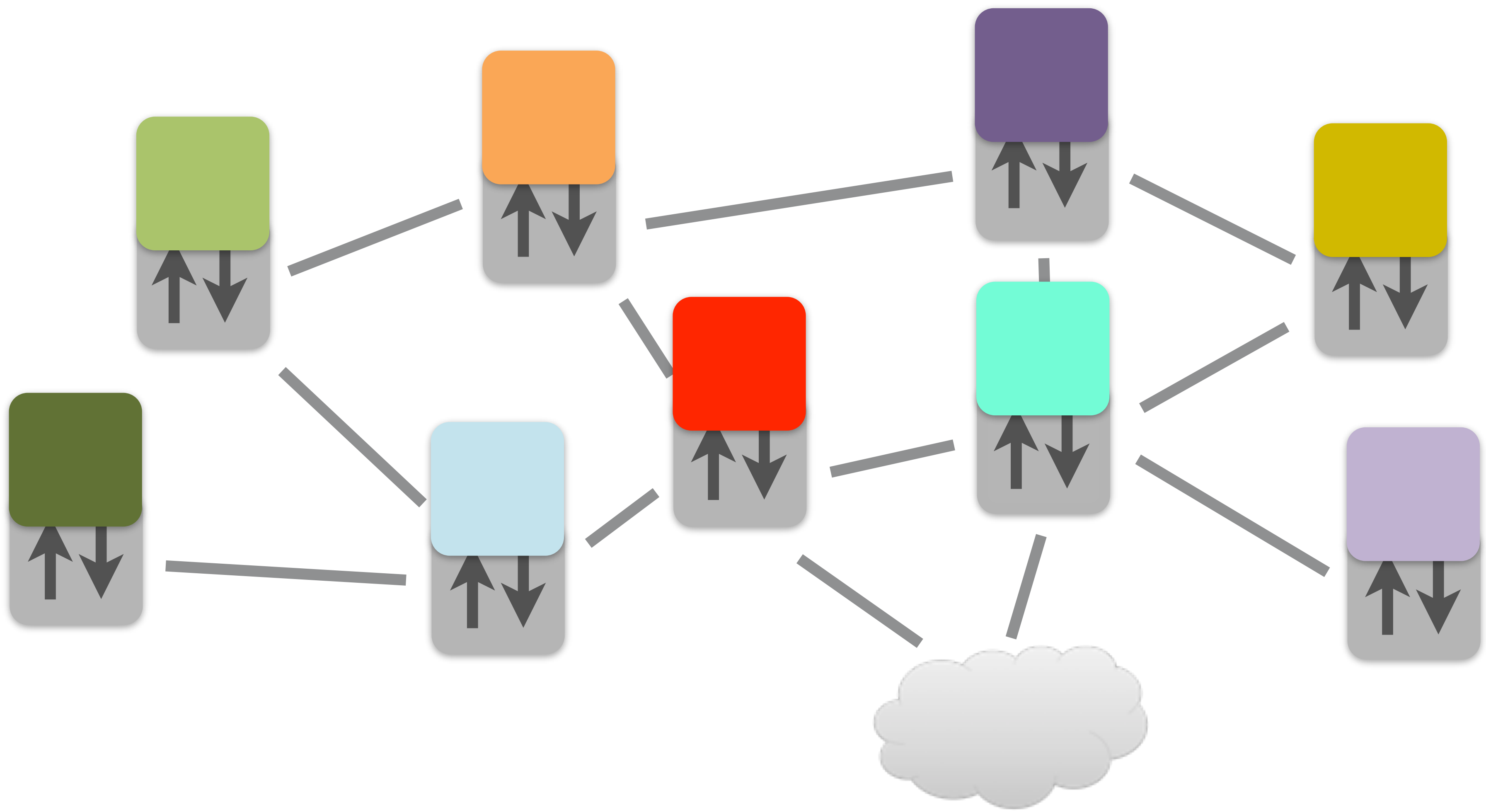
***Semantic foundations.*** We begin our development by focusing on the behavior of the whole network. This is in contrast to previous languages, which have focused locally on the behavior of individual switches. Abstractly, a network can be seen as an automaton that shuttles packets from node to node along the links that make up its topology. Hence, from a linguistic perspective, it is natural to begin with regular expressions, the language of automata. Regular expressions are a natural way to specify the components of a network: a path through a network is represented as a concatenation of processing steps ($p; q; \cdots$), a set of paths is represented using union ($p + q + \cdots$) and iterated processing is represented using Kleene star. Moreover, by modeling networks in this way, we get a ready-made reasoning theory: *Kleene algebra*, a decades-old sound and complete equational theory of regular expressions.

With Kleene algebra as the choice for representing global network structure, we can turn our attention to specifying local switch-processing functionality. Fundamentally, a switch imple-

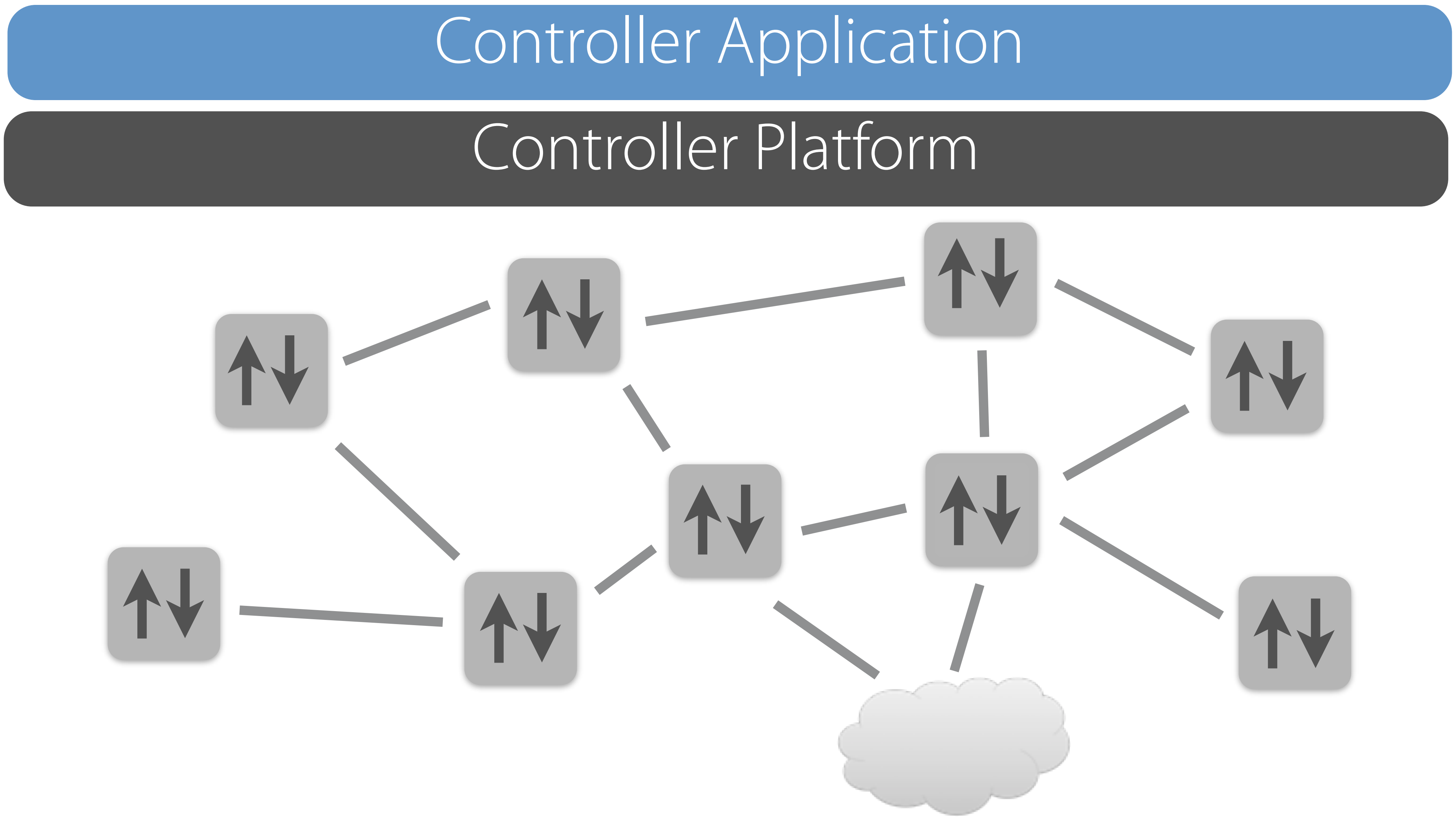
# Programmable Networks



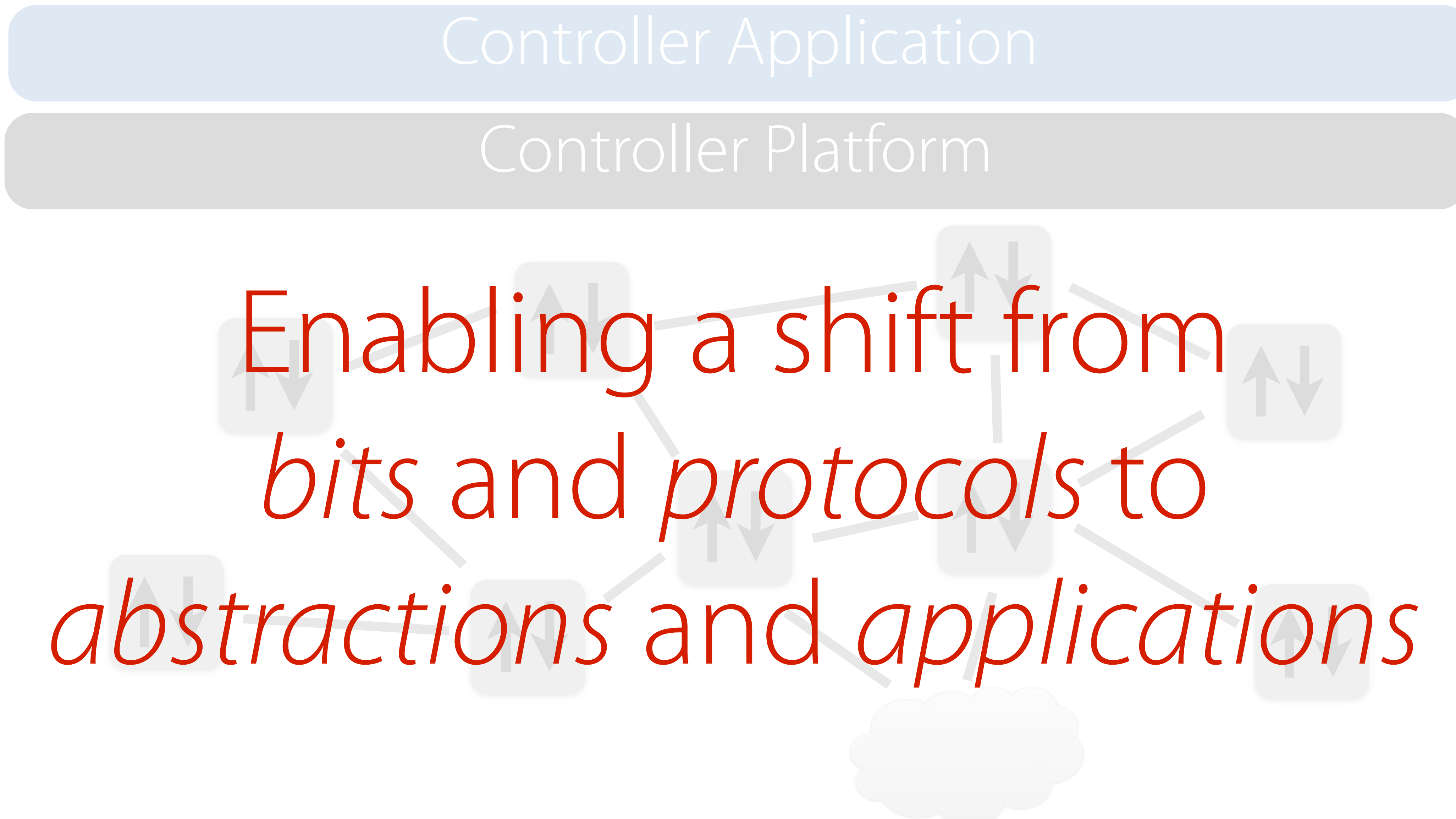
# Programmable Networks



# Programmable Networks

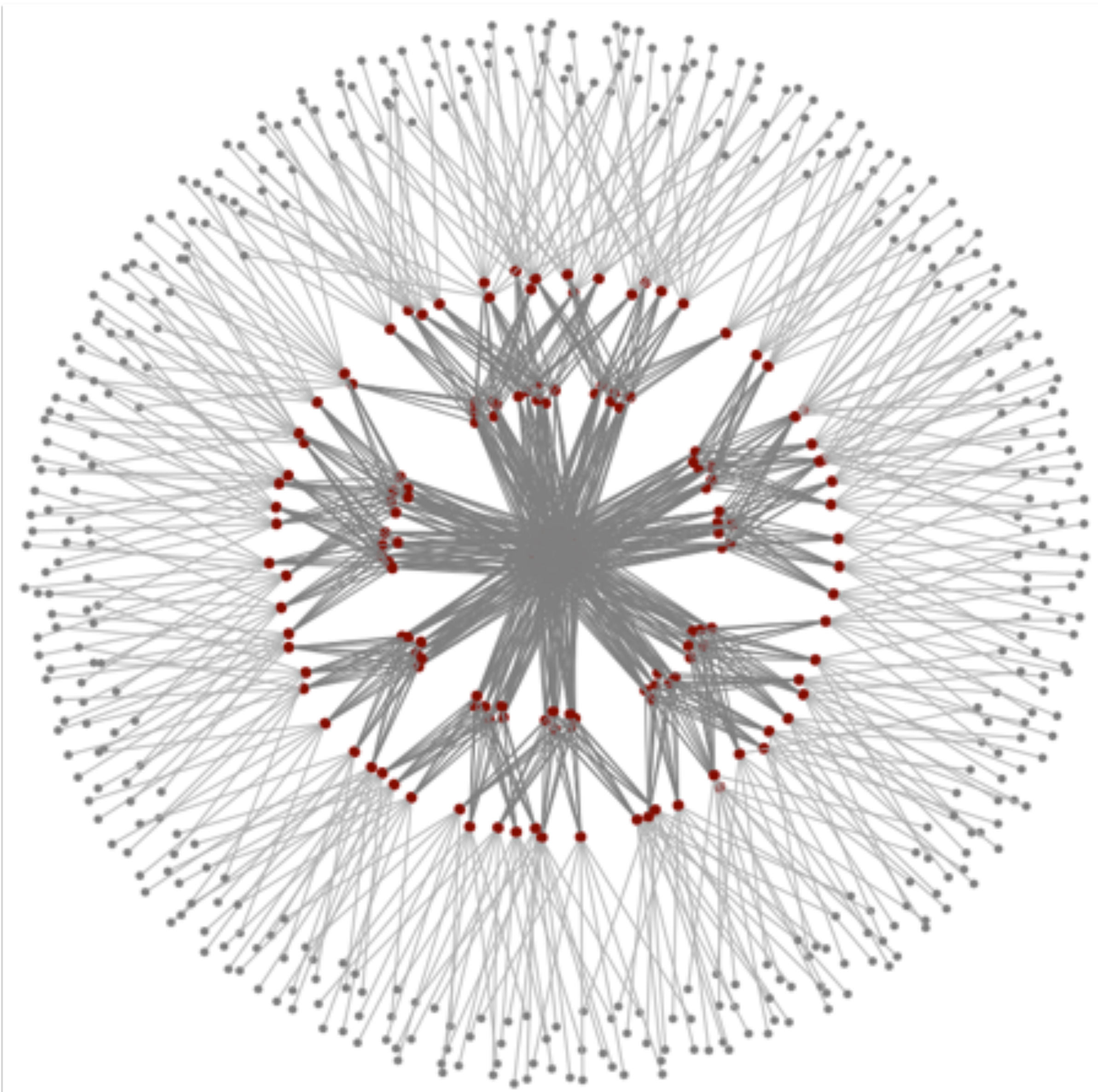


# Programmable Networks




Enabling a shift from *bits* and *protocols* to *abstractions* and *applications*

# Challenges



Thousands of nodes...

Heterogeneous devices...

Complex configurations...

Difficult to reason about...

# Current Abstractions

*Software-Defined Networks*

- Maple [SIGCOMM '13]
- Corybantic [HotNets '13]
- Frenetic [ICFP '11, POPL, 12', NSDI '13]

*Middleboxes*

- CoMB [NSDI '12]
- APLOMB [SIGCOMM '12]
- SIMPLE [SIGCOMM '13]

*End Hosts*

- PANE [SIGCOMM '13]
- EyeQ [NSDI '13]
- ETTM [NSDI '11]

# Limitations


We still lack *unified* abstractions for programming networks...

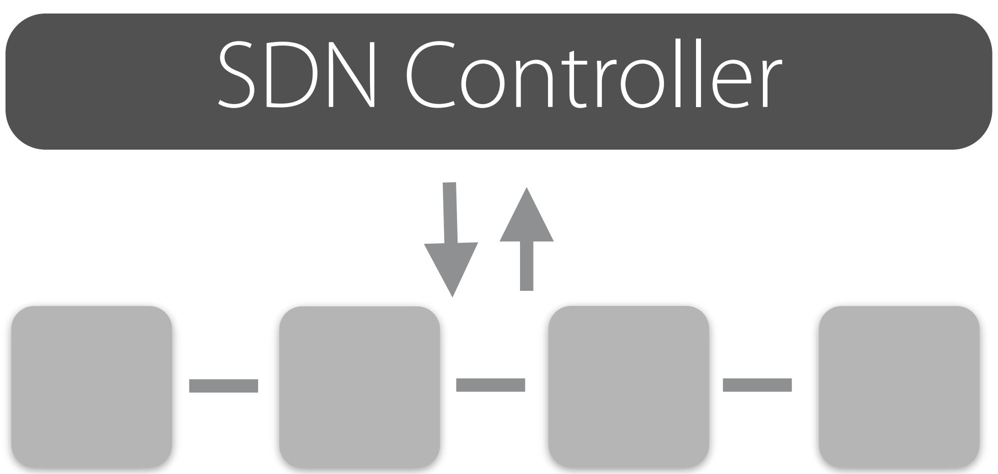
There are *complex interactions* between components...

Progress on *verification tools* is encouraging but nascent...

Most existing systems assume a *single point of control*...

# SDN Limitations

SDN is not the right abstraction for network management!




*What network operators want:*

- "Ensure that all traffic traverses at least one firewall."
- "Give Hadoop traffic priority over backup traffic"
- "Let the PPT group manage their network (in Haskell?)"

*What SDN provides:*

Match HTTP traffic and forward it out physical port 4


Policy Language

Specify global network policy in a high-level declarative language

↓

Transformations

Transform policies into ones that can be delegated or enforced locally

↓

Enforcement

Interpose on network traffic to ensure policy compliance

# Policy Language

# Formalism

$$
\begin{aligned}
loc &\in \textit{Locations} \\
t &\in \textit{Transformation Functions} \\
pol &::= (s_1; \ldots ; s_n) \\
s &::= q\; p \;\rightarrow\; e \text{ at } r \\
q &::= \text{foreach} \mid \text{forall} \mid \text{forsome} \\
p &::= m \mid p_1 \text{ and } p_2 \mid p_1 \text{ or } p_2 \mid !\, p_1 \\
m &::= h.f = n \\
e &::= . \mid c \mid e\, e \mid e | e \mid e^* \mid !\, e \\
c &::= loc \mid t \\
r &::= \text{max}(n) \mid \text{min}(n)
\end{aligned}
$$

*Syntax*

- Logical predicates
- Path expressions
- Bandwidth constraints

*Properties*

- Network paths
- Function sequences
- Resource usage

# Examples

```
( ethType = 0x800 and
  ipProto = 0x06 )
-> h1 .* nat .* dpi .* h2
 @ max(1GB/s)
```

*Informally:* ensure that all TCP traffic between two hosts is processed by NAT and DPI functions (in that order) and abides by a rate limit of 1GB/s.

# Examples

```
( ipSrc = 192.168.1.1/16 and
  ipDst = 192.168.1.1/16 and
  ipProto = 0x06 and
  ipPort = 50060 )
-> .*
 @ min(100MB/s)
```

*Informally:* ensure there is at least 100MB/s of bandwidth for Hadoop traffic.

# Examples

```
( ipSrc = 192.168.1.1/16 )
-> .* m1 .*
( !ipSrc = 192.168.1.1/16 )
-> !(.* m1 .*)
```

*Informally:* ensure resource isolation between two subnetworks for a given middlebox

# Examples

```
true
-> ( .* fire1 .* fire2 .*
   | .* fire2 .* fire1 .* )
```

*Informally:* ensure that all traffic traverses at least two firewalls, in either order.

# Examples

```
forall
true
-> ( .* mb1 .*)
 @ max(10GB/s)
```

*Informally:* ensure that all traffic across the middlebox is capped at 10GB/s.

# Flow Quantifiers

Many statements involve multiple hosts:

```
true -> (h1|h2) .* h3 @ max(100MB/s)
```

# Flow Quantifiers

Many statements involve multiple hosts:

```
true -> (h1|h2) .* h3 @ max(100MB/s)
```

Quantifiers determine devision of bandwidth:

```
  foreach
  true -> (h1|h2) .* h3 @ max(100MB/s)

≡

  true
  -> h1 .* h3 @ max(100MB/s)
  true
  -> h2 .* h3 @ max(100MB/s  )
```

# Flow Quantifiers

Many statements involve multiple hosts:

```
true -> (h1|h2) .* h3 @ max(100MB/s)
```

Quantifiers determine devision of bandwidth:

```
  forall
  true -> .* h3 @ max(100MB/s)

≡

  true
  -> h1 .* h3 @ max(50MB/s)
  true
  -> h2 .* h3 @ max(50MB/s  )
```

# Flow Quantifiers

Many statements involve multiple hosts:

```
true -> (h1|h2) .* h3 @ max(100MB/s)
```

Quantifiers determine devision of bandwidth:

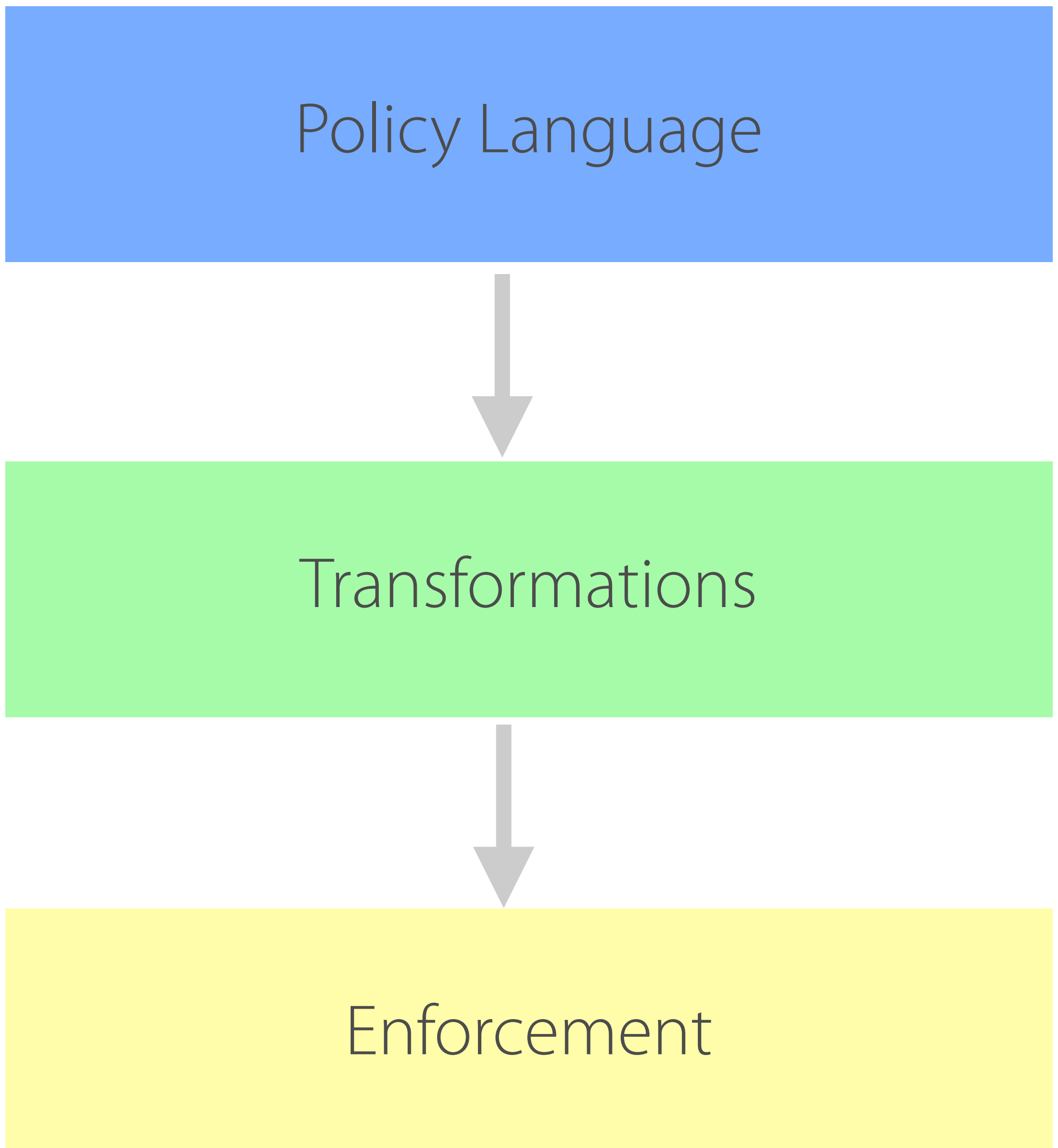
```
  forsome
  true -> (h1|h2) .* h3 @ max(100MB/s)

≡

  true
  -> (h1|h2) .* h3 @ max(100MB/s)
```

# Compiler

# Compiler




*Tasks:*

- Path selection
- Bandwidth allocation
- Code generation

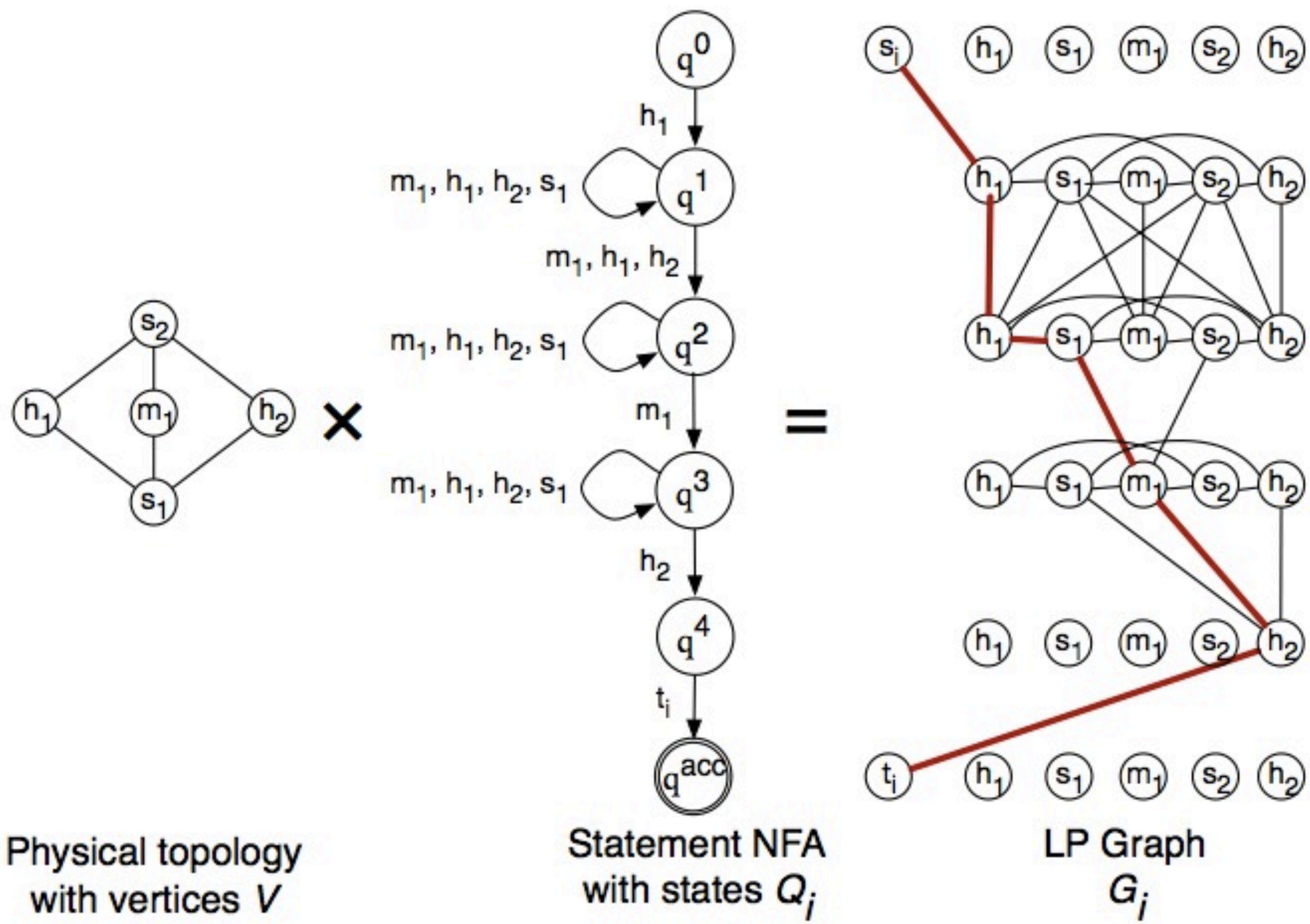
*Challenges:*

- Heterogeneous devices
- Network-wide resources

*Approach:*

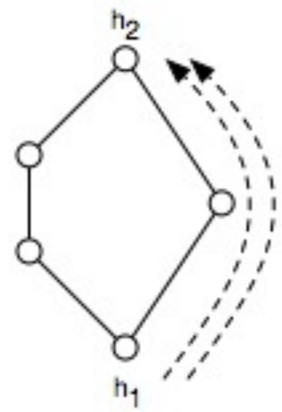
- Encode as a constraint problem
- Solve using linear programming

# Constraint Problem

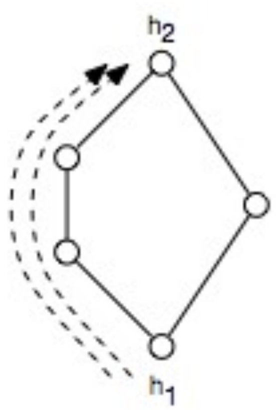



Encode with standard flow conservation and capacity constraints

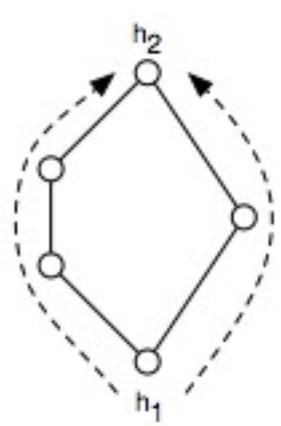
# Optimization Criteria




*Weighted Shortest Path:*
Minimizes total number of hops in assigned paths (standard)




*Min-Max Ratio:*
Minimizes the maximum fraction of reserved capacity (balance)




*Min-Max Reserved:*
Minimizes the maximum amount of reserved bandwidth (failures)

# Code Generation

*Network Switches*

- Encode paths using NetCore [POPL '12]
- Generate tags to identify paths
- Install rules on OpenFlow switches

*Middleboxes*

- Translate functions to Click [TOCS '00]
- Install on software middleboxes

*End Hosts*

- Generate code for Linux `tc` and `iptables`
- Experimental support for a custom Merlin kernel module based on `netfilter`

# Delegation

# Federated Control

*Every network has multiple tenants*

- Campuses and enterprises
- Data centers
- Wide-area

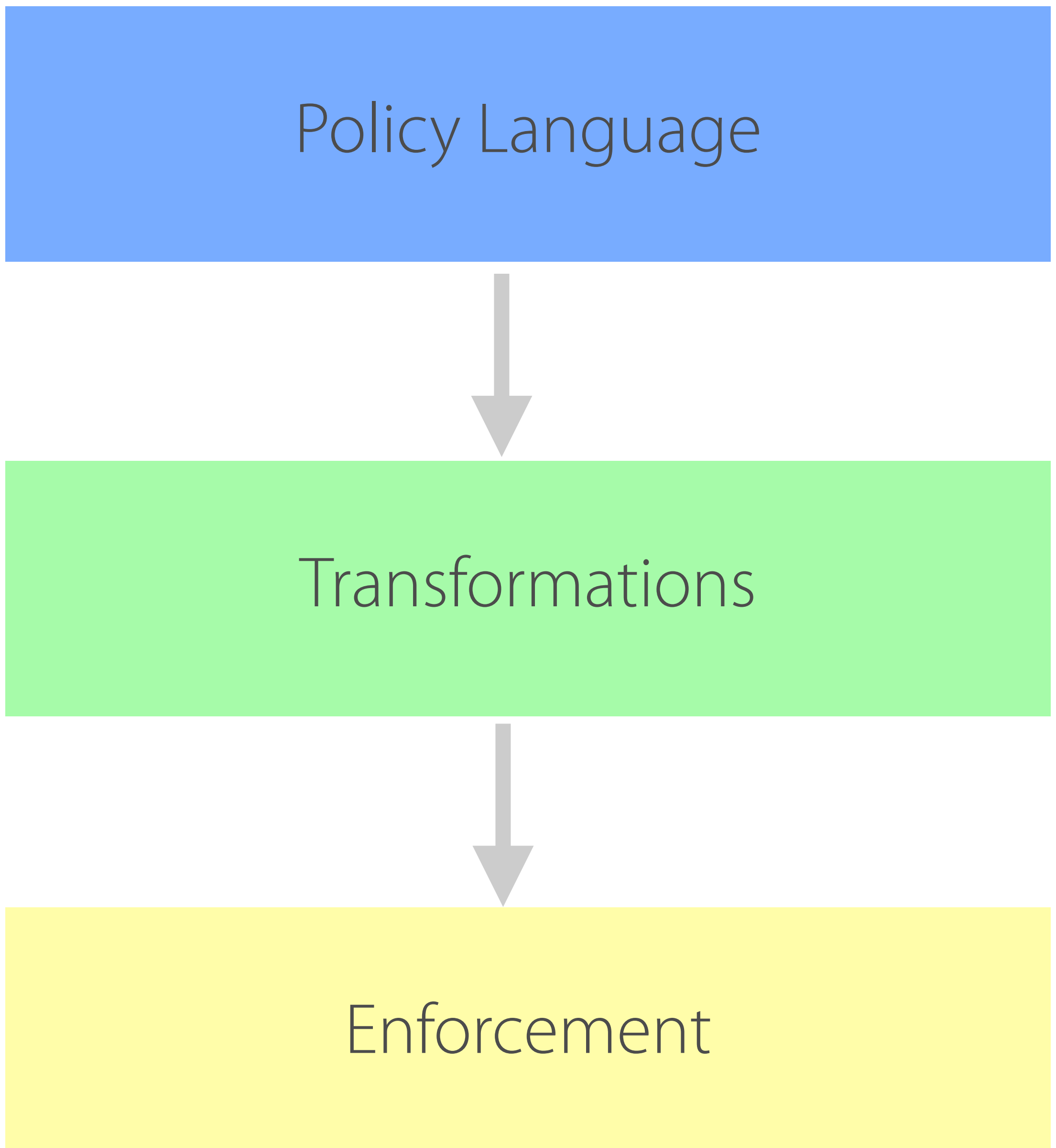
*But many platforms assume a single omnipotent programmer*

- SDN controllers
- Middlebox platforms

*Merlin provides mechanisms for*

- Delegating functionality
- Verifying policy modifications

# Compiler




*Policy restriction*

Restrict global policy to a subset of the overall traffic

*Modification:*

Tenants modify the restricted policy to suit their custom needs

*Verification:*

Owner checks that the modified policy refines the original...

*Integrate:*

...and then reintegrates the modified policy back into the global policy

# Delegation Example

Global policy

```
foreach
true
-> (h1|h2) .* h3
 @ max(100MB/s)
```

Restriction to host 1

```
foreach
true
-> h1 .* h3
 @ max(100MB/s)
```

# Modified Policy

```
foreach
(tcpDst = 80)
-> h1 .* lb .* h3
 @ max(50MB/s)

foreach
(tcpDst = 22)
-> h1 .* dpi .* h3
 @ max(25MB/s)

foreach
(!(tcpDst = 22 | tcpDst = 80))
-> h1 .* h3
 @ max(25MB/s)
```

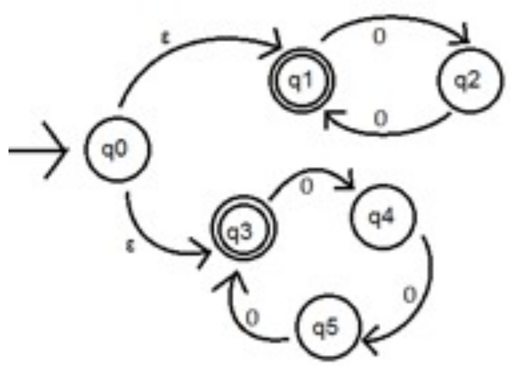
# Verification


*Essential operation:*

Policy inclusion: $P_1 \subseteq P_2$

*Algorithm:*

- Pair-wise comparison of statements
- Check for path inclusion on overlaps
- Aggregate bandwidth constraints




*Implementation:*

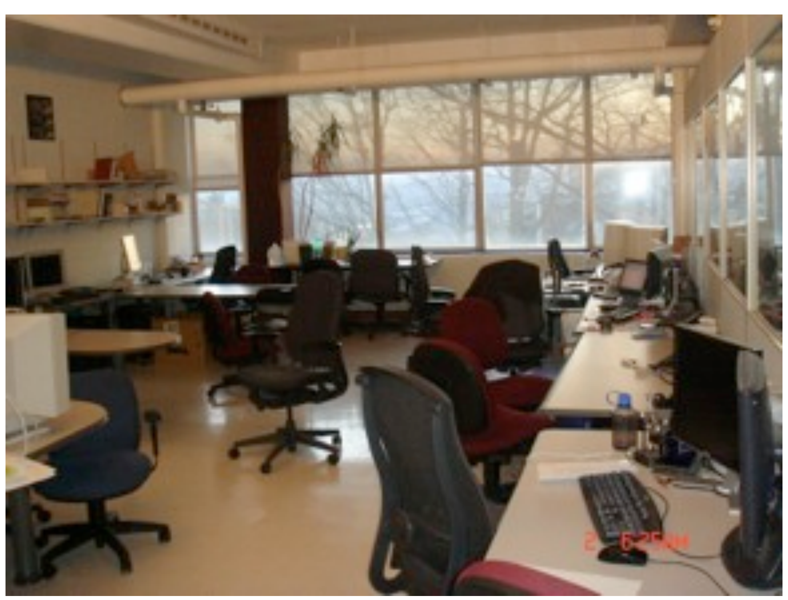
- Decide predicate overlap using SAT
- Decide path inclusion using NFAs

# Experience

# Implementation

*Prototype implementation*

- OCaml
- Gurobi solver
- Z3 theorem prover
- DPrle NFA library
- Linux kernel modules
- Frenetic language



*Preliminary deployment*

- Pronto 3290 switches
- Dell Force10 switch
- OpenVSwitch
- Dell r720 servers

# Evaluation

*Can Merlin simplify network management while providing end-to-end performance improvements for applications?*

- Hadoop benchmark
- Microbenchmarks

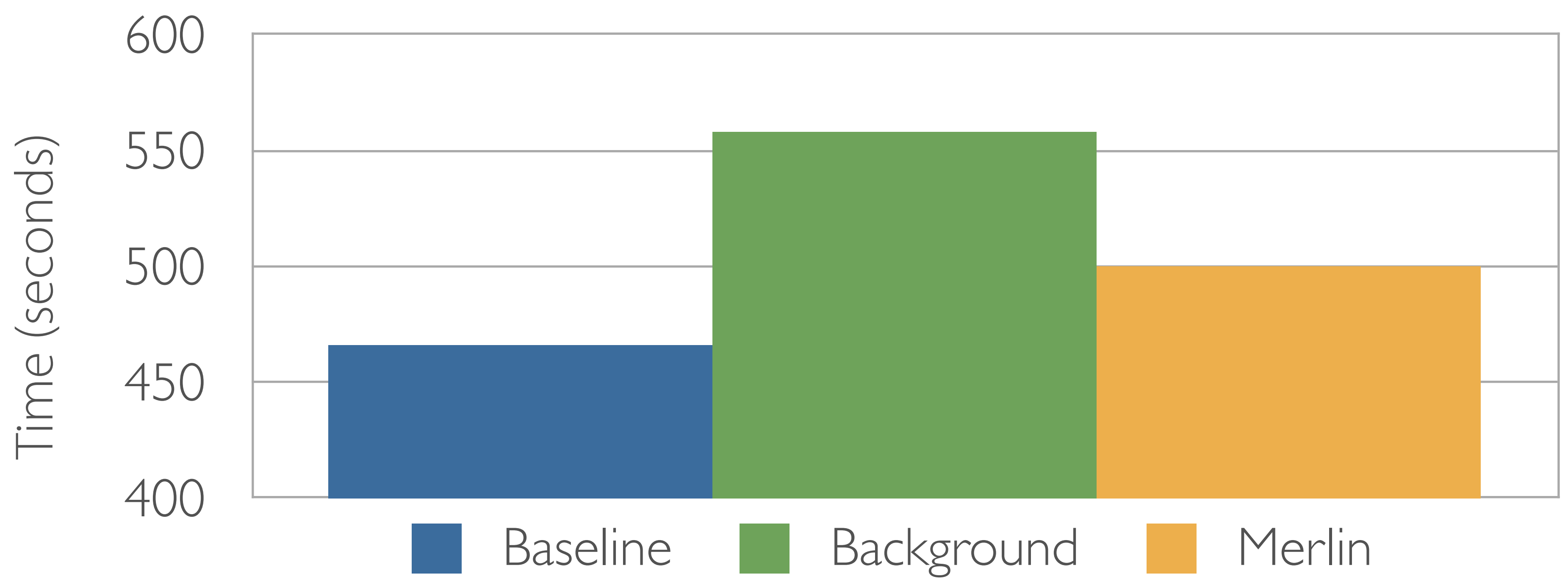
*Does our compilation and verification infrastructure scale?*

- Simple simulations on realistic data center topologies and policies of increasing size
- Data from Benson et al. [IMC '10]

# Performance Benchmark

*Experimental setup:*

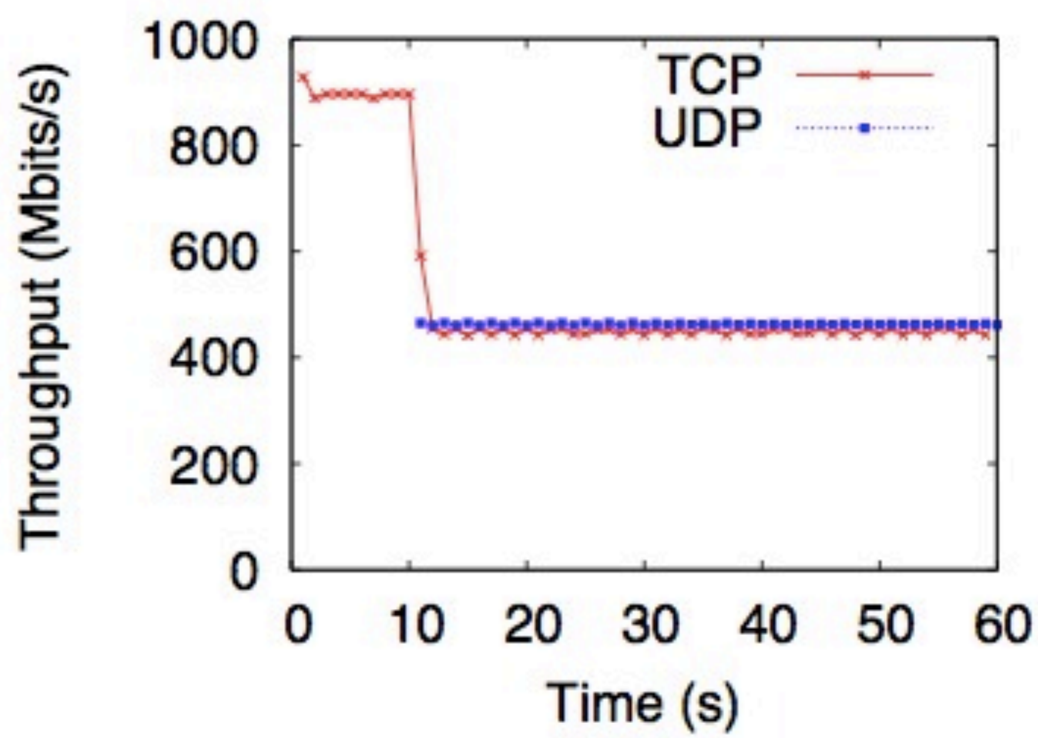
- Four 16-core Dell r720 servers with 32GB RAM
- Pronto 3290 switch with 1GB links
- Two applications: sorting and word count
- Input: 10GB data
- Traffic: UDP packets generated with **`iperf`**
- Merlin policy: reserve 90% capacity for Hadoop

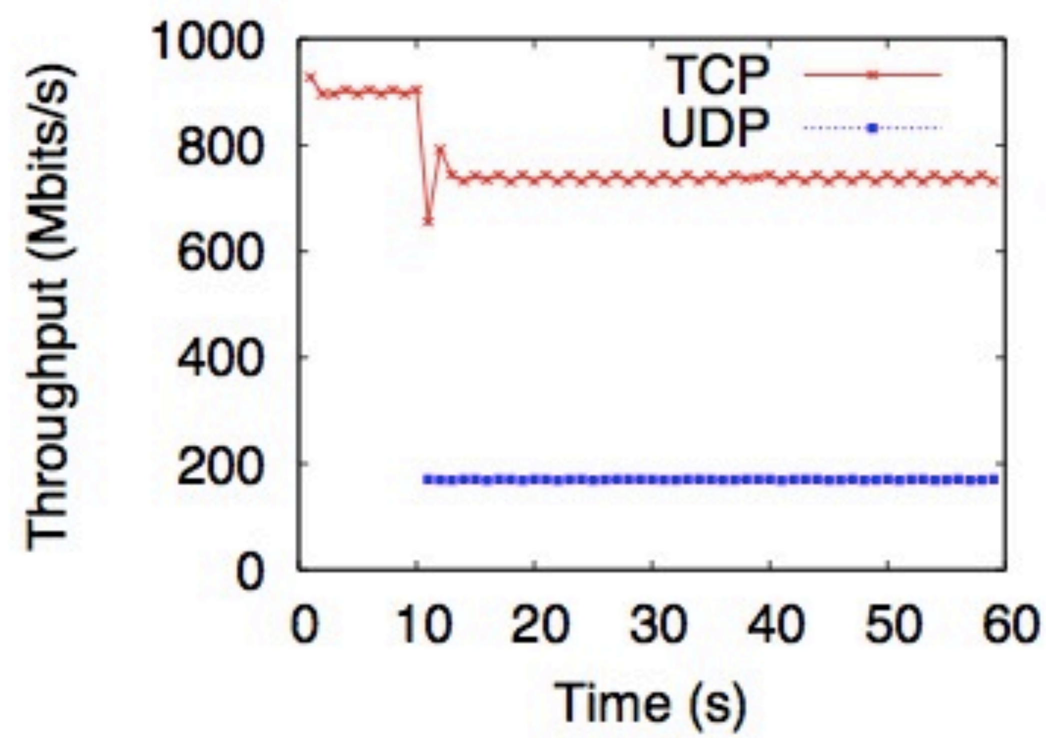


# Microbenchmark: TCP/UDP

*Experimental setup:*

- Two 16-core Dell r720 servers with 32GB RAM
- Pronto 3290 switch with 1GB links
- Traffic: TCP/UDP flows/packets generated with `iperf`
- Merlin policy: reserve 70% capacity for TCP

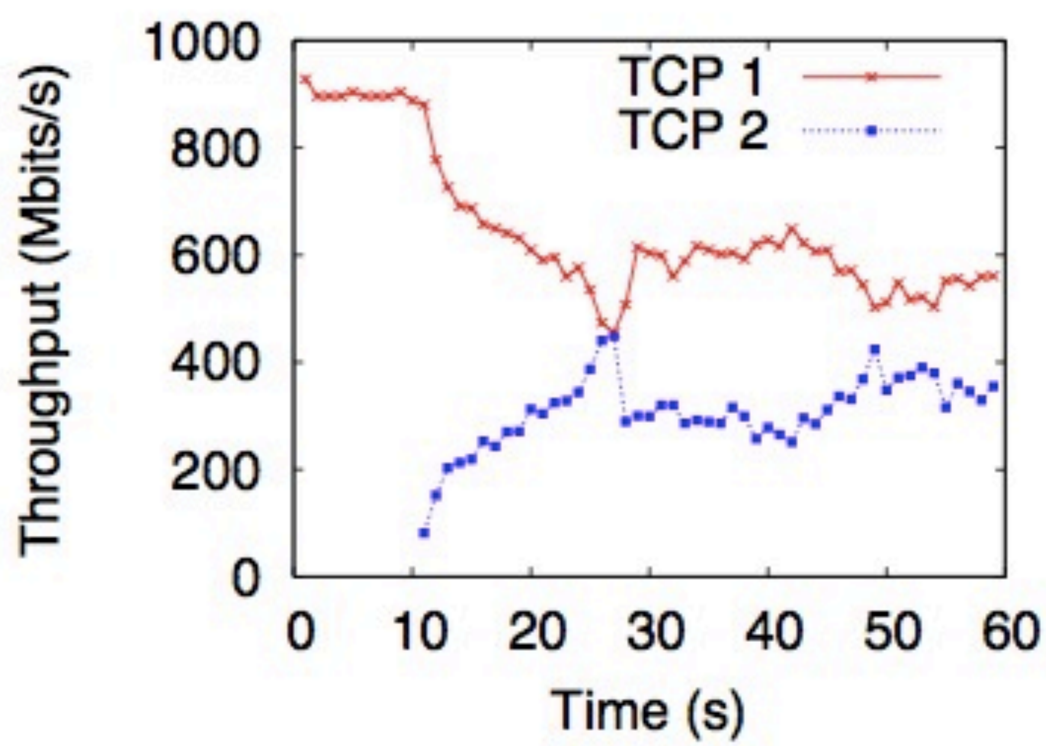


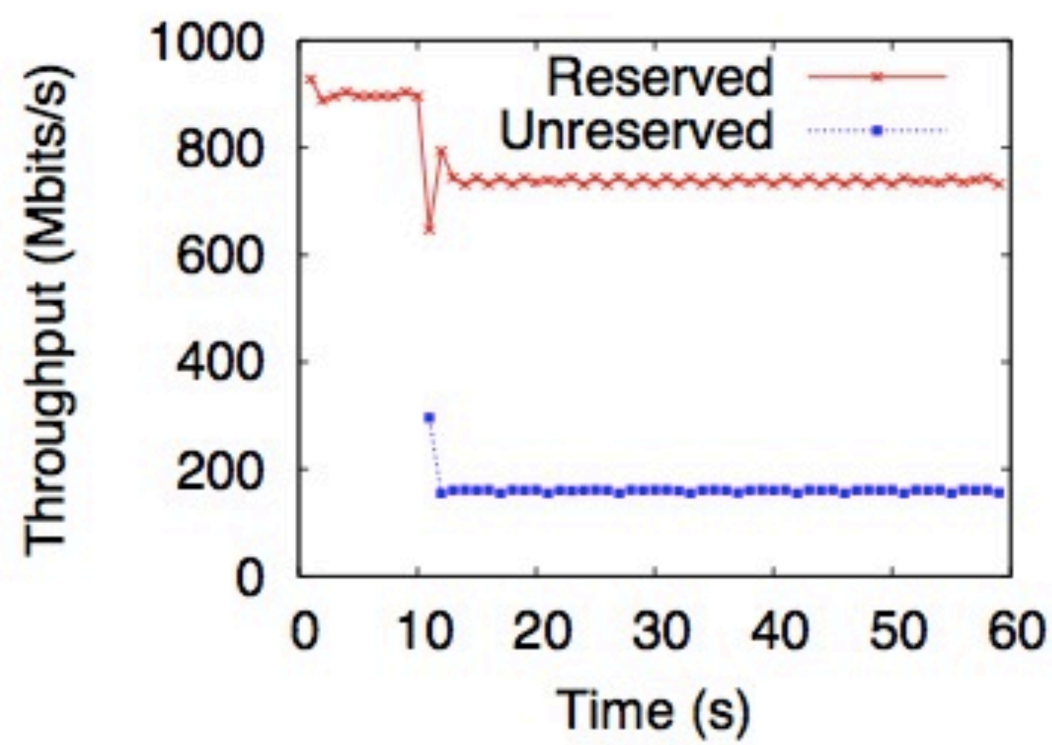



# Microbenchmark: TCP/TCP

*Experimental setup:*

- Two 16-core Dell r720 servers with 32GB RAM
- Pronto 3290 switch with 1GB links
- Traffic: TCP flows generated with **`iperf`**
- Merlin policy: reserve 70% capacity for TCP



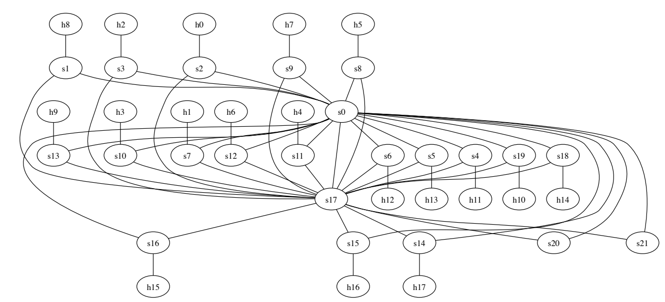



# Scalability: Compiler

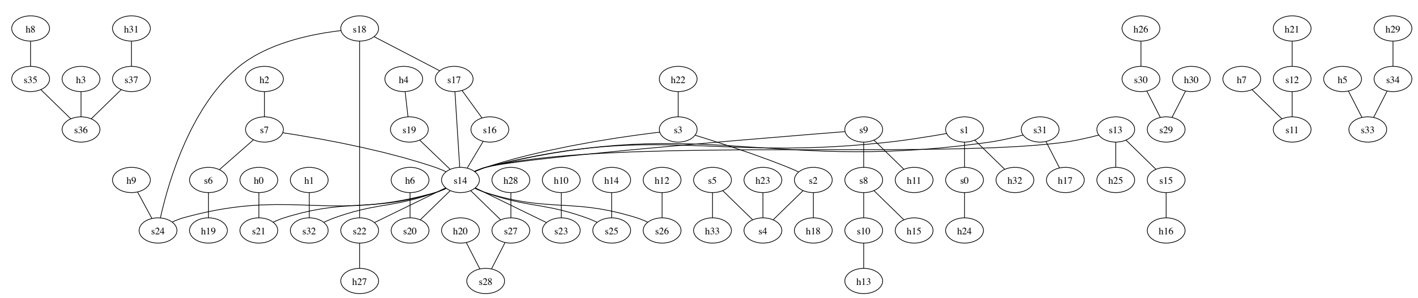
*Experimental setup:*

- 16-core Dell r720 server with 32GB RAM
- Topologies from Benson et al. [IMC ’10]
- Statements encode shortest paths between hosts

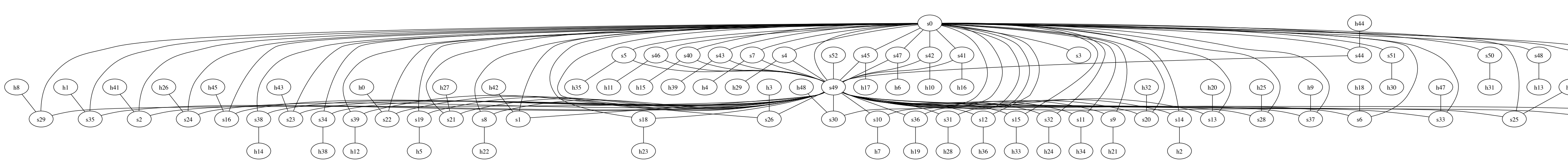
Unv1



Unv2



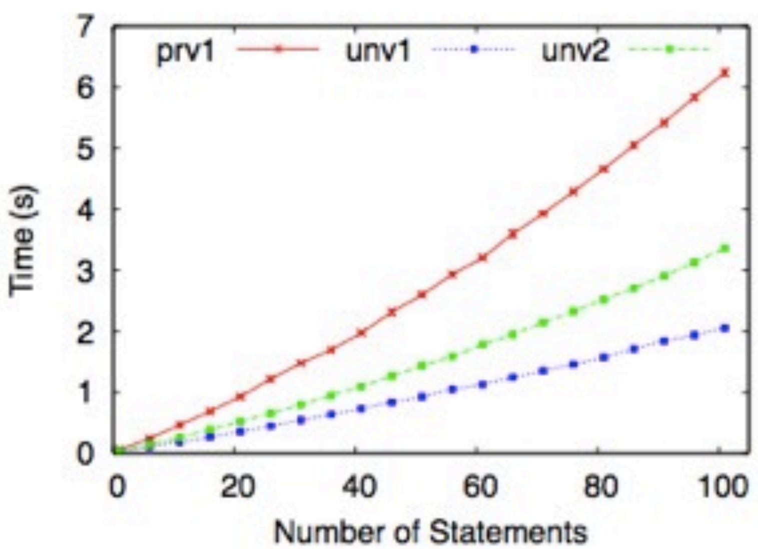
Prv



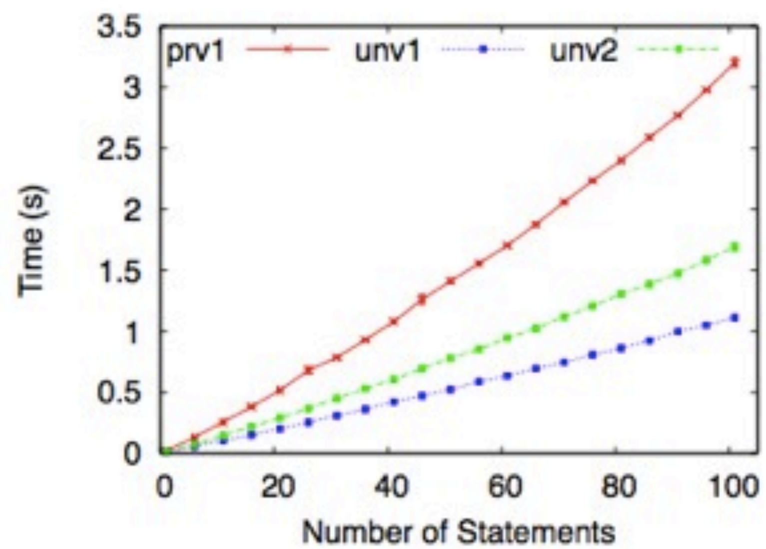
# Scalability: Compiler

*Experimental setup:*

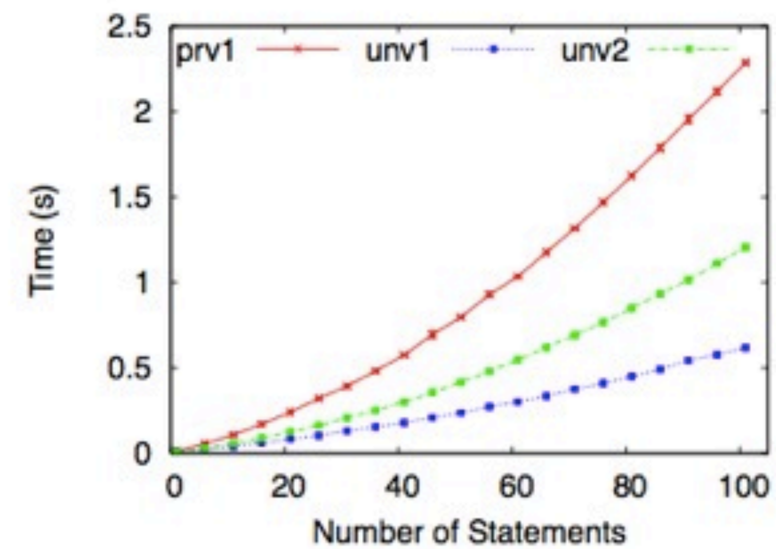
- 16-core Dell r720 server with 32GB RAM
- Topologies from Benson et al. [IMC '10]
- Statements encode shortest paths between hosts




Total




LP Generation

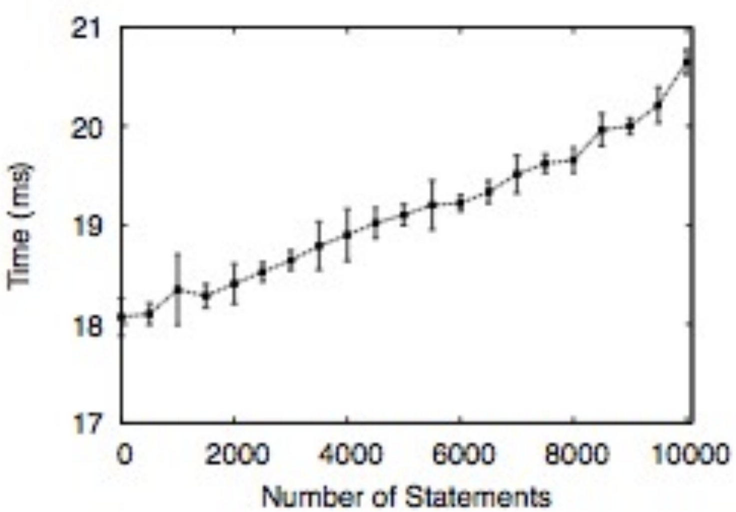



LP Solution

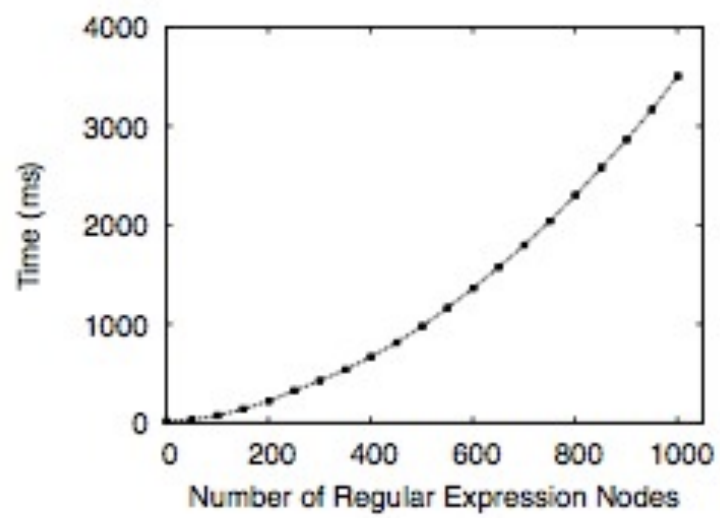
# Scalability: Verification

*Experimental setup:*

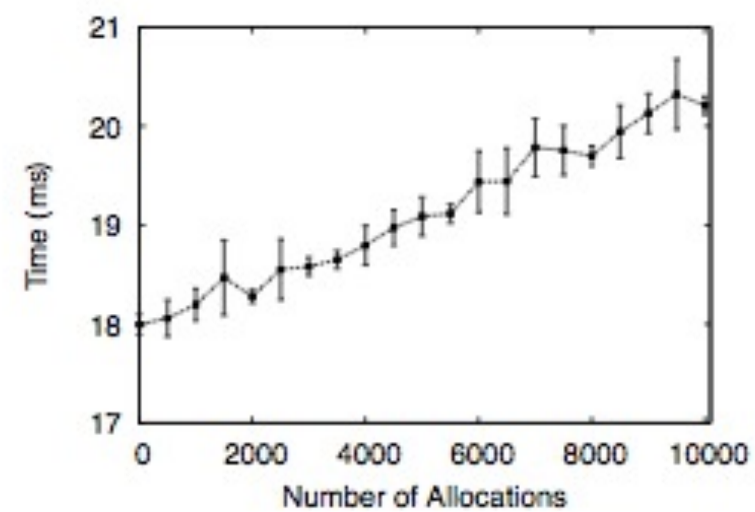
- 16-core Dell r720 server with 32GB RAM
- Topologies from Benson et al. [IMC '10]
- Statements encode shortest paths between hosts




Increasing Statements




Increasing Path Expressions




Increasing Bandwidth Constraints

# Wrapping Up

# Conclusion

Merlin is a *unified* network management framework...

It supports heterogeneous devices...

It handles paths, network functions, and bandwidth...

It generates complex configurations...

It provides delegation and automatic verification...

# Thank you!


**Collaborators**

- Robert Soulé (Postdoc)
- Shrutarshi Basu (PhD)
- Robert Kleinberg (Faculty)
- Emin Gün Sirer (Faculty)

SCHLOSS DAGSTUHL
Leibniz-Zentrum für Informatik

**Formal Foundations for Networks**

Seminar 30-0613, February 2015

Co-organizers:

- Nikolaj Bjørner (MSR)
- Nate Foster (Cornell)
- Brighten Godfrey (UIUC)
- Pamela Zave (AT&T)

flickr.com/photos/retiredinwasaga