NetKAT: Semantic Foundations for Networks

Nate Foster
Cornell University

Team NetKAT


Carolyn Anderson


Nate Foster


Arjun Guha


Jean-Baptiste Jeannin


Dexter Kozen

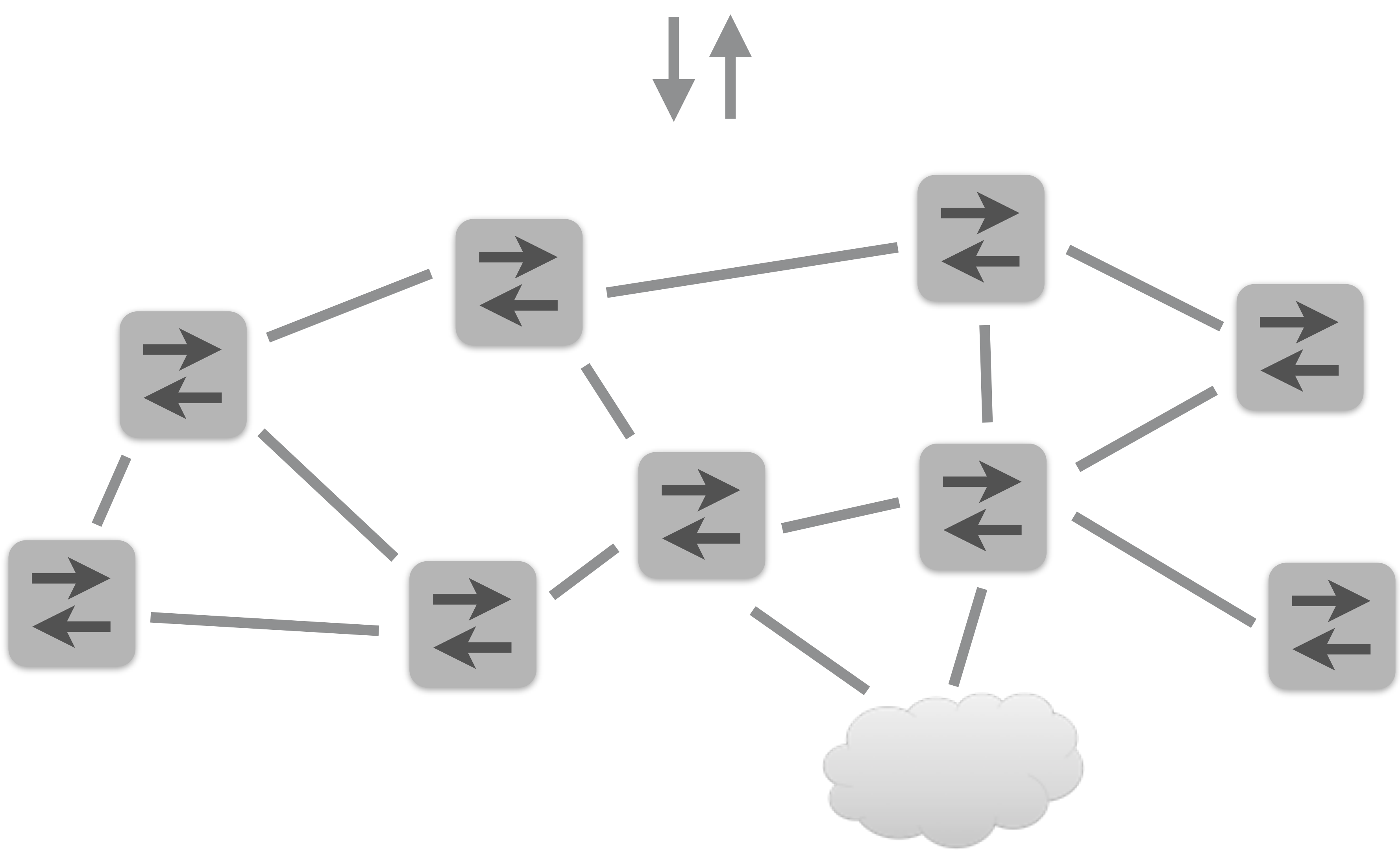

Cole Schlesinger


David Walker


Carbon

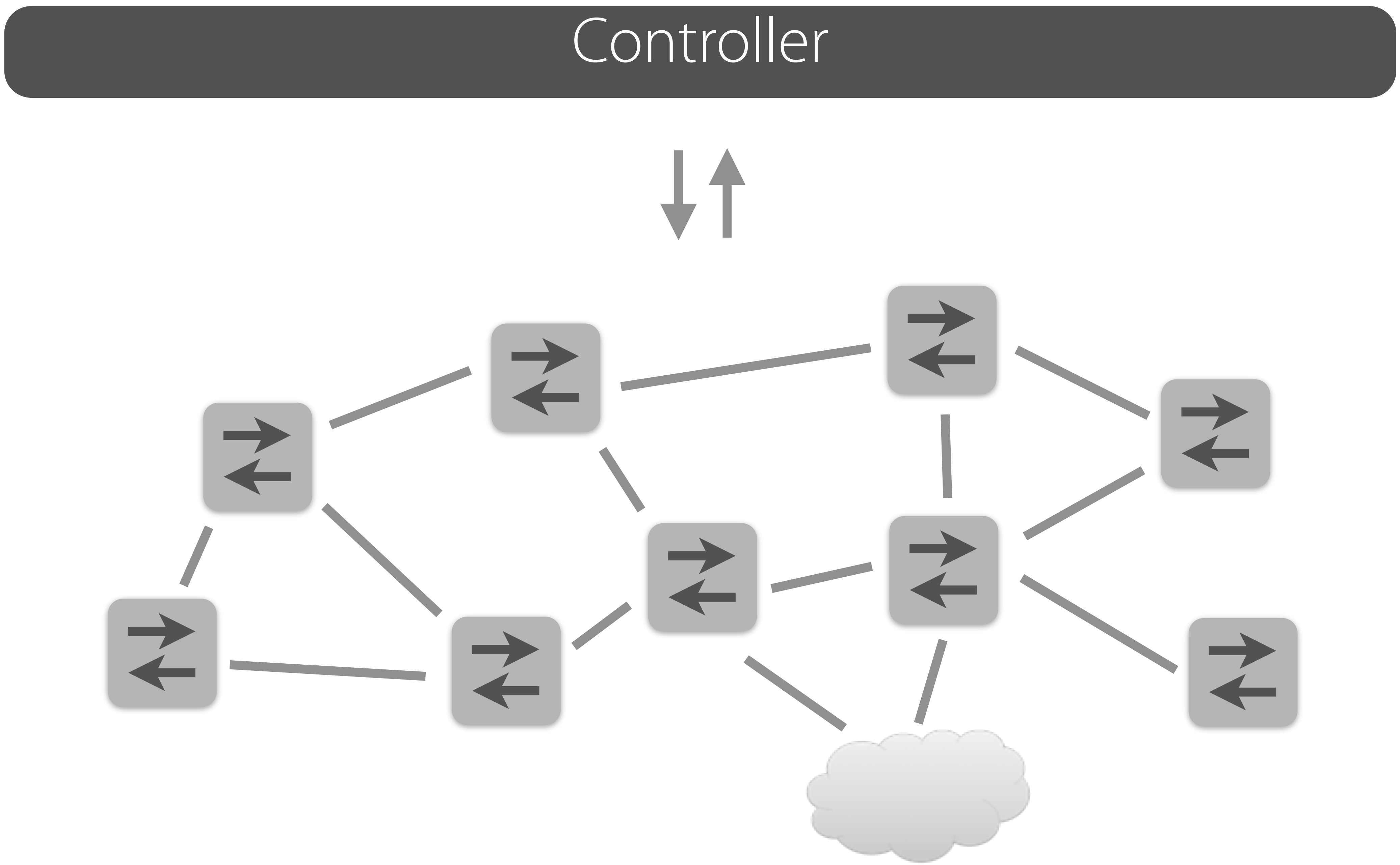
Software-Defined Networking

Key ideas: generalize devices, separate control and forwarding

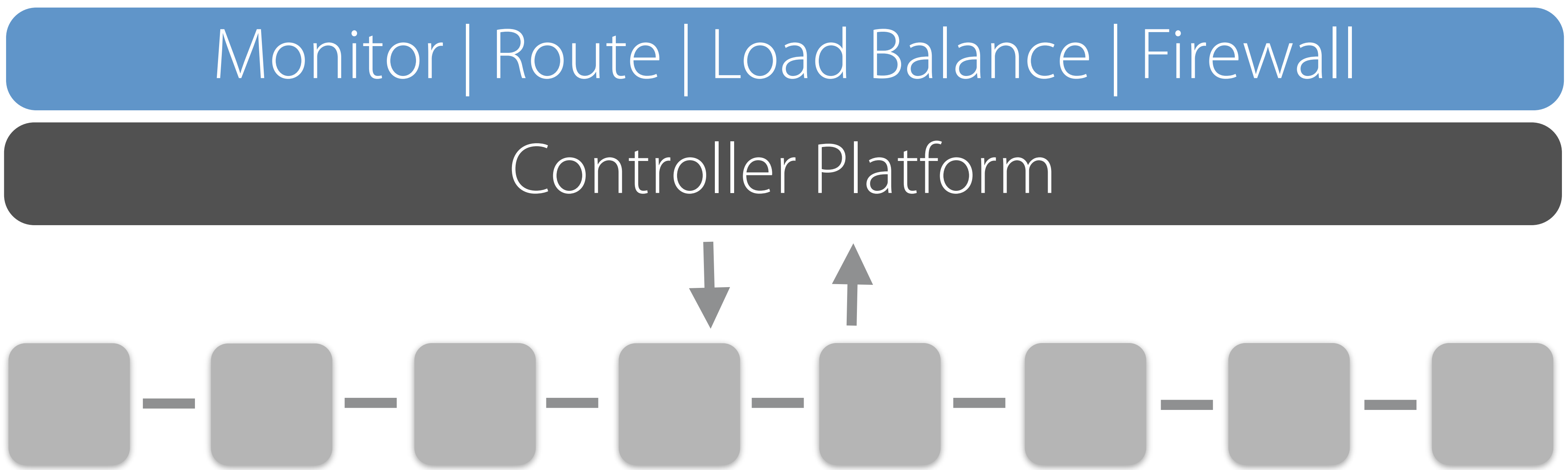


Software-Defined Networking

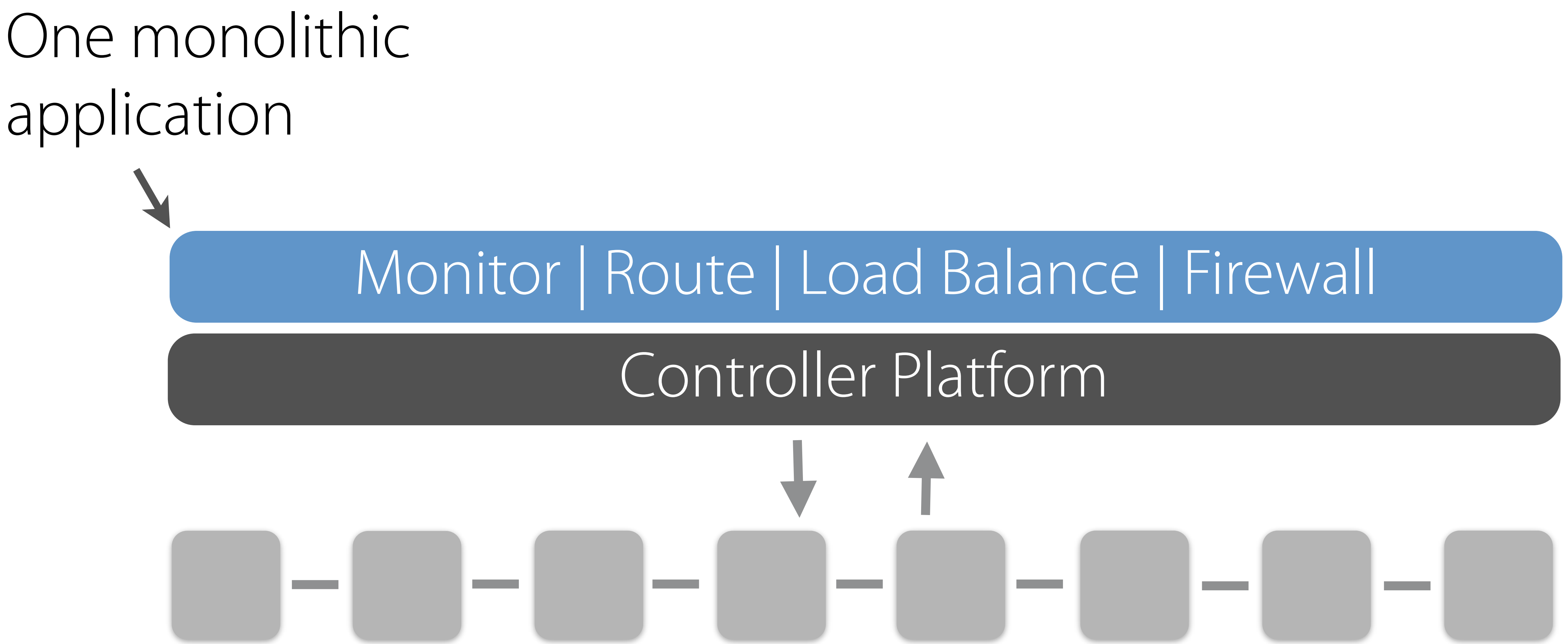
Key ideas: generalize devices, separate control and forwarding



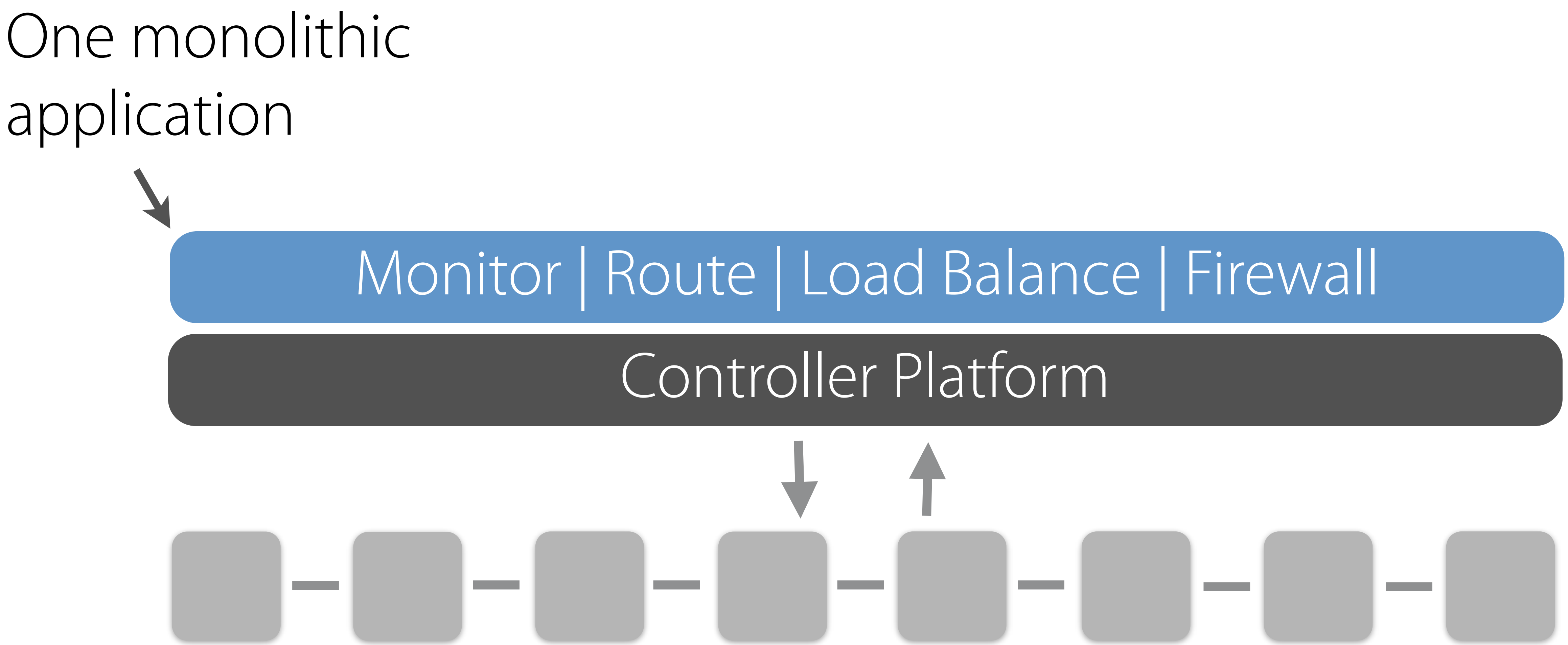
Current Controllers



Current Controllers



Current Controllers



Challenges:

- Writing, testing, and debugging programs
- Reusing code across applications
- Porting applications to new platforms

Route

Pattern	Actions
dstip=10.0.0.1	Fwd 1
dstip=10.0.0.2	Fwd 2

Monitor

Pattern	Actions
srcip=1.2.3.4	Count

Route

Pattern	Actions
dstip=10.0.0.1	Fwd 1
dstip=10.0.0.2	Fwd 2

| Monitor

Pattern	Actions
srcip=1.2.3.4	Count

Route
| Monitor

Pattern	Actions
srcip=1.2.3.4, dstip=10.0.0.1	Fwd 1, Count
srcip=1.2.3.4, dstip=10.0.0.2	Fwd 2, Count
srcip=1.2.3.4	Count
dstip=10.0.0.1	Fwd 1
dstip=10.0.0.1	Fwd 2

Route
| Monitor

Pattern	Actions
srcip=1.2.3.4, dstip=10.0.0.1	Fwd 1, Count
srcip=1.2.3.4, dstip=10.0.0.2	Fwd 2, Count
srcip=1.2.3.4	Count
dstip=10.0.0.1	Fwd 1
dstip=10.0.0.2	Fwd 2

Route
| Monitor

Pattern	Actions
srcip=1.2.3.4, dstip=10.0.0.1	Fwd 1, Count
srcip=1.2.3.4, dstip=10.0.0.2	Fwd 2, Count
srcip=1.2.3.4	Count
dstip=10.0.0.1	Fwd 1
dstip=10.0.0.2	Fwd 2

; Firewall

Pattern	Actions
tcpdst = 22	Drop
*	Fwd ?

Route
| Monitor

Pattern	Actions
srcip=1.2.3.4, dstip=10.0.0.1	Fwd 1, Count
srcip=1.2.3.4, dstip=10.0.0.2	Fwd 2, Count
srcip=1.2.3.4	Count
dstip=10.0.0.1	Fwd 1
dstip=10.0.0.2	Fwd 2

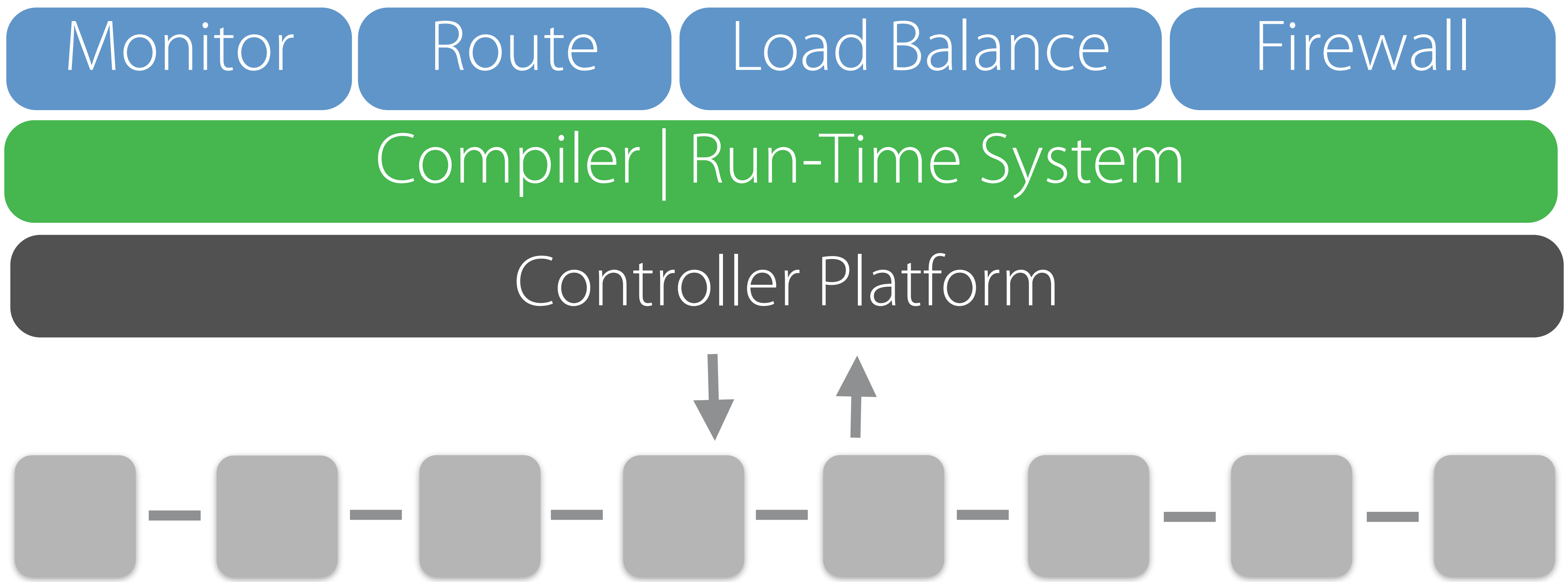
; Firewall

Pattern	Actions
tcpdst = 22	Drop
*	Fwd ?

(Route
| Monitor)
; Firewall

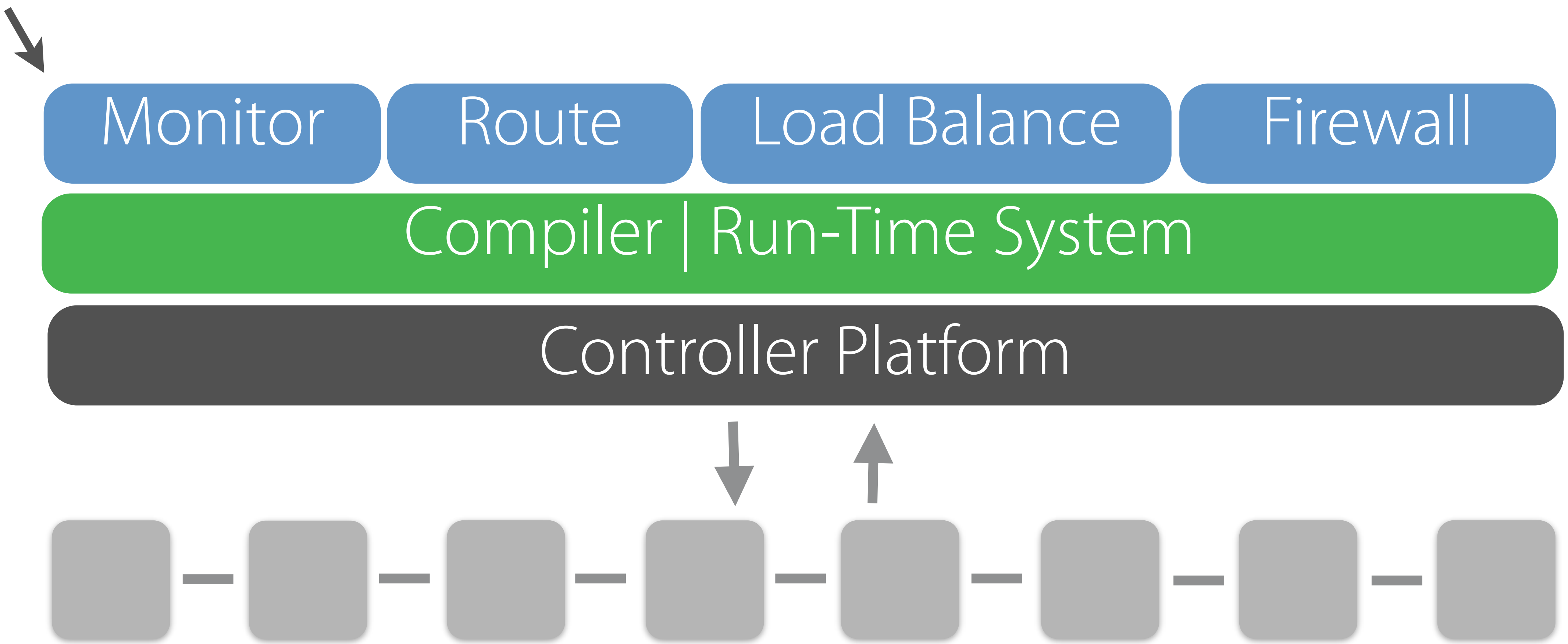
Pattern	Actions
srcip=1.2.3.4, tcpdst = 22	Count, Drop
srcip=1.2.3.4, dstip=10.0.0.1	Fwd 1, Count
srcip=1.2.3.4, dstip=10.0.0.2	Fwd 2, Count
srcip=1.2.3.4	Count
tcpdst = 22	Drop
dstip=10.0.0.1	Fwd 1
dstip=10.0.0.2	Fwd 2

Language-Based Approach



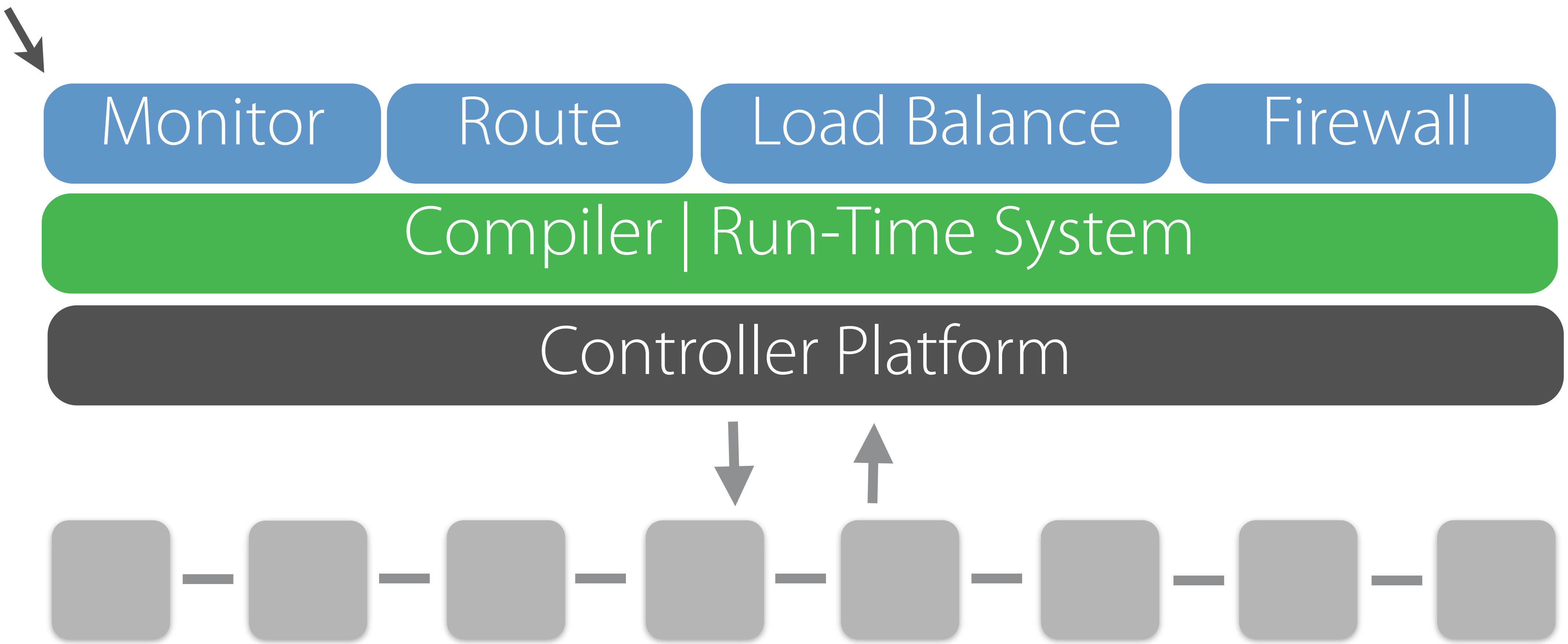
Language-Based Approach

One module
for each task



Language-Based Approach

One module
for each task



Benefits:

- Easier to write, test, and debug programs
- Can reuse modules across applications
- Possible to port applications to new platforms

Language I

[ICFP '11]

Key ideas:

- A language abstraction between programs and hardware
- Constructs for reading state and specifying forwarding policies
- Support for modular composition through policy combinators
- Run-time system pushes rules to switches reactively

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

Modern networks provide a variety of interrelated services including routing, traffic monitoring, load balancing, and access control. Unfortunately, the languages used to program today's networks lack modern features—they are usually defined at the low level of abstraction supplied by the underlying hardware and they fail to provide even rudimentary support for modular programming. As a result, network programs tend to be complicated, error-prone, and difficult to maintain.

This paper presents Frenetic, a high-level language for programming distributed collections of network switches. Frenetic provides a declarative query language for classifying and aggregating network traffic as well as a functional reactive combinator library for describing high-level packet-forwarding policies. Unlike prior work in this domain, these constructs are—by design—fully compositional, which facilitates modular reasoning and enables code reuse. This important property is enabled by Frenetic's novel run-time system which manages all of the details related to installing, uninstalling, and querying low-level packet-processing rules on physical switches.

Overall, this paper makes three main contributions: (1) We analyze the state-of-the art in languages for programming networks and identify the key limitations; (2) We present a language design that addresses these limitations, using a series of examples to motivate and validate our choices; (3) We describe an implementation of the language and evaluate its performance on several benchmarks.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

General Terms Languages, Design

Keywords Network programming languages, domain-specific languages, functional reactive programming, OpenFlow

1. Introduction

Today's networks consist of hardware and software components that are closed and proprietary. The difficulty of changing these components has had a chilling effect on innovation, and forced network administrators to express policies through complicated and frustratingly brittle interfaces. As discussed in recent a *New York Times* article [30], the rise of data centers and cloud computing have brought these problems into sharp relief and led a number of networks researchers to reconsider the fundamental assumptions that underpin today's network architectures.

In particular, significant momentum has gathered behind OpenFlow, a new platform that opens up the software that controls the network while also allowing packets to be processed using fast, commodity switching hardware [31]. OpenFlow defines a standard interface for installing flexible packet-forwarding rules on physical network switches using a programmable *controller* that runs separately on a stock machine. The most well-known controller platform is NOX [20], though there are several others [1, 8, 25, 39]. OpenFlow is supported by a number of commercial Ethernet switch vendors, and has been deployed in several campus and backbone networks. Using OpenFlow, researchers have already created a variety of controller applications that introduce new network functionality, like flexible access control [9, 33], Web server load balancing [21, 40], energy-efficient networking [22], and seamless virtual-machine migration [18].

Unfortunately, while OpenFlow and NOX now make it *possible* to implement exciting new network services, they do not make it *easy*. OpenFlow programmers must constantly grapple with several difficult challenges.

First, networks often perform multiple tasks, like routing, access control, and traffic monitoring. Unfortunately, decoupling these tasks from each other and implementing them independently in separate modules is effectively impossible, since packet-handling rules (un)installed by one module often interfere with overlapping rules (un)installed by other modules.

Second, the OpenFlow/NOX interface is defined at a very low level of abstraction. For example, the OpenFlow rule algebra directly reflects the capabilities of the switch hardware (*e.g.*, bit patterns and integer priorities). Simple high-level concepts such as set difference require multiple rules and priorities to implement correctly and more powerful "wildcard" rules are a limited hardware resource that programmers must manage by hand.

Third, controller programs only receive events for packets the switches do not know how to handle. Code that installs a forwarding rule might prevent another, different event-driven call-back from being triggered. As a result, writing programs for OpenFlow/NOX quickly becomes a difficult exercise in *two-tiered* programming—programmers must simultaneously reason about the packets that will processed on switches and those that will be processed on the controller.

Fourth, because a network of switches is a distributed system, it is susceptible to various kinds of race conditions. For example, a common NOX programming idiom is to handle the first packet of each network flow on the controller and install switch-level rules to handle the remaining packets. However, such programs can be susceptible to errors if the second, third, or fourth packets in a

Language II [POPL '12]

Key ideas:

- NetCore policy language

```
e ::= h:w
    | switch s
    | inspect e f
    | e1 & e2
    | !e1
S ::= { s1, ..., sk }
t ::= e => S
    | t1 & t1
    | !t
```

- Compiler pushes forwarding rules to switches proactively
- Reactive specialization handles features that cannot be translated

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

Software-defined networks (SDNs) are a new kind of network architecture in which a controller machine manages a distributed collection of switches by instructing them to install or uninstall packet-forwarding rules and report traffic statistics. The recently formed Open Networking Consortium, whose members include Google, Facebook, Microsoft, Verizon, and others, hopes to use this architecture to transform the way that enterprise and data center networks are implemented.

In this paper, we define a high-level, declarative language, called *NetCore*, for expressing packet-forwarding policies on SDNs. NetCore is expressive, compositional, and has a formal semantics. To ensure that a majority of packets are processed efficiently on switches—instead of on the controller—we present new compilation algorithms for NetCore and couple them with a new run-time system that issues rule installation commands and traffic-statistics queries to switches. Together, the compiler and run-time system generate efficient rules whenever possible and outperform the simple, manual techniques commonly used to program SDNs today. In addition, the algorithms we develop are generic, assuming only that the packet-matching capabilities available on switches satisfy some basic algebraic laws.

Overall, this paper delivers a new design for a high-level network programming language; an improved set of compiler algorithms; a new run-time system for SDN architectures; the first formal semantics and proofs of correctness in this domain; and an implementation and evaluation that demonstrates the performance benefits over traditional manual techniques.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

General Terms Languages, Design

Keywords Software-defined Networking, OpenFlow, Frenetic, Network programming languages, Domain specific languages

* The views expressed in this paper are those of the authors and do not reflect the official policy or position of the US Military Academy, the Department of the Army, the Department of Defense, or the US Government.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1083-3/12/01…$10.00

1. Introduction

A network is a collection of connected devices that route traffic from one place to another. Networks are pervasive: they connect students and faculty on university campuses, they send packets between a variety of mobile devices in modern households, they route search requests and shopping orders through data centers, they tunnel between corporate networks in San Francisco and Helsinki, and they connect the steering wheel to the drive train in your car. Naturally, these networks have different purposes, properties, and requirements. To service these requirements, companies like Cisco, Juniper, and others manufacture a variety of devices including routers (which forward packets based on IP addresses), switches (which forward packets based on MAC addresses), NAT boxes (which translate addresses within a network), firewalls (which squelch forbidden or unwanted traffic), and load balancers (which distribute work among servers), to name a few.

While each of these devices behaves differently, internally they are all built on top of a *data plane* that buffers, forwards, drops, tags, rate limits, and collects statistics about packets at high speed. More complicated devices like routers also have a *control plane* that run algorithms for tracking the topology of the network and computing routes through it. Using statistics gathered from the data plane and the results computed using the device's specialized algorithms, the control plane installs or uninstalls forwarding rules in the data plane. The data plane is built out of fast, special-purpose hardware, capable of forwarding packets at the rate at which they arrive, while the control plane is typically implemented in software.

Remarkably, however, traditional networks appear to be on the verge of a major upheaval. On March 11th, 2011, Deutsche Telekom, Facebook, Google, Microsoft, Verizon, and Yahoo!, owners of some of the largest networks in the world, announced the formation of the Open Networking Foundation [19]. The foundation's proposal is extraordinarily simple: *eliminate the control plane from network devices*. Instead of baking specific control software into each device, the foundation proposes a standard protocol that a separate, general-purpose machine called a *controller* can use to program and query the data planes of many cooperating devices. By moving the control plane from special-purpose devices onto stock machines, companies like Google will be able to buy cheap, commodity switches, and write controller programs to customize and optimize their networks however they choose.

Networks built on this new architecture, which arose from earlier work on Ethane [4] and 4D [10], are now commonly referred to as *Software-Defined Networks* (SDNs). Already, several commercial switch vendors support OpenFlow [17], a concrete realization of the switch-controller protocol required for implementing SDNs, and researchers have used OpenFlow to develop new network-wide algorithms for server load-balancing, data center routing, energy-efficient network management, virtualization, fine-grained access

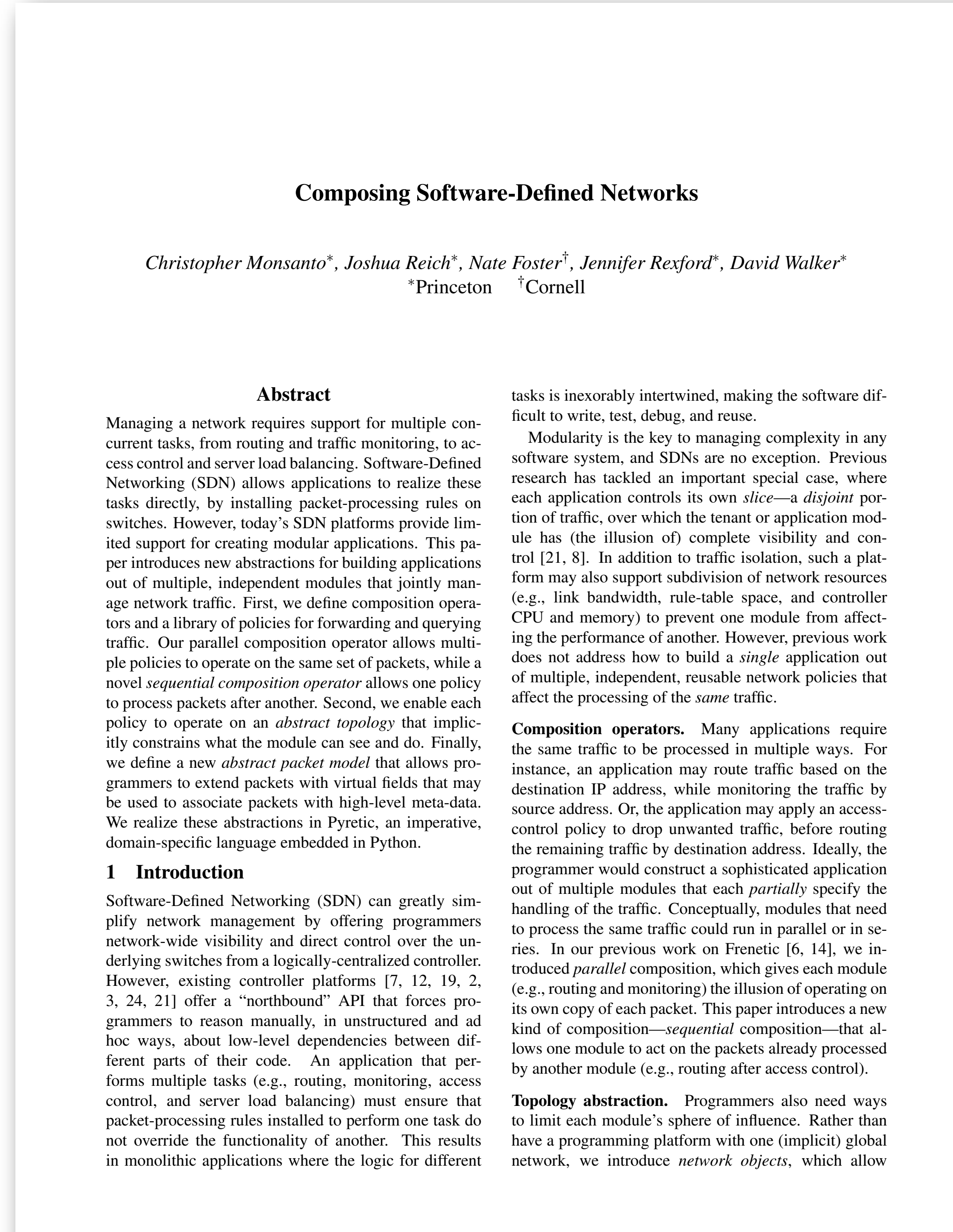
Language III [NSDI '13]

Key ideas:

- NetCore||

```
P ::= f=n
    | switch s
    | P1 | P2
    | P1 & P2
A ::= drop
    | id
    | fwd n
    | flood
C ::= P => A
    | C1 | C2
    | C1 >> C2
```

- Sequential composition
- Virtual fields



Composing Software-Defined Networks

Christopher Monsanto[*], *Joshua Reich*[*], *Nate Foster*[†], *Jennifer Rexford*[*], *David Walker*[*]
[*]Princeton [†]Cornell

Abstract

Managing a network requires support for multiple concurrent tasks, from routing and traffic monitoring, to access control and server load balancing. Software-Defined Networking (SDN) allows applications to realize these tasks directly, by installing packet-processing rules on switches. However, today's SDN platforms provide limited support for creating modular applications. This paper introduces new abstractions for building applications out of multiple, independent modules that jointly manage network traffic. First, we define composition operators and a library of policies for forwarding and querying traffic. Our parallel composition operator allows multiple policies to operate on the same set of packets, while a novel *sequential composition operator* allows one policy to process packets after another. Second, we enable each policy to operate on an *abstract topology* that implicitly constrains what the module can see and do. Finally, we define a new *abstract packet model* that allows programmers to extend packets with virtual fields that may be used to associate packets with high-level meta-data. We realize these abstractions in Pyretic, an imperative, domain-specific language embedded in Python.

1 Introduction

Software-Defined Networking (SDN) can greatly simplify network management by offering programmers network-wide visibility and direct control over the underlying switches from a logically-centralized controller. However, existing controller platforms [7, 12, 19, 2, 3, 24, 21] offer a "northbound" API that forces programmers to reason manually, in unstructured and ad hoc ways, about low-level dependencies between different parts of their code. An application that performs multiple tasks (e.g., routing, monitoring, access control, and server load balancing) must ensure that packet-processing rules installed to perform one task do not override the functionality of another. This results in monolithic applications where the logic for different tasks is inexorably intertwined, making the software difficult to write, test, debug, and reuse.

Modularity is the key to managing complexity in any software system, and SDNs are no exception. Previous research has tackled an important special case, where each application controls its own *slice*—a *disjoint* portion of traffic, over which the tenant or application module has (the illusion of) complete visibility and control [21, 8]. In addition to traffic isolation, such a platform may also support subdivision of network resources (e.g., link bandwidth, rule-table space, and controller CPU and memory) to prevent one module from affecting the performance of another. However, previous work does not address how to build a *single* application out of multiple, independent, reusable network policies that affect the processing of the *same* traffic.

Composition operators. Many applications require the same traffic to be processed in multiple ways. For instance, an application may route traffic based on the destination IP address, while monitoring the traffic by source address. Or, the application may apply an access-control policy to drop unwanted traffic, before routing the remaining traffic by destination address. Ideally, the programmer would construct a sophisticated application out of multiple modules that each *partially* specify the handling of the traffic. Conceptually, modules that need to process the same traffic could run in parallel or in series. In our previous work on Frenetic [6, 14], we introduced *parallel* composition, which gives each module (e.g., routing and monitoring) the illusion of operating on its own copy of each packet. This paper introduces a new kind of composition—*sequential* composition—that allows one module to act on the packets already processed by another module (e.g., routing after access control).

Topology abstraction. Programmers also need ways to limit each module's sphere of influence. Rather than have a programming platform with one (implicit) global network, we introduce *network objects*, which allow

Language IV [PLDI '13]

Key ideas:

- Network-wide semantics

$$\frac{[\![p]\!](lp) = M'}{p \vdash \{\!|lp|\!\} \uplus M \xrightarrow{lp} M \uplus M'}$$

- Detailed "featherweight" model of software-defined networks
- Machine-checked proofs of correctness in Coq
- First real deployment

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.

Report Card

☺

- Established a beachhead for network programming languages
- Got a lot of folks thinking seriously about modular composition
- Details of the compiler and run-time system interesting

Report Card

☺

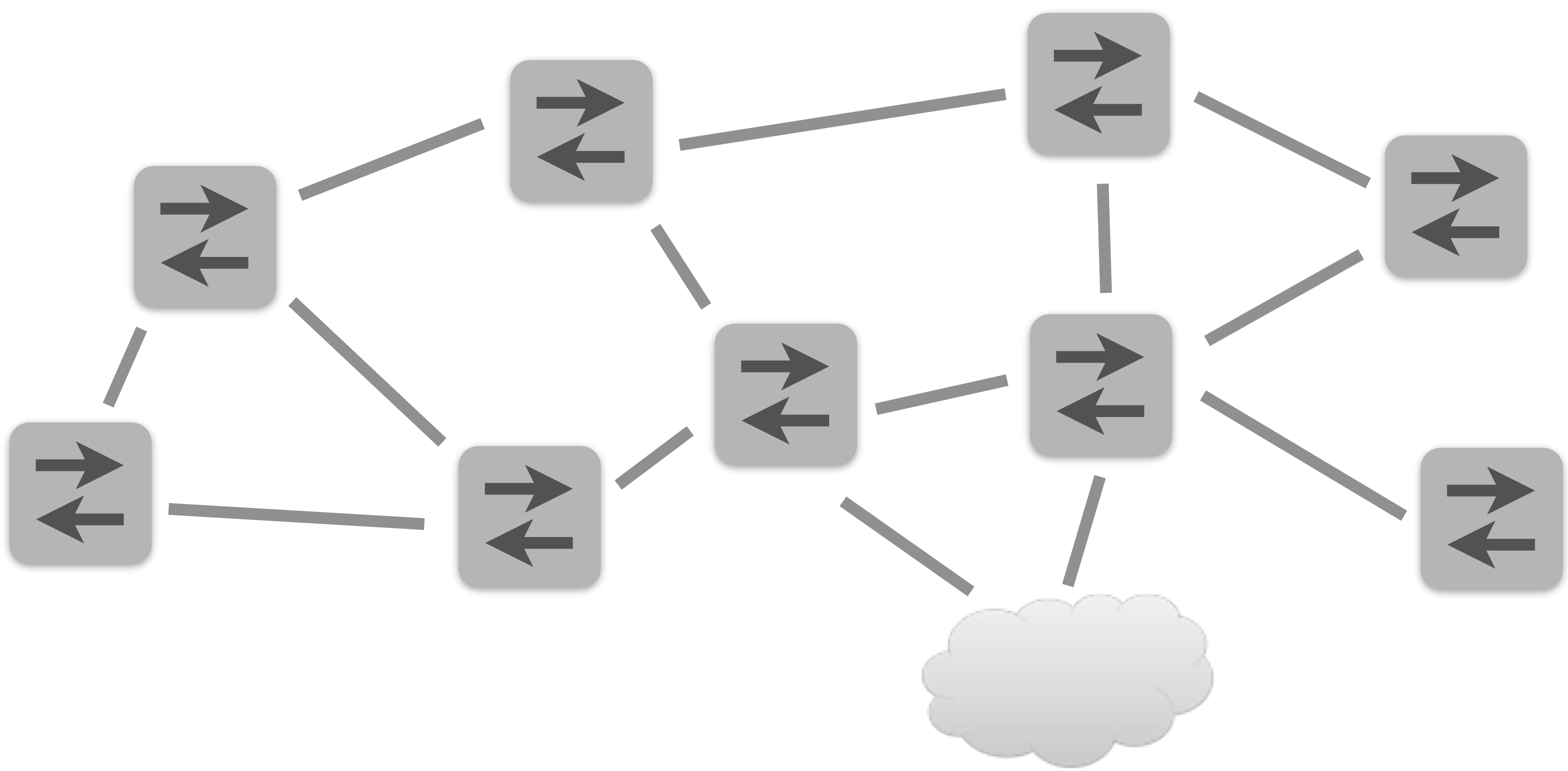
- Established a beachhead for network programming languages
- Got a lot of folks thinking seriously about modular composition
- Details of the compiler and run-time system interesting

☹

- Key design choices revisited on each iteration
- Each semantics had a precise definition but was rather ad hoc
- Unclear how new features should interact with old ones
- Could not reason equationally about network-wide behavior

Language Features

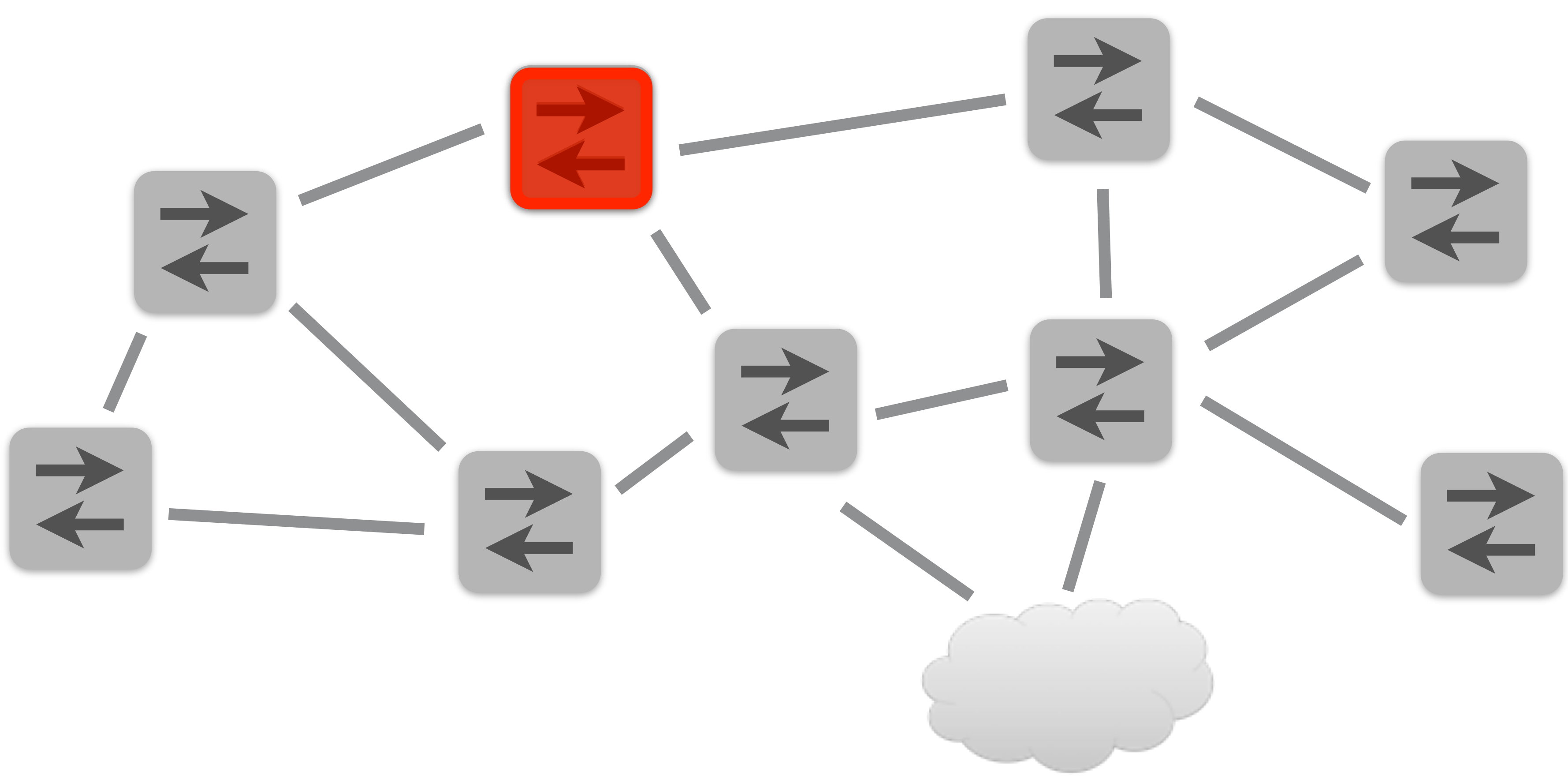
What features should a network programming language provide?*



*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

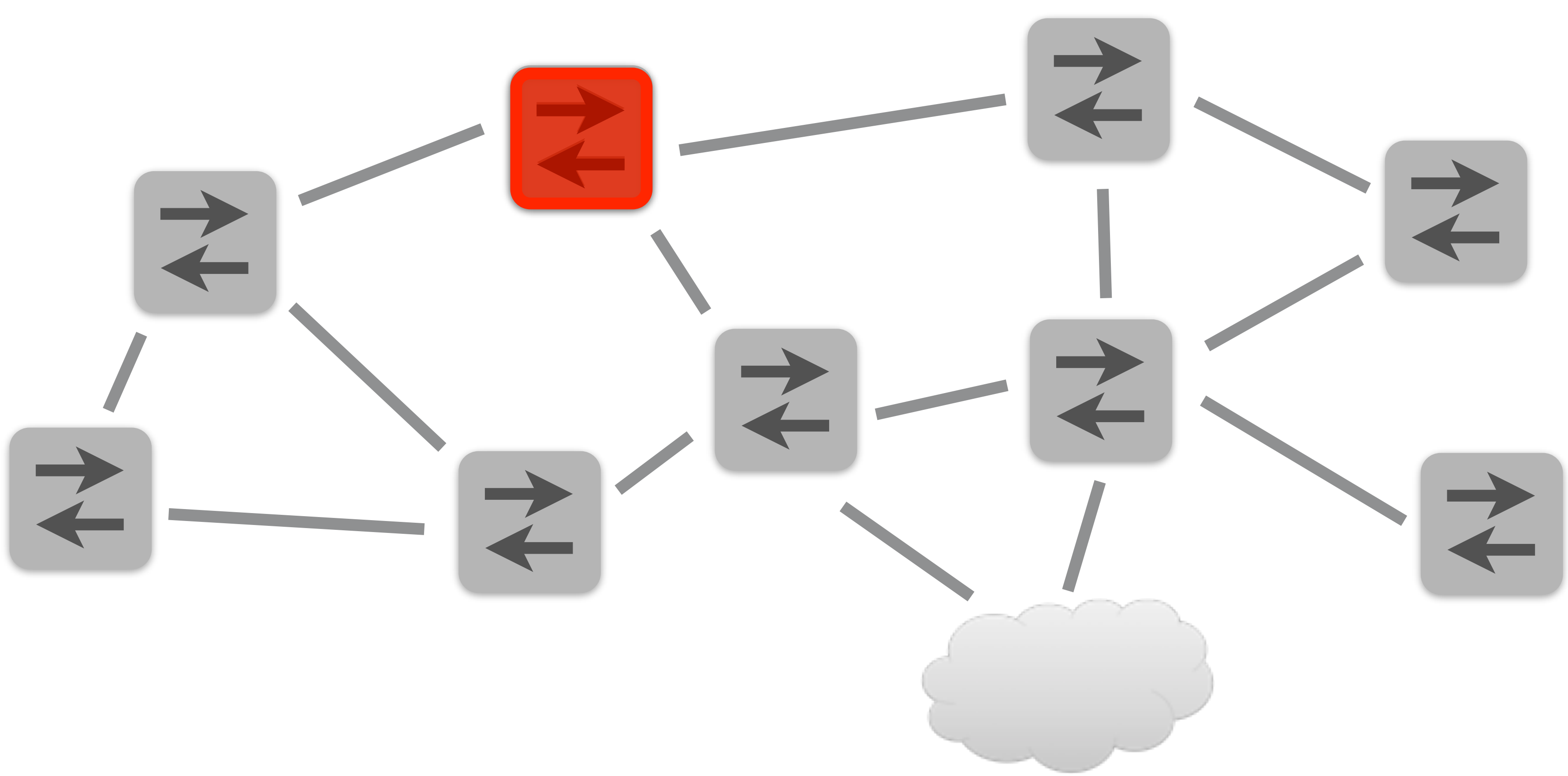


*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

- Packet predicates
- Packet transformations

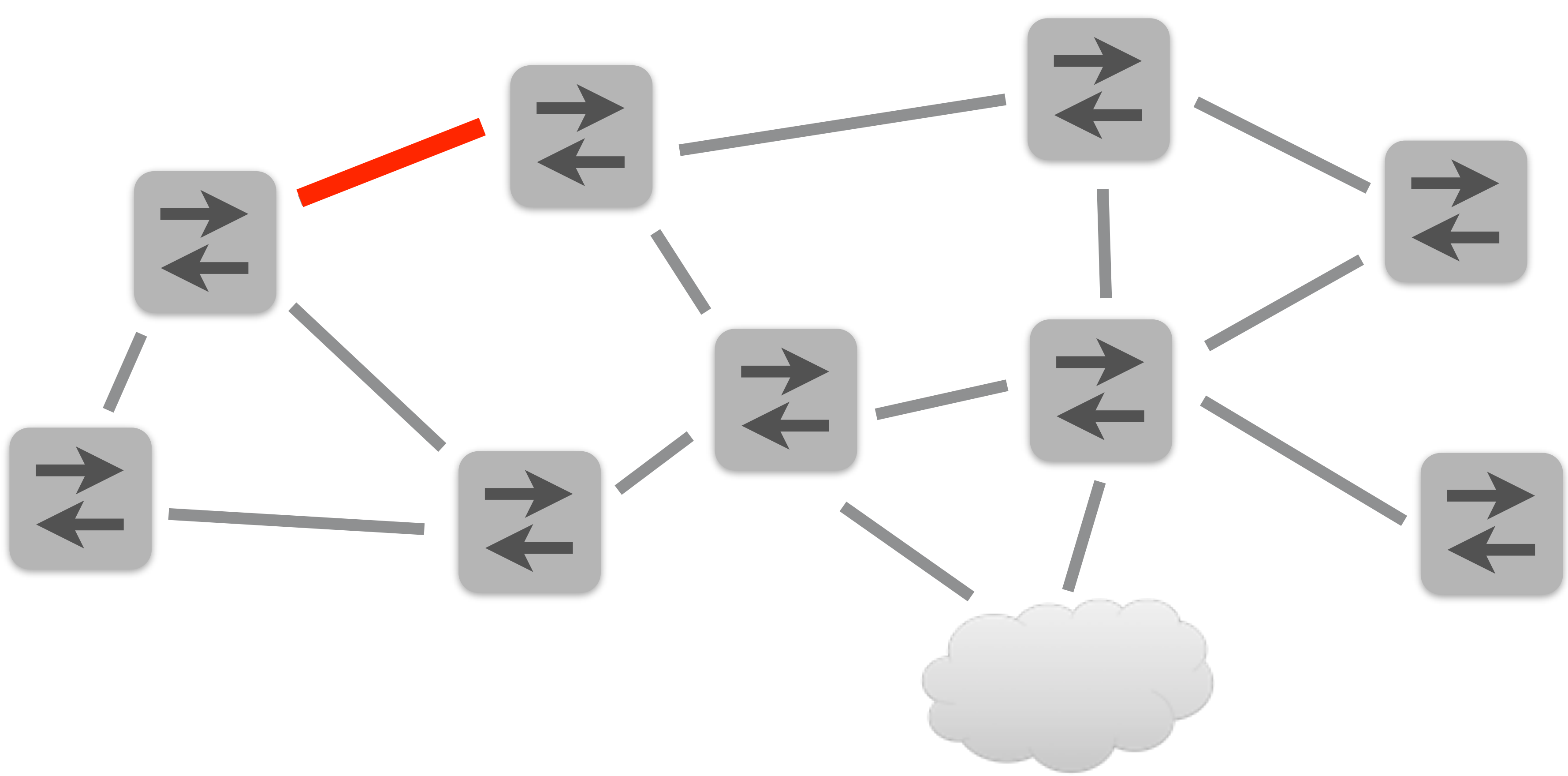


*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

- Packet predicates
- Packet transformations

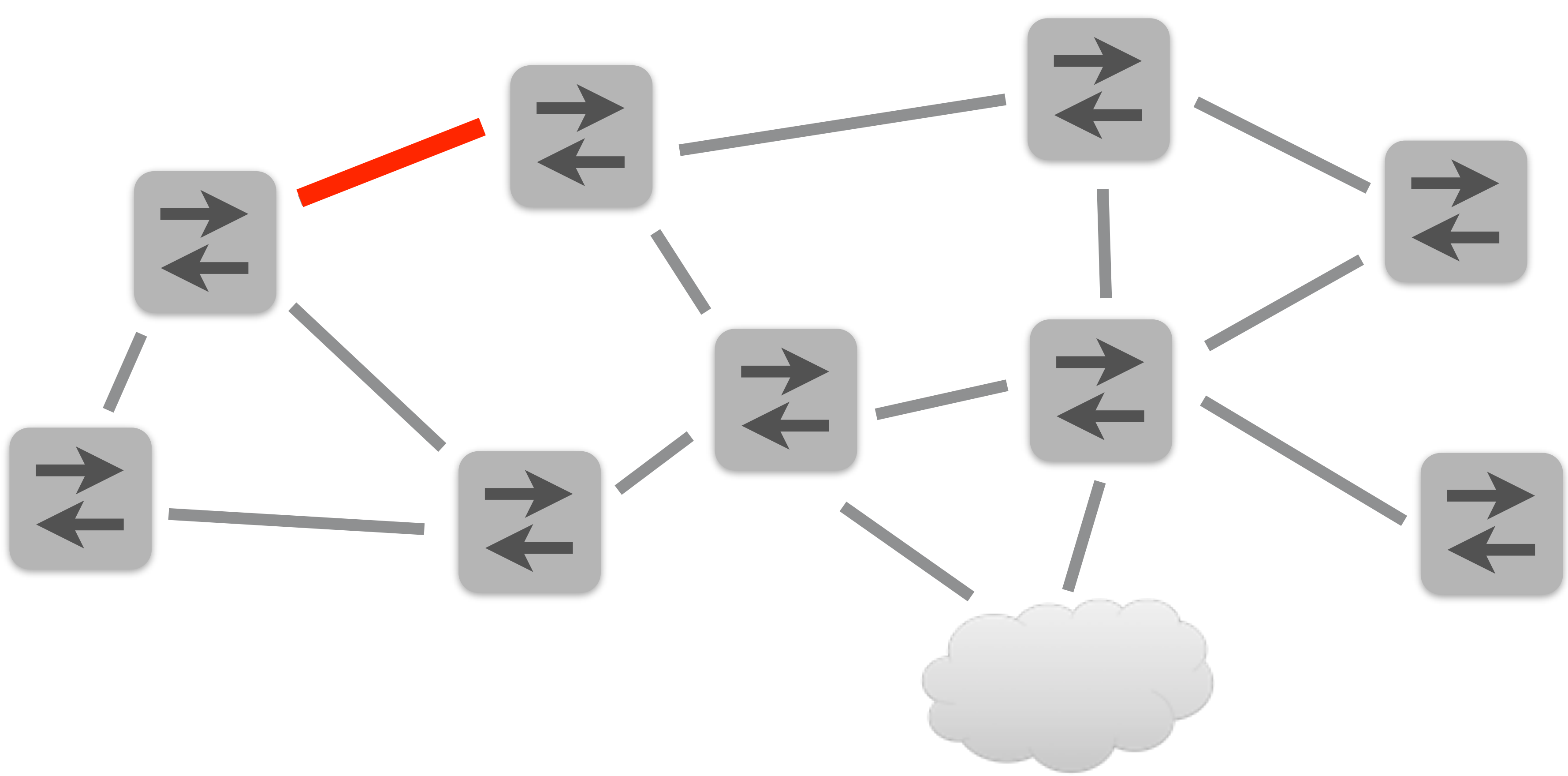


*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

- Packet predicates
- Packet transformations
- Path construction

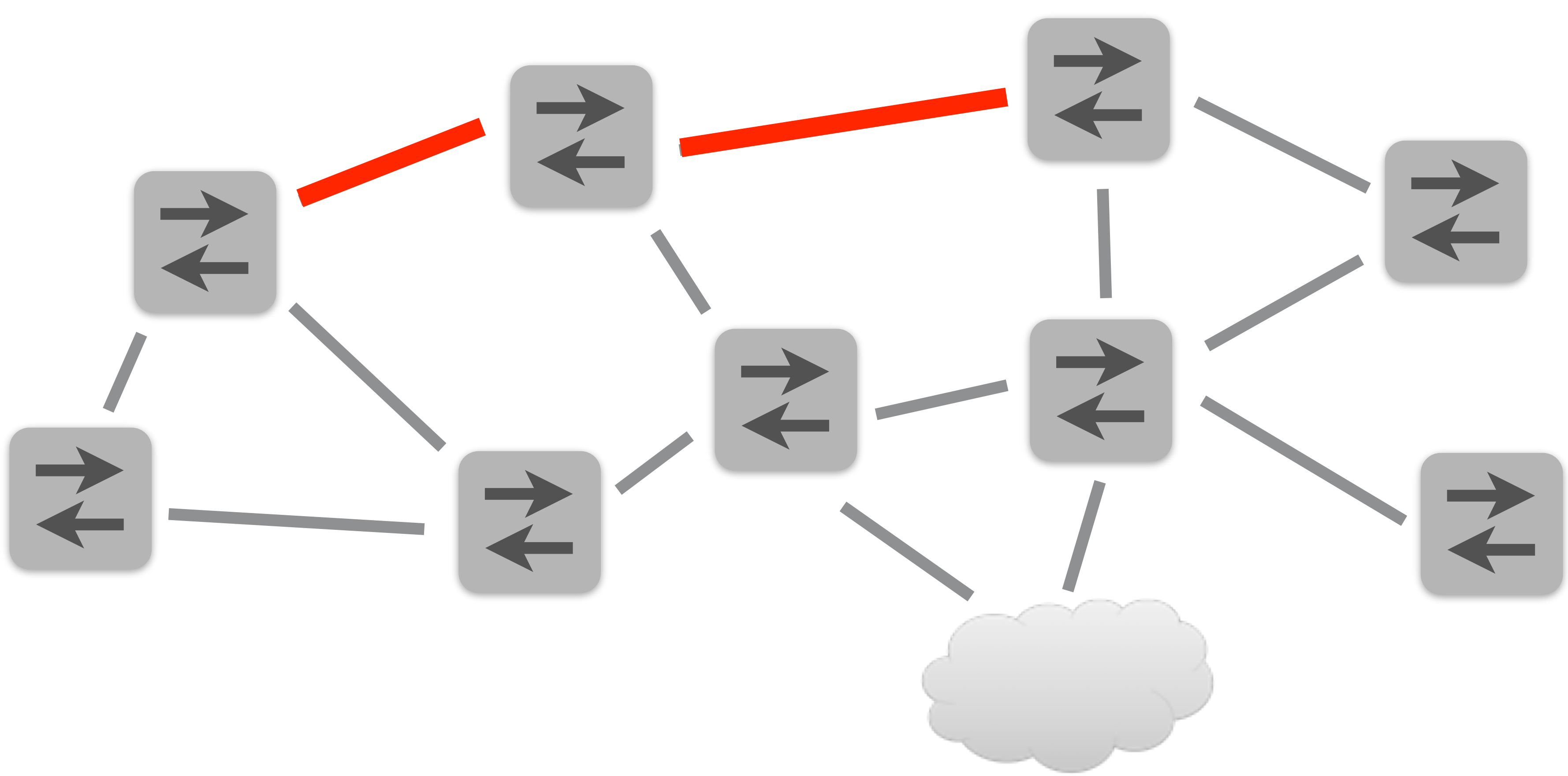


*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

- Packet predicates
- Packet transformations
- Path construction

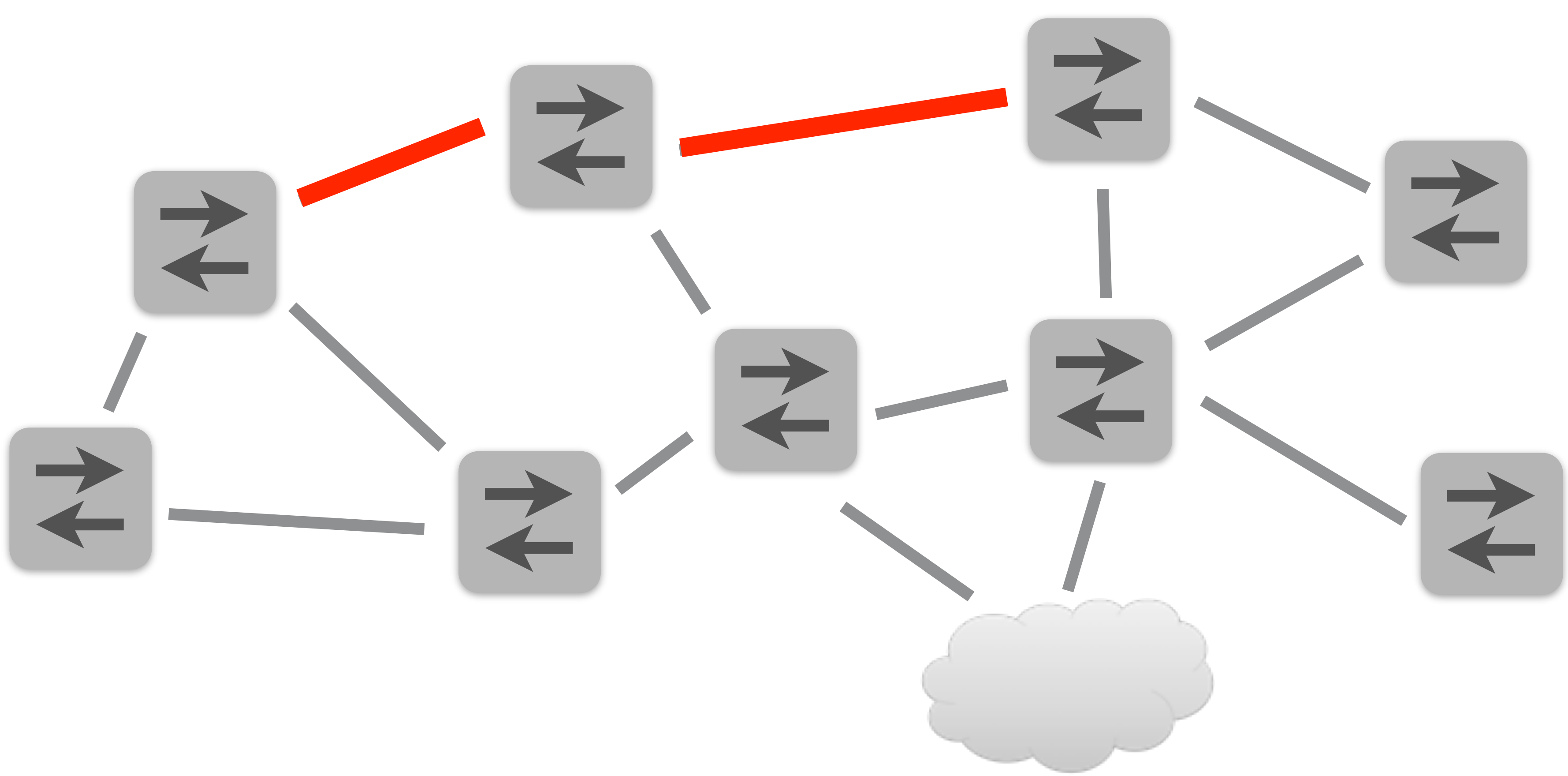


*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation

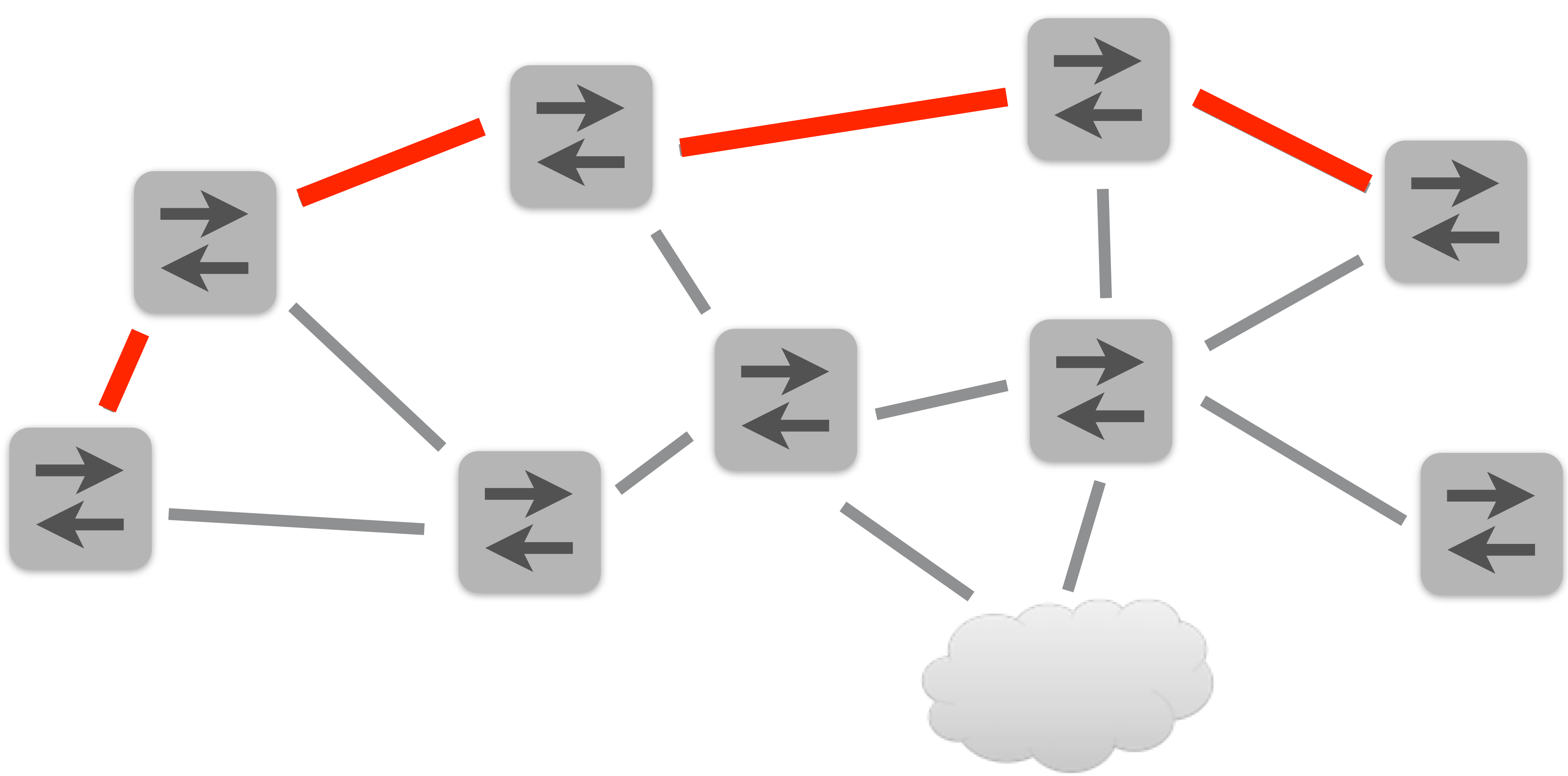


*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation

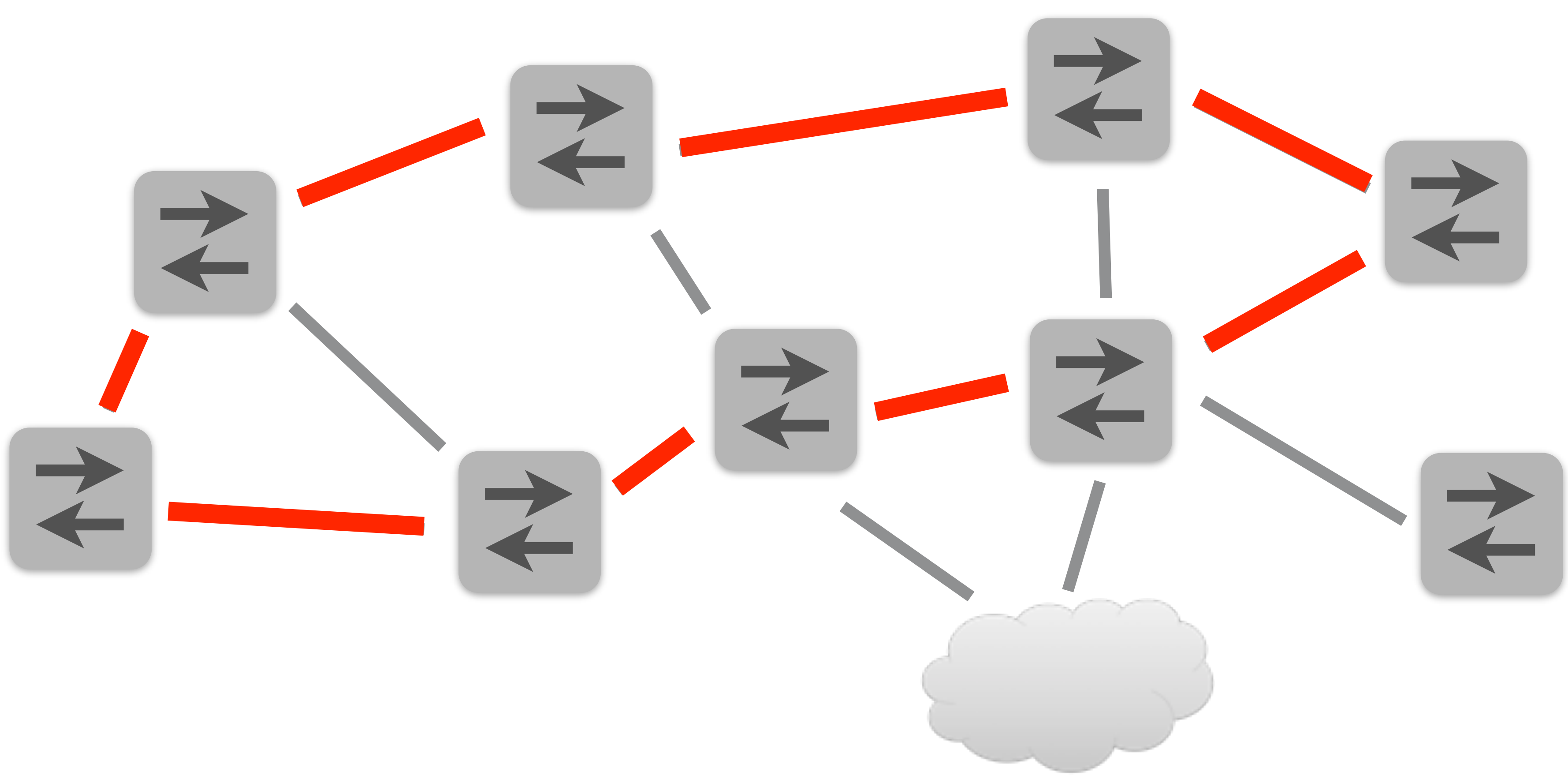


*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation

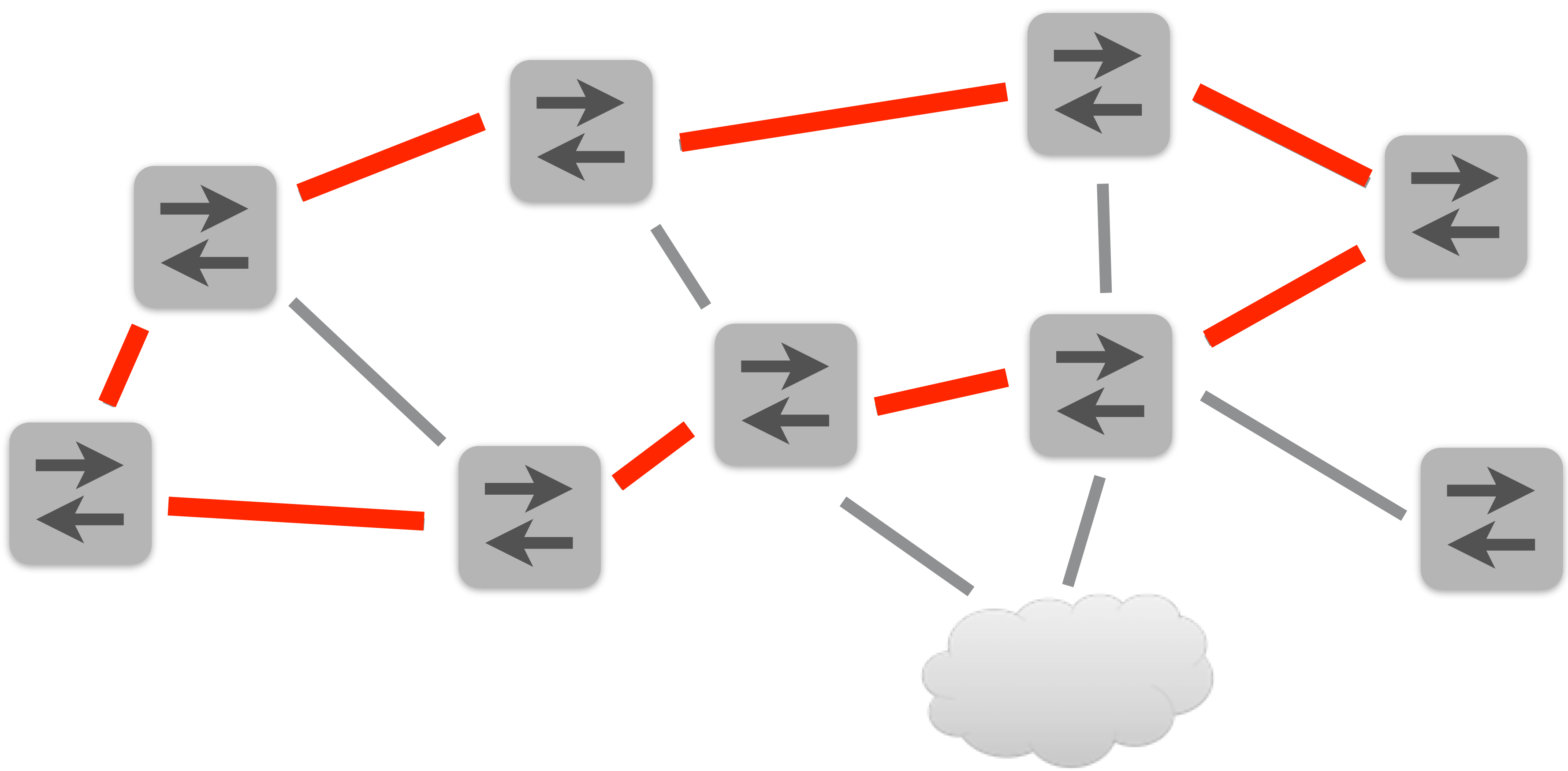


*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation
- Path union

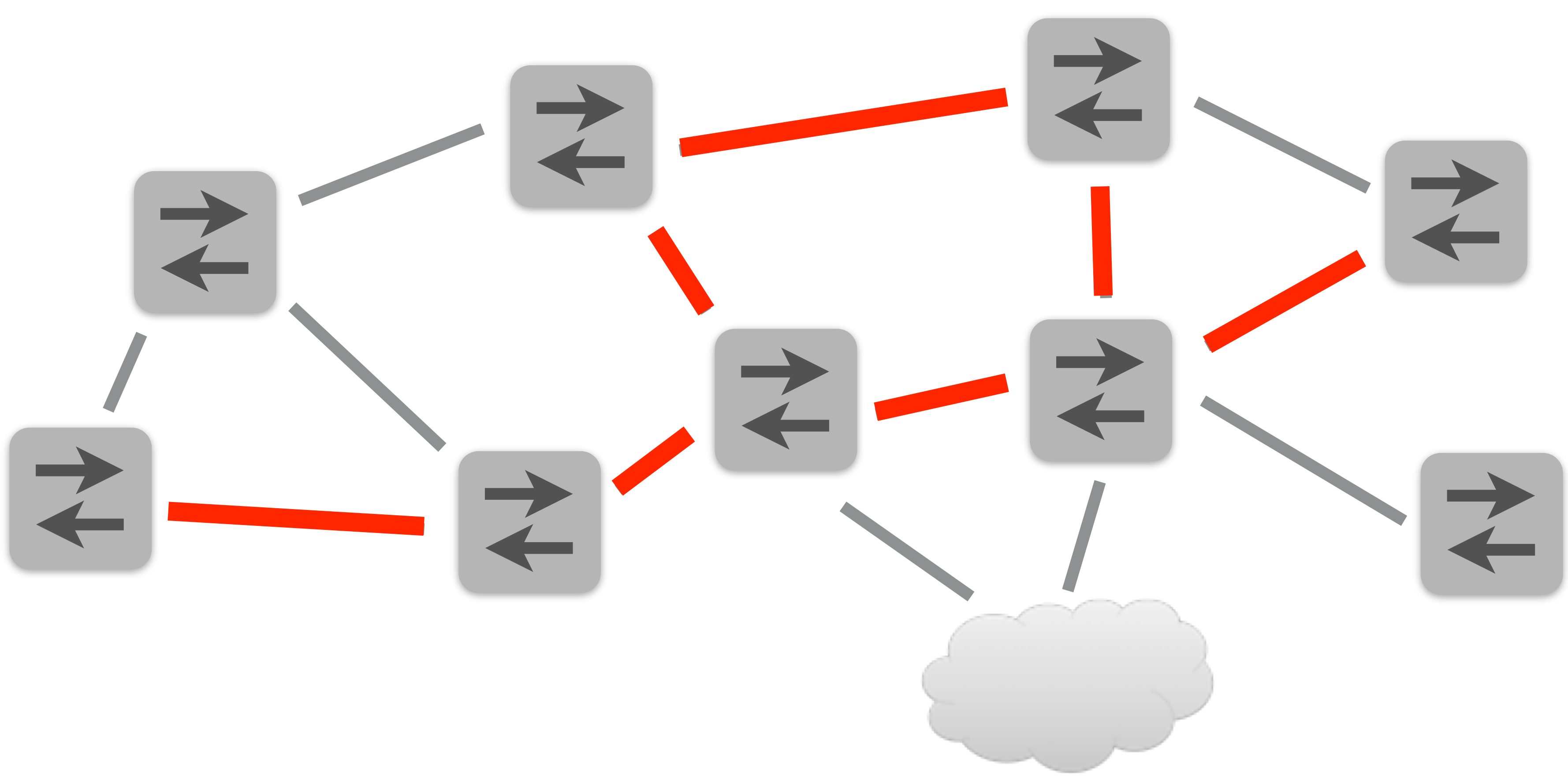


*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation
- Path union

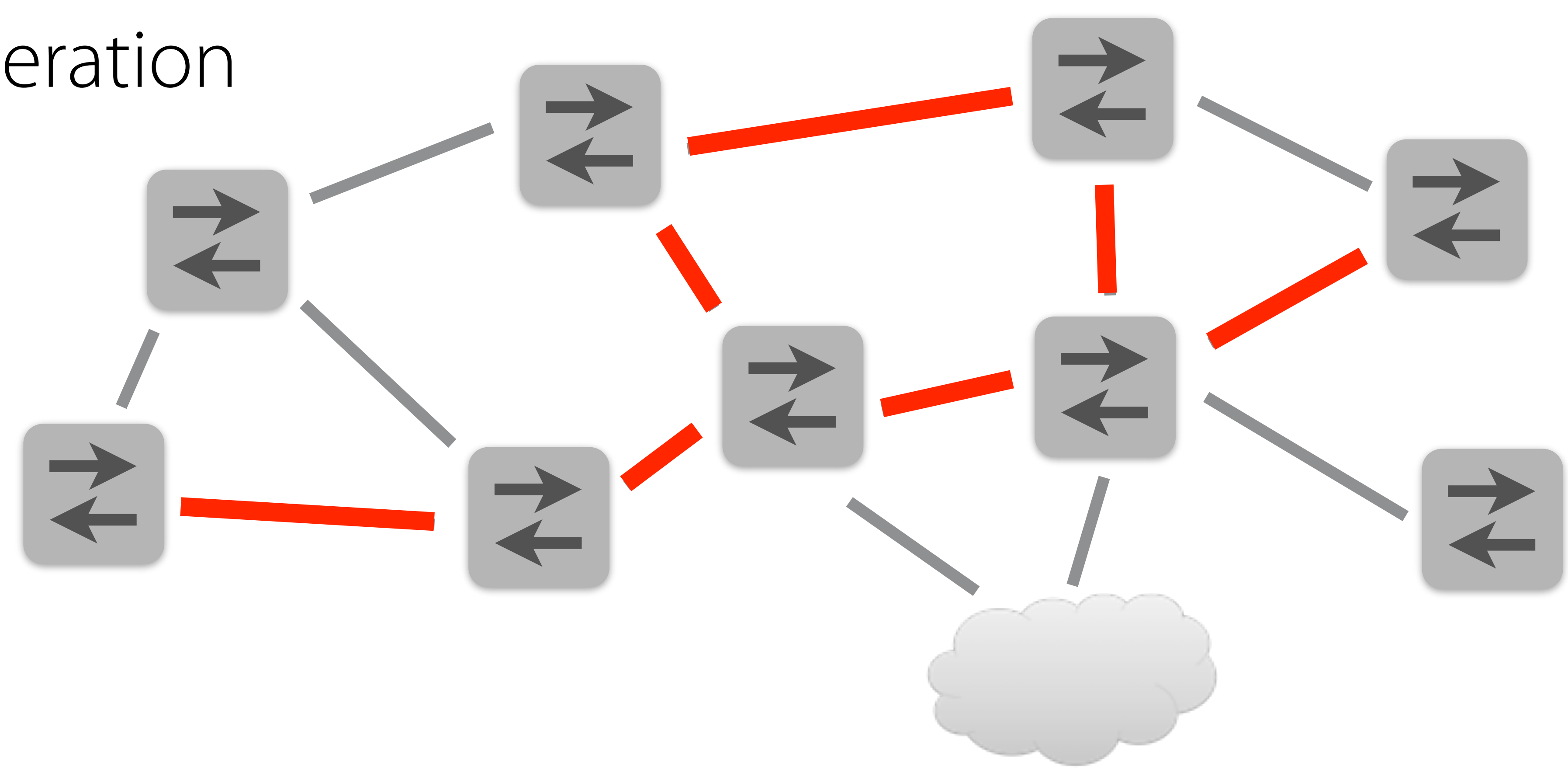


*Focusing just on packet forwarding

Language Features

What features should a network programming language provide?*

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation
- Path union
- Path iteration



*Focusing just on packet forwarding

NetKAT

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...
v ::= **0** | **1** | **2** | **3** | ...

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...
v ::= **0** | **1** | **2** | **3** | ...
a,b,c ::= **true** (* true *)

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...

v ::= **0** | **1** | **2** | **3** | ...

a,b,c ::= **true** (* true *)

| **false** (* false *)

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...

v ::= **0** | **1** | **2** | **3** | ...

a,b,c ::= **true**	(* true *)
| **false**	(* false *)
| f **=** v	(* test *)

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...

v ::= **0** | **1** | **2** | **3** | ...

a,b,c ::= **true**	(* true *)
| **false**	(* false *)
| f **=** v	(* test *)
| a_1 **|** a_2	(* disjunction *)

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...

v ::= **0** | **1** | **2** | **3** | ...

a,b,c ::= **true**	(* true *)
| **false**	(* false *)
| f **=** v	(* test *)
| a_1 **|** a_2	(* disjunction *)
| a_1 **&** a_2	(* conjunction *)

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...
v ::= **0** | **1** | **2** | **3** | ...
a,b,c ::= **true** (* true *)
| **false** (* false *)
| f **=** v (* test *)
| a_1 **|** a_2 (* disjunction *)
| a_1 **&** a_2 (* conjunction *)
| **!** a (* negation *)

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...

v ::= **0** | **1** | **2** | **3** | ...

a,b,c ::= **true** (* true *)

| **false** (* false *)

| f **=** v (* test *)

| a_1 **|** a_2 (* disjunction *)

| a_1 **&** a_2 (* conjunction *)

| **!** a (* negation *)

p,q,r ::= **filter** a (* filter *)

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...

v ::= **0** | **1** | **2** | **3** | ...

a,b,c ::= **true** (* true *)
| **false** (* false *)
| f **=** v (* test *)
| a_1 **|** a_2 (* disjunction *)
| a_1 **&** a_2 (* conjunction *)
| **!** a (* negation *)

p,q,r ::= **filter** a (* filter *)
| f **:=** v (* modification *)

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...

v ::= **0** | **1** | **2** | **3** | ...

a,b,c ::= **true** (* true *)

| **false** (* false *)

| f **=** v (* test *)

| a_1 **|** a_2 (* disjunction *)

| a_1 **&** a_2 (* conjunction *)

| **!** a (* negation *)

p,q,r ::= **filter** a (* filter *)

| f **:=** v (* modification *)

| p_1 **|** p_2 (* union *)

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...
v ::= **0** | **1** | **2** | **3** | ...

a,b,c ::= **true**	(* true *)
| **false**	(* false *)
| f **=** v	(* test *)
| a_1 **|** a_2	(* disjunction *)
| a_1 **&** a_2	(* conjunction *)
| **!** a	(* negation *)
p,q,r ::= **filter** a	(* filter *)
| f **:=** v	(* modification *)
| p_1 **|** p_2	(* union *)
| p_1 **;** p_2	(* sequence *)

NetKAT

```
f ::= switch | inport | srcmac | dstmac | ...
v ::= 0 | 1 | 2 | 3 | ...
a,b,c ::= true                        (* true *)
        | false                       (* false *)
        | f = v                       (* test *)
        | a1 | a2                     (* disjunction *)
        | a1 & a2                     (* conjunction *)
        | ! a                         (* negation *)
p,q,r ::= filter a                    (* filter *)
        | f := v                      (* modification *)
        | p1 | p2                     (* union *)
        | p1 ; p2                     (* sequence *)
        | p*                          (* iteration *)
```

NetKAT

f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...
v ::= **0** | **1** | **2** | **3** | ...

a,b,c ::=	**true**	(* true *)
	| **false**	(* false *)
	| f **=** v	(* test *)
	| a_1 **|** a_2	(* disjunction *)
	| a_1 **&** a_2	(* conjunction *)
	| **!** a	(* negation *)
p,q,r ::=	**filter** a	(* filter *)
	| f **:=** v	(* modification *)
	| p_1 **|** p_2	(* union *)
	| p_1 **;** p_2	(* sequence *)
	| p^*	(* iteration *)
	| **dup**	(* duplication *)

NetKAT

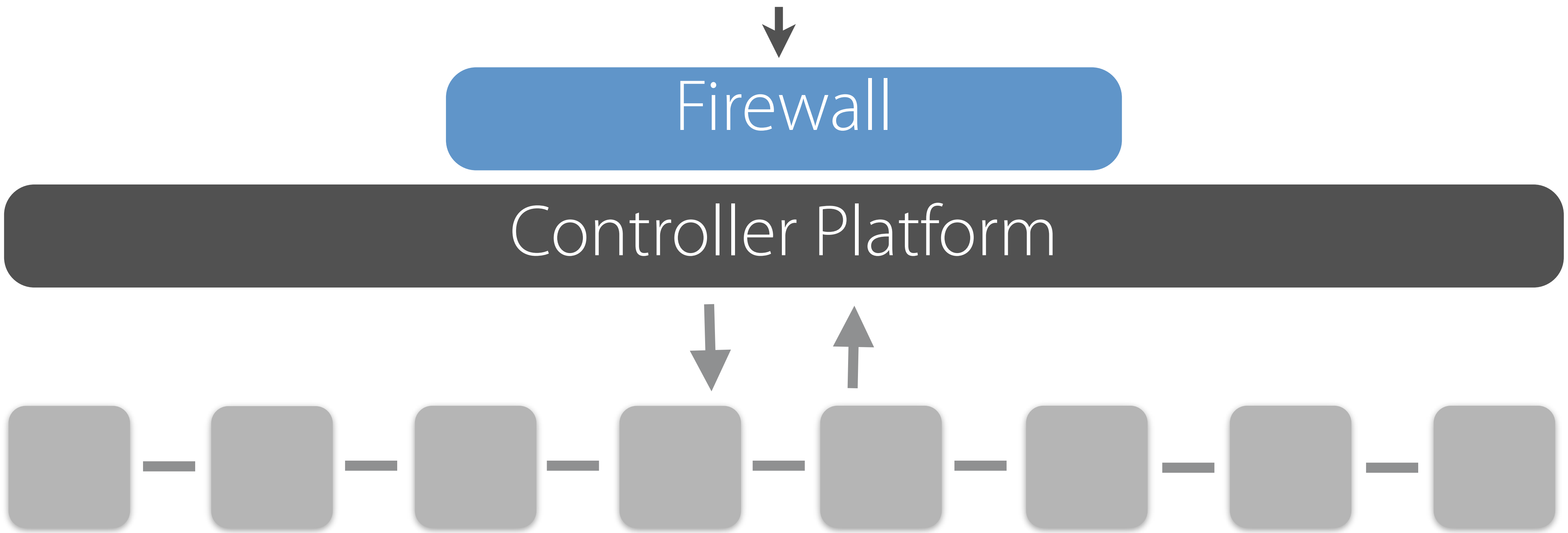
f ::= **switch** | **inport** | **srcmac** | **dstmac** | ...
v ::= **0** | **1** | **2** | **3** | ...

a,b,c ::=	**true**	(* true *)
	| **false**	(* false *)
	| f **=** v	(* test *)
	| a_1 **|** a_2	(* disjunction *)
	| a_1 **&** a_2	(* conjunction *)
	| **!** a	(* negation *)
p,q,r ::=	**filter** a	(* filter *)
	| f **:=** v	(* modification *)
	| p_1 **|** p_2	(* union *)
	| p_1 **;** p_2	(* sequence *)
	| p^*	(* iteration *)
	| **dup**	(* duplication *)

if a **then** p_1 **else** p_2 $\triangleq$ (**filter** a**;** p_1) **|** (**filter !**a**;** p_2)

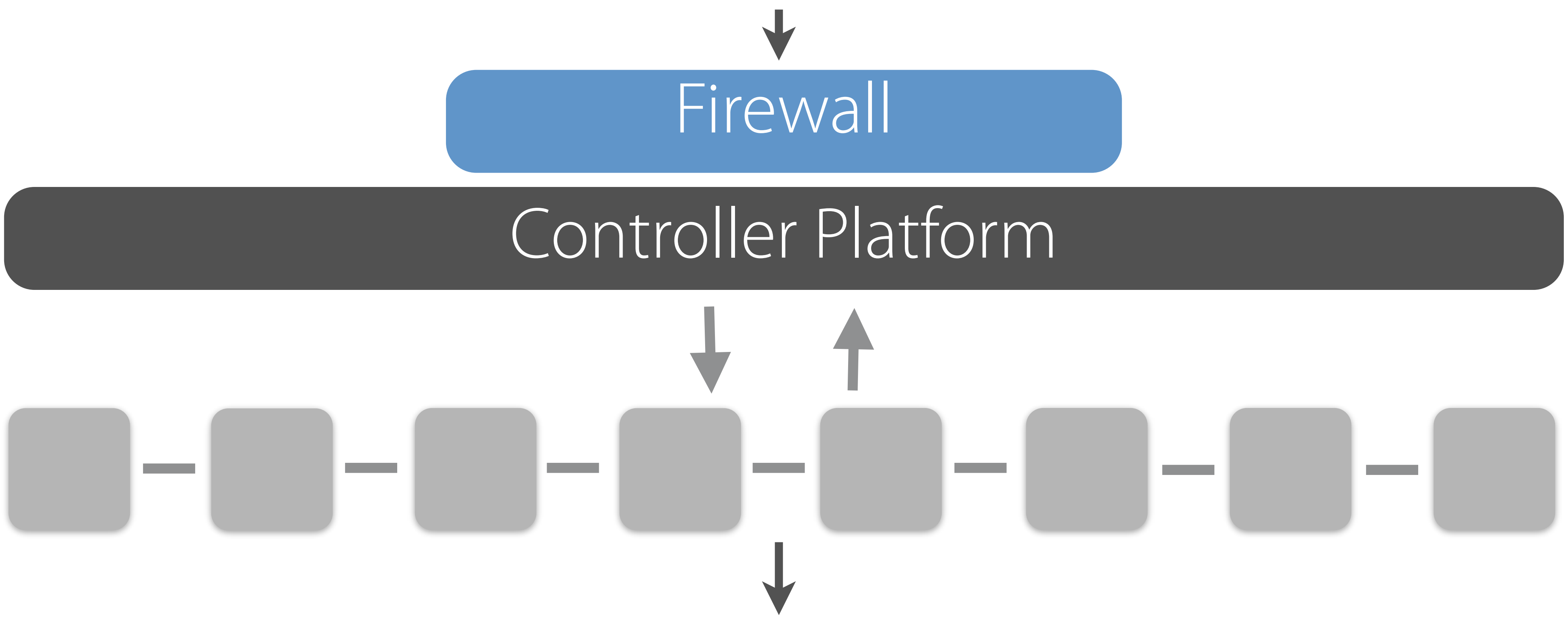
Basic Primitives

if srcip **=** 10.0.0.1 **&** !(dstport **=** 22) **then**
  port **:=** 1
else
  port **:=** 2



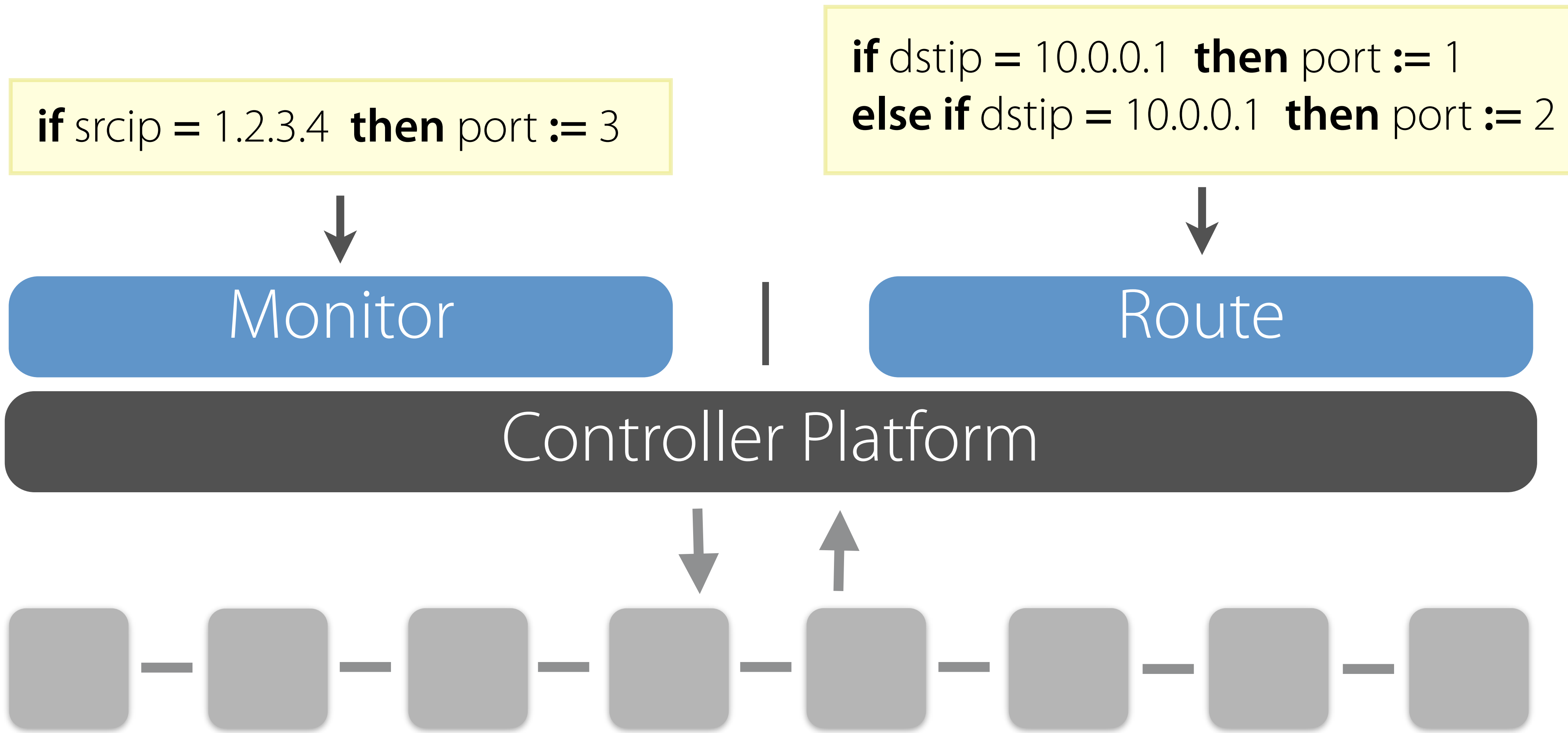
Basic Primitives

if srcip **=** 10.0.0.1 **&** !(dstport **=** 22) **then**
  port **:=** 1
else
  port **:=** 2



Pattern	**Actions**
`dstport=22`	`Drop`
`srcip=10.0.0.1`	`Forward 1`
`*`	`Forward 2`

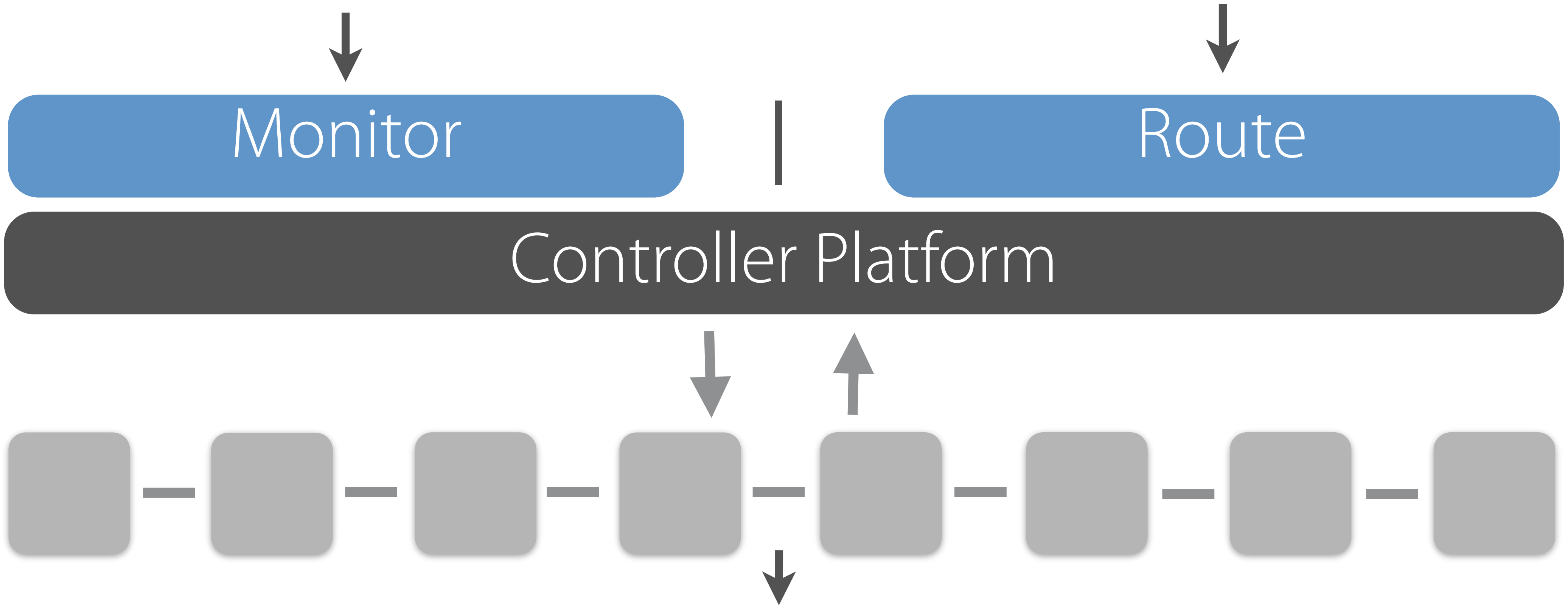
Union


if srcip = 1.2.3.4 **then** port := 3
if dstip = 10.0.0.1 **then** port := 1
else if dstip = 10.0.0.1 **then** port := 2
Monitor
|
Route
Controller Platform

Union

if srcip = 1.2.3.4 **then** port := 3

if dstip = 10.0.0.1 **then** port := 1
else if dstip = 10.0.0.1 **then** port := 2

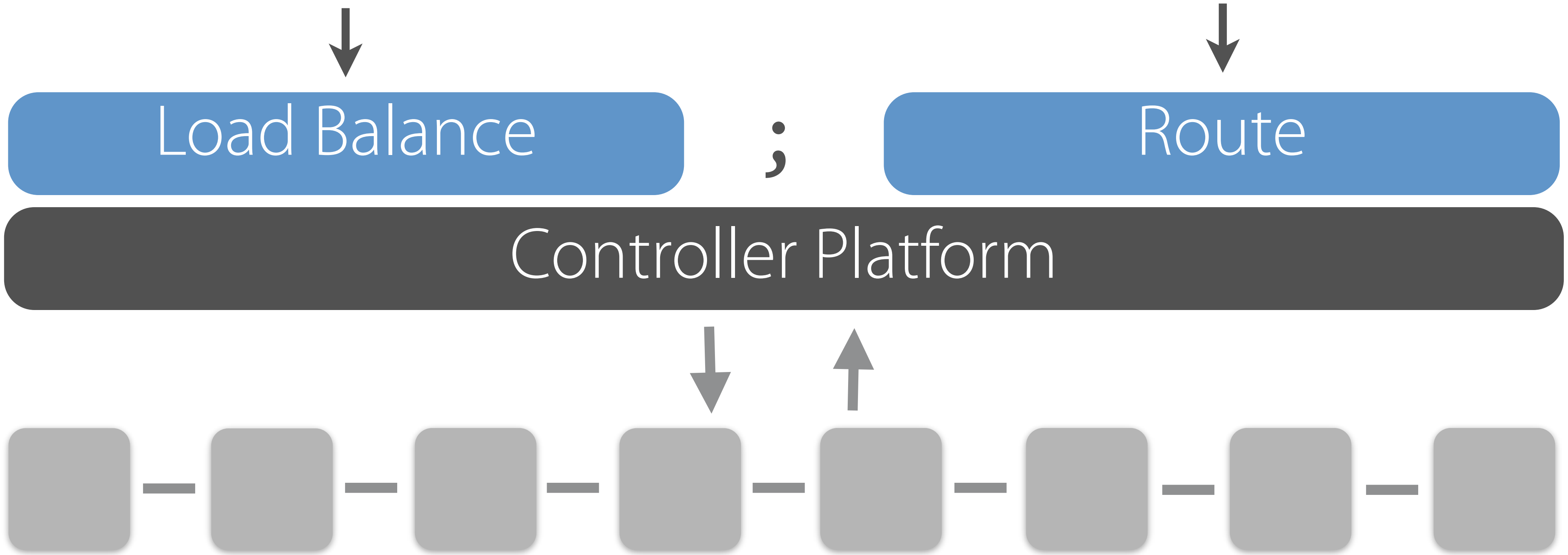


Pattern	Actions
`srcip=1.2.3.4, dstip=10.0.0.1`	`Forward 1, Forward 3`
`srcip=1.2.3.4, dstip=10.0.0.2`	`Forward 2, Forward 3`
`srcip=1.2.3.4`	`Forward 3`
`dstip=10.0.0.1`	`Forward 1`
`dstip=10.0.0.2`	`Forward 2`

Sequence

if srcip = *0 **then** dstip := 10.0.0.1
else if srcip = *1 **then** dstip := 10.0.0.2

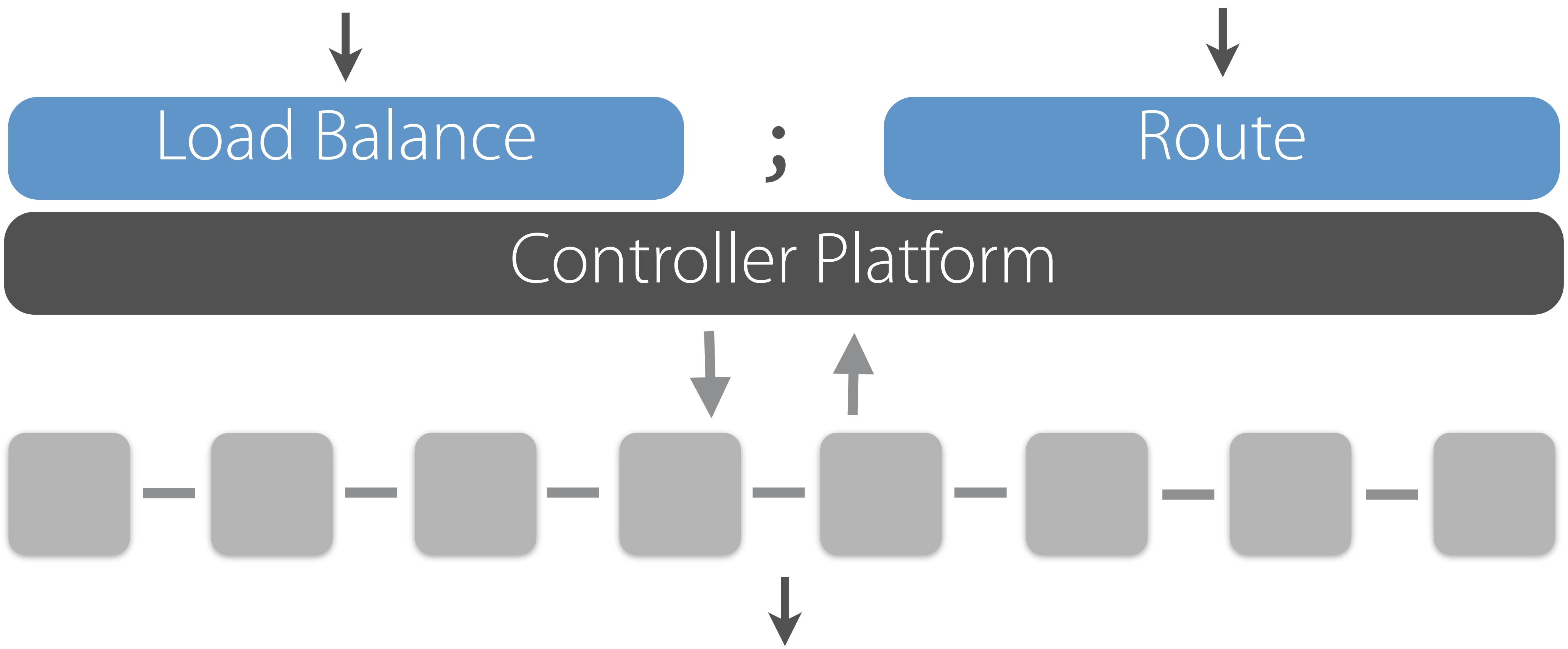
if dstip = 10.0.0.1 **then** port := 1
else if dstip = 10.0.0.2 **then** port := 2



Sequence

if srcip = *0 **then** dstip := 10.0.0.1
else if srcip = *1 **then** dstip := 10.0.0.2

if dstip = 10.0.0.1 **then** port := 1
else if dstip = 10.0.0.2 **then** port := 2

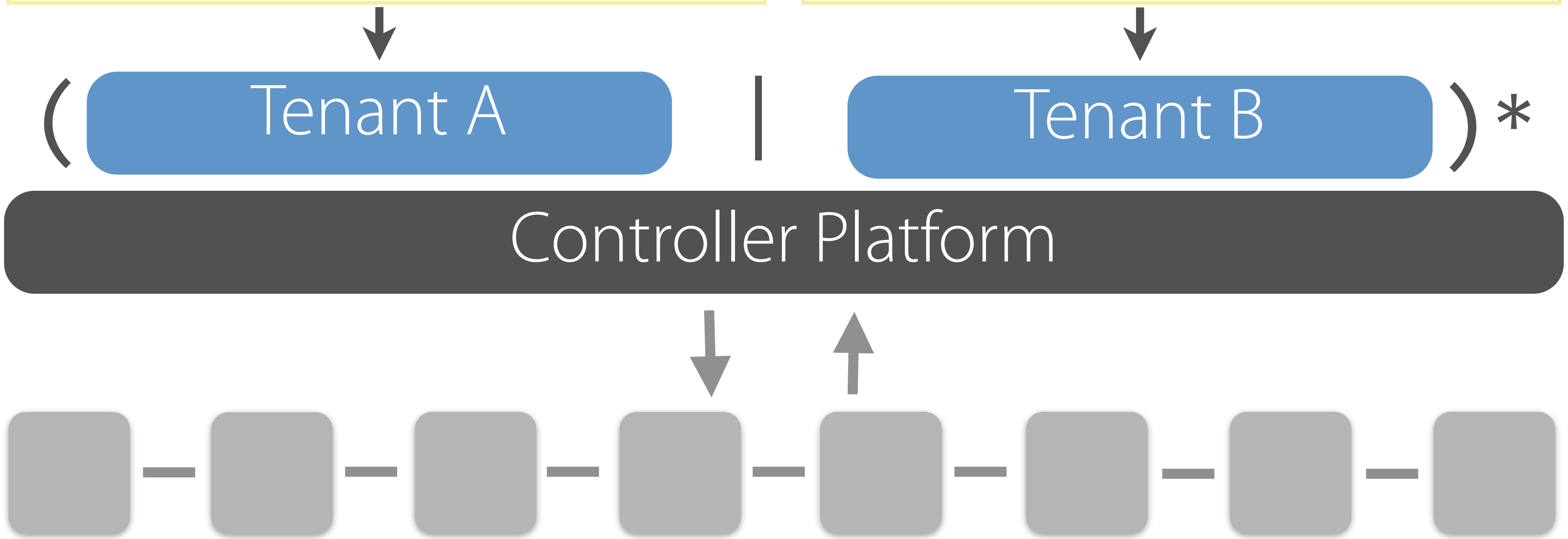


Pattern	Actions
`srcip=*0`	`dstip:=10.0.0.1, Forward 1`
`srcip=*1`	`dstip:=10.0.0.2, Forward 2`

Iteration

if dstip **=** 192.168.0.0/16 **then**
  port **:=** B
else if port = A **&** dstport **=** 80 **then**
  port **:=** 1

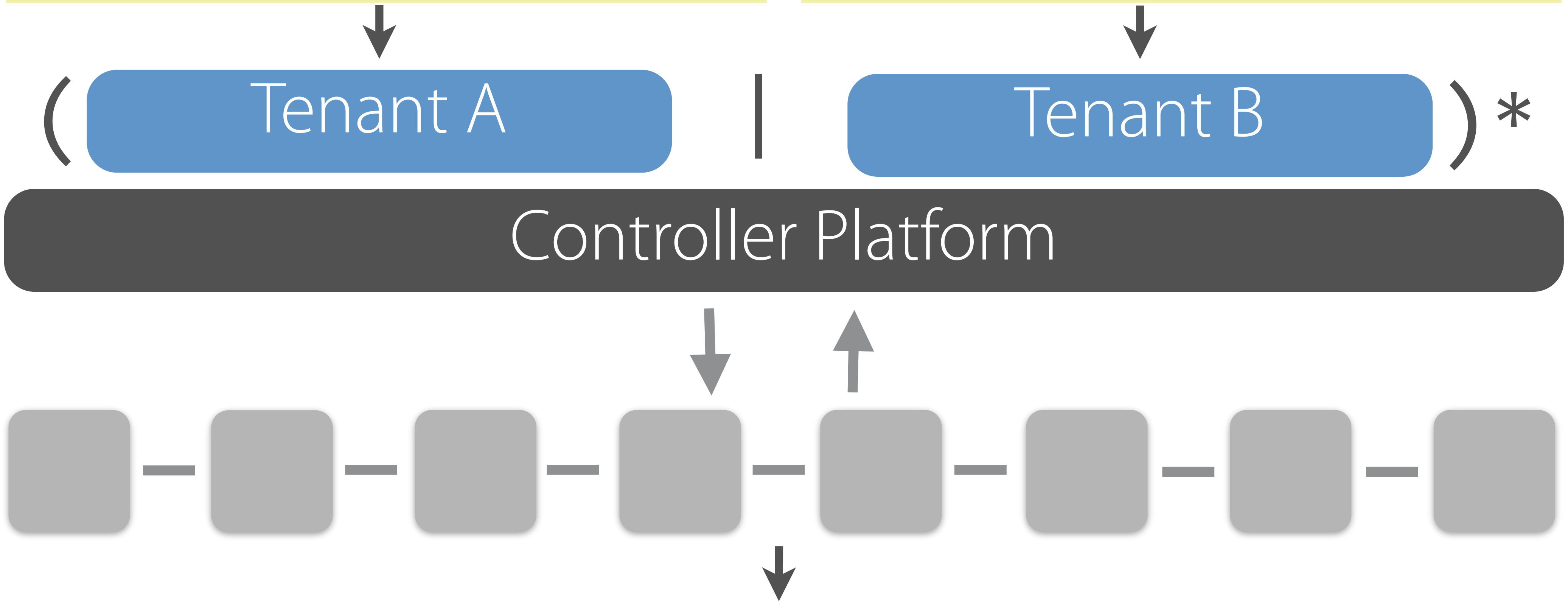
if dstip **=** 10.0.0.0/8 **then**
  port **:=** A
else if port = B **&** dstport = 22 **then**
  port **:=** 2



Iteration

if dstip **=** 192.168.0.0/16 **then**
  port **:=** B
else if port = A **&** dstport **=** 80 **then**
  port **:=** 1

if dstip **=** 10.0.0.0/8 **then**
  port **:=** A
else if port = B **&** dstport = 22 **then**
  port **:=** 2



Pattern	Actions
dstip=10.0.0.0/8, dstport=80	Forward 1
dstip=192.168.0.0/16, dstport=22	Forward 2
*	Drop

Semantic Foundation

The design of NetKAT is not an accident!

Its foundation rests upon canonical mathematical structure:

- Regular operators (**|**, **;**, and *****) encode paths through topology
- Boolean operators (**&**, **|**, and **!**) encode switch tables

This is called a ***Kleene Algebra with Tests (KAT)*** [Kozen '96]

KAT has an accompanying proof system for showing equivalences of the form p ~ q

Semantic Foundation

The design of NetKAT is not an accident!

Its foundation rests upon canonical mathematical structure:

- Regular operators (**|**, **;**, and *****) encode paths through topology
- Boolean operators (**&**, **|**, and **!**) encode switch tables

This is called a ***Kleene Algebra with Tests (KAT)*** [Kozen '96]

KAT has an accompanying proof system for showing equivalences of the form p ~ q

Theorems

- *Soundness*: programs related by the axioms are equivalent
- *Completeness*: equivalent programs are related by the axioms
- *Decidabilty*: there is an algorithm for deciding equivalence

NetKAT Equational Theory

Kleene Algebra

p **|** (q **|** r) ~ (p **|** q) **|** r

p **|** q ~ q **|** p

p **| filter false** ~ p

p **|** p ~ p

p **;** (q **;** r) ~ (p **;** q) **;** r

p; (q **|** r) ~ p **;** q **|** p **;** r

(p **|** q) **;** r ~ p **;** r **|** q **;** r

filter true ; p ~ p

p ~ p **; filter true**

filter false ; p ~ **filter false**

p **; filter false** ~ **filter false**

filter true | p **;** p^* ~ p^*

filter true | p^* **;** p ~ p^*

p **|** q **;** r **|** r ~ r ⇒ p^* **;** q **|** r ~ r

p **|** q **;** r **|** q ~ q ⇒ p **;** r^* **|** q ~q

Boolean Algebra

a **|** (b **&** c) ~ (a **|** b) **&** (a **|** c)

a **| true** ~ **true**

a **| !** a ~ **true**

a **&** b ~ b **&** a

a **& !**a ~ **false**

a **&** a ~ a

Packet Algebra

f **:=** n**;** f′ **:=** n′ ~ f′ **:=** n′ ; f **:=** n   if f ≠ f′

f **:=** n**;** f′ **=** n′ ~ f′ **=** n′; f **:=** n   if f ≠ f′

f **:=** n**;** f **=** n ~ f **:=** n

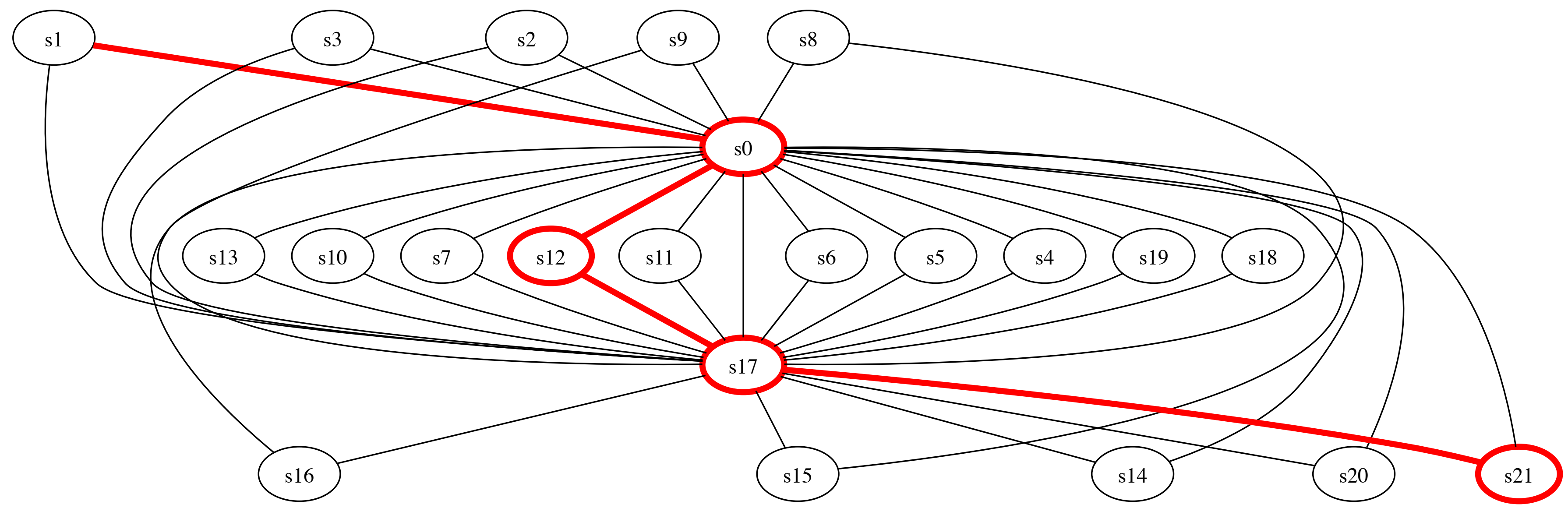
f **=** n**;** f **:=** n ~ f **=** n

f **:=** n**;** f **:=** n′ ~ f **:=** n′

f **=** n; f **=** n′ ~ **filter false**   if n ≠ n′

dup ; f **=** n ~ f **=** n **; dup**

Application: Reachability



Given:

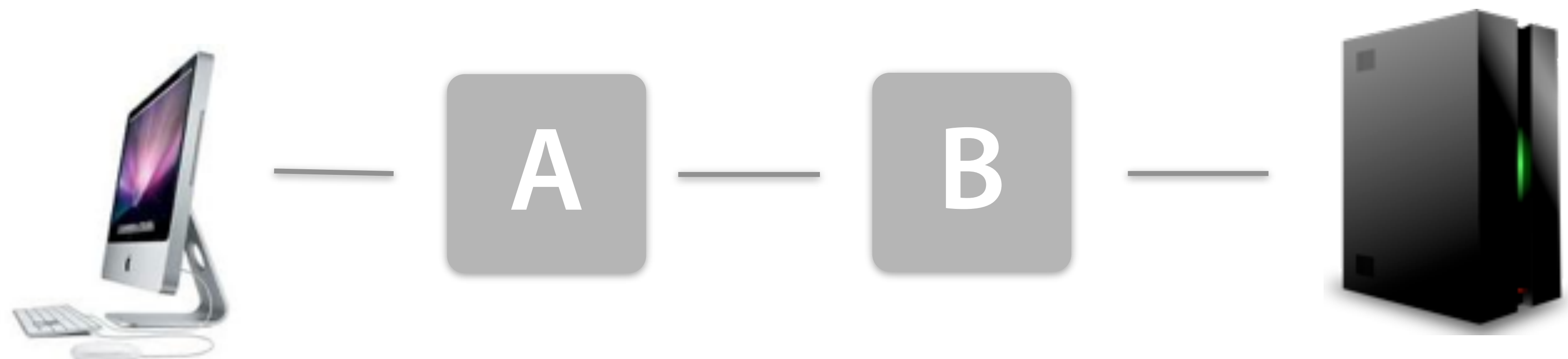
- Ingress predicate i
- Topology t
- Switch program p
- Egress predicate e

Test:

filter i**; dup;** (p**; dup;** t)***; filter** e ~ **filter false**

Application: Optimization

Given a program and a topology:

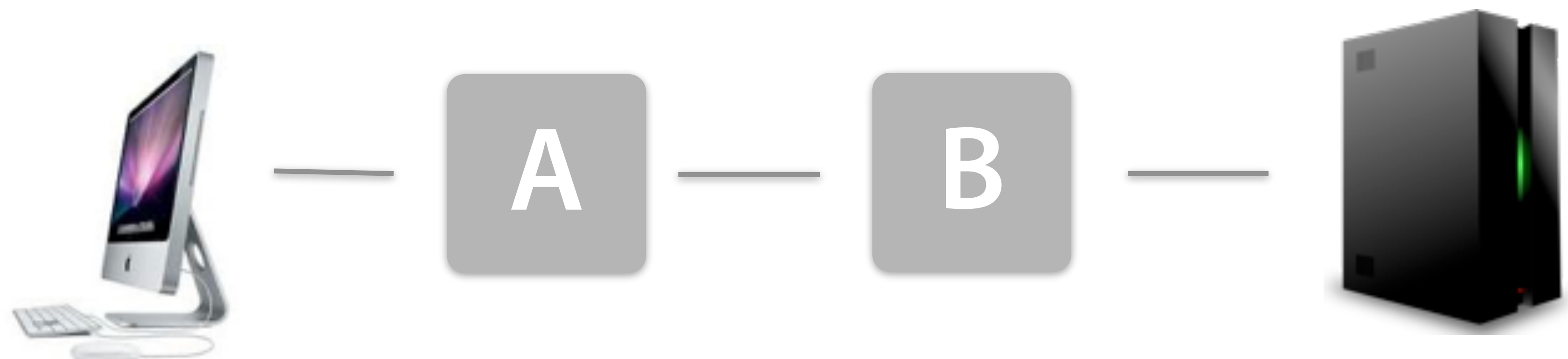


Want to be able to answer questions like:

“Will my network behave the same if I put the firewall rules on A, or on switch B (or both)?”

Application: Optimization

Given a program and a topology:



Want to be able to answer questions like:

“Will my network behave the same if I put the firewall rules on A, or on switch B (or both)?”

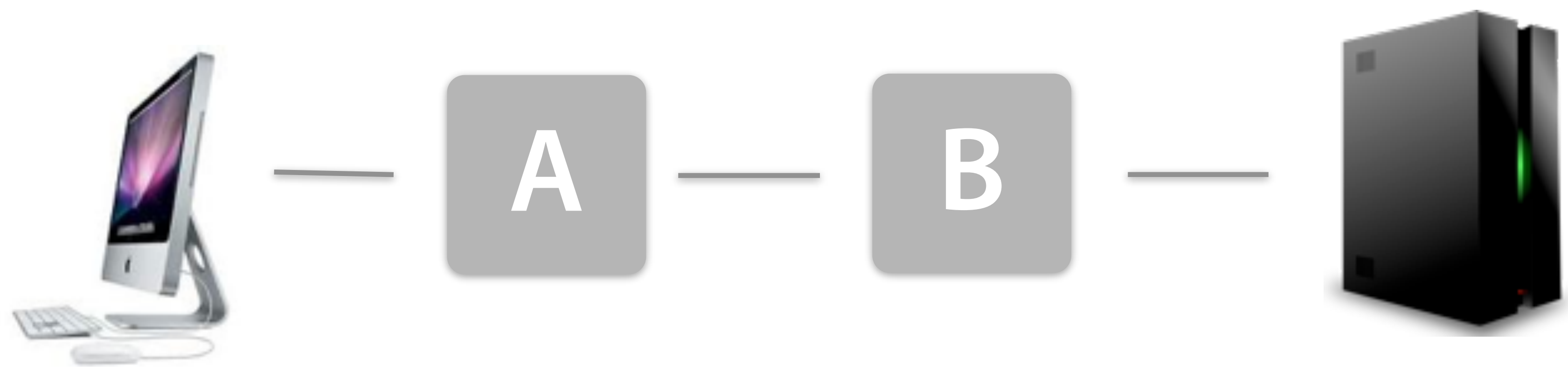
Formally, does the following equivalence hold?

(**filter** switch = A **;** <u>firewall</u>; routing) **|** (**filter** switch = B**;** routing)

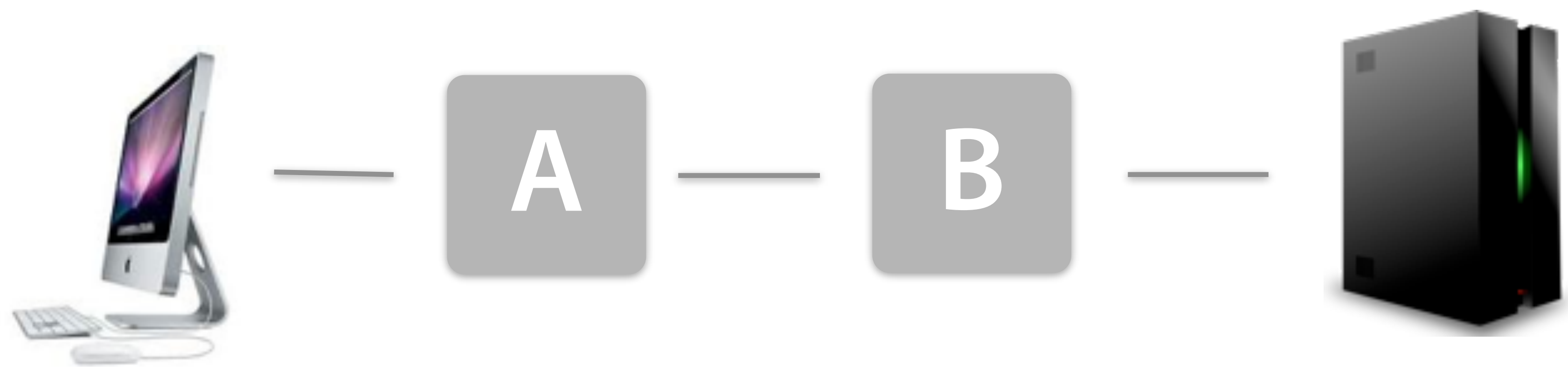
~

(**filter** switch = A **;** routing) **|** (**filter** switch = B**;** <u>firewall</u>**;** routing)

Optimization Proof



Optimization Proof



$in; (p_A; t)^*; p_A; out$

$\equiv$ { definition in, out, and p_A }

$s_A; \textsc{ssh}; ((s_A; \neg\textsc{ssh}; p + s_B; p); t)^*; p_A; s_B$

$\equiv$ { KAT-Invariant }

$s_A; \textsc{ssh}; ((s_A; \neg\textsc{ssh}; p + s_B; p); t; \textsc{ssh})^*; p_A; s_B$

$\equiv$ { KA-Seq-Dist-R }

$s_A; \textsc{ssh}; (s_A; \neg\textsc{ssh}; p; t; \textsc{ssh} + s_B; p; t; \textsc{ssh})^*; p_A; s_B$

$\equiv$ { KAT-Commute }

$s_A; \textsc{ssh}; (s_A; \neg\textsc{ssh}; \textsc{ssh}; p; t + s_B; p; t; \textsc{ssh})^*; p_A; s_B$

$\equiv$ { BA-Contra }

$s_A; \textsc{ssh}; (s_A; \mathsf{drop}; p; t + s_B; p; t; \textsc{ssh})^*; p_A; s_B$

$\equiv$ { KA-Seq-Zero, KA-Zero-Seq,KA-Plus-Comm, KA-Plus-Zero }

$s_A; \textsc{ssh}; (s_B; p; t; \textsc{ssh})^*; p_A; s_B$

$\equiv$ { KA-Unroll-L }

$s_A; \textsc{ssh}; (\mathsf{id} + (s_B; p; t; \textsc{ssh}); (s_B; p; t; \textsc{ssh})^*); p_A; s_B$

$\equiv$ { KA-Seq-Dist-L and KA-Seq-Dist-R }

$(s_A; \textsc{ssh}; p_A; s_B) +$

$(s_A; \textsc{ssh}; s_B; p; t; \textsc{ssh}; (s_B; p; t; \textsc{ssh})^*; p_A; s_B)$

$\equiv$ { KAT-Commute }

$(s_A; s_B; \textsc{ssh}; p_A) +$

$(s_A; s_B; \textsc{ssh}; p; t; \textsc{ssh}; (s_B; p; t; \textsc{ssh})^*; p_A; s_B)$

$\equiv$ { PA-Contra }

$(\mathsf{drop}; \textsc{ssh}; p_A) +$

$(\mathsf{drop}; \textsc{ssh}; p; t; \textsc{ssh}; (s_B; p; t; \textsc{ssh})^*; p_A; s_B)$

$\equiv$ { KA-Zero-Seq, KA-Plus-Idem }

drop

$\equiv$ { KA-Seq-Zero, KA-Zero-Seq, KA-Plus-Idem }

$s_A; (p_B; t)^*; (\textsc{ssh}; \mathsf{drop}; p + s_B; \mathsf{drop}; p; s_B)$

$\equiv$ { PA-Contra and BA-Contra }

$s_A; (p_B; t)^*; (\textsc{ssh}; s_A; s_B; p + s_B; \textsc{ssh}; \neg\textsc{ssh}; p; s_B)$

$\equiv$ { KAT-Commute }

$s_A; (p_B; t)^*; (\textsc{ssh}; s_A; p; s_B + \textsc{ssh}; s_B; \neg\textsc{ssh}; p; s_B)$

$\equiv$ { KA-Seq-Dist-L and KA-Seq-Dist-R }

$s_A; (p_B; t)^*; \textsc{ssh}; (s_A; p + s_B; \neg\textsc{ssh}; p); s_B$

$\equiv$ { KAT-Commute }

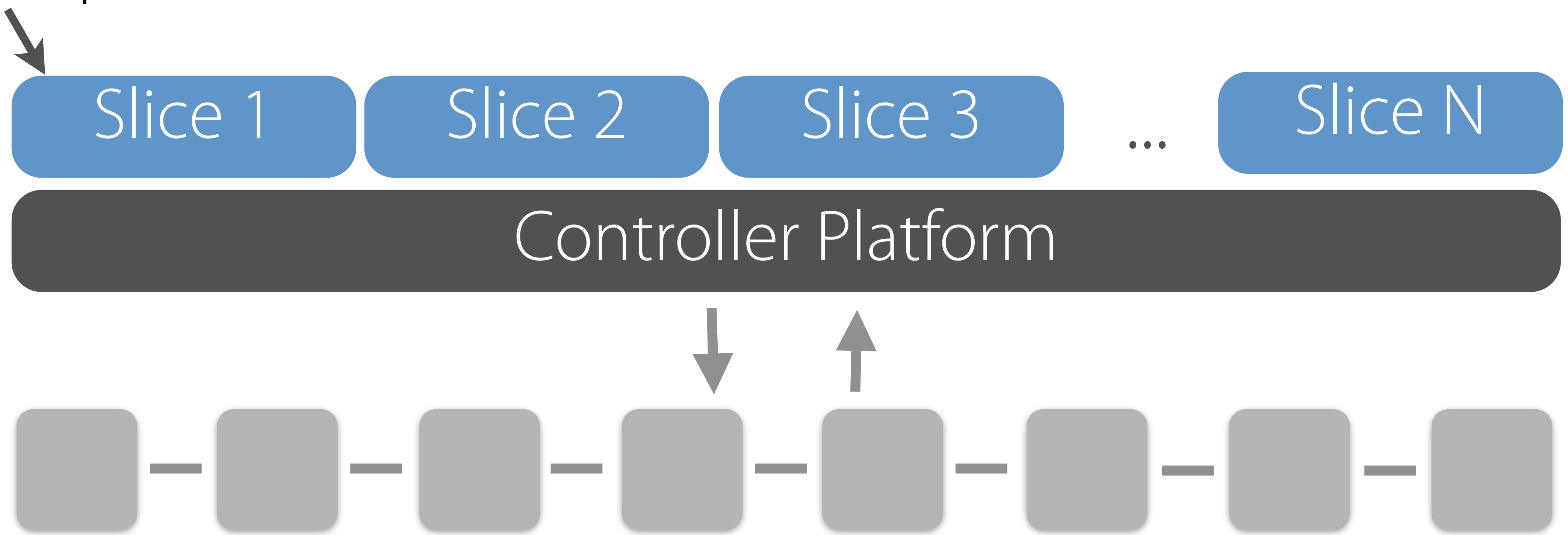
$s_A; \textsc{ssh}; (p_B; t)^*; (s_A; p + s_B; \neg\textsc{ssh}; p); s_B$

$\equiv$ { definition in, out, and p_B }

$in; (p_B; t)^*; p_B; out$

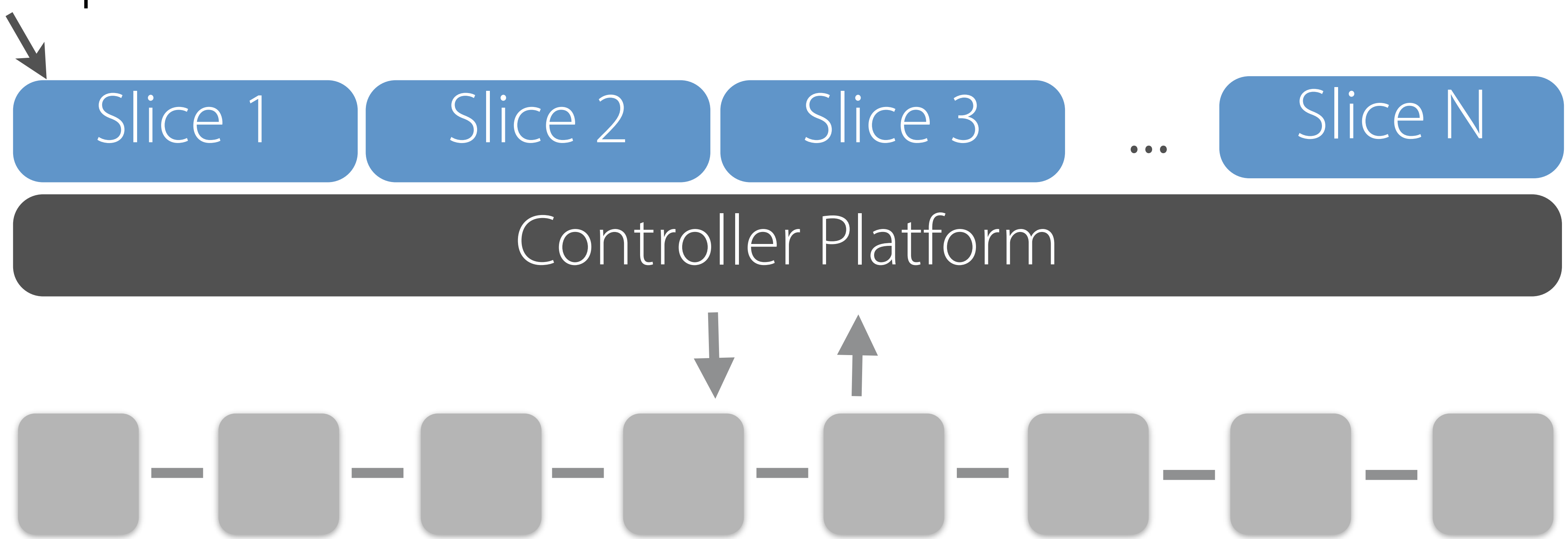
Application: Security

Each module controls a *different* portion of the traffic



Application: Security

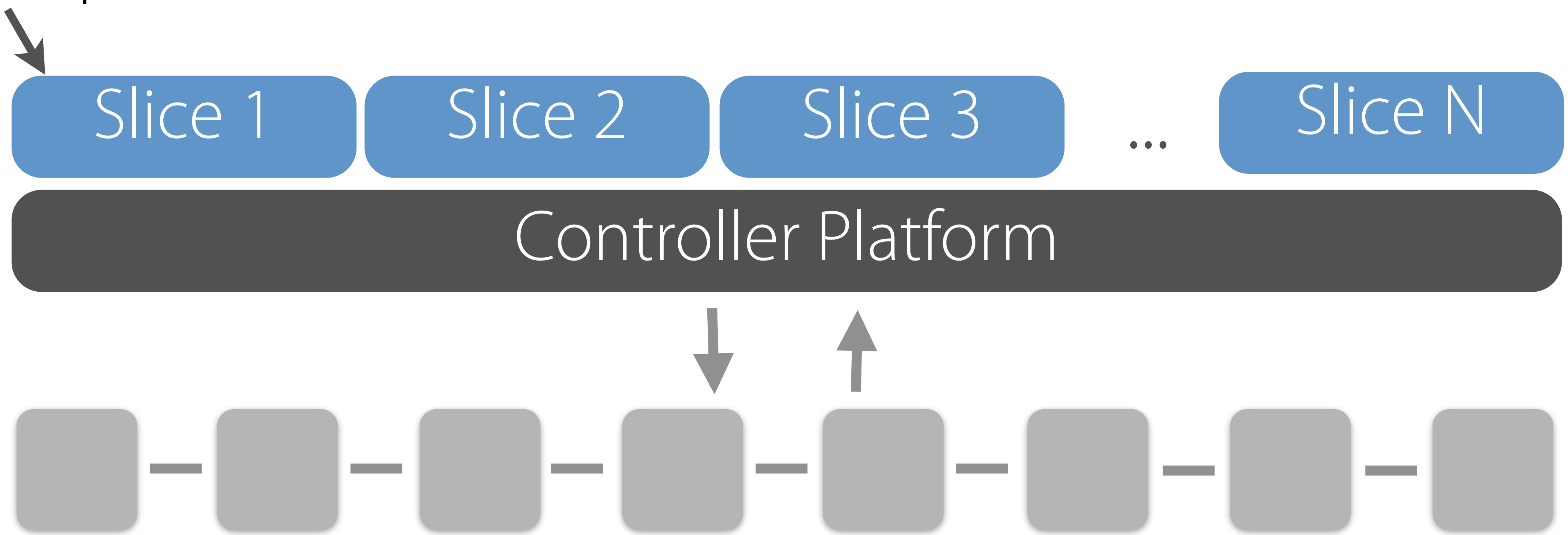
Each module controls a
different portion of the traffic



A *slice* is a lightweight abstraction for expressing isolated programs:

Application: Security

Each module controls a *different* portion of the traffic

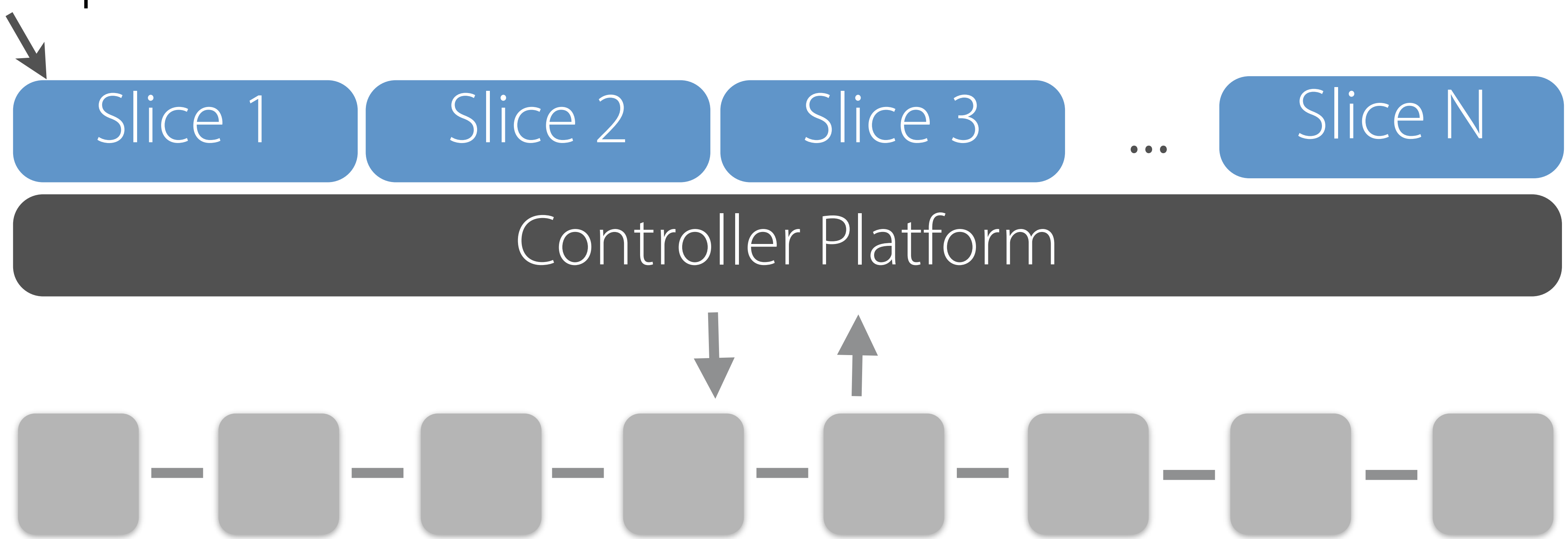


A *slice* is a lightweight abstraction for expressing isolated programs:

```
s::= {i} p {e}
```

Application: Security

Each module controls a
different portion of the traffic



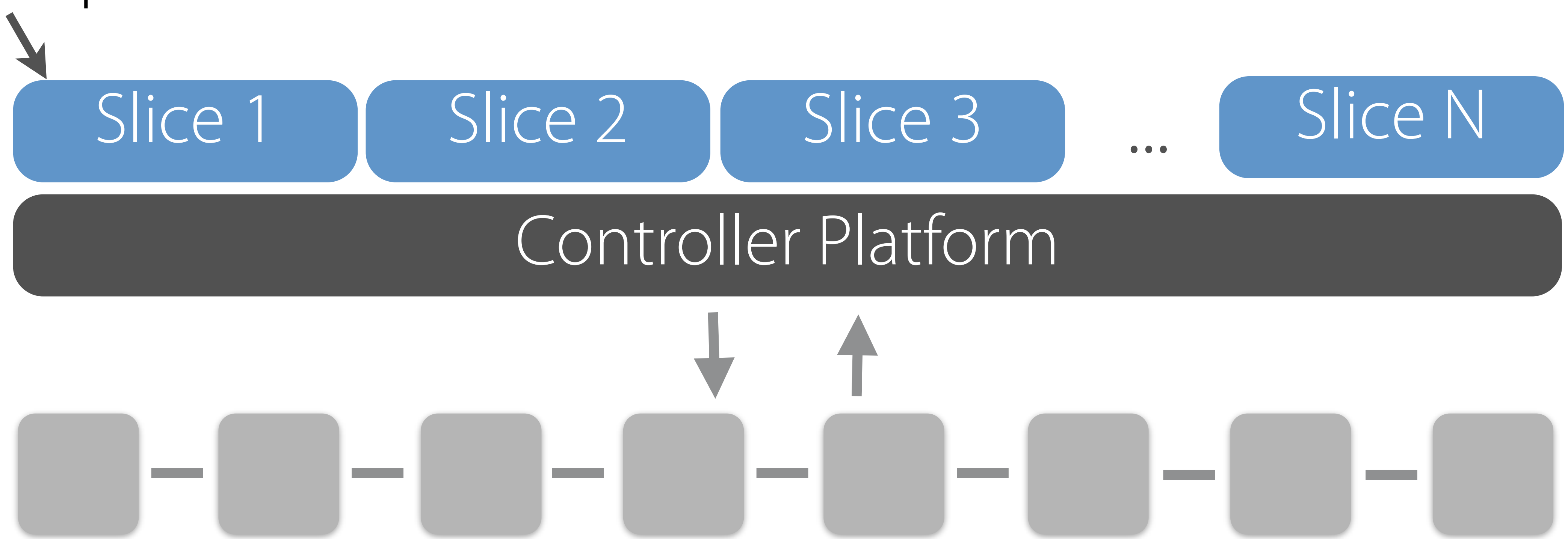
A *slice* is a lightweight abstraction for expressing isolated programs:

```
s::= {i} p {e}
```

Slices can be desugared into NetKAT by a simple translation

Application: Security

Each module controls a
different portion of the traffic



A *slice* is a lightweight abstraction for expressing isolated programs:

```
s::= {i} p {e}
```

Slices can be desugared into NetKAT by a simple translation

Theorem: `(s1 | s2;topo)* = (s1;topo)* | (s2;topo)*`

Ongoing Work

- NetKAT co-algebraically
  - Brzozowski derivative (DFA)
  - Antimirov partial derivative (NFA)
  - Algorithms for deciding equivalence
- Global compilation
  - Simulate automata state using header bits
  - Optimizations to reduce state space
  - Useful for compiling virtual paths, trees, etc.
- Non-deterministic NetKAT
  - Fault-tolerance
  - In-network load-balancing
- Verification tools
  - Z3-based backend
  - Automata-based backend (in progress)

Thank you!

NetKAT collaborators

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

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)

https://github.com/frenetic-lang/frenetic
https://github.com/frenetic-lang/pyretic

frenetic

Completeness

Syntax

$$\text{Atoms} \quad \alpha, \beta \triangleq f_1 = n_1; \cdots; f_k = n_k$$

$$\text{Assignments} \quad \pi \triangleq f_1 \leftarrow n_1; \cdots; f_k \leftarrow n_k$$

$$\text{Join-irreducibles} \quad x, y \in At; (P; \mathsf{dup})^*; P$$

Simplified axioms for At and P

$$\pi \equiv \pi; \alpha_\pi \quad \alpha; \mathsf{dup} \equiv \mathsf{dup}; \alpha \quad \sum_\alpha \alpha \equiv \mathsf{id},$$

$$\alpha \equiv \alpha; \pi_\alpha \quad \pi; \pi' \equiv \pi' \quad \alpha; \beta \equiv \mathsf{drop}, \; \alpha \neq \beta$$

Join-irreducible concatenation

$$\alpha; p; \pi \diamond \beta; q; \pi' = \begin{cases} \alpha; p; q; \pi' & \text{if } \beta = \alpha_\pi \\ \text{undefined} & \text{if } \beta \neq \alpha_\pi \end{cases}$$

$$A \diamond B = \{p \diamond q \mid p \in A, \; q \in B\} \subseteq I$$

Regular Interpretation: $R(p) \subseteq (P + At + \mathsf{dup})^*$

$$\begin{aligned} R(\pi) &= \{\pi\} \\ R(p+q) &= R(p) \cup R(q) \\ R(\alpha) &= \{\alpha\} \\ R(p;q) &= \{xy \mid x \in R(p), y \in R(q)\} \\ R(\mathsf{dup}) &= \{\mathsf{dup}\} \\ R(p^*) &= \bigcup_{n \geq 0} R(p^n) \end{aligned}$$

with $p^0 = 1$ and $p^{n+1} = p; p^n$

Language Model: $G(p) \subseteq I = At; (P; \mathsf{dup})^*; P$

$$\begin{aligned} G(\pi) &= \{\alpha; \pi \mid \alpha \in At\} \\ G(p+q) &= G(p) \cup G(q) \\ G(\alpha) &= \{\alpha; \pi_\alpha\} \\ G(p;q) &= G(p) \diamond G(q) \\ G(\mathsf{dup}) &= \{\alpha; \pi_\alpha; \mathsf{dup}; \pi_\alpha \mid \alpha \in At\} \\ G(p^*) &= \bigcup_{n \geq 0} G(p^n) \end{aligned}$$