Open Programmable Networks

Spiros Eliopoulos (Cornell)
Nate Foster (Cornell)
Arjun Guha (UMass Amherst)

Open Networking
USER GROUP

ONUG Fall 2014

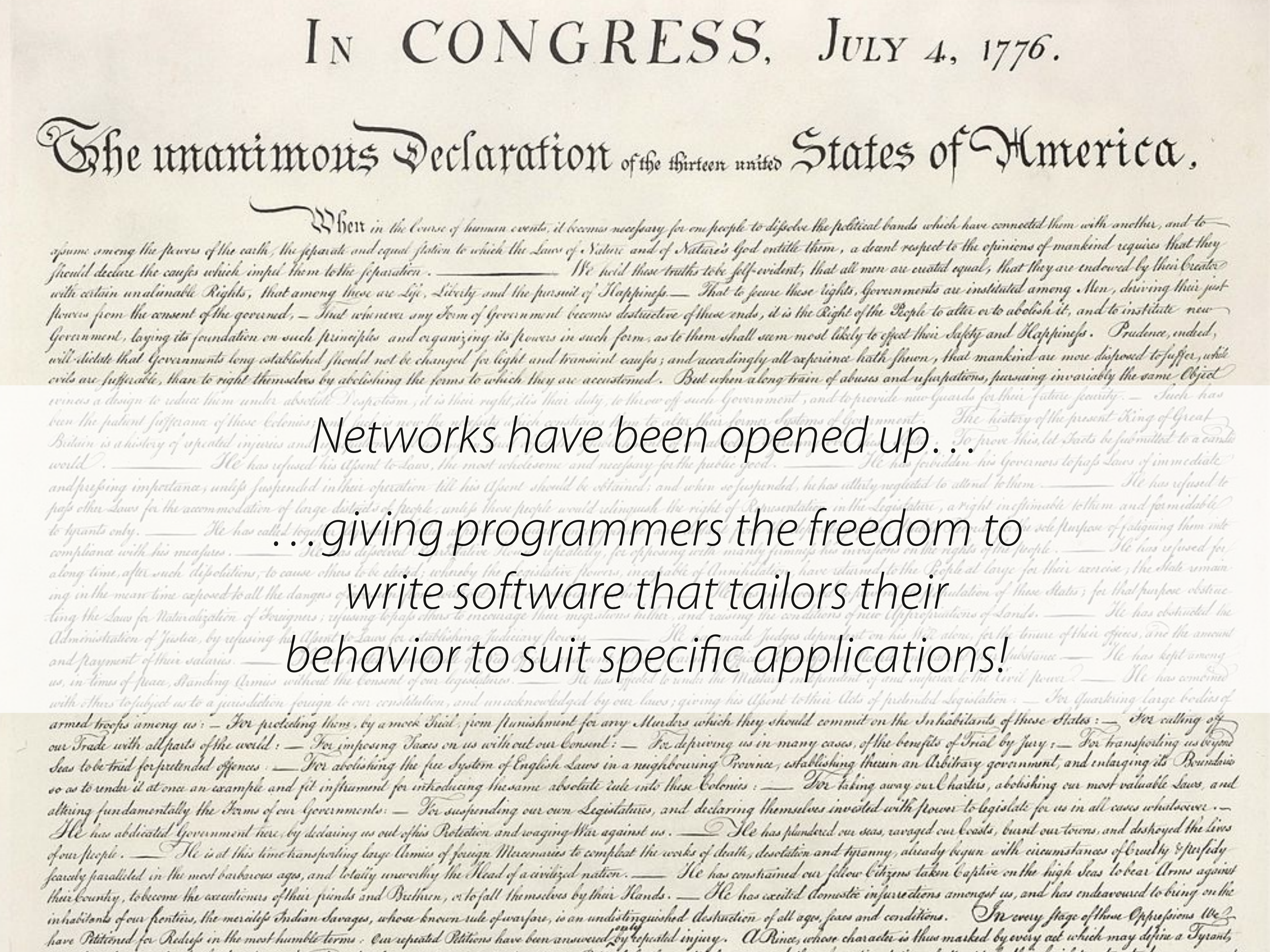
frenetic


We are at the start of a revolution!

IN CONGRESS, JULY 4, 1776.

The unanimous Declaration of the thirteen united States of America,

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation. —— We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the pursuit of Happiness. — That to secure these rights, Governments are instituted among Men, deriving their just powers from the consent of the governed, — That whenever any Form of Government becomes destructive of these ends, it is the Right of the People to alter or to abolish it, and to institute new Government, laying its foundation on such principles and organizing its powers in such form, as to them shall seem most likely to effect their Safety and Happiness. Prudence, indeed, will dictate that Governments long established should not be changed for light and transient causes; and accordingly all experience hath shewn, that mankind are more disposed to suffer, while evils are sufferable, than to right themselves by abolishing the forms to which they are accustomed. But when a long train of abuses and usurpations, pursuing invariably the same Object evinces a design to reduce them under absolute Despotism, it is their right, it is their duty, to throw off such Government, and to provide new Guards for their future security. — Such has been the patient sufferance of these Colonies; and such is now the necessity which constrains them to alter their former Systems of Government. The history of the present King of Great Britain is a history of repeated injuries and usurpations, all having in direct object the establishment of an absolute Tyranny over these States. To prove this, let Facts be submitted to a candid world. —— He has refused his Assent to Laws, the most wholesome and necessary for the public good. —— He has forbidden his Governors to pass Laws of immediate and pressing importance, unless suspended in their operation till his Assent should be obtained; and when so suspended, he has utterly neglected to attend to them. —— He has refused to pass other Laws for the accommodation of large districts of people, unless those people would relinquish the right of Representation in the Legislature, a right inestimable to them and formidable to tyrants only. —— He has called together legislative bodies at places unusual, uncomfortable, and distant from the depository of their public Records, for the sole purpose of fatiguing them into compliance with his measures. —— He has dissolved Representative Houses repeatedly, for opposing with manly firmness his invasions on the rights of the people. —— He has refused for a long time, after such dissolutions, to cause others to be elected; whereby the Legislative powers, incapable of Annihilation, have returned to the People at large for their exercise; the State remaining in the mean time exposed to all the dangers of invasion from without, and convulsions within. —— He has endeavoured to prevent the population of these States; for that purpose obstructing the Laws for Naturalization of Foreigners; refusing to pass others to encourage their migrations hither, and raising the conditions of new Appropriations of Lands. —— He has obstructed the Administration of Justice, by refusing his Assent to Laws for establishing Judiciary powers. —— He has made Judges dependent on his Will alone, for the tenure of their offices, and the amount and payment of their salaries. —— He has erected a multitude of New Offices, and sent hither swarms of Officers to harrass our people, and eat out their substance. —— He has kept among us, in times of peace, Standing Armies without the Consent of our legislatures. —— He has affected to render the Military independent of and superior to the Civil power. —— He has combined with others to subject us to a jurisdiction foreign to our constitution, and unacknowledged by our laws; giving his Assent to their Acts of pretended Legislation: — For Quartering large bodies of armed troops among us: — For protecting them, by a mock Trial, from punishment for any Murders which they should commit on the Inhabitants of these States: — For cutting off our Trade with all parts of the world: — For imposing Taxes on us without our Consent: — For depriving us in many cases, of the benefits of Trial by Jury: — For transporting us beyond Seas to be tried for pretended offences: — For abolishing the free System of English Laws in a neighbouring Province, establishing therein an Arbitrary government, and enlarging its Boundaries so as to render it at once an example and fit instrument for introducing the same absolute rule into these Colonies: — For taking away our Charters, abolishing our most valuable Laws, and altering fundamentally the Forms of our Governments: — For suspending our own Legislatures, and declaring themselves invested with power to legislate for us in all cases whatsoever. — He has abdicated Government here, by declaring us out of his Protection and waging War against us. —— He has plundered our seas, ravaged our Coasts, burnt our towns, and destroyed the lives of our people. —— He is at this time transporting large Armies of foreign Mercenaries to compleat the works of death, desolation and tyranny, already begun with circumstances of Cruelty & perfidy scarcely paralleled in the most barbarous ages, and totally unworthy the Head of a civilized nation. —— He has constrained our fellow Citizens taken Captive on the high Seas to bear Arms against their Country, to become the executioners of their friends and Brethren, or to fall themselves by their Hands. —— He has excited domestic insurrections amongst us, and has endeavoured to bring on the inhabitants of our frontiers, the merciless Indian Savages, whose known rule of warfare, is an undistinguished destruction of all ages, sexes and conditions. In every stage of these Oppressions We have Petitioned for Redress in the most humble terms: Our repeated Petitions have been answered only by repeated injury. A Prince, whose character is thus marked by every act which may define a Tyrant,



Networks have been opened up…

…giving programmers the freedom to write software that tailors their behavior to suit specific applications!

Open Networking Successes

Data Center Virtualization

- Write programs against virtual topologies
- Controller maps virtual programs to physical network

Traffic Monitoring

- Declare continuous traffic queries
- Controller polls counters and aggregates results

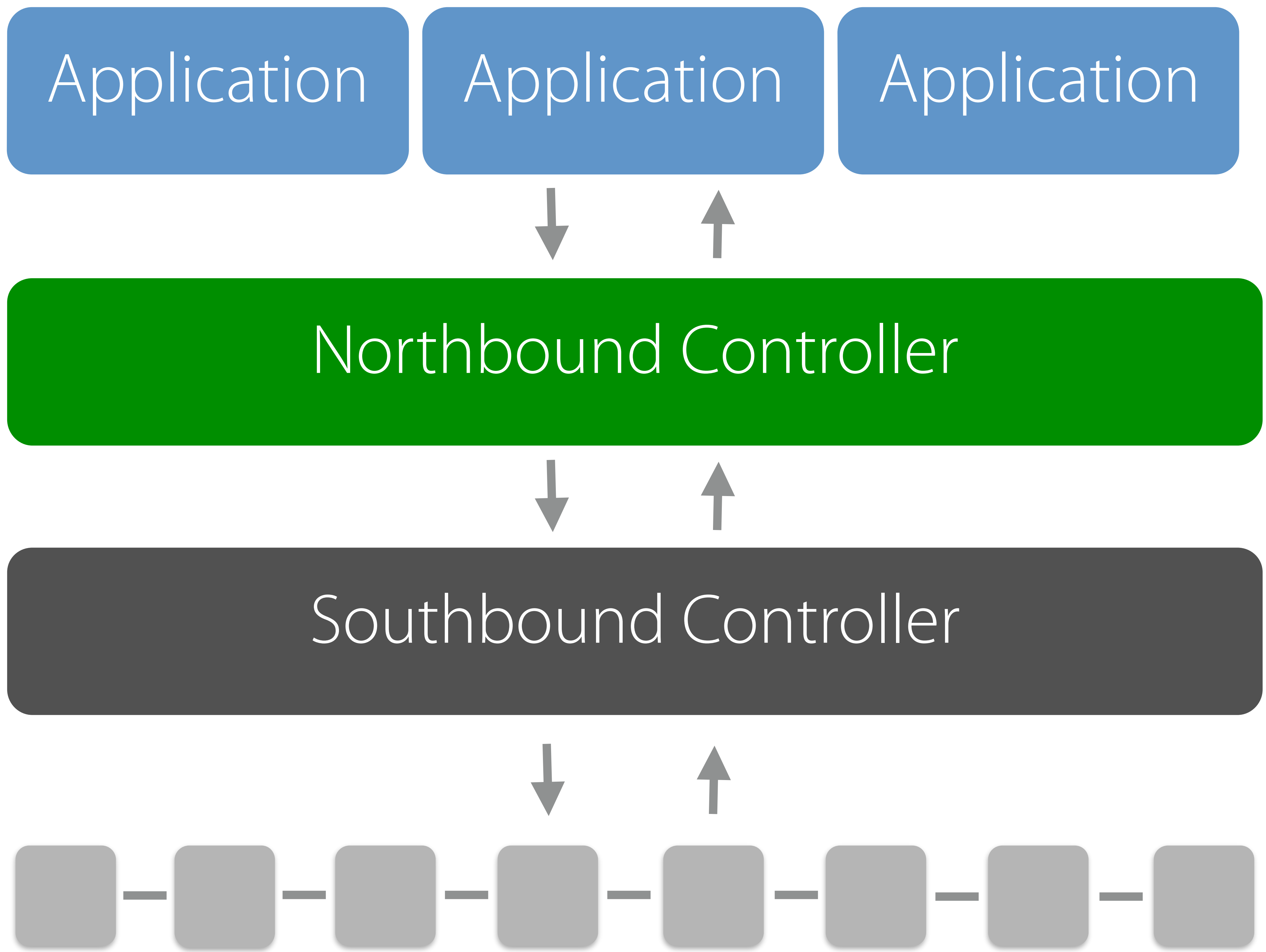
Verification and Debugging

- Specify behavior using high-level properties
- Controller generates code to enforce key invariants

Traffic Engineering

- Optimize bandwidth according to natural criteria
- Controller provisions paths using constraint solver

Open Networking Architecture




https://www.flickr.com/photos/el_ramon/

Southbound Interfaces

There are now many ways to manage network device configurations programmatically

- NetConf
- OpenFlow
- OVS
- P4
- SNMP
- YANG
- etc.

These interfaces, which are rapidly maturing, provide a solid foundation for network programming

https://www.flickr.com/photos/el_ramon/

Northbound Interfaces

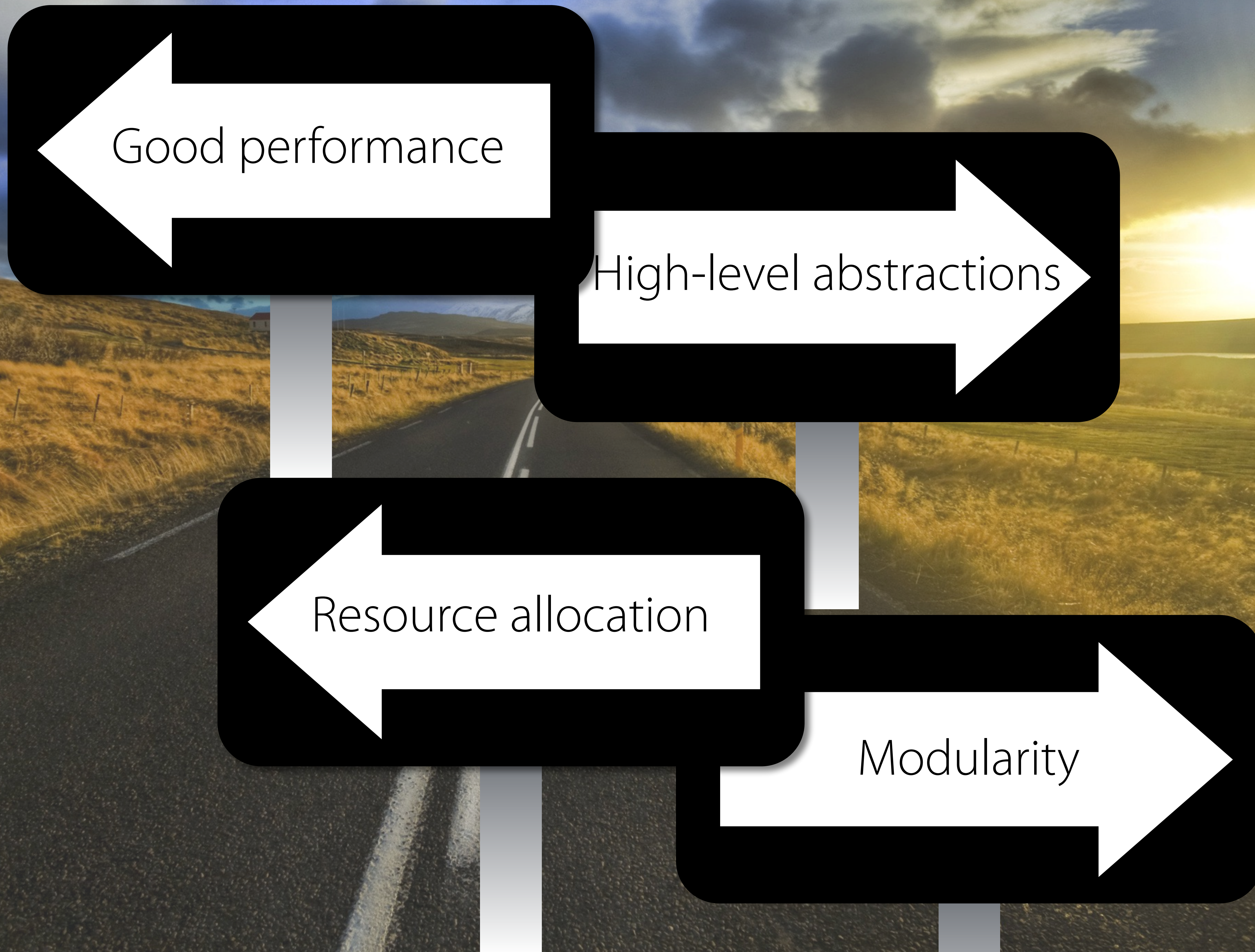
But on the northbound side… the situation is bleak

Current controllers provide a variety of abstractions:

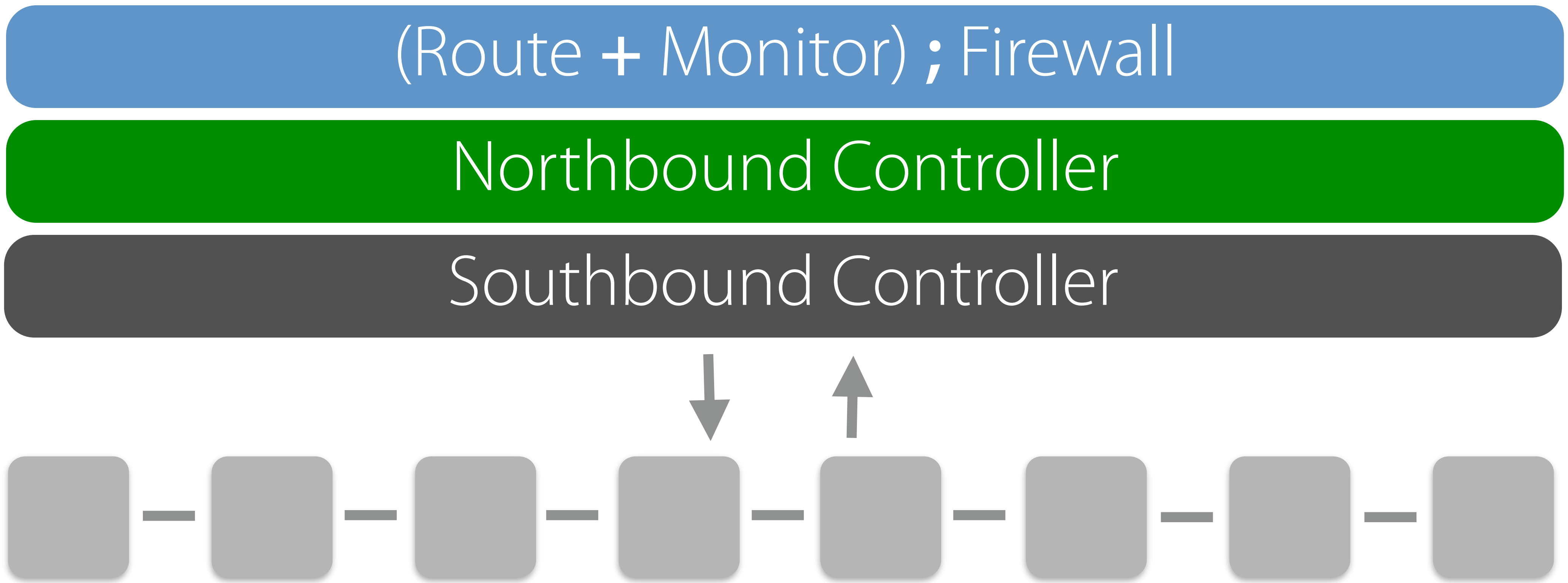
- Device abstraction layers
- Isolated slices
- Virtual networks
- QoS provisioning
- NFV service chaining
- Custom services (discovery, firewall, etc.)

But the development of these abstractions has been ad hoc, driven more by the needs of particular applications than by fundamental principles

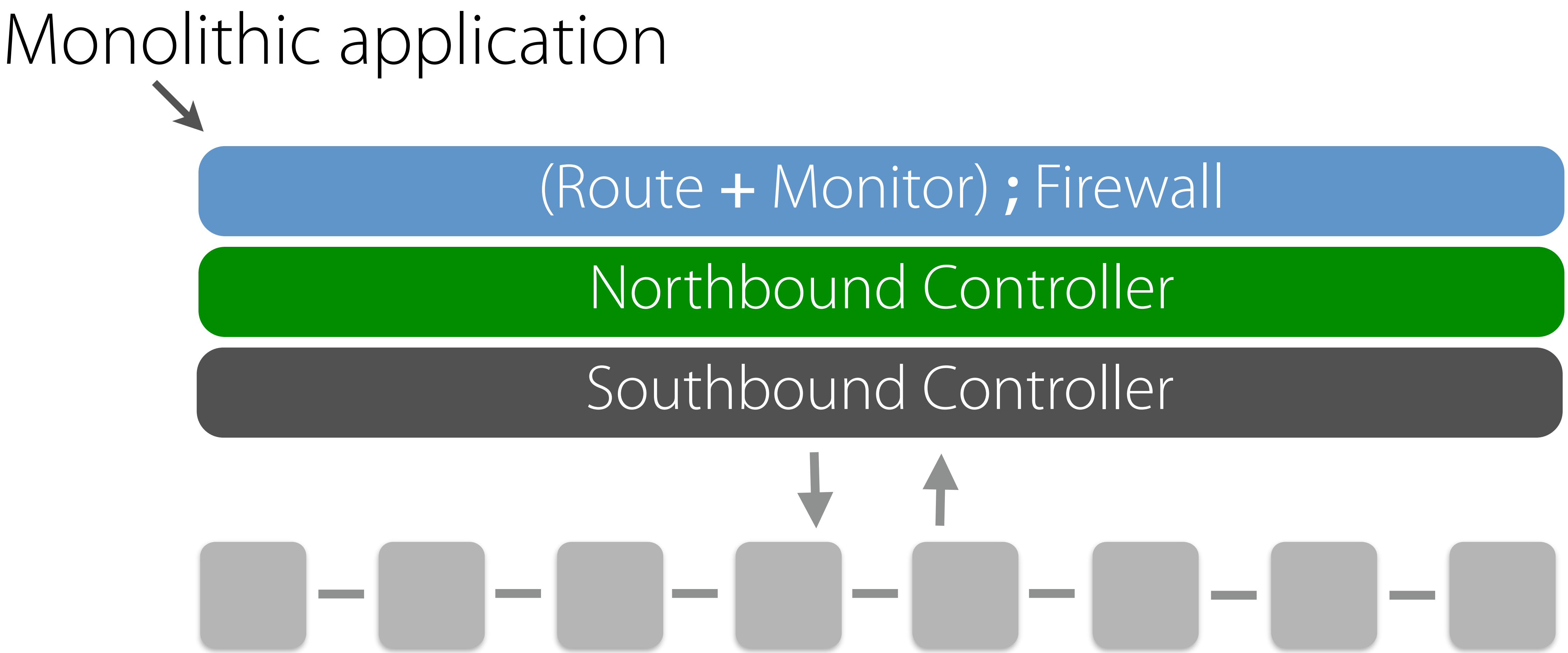
Northbound Interface Design



Modular Composition

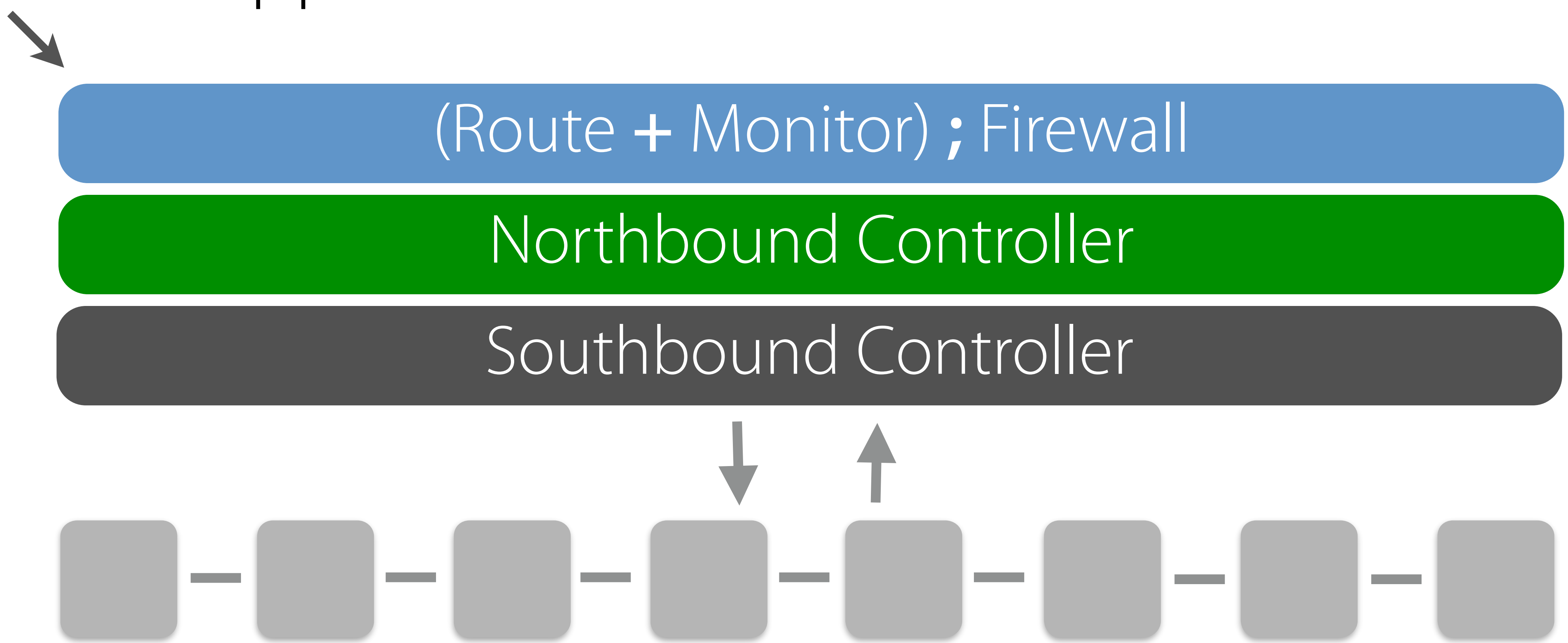


Modular Composition



Modular Composition

Monolithic application



This style of programming complicates:

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

Route

+

Monitor

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

Pattern	Actions
srcip=1.2.3.4	Count

Route

+

Monitor

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

Pattern	Actions
srcip=1.2.3.4	Count

Route + Monitor

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

Route + Monitor

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

Route + Monitor

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

;

Firewall

Pattern	Actions
tcpdst = 22	Drop
*	Fwd ?

Route + Monitor

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

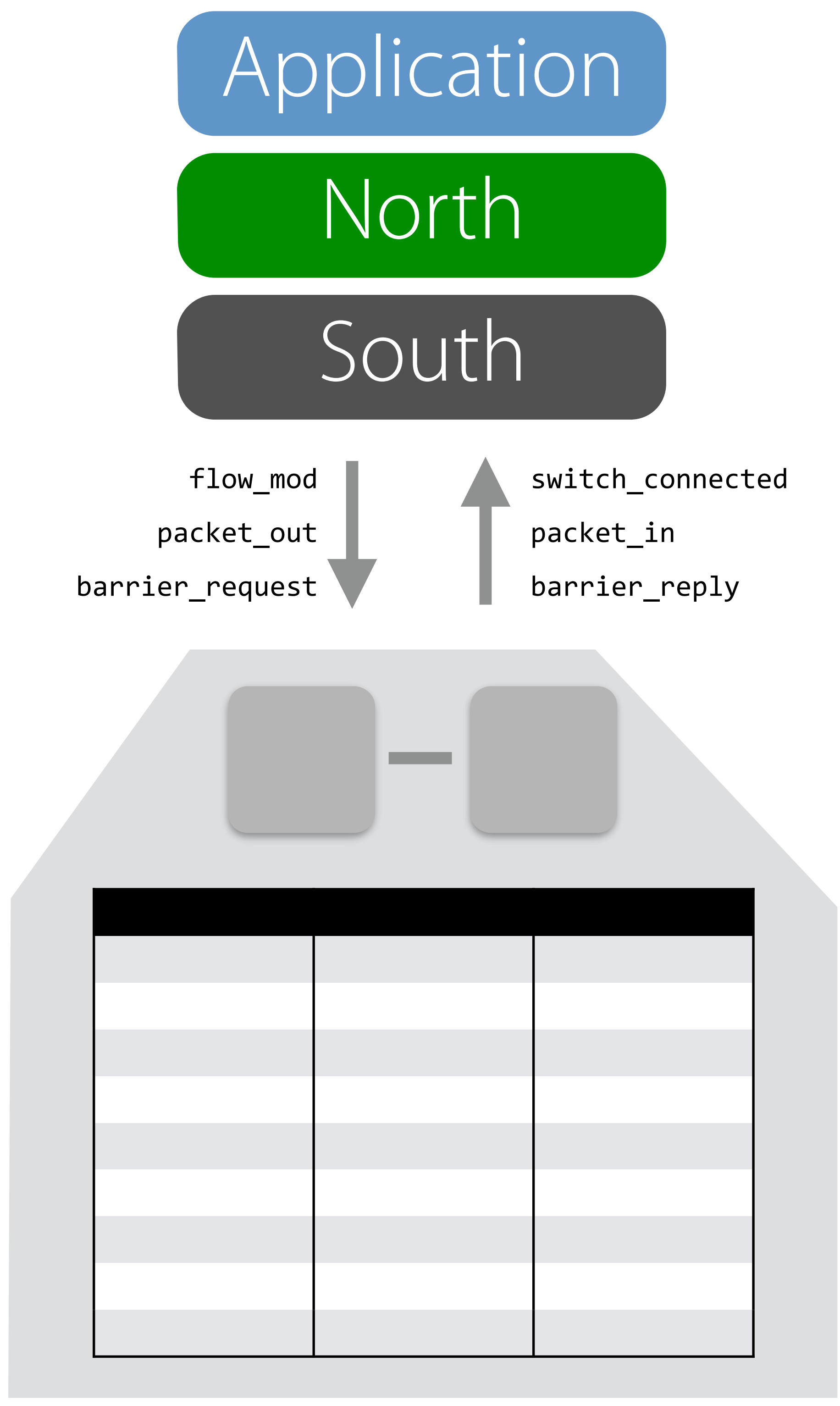
Machine Languages

Current APIs are derived from the underlying machine languages

Programmers must work in terms of low-level concepts such as:

- Flow tables
- Matches
- Priorities
- Timeouts
- Events

This approach complicates programs and reasoning



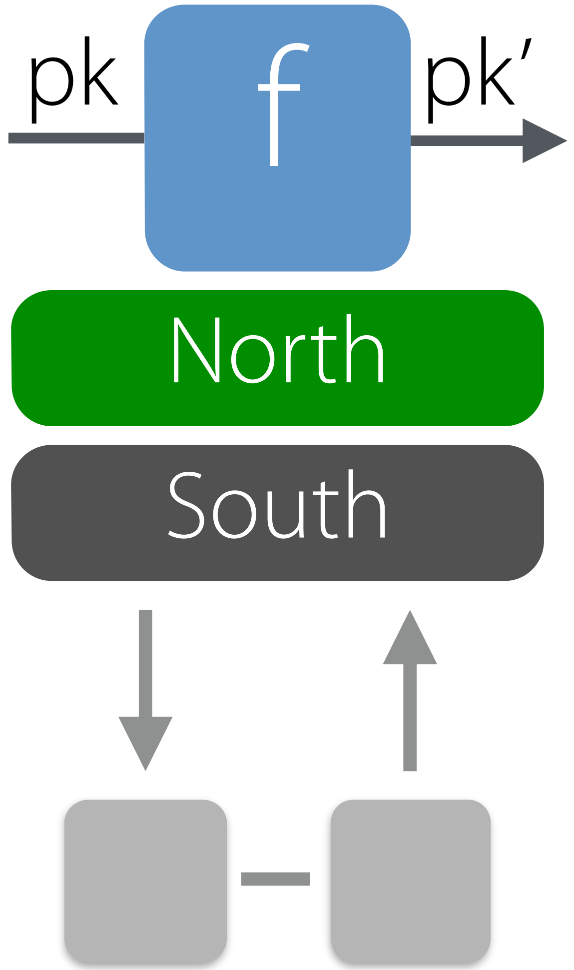
Programming Languages

Better would be to have APIs based on higher-level, more intuitive abstractions

Then, programmers could work in terms of natural concepts such as:

- Logical predicates
- Mathematical functions
- Network-wide paths
- Policy combinators
- Atomic transactions

which would streamline many programs and simplify reasoning



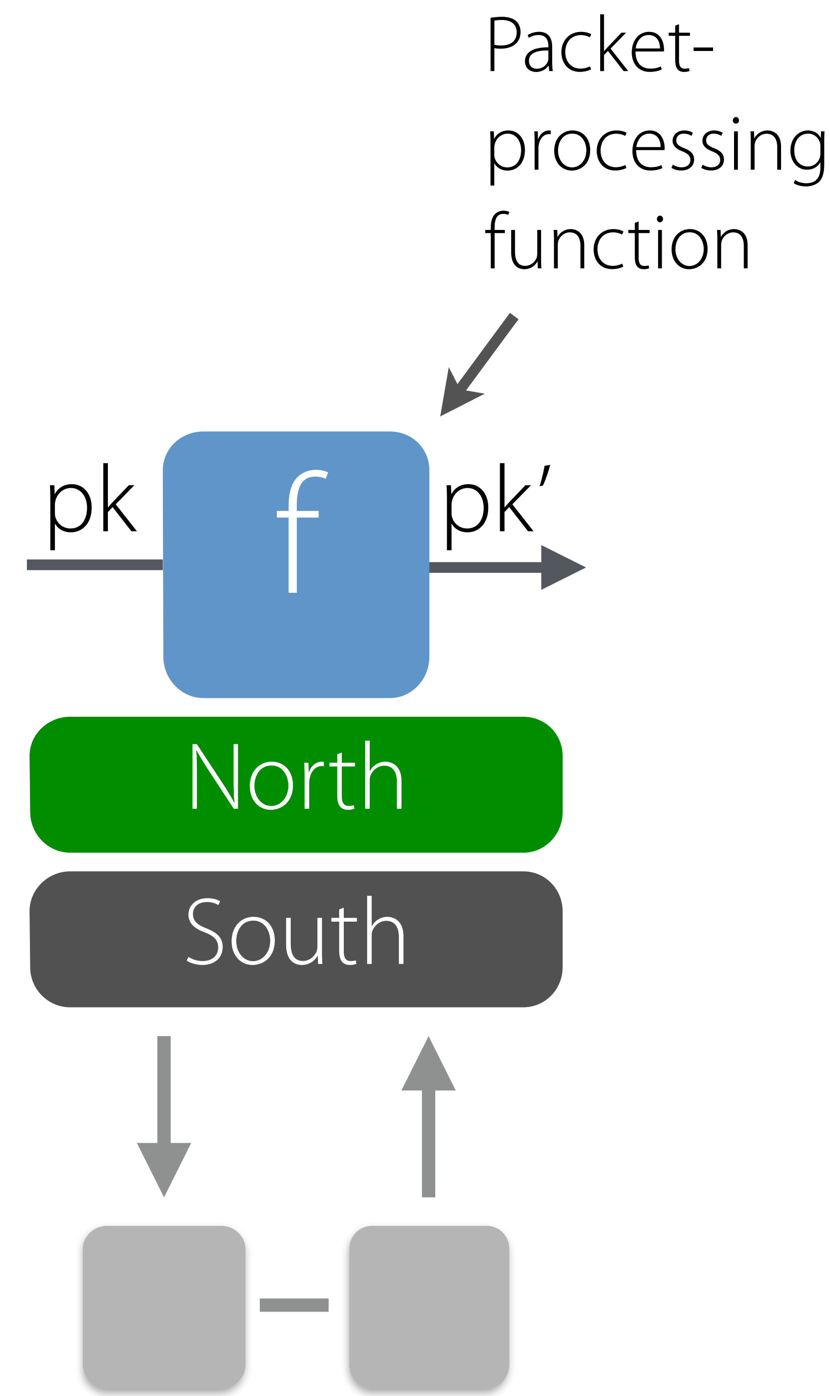
Programming Languages

Better would be to have APIs based on higher-level, more intuitive abstractions

Then, programmers could work in terms of natural concepts such as:

- Logical predicates
- Mathematical functions
- Network-wide paths
- Policy combinators
- Atomic transactions

which would streamline many programs and simplify reasoning



frenetic

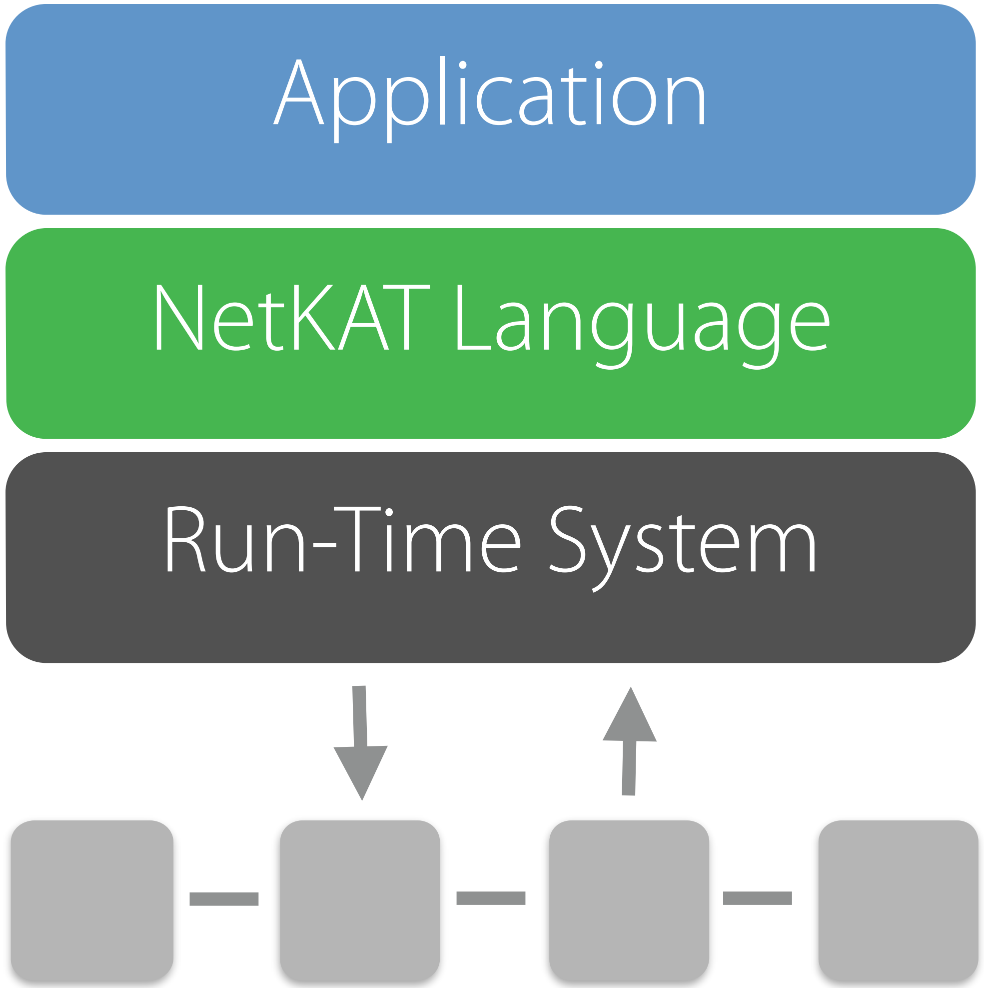
Our Vision

- Write network programs in a high-level language
- Generate efficient low-level code using a compiler
- Reason about network properties automatically

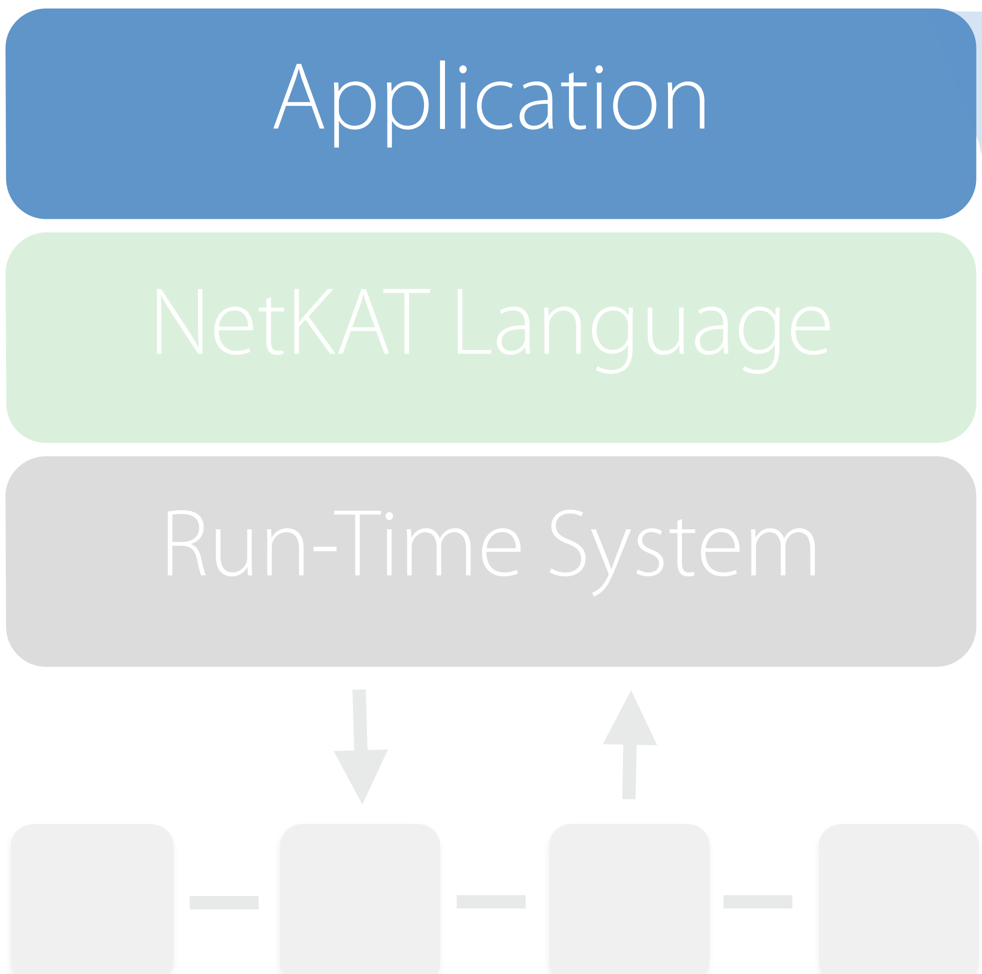
Main Results

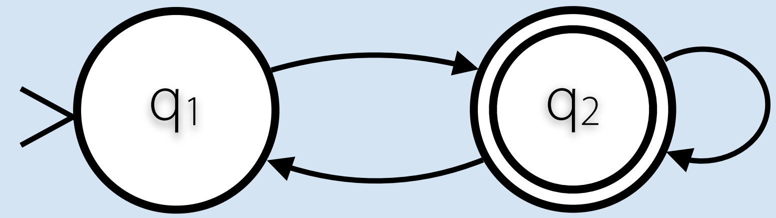
- Designed language based on an elegant and compositional foundation: packet-processing functions
- Built compilers and run-time systems that implement these languages efficiently on OpenFlow hardware
- Showed how to encode other high-level abstractions using functions (e.g., slicing, virtualization, QoS, etc.)

Frenetic Architecture



Frenetic Architecture

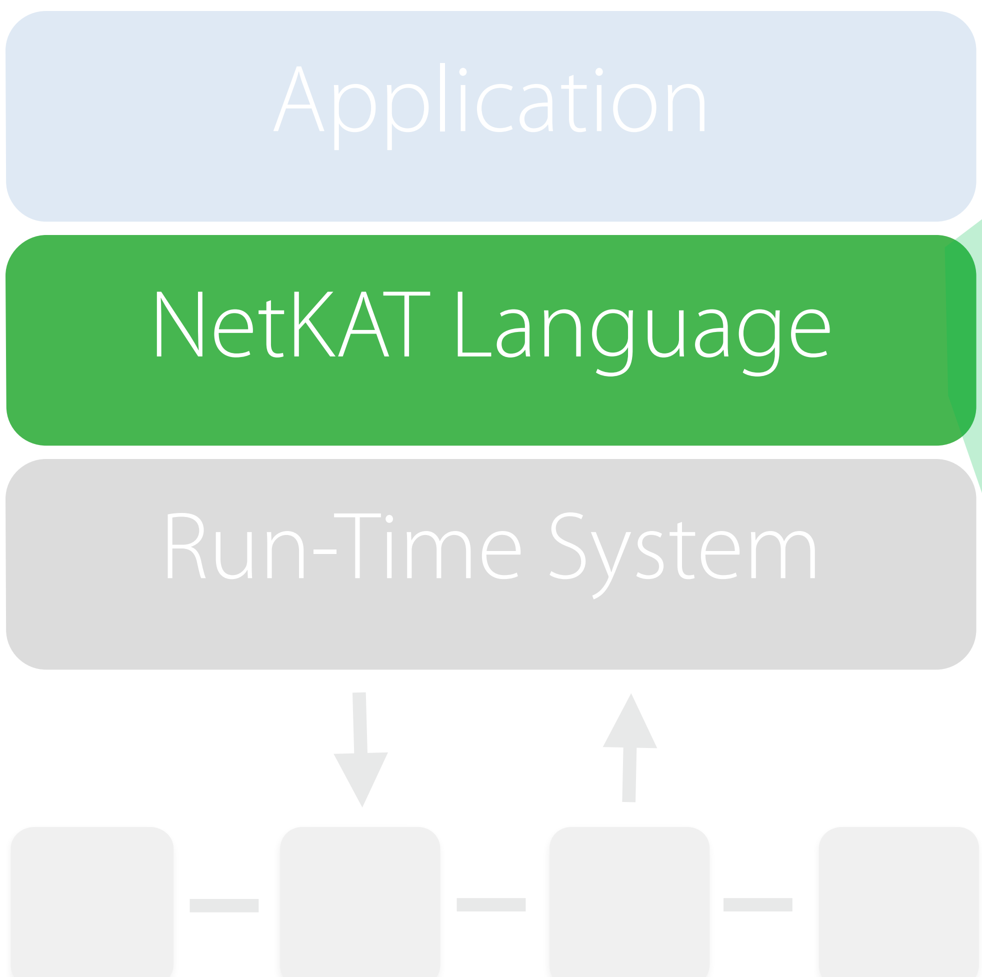




High-level application logic

Often expressed as a finite-state machine on network events (topology changes, new connections, etc.)

Frenetic Architecture

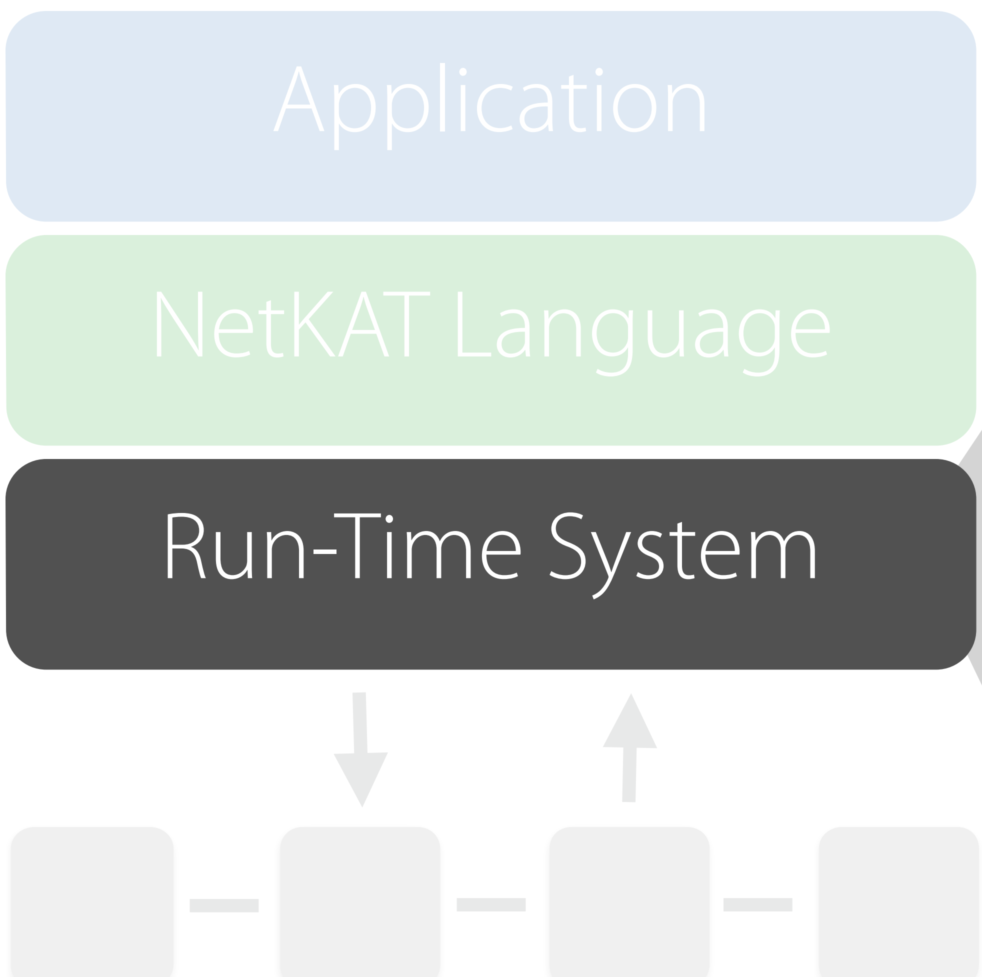


Pattern	Actions
srcip=1.2.3.4, tcpdst = 22	Count, Drop
srcip=1.2.3.4,	Forward 1, Count
srcip=1.2.3.4,	Forward 2, Count
srcip=1.2.3.4	Count
tcpdst = 22	Drop

Programs describe packet-processing functions

Compile to a collection of forwarding tables, one per switch in the network

Frenetic Architecture

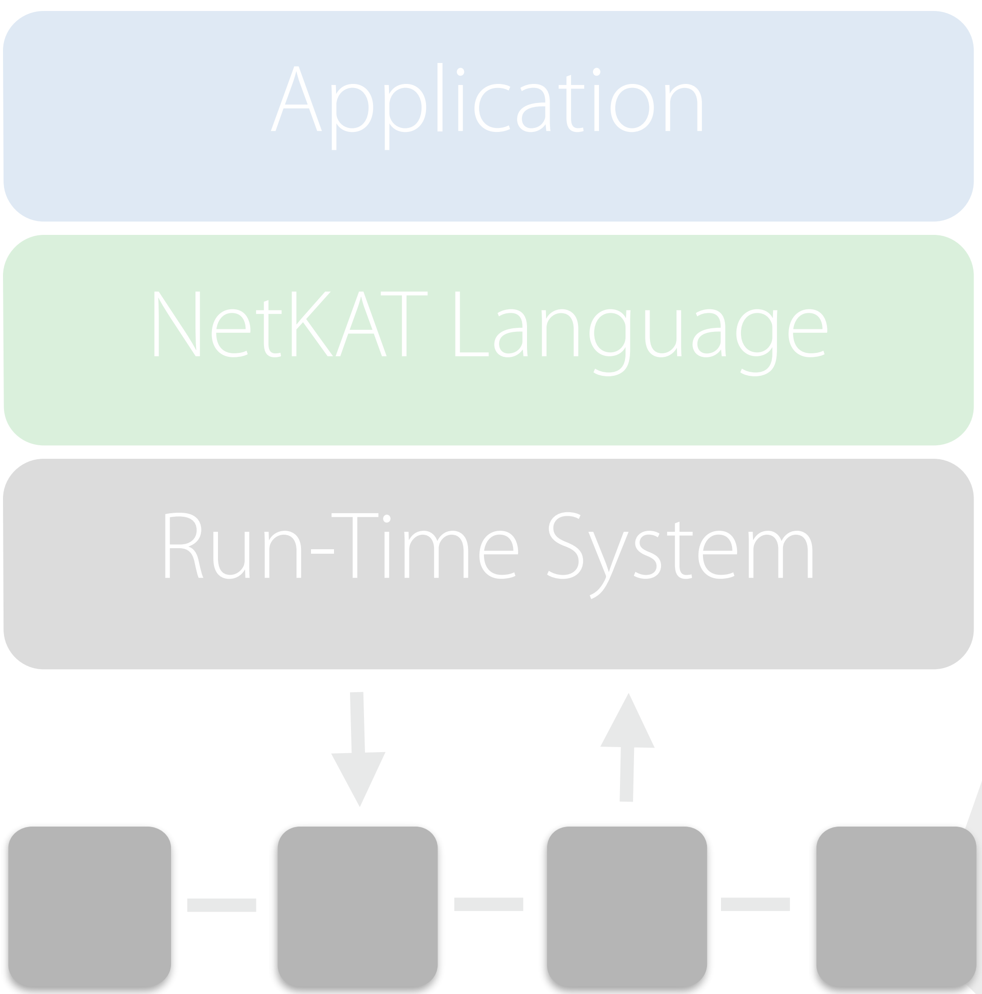



```
let swap_update_for (t : t) sw_id c_id new_table : unit Deferred.t =
   let max_priority = 65535 in
   let old_table = match SwitchMap.find t.edge sw_id with | Some ft -> ft | None -> [] in
   let (new_table, _) = List.fold new_table ~init:([], max_priority)
        ~f:(fun (acc,pri) x -> ((x,pri) :: acc, pri - 1)) in
    let new_table = List.rev new_table in
    let del_table = List.rev (flowtable_diff old_table new_table) in
    let to_flow_mod prio flow =
      M.FlowModMsg (SDN_OpenFlow0x01.from_flow prio flow) in
    let to_flow_del prio flow =
      M.FlowModMsg ({SDN_OpenFlow0x01.from_flow prio flow with command = DeleteStrictFlow}) in
   Deferred.List.iter new_table ~f:(fun (flow, prio) ->
      send t.ctl c_id (0l, to_flow_mod prio flow))
    >>= fun () -> Deferred.List.iter del_table ~f:(fun (flow, prio) ->
      send t.ctl c_id (0l, to_flow_del prio flow))
    >>| fun () -> t.edge <- SwitchMap.add t.edge sw_id new_table
```

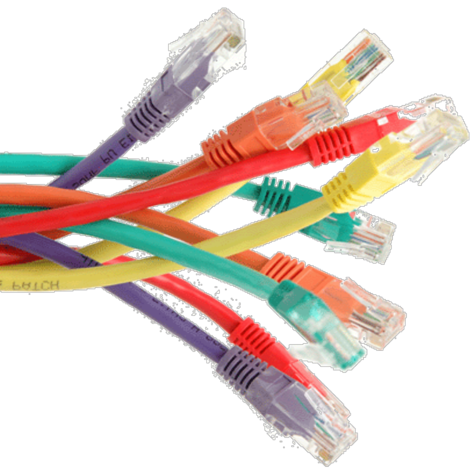
Code that manages the rules installed on switches

Translate configuration updates into sequences of OpenFlow instructions

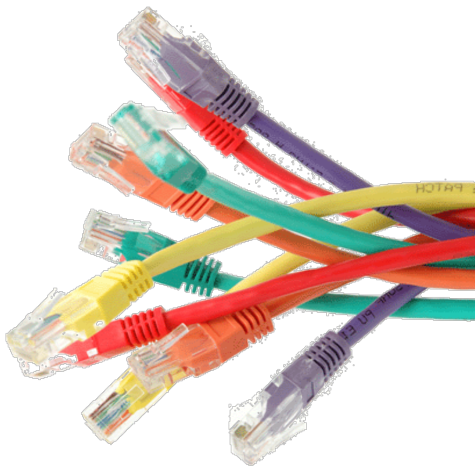
Frenetic Architecture



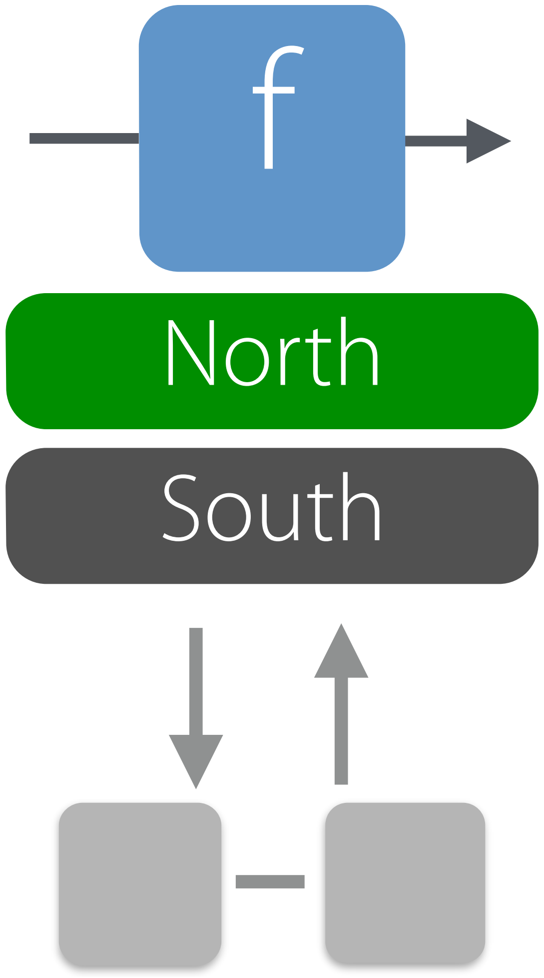
Forwarding elements that implement packet-processing functionality efficiently in hardware



NetKAT Language



Semantic Design

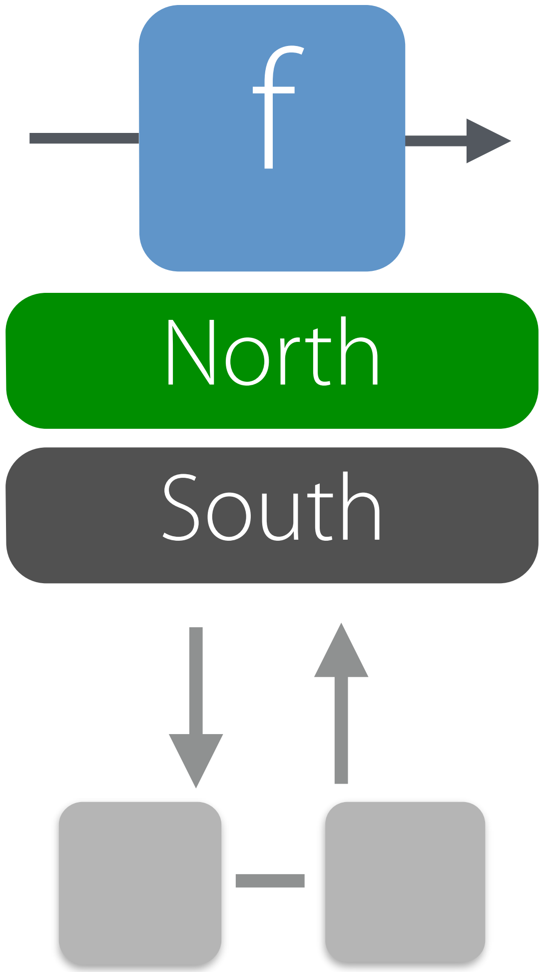


Semantic Design

Packets

A record comprising the location of the packet and a collection of header values

```
{switch=n₁, port=n₂, ethSrc=n₃,…}
```



Semantic Design

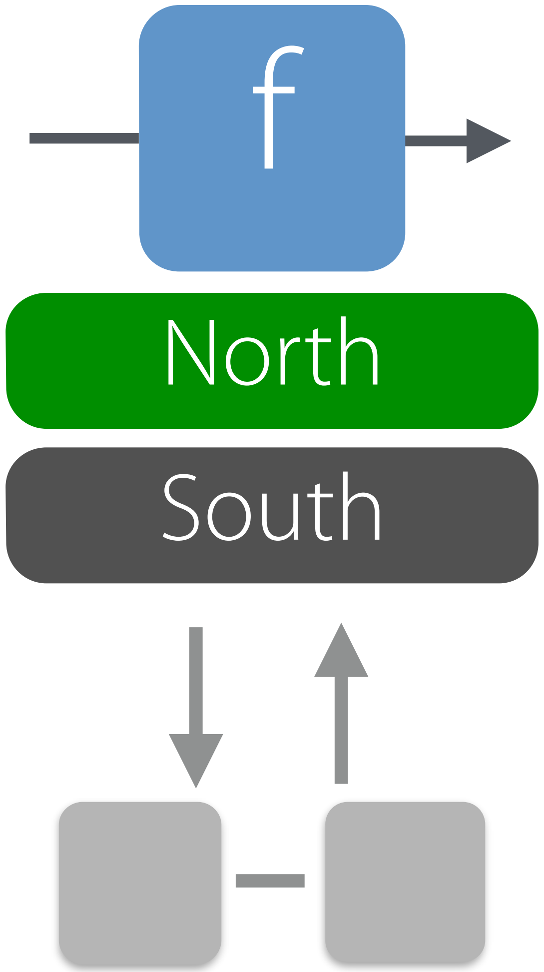
Packets

A record comprising the location of the packet and a collection of header values

`{switch=n₁, port=n₂, ethSrc=n₃,…}`

Policies

Denotes total functions on packets (really packet histories)



Semantic Design

Packets

A record comprising the location of the packet and a collection of header values

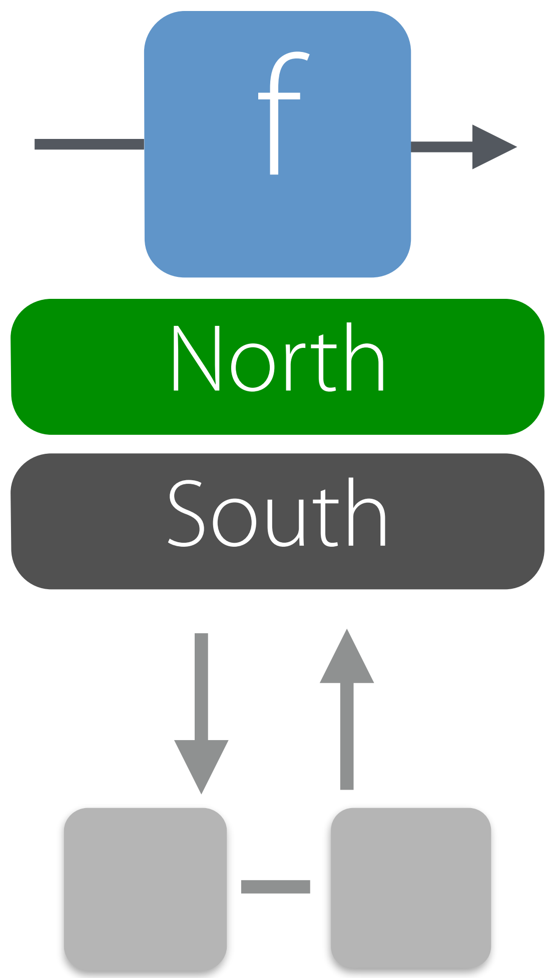
`{switch=n₁, port=n₂, ethSrc=n₃,…}`

Policies

Denotes total functions on packets (really packet histories)

Predicates

Restrict the behavior of a program to a particular set of packets using predicates



Semantic Design

Packets

A record comprising the location of the packet and a collection of header values

```
{switch=n₁, port=n₂, ethSrc=n₃,…}
```

Policies

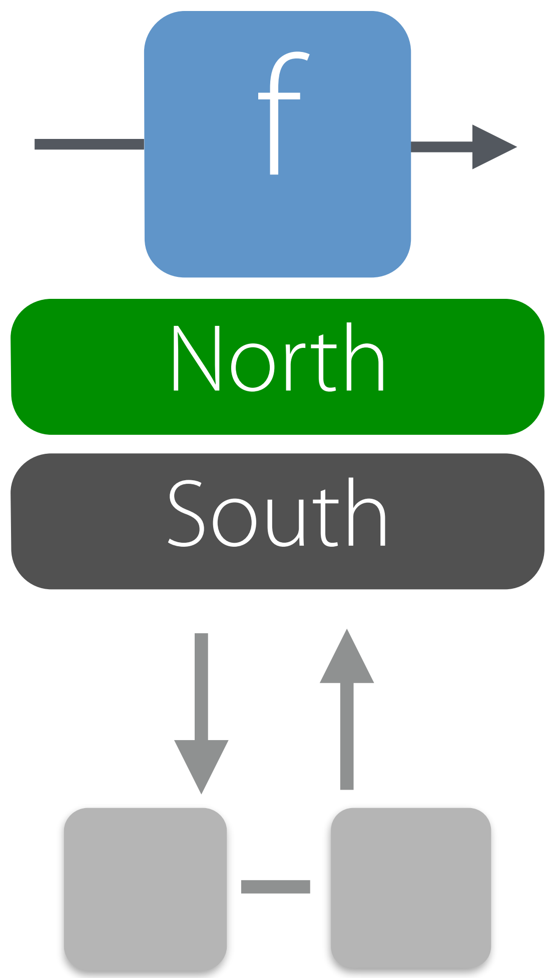
Denotes total functions on packets (really packet histories)

Predicates

Restrict the behavior of a program to a particular set of packets using predicates

Combinators

Combine smaller programs into bigger ones via natural mathematical operations



Semantic Design

Packets

A record comprising the location of the packet and a collection of header values

```
{switch=n₁, port=n₂, ethSrc=n₃,…}
```

Policies

packet histories)

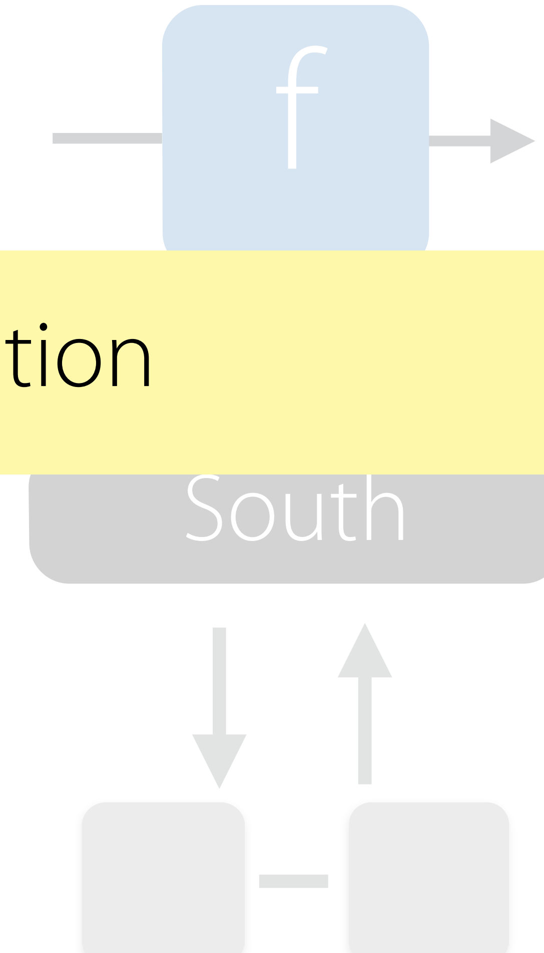
Functional "see every packet" abstraction

Predicates

Restrict the behavior of a program to a particular set of packets using predicates

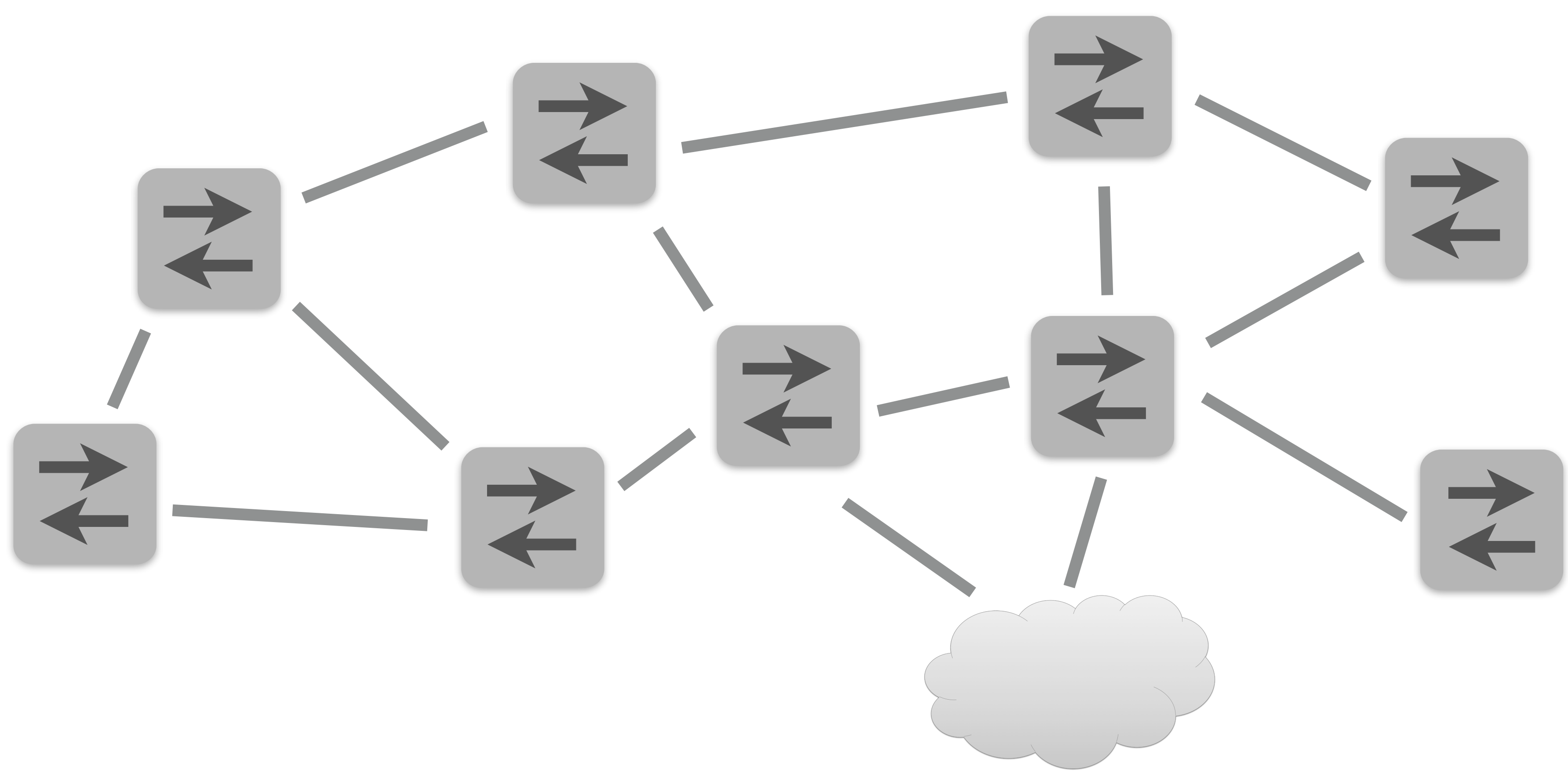
Combinators

Combine smaller programs into bigger ones via natural mathematical operations



Primitive Design

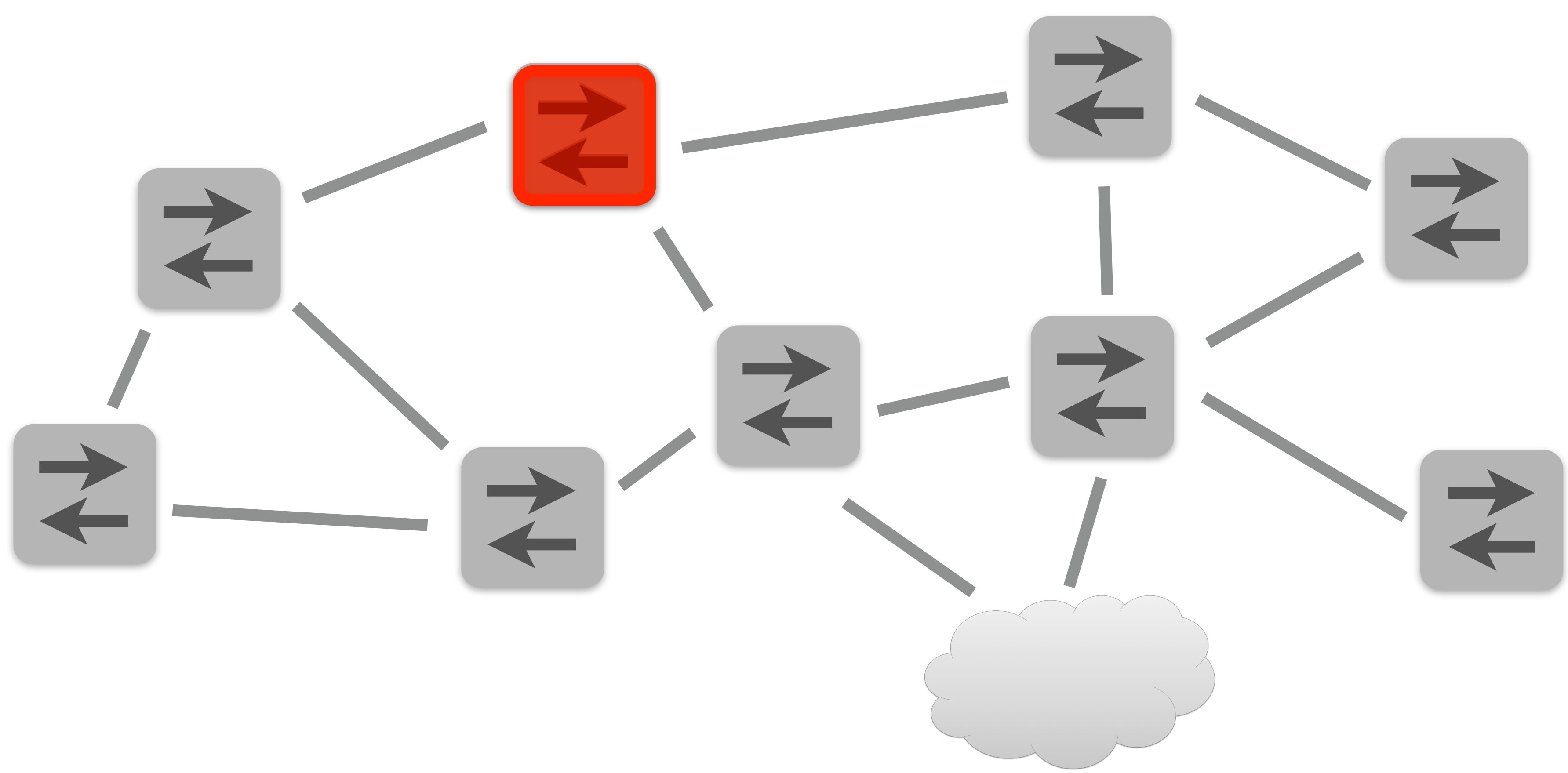
What constructs should an SDN language provide?



Primitive Design

What constructs should an SDN language provide?

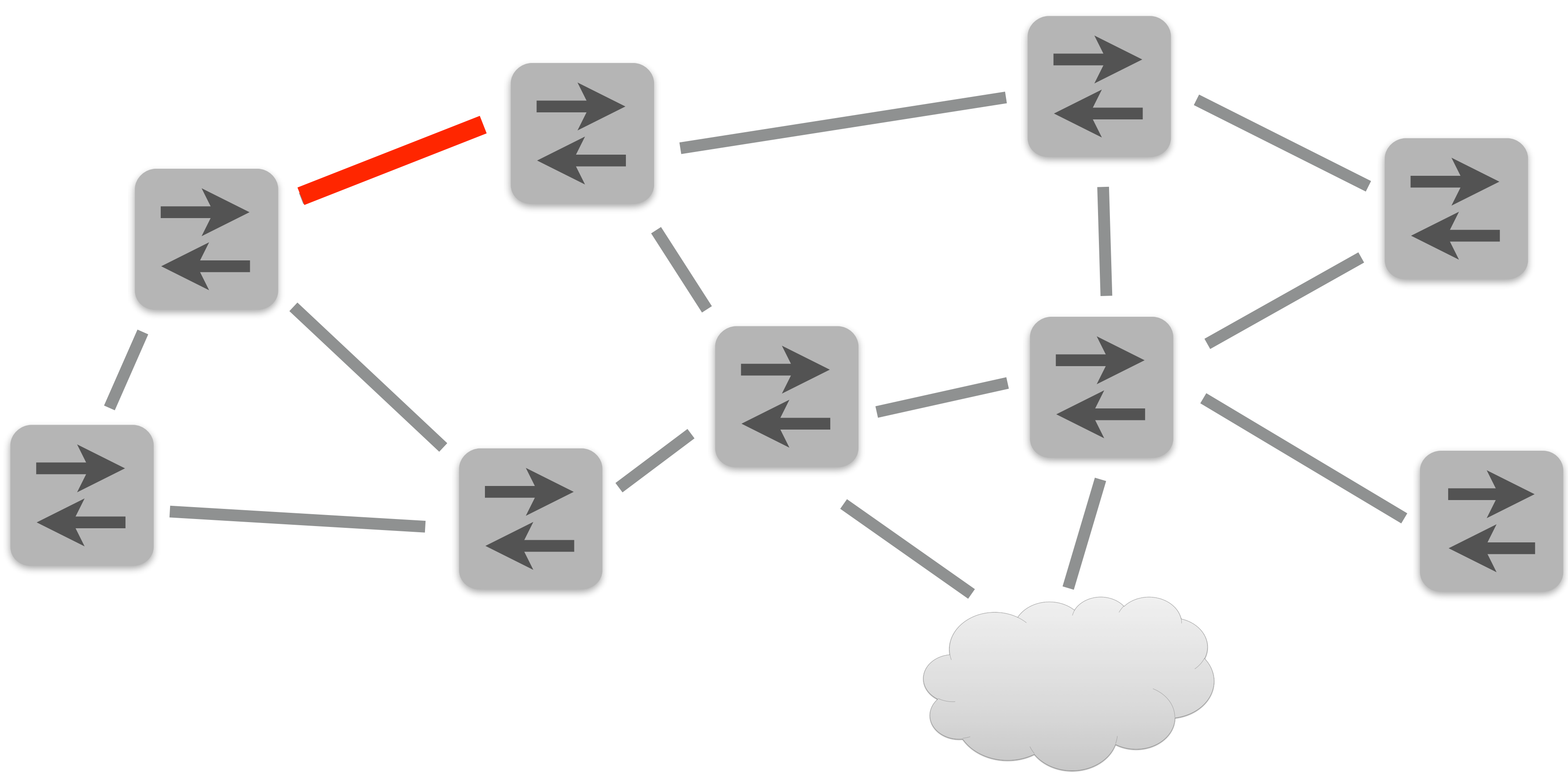
- Packet predicates
- Packet transformations



Primitive Design

What constructs should an SDN language provide?

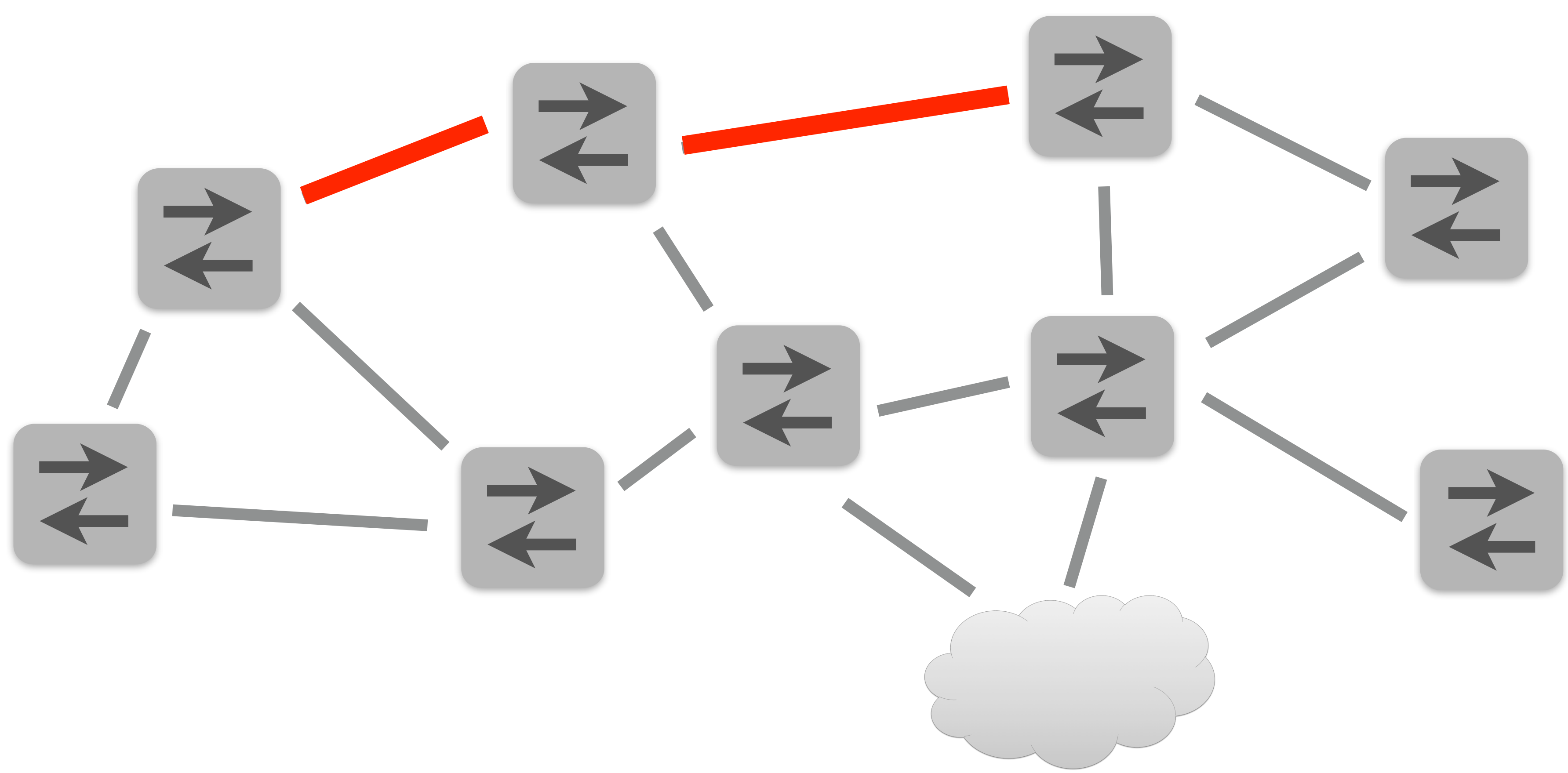
- Packet predicates
- Packet transformations
- Path construction



Primitive Design

What constructs should an SDN language provide?

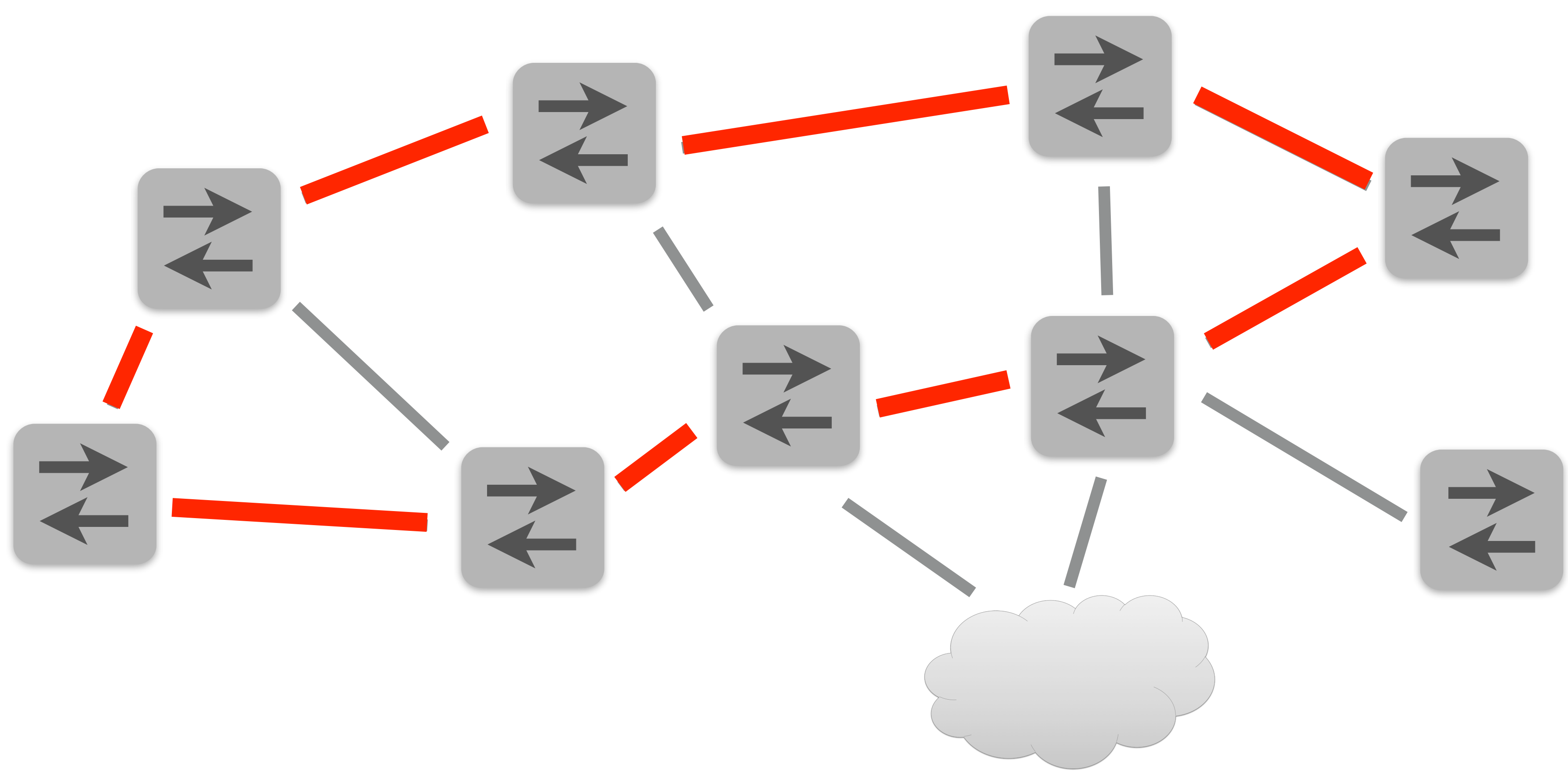
- Packet predicates
- Packet transformations
- Path construction
- Path concatenation



Primitive Design

What constructs should an SDN language provide?

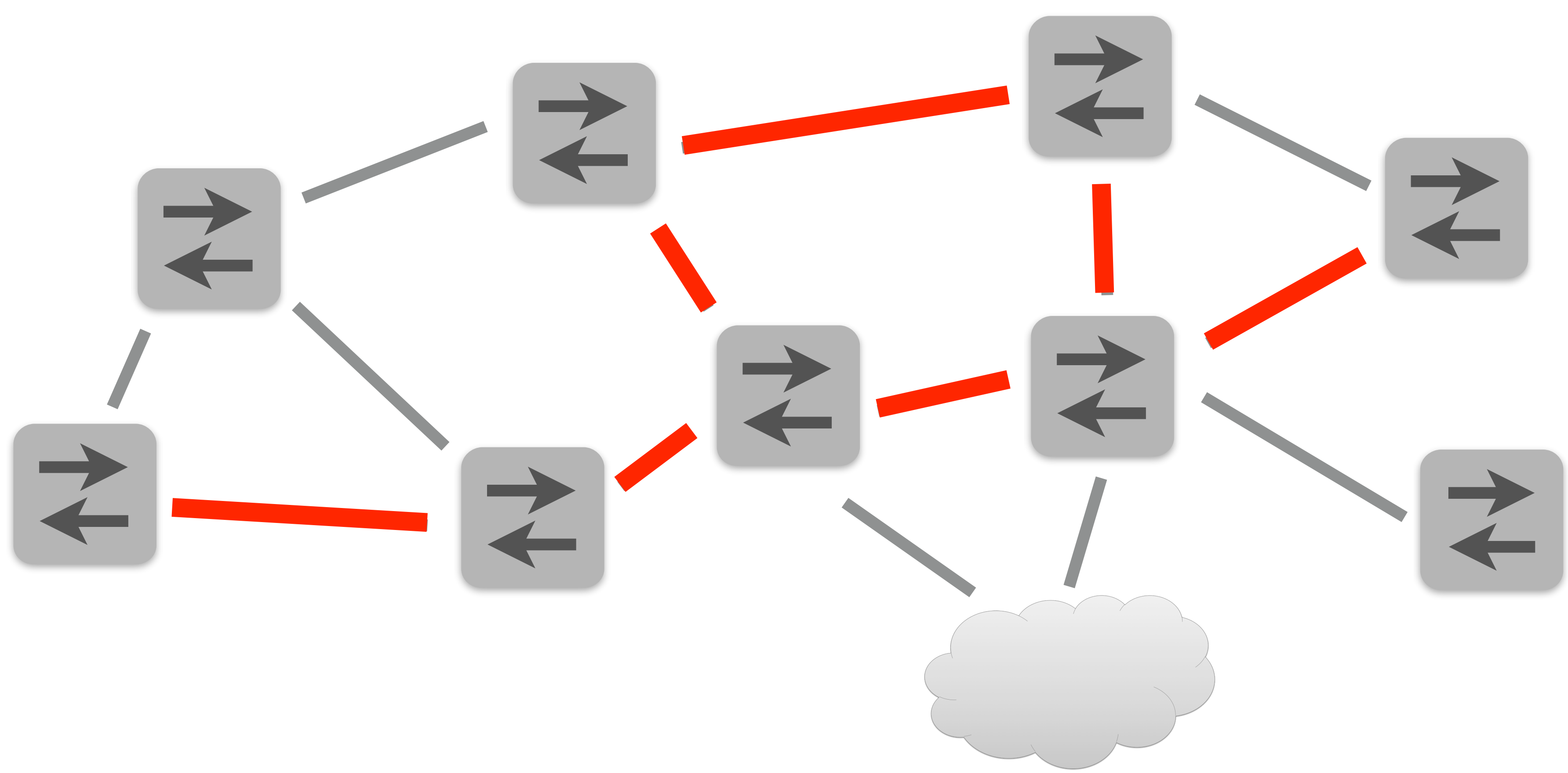
- Packet predicates
- Packet transformations
- Path construction
- Path concatenation
- Path union



Primitive Design

What constructs should an SDN language provide?

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



NetKAT Language

Syntax

pol ::= **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| !pol
| pol*
| **dup**

Semantics

Functions from packet histories to sets of packet histories

Syntactic Sugar

if pol **then** pol_1 **else** pol_2 $\triangleq$
(pol**;** pol_1) **+** (**!**pol**;** pol_2)

NetKAT Language

Syntax

pol ::= **false**
| **true**
| field = val

NetKAT can encode switch configurations, network-wide paths, and even topologies

| !pol
| pol*
| **dup**

Semantics

Functions from packet histories

if pol **then** pol_1 **else** pol_2 ≜ (pol; pol_1) + (!pol; pol_2)

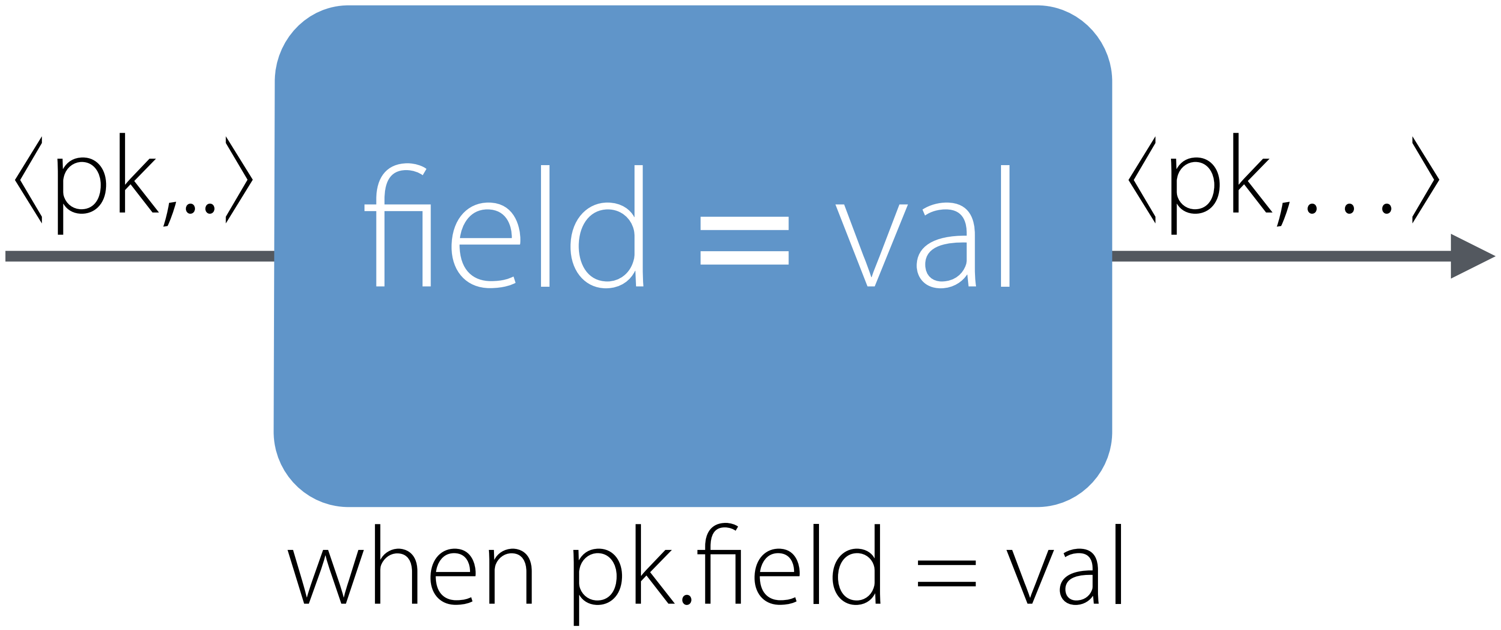
pol ::= **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| !pol
| pol*
| **dup**

false drops its input

pol ::= **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| !pol
| pol*
| **dup**

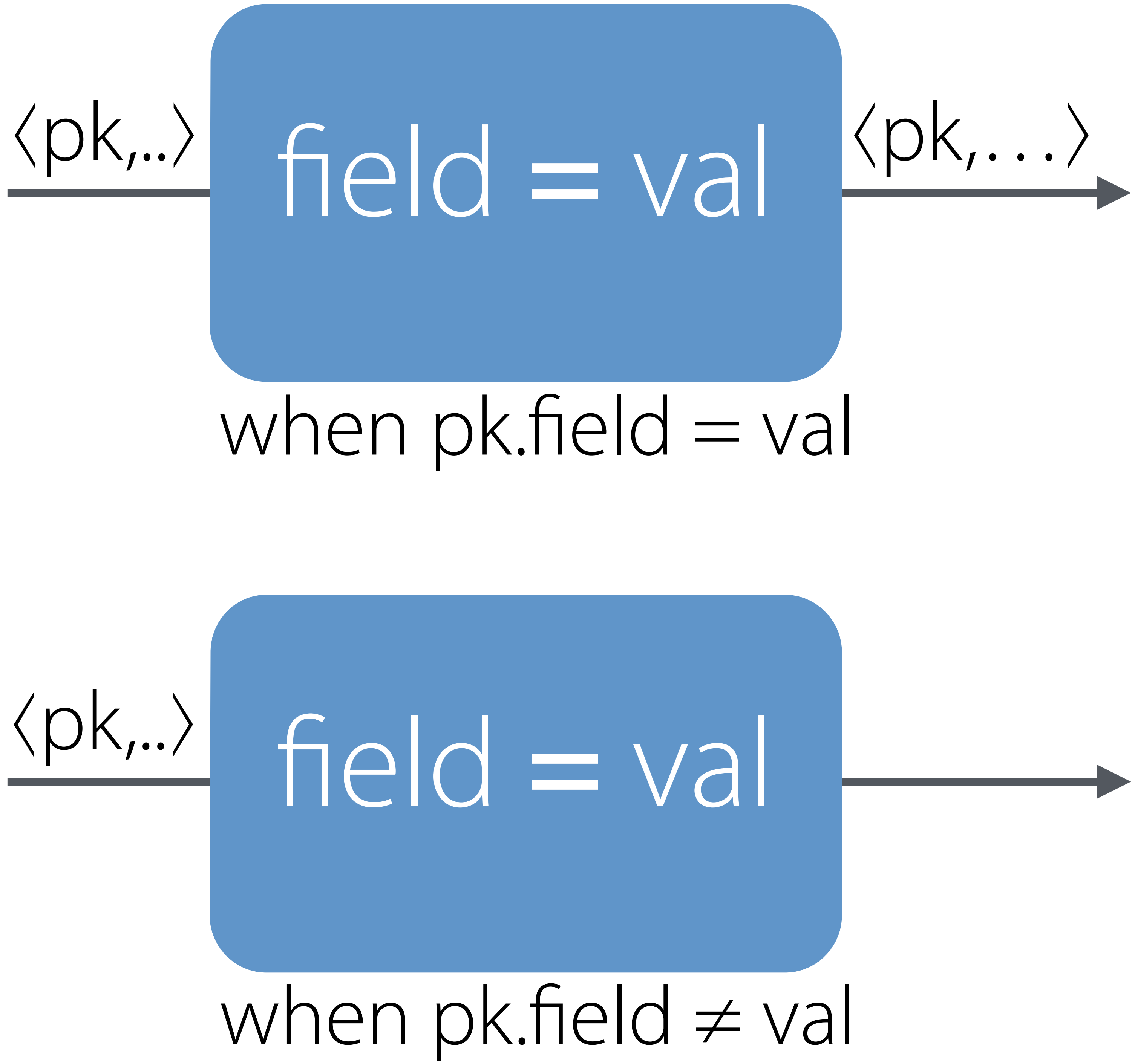
true copies its input

pol ::= **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| !pol
| pol*
| **dup**



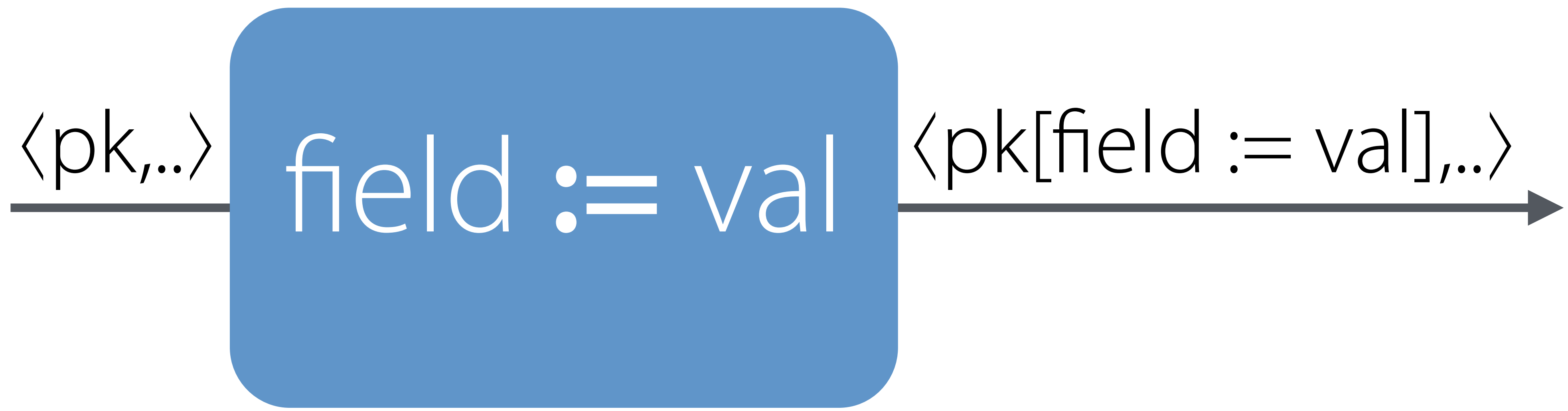
field **=** val copies its input if pk.field = val or drops it if not

pol ::= **false**
| **true**
| field = val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| !pol
| pol*
| **dup**



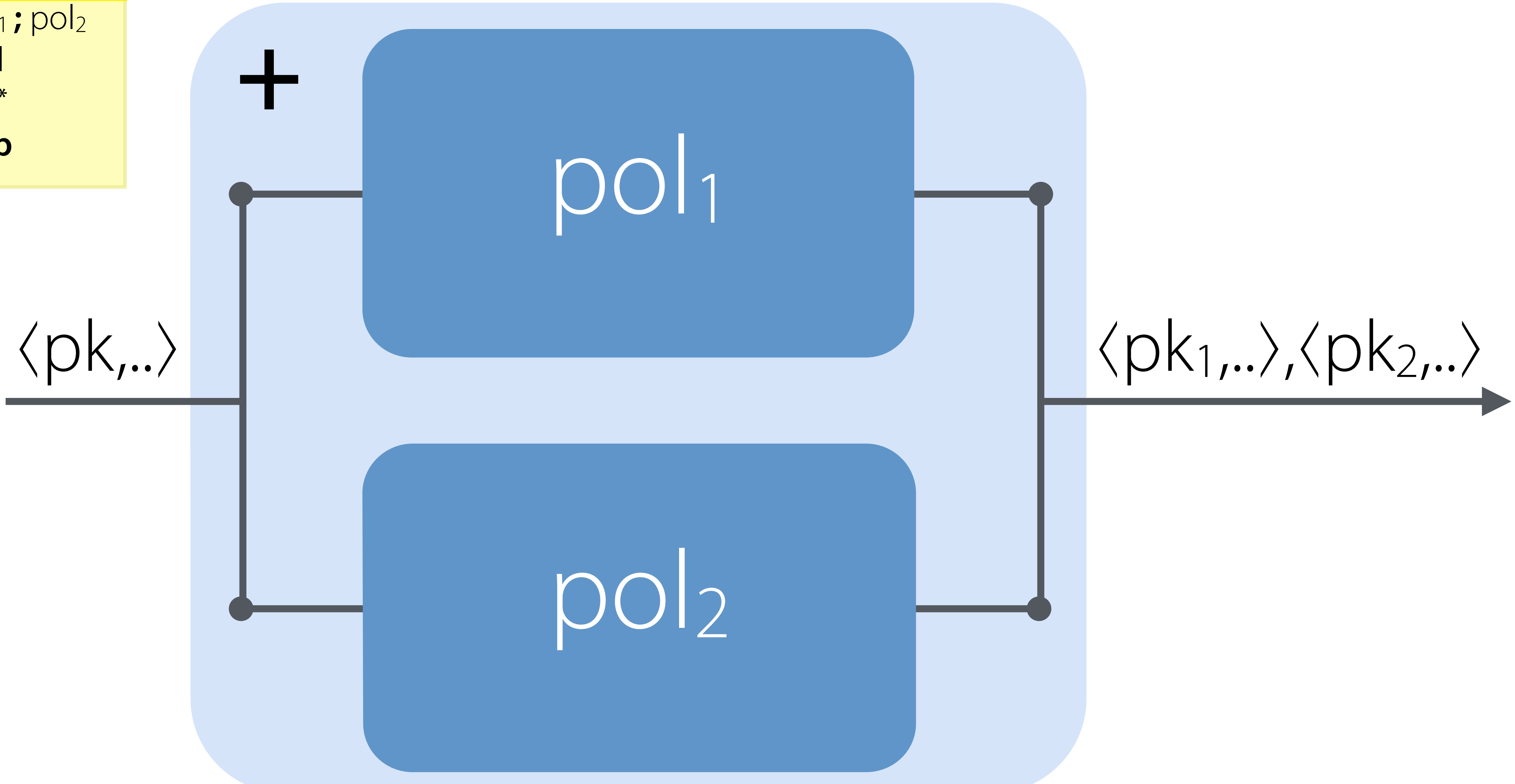
field **=** val copies its input if pk.field = val or drops it if not

pol ::= **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| !pol
| pol*
| **dup**



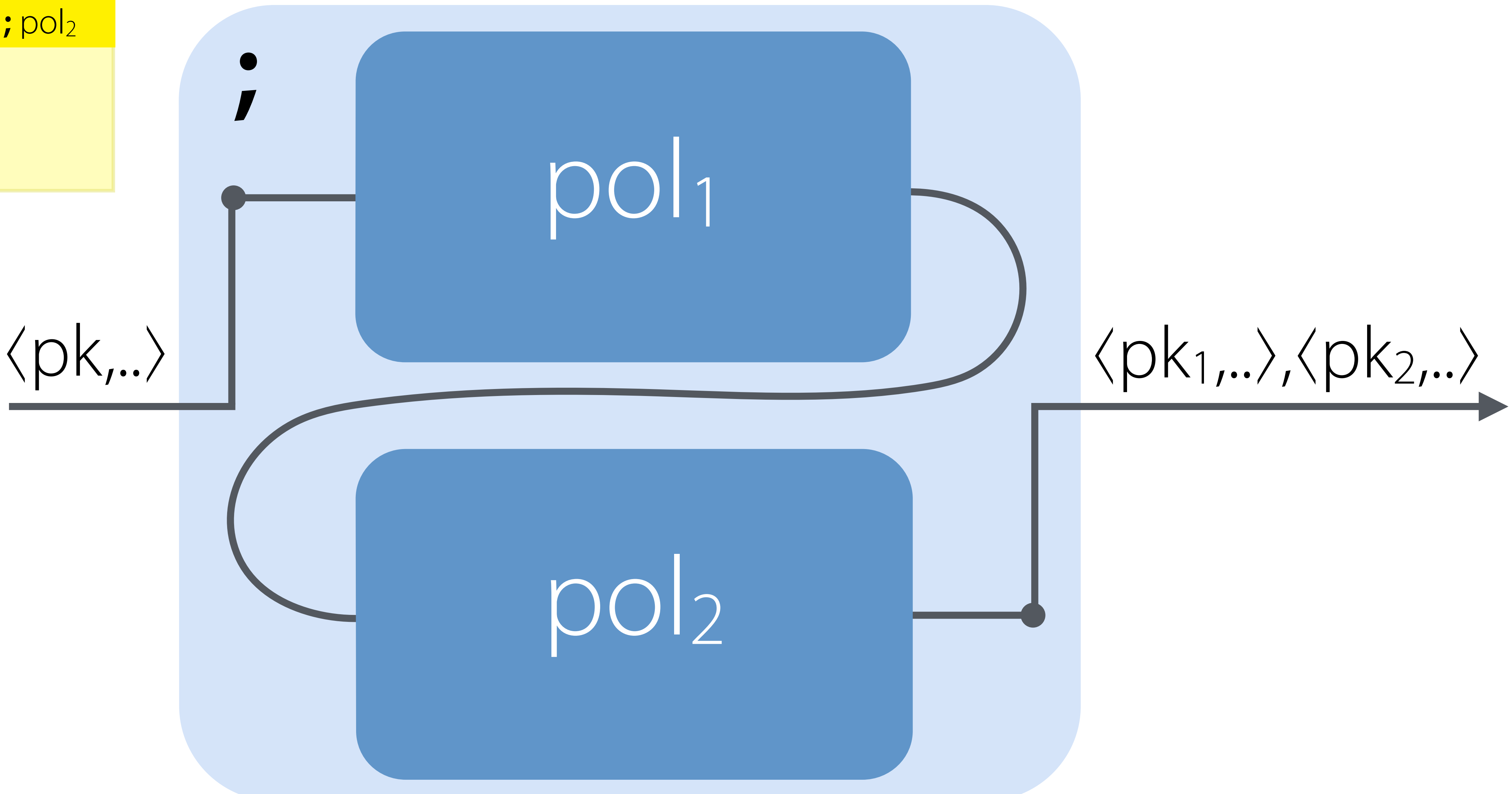
field **:=** val sets the input's field component to val

pol ::= **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| !pol
| pol*
| **dup**



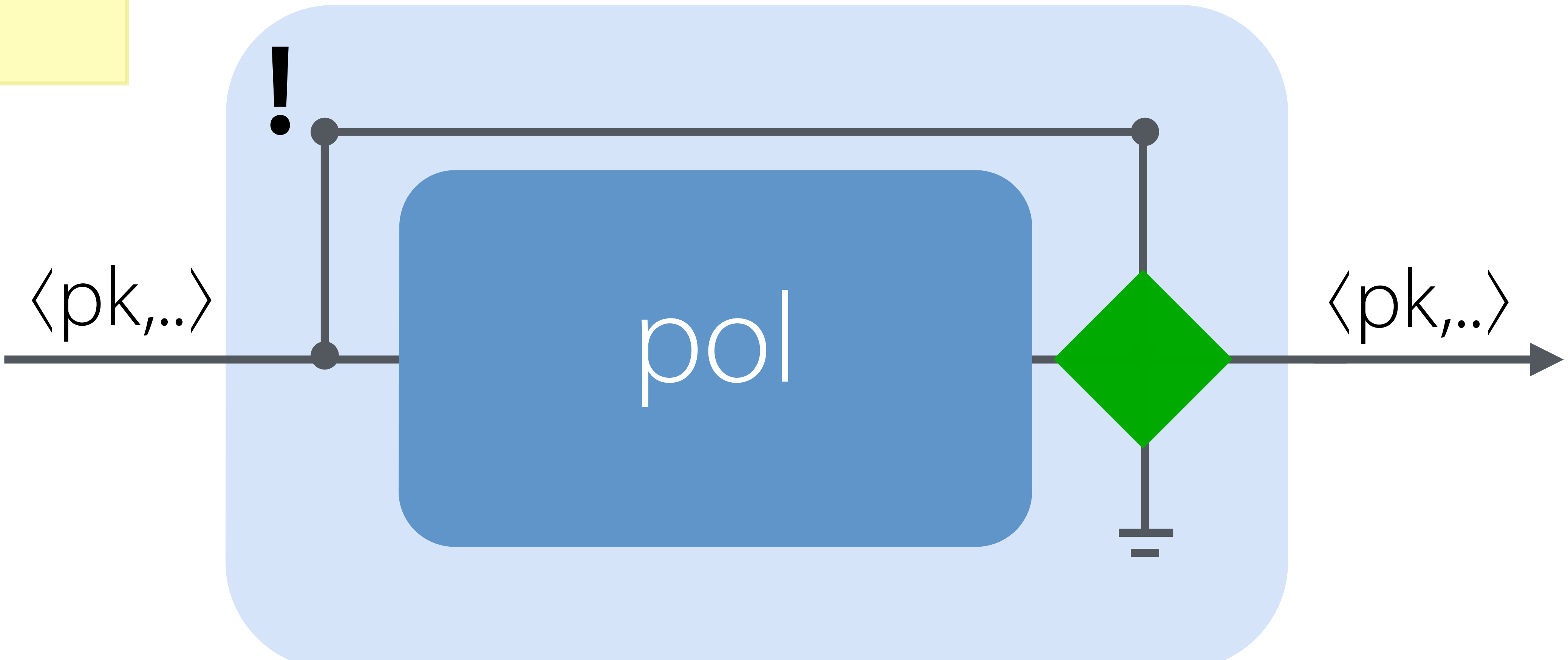
pol_1 **+** pol_1 duplicates the input, sends one copy to each sub-policy, and takes the *union* of their outputs

pol ::= **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| **!**pol
| pol*
| **dup**



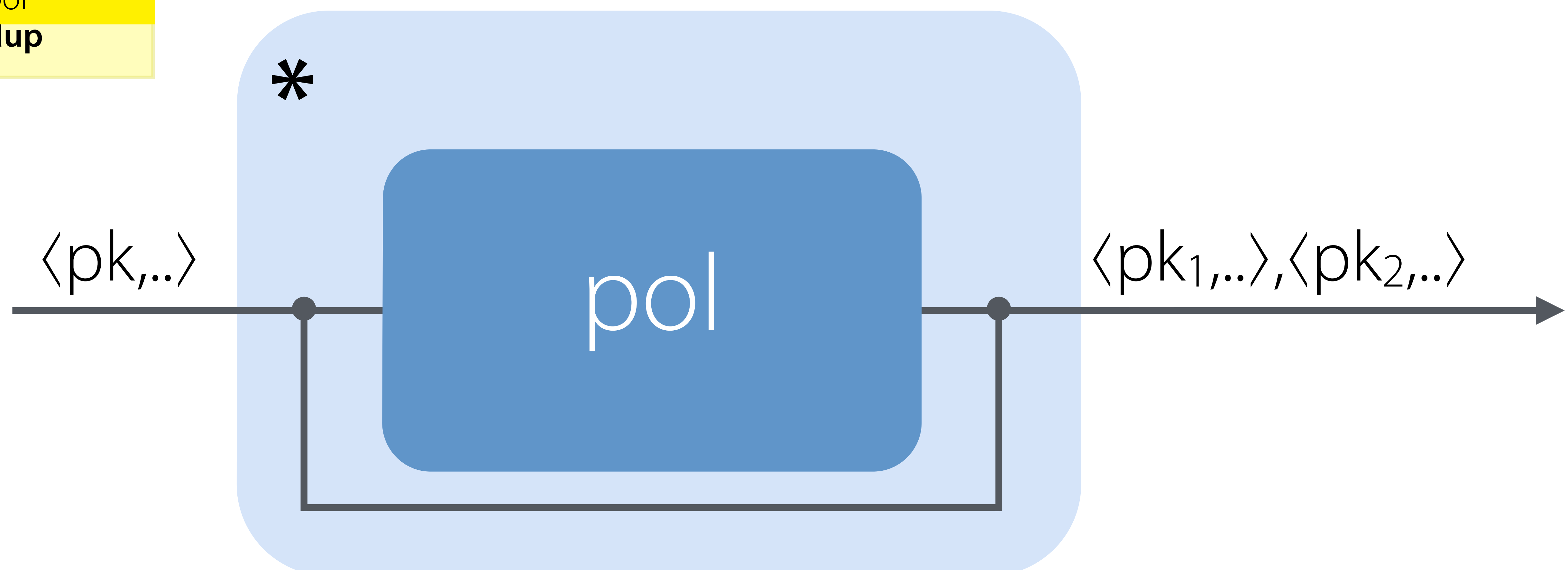
pol_1 **;** pol_2 runs the input through pol_1 and then runs every output produced by pol_1 through pol_2

pol ::= **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| **!**pol
| pol*
| **dup**



!pol drops the input if pol produces any output and copies it otherwise

pol ::= **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| **!**pol
| pol*
| **dup**



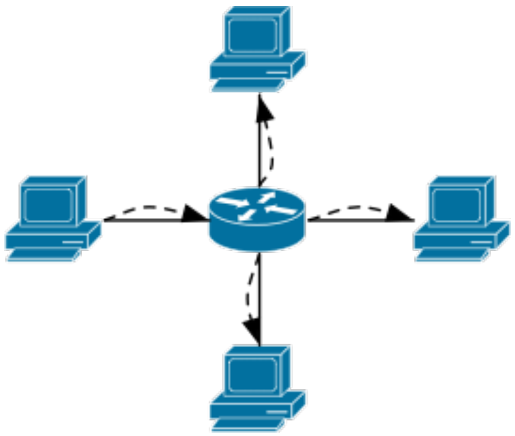
pol* repeatedly runs packets through pol to a fixpoint

pol ::= **false**
| **true**
| field **=** val
| field **:=** val
| pol_1 **+** pol_2
| pol_1 **;** pol_2
| **!**pol
| pol*
| **dup**

dup duplicates the head packet of the input

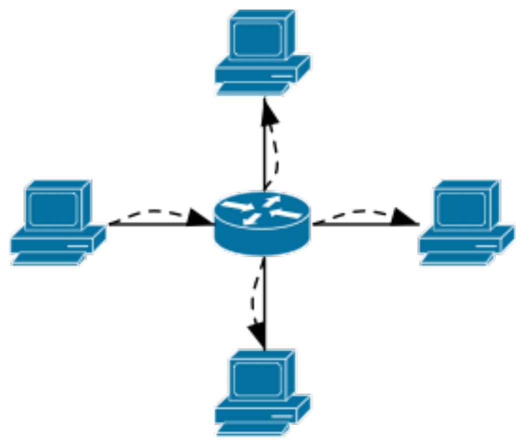
Example

Topology



Example

Topology



Specification

- Forward packets to hosts 1-4
- Monitor traffic to unknown hosts
- Flood broadcast traffic to all hosts
- Disallow SSH traffic from hosts 1-2

Flow Table

```
{pattern={ethSrc=00:00:00:00:00:01,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethSrc=00:00:00:00:00:02,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethDst=00:00:00:00:00:01},action=[Output(1)]}
{pattern={ethDst=00:00:00:00:00:02},action=[Output(2)]}
{pattern={ethDst=00:00:00:00:00:03},action=[Output(3)]}
{pattern={ethDst=00:00:00:00:00:04},action=[Output(4)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=1},action=[Output(4), Output(3), Output(2)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=2},action=[Output(4), Output(3), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=3},action=[Output(4), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=4},action=[Output(3), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff},action=[]}
{pattern={},action=[Controller]}
```

Example: Forward

```
let forward =
  if ethDst = 00:00:00:00:00:01 then
    port := 1
  else if ethDst = 00:00:00:00:00:02 then
    port := 2
  else if ethDst = 00:00:00:00:00:03 then
    port := 3
  else if ethDst = 00:00:00:00:00:04 then
    port := 4
  else
    false
```

Example: Broadcast

```
let flood =
  if port = 1 then
    port := 2 + port := 3 + port := 4
  else if port = 2 then
    port := 1 + port := 3 + port := 4
  else if port = 3 then
    port := 1 + port := 2 + port := 4
  else if port = 4 then
    port := 1 + port := 2 + port := 3
  else
    false

let broadcast =
  if ethDst = ff:ff:ff:ff:ff:ff then
    flood
  else
    false
```

Example: Routing

```
let route = forward + broadcast
```

Example: Monitor

```
let monitor =
  if !(ethDst = 00:00:00:00:00:01 +
       ethDst = 00:00:00:00:00:02 +
       ethDst = 00:00:00:00:00:03 +
       ethDst = 00:00:00:00:00:04 +
       ethDst = ff:ff:ff:ff:ff:ff) then
    port := unknown
  else
    false
```

Example: Firewall

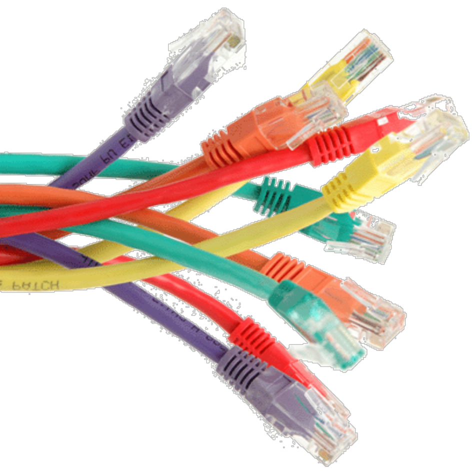
```
let firewall =
  if (ethSrc = 00:00:00:00:00:01 +
      ethSrc = 00:00:00:00:00:02) ;
     ethTyp = 0x800 ;
     ipProto = 0x06 ;
     tcpDstPort = 22 then
     false
   else
     true
```

Example: Main Policy

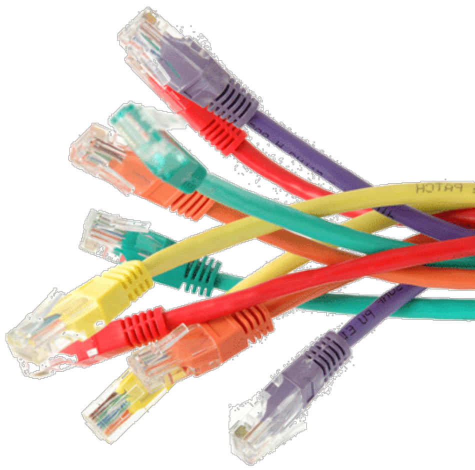
```
let main = (route + monitor); firewall
```

compiles to…

```
{pattern={ethSrc=00:00:00:00:00:01,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethSrc=00:00:00:00:00:02,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethDst=00:00:00:00:00:01},action=[Output(1)]}
{pattern={ethDst=00:00:00:00:00:02},action=[Output(2)]}
{pattern={ethDst=00:00:00:00:00:03},action=[Output(3)]}
{pattern={ethDst=00:00:00:00:00:04},action=[Output(4)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=1},action=[Output(4), Output(3), Output(2)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=2},action=[Output(4), Output(3), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=3},action=[Output(4), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=4},action=[Output(3), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff},action=[]}
{pattern={},action=[Controller]}
```



Demo

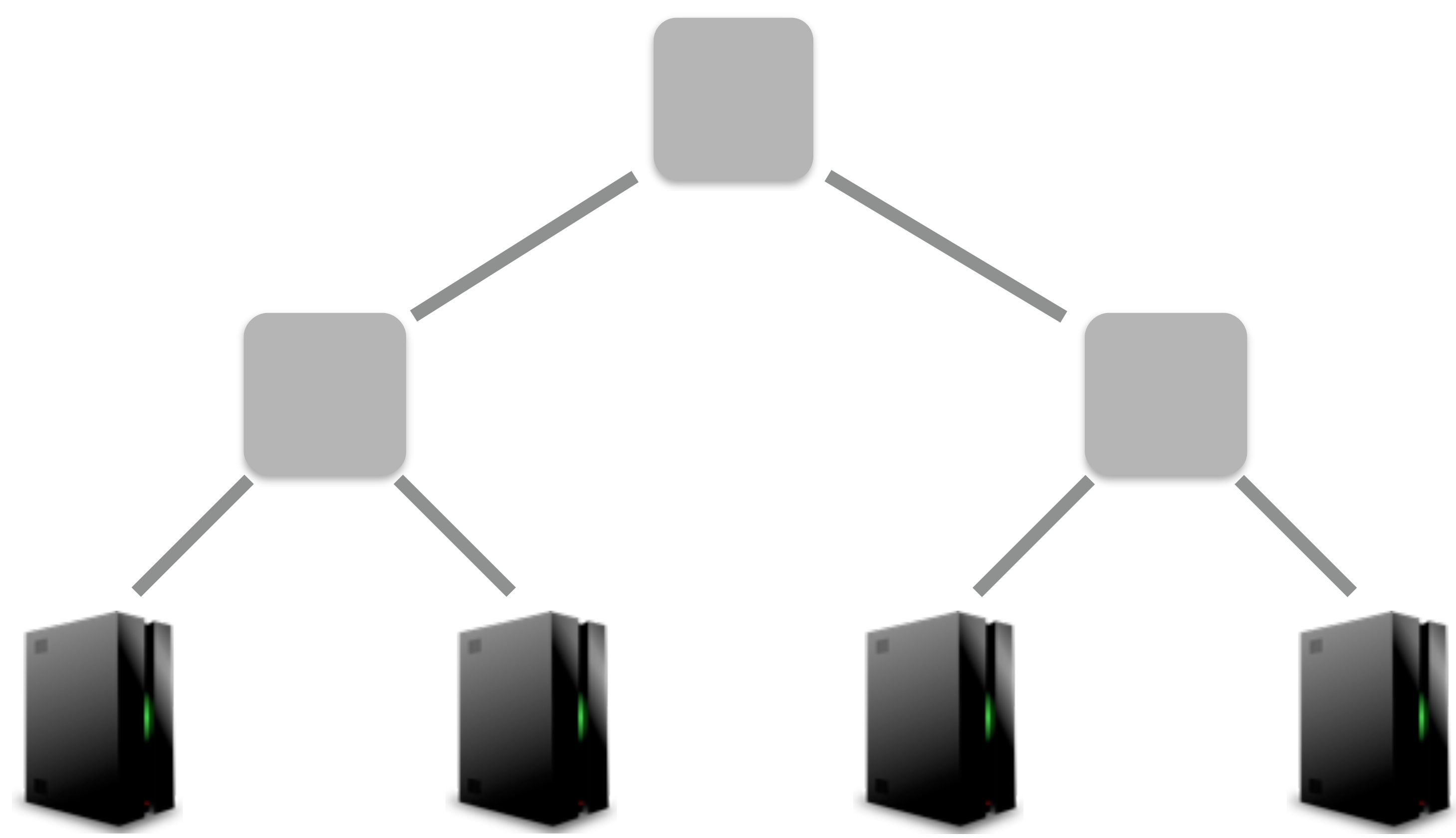


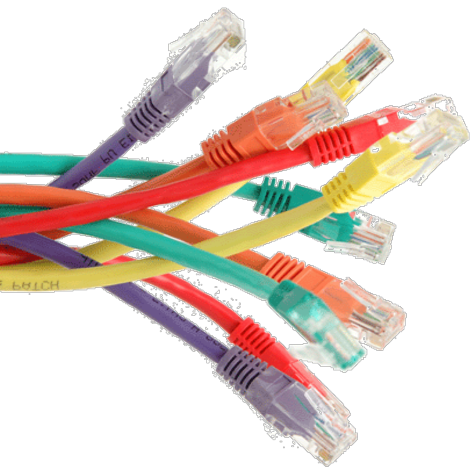
Demo Application

Discovery + (Whitelist ; Learning)

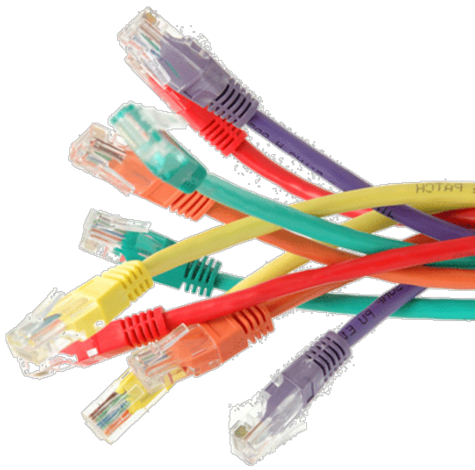
NetKAT Language

Run-Time System

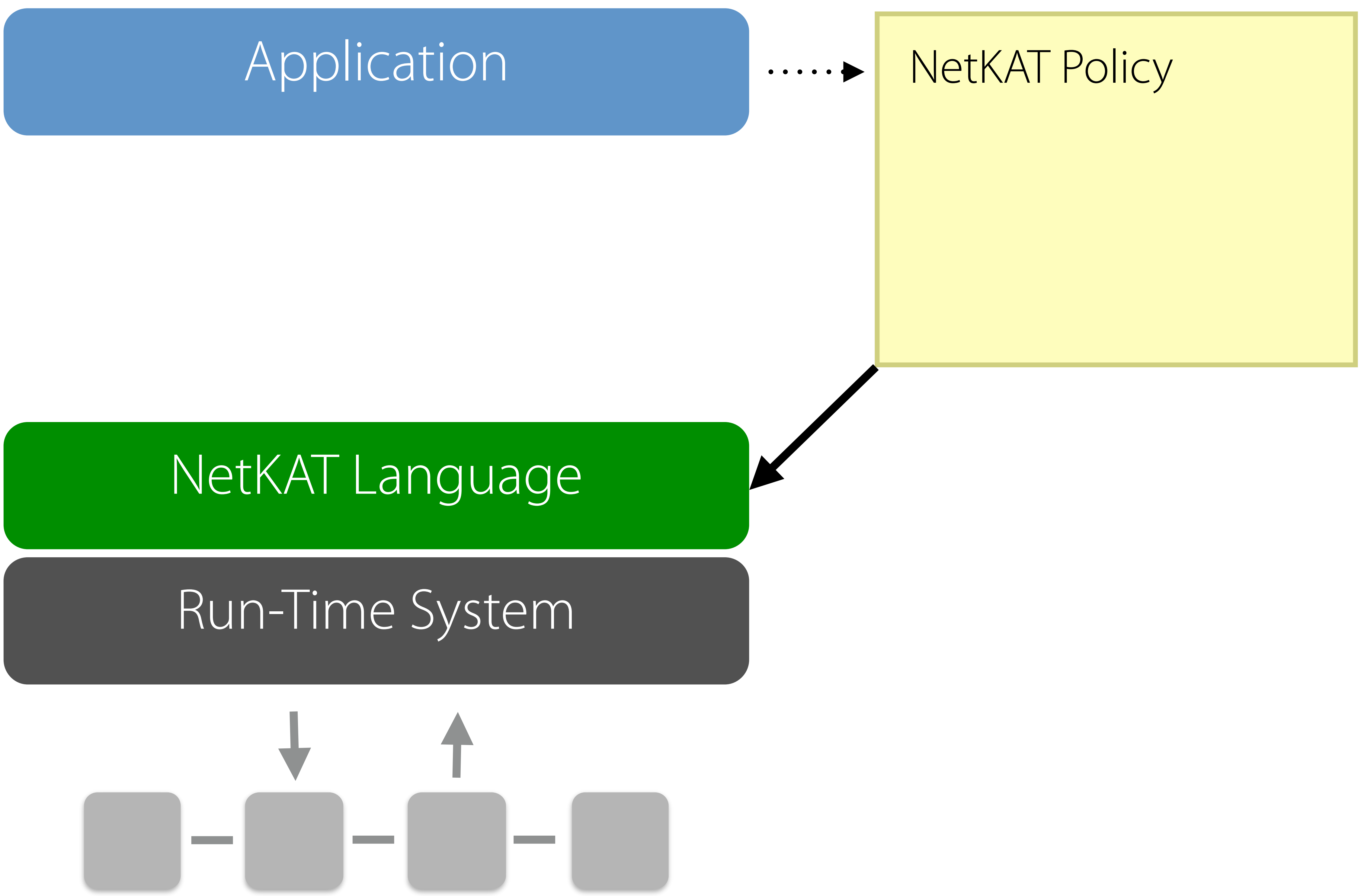




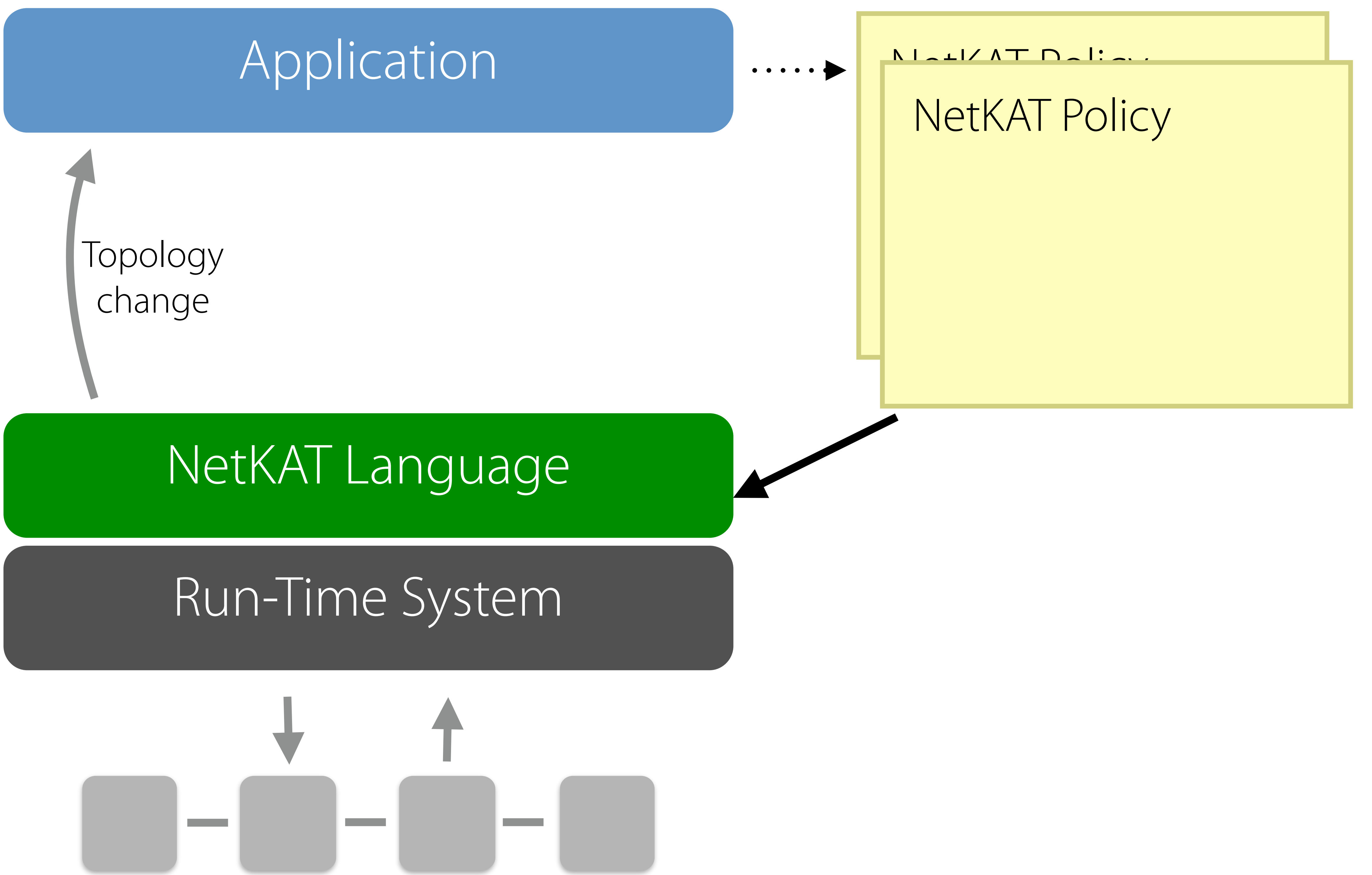
Dynamic Applications



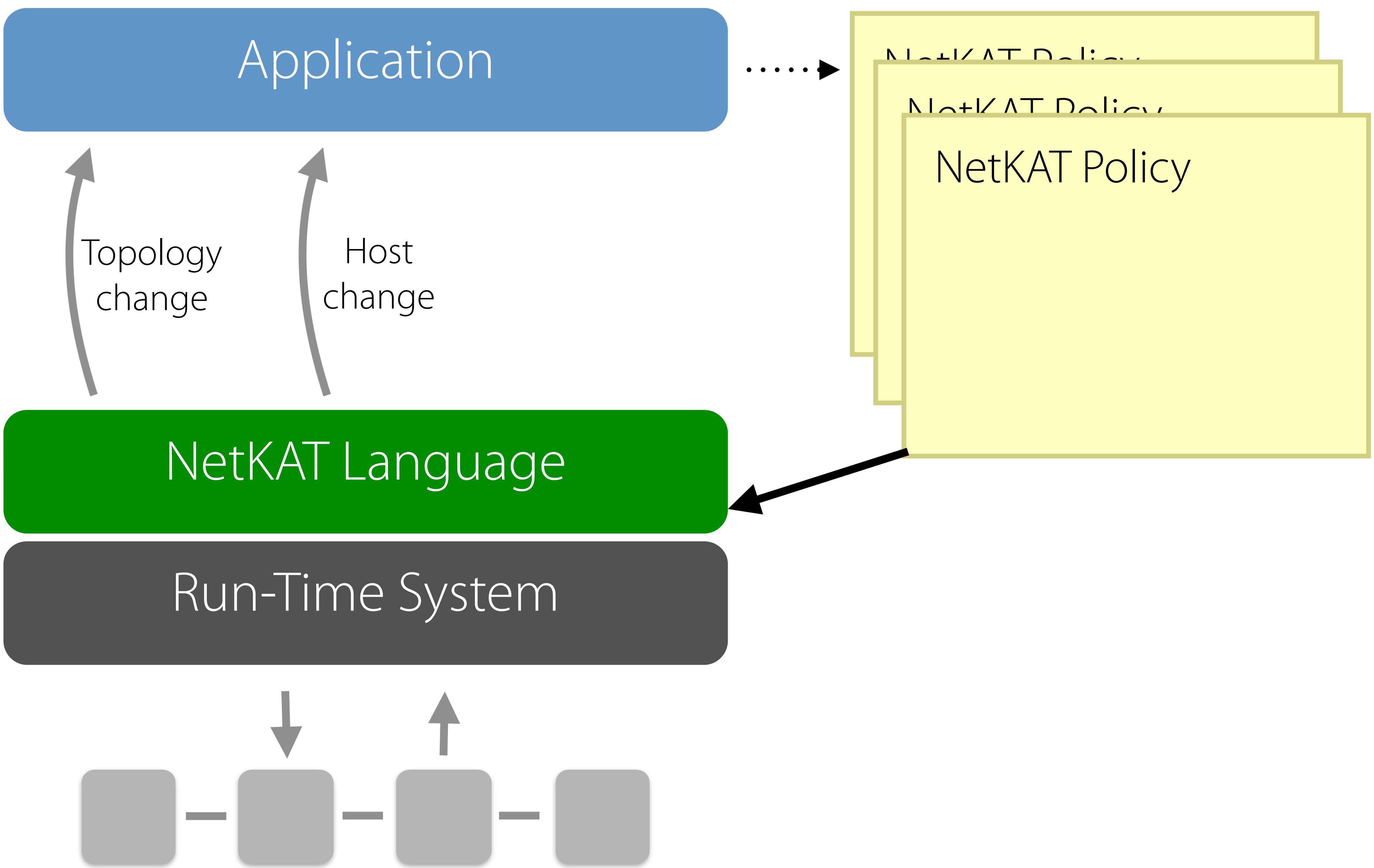
Dynamic Applications



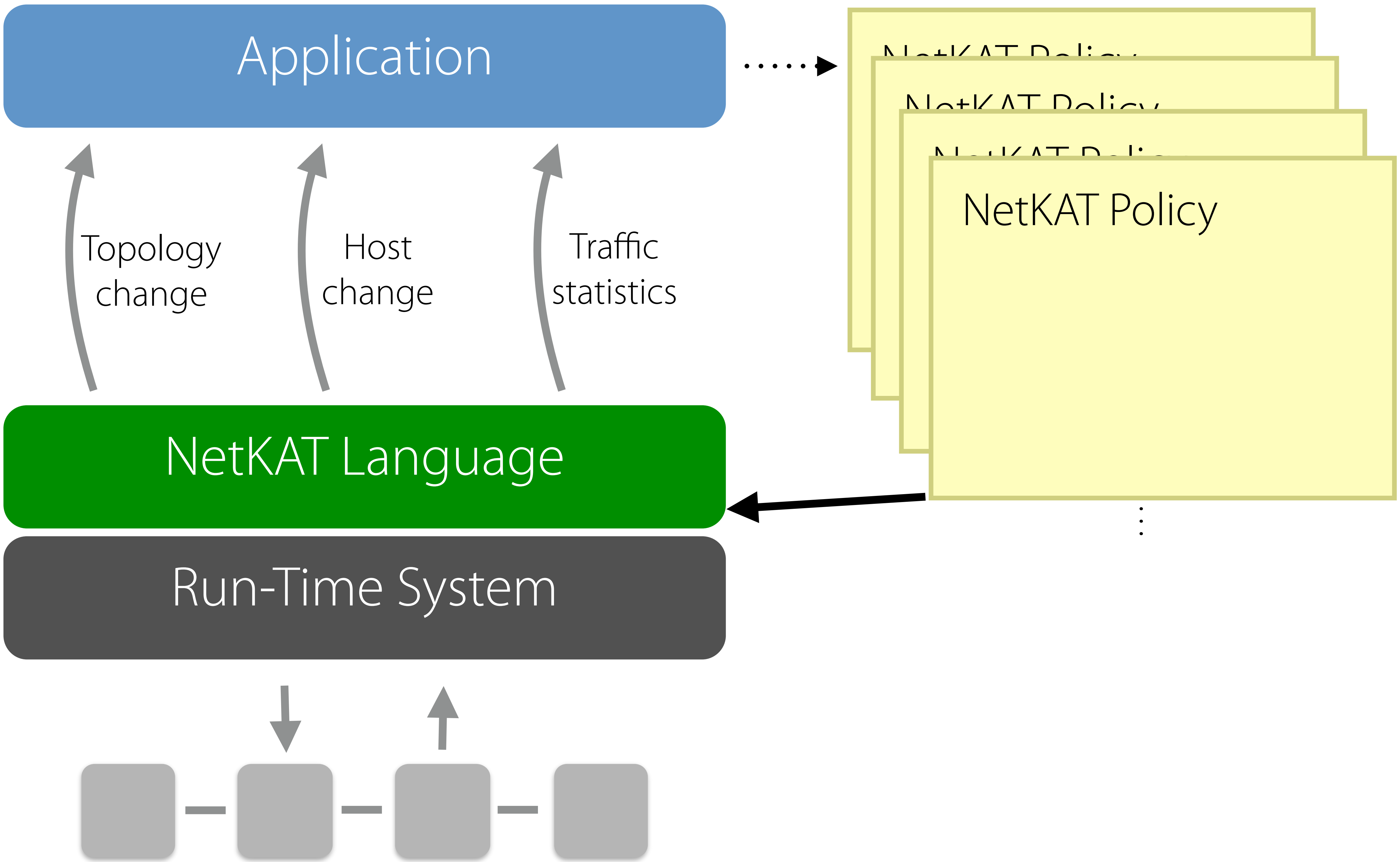
Dynamic Applications



Dynamic Applications

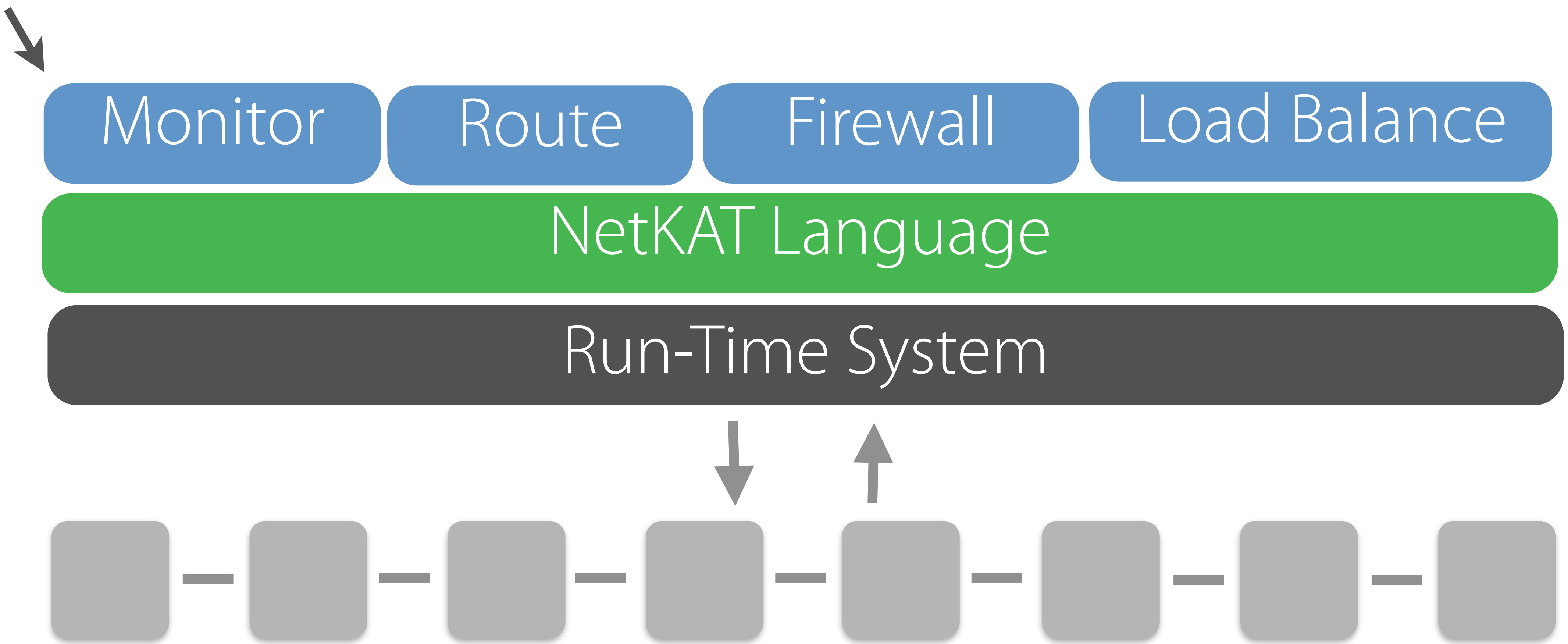


Dynamic Applications



Composing Applications

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

Application Interface

Supports dynamic, stateful applications

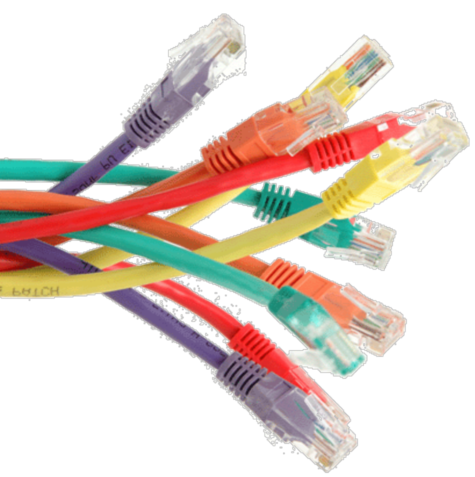
```
type app
type event =
  | PacketIn of string * switchId * portId * payload * int
  | SwitchUp of switchId
  | SwitchDown of switchId
  | ...

type handler =  event -> policy option
val create: policy -> handler -> app
val par: app -> app -> app
val seq: app -> app -> app
```

Concurrency via Jane Street's **async** library

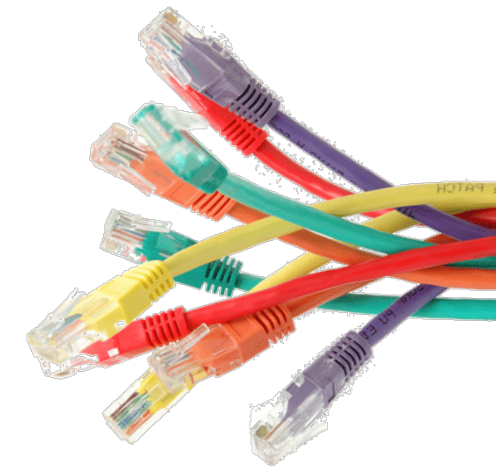
RESTful interface: write NetKAT applications in Python!

Python Learning Switch

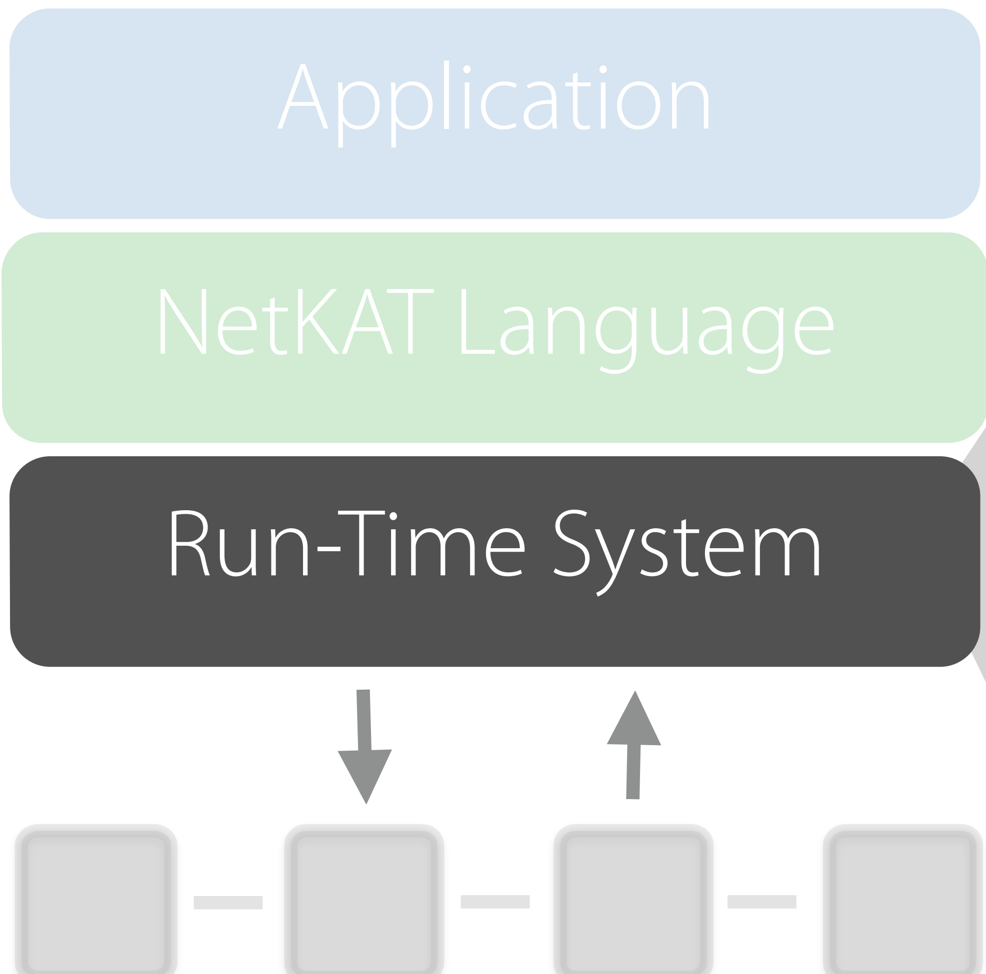
```
# switch state
table = {}

# helper functions
def learn(sw,pkt,pt):
    table[sw][get_ethernet(pkt).src] = pt

def switch_policy(sw):
    def f((known,unknown),mac):
       src = test("ethSrc", mac)
       dst = test("ethDst", mac)
       return (known | filter(dst) >> output(table[sw][mac]), unknown & ~src)
    (known_pol, unknown_pred) = reduce(f, table[sw].keys(), (drop(), true()))
    return known_pol | filter(unknown_pred) >> (controller() | flood(sw))

def policy():
    return union(switch_policy(sw) for sw in table.keys())

# event handler
def handler(_, event):
    print event
    typ = event['type']
    if typ == 'packet_in':
       pkt = packet.Packet(base64.decode(event[‘payload’]['buffer']))
       learn(event[‘switch_id'], pkt, event[‘port_id'])
    else:
       pass
    return PolicyResult(policy())
```



Run-Time System



Run-Time System



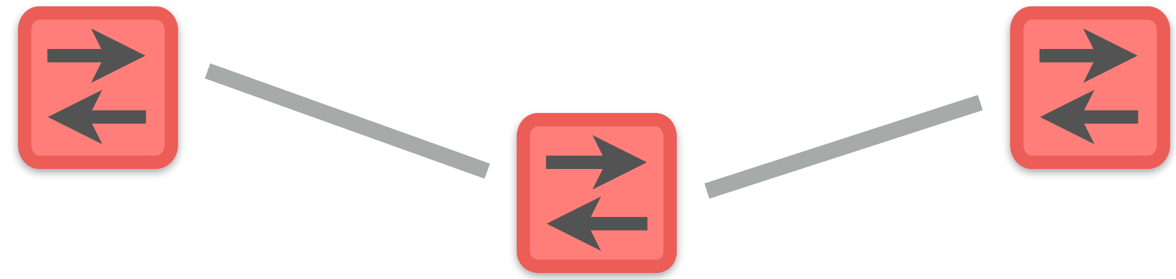

```
let swap_update_for (t : t) sw_id c_id new_table : unit Deferred.t =
   let max_priority = 65535 in
   let old_table = match SwitchMap.find t.edge sw_id with | Some ft -> ft | None -> [] in
   let (new_table, _) = List.fold new_table ~init:([], max_priority)
        ~f:(fun (acc,pri) x -> ((x,pri) :: acc, pri - 1)) in
    let new_table = List.rev new_table in
    let del_table = List.rev (flowtable_diff old_table new_table) in
    let to_flow_mod prio flow =
      M.FlowModMsg (SDN_OpenFlow0x01.from_flow prio flow) in
    let to_flow_del prio flow =
      M.FlowModMsg ({SDN_OpenFlow0x01.from_flow prio flow with command = DeleteStrictFlow}) in
   Deferred.List.iter new_table ~f:(fun (flow, prio) ->
      send t.ctl c_id (0l, to_flow_mod prio flow))
    >>= fun () -> Deferred.List.iter del_table ~f:(fun (flow, prio) ->
      send t.ctl c_id (0l, to_flow_del prio flow))
    >>| fun () -> t.edge <- SwitchMap.add t.edge sw_id new_table
```

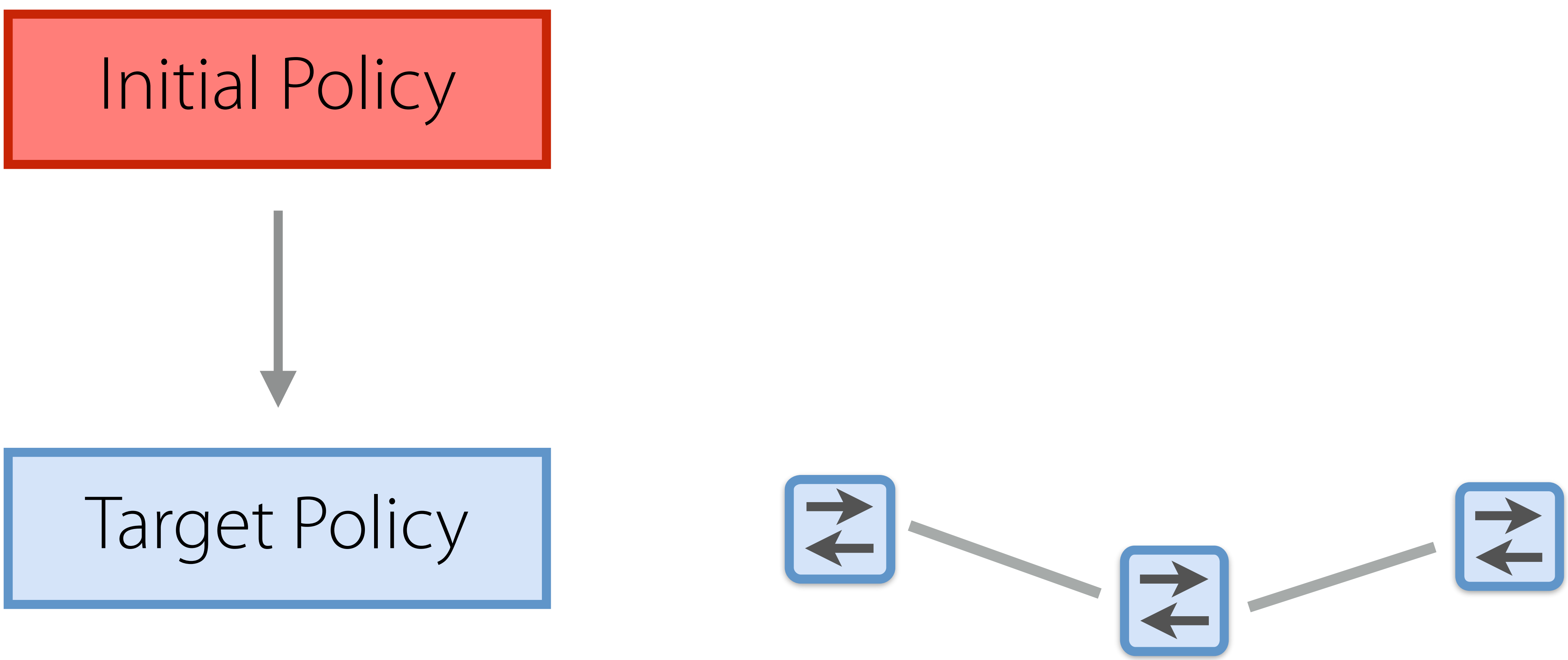
Code that manages the rules installed on switches

Translate configuration updates into sequences of OpenFlow instructions

Network Updates

Question: how can we gracefully transition the network from one configuration to another?





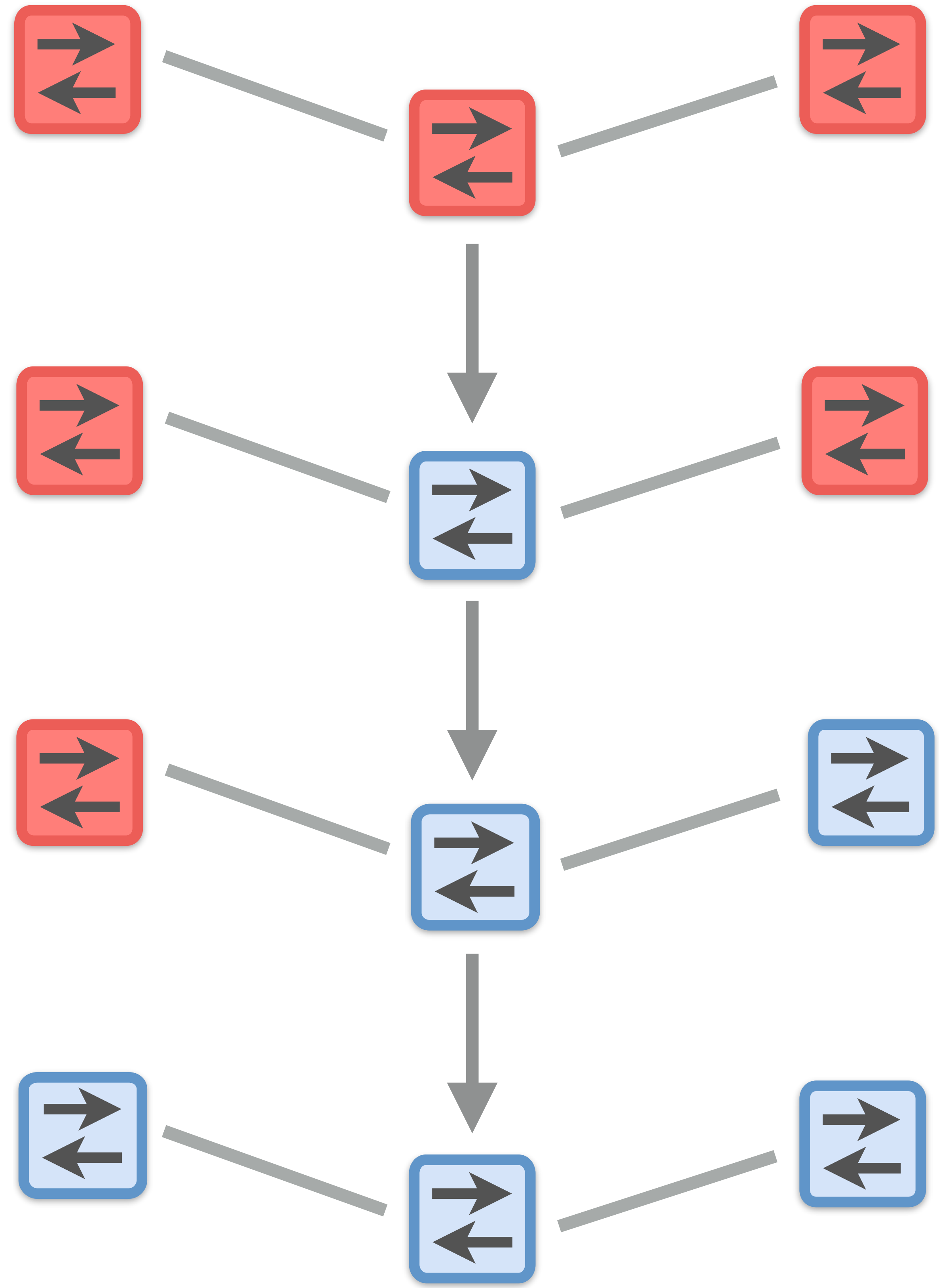
Network Updates

Question: how can we gracefully transition the network from one configuration to another?

Initial Policy

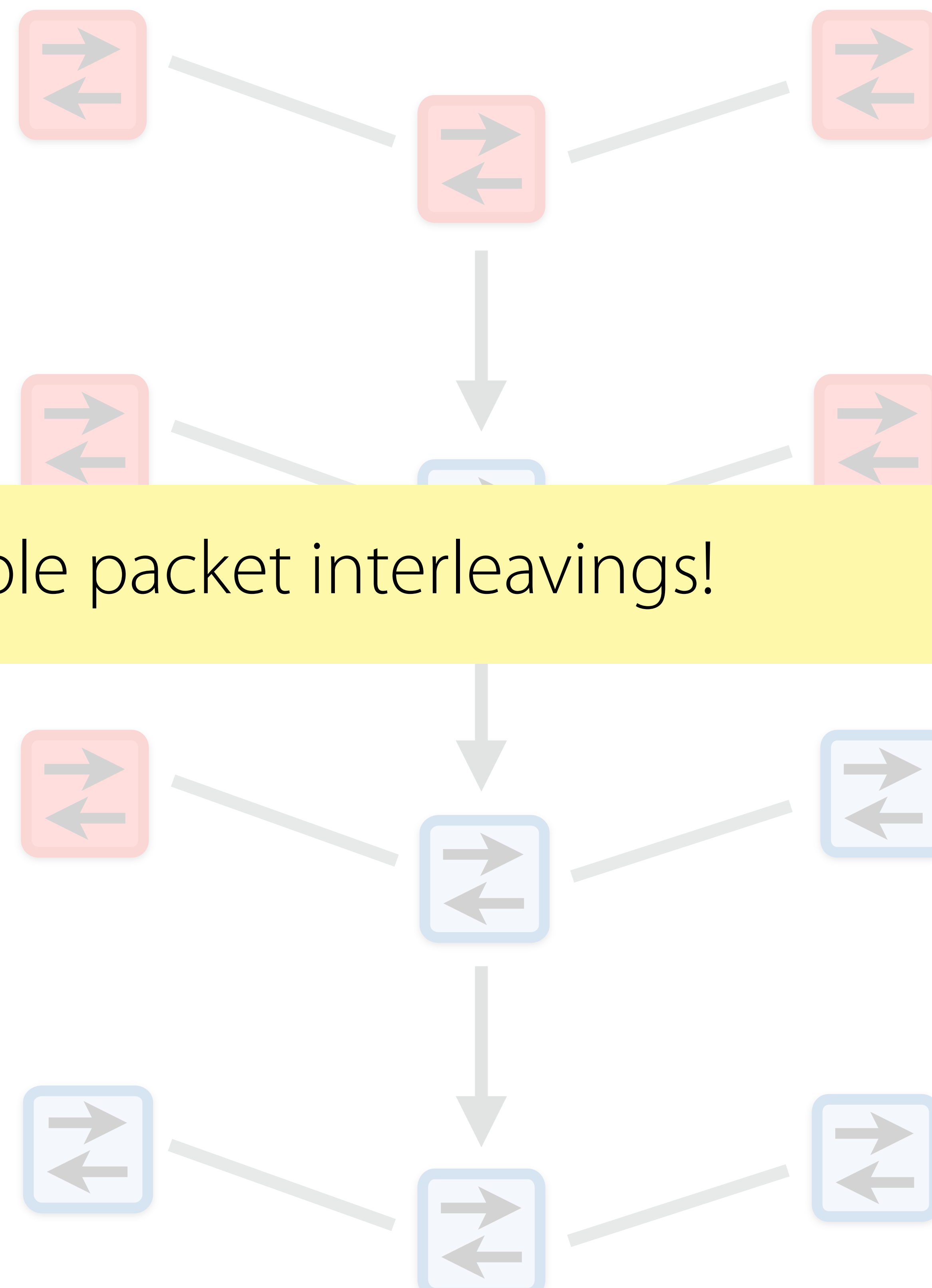
↓

Target Policy



Network Updates

Question: how can we gracefully transition the network from one configuration to another?



Must reason about all possible packet interleavings!

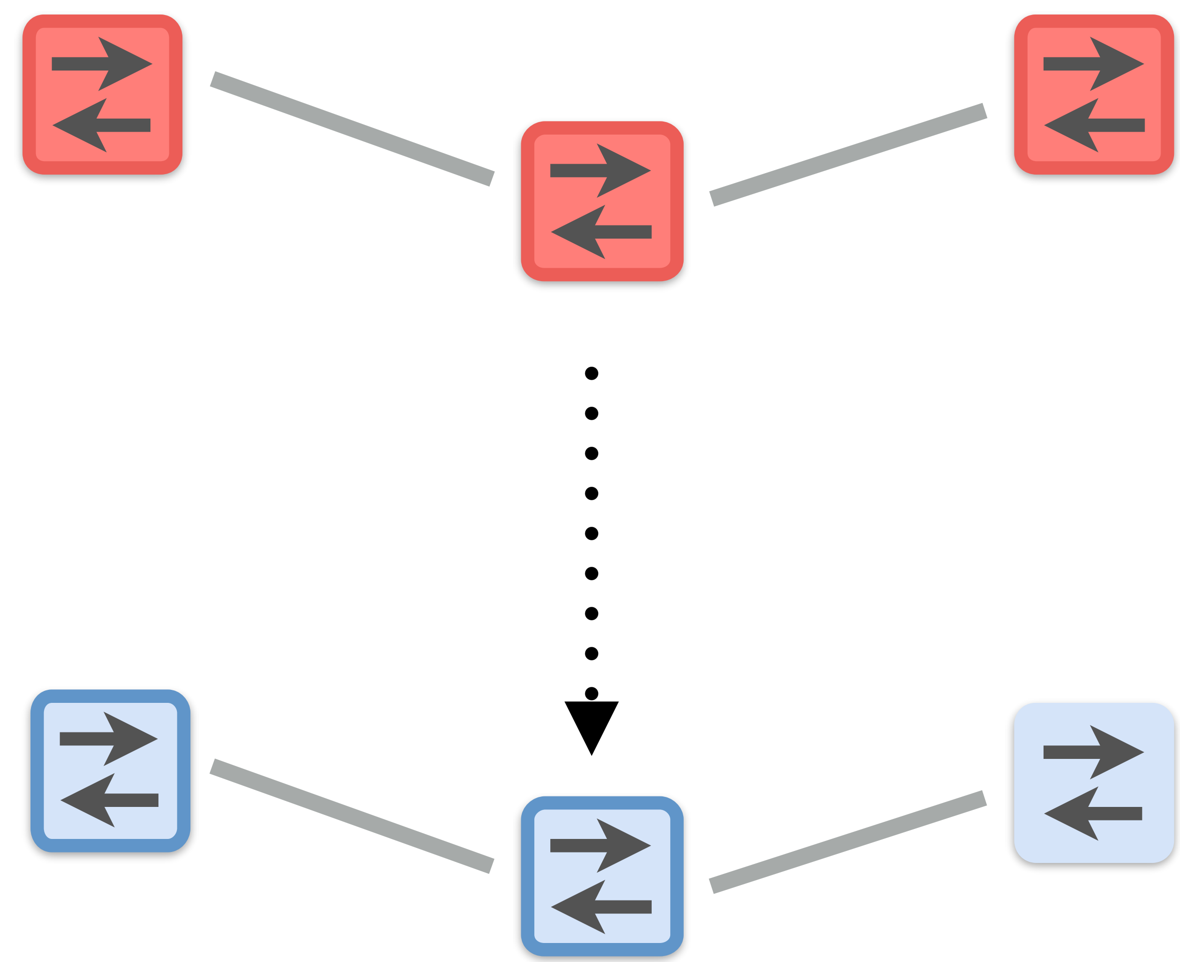
Consistent Updates

Approach: develop abstractions that appear to update *all* of the switches in the network at once

Consistency Property: every packet (or flow) in the network "sees" a single policy version

Implementations:

- Order updates
- Unobservable updates
- One-touch updates
- Compositions of consistent
- Two-phase update



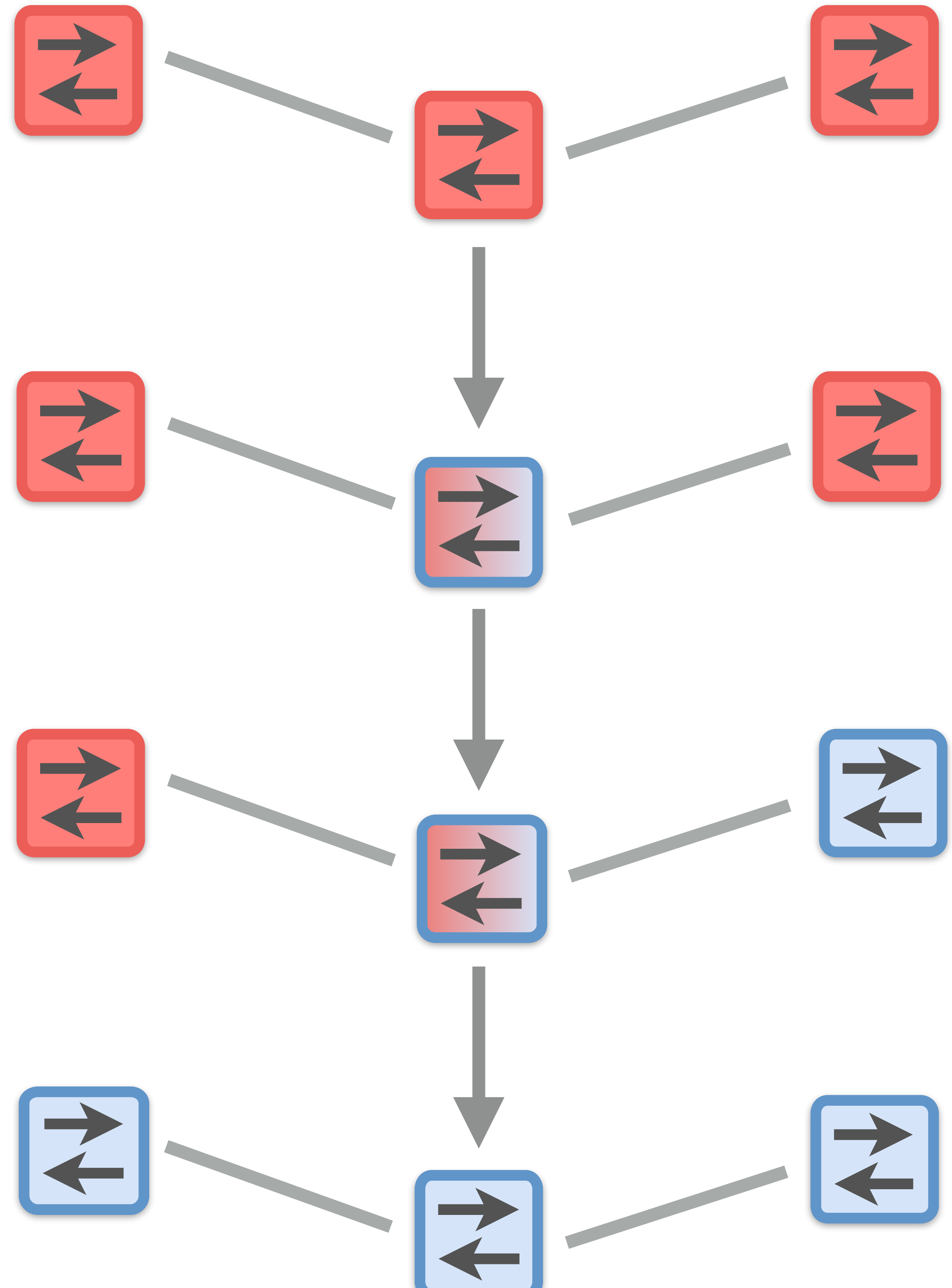
Two-Phase Updates

Versioning: instrument the compiler so that all forwarding rules match on a policy version

Unobservable Update: install the rules for the new policy in the interior of the network

One-Touch Updates: install rules at the edge that stamp packets with new version

Garbage Collect: delete the rules for the old policy



Two-Phase Updates

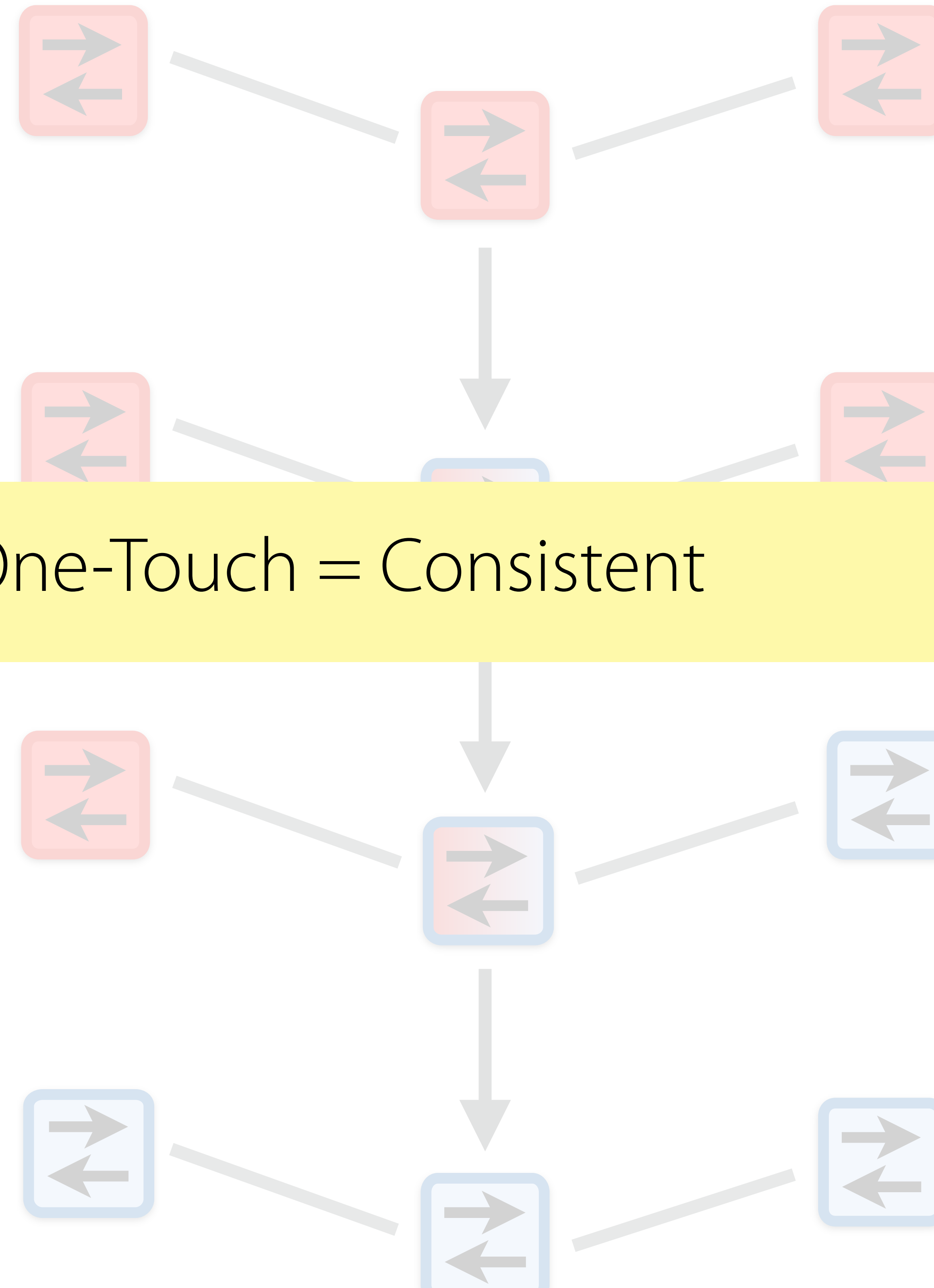
Versioning: instrument the compiler so that all forwarding rules match on a policy version

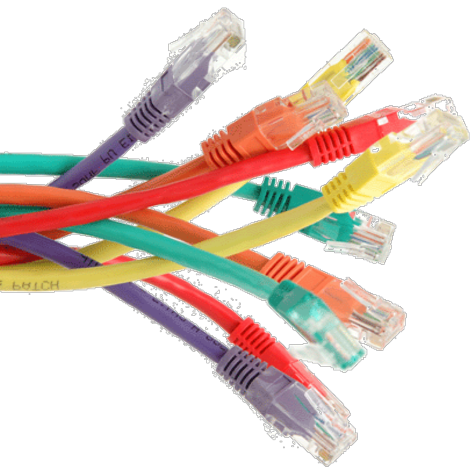
Unobservable Update: install the rules for the new policy in

Theorem: Unobservable + One-Touch = Consistent

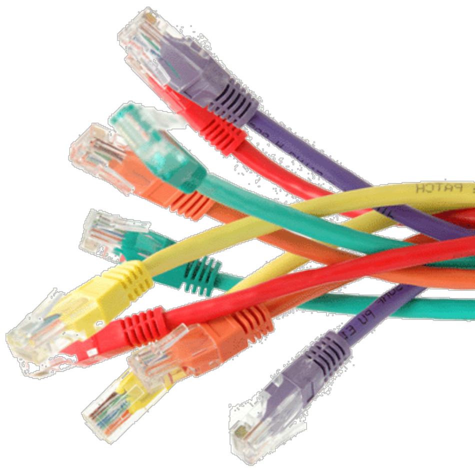
One-Touch Updates: install rules at the edge that stamp packets with new version

Garbage Collect: delete the rules for the old policy





Applications



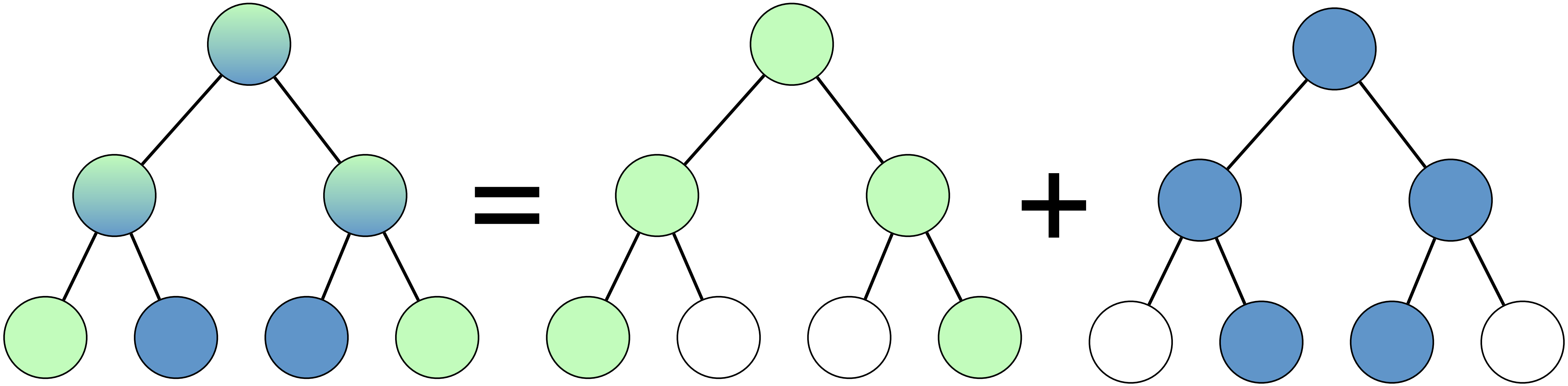
Rich Applications with NetKAT

- Isolated Slices
- Virtual Networks
- Network Debugging
- Fault Tolerance
- Quality of Service Provisioning
- Network Function Virtualization

Isolated Slices

In many situations, multiple tenants must share the network…

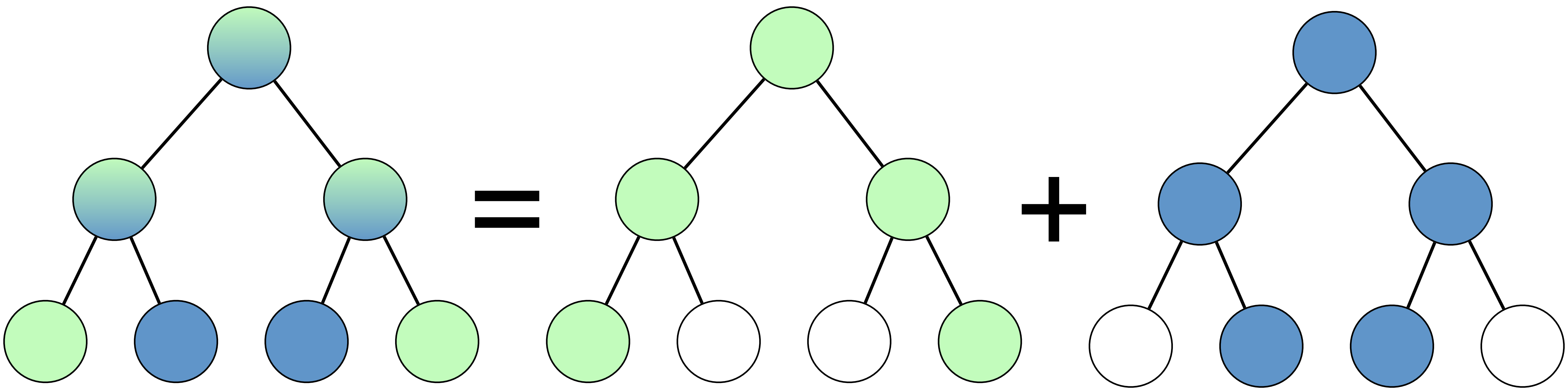
…but we don't want their traffic to interfere with each other!



Isolated Slices

In many situations, multiple tenants must share the network…

…but we don't want their traffic to interfere with each other!



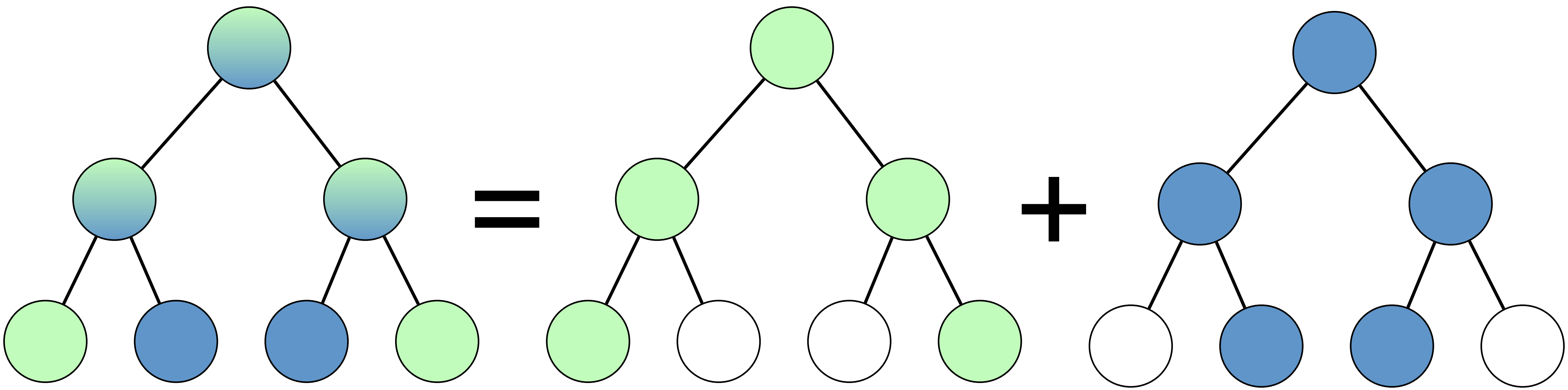
Ingress Tag Policy Egress

{ in } x : pol { out }

Isolated Slices

In many situations, multiple tenants must share the network…

…but we don't want their traffic to interfere with each other!



Ingress Tag Policy Egress

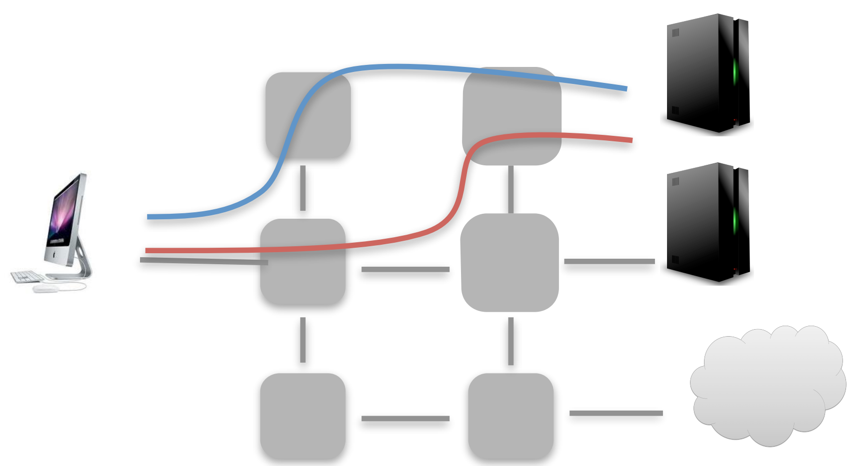
{ in } x : pol { out }

→

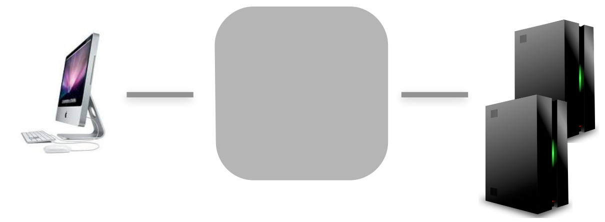
let pre = (tag **=** none**;** in**;** tag **:=** x **+** tag **=** x) **in**
let post = (out**;** tag **:=** none **+** **!**out**)** in
(pre**;** pol**;** post)

Virtualization

Often useful to programs against a simplified network topology



Physical topology

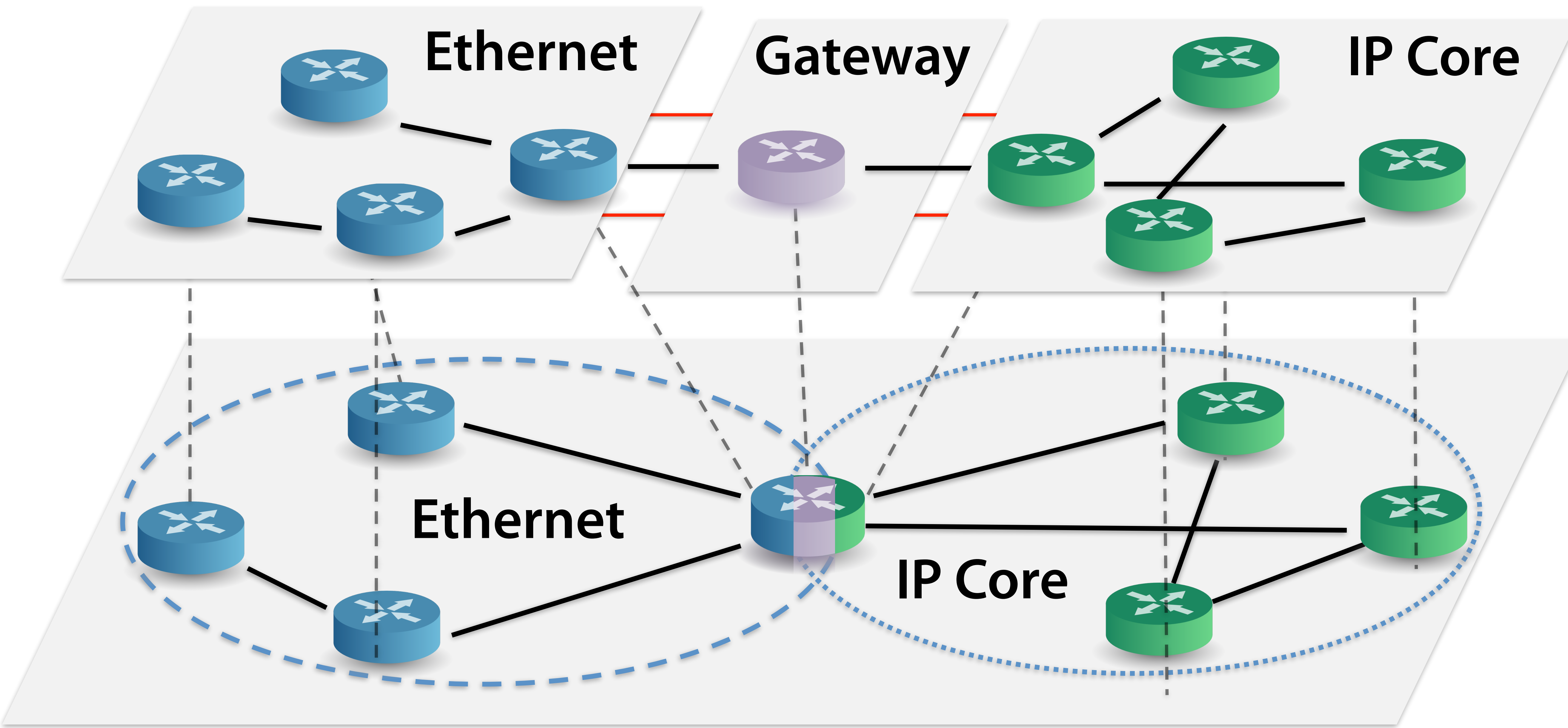


Virtual topology

Benefits:

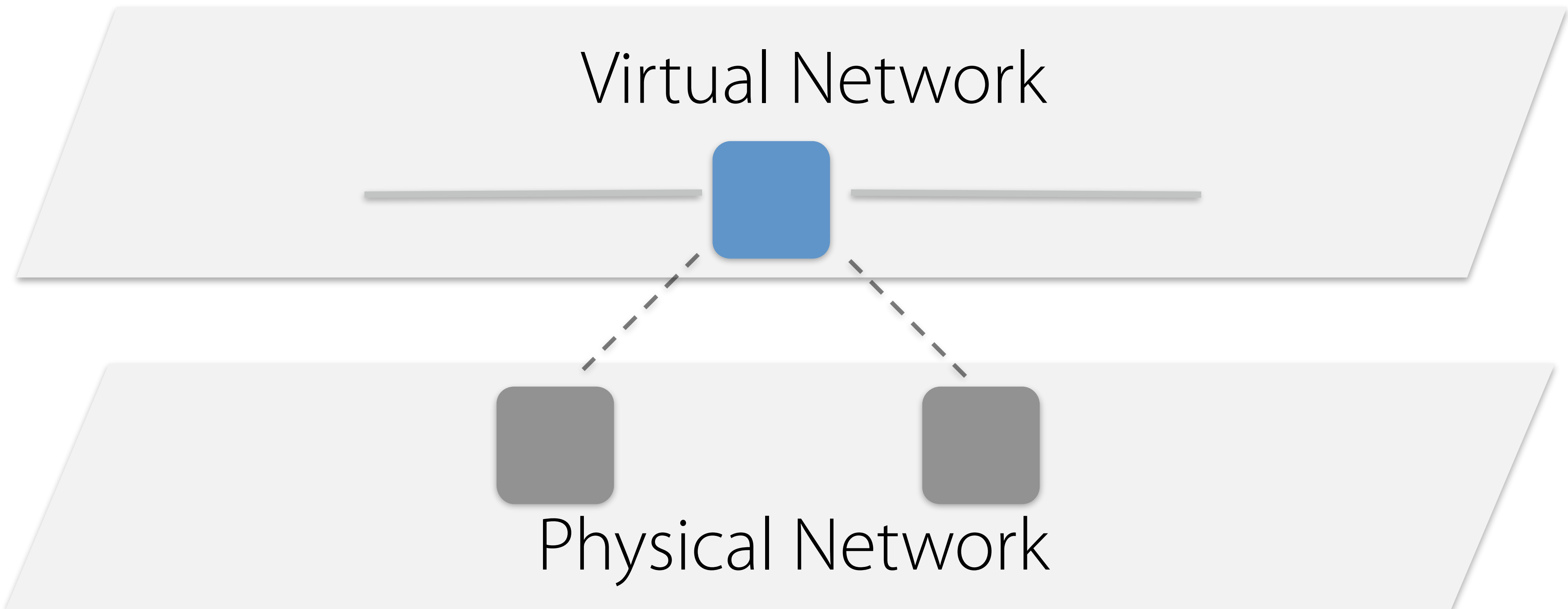
- *Information hiding:* limit what modules see
- *Protection:* limit what modules can do
- *Code reuse:* limit what dependencies modules have

Example: Gateway

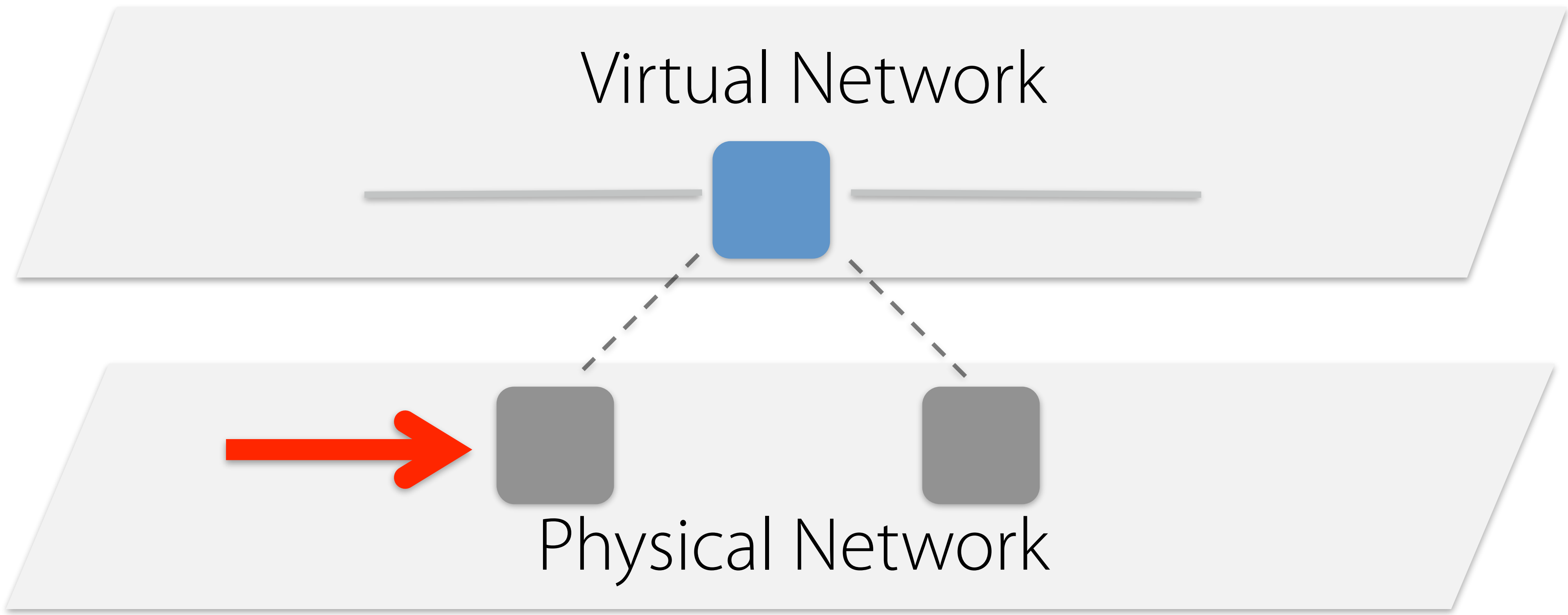


- Left: learning switch on MAC addresses
- Middle: ARP on gateway, plus simple repeater
- Right: shortest-path forwarding on IP prefixes

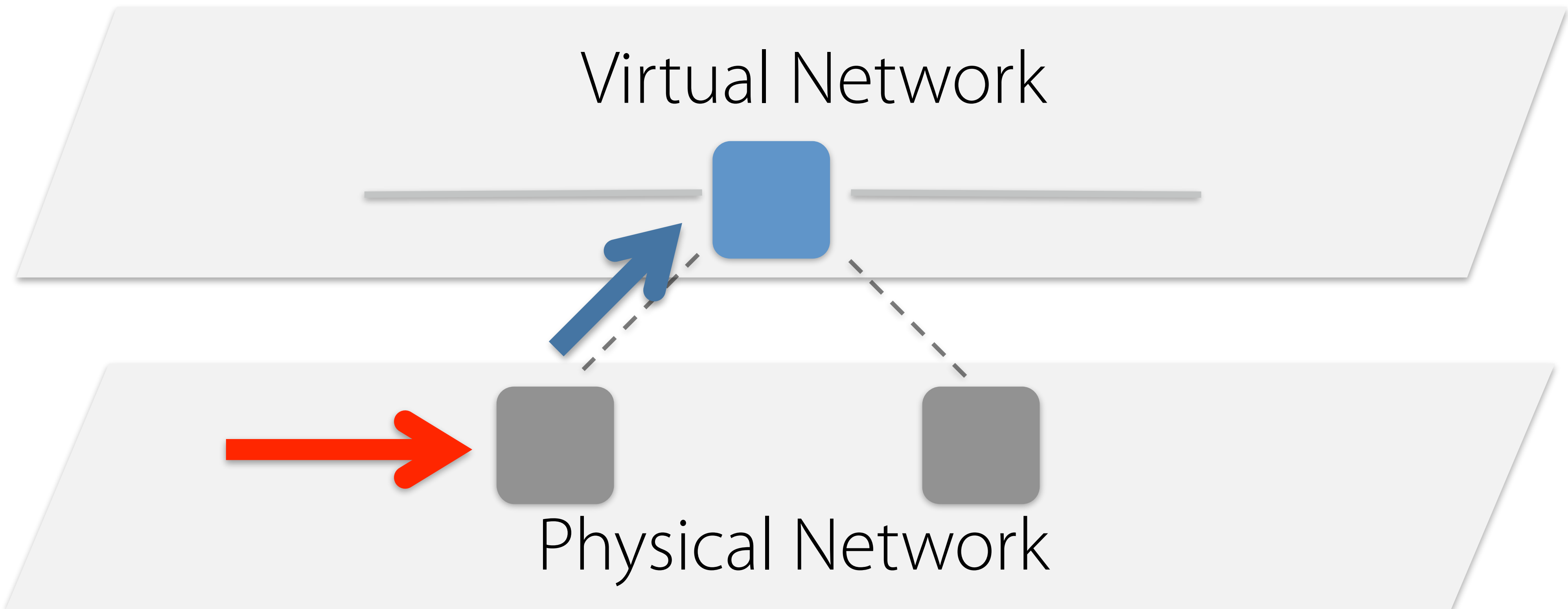
Implementing Virtualization



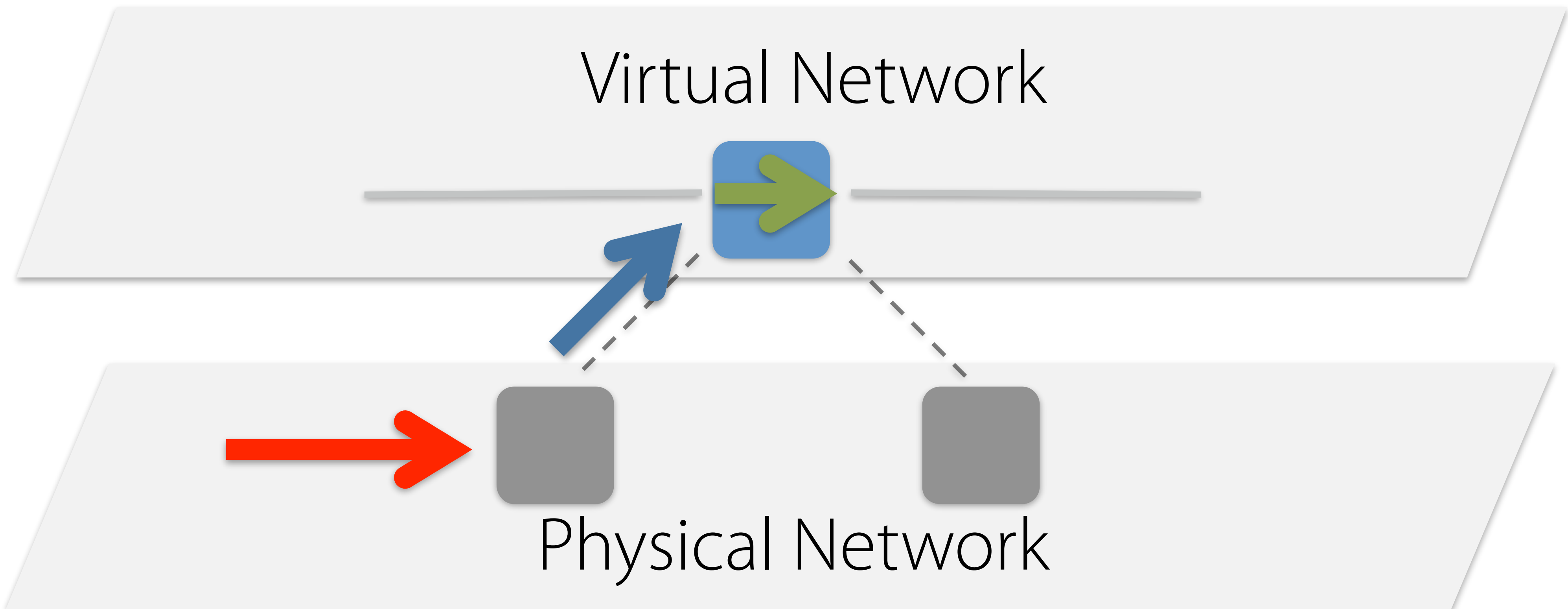
Implementing Virtualization



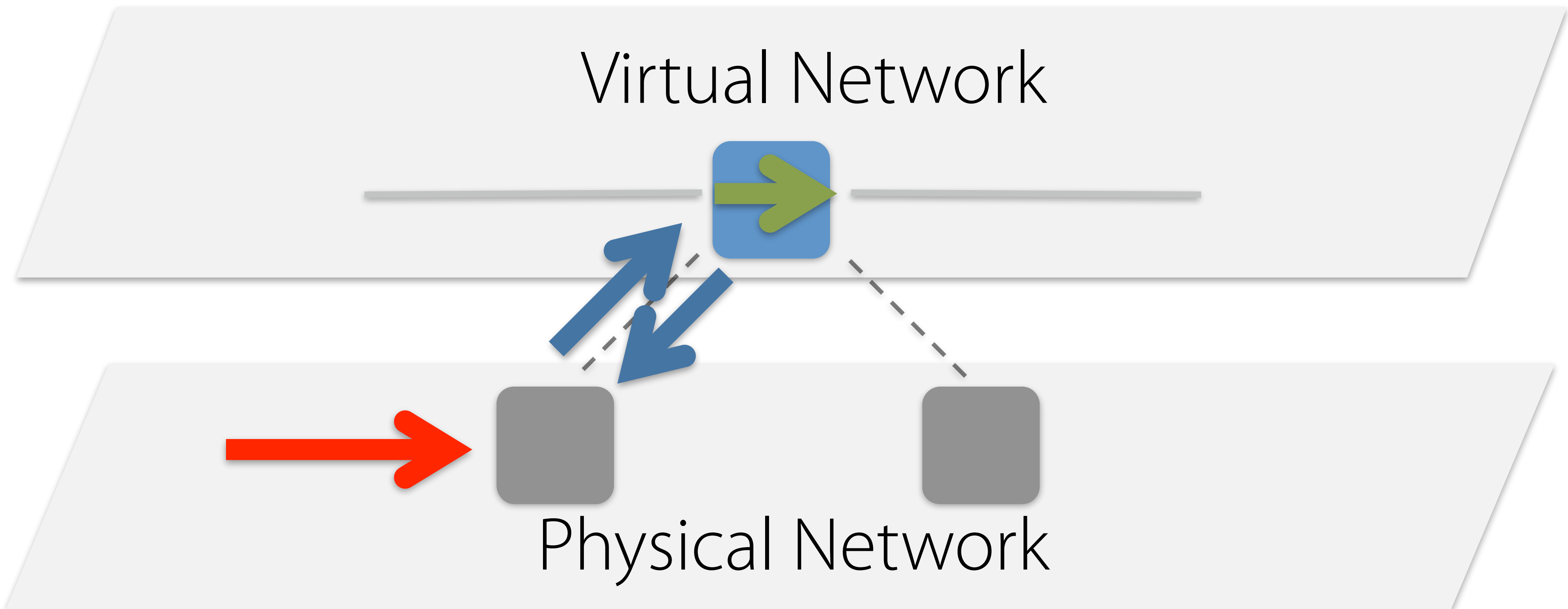
Implementing Virtualization



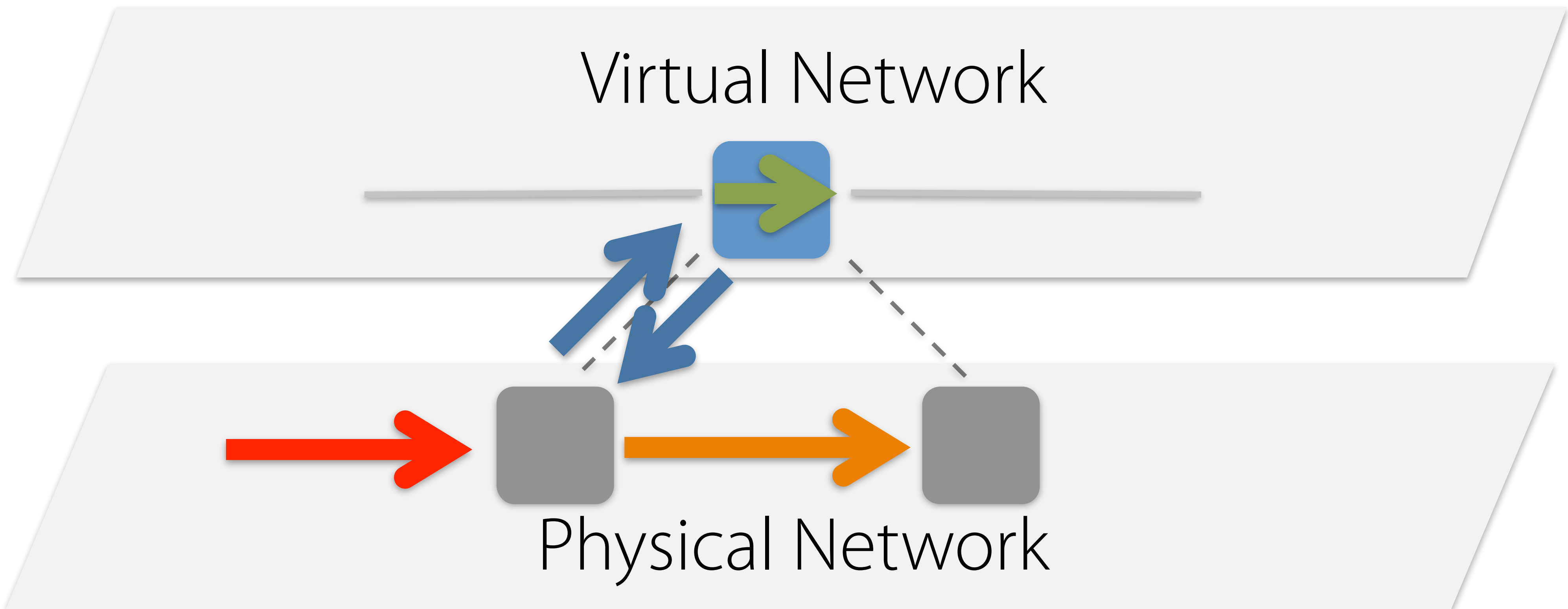
Implementing Virtualization



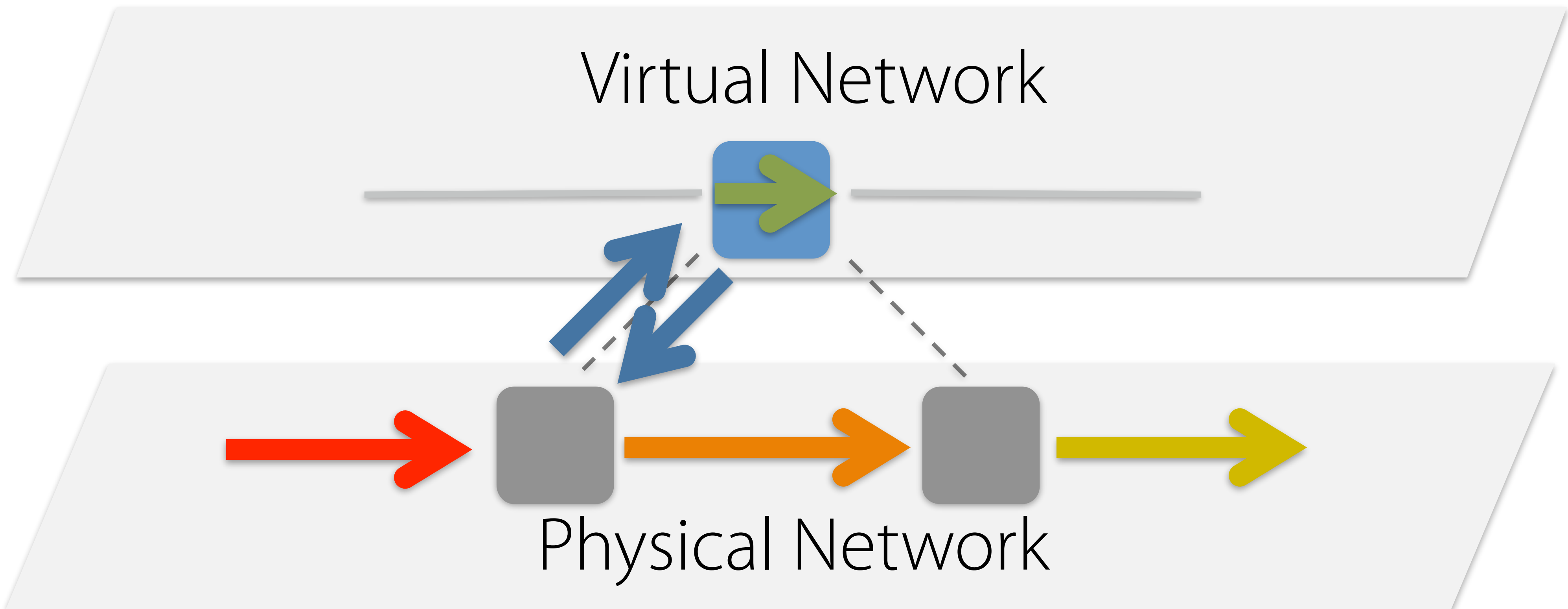
Implementing Virtualization



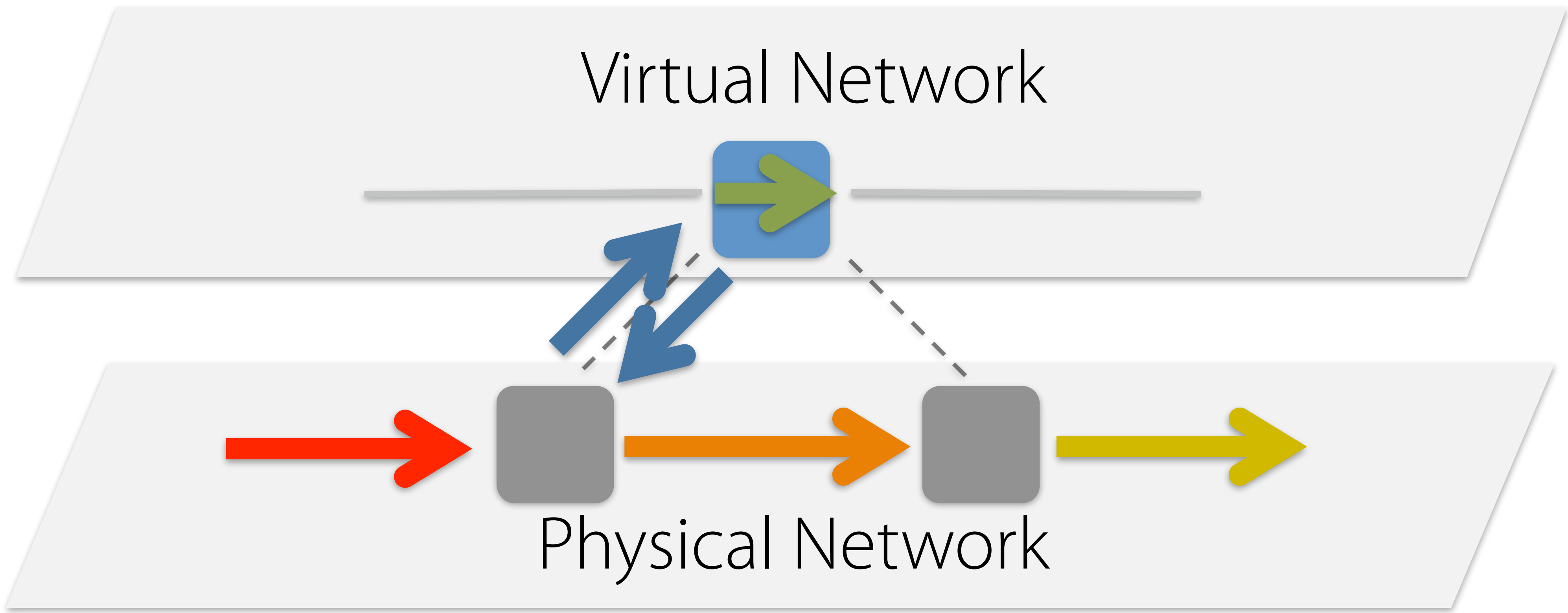
Implementing Virtualization



Implementing Virtualization



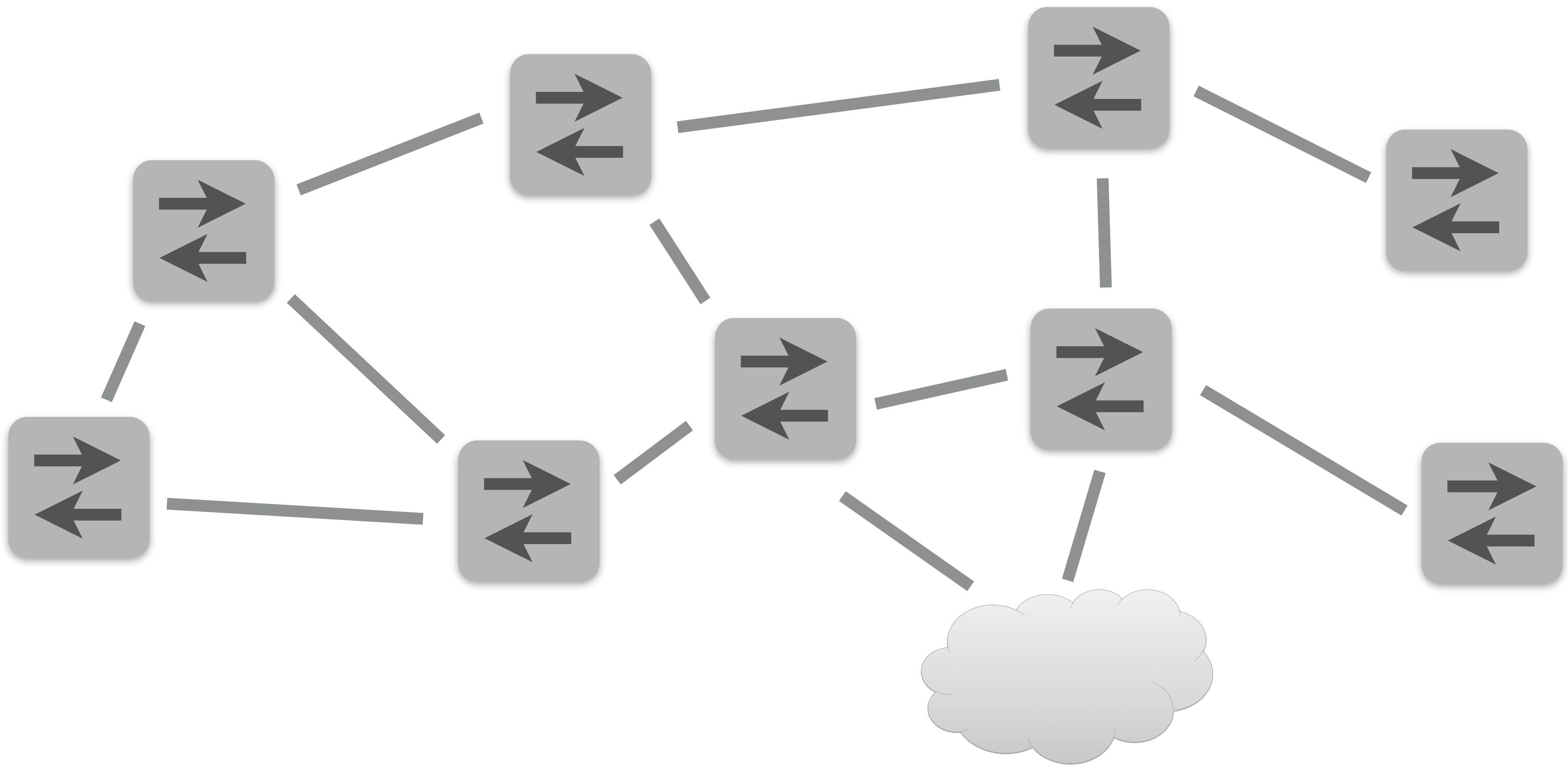
Implementing Virtualization



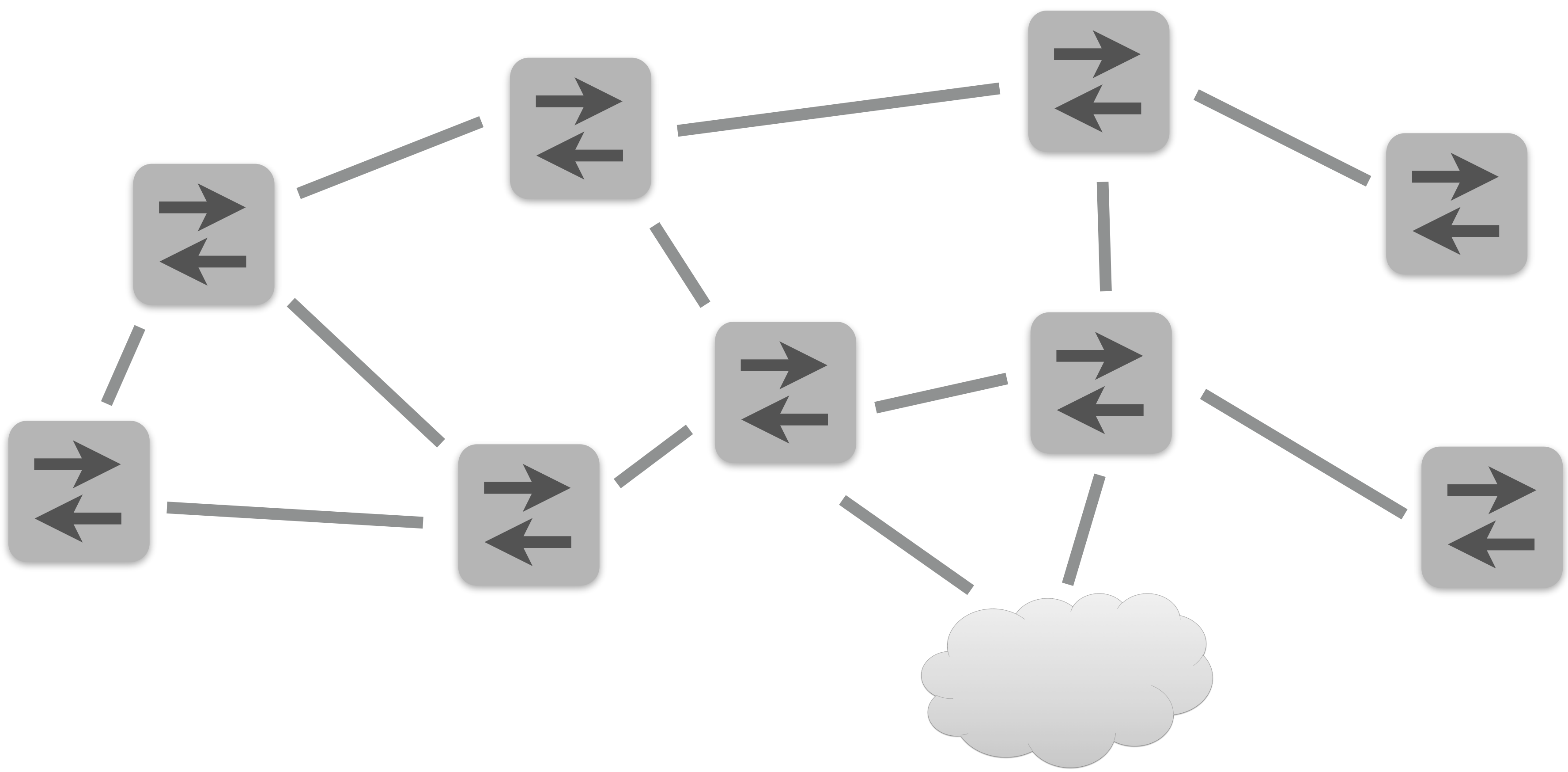
This idiom can be implemented in NetKAT!

```
ingress;
(raise; application; lower; fabric)*;
egress
```

Network Debugging



Network Debugging



Often want to answer questions like:

- Does the network forward packets from A to B?
- Does the network block packets of type X?
- Does the network contain forwarding loops?

Network Debugging

We can encode entire networks as NetKAT policies and check these properties (and others) automatically

Often want to answer questions like:

- Does the network forward packets from A to B?
- Does the network block packets of type X?
- Does the network contain forwarding loops?

Encoding Tables

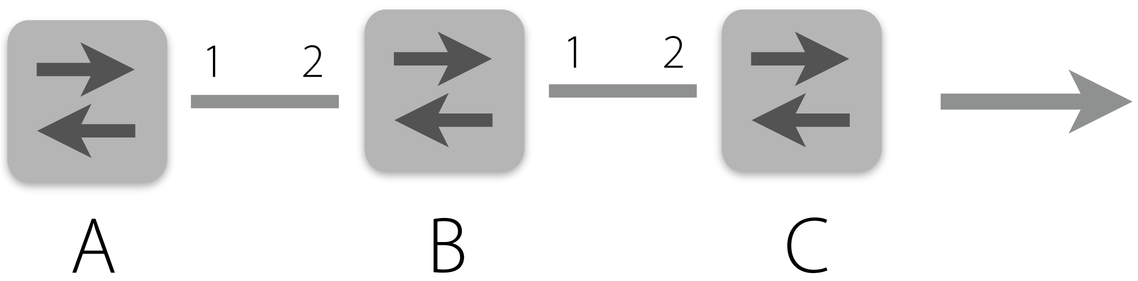
Encoding switch forwarding tables is straightforward using NetKAT's conditionals

Pattern	Actions
`dstport=22`	`Drop`
`srcip=10.0.0.0/8`	`Forward 1`
`*`	`Forward 2`

if dstport=22 **then**
  false
else if srcip=10.0.0.0/8 **then**
  port **:=** 1
else
  port **:=** 2

Encoding Topologies

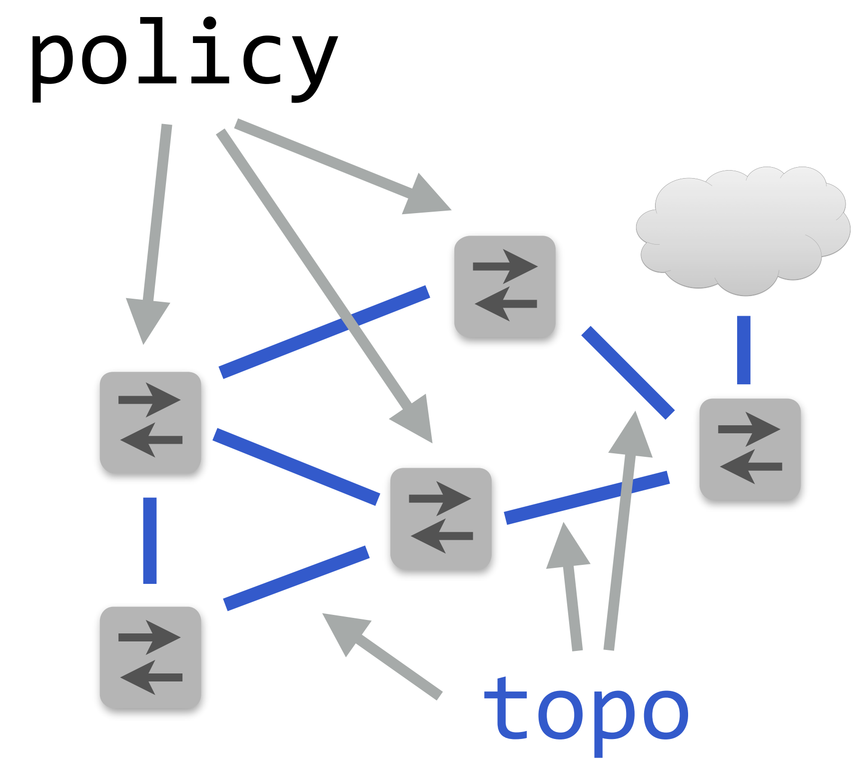
Encoding topologies is also straightforward using NetKAT's tests, modifications, and union



switch=A; port=1; switch:=B; port:=2 +
switch=B; port=2; switch:=A; port:=1 +
switch=B; port=1; switch:=C; port:=2 +
switch=C; port=2; switch:=B; port:=1

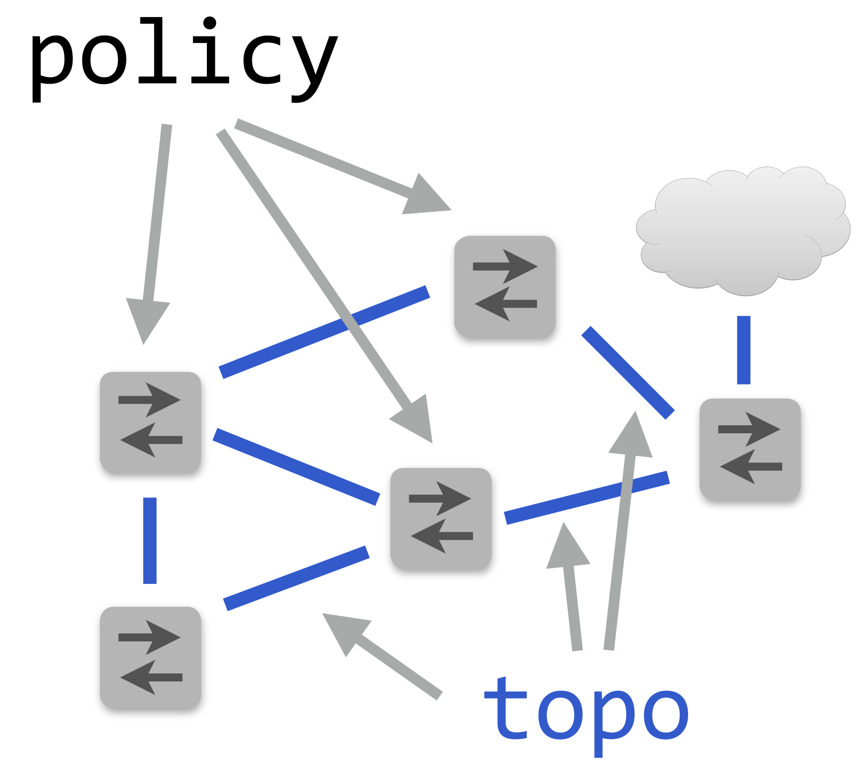
Encoding Networks

A network can be encoded by alternating between policy and topology packet-processing steps



Encoding Networks

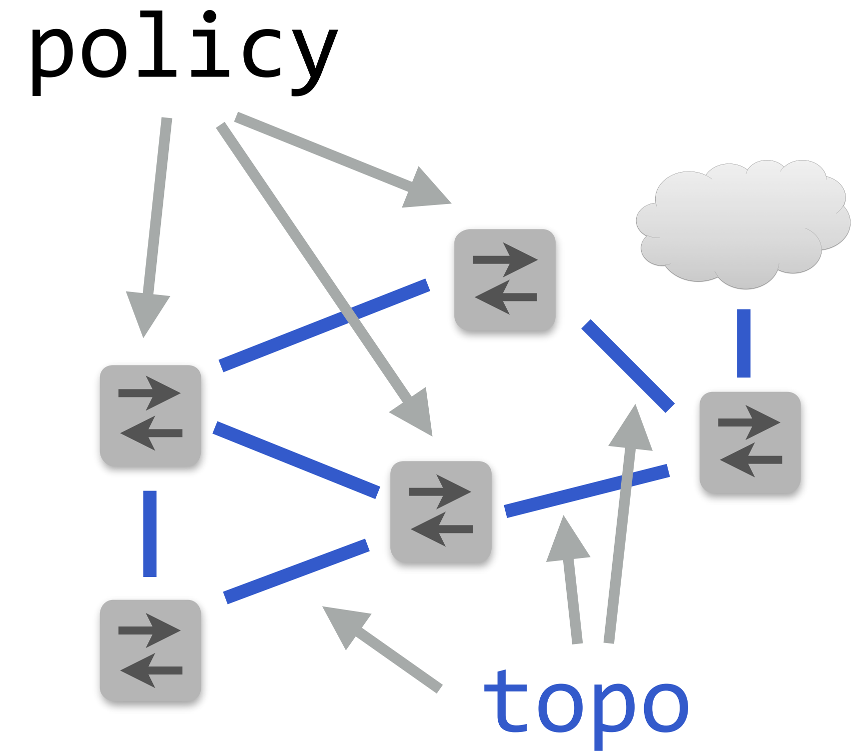
A network can be encoded by alternating between policy and topology packet-processing steps



true

Encoding Networks

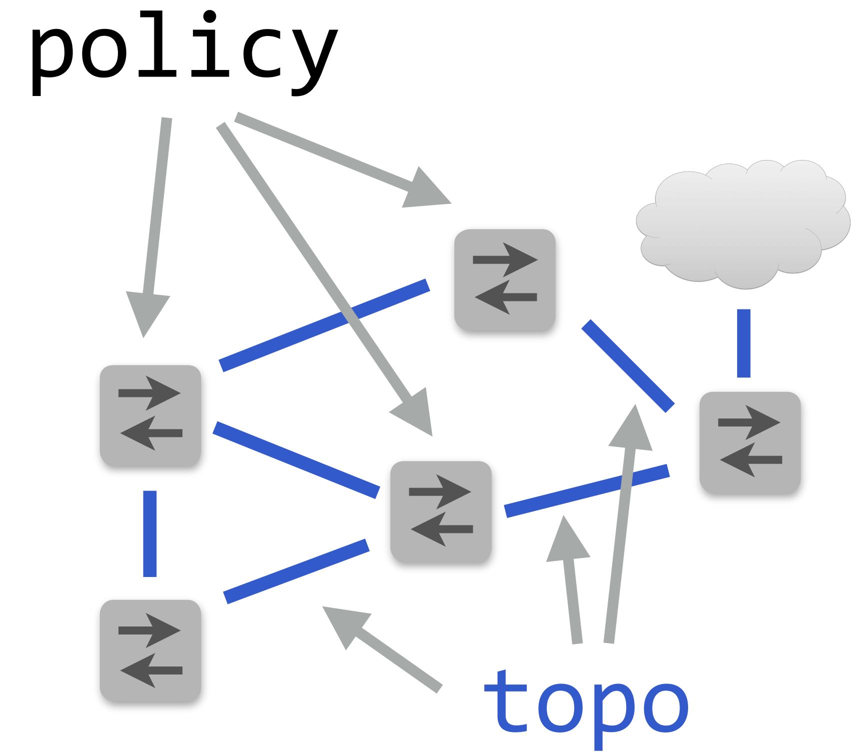
A network can be encoded by alternating between policy and topology packet-processing steps



true
+
(policy; topo)

Encoding Networks

A network can be encoded by alternating between policy and topology packet-processing steps



true

+

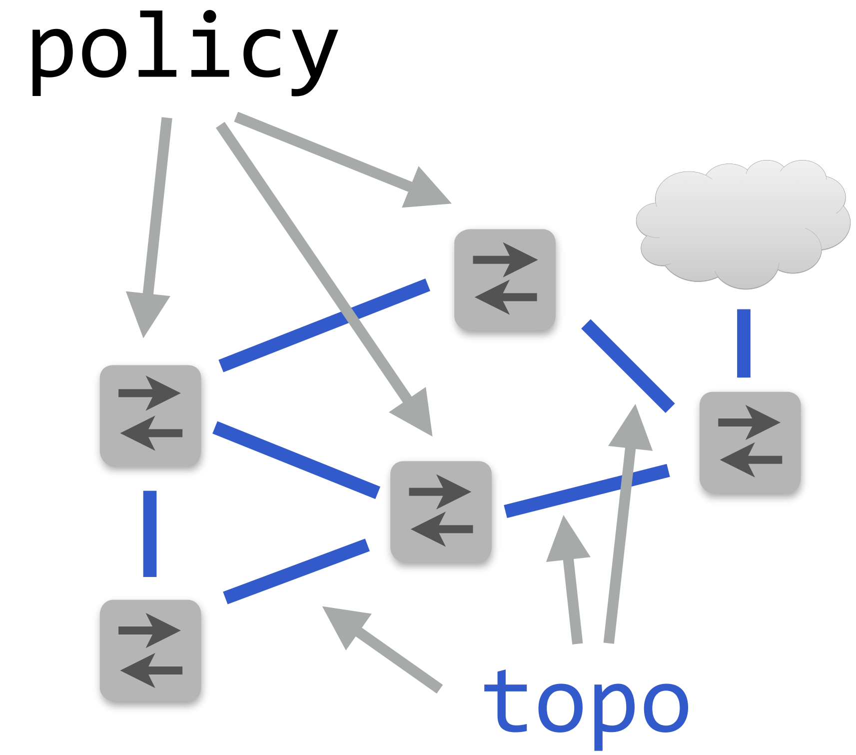
(policy; topo)

+

(policy; topo; policy; topo)

Encoding Networks

A network can be encoded by alternating between policy and topology packet-processing steps



true

+

(policy; topo)

+

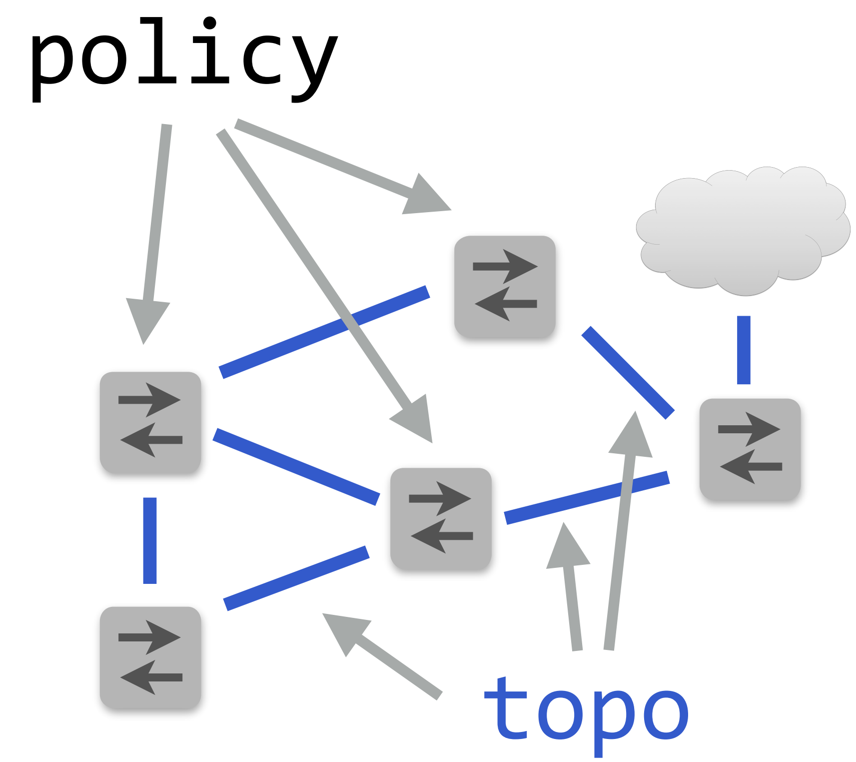
(policy; topo; policy; topo)

+

(policy; topo; policy; topo; policy; topo)

Encoding Networks

A network can be encoded by alternating between policy and topology packet-processing steps



true

+

(policy; topo)

+

(policy; topo; policy; topo)

+

(policy; topo; policy; topo; policy; topo)

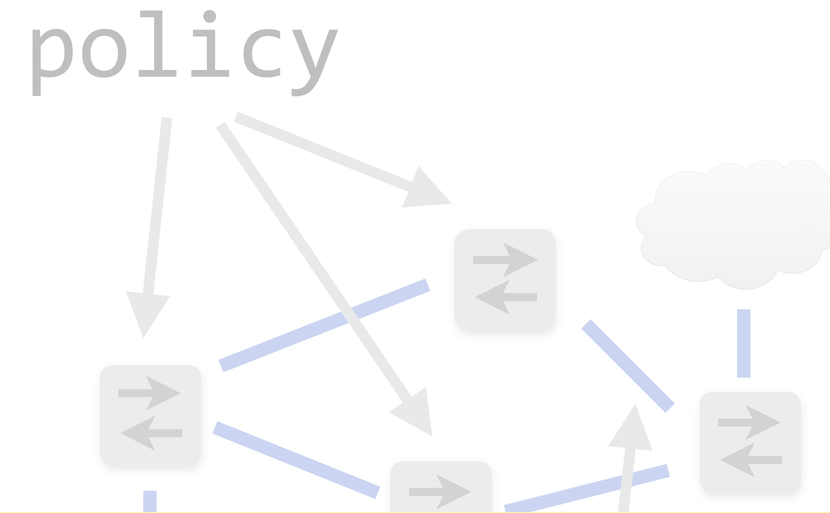
⋮

+

(policy; topo)*

Encoding Networks

A network can be encoded by alternating between policy and topology packet-processing steps



To check whether the network drops packets of type X, check if type=X**;** (policy**;** topo)***** is equivalent to **false**

(policy; topo)

+

(policy; topo; policy; topo)

+

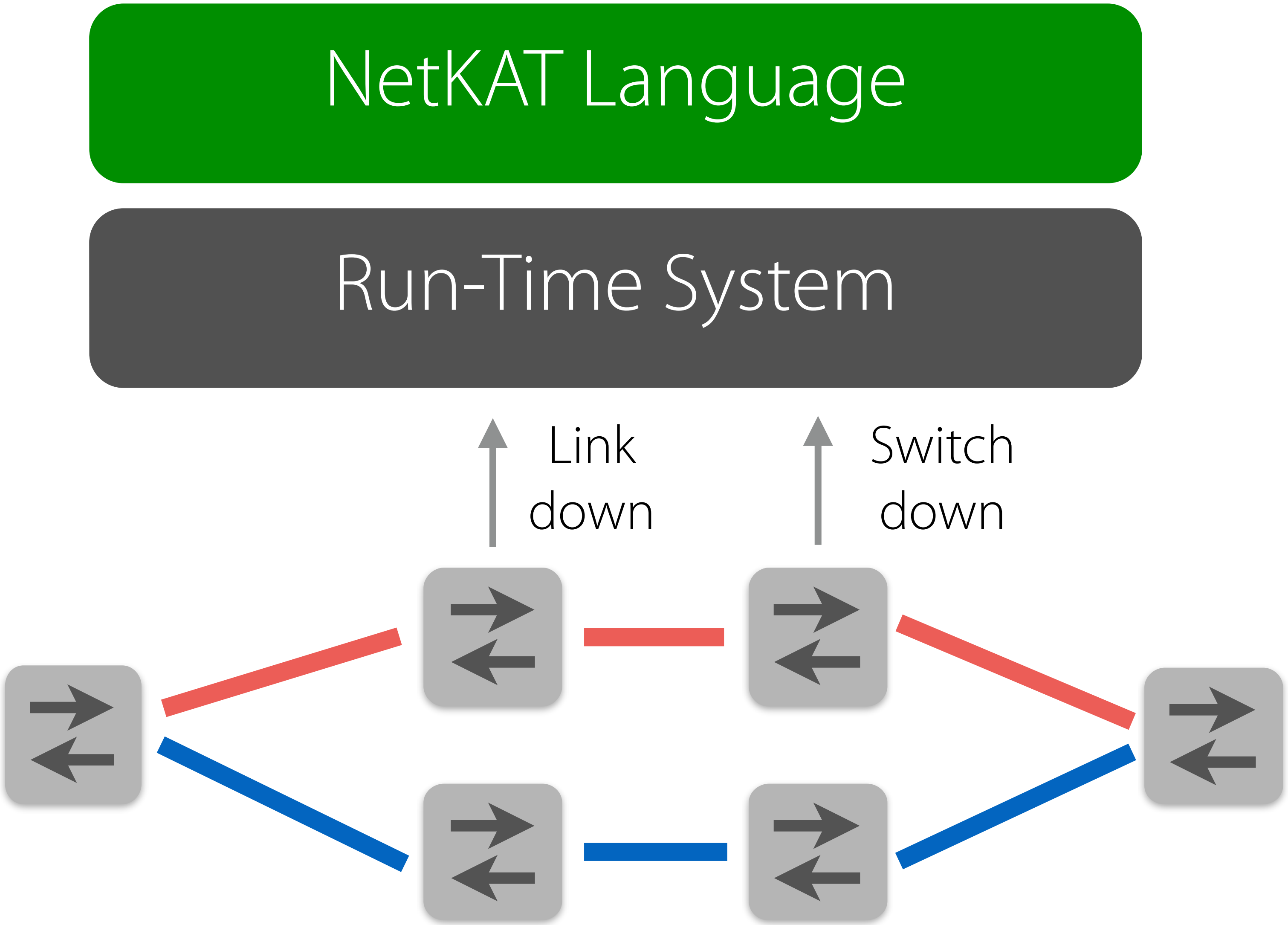
(policy; topo; policy; topo; policy; topo)

⋮

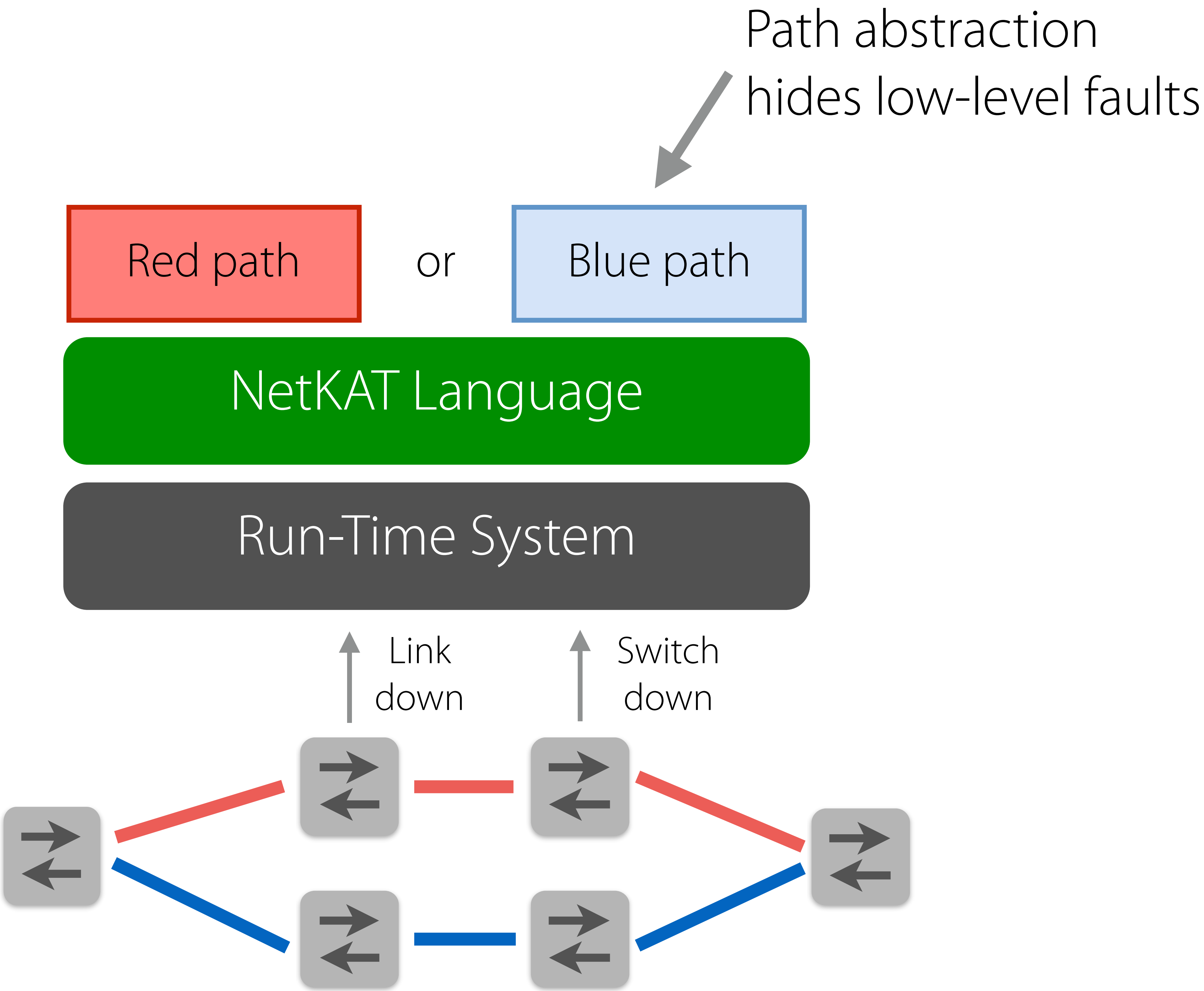
+

(policy; topo)*

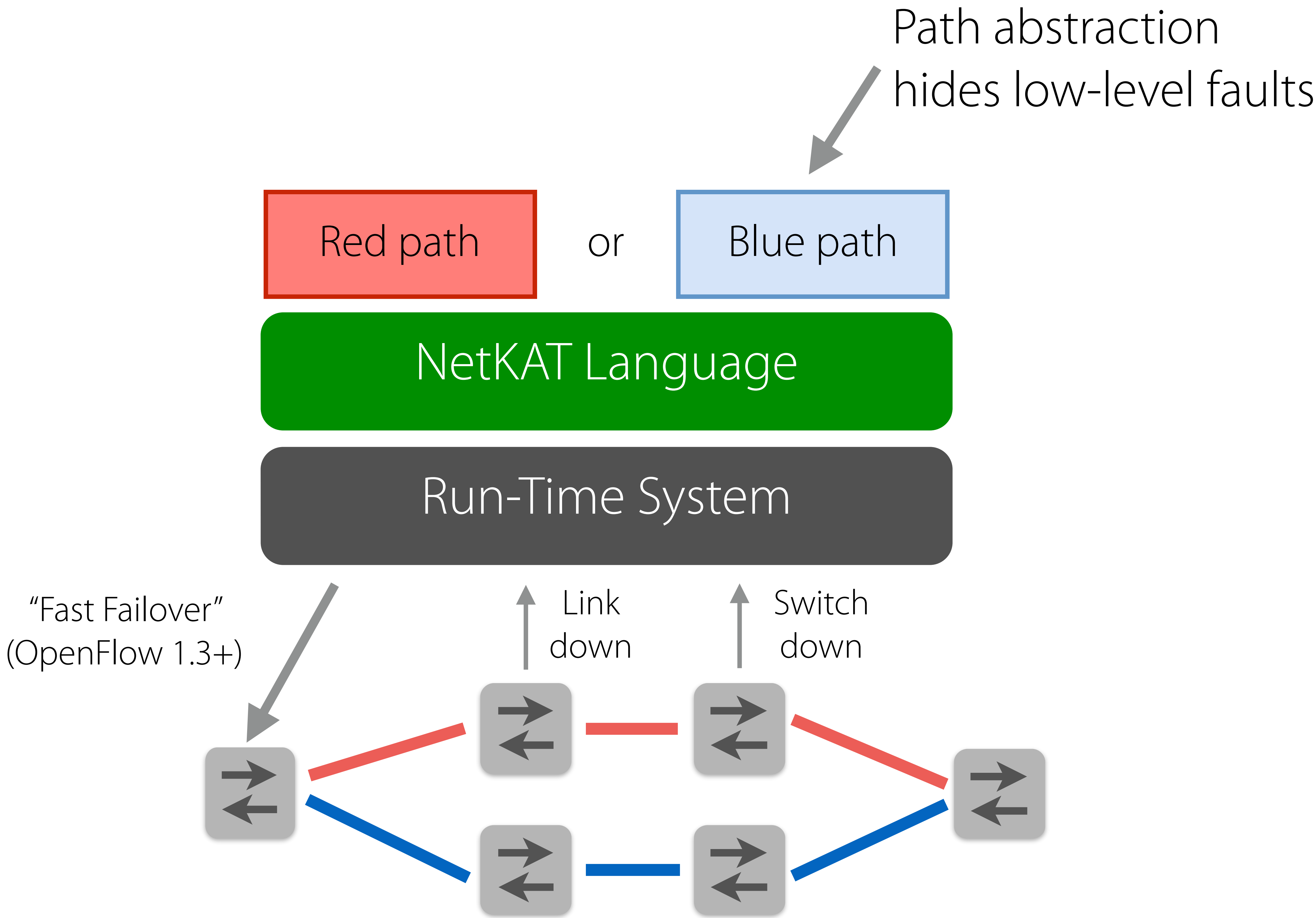
Fault Tolerance



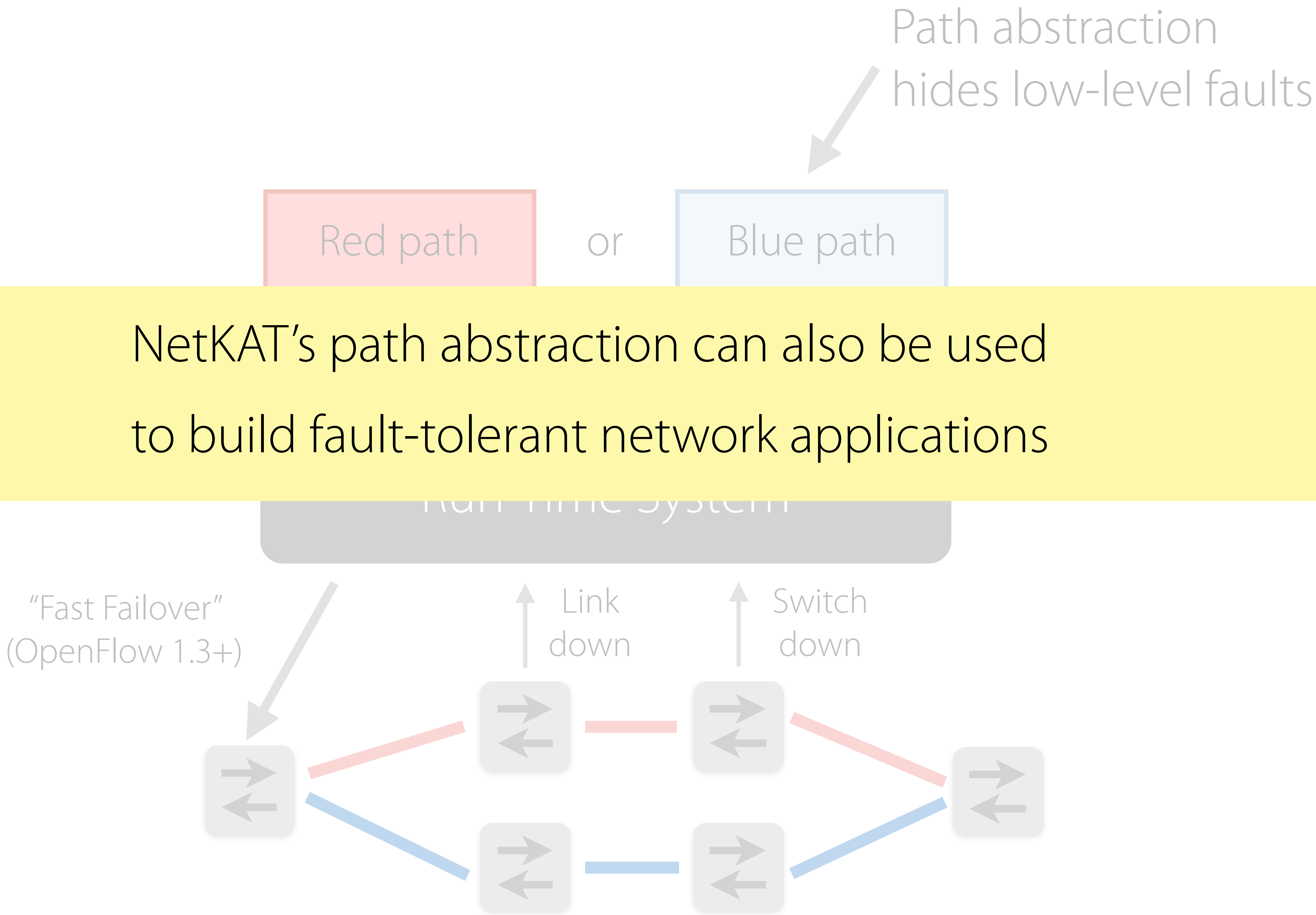
Fault Tolerance



Fault Tolerance



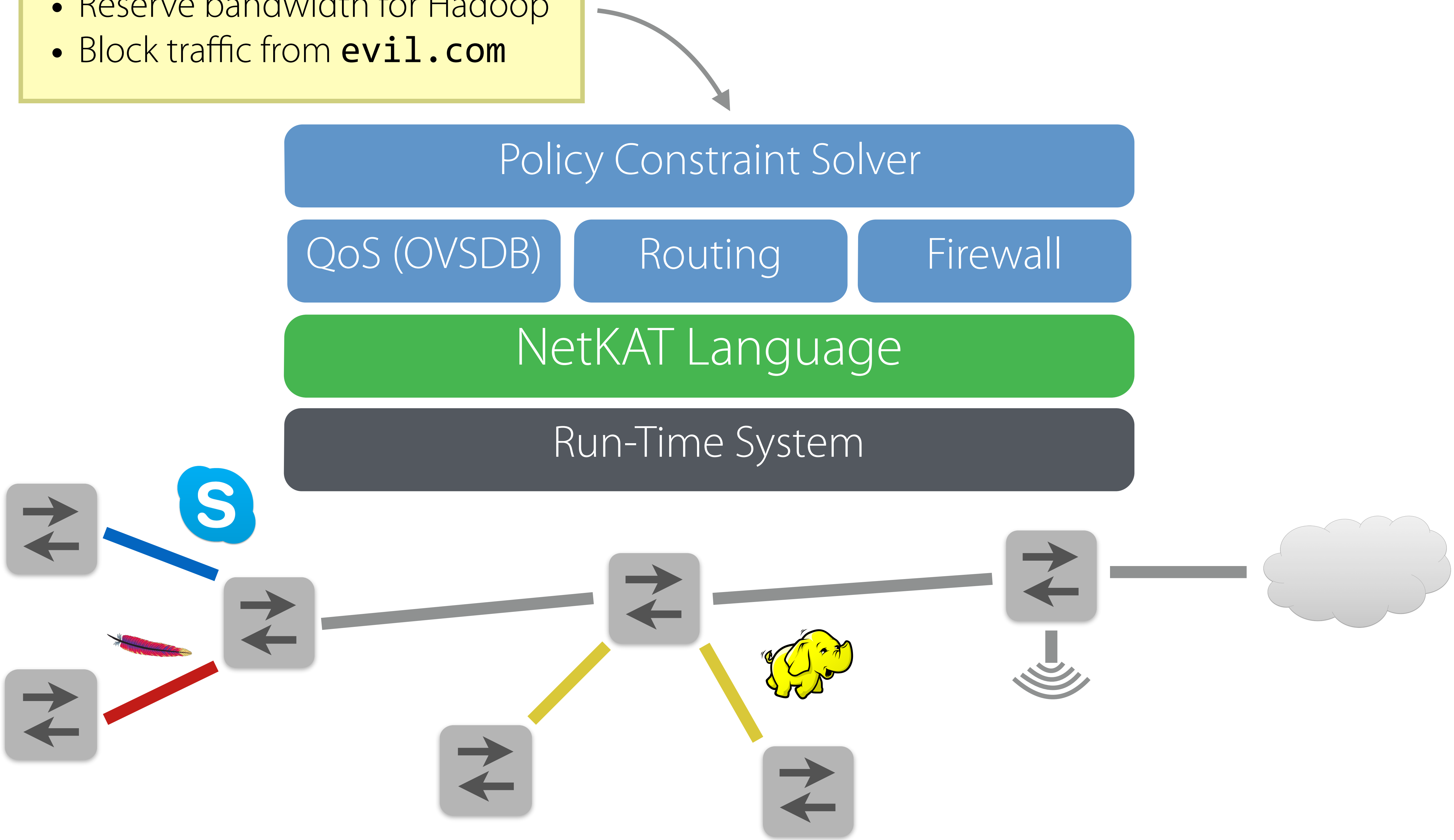
Fault Tolerance



NetKAT's path abstraction can also be used to build fault-tolerant network applications

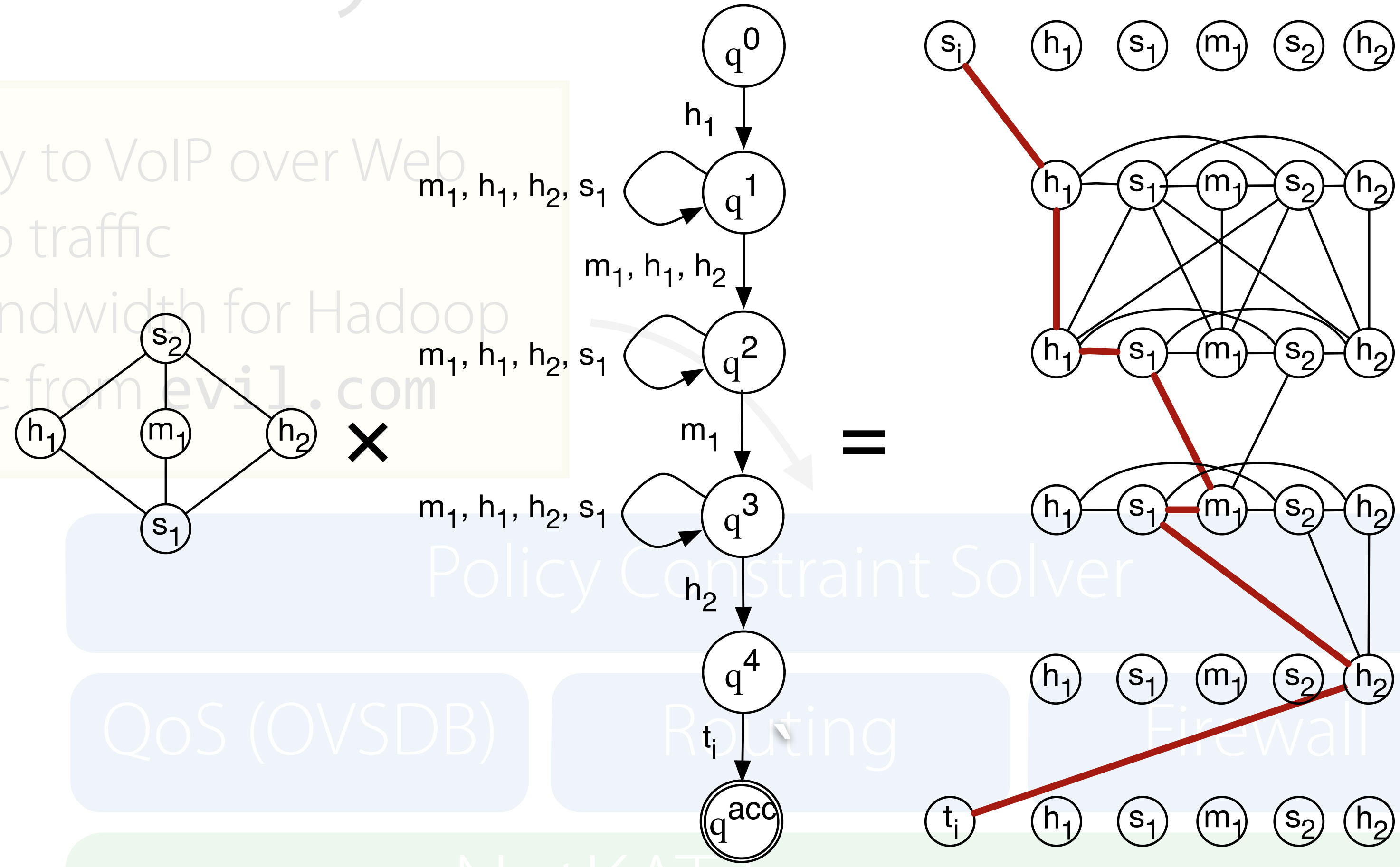
Quality of Service and NFV

- Give priority to VoIP over Web
- DPI on Web traffic
- Reserve bandwidth for Hadoop
- Block traffic from **`evil.com`**



Quality of Service and NFV

- Give priority to VoIP over Web
- DPI on Web traffic
- Reserve bandwidth for Hadoop
- Block traffic from evil.com



Place middlebox functions, provision bandwidth,
and select paths using a constraint solver

Conclusion

- Programming languages have important role to play in the evolution of SDN programming interfaces
- NetKAT policy language provides a solid foundation for expressing and reasoning about packet-processing functions
- Many higher-level abstractions can be built on top of NetKAT
  - Isolated Slices
  - Virtual Networks
  - Network Debugging
  - Fault Tolerance
  - Quality of Service Provisioning
  - Network Function Virtualization

Thank you!

Collaborators

- Carolyn Anderson (Victoria)
- Shrutarshi Basu (Cornell)
- Marco Canini (UC Louvain)
- Andrew Ferguson (Google)
- Rodrigo Fonseca (Brown)
- Jean-Baptiste Jeannin (CMU)
- Dexter Kozen (Cornell)
- Robert Kleinberg (Cornell)
- Shriram Krishnamurthi (Brown)
- Chen Liang (Duke)
- Matthew Milano (Cornell)
- Jennifer Rexford (Princeton)
- Mark Reitblatt (Cornell)
- Cole Schlesinger (Princeton)
- Alexandra Silva (Nijmegen)
- Emin Gün Sirer (Cornell)
- Robert Soulé (Lugano)
- Laure Thompson (Cornell)
- Dave Walker (Princeton)

frenetic

Papers, Code, etc.

http://frenetic-lang.org/