Probabilistic Network Programming

Nate Foster
Cornell

Outline

I: Introduction

- Semantics Primer
- Software-Defined Networking

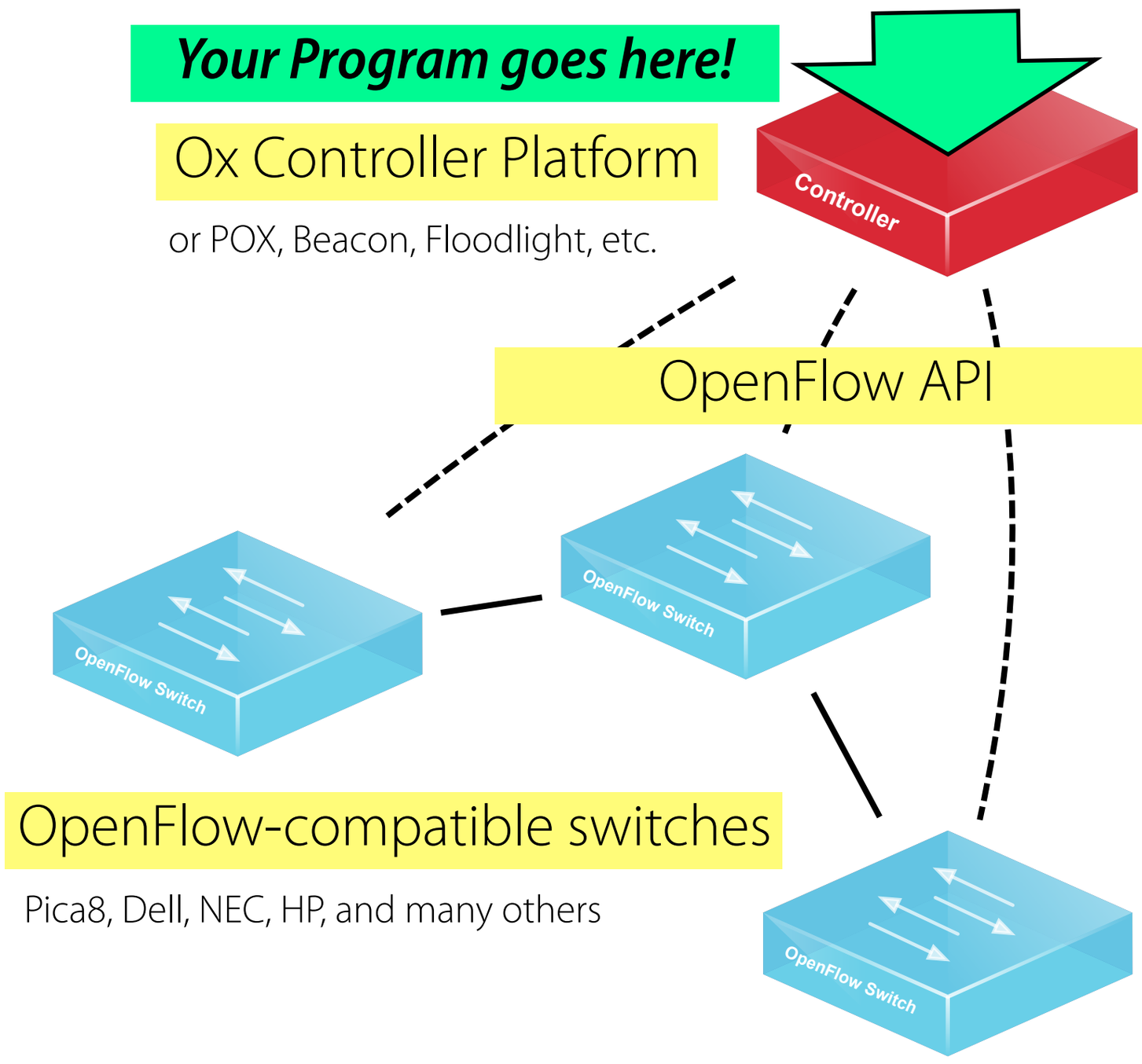
II: NetKAT

- Language Design
- Formal Semantics

III: Probabilistic NetKAT

- Formal Semantics
- Applications

SDN Architecture



Non-Atomic Updates

Priority	Predicate	Action

$\subseteq$

Priority	Predicate	Action
10	SSH	Drop

Must use a **barrier** to synchronize with the switch after installing the first rule...

5	dst_ip = H1	Fwd 1

update re-ordering

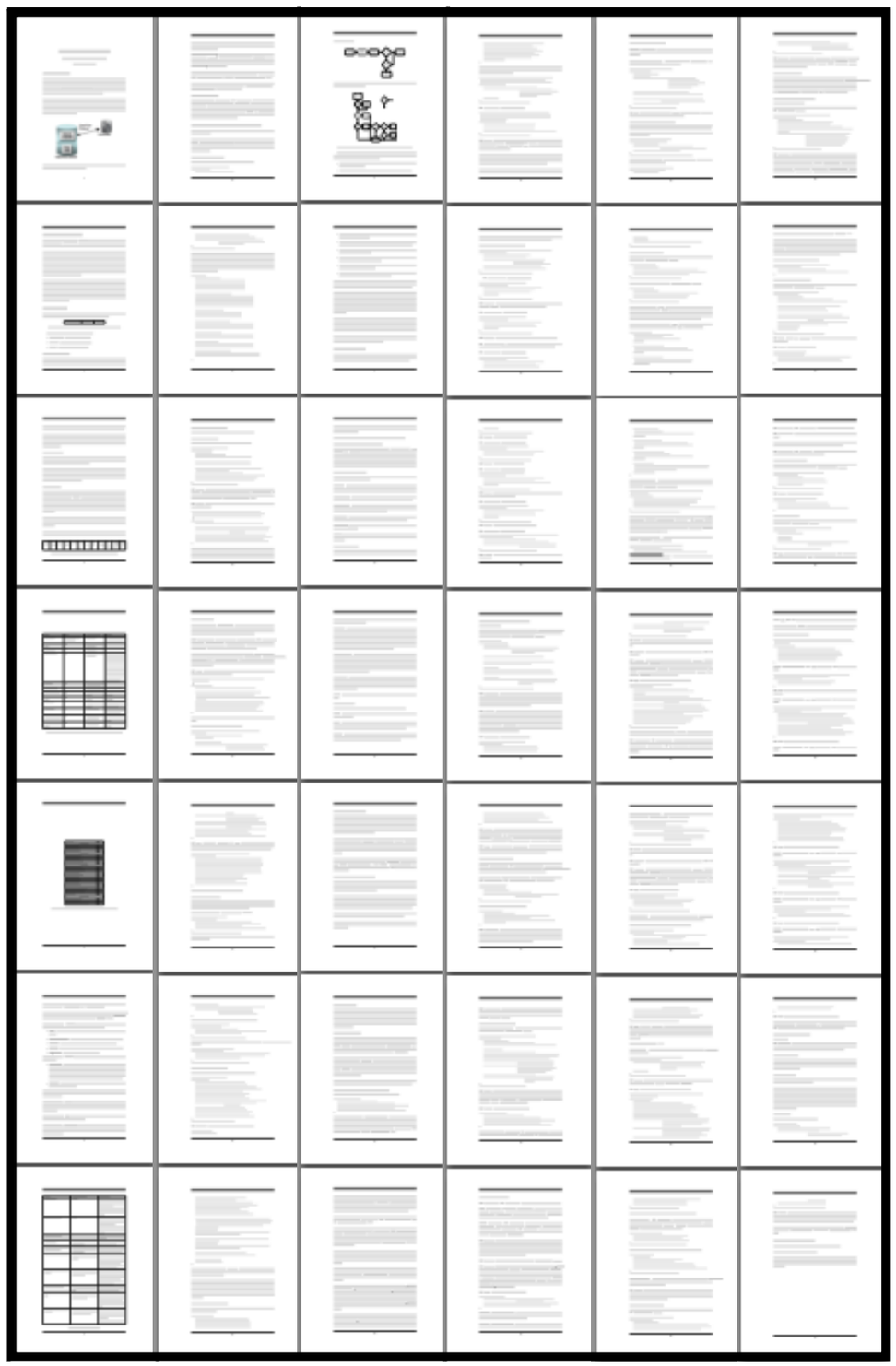
Priority	Predicate	Action
5	dst_ip = H1	Fwd 1
5	dst_ip = H2	Fwd 2

5	dst_ip = H1	Fwd 1

$\subseteq$

Priority	Predicate	Action
10	SSH	Drop
5	dst_ip = H1	Fwd 1
5	dst_ip = H2	Fwd 2

OpenFlow Specification



42 pages...

...of informal prose

...diagrams and flow charts

...and C struct definitions

Featherweight OpenFlow

Syntax

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

Models all features related to packet forwarding, and *all* essential concurrency

Semantics

$$\frac{(outp', outm') = [\![RT]\!](lp)}{\mathbb{S}(sw, pts, RT, \{\!\vert lp \vert\!\} \uplus inp, outp, inm, outm) \xrightarrow{lp} \mathbb{S}(sw, pts, RT, inp, outp' \uplus outp, inm, outm' \uplus outm)} \text{(Pkt-Process)}$$

$$\frac{}{\begin{array}{l}\mathbb{S}(sw, pts, RT, inp, \{\!\vert (sw, pt, pk) \vert\!\} \uplus outp, inm, outm) \mid \mathbb{L}((sw, pt), pks, loc') \\ \longrightarrow\ \mathbb{S}(sw, pts, RT, inp, outp, inm, outm) \mid \mathbb{L}((sw, pt), [pk] +\!\!+ pks, loc')\end{array}} \text{(Send-Wire)}$$

$$\frac{}{\begin{array}{l}\mathbb{L}(loc, pks +\!\!+ [pk], (sw, pt)) \mid \mathbb{S}(sw, pts, RT, inp, outp, inm, outm) \\ \xrightarrow{(sw,pt,pk)}\ \mathbb{L}(loc, pks, (sw, pt)) \mid \mathbb{S}(sw, pts, RT, \{\!\vert (sw, pt, pk) \vert\!\} \uplus inp, outp, inm, outm)\end{array}} \text{(Recv-Wire)}$$

$$\frac{RT' = \text{apply}(\Delta RT, RT)}{\mathbb{S}(sw, pts, RT, inp, outp, \{\!\vert \textbf{FlowMod}\ \Delta RT \vert\!\} \uplus inm, outm) \longrightarrow \mathbb{S}(sw, pts, RT', inp, outp, inm, outm)} \text{(Switch-FlowMod)}$$

$$\frac{pt \in pts}{\mathbb{S}(sw, pts, RT, inp, outp, \{\!\vert \textbf{PktOut}\ pt\ pk \vert\!\} \uplus inm, outm) \longrightarrow \mathbb{S}(sw, pts, RT, inp, \{\!\vert (sw, pt, pk) \vert\!\} \uplus outp, inm, outm)} \text{(Switch-PktOut)}$$

$$\frac{f_{out}(\sigma) \rightsquigarrow (sw, SM, \sigma')}{\mathbb{C}(\sigma, f_{in}, f_{out}) \mid \mathbb{M}(sw, SMS, CMS) \longrightarrow \mathbb{C}(\sigma', f_{in}, f_{out}) \mid \mathbb{M}(sw, [SM] +\!\!+ SMS, CMS)} \text{(Ctrl-Send)}$$

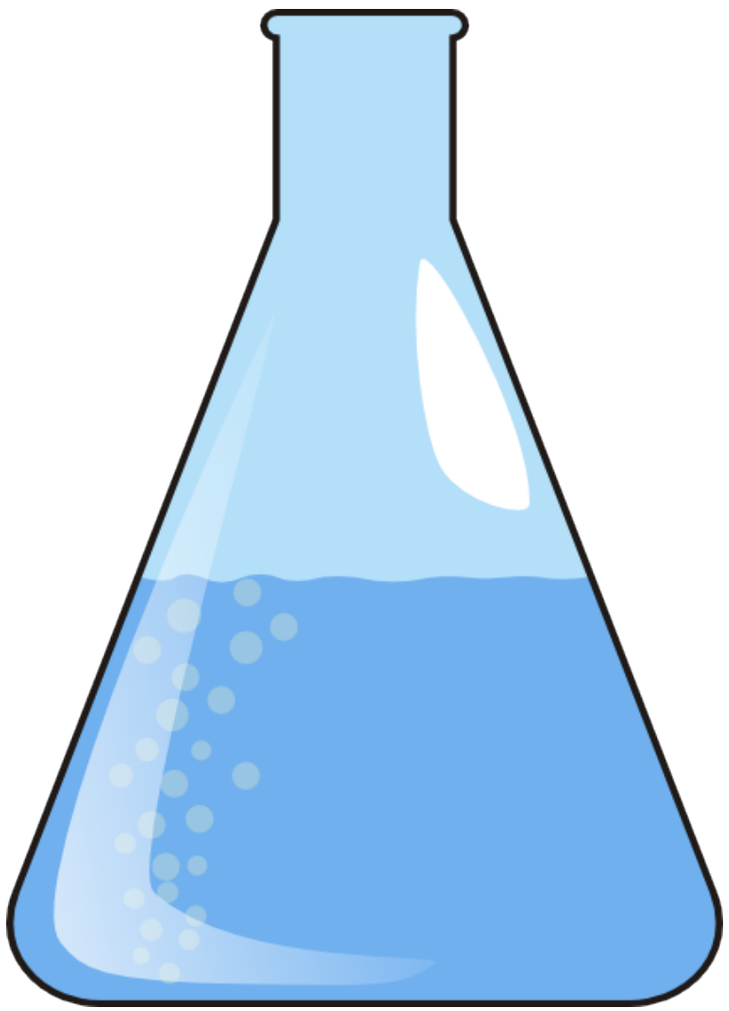
$$\frac{f_{in}(sw, \sigma, CM) \rightsquigarrow \sigma'}{\mathbb{C}(\sigma, f_{in}, f_{out}) \mid \mathbb{M}(sw, SMS, CMS +\!\!+ [CM]) \longrightarrow \mathbb{C}(\sigma', f_{in}, f_{out}) \mid \mathbb{M}(sw, SMS, CMS)} \text{(Ctrl-Recv)}$$

$$\frac{SM \neq \textbf{BarrierRequest}\ n}{\begin{array}{l}\mathbb{M}(sw, SMS +\!\!+ [SM], CMS) \mid \mathbb{S}(sw, pts, RT, inp, outp, inm, outm) \\ \longrightarrow\ \mathbb{M}(sw, SMS, CMS) \mid \mathbb{S}(sw, pts, RT, inp, outp, \{\!\vert SM \vert\!\} \uplus inm, outm)\end{array}} \text{(Switch-Recv-Ctrl)}$$

$$\frac{}{\begin{array}{l}\mathbb{M}(sw, SMS +\!\!+ [\textbf{BarrierRequest}\ n], CMS) \mid \mathbb{S}(sw, pts, RT, inp, outp, \emptyset, outm) \\ \longrightarrow\ \mathbb{M}(sw, SMS, CMS) \mid \mathbb{S}(sw, pts, RT, inp, outp, \emptyset, \{\!\vert \textbf{BarrierReply}\ n \vert\!\} \uplus outm)\end{array}} \text{(Switch-Recv-Barrier)}$$

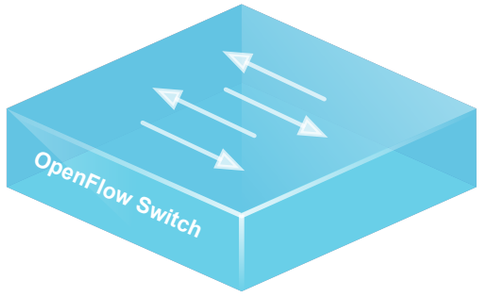
$$\frac{}{\begin{array}{l}\mathbb{S}(sw, pts, RT, inp, outp, inm, \{\!\vert CM \vert\!\} \uplus outm) \mid \mathbb{M}(sw, SMS, CMS) \\ \longrightarrow\ \mathbb{S}(sw, pts, RT, inp, outp, inm, outm) \mid \mathbb{M}(sw, SMS, [CM] +\!\!+ CMS)\end{array}} \text{(Switch-Send-Ctrl)}$$

Chemical Abstract Machine



- A general technique for modeling concurrent systems using operational semantics
- Proposed by Berry and Boudol [POPL '90]
- Intuition: computational components are "molecules" in a "solution" that can react with each other non-deterministically

OpenFlow Molecules



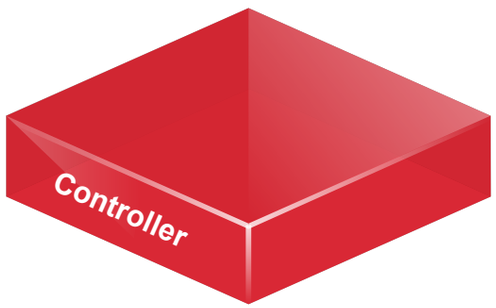
Switches

- $\mathbb{S}$(sw, T, in, out, msgs)
- sw: unique identifier
- T: flow table
- in/out: buffered packets
- msgs: control messages

Data Links

- $\mathbb{L}$(sw,pt,pkts,sw',pt')
- sw/pt: source
- sw'/pt': target
- pkts: queued packets

OpenFlow Molecules



Controller

- $\mathbb{C}(\sigma, f_{in}, f_{out})$
- σ: state
- f_{in}: ingress function
- f_{out}: output function

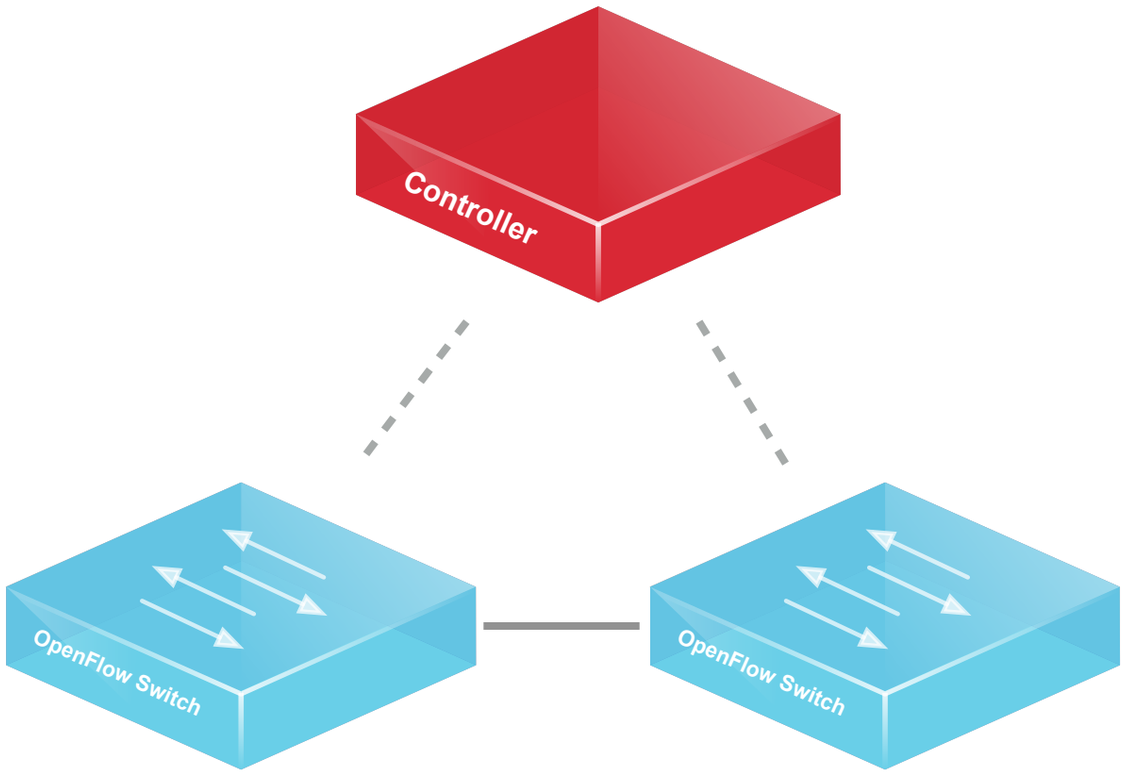
Control Links

- $\mathbb{M}$(sw,msgs)
- msgs: control messages

OpenFlow Messages

```
m ::= FlowMod f     (* switch-controller messages *)
    | BarrierReq n
    | Barrier Rep n
    | PktIn pk pt
    | PktOut pk pt
f ::= Add(n, pat, act)     (* table manipulation *)
    | Del pat
```

OpenFlow Network



To model an OpenFlow network in we use:

- One ℂ molecule
- An 𝕄 molecule per switch
- An 𝕊 molecule per switch
- An 𝕃 molecule per link
- The overall model is a concurrent composition "|" of these molecules

Reaction

$$\frac{Sys_1 \rightarrow Sys_1'}{Sys_1 \mid Sys_2 \rightarrow Sys_1' \mid Sys_2}$$

Intuition: if any sub-collection of molecules in the system react with each other, then the whole system may also exhibit the same reaction

Structural Congruence

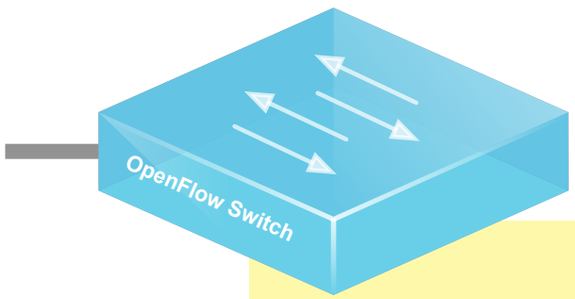
$$\frac{Sys_1 \equiv Sys_1' \qquad Sys_1' \rightarrow Sys_2' \qquad Sys_2' \equiv Sys_2}{Sys_1 \rightarrow Sys_2}$$

$$Sys_1 \mid Sys_2 \equiv Sys_2 \mid Sys_1$$

$$Sys_1 \mid (Sys_2 \mid Sys_3) \equiv (Sys_1 \mid Sys_2) \mid Sys_3$$

Intuition: concurrent composition ("|") is both associative and commutative

Switch Ingress



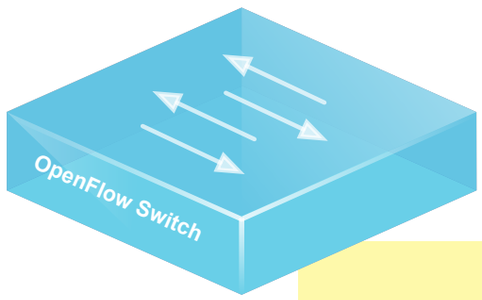
𝕃(sw, pt, pkts @ [pkt], sw' ,pt') |
𝕊(sw', T, in, out, msgs)

→

𝕃(sw, pt, pkts, sw', pt') |
𝕊(sw', T, in ∪ {(pkt,pt')}, out, msgs)

Intuition: a switch can input a packet from an adjacent link into its ingress buffer

Forwarding



$$[\![T]\!]\ (pkt,pt) = (out', msgs')$$

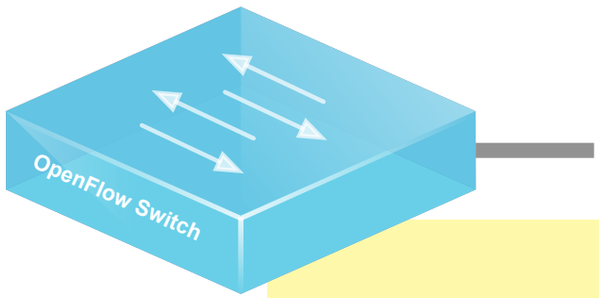
$$\mathbb{S}(sw, T, in \cup \{(pkt,pt)\}, out, msgs)$$

$$\rightarrow$$

$$\mathbb{S}(sw, T, in, out \cup out', msgs \cup msgs')$$

Intuition: a switch processes a single packet using its flow table, which generates zero or more output packets and zero or more control messages

Switch Egress



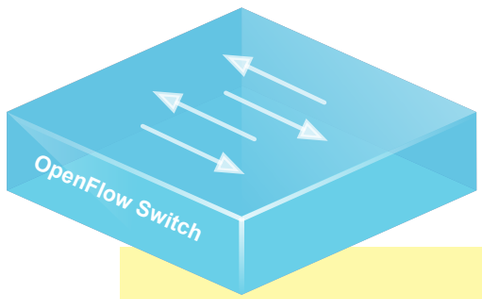
𝕊(sw, T, in, out ∪ {(pkt,pt)}, msgs) |
𝕃(sw, pt, pkts, sw’ ,pt’)

→

𝕊(sw, T, in, out, msgs) |
𝕃(sw, pt, pkt :: pkts, sw’, pt’)

Intuition: a switch can output a packet from its output buffer onto the adjacent link

Add



$$[\![\text{Add(n,pat,act)}]\!]\ T = T'$$

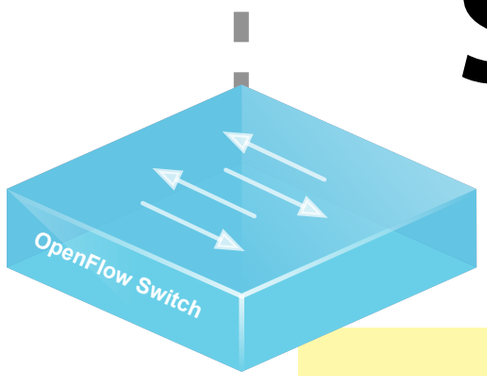
$$\mathbb{S}(\text{sw}, T, \text{in}, \text{out}, \text{msgs} \cup \{\text{FlowMod (Add (n,pat,act))}\})$$

$$\rightarrow$$

$$\mathbb{S}(\text{sw}, T', \text{in}, \text{out}, \text{msgs})$$

Intuition: a switch may modify its flow table by adding a rule sent previously by the controller...

Switch-to-Controller



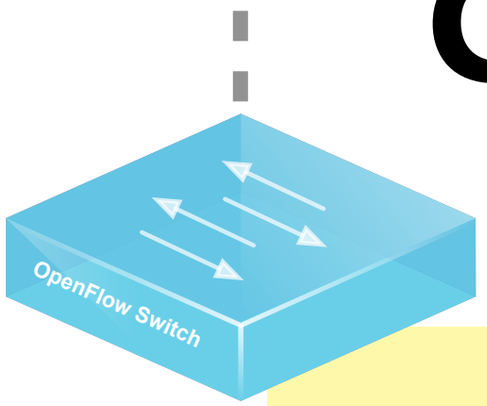
$\mathbb{S}$(sw, T, in, out, msgs ∪ {m}) |
$\mathbb{M}$(sw, msgs)

→

$\mathbb{S}$(sw, T, in, out, msgs) |
$\mathbb{M}$(sw, m:: msgs)

Intuition: a switch can send a message to the controller by moving it from its internal buffer to the shared queue in the controller link

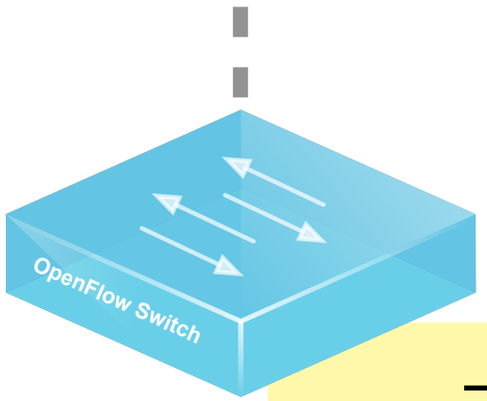
Controller-to-Switch



$$\frac{m \neq \text{BarrierReq } n}{\mathbb{M}(sw, msgs @ [m]) \mid \mathbb{S}(sw, T, in, out, msgs) \rightarrow \mathbb{M}(sw, msgs) \mid \mathbb{S}(sw, T, in, out, msgs \cup \{m\})}$$

Intuition: a switch can receive a controller message by moving it from the controller link to its internal buffer

Barrier

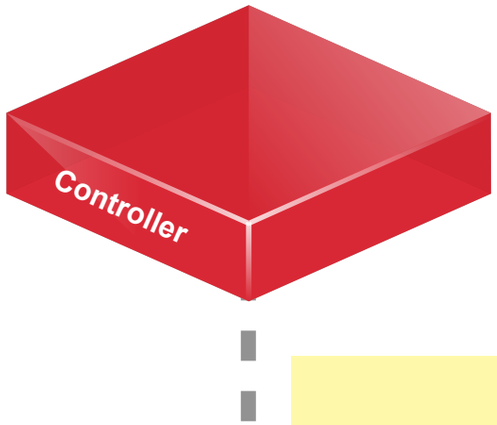


𝕄(sw, @ [BarrierReq n]) |
𝕊(sw, T, in, out, {})

→

𝕄(sw, msgs) |
𝕊(sw, T, in, out ∪ {BarrierRep n} , {})

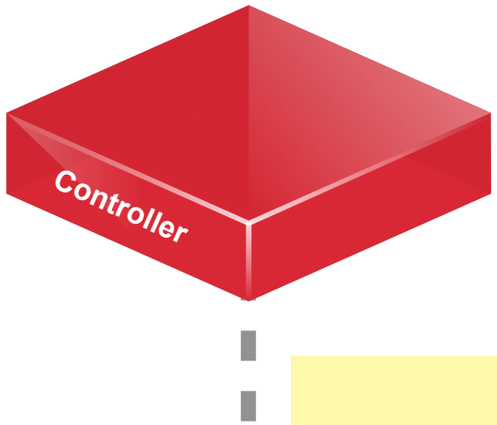
Intuition: a switch can receive a barrier request message *only* if it has processed all outstanding locally-buffered messages



Controller Input

$$\frac{f_{in}(\sigma, sw, m) = \sigma'}{\begin{array}{l}\mathbb{M}(sw, msgs @ [m]) \mid \\ \mathbb{C}(\sigma, f_{in}, f_{out})\end{array} \rightarrow \begin{array}{l}\mathbb{M}(sw, msgs) \mid \\ \mathbb{C}(\sigma', f_{in}, f_{out})\end{array}}$$

Intuition: the controller processes a message received from a switch by running its input function



Controller Output

$f_{out}(\sigma) = (\sigma', sw, m)$

$\mathbb{M}(sw, msgs) \mid$
$\mathbb{C}(\sigma, f_{in}, f_{out})$

→

$\mathbb{M}(sw, m::msgs) \mid$
$\mathbb{C}(\sigma', f_{in}, f_{out})$

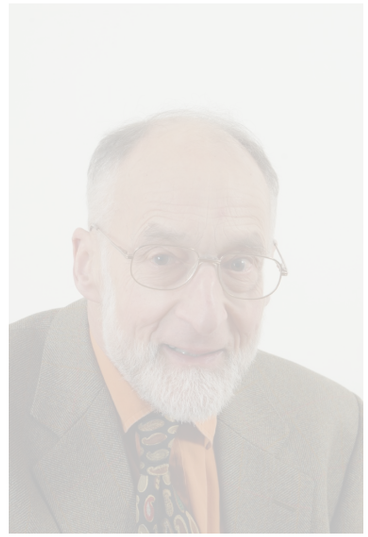
Intuition: the controller may spontaneously execute its output function

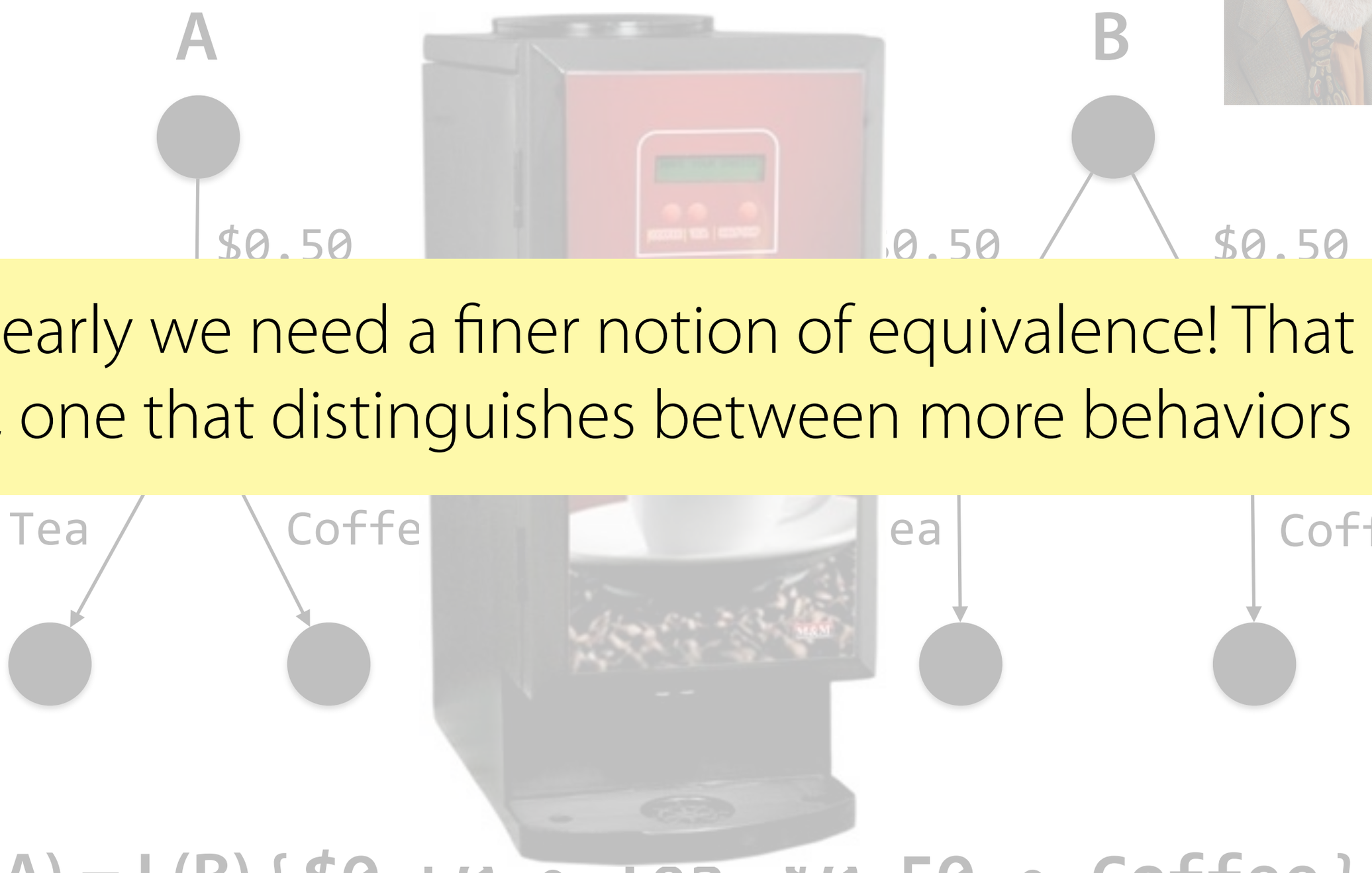
Equivalence

Given a specification of a packet-processing function, how can we establish that an OpenFlow implementation is correct?

It turns out that simple trace equivalence won't do...

Which Equivalence?



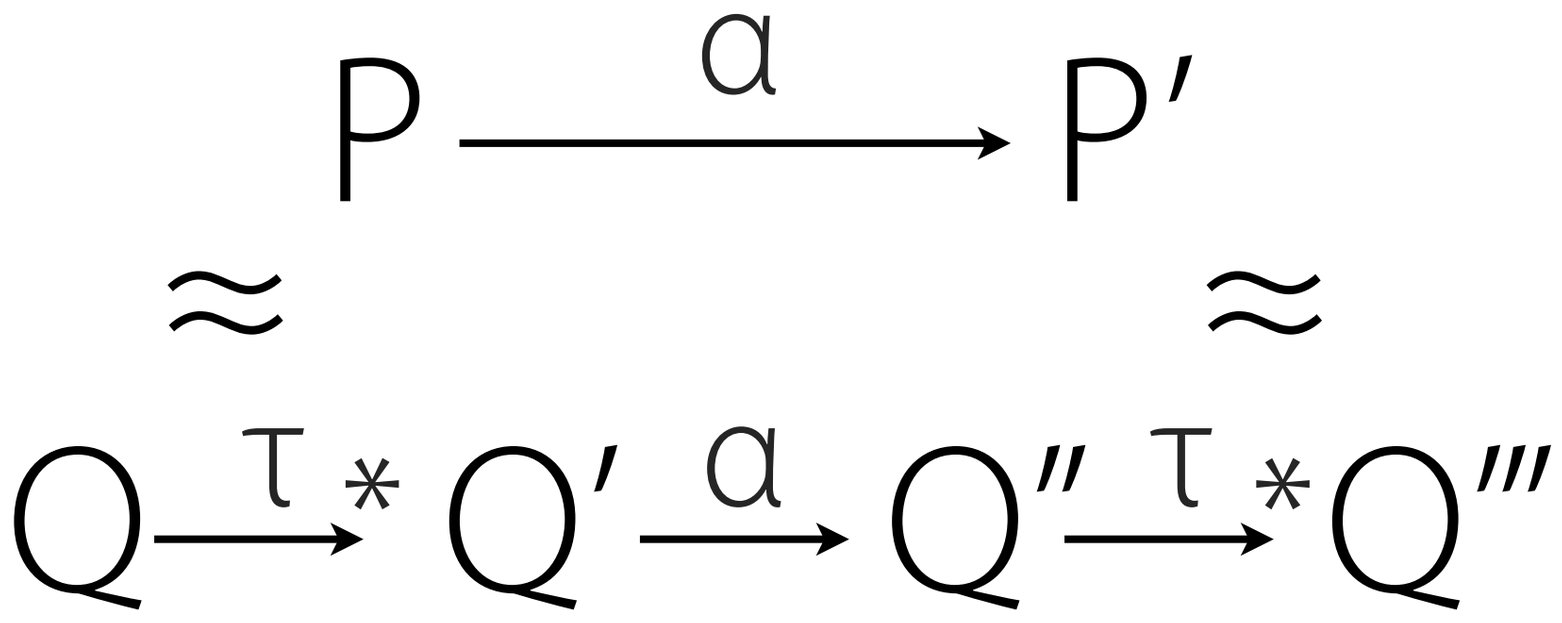


Clearly we need a finer notion of equivalence! That is, one that distinguishes between more behaviors

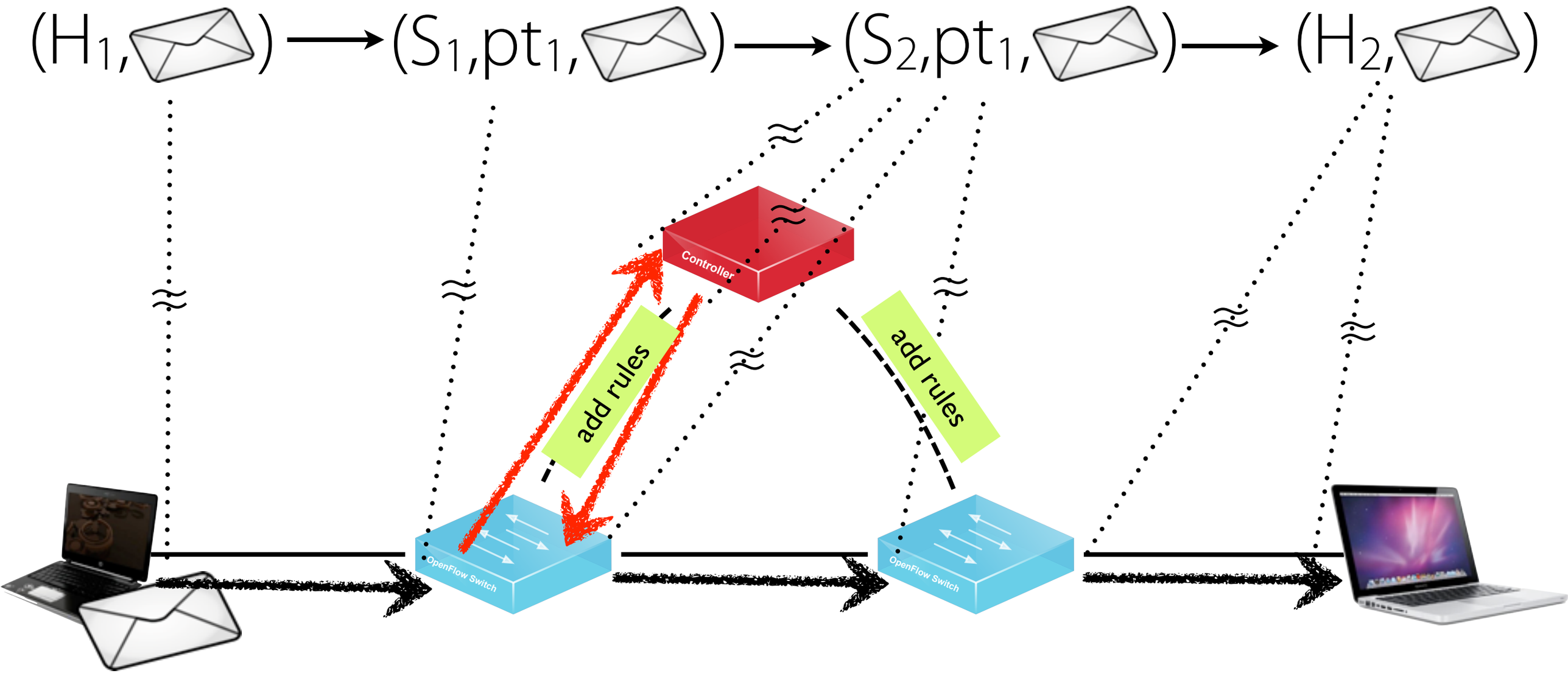
L(A) = L(B) { $0.50 • Tea, $0.50 • Coffee }

Weak Bisimulation

- **α:** observable actions such as the receipt of a packet at a given switch and port
- **τ:** "internal" actions such as a switch moving a packet into a buffer



Weak Bisimulation



Parameterized Weak Bisimulation

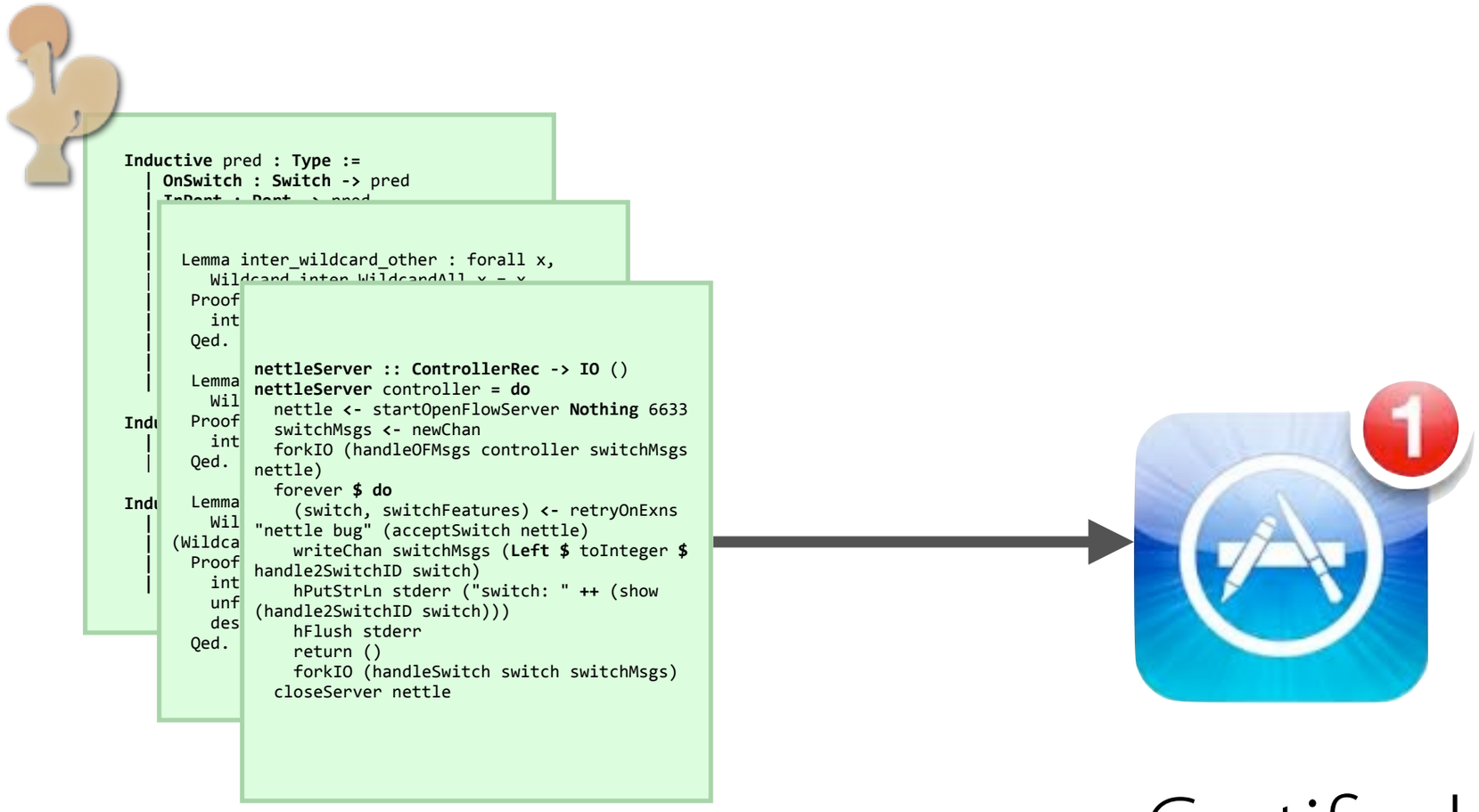
Invariants

- *Safety:* at all times, the rules installed on switches are a *subset* of the controller function
- *Liveness:* the controller eventually processes all packets diverted to it by switches

Theorem

```
Module RelationDefinitions :=
  FwOF.FwOFRelationDefinitions.Make (AtomsAndController).
  ...
Theorem fwof_abst_weak_bisim :
  weak_bisimulation
  concreteStep
  abstractStep
  bisim_relation.
```

Coq Implementation



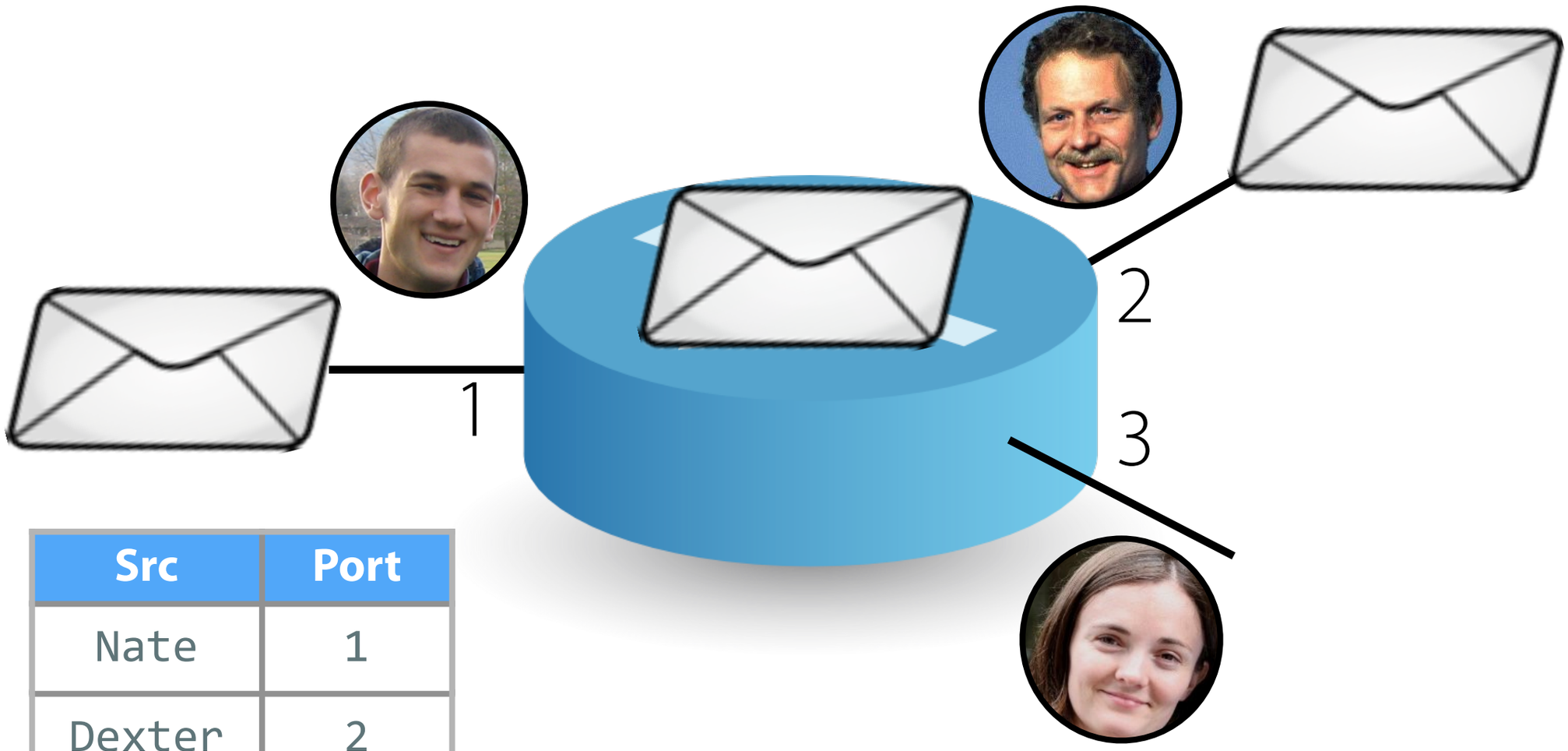
Develop, prove, extract

Certified executable

`github.com/frenetic-lang/featherweight-openflow`

NetKAT Language

Learning Switch



Src	Port
Nate	1
Dexter	2

- Learn locations of hosts in a table
- Flood packets going to unknown hosts; forward packets directly to known hosts

Learning Switch

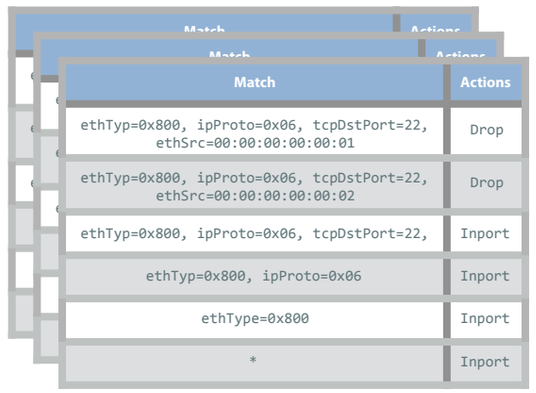
```
let known_hosts : (dlAddr,portId) Hashtbl.t = Hashtbl.create 101

let learning_packet_in (sw:switchId) (pktIn:packetIn) : unit =
  Hashtbl.add known_hosts pk.dlSrc pktIn.port

let routing_packet_in (sw:switchId) (pktIn : packetIn) : unit =
  try
    let out_port = Hashtbl.find known_hosts pk.dlDst in
    let sd = {match_all with dlDst=Some pk.dlDst; dlSrc=Some pk.dlSrc} in
    let ds = {match_all with dlDst=Some pk.DlSrc; dlSrc=Some pk.dlDst} in
    send_flow_mod sw 0l (add_flow 0 sd [Output out_port]);
    send_flow_mod sw 0l (add_flow 0 ds [Output pktIn.port]);
    send_packet_out sw 0l
      { output_payload = pktIn.input_payload;
        port_id = None;
        apply_actions = [Output out_port] }
  with Not found ->
    send_packet_out sw 0l
    { output_payload = pktIn.input_payload;
      port_id = None;
      apply_actions = [Flood] }

let packet_in (sw:switchId) _ (pk:packetIn) : unit =
  learning_packet_in sw pk; routing_packet_in sw pk
```

PyNetKAT Learning Switch

```
table = {}

def learn(app, switch, payload, port):
    pkt = Packet.from_payload(switch, port, payload)
    mac = pkt.ethSrc
    table[switch_id][mac] = port

def mac((fwd,unk),mac):
    fwd = fwd | Filter(EthDstEq(mac)) >> SetPort(table[sw][mac])
    unk = unk & ~EthSrcEq(mac)
    return (fwd, unk)

def switch_policy(sw):
   (fwd, unk) = reduce(mac, table[sw].keys(), (drop, true))
   unk = Filter(unk) >> (SendToController("learning") | flood(sw))
   return (fwd | unk)

class LearningApp(frenetic.App):
   ...
   def packet_in(self,switch_id, port_id, payload):
       learn(self, switch_id,payload,port_id)
       self.update(Union(switch_policy(sw) for sw in table.keys()))
```

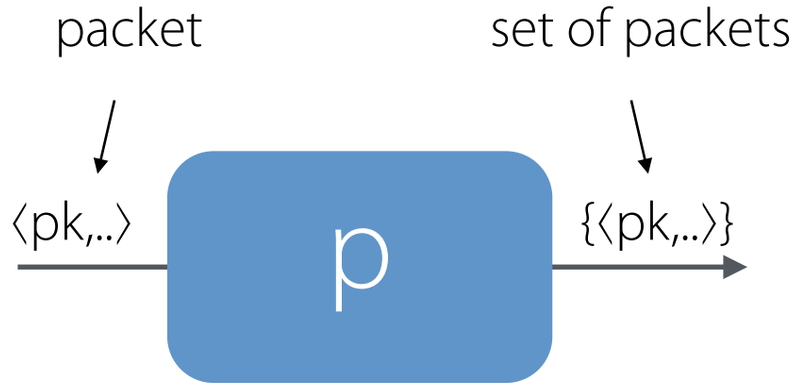
Machines

A *machine model* describes behavior in terms of concepts like pipelines of hardware lookup tables



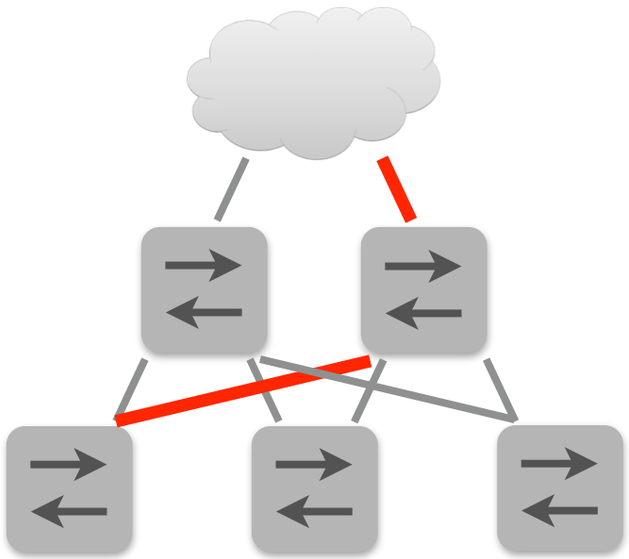
Languages

A *programming model* describes behavior in terms of concepts like mathematical functions on packets

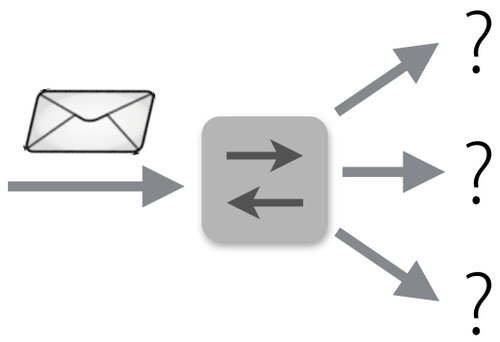


Language Design

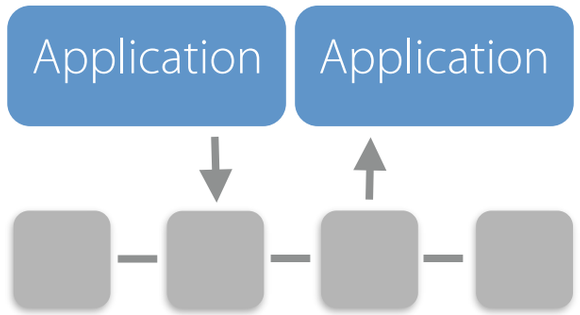
Desiderata



High-level, Network-wide Abstractions

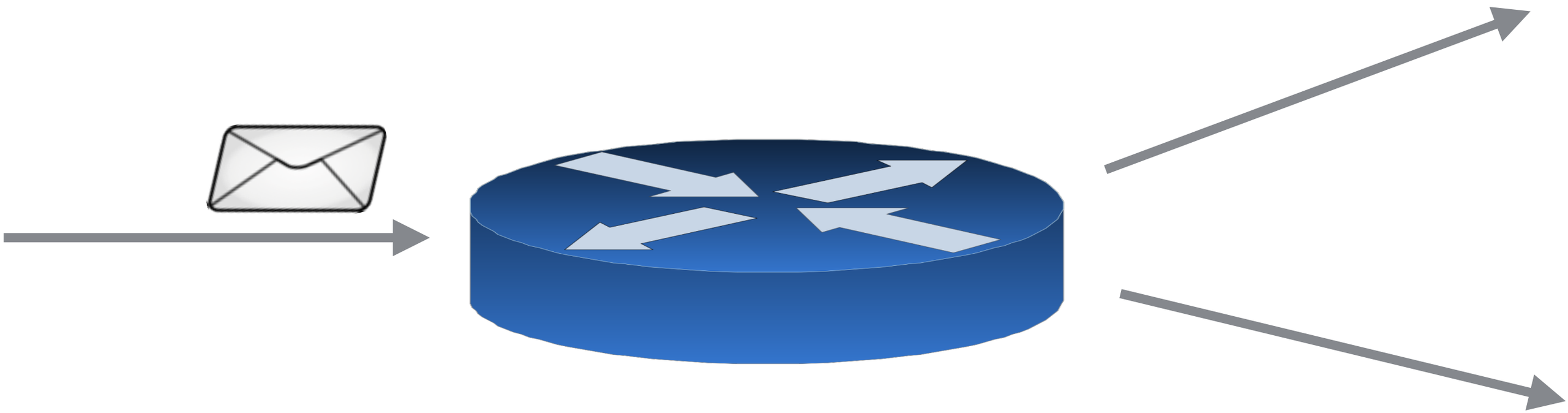


Rich Packet Classification



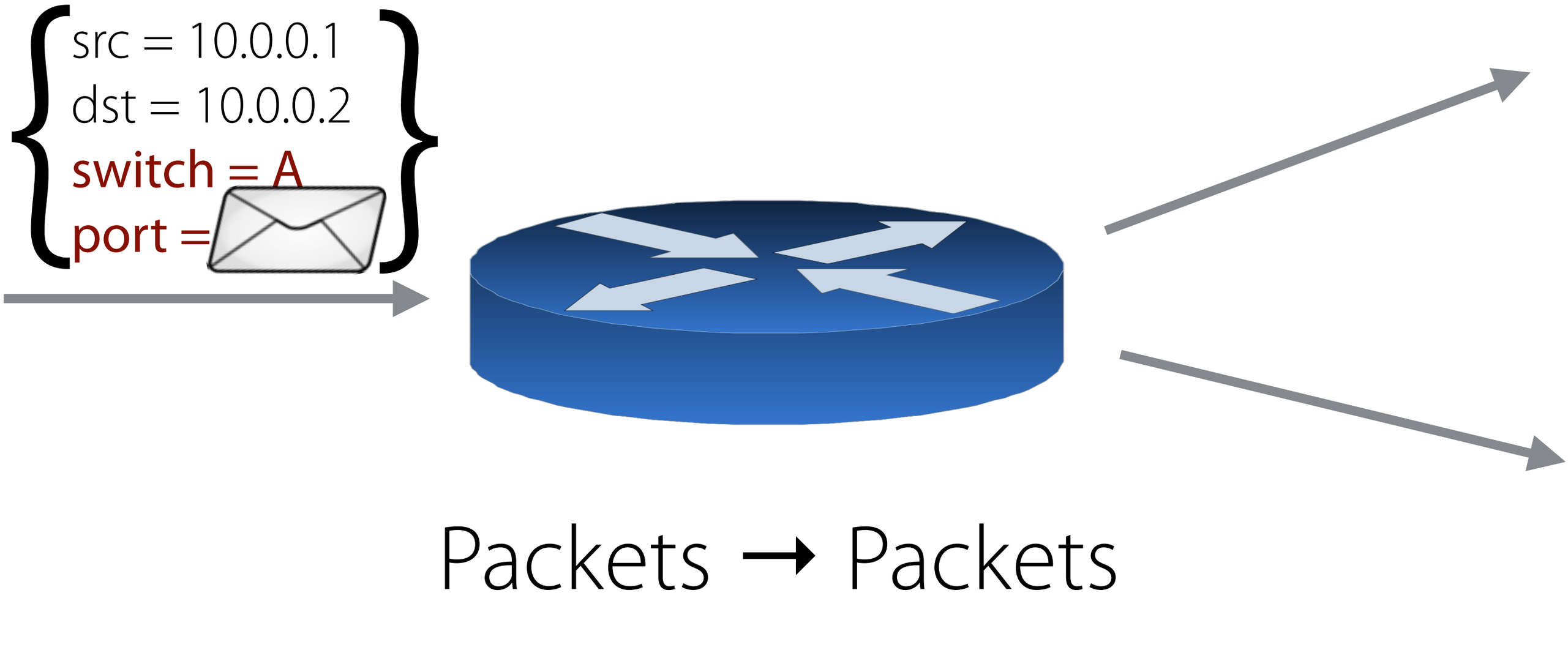
Modular Composition

Model



Packets → Packets

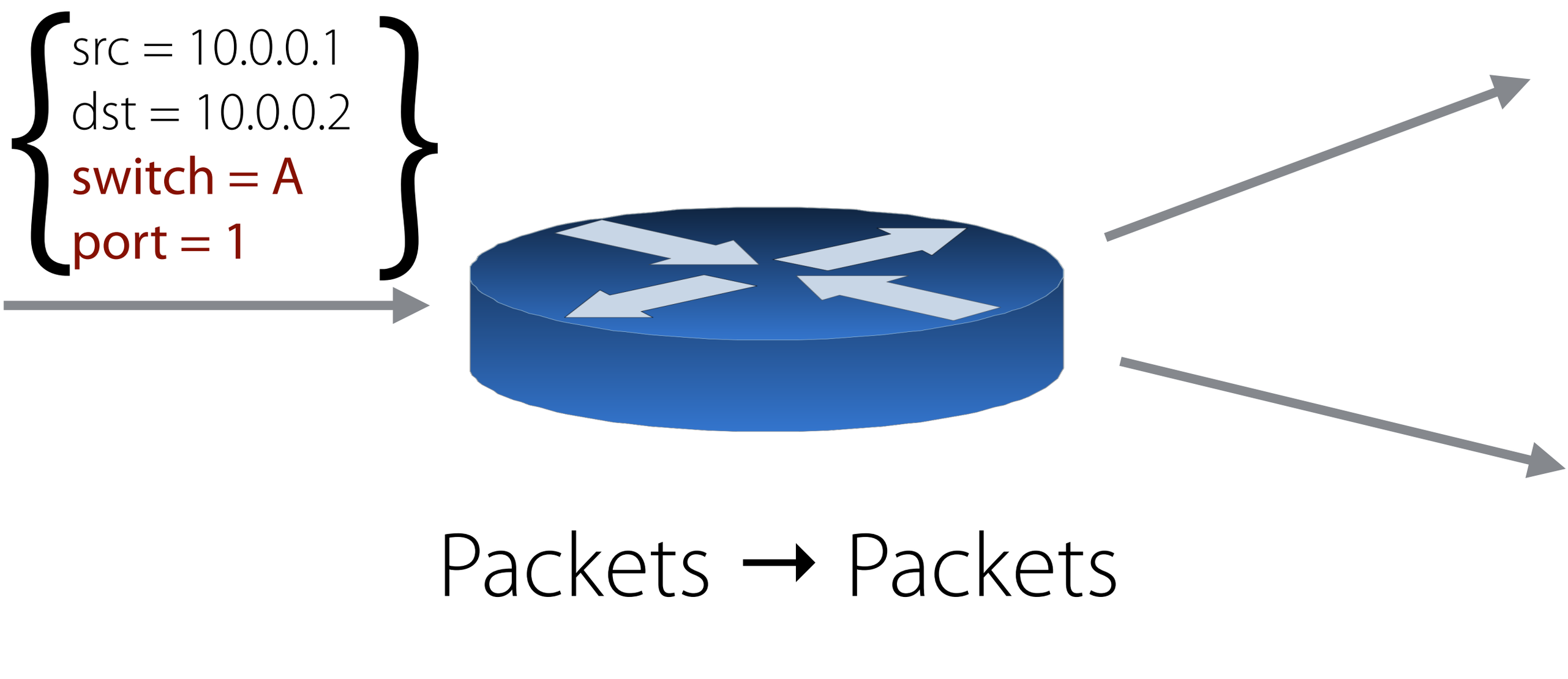
Model



Packets → Packets

Packet Manipulation

- packet filtering
- packet modification



Packets → Packets

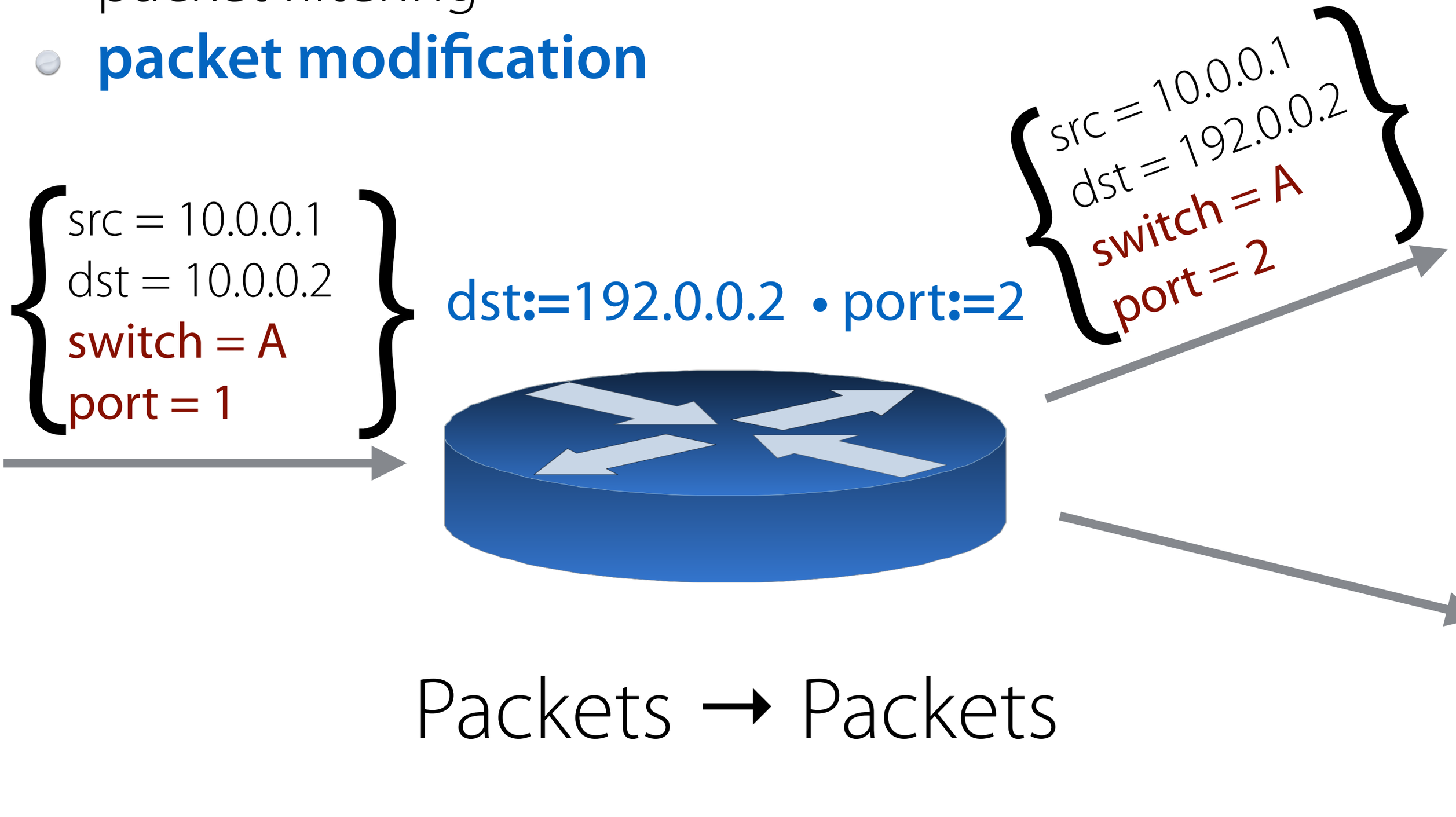
Packet Manipulation

- **packet filtering**
- packet modification

Packets → Packets

Packet Manipulation

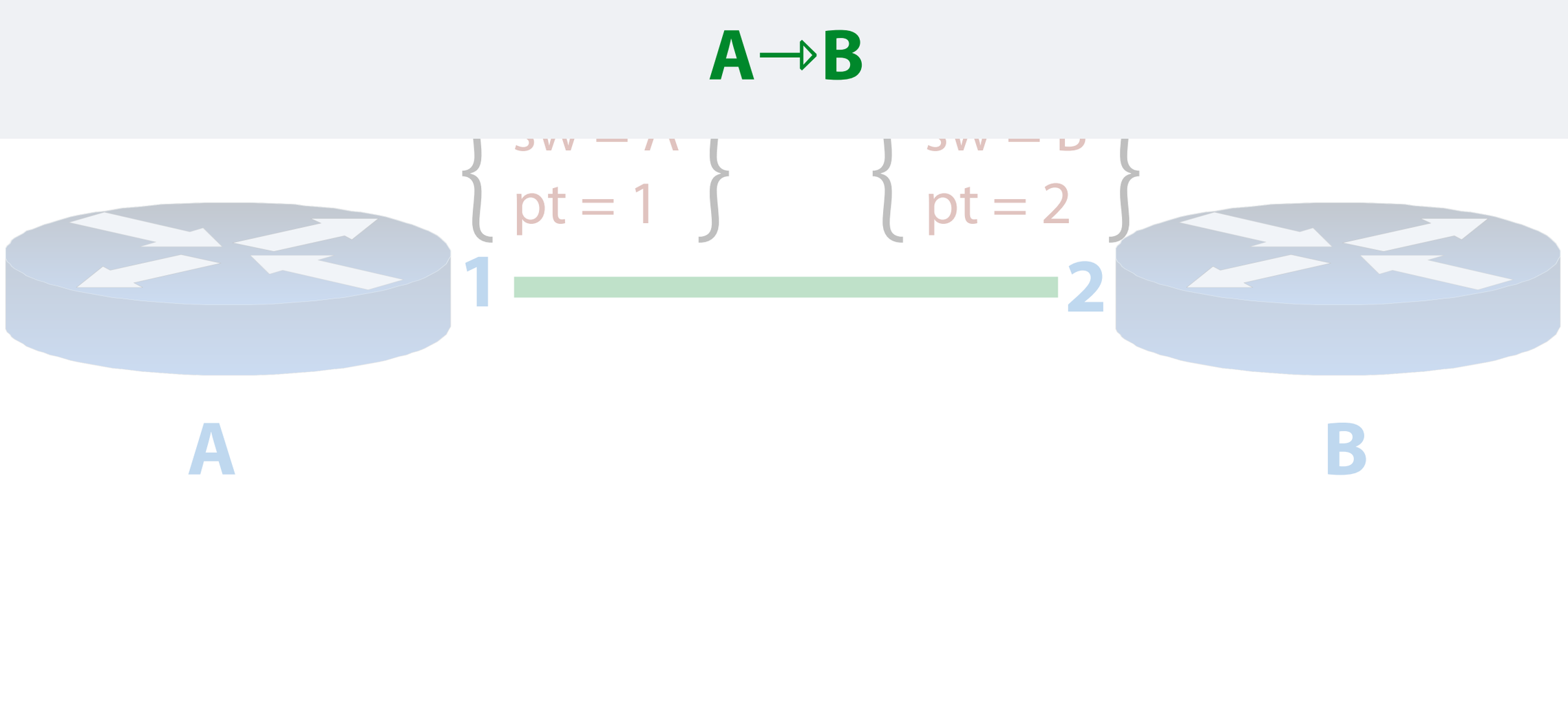
- packet filtering
- **packet modification**



Packets → Packets

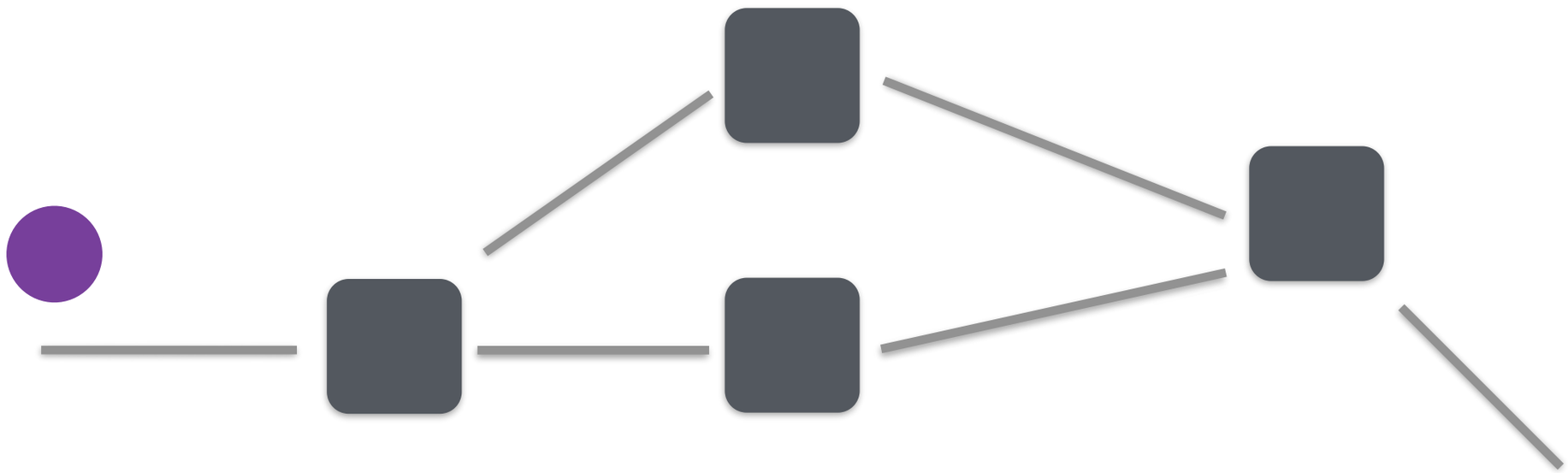
Topology

switch=A • pt=1 • switch:=B • pt:=2



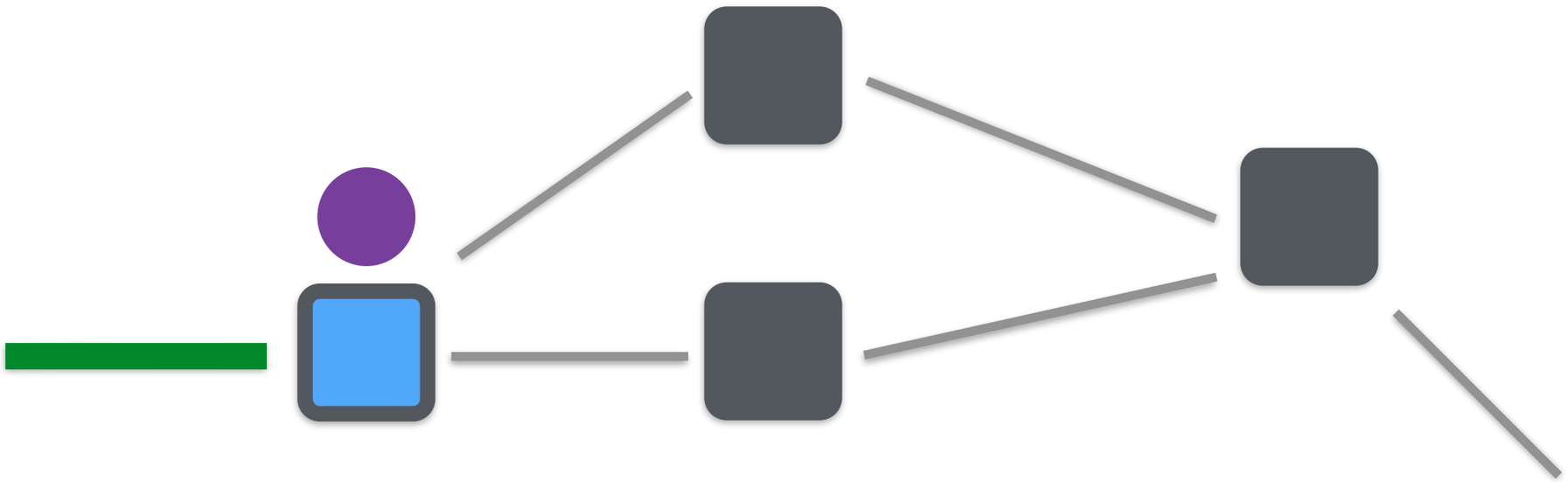
Iteration

(topology • switch)*



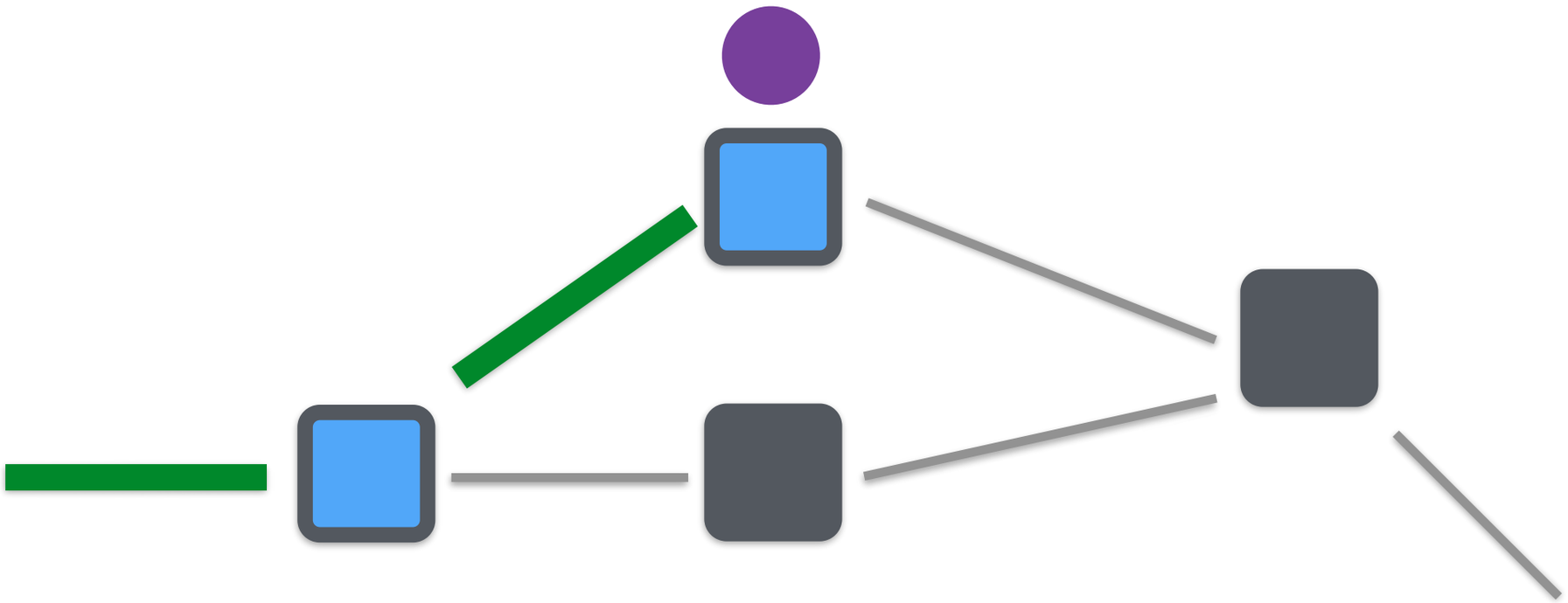
Iteration

(topology • switch)*



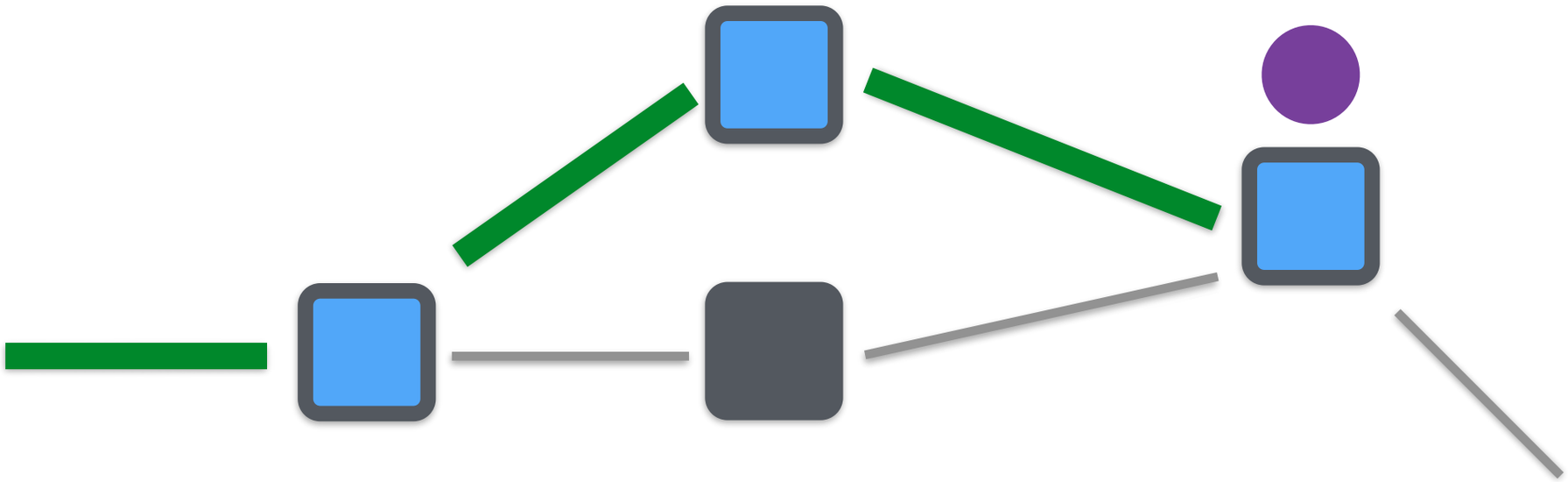
Iteration

(topology • switch)*



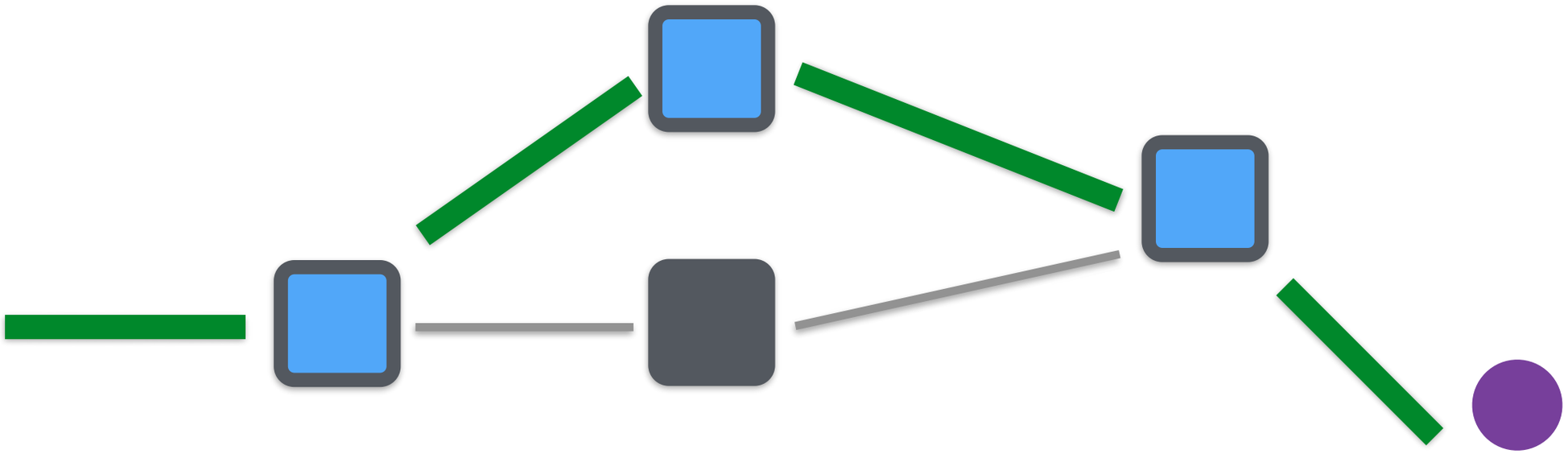
Iteration

(topology • switch)*



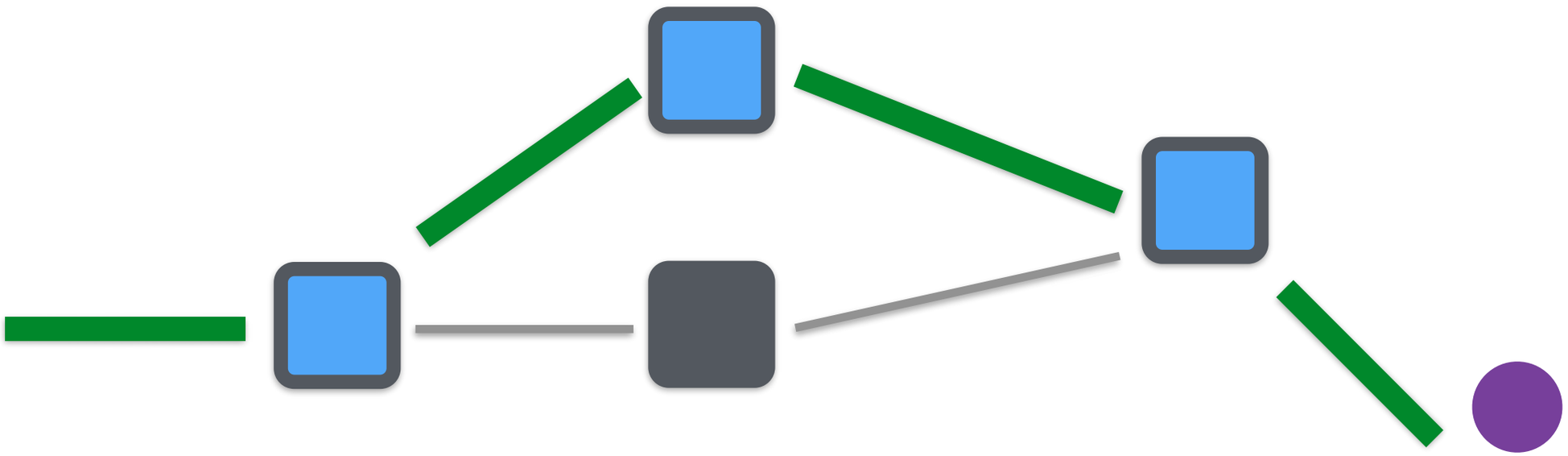
Iteration

(topology • switch)*



Iteration

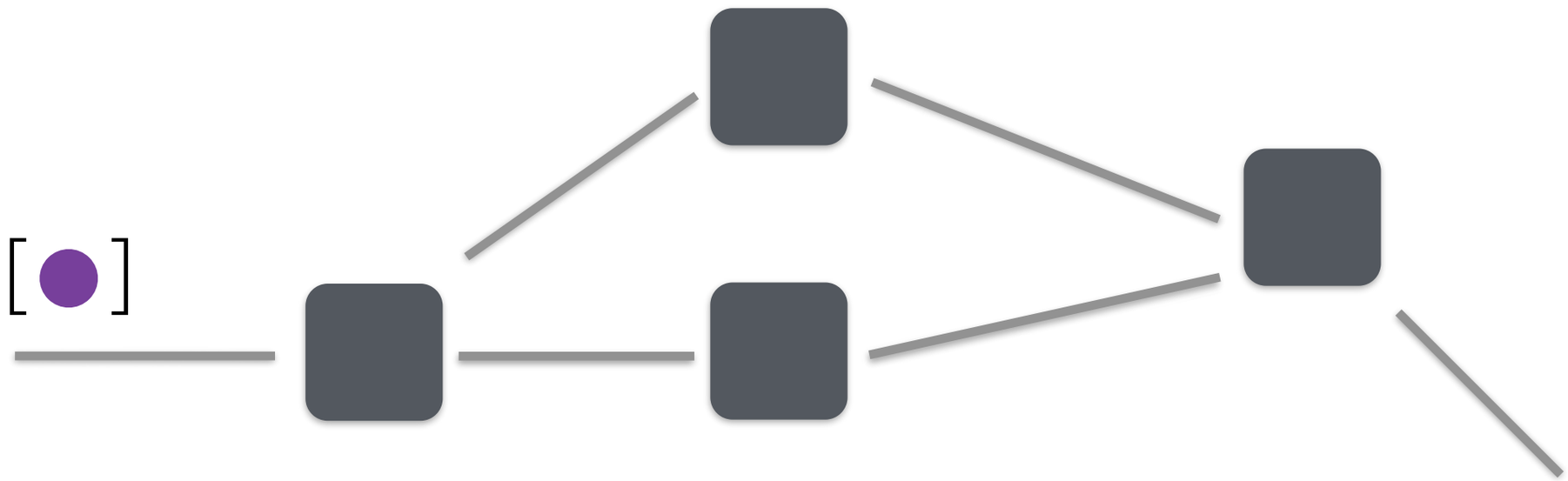
(topology • switch)*



~~Packets → Packets~~

Iteration

(topology • switch)*

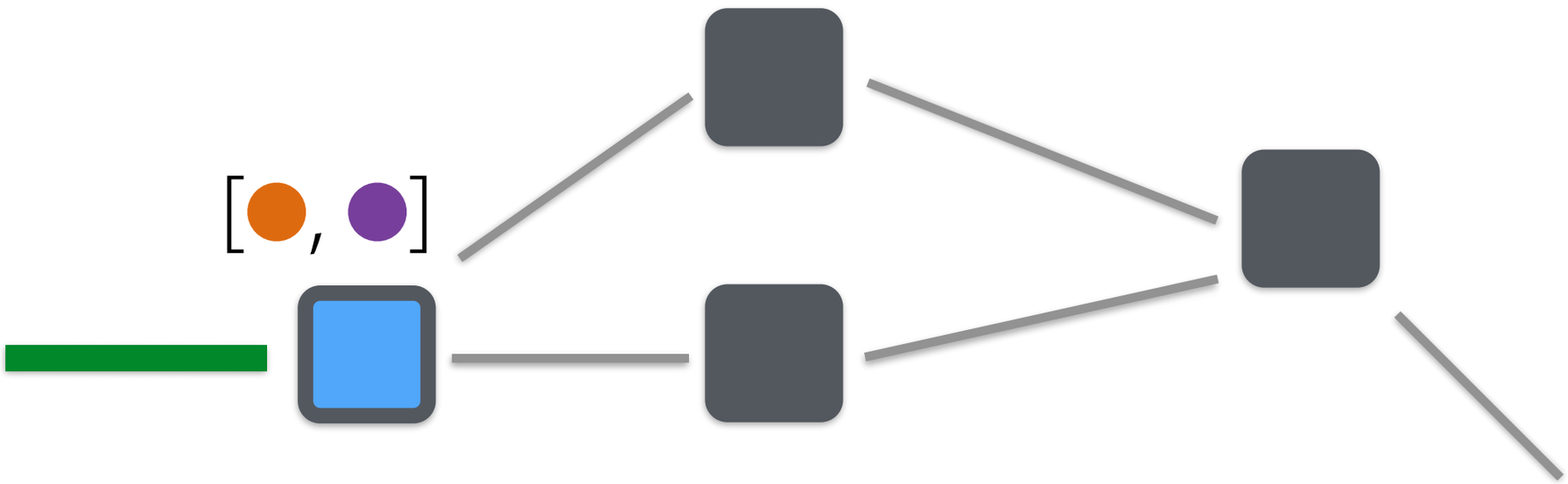


~~Packets → Packets~~

Histories → Histories

Iteration

(topology • switch)*

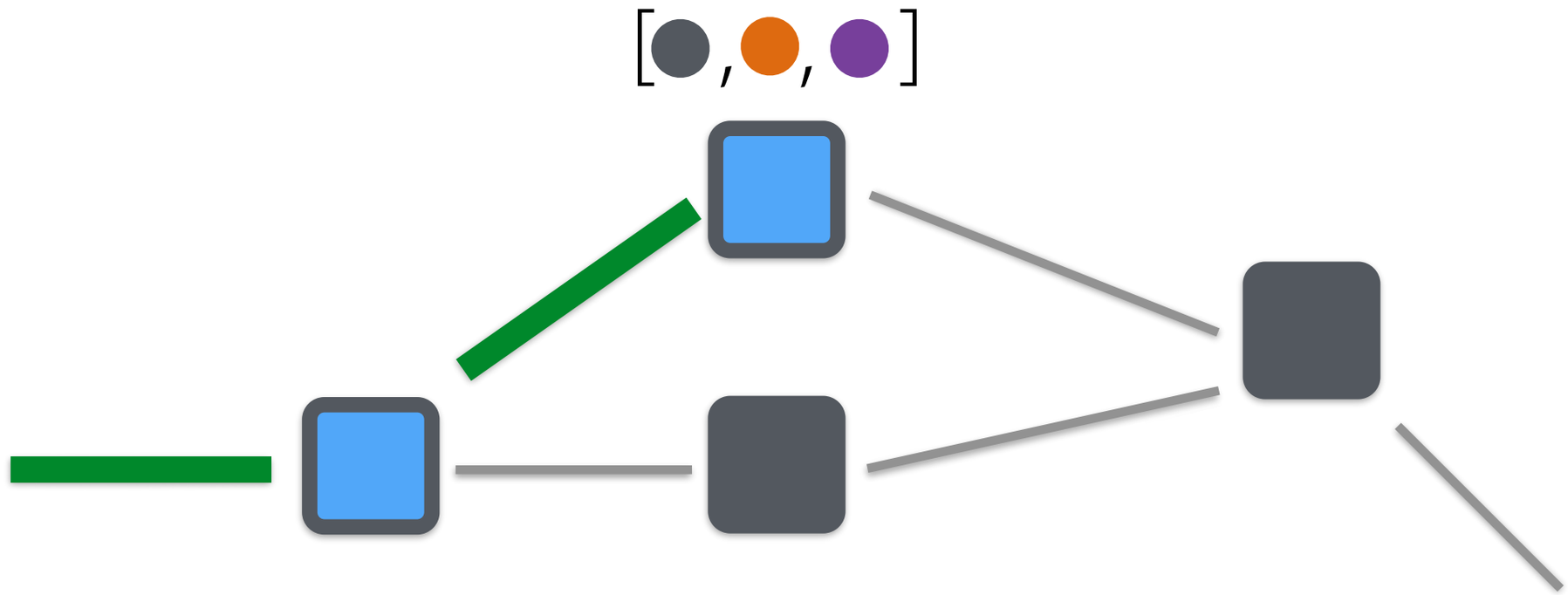


~~Packets → Packets~~

Histories → Histories

Iteration

(topology • switch)*

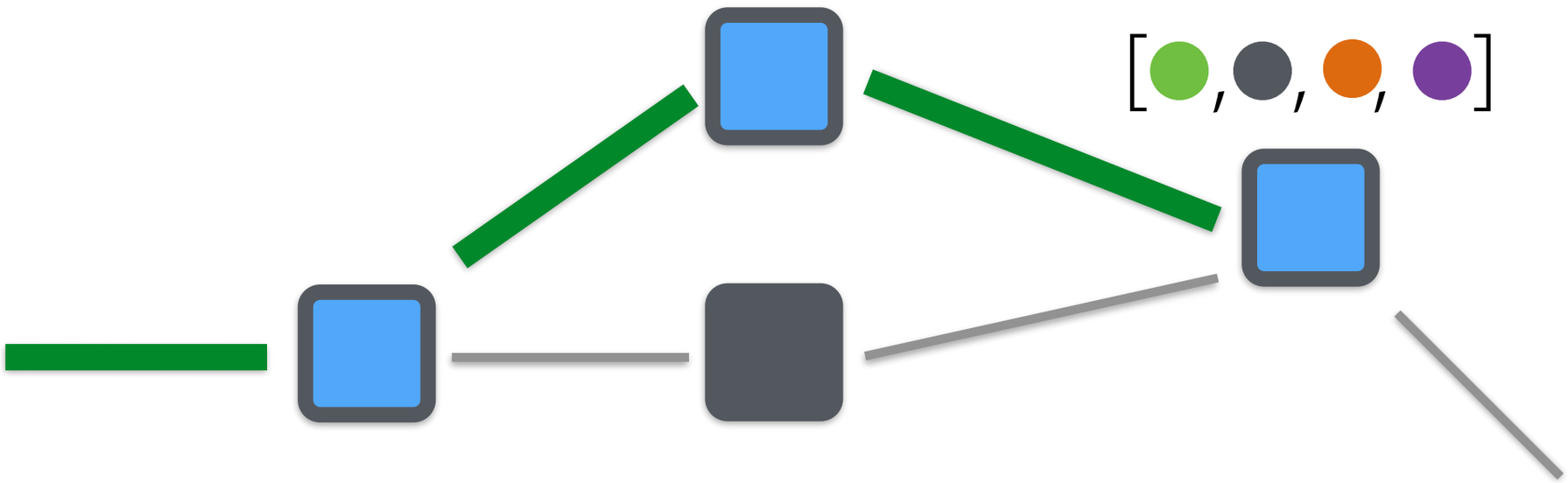


~~Packets → Packets~~

Histories → Histories

Iteration

(topology • switch)*

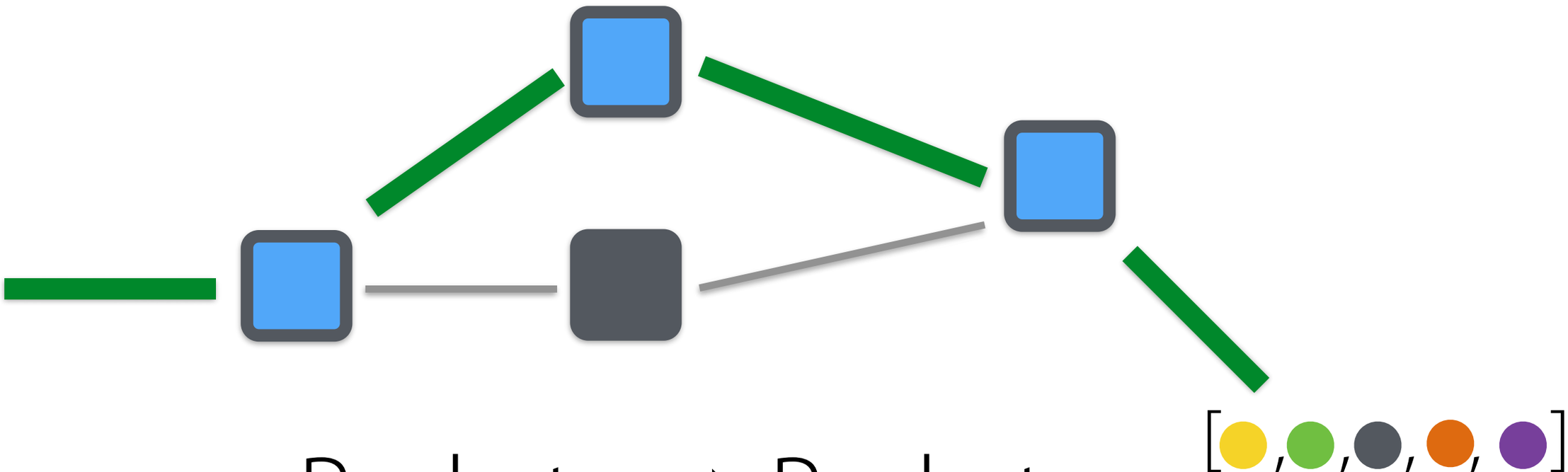


~~Packets → Packets~~

Histories → Histories

Iteration

(topology • switch)*

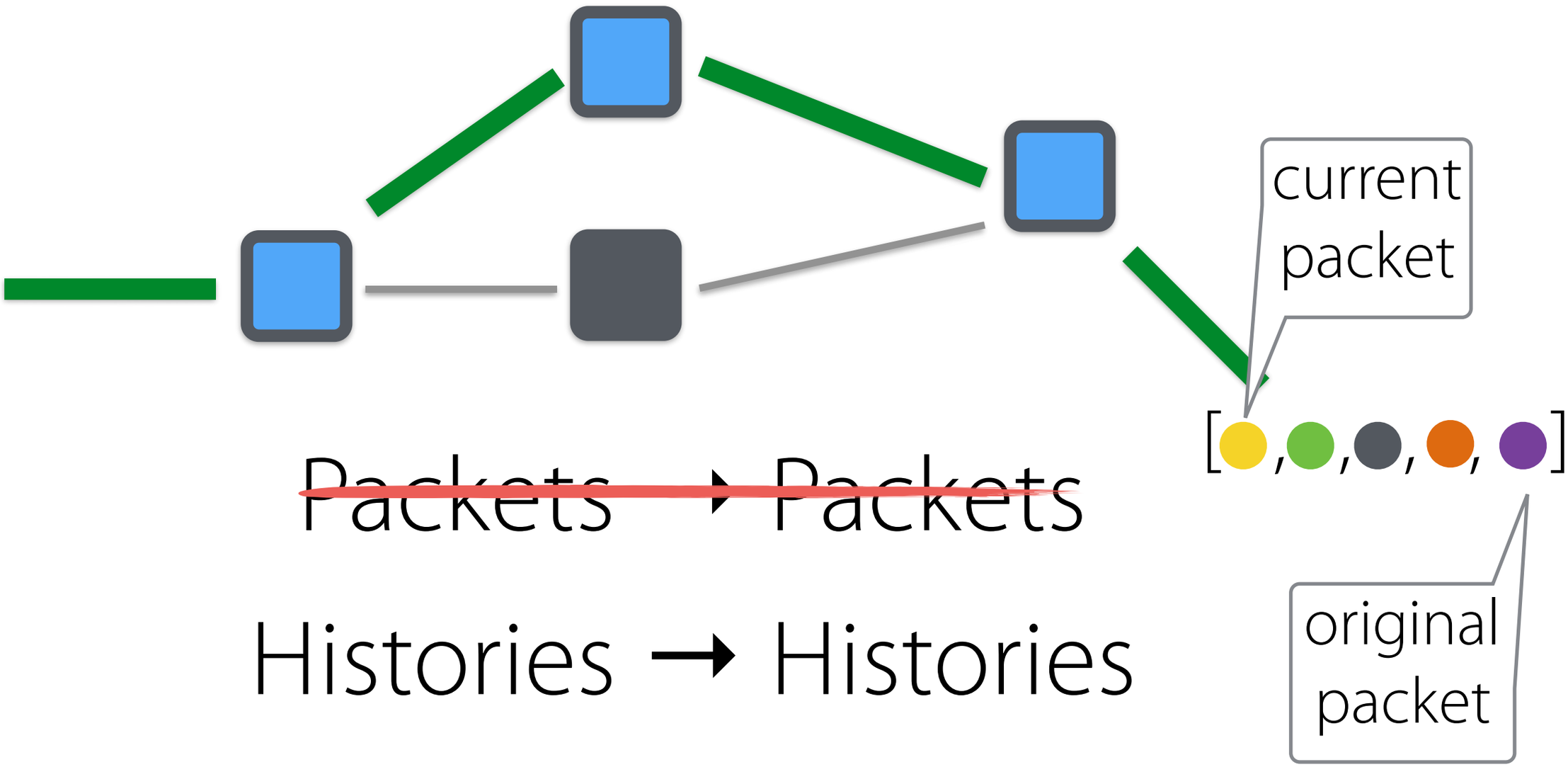


~~Packets → Packets~~

Histories → Histories

Iteration

(topology • switch)*



~~Packets → Packets~~

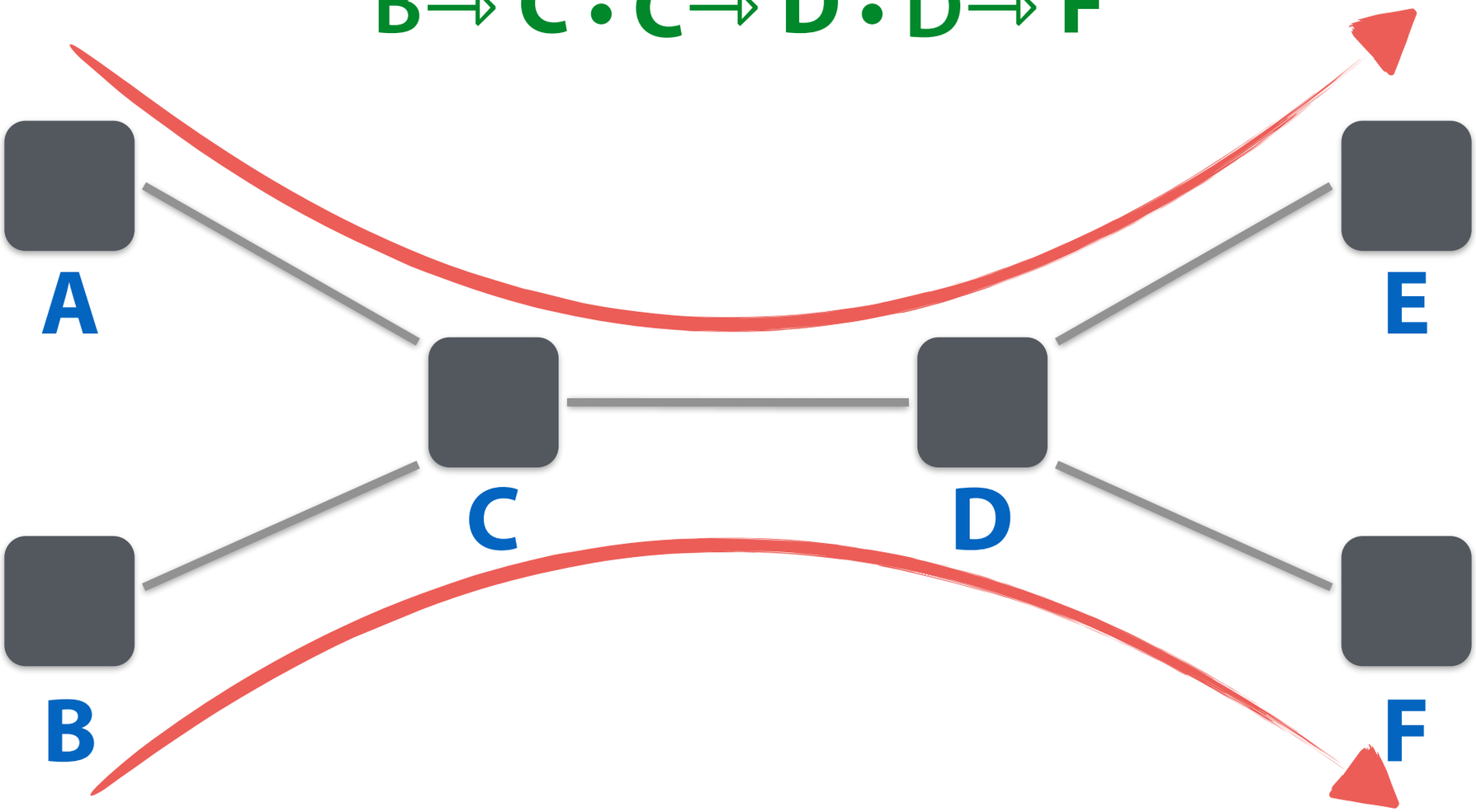
Histories → Histories

Regular Paths

$$A \rightarrow C \bullet C \rightarrow D \bullet E \rightarrow E$$

$$+$$

$$B \rightarrow C \bullet C \rightarrow D \bullet D \rightarrow F$$



NetKAT Syntax

pol ::= **false**
| **true**
| f **=** n
| pol **+** pol
| pol **•** pol
| pol*
| **!**pol
| **dup**

Boolean Predicates

Primitives

if b **then** p_1 **else** p_2 ≜ (b • p_1) **+** (**!**b • p_2)

while b **do** p ≜ (b • p)* • **!**b

S→S′ ≜ sw**=**S • **dup** • sw**:=**S′ • **dup**

Negation may only be applied to B*oolean predicates:*
true, **false**, f **=** n, closed under +, •, and **!**

NetKAT Semantics

NetKAT Semantics

Denotational

Soundness+Completeness
[POPL '14]

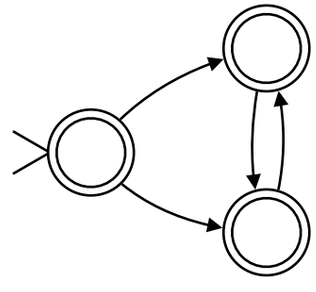
Kleene's Theorem
[POPL '15]

Axiomatic

$\vdash p \equiv q$

Proof-Carrying Code
[Ongoing Work]

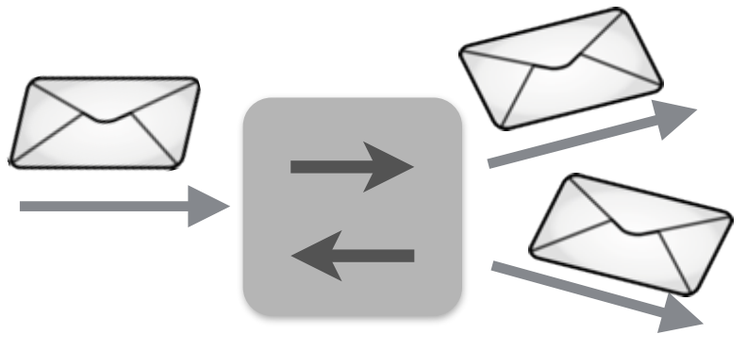
Operational



Denotational Semantics

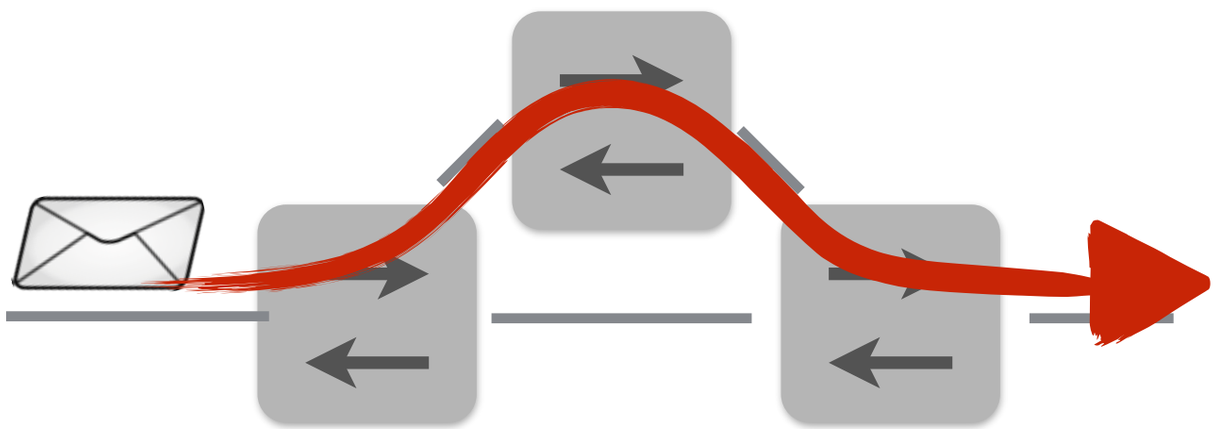
pol ::=
- | **false**
- | **true**
- | field **=** val
- | field **:=** val
- | pol_1 **+** pol_2
- | pol_1 • pol_2
- | !pol
- | pol*
- | ~~**dup**~~

Local: input-output behavior of switches



⟦pol⟧ ∈ Packets → Packets

Global: network-wide paths



⟦pol⟧ ∈ Histories → Histories

NetKAT Semantics

$[\![pol]\!] \in \textbf{History} \rightarrow \textbf{History Set}$

$[\![\textbf{true}]\!]\ h = \{ h \}$

$[\![\textbf{false}]\!]\ h = \{\}$

$$[\![f = n]\!]\ pk :: h = \begin{cases} \{ pk :: h \} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases}$$

$[\![\textbf{!}\ pol]\!]\ h = \{ h \} \setminus [\![pol]\!]$

$[\![f := n]\!]\ pk :: h = \{ pk[f:=n] :: h \}$

$[\![pol_1 + pol_2]\!]\ h = [\![pol_1]\!]\ h \cup [\![pol_2]\!]\ h$

$[\![pol_1 \bullet pol_2]\!]\ h = ([\![pol_1]\!] \bullet [\![pol_2]\!])\ h$

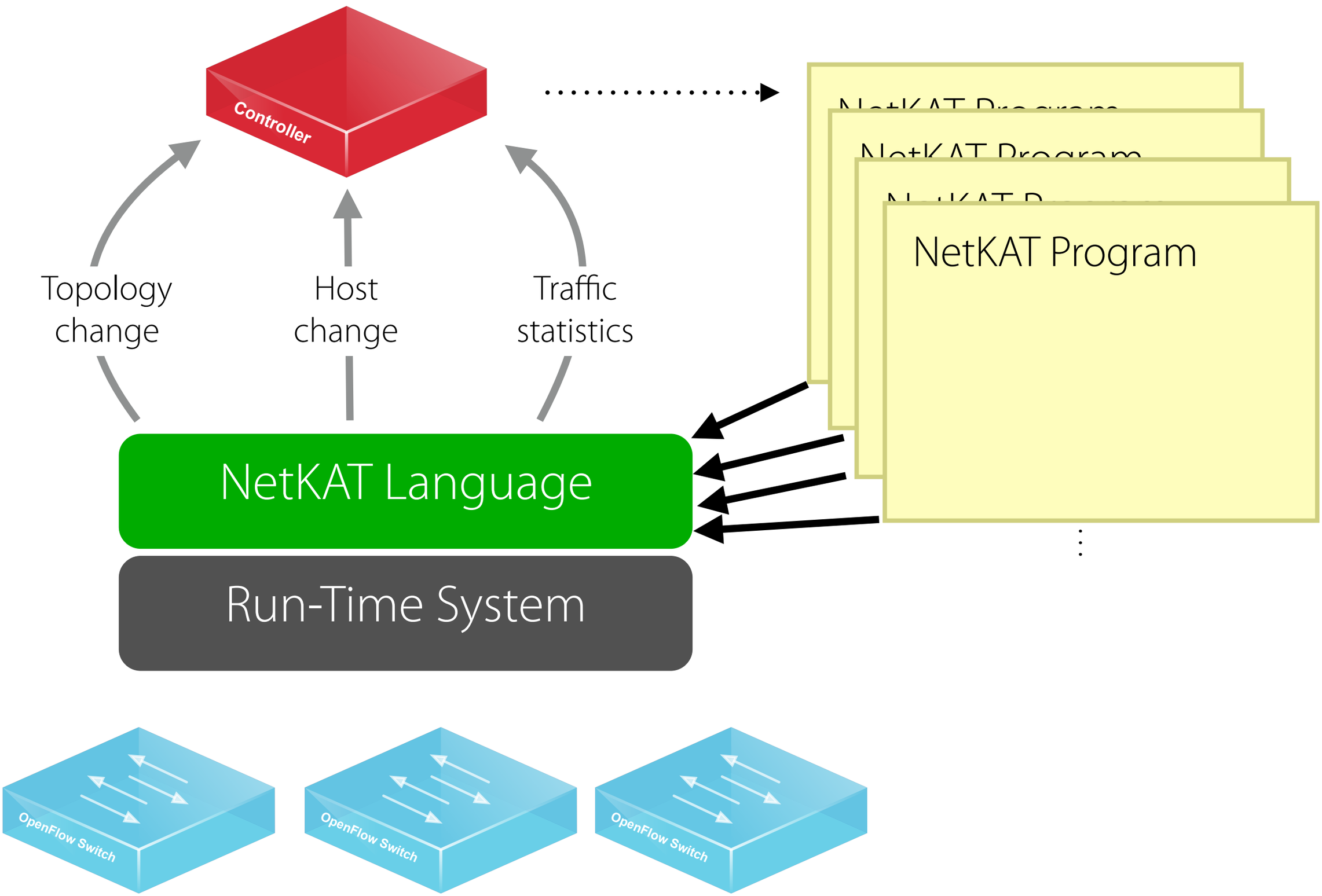
$[\![pol^*]\!]\ h = (\ \bigcup_i [\![pol]\!]^i\ h)$

$$[\![\ S \rightarrow S']\!]\ pk :: h = \begin{cases} \{ pk[sw:=S'] :: pk :: h \} & \text{if } pk.sw = S \\ \{\} & \text{otherwise} \end{cases}$$

$f,g \in \textbf{History} \rightarrow \textbf{History Set}$

$(f \bullet g)\ h = \bigcup \{ g\ h' \mid h' \in f\ h \}$

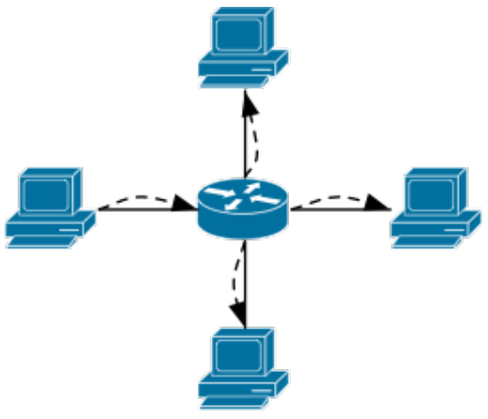
Stateful Applications



Modular Composition

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…

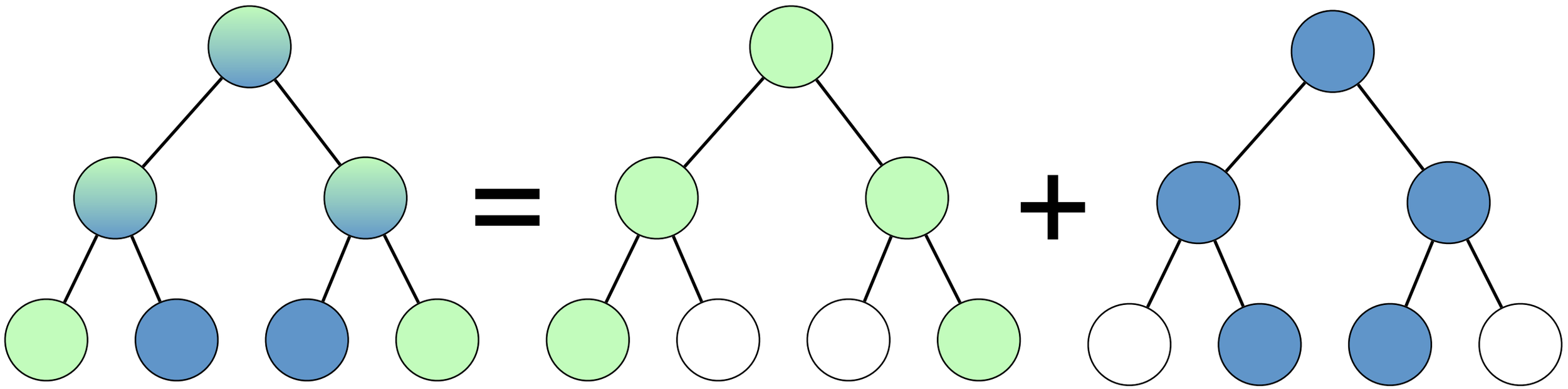
```
{pattern={ethSrc=00:00:00:00:00:01,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethSrc=00:00:00:00:00:02,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethDst=00:00:00:00:00:01},action=[Output(1)]}
{pattern={ethDst=00:00:00:00:00:02},action=[Output(2)]}
{pattern={ethDst=00:00:00:00:00:03},action=[Output(3)]}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

Applications

Isolated Slices [POPL '14]

In many situations, multiple tenants must share the network…

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



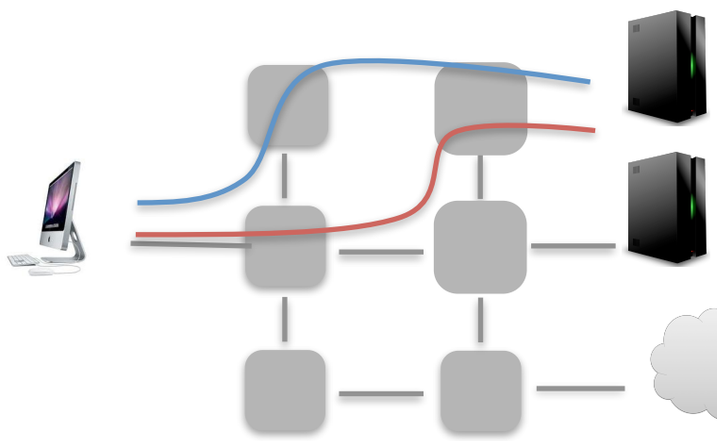
Ingress Tag Policy Egress

{ in **}** x **:** pol **{** out **}**

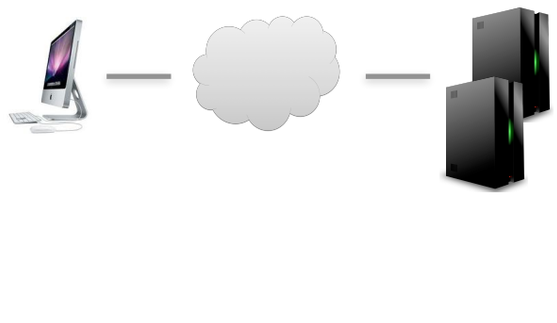
```
let pre = (tag = none; in; tag := x + tag = x)
in
let post = (out; tag := none + !out) in
```

Virtualization [ICFP '15]

Often useful to programs against a simplified network topology



Concrete

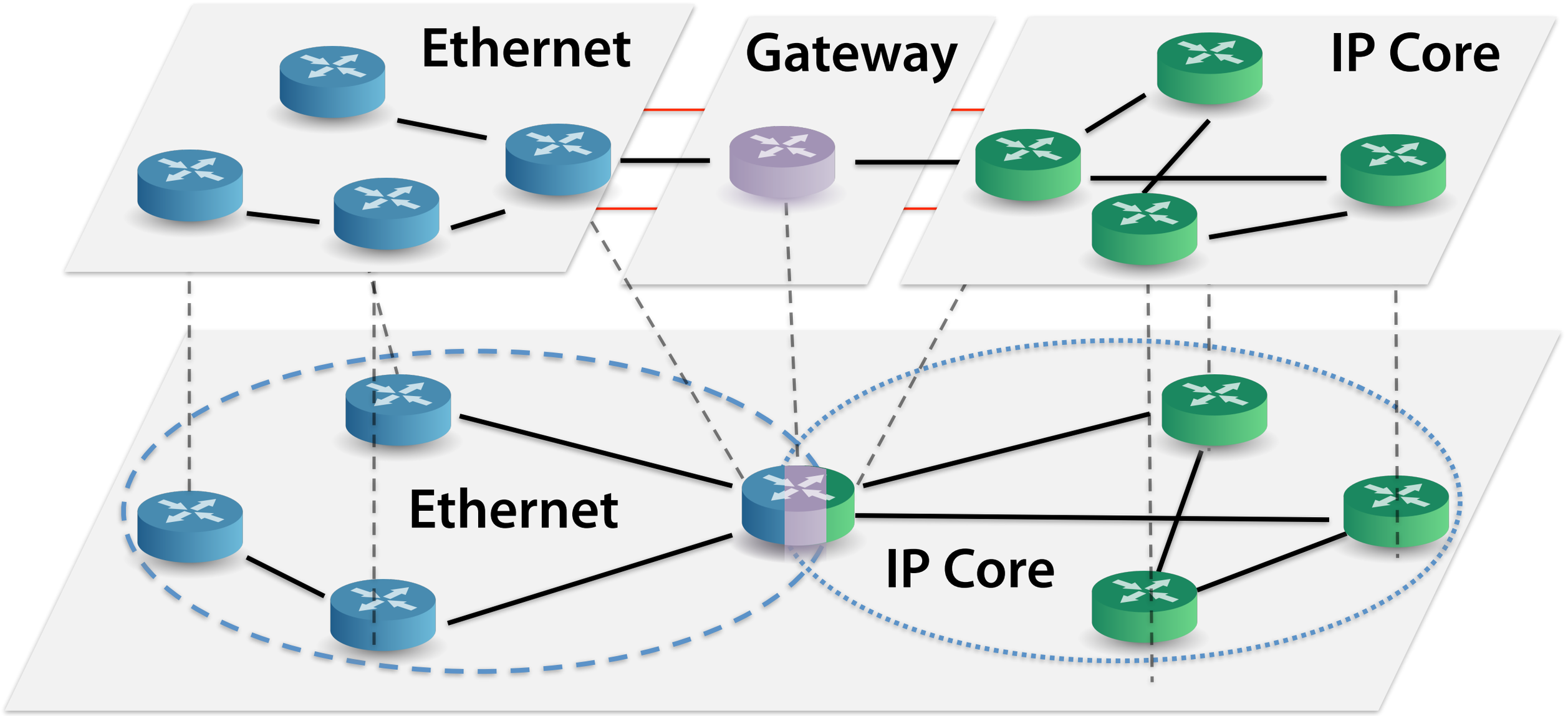


Abstract

Benefits:

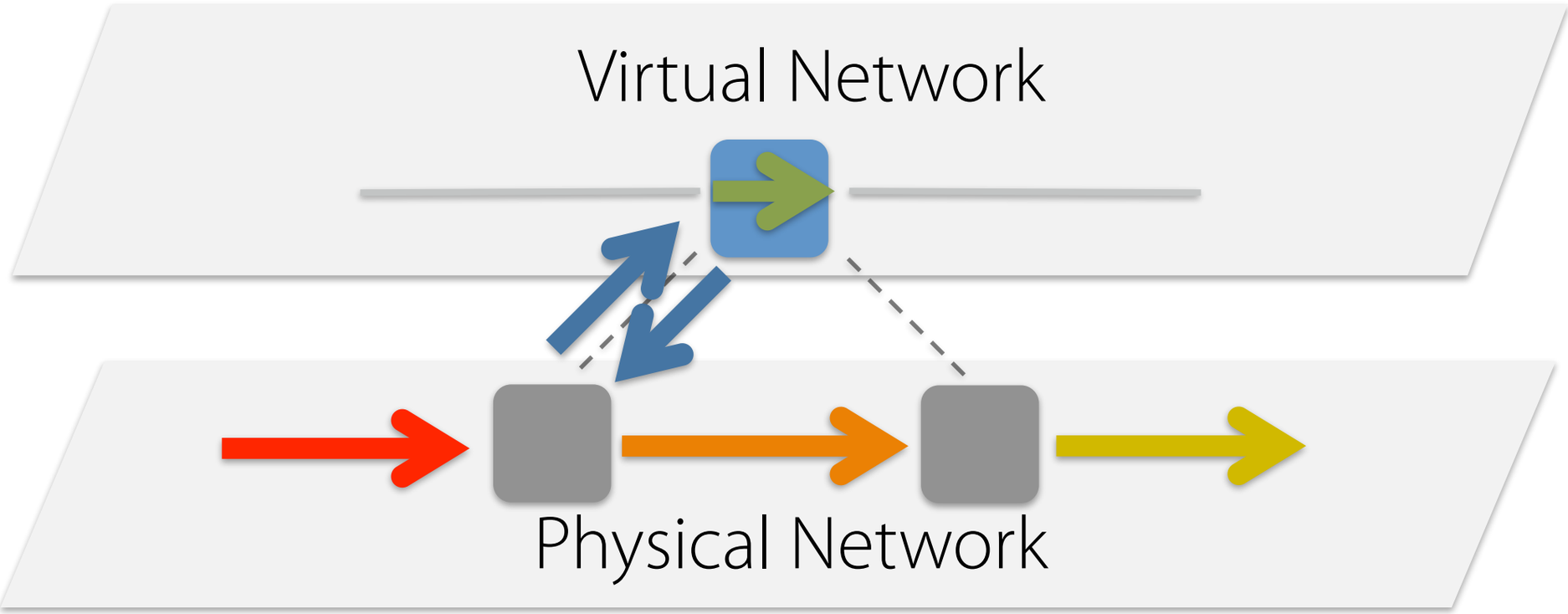
- *Information hiding:* limit what modules see
- *Protection:* limit what modules can do
- *Code reuse:* limit what dependencies modules have

Example: Gateway



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

Implementing Virtualization



This idiom can be implemented in NetKAT!

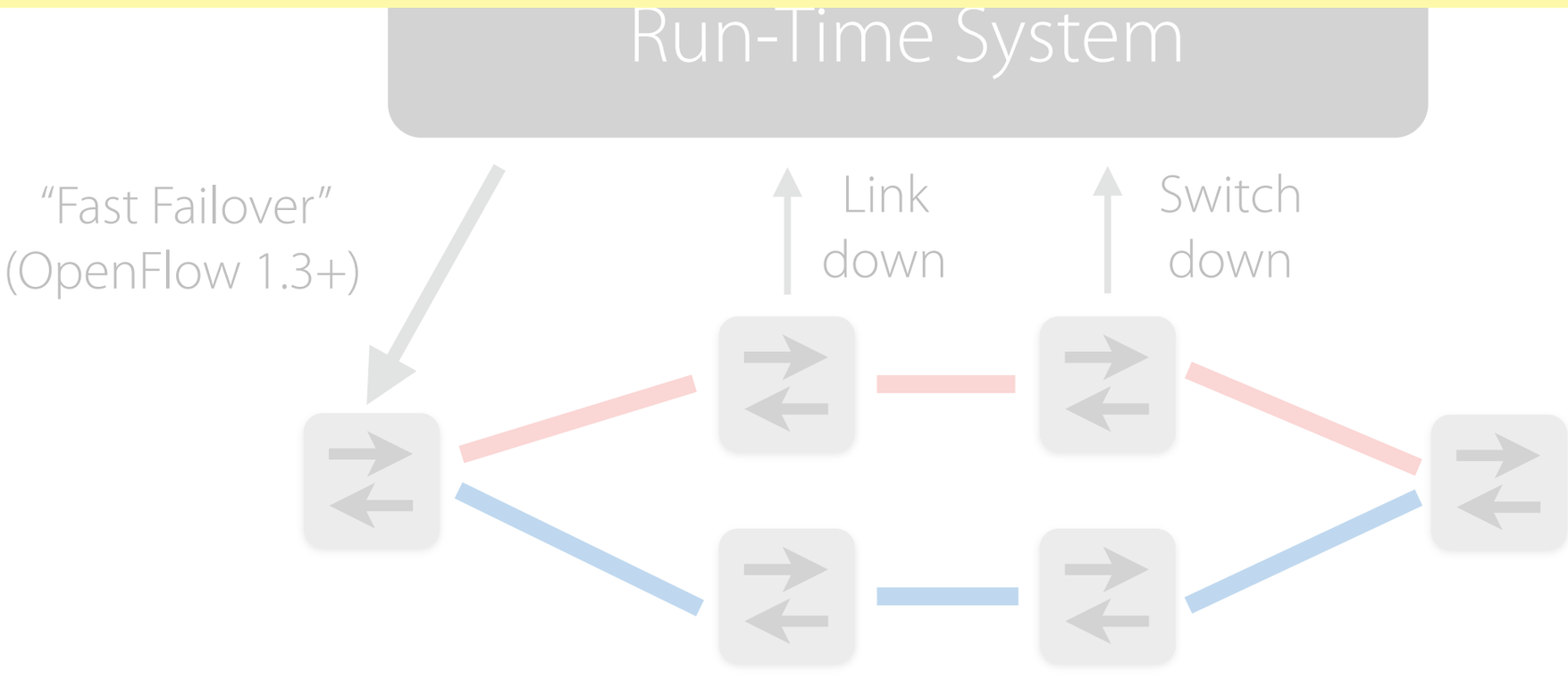
```
ingress;
(raise; application; lower; fabric)*;
egress
```

Fault Tolerance [HotSDN '13]

Path abstraction
hides low-level faults

Red path or Blue path

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



Wrapping Up...

Conclusion

- Programming languages have key role to play in the design and 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
  - Fault Tolerance
  - and many others...

Reading

- Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. **NetKAT: Semantic Foundations for Networks**. In *ACM SIGPLAN--SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2014.
- Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. **A Fast Compiler for NetKAT.** *In ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2015.
- Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. **FatTire: Declarative Fault Tolerance for Software-Defined Networks**. In ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), August 2013.

Acknowledgments

- Carolyn Anderson (UMass)
- Shrutarshi Basu (Cornell)
- Marco Canini (KAUST)
- Arjun Guha (UMass)
- Jean-Baptiste Jeannin (Michigan)
- Dexter Kozen (Cornell)
- Robert Kleinberg (Cornell)
- Mark Reitblatt (Facebook)
- Cole Schlesinger (Amazon)
- Emin Gün Sirer (Cornell)
- Robert Soulé (Lugano)
- Dave Walker (Princeton)

Later today...

I: Introduction

- Semantics Primer
- Software-Defined Networking

II: NetKAT

- Language Design
- Formal Semantics

III: Probabilistic NetKAT

- Formal Semantics
- Applications