Network Programming with frenetic

Nate Foster (Cornell)
Arjun Guha (UMass)
Victoria Yang (Cornell)

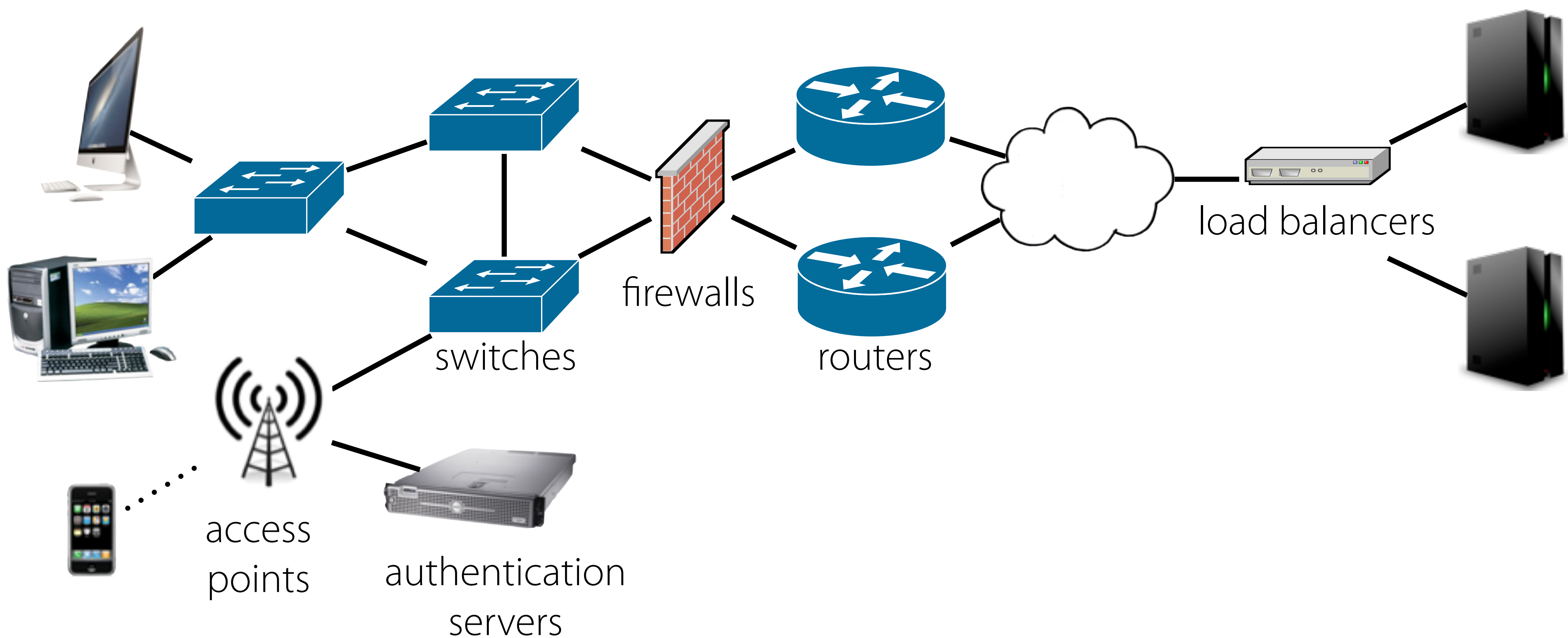
ECOOP Summer School

Networks Today

Reasoning about network behavior is *extremely* difficult…

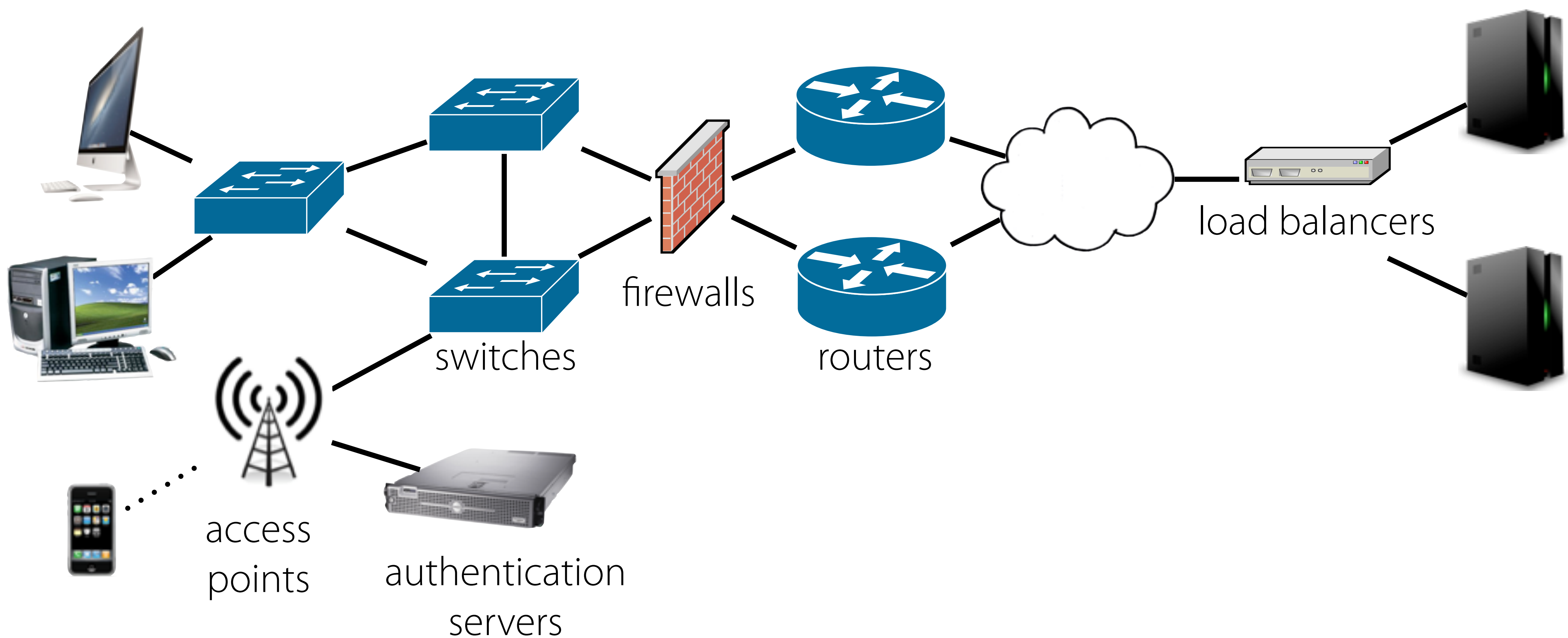
Networks Today

Reasoning about network behavior is *extremely* difficult…



Networks Today

Reasoning about network behavior is *extremely* difficult…

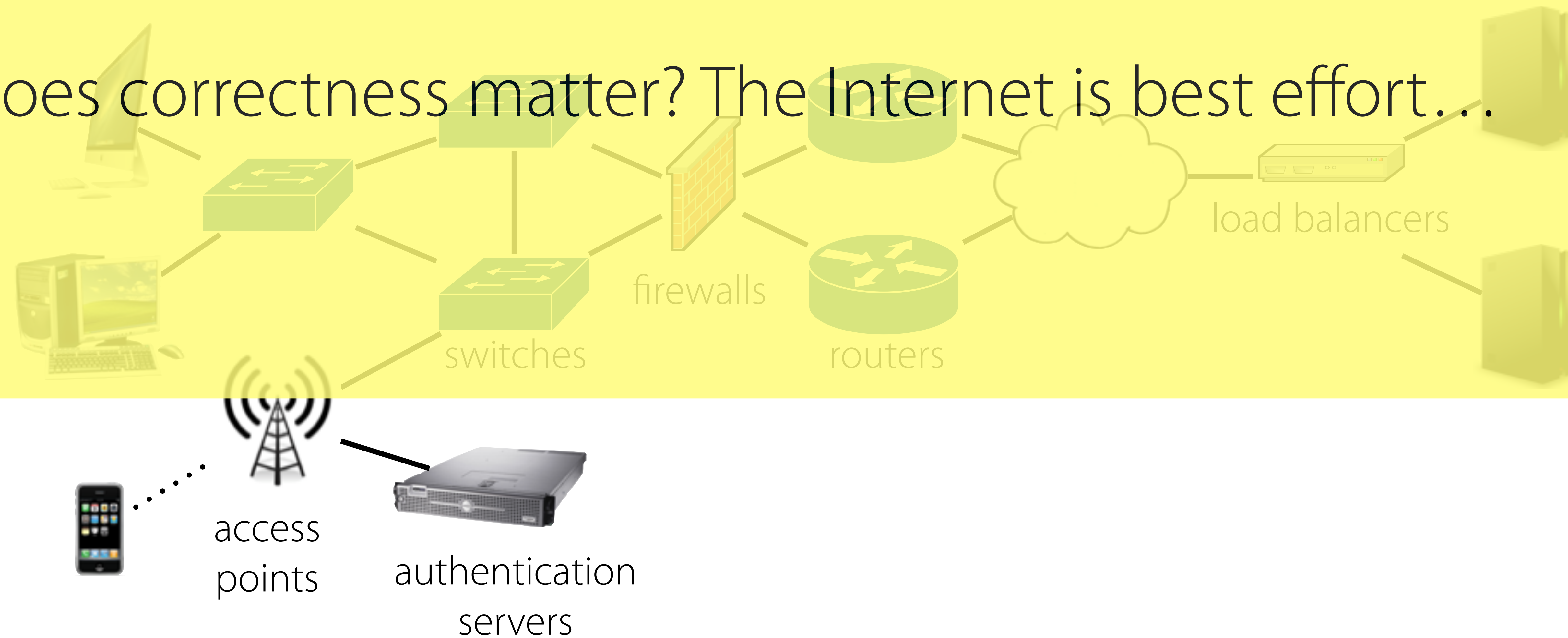


…due to the proliferation of devices, protocols, languages

Networks Today

Reasoning about network behavior is *extremely* difficult…

Does correctness matter? The Internet is best effort…



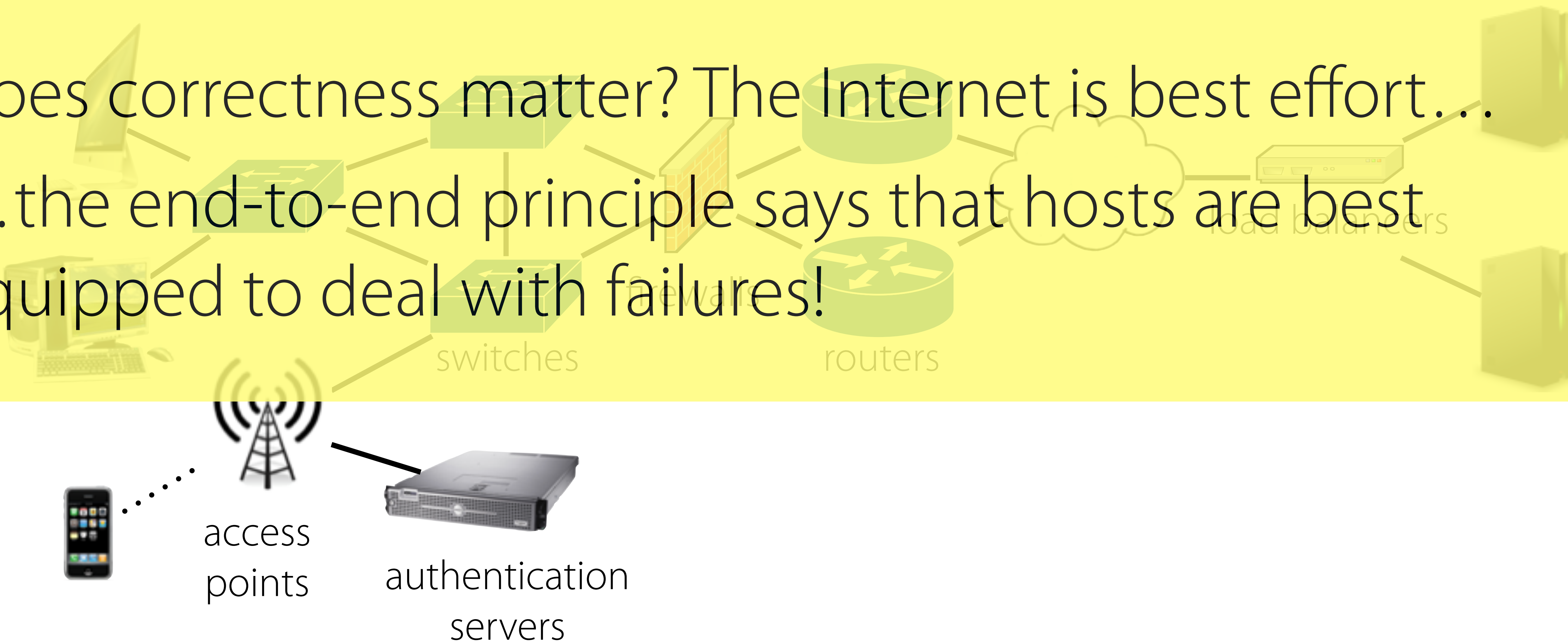
…due to the proliferation of devices, protocols, languages

Networks Today

Reasoning about network behavior is *extremely* difficult…

Does correctness matter? The Internet is best effort…

…the end-to-end principle says that hosts are best equipped to deal with failures!



…due to the proliferation of devices, protocols, languages

Example: Outages

We **discovered a misconfiguration** on this pair of switches that caused what's called a “bridge loop” in the network.

A network **change was […] executed incorrectly** […] more “stuck” volumes and added more requests to the re-mirroring storm

The malware utilized is absolutely unsophisticated […] **If Target had had a firm grasp on its network security** […] they absolutely would have observed this behavior

Experienced a network connectivity issue […] **interrupted the airline's flight departures**, airport processing and reservations systems

Example: Outages

We **discovered a misconfiguration** on this pair of switches that caused what's called a "bridge loop" in the network.

A network **change was [...] executed**

more requests to the re-mirroring storm

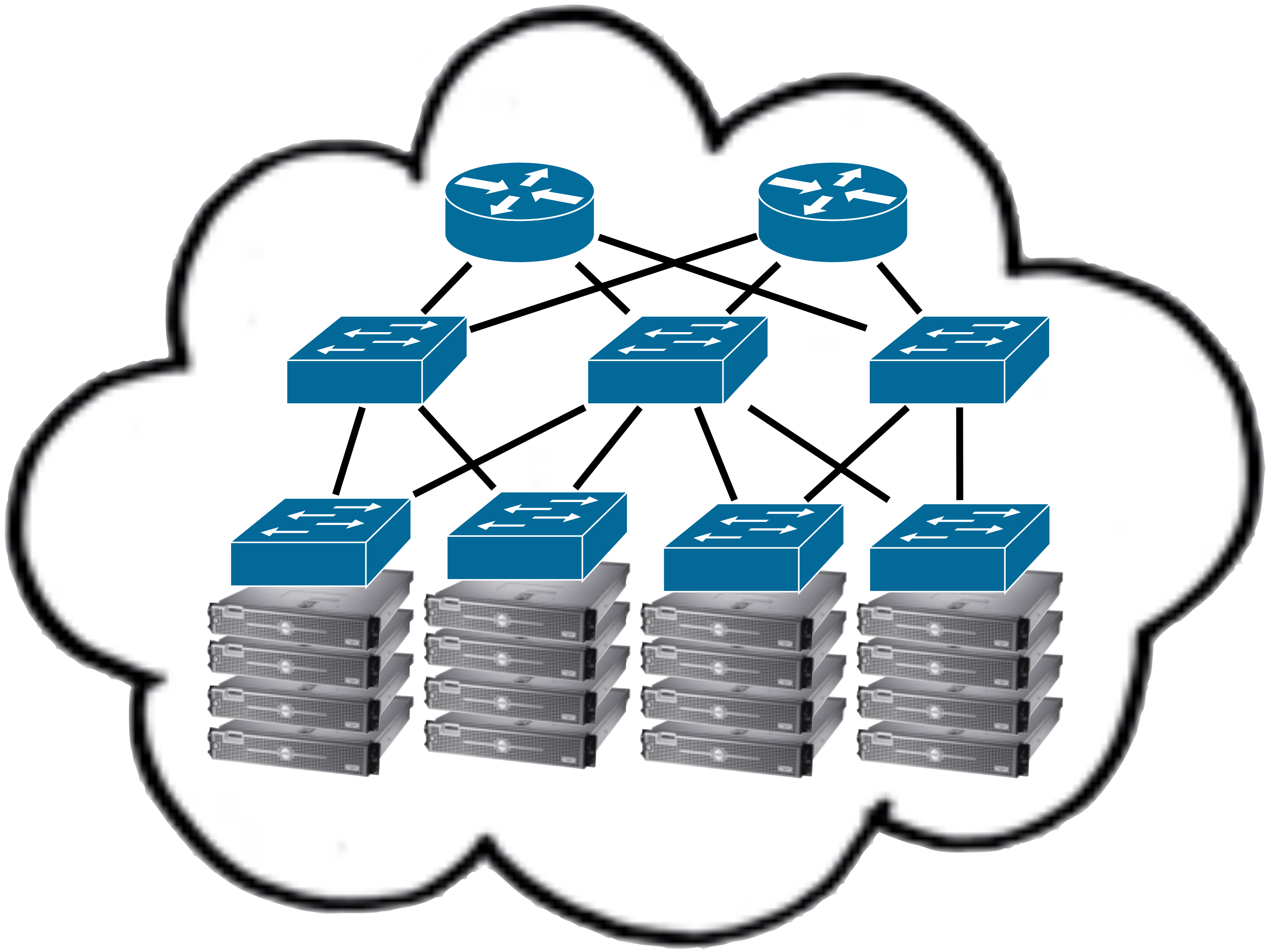
Even technically sophisticated companies are struggling to build networks that provide reliable performance.

The malware utilized is absolutely unsophisticated [...] **if Target had had a firm grasp on its network security** [...] they absolutely would have observed this behavior

Experienced a network connectivity issue [...] **interrupted the airline's flight departures**, airport processing and reservations systems

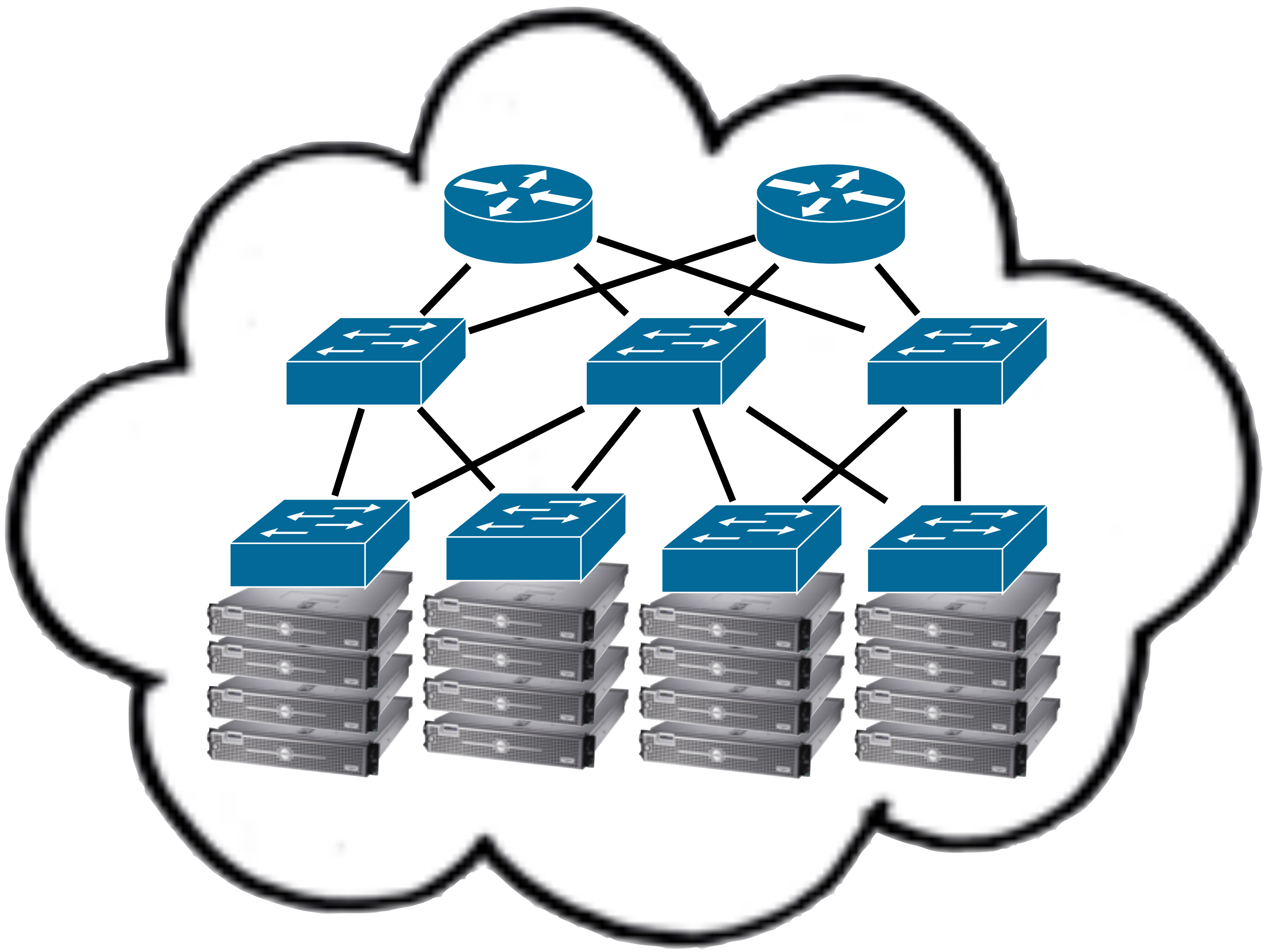
Example: Cloud Computing

Would you relocate critical infrastructure to the cloud…



Example: Cloud Computing

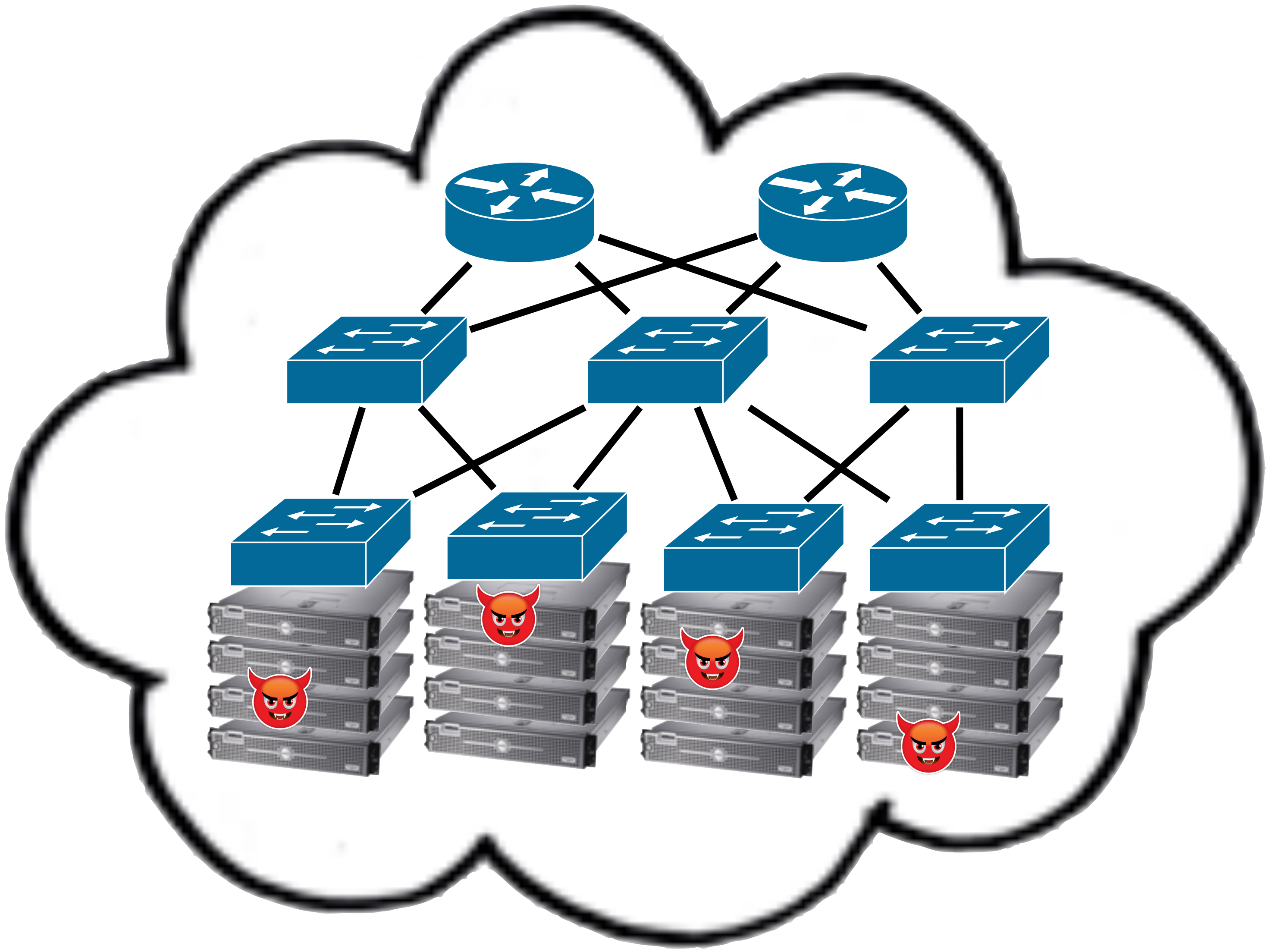
Would you relocate critical infrastructure to the cloud…



…if your traffic was *not* guaranteed to be isolated from other tenants during periods of routine maintenance?

Example: Cloud Computing

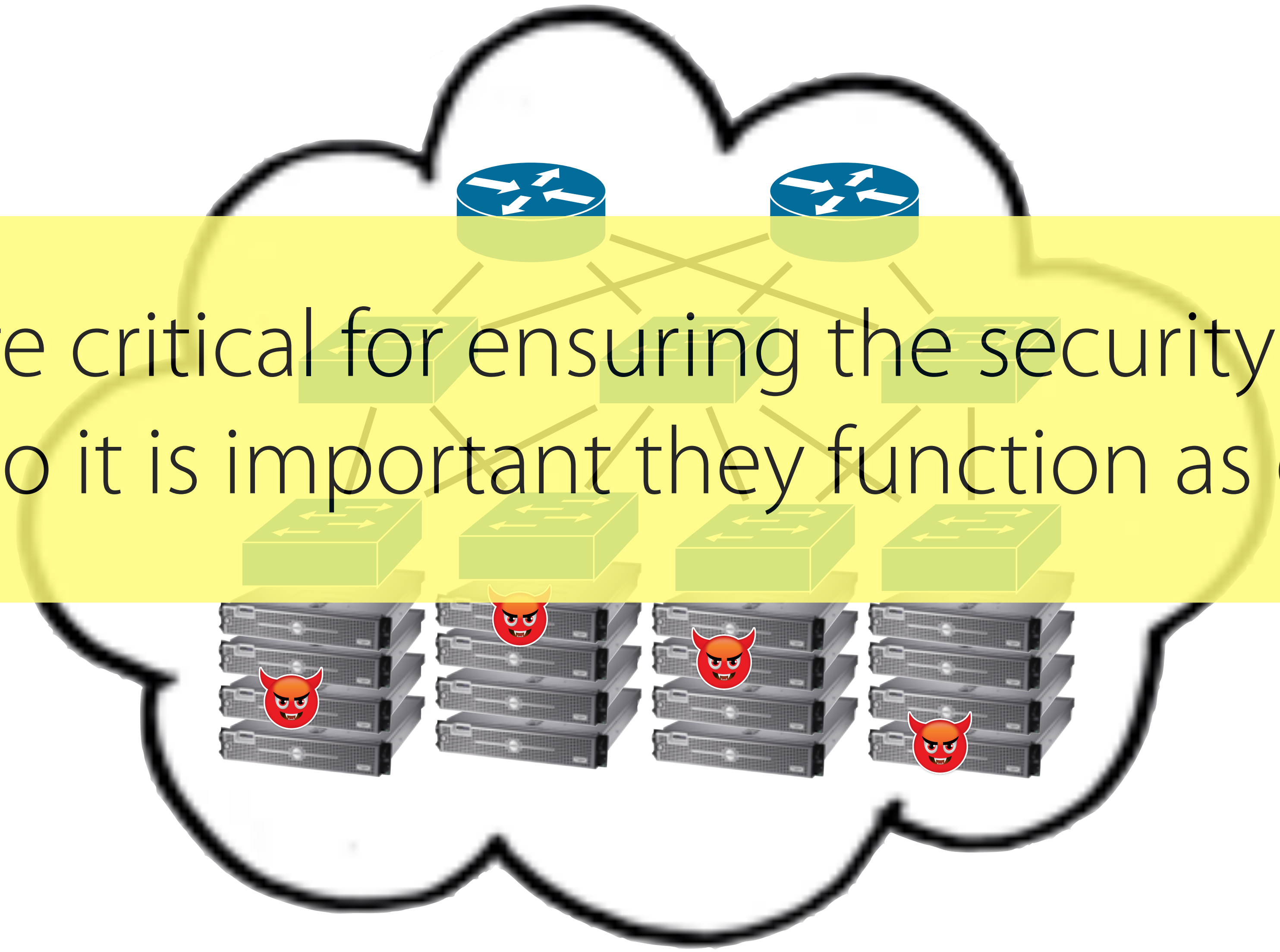
Would you relocate critical infrastructure to the cloud…



…if your traffic was *not* guaranteed to be isolated from other tenants during periods of routine maintenance?

Example: Cloud Computing

Would you relocate critical infrastructure to the cloud…

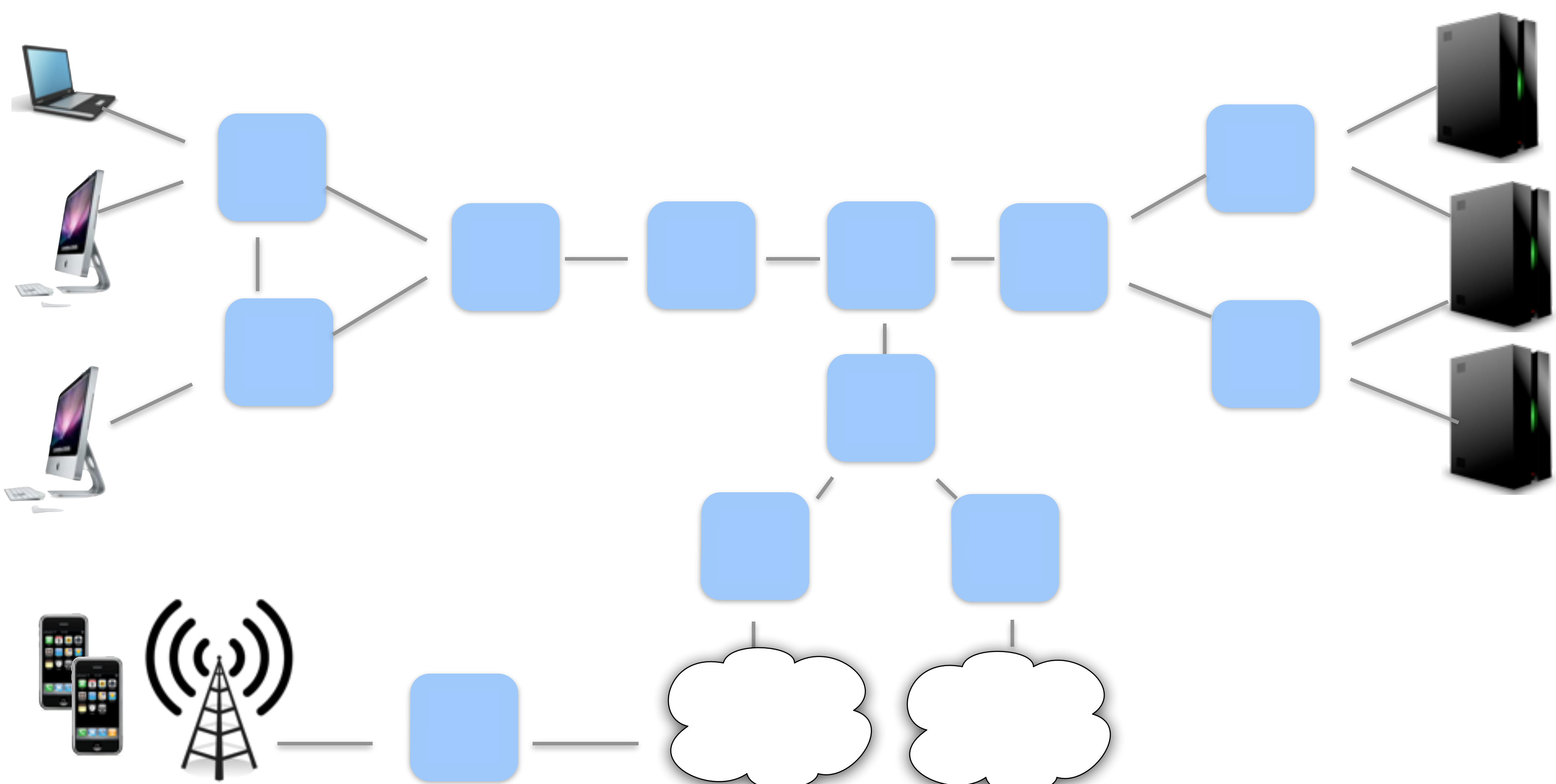


Networks are critical for ensuring the security of many systems… so it is important they function as expected

…if your traffic was *not* guaranteed to be isolated from other tenants during periods of routine maintenance?

Software-Defined Networking

A clean-slate programmable network architecture



A Major Trend in Networking

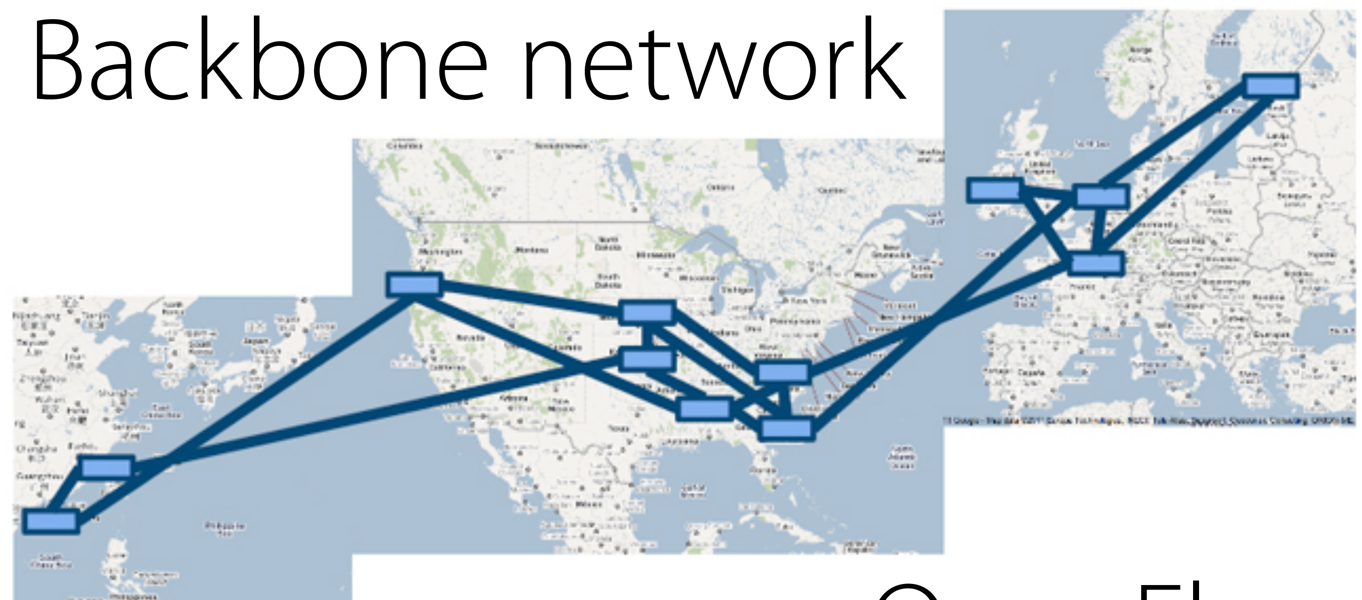
ONF

Deutsche Telekom · facebook · Goldman Sachs · verizon

Google · Microsoft · NTT Communications · YAHOO!

Google

Backbone network



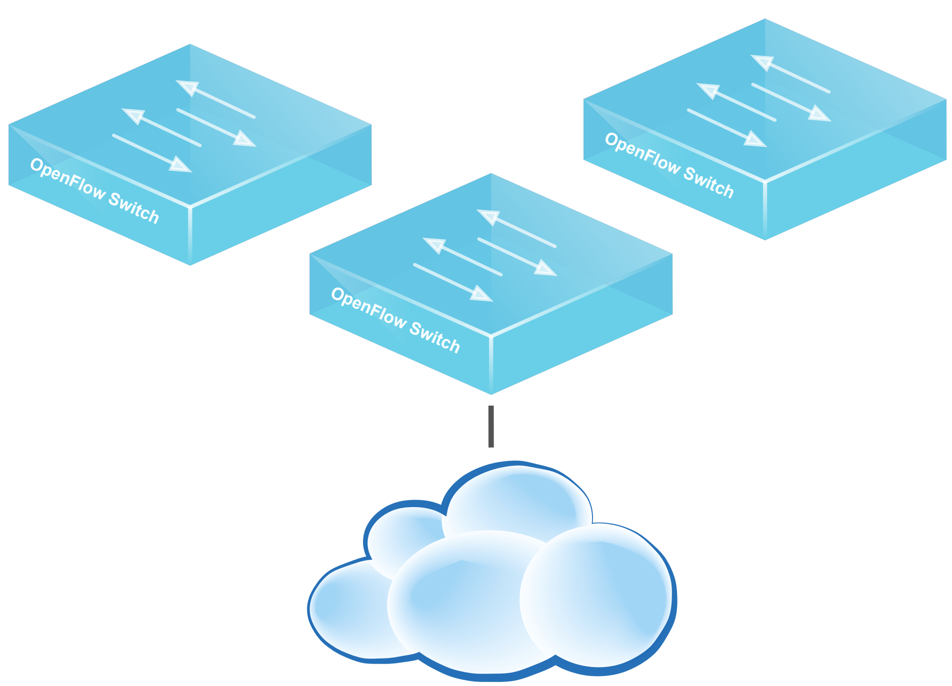
runs OpenFlow

nicira

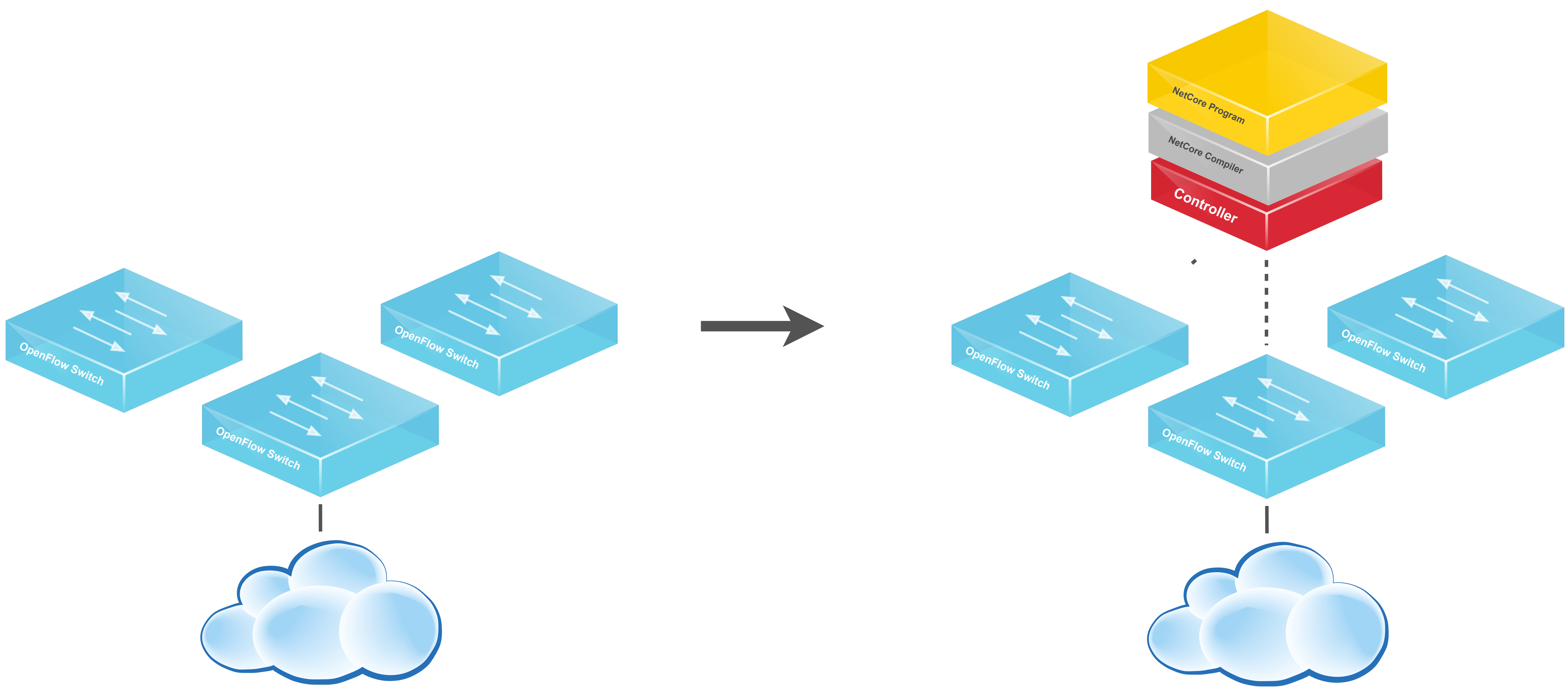
Acquired for $1.2B

frenetic

Vision: program networks using a high-level language, generate low-level machine code using a compiler, and verify formal properties of networks automatically



Vision: program networks using a high-level language, generate low-level machine code using a compiler, and verify formal properties of networks automatically



Tutorial Outline

frenetic

Tutorial Outline

Part I: Ox

- OpenFlow Overview
- Ox Applications

frenetic

Part II: Frenetic

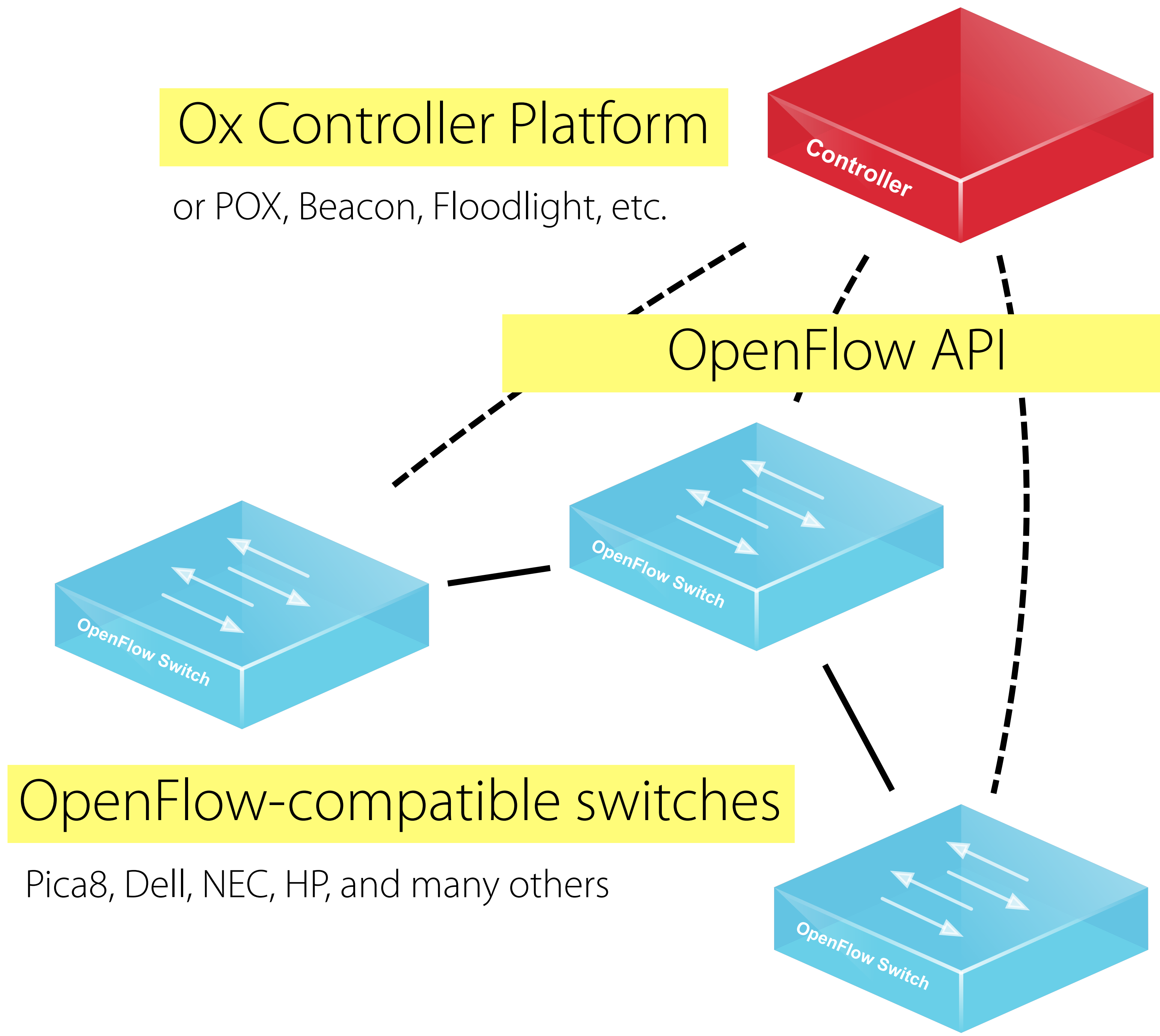
- NetKAT Overview
- NetKAT Applications

Part III: Formal methods

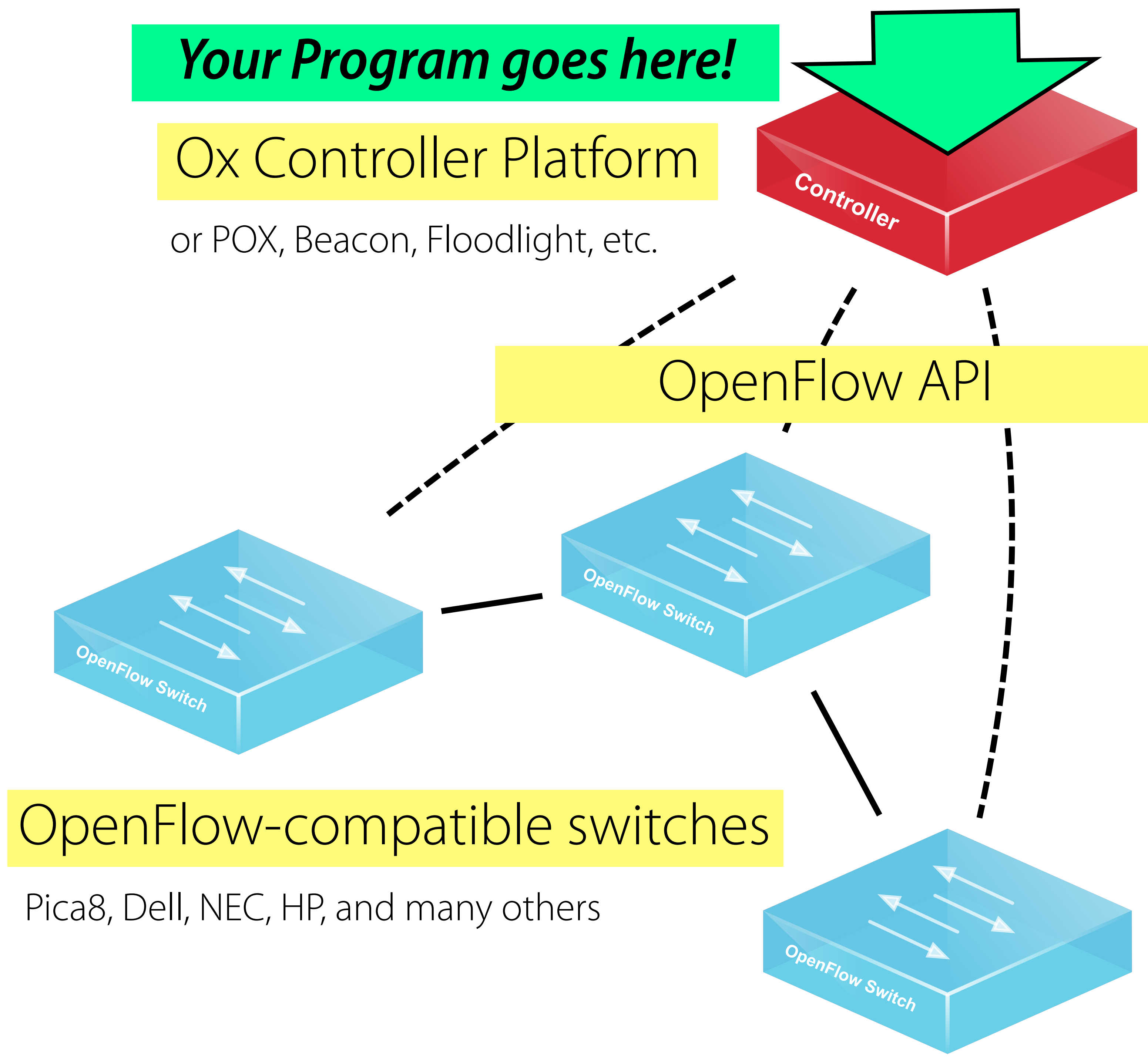
- Update consistency
- Verification and reasoning

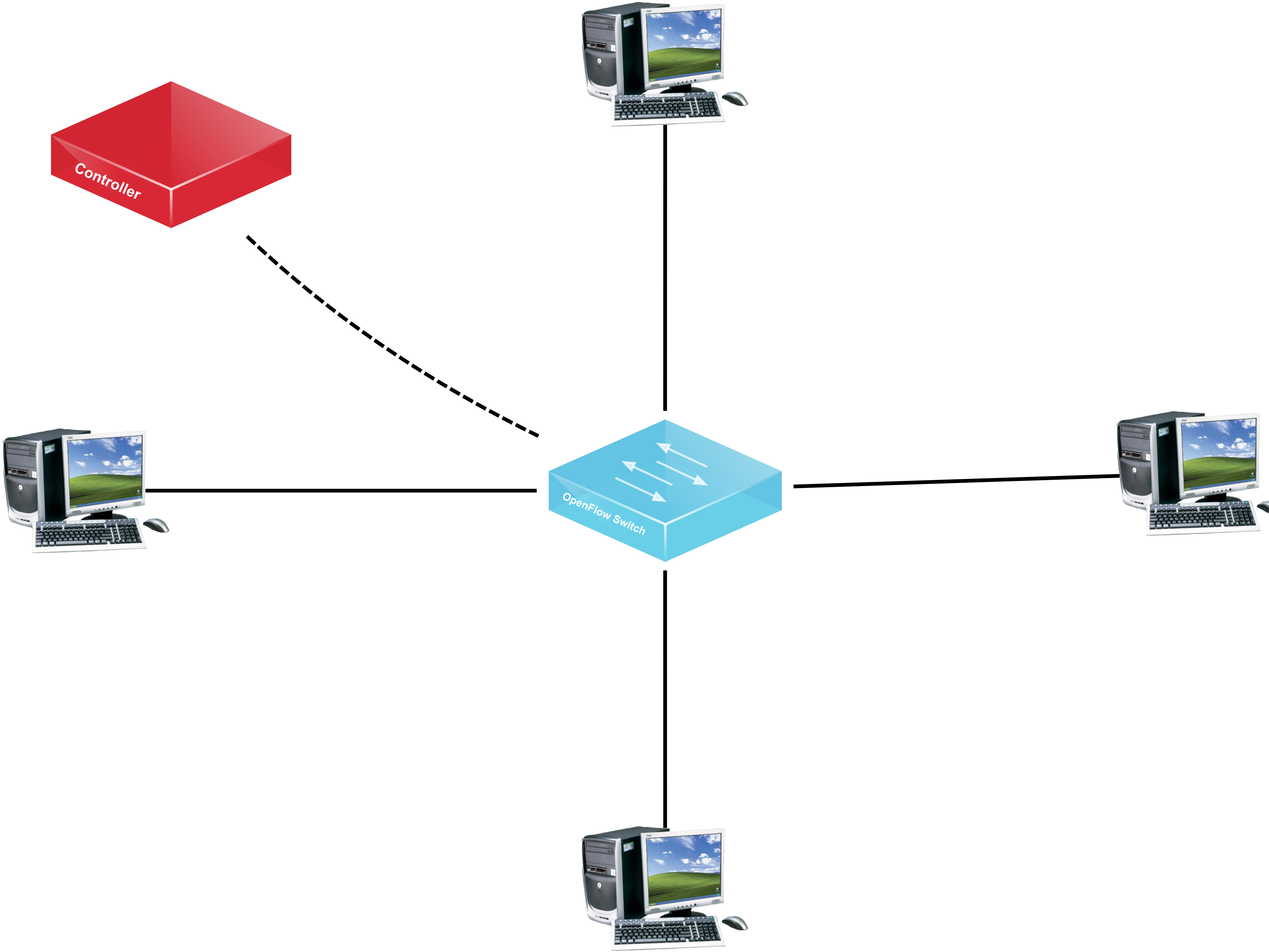
OpenFlow Overview

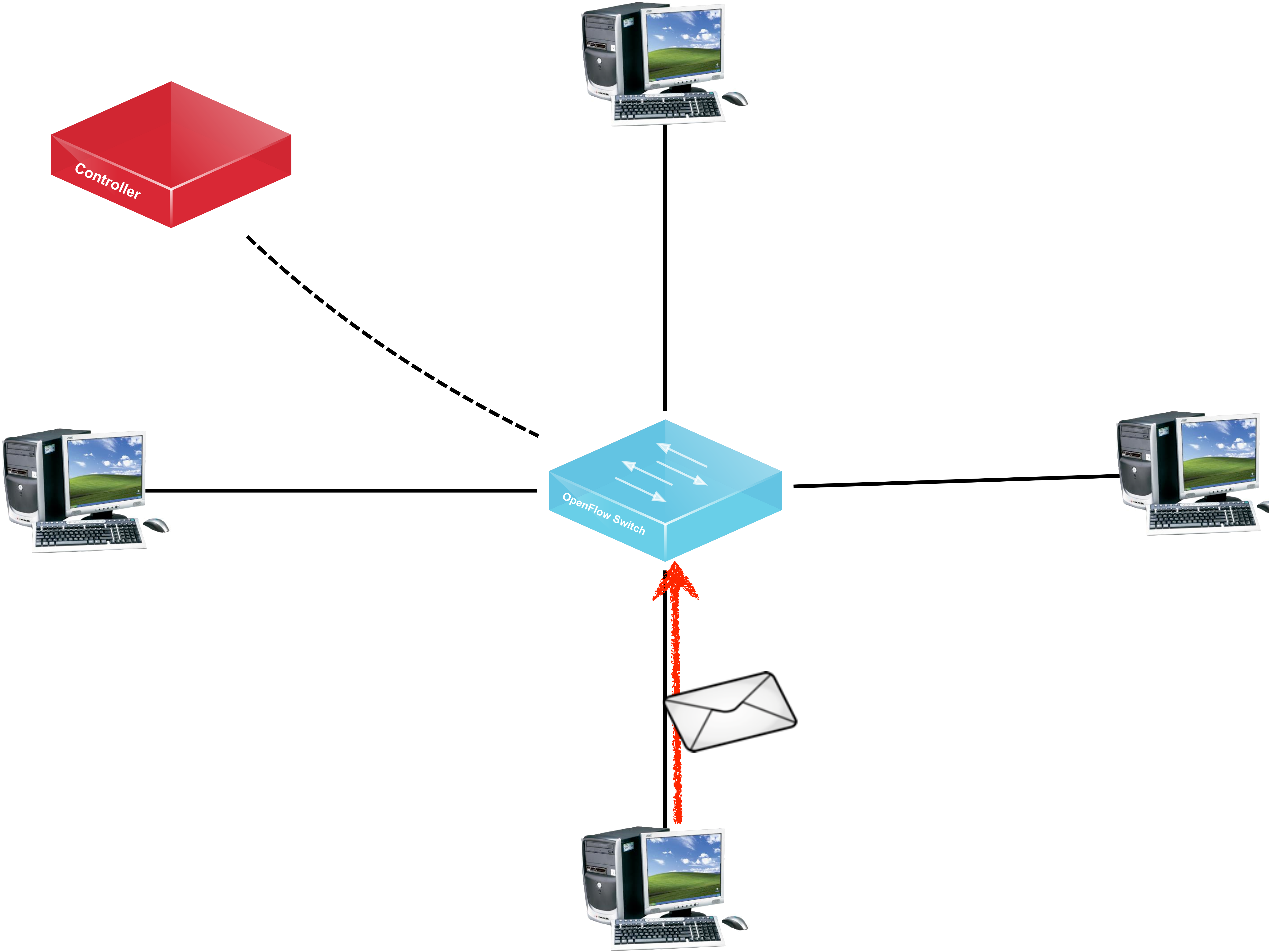
OpenFlow Architecture

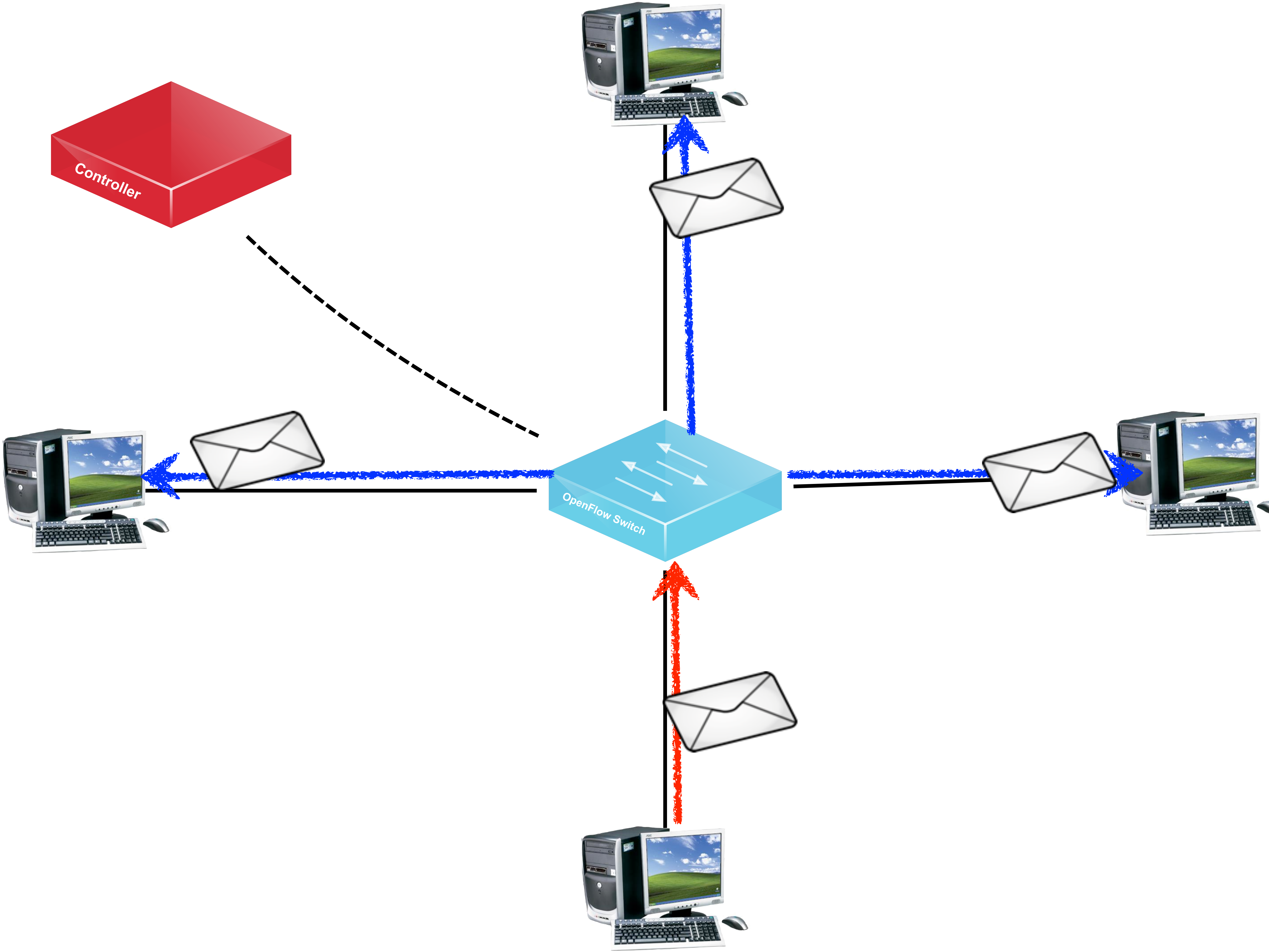


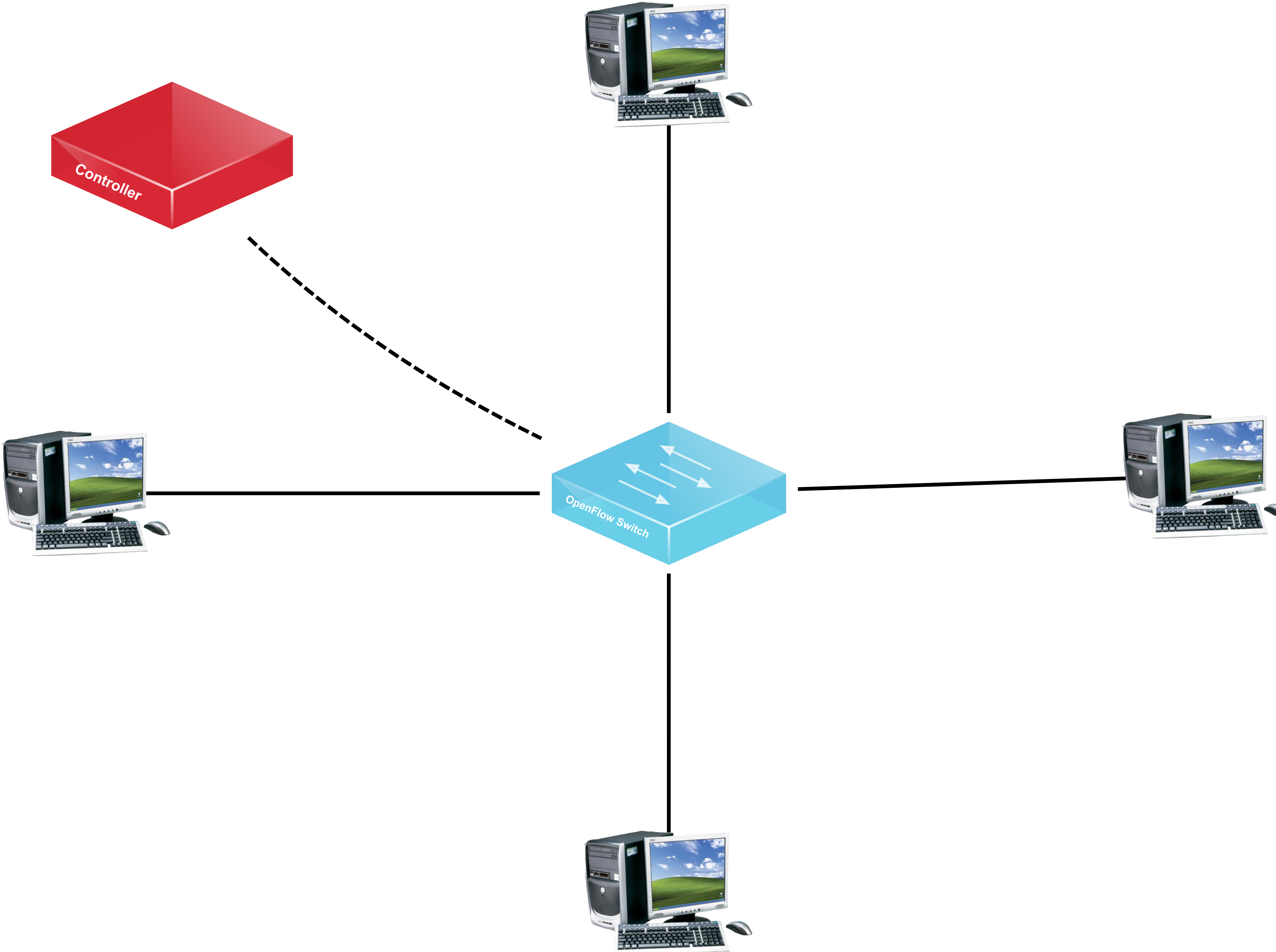
OpenFlow Architecture

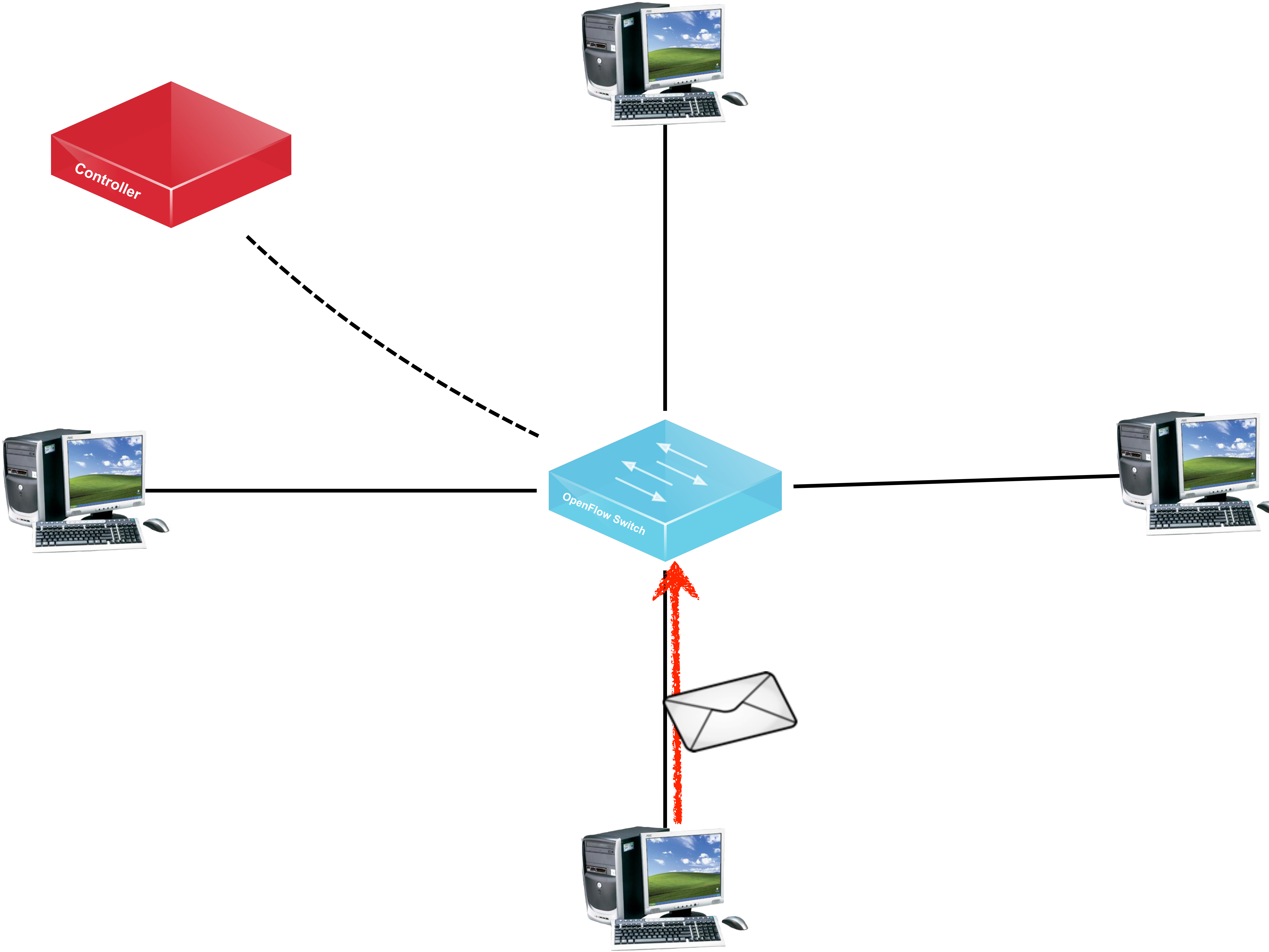


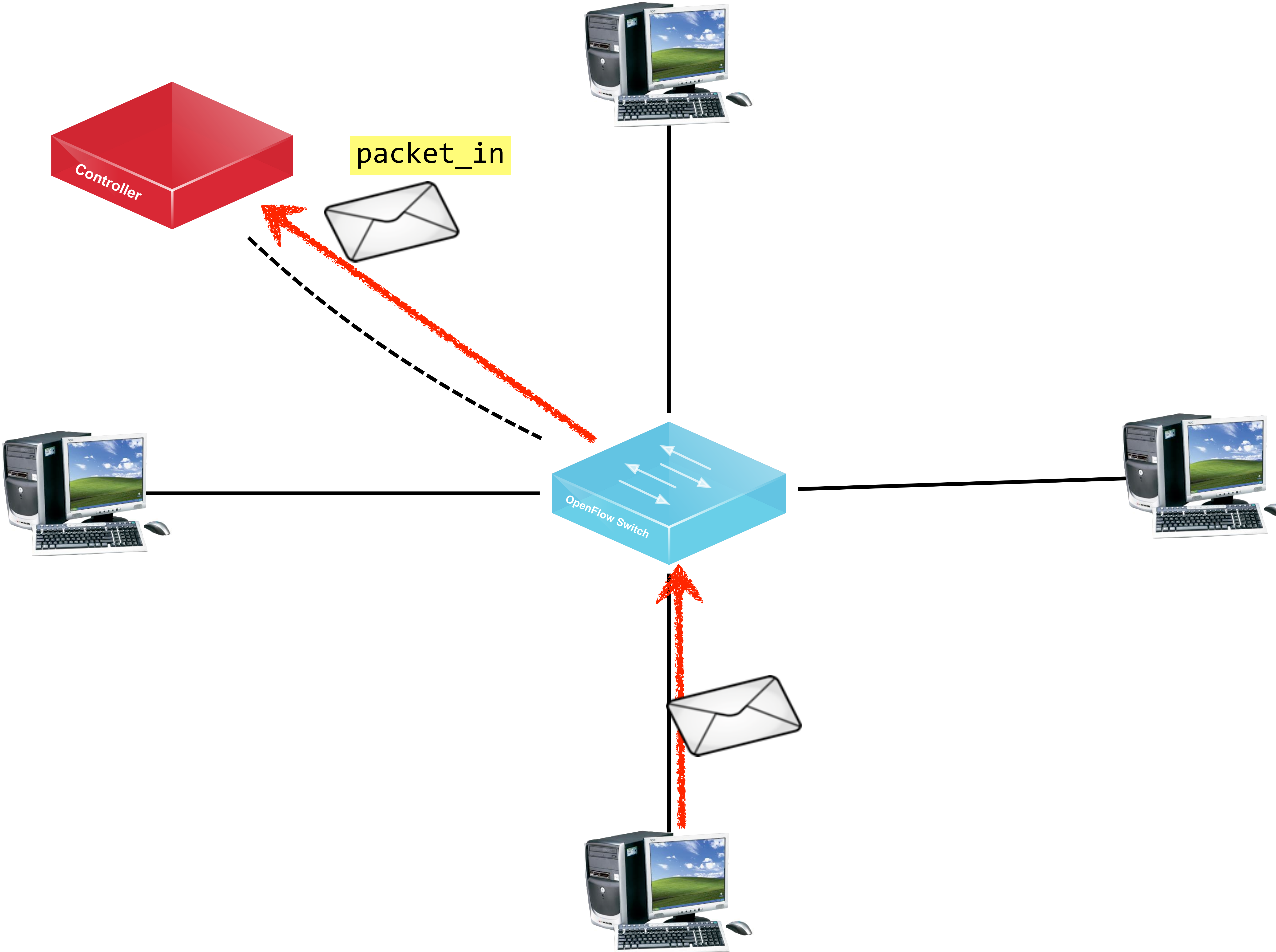

Controller
OpenFlow Switch


Controller
OpenFlow Switch

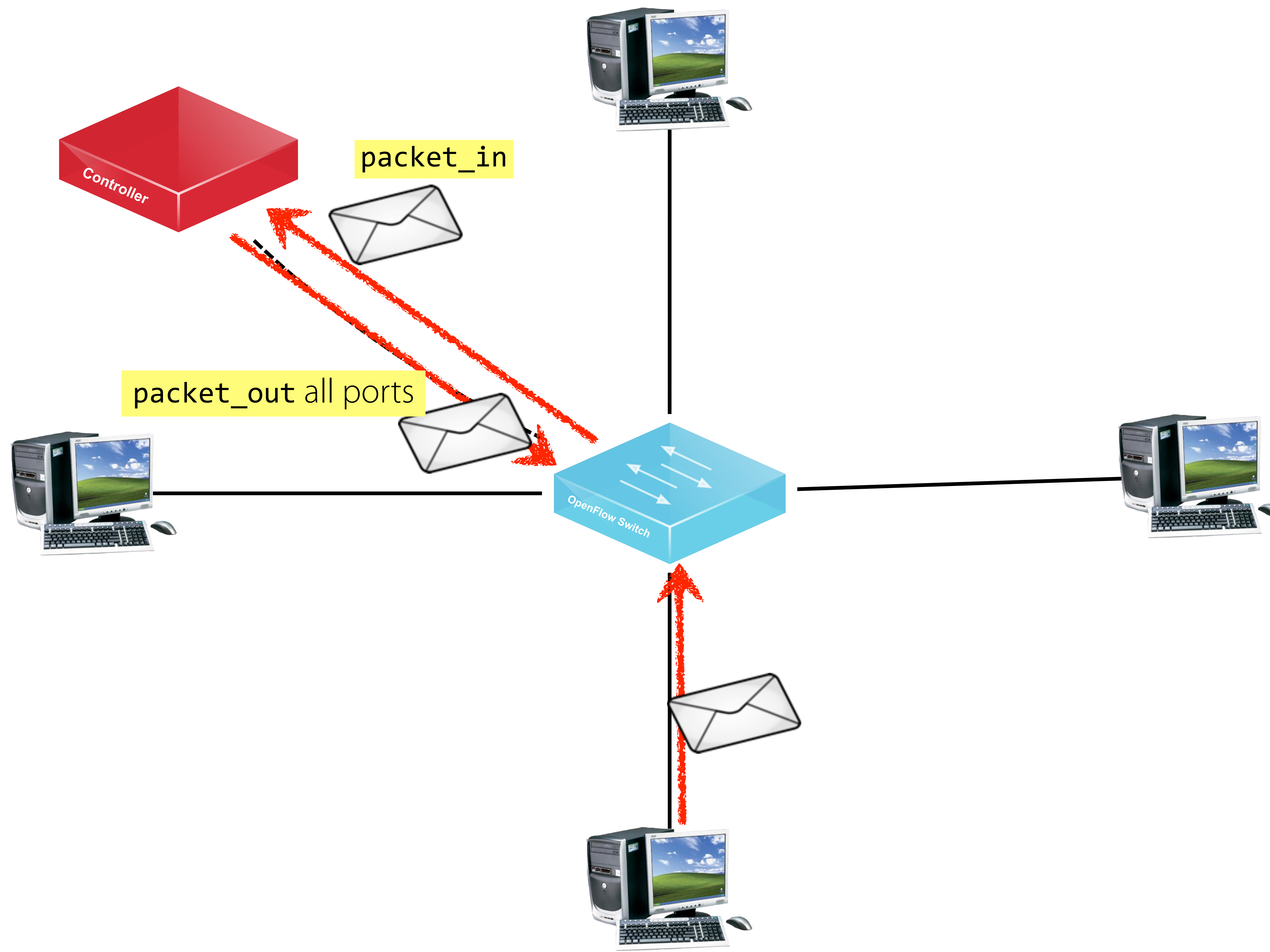

Controller
OpenFlow Switch

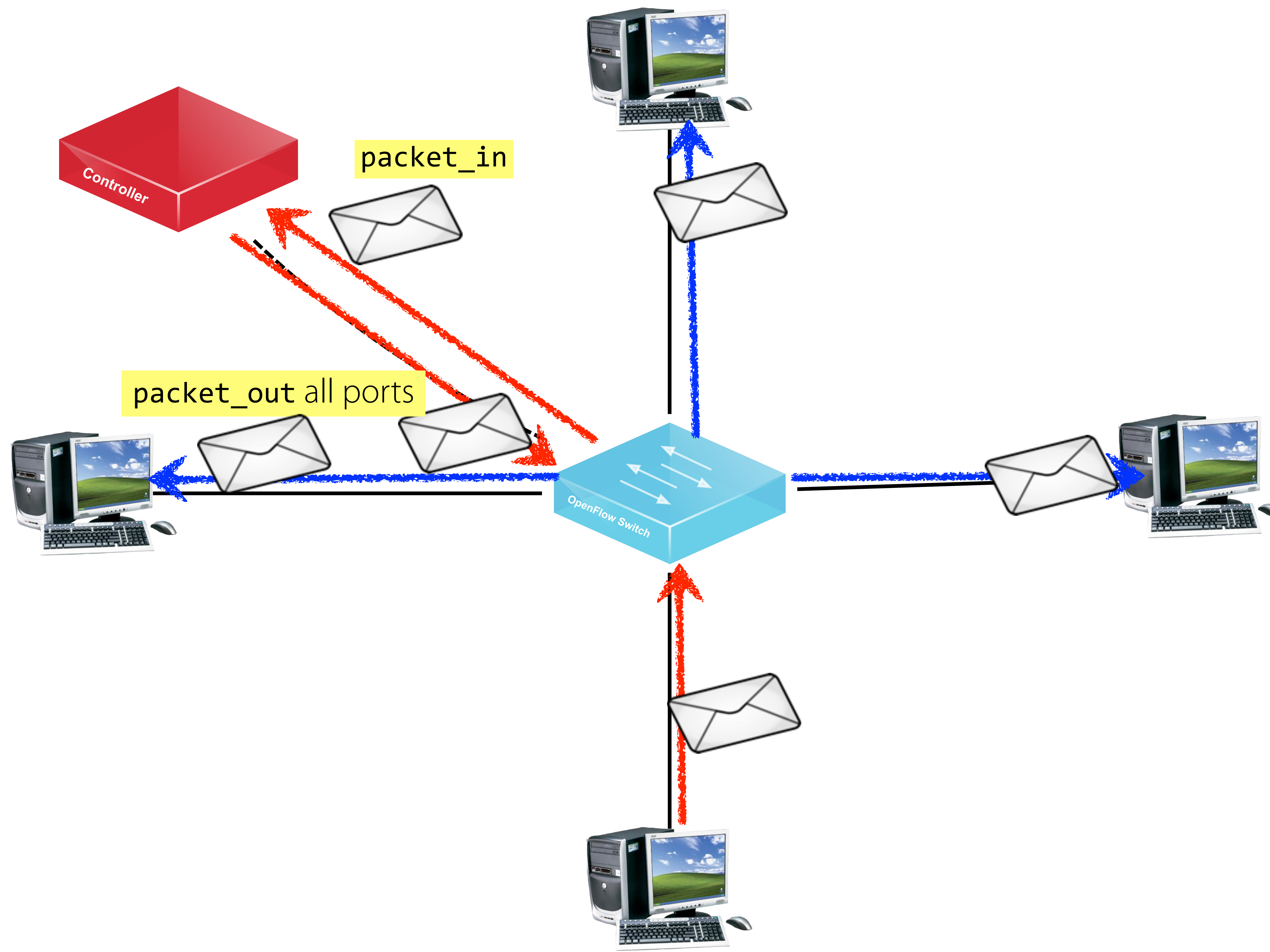

Controller
OpenFlow Switch

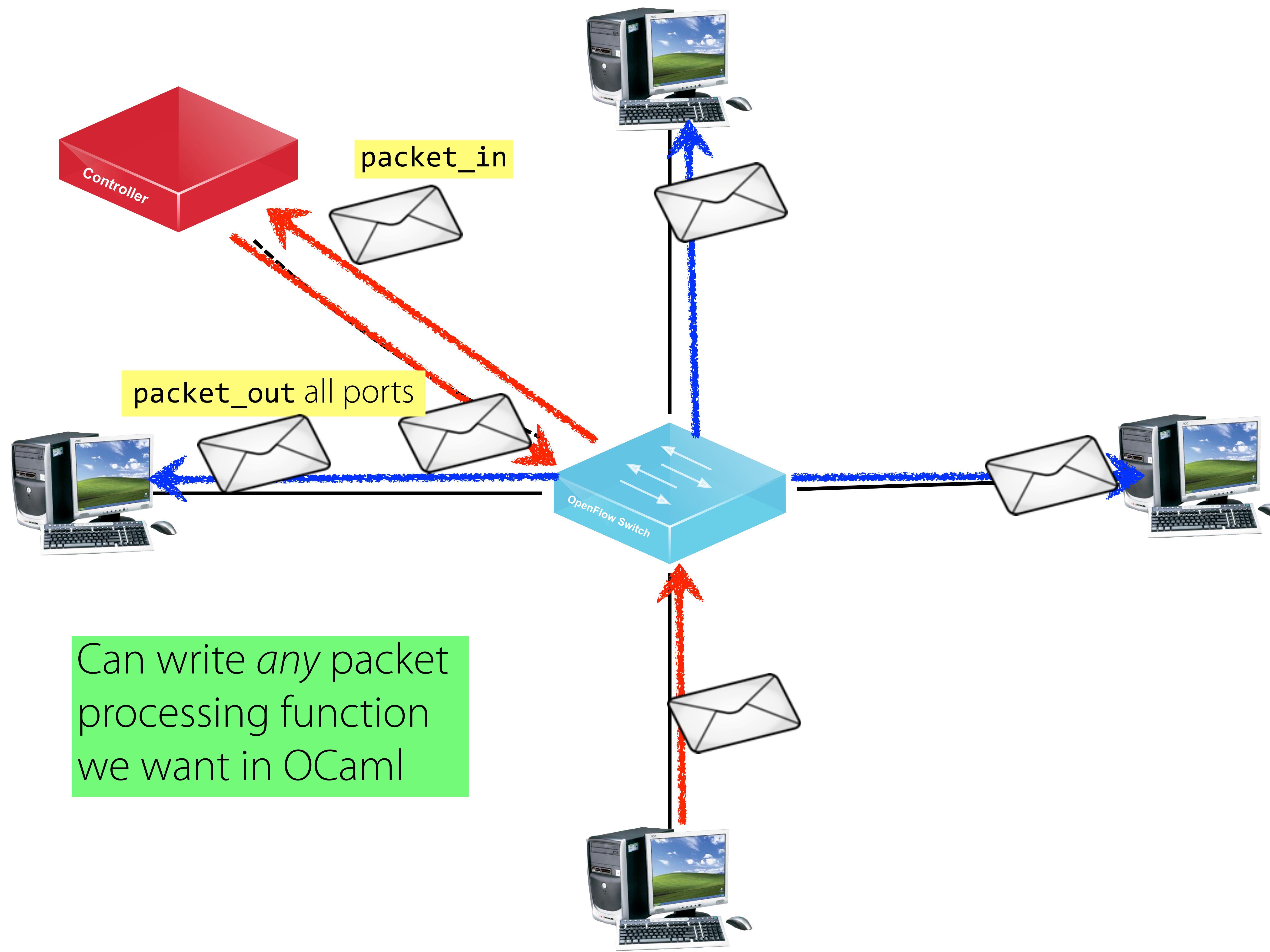

Controller
OpenFlow Switch



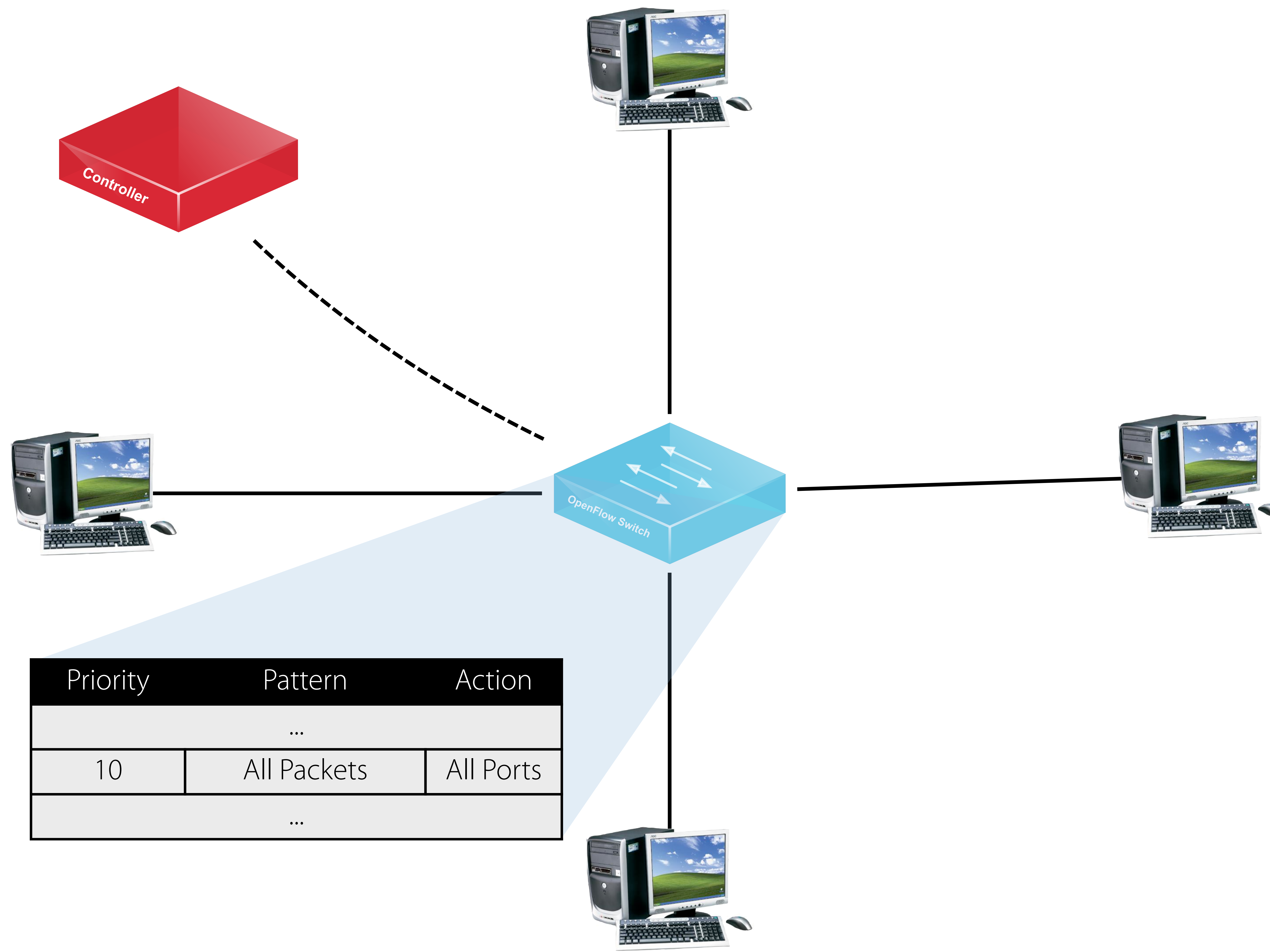
packet_in
Controller
OpenFlow Switch


packet_in
Controller
packet_out all ports
OpenFlow Switch

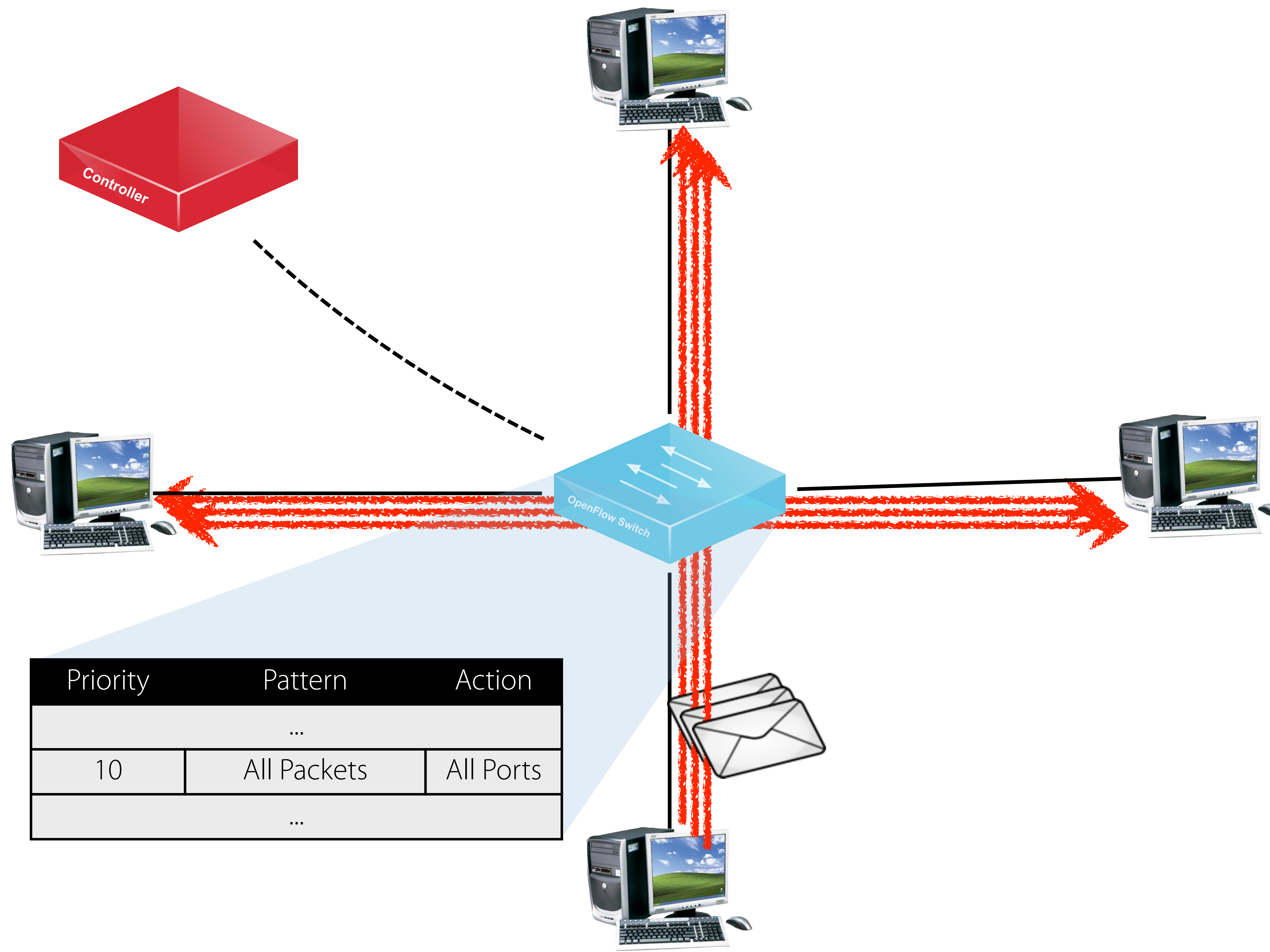

Controller
packet_in
packet_out all ports
OpenFlow Switch



Can write *any* packet processing function we want in OCaml



Priority	Pattern	Action
...		
10	All Packets	All Ports
...		

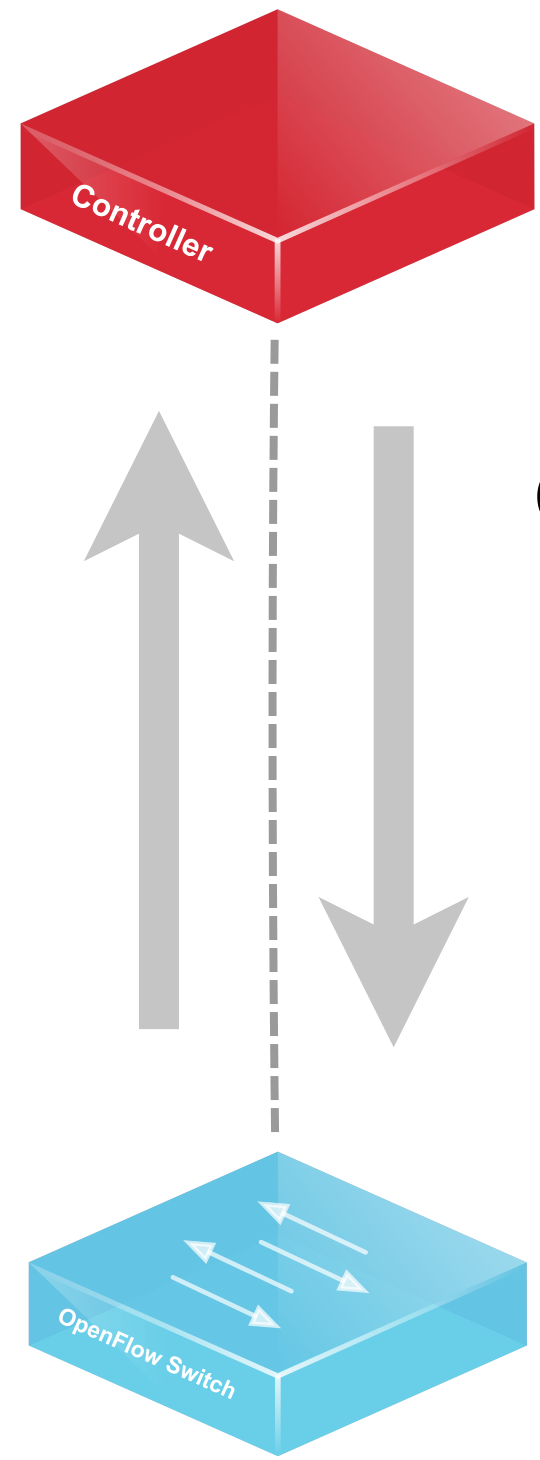


Priority	Pattern	Action
	...	
10	All Packets	All Ports
	...	

OpenFlow API

Switch to controller:

- `switch_connected`
- `switch_disconnected`
- `packet_in`
- `stats_reply`



Controller to switch:

- `packet_out`
- `flow_mod`
- `stats_request`

Demo: Ox Repeater

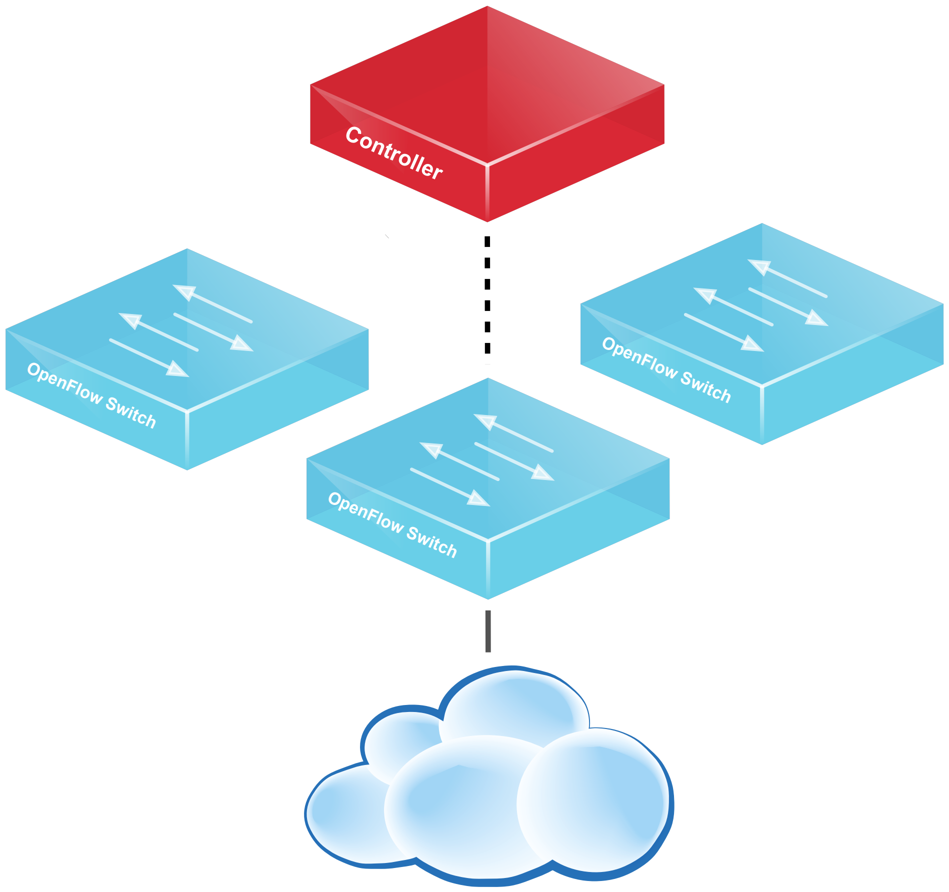
Frenetic Overview

Machine Languages

OpenFlow is a machine language

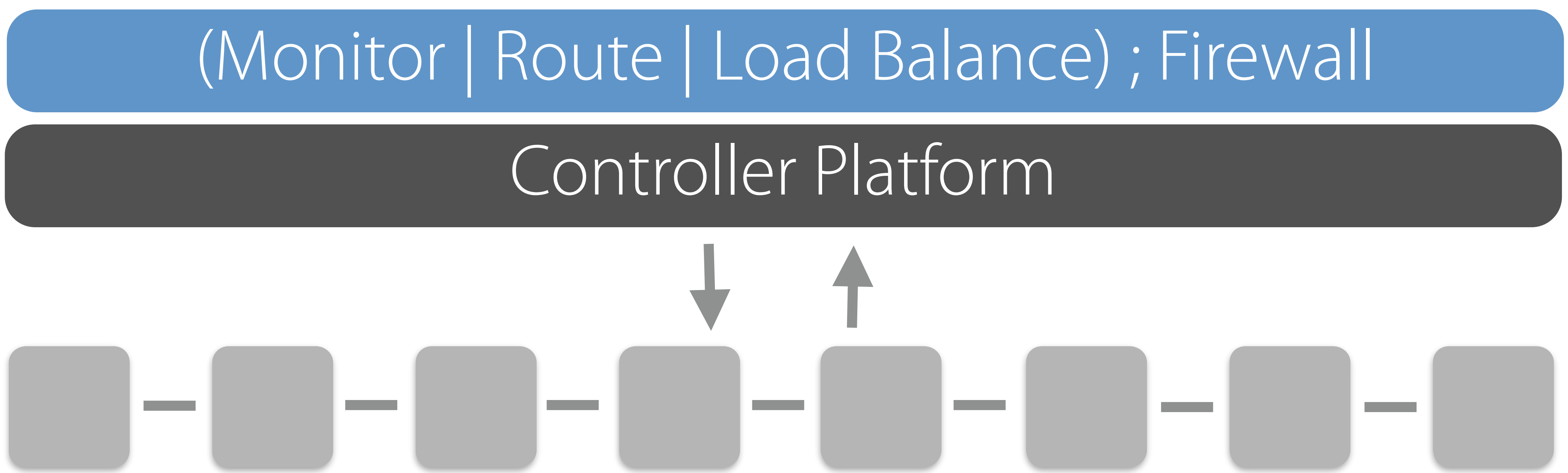
Programmers must think in terms of low-level concepts such as:

- Flow tables
- Matches
- Priorities
- Timeouts
- Events
- Callbacks



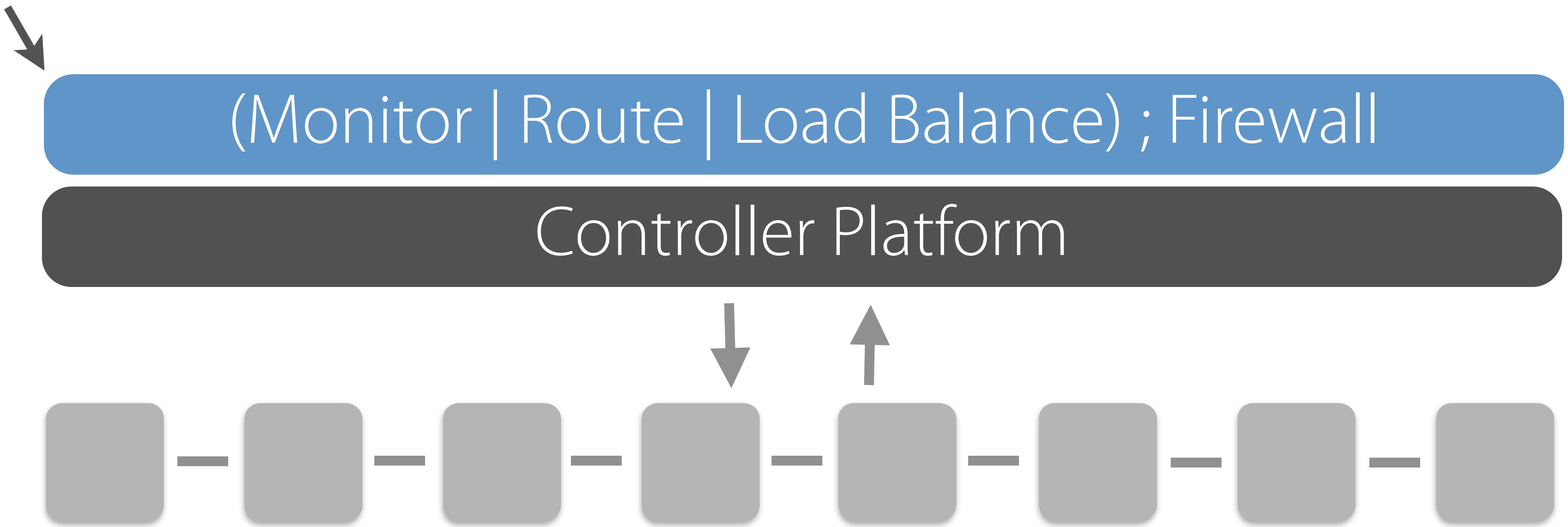
Key issue: programs don't compose!

Current Controllers



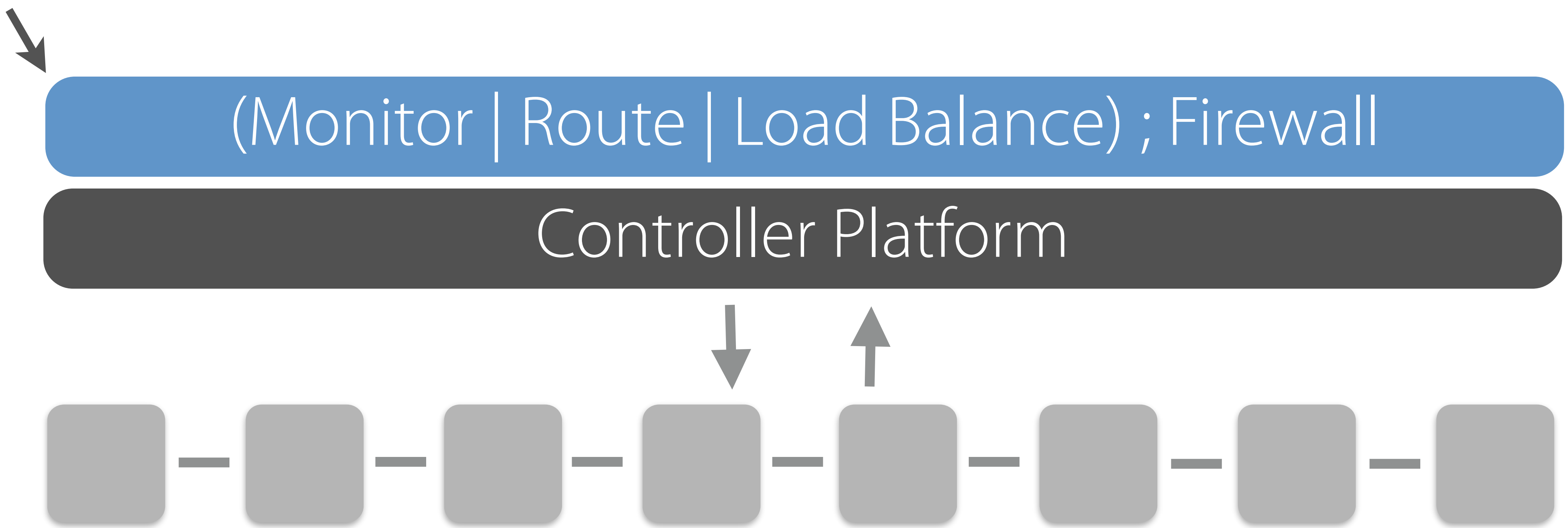
Current Controllers

One monolithic application



Current Controllers

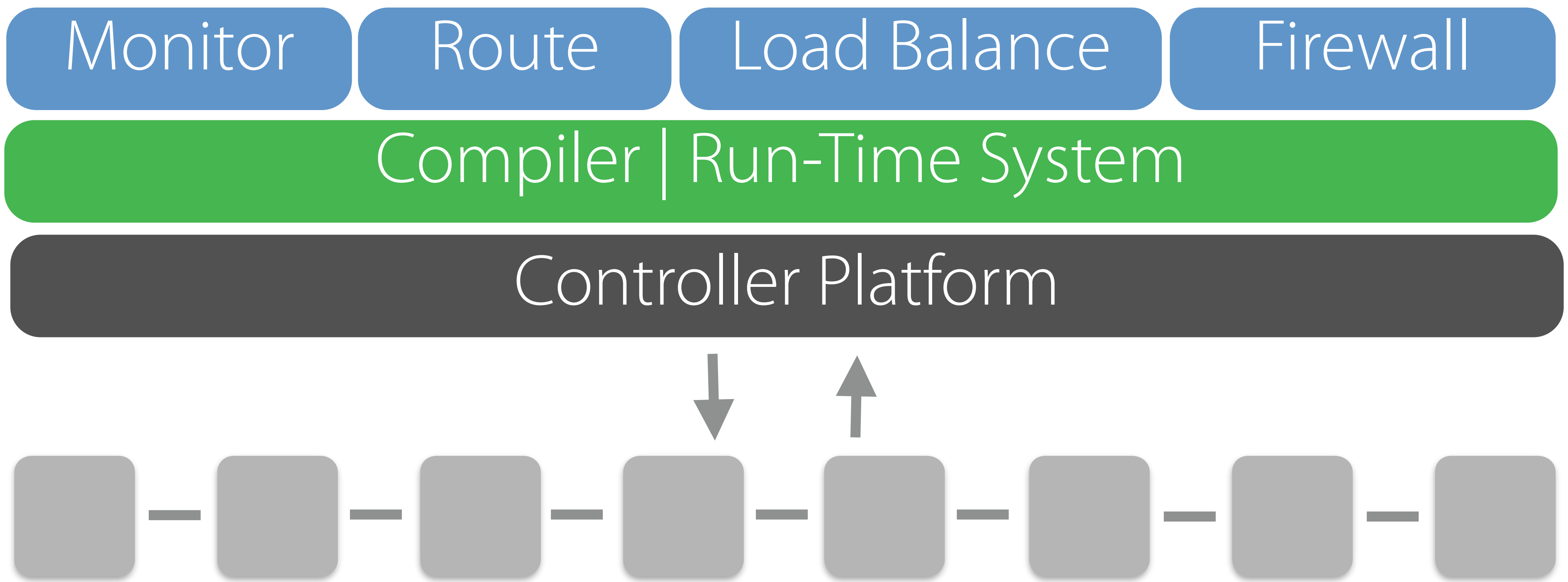
One monolithic application



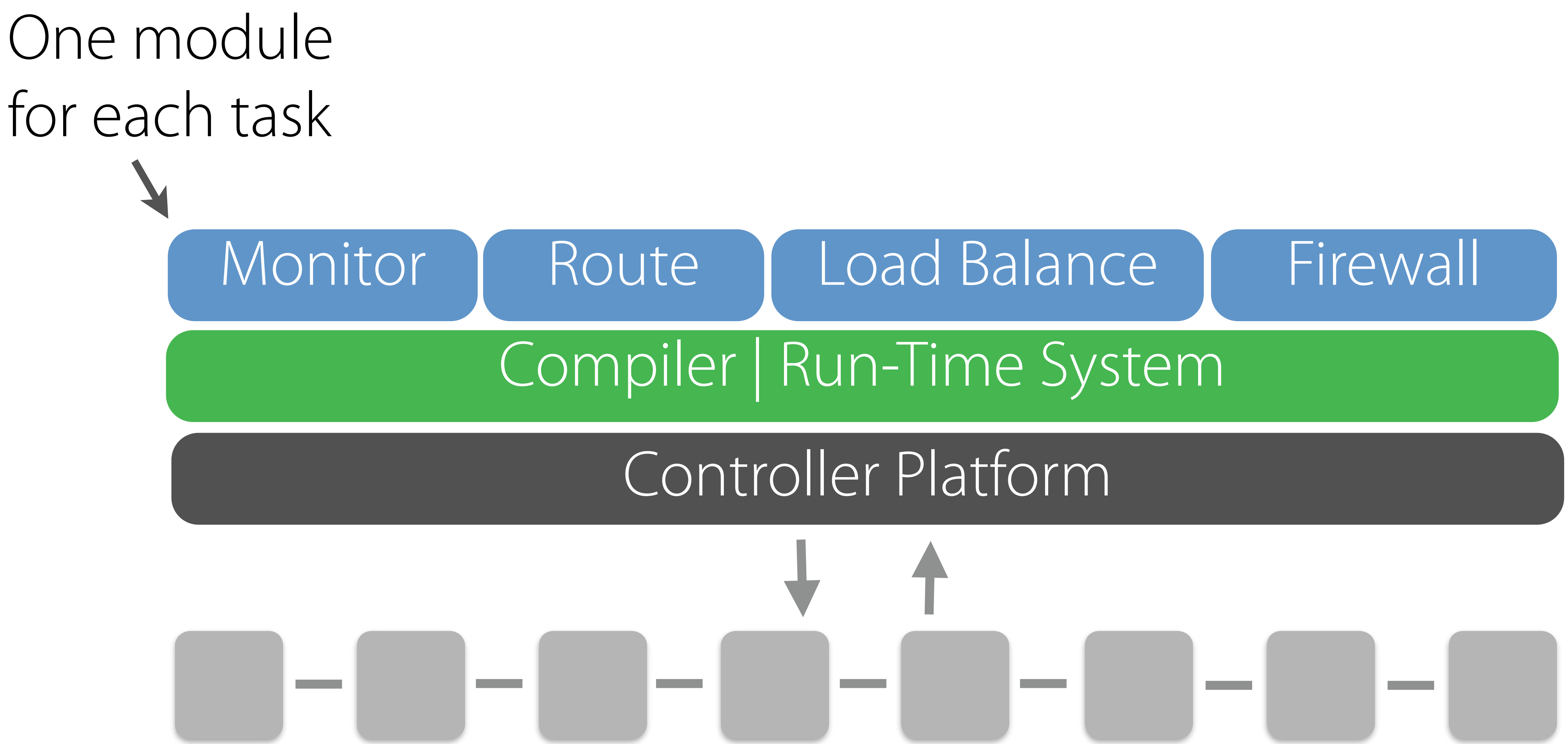
Challenges:

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

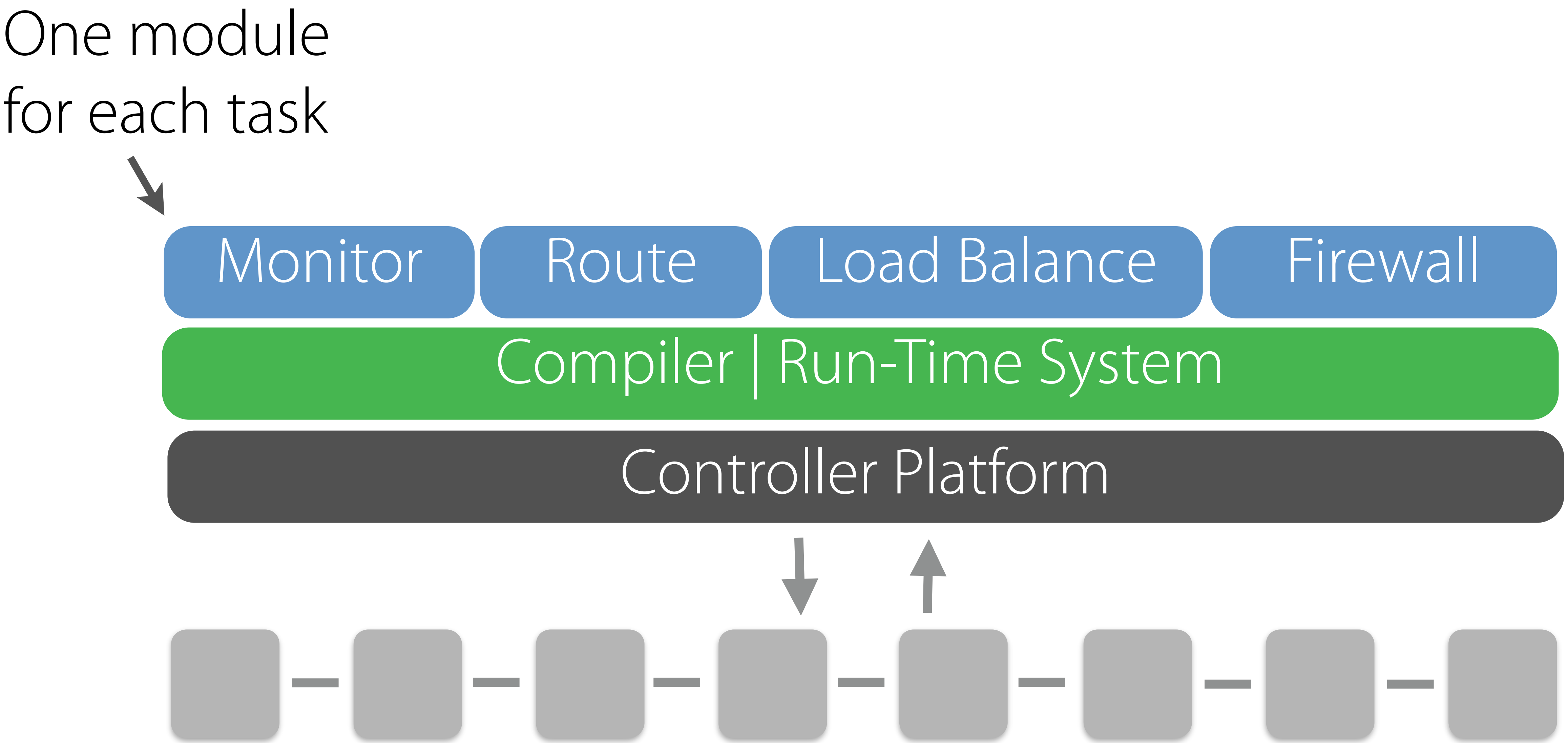
Language-Based Approach



Language-Based Approach



Language-Based Approach



Benefits:

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

Programming Languages

Frenetic is a programming language

Programmers work in terms of natural constructs:

- Functions
- Predicates
- Relational operators
- Logical properties

Compiler bridges the gap between these abstractions and their implementations in OpenFlow

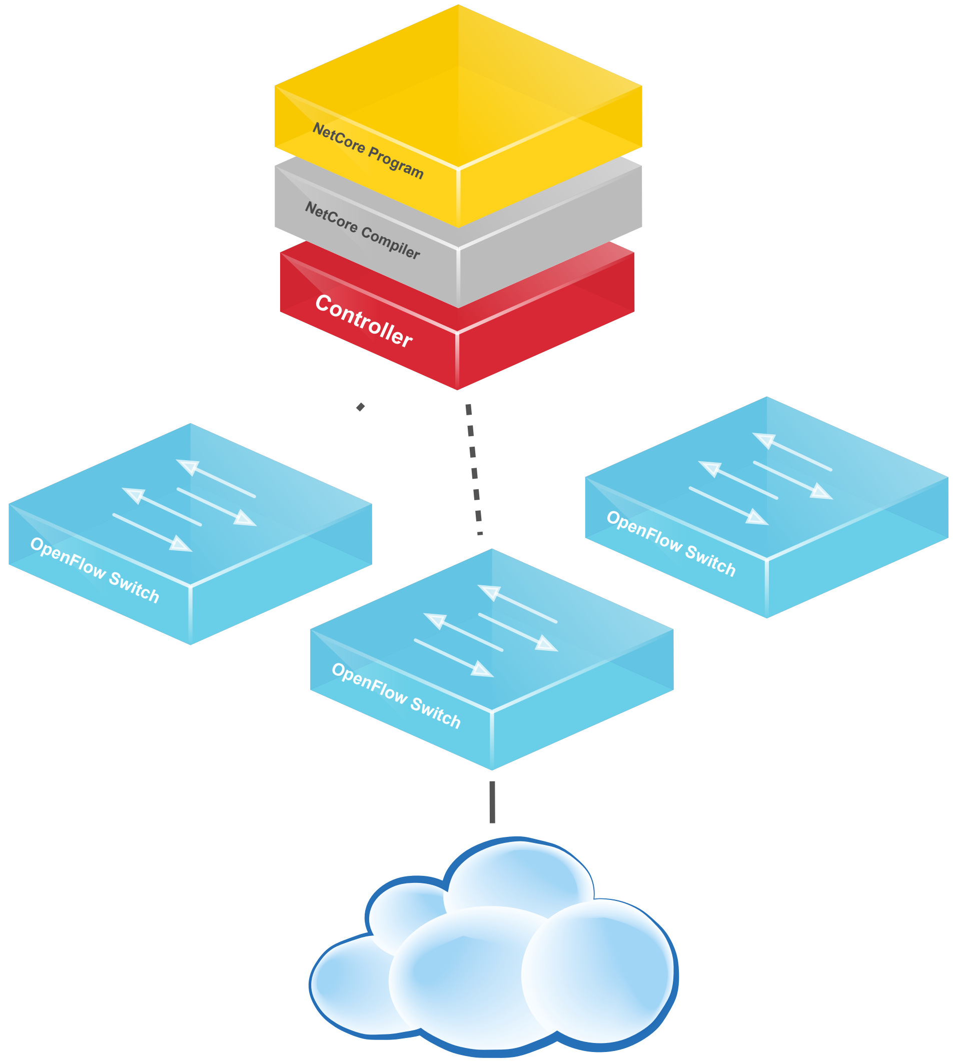
Programming Languages

Frenetic is a programming language

Programmers work in terms of natural constructs:

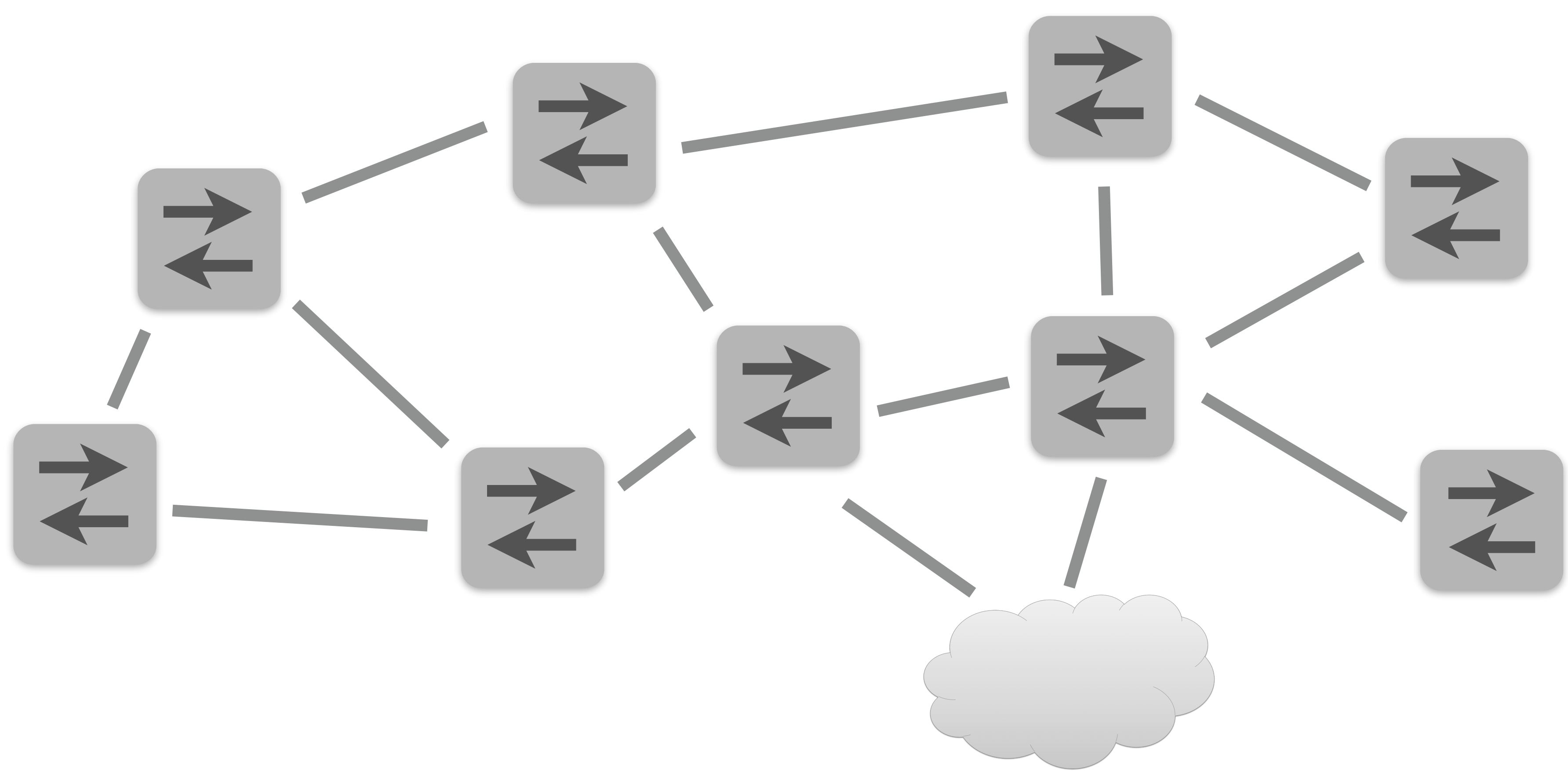
- Functions
- Predicates
- Relational operators
- Logical properties

Compiler bridges the gap between these abstractions and their implementations in OpenFlow



Network-Wide Programming

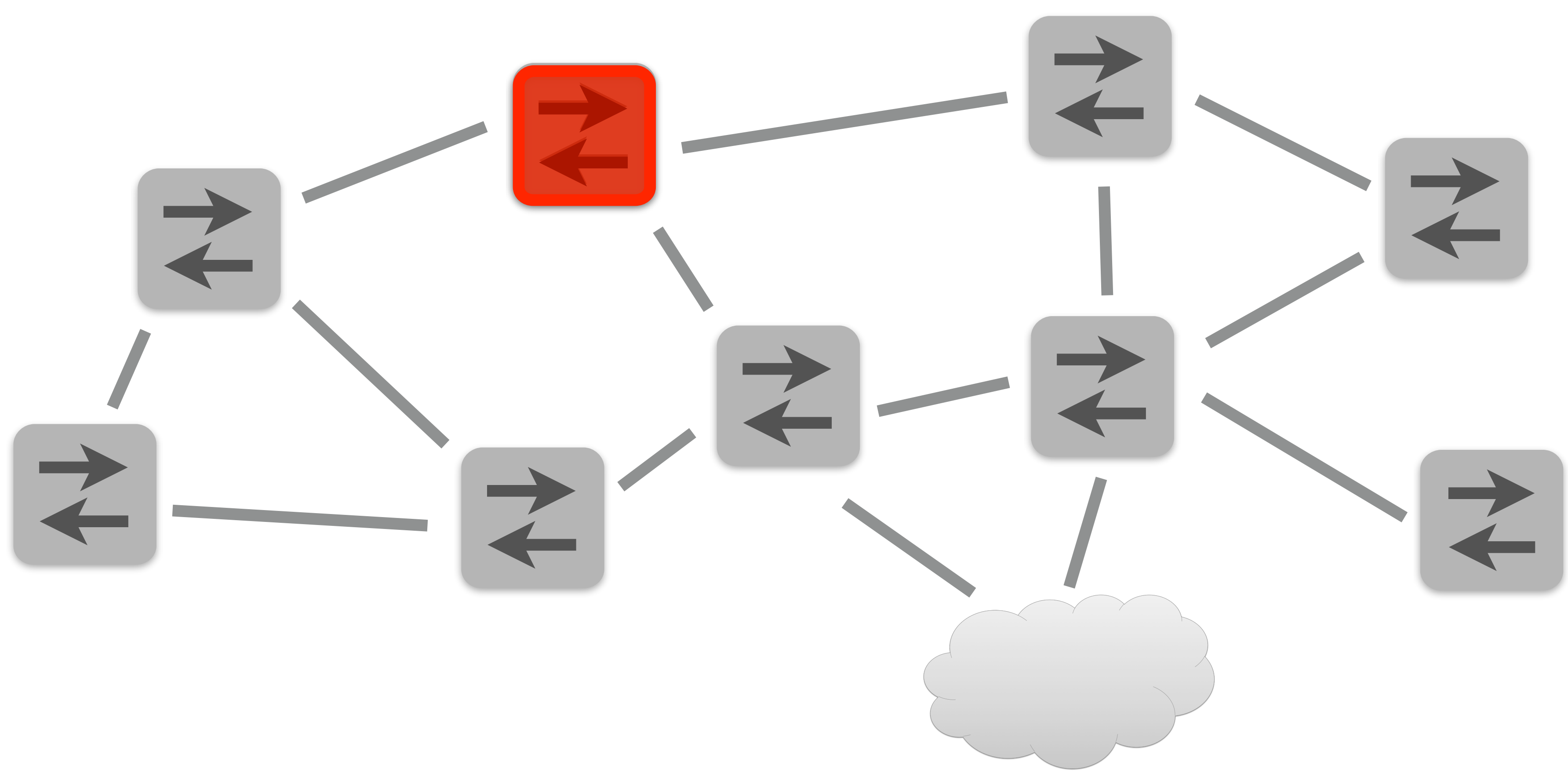
What features should an SDN language provide?



Network-Wide Programming

What features should an SDN language provide?

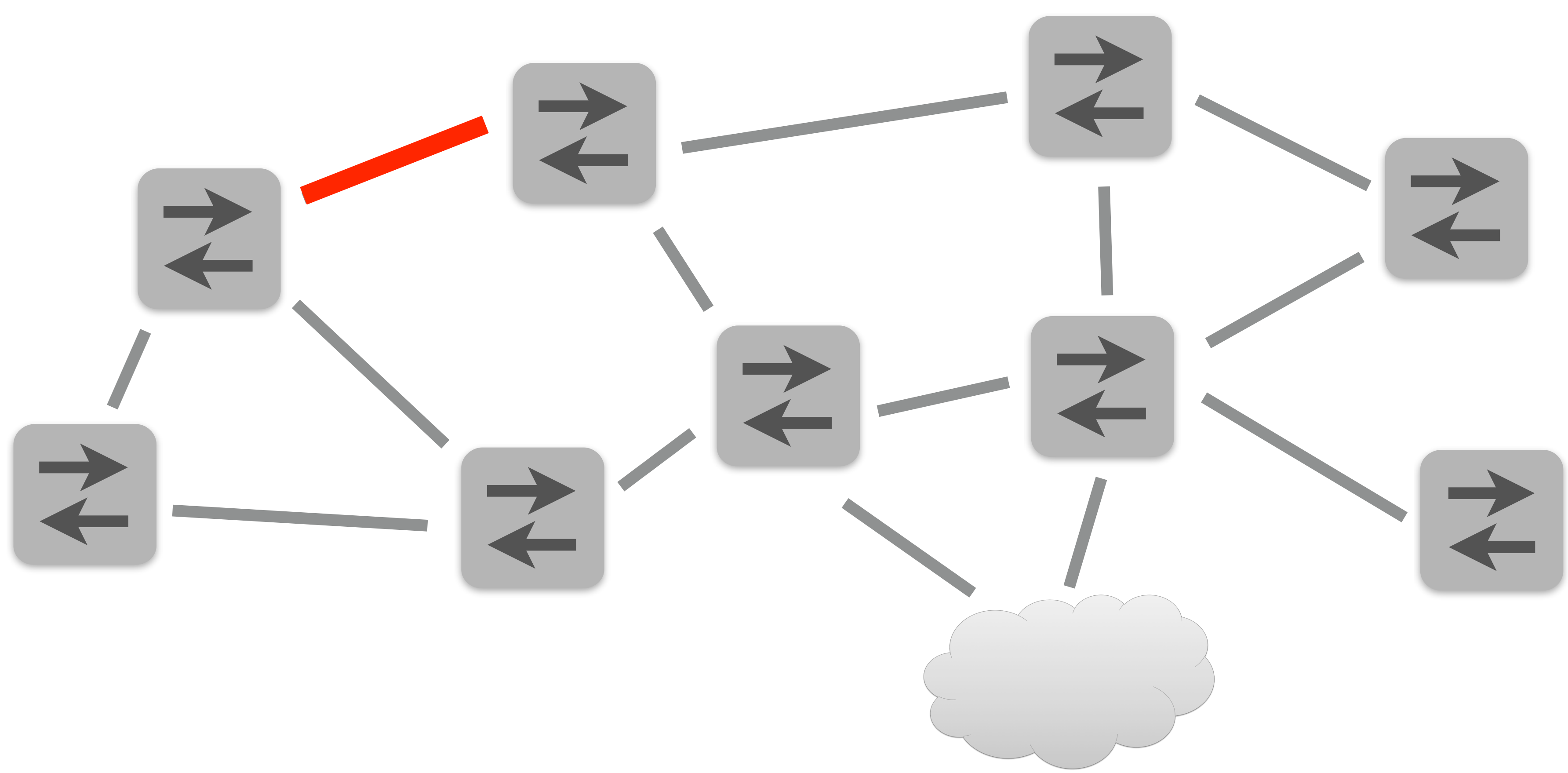
- Packet predicates
- Packet transformations



Network-Wide Programming

What features should an SDN language provide?

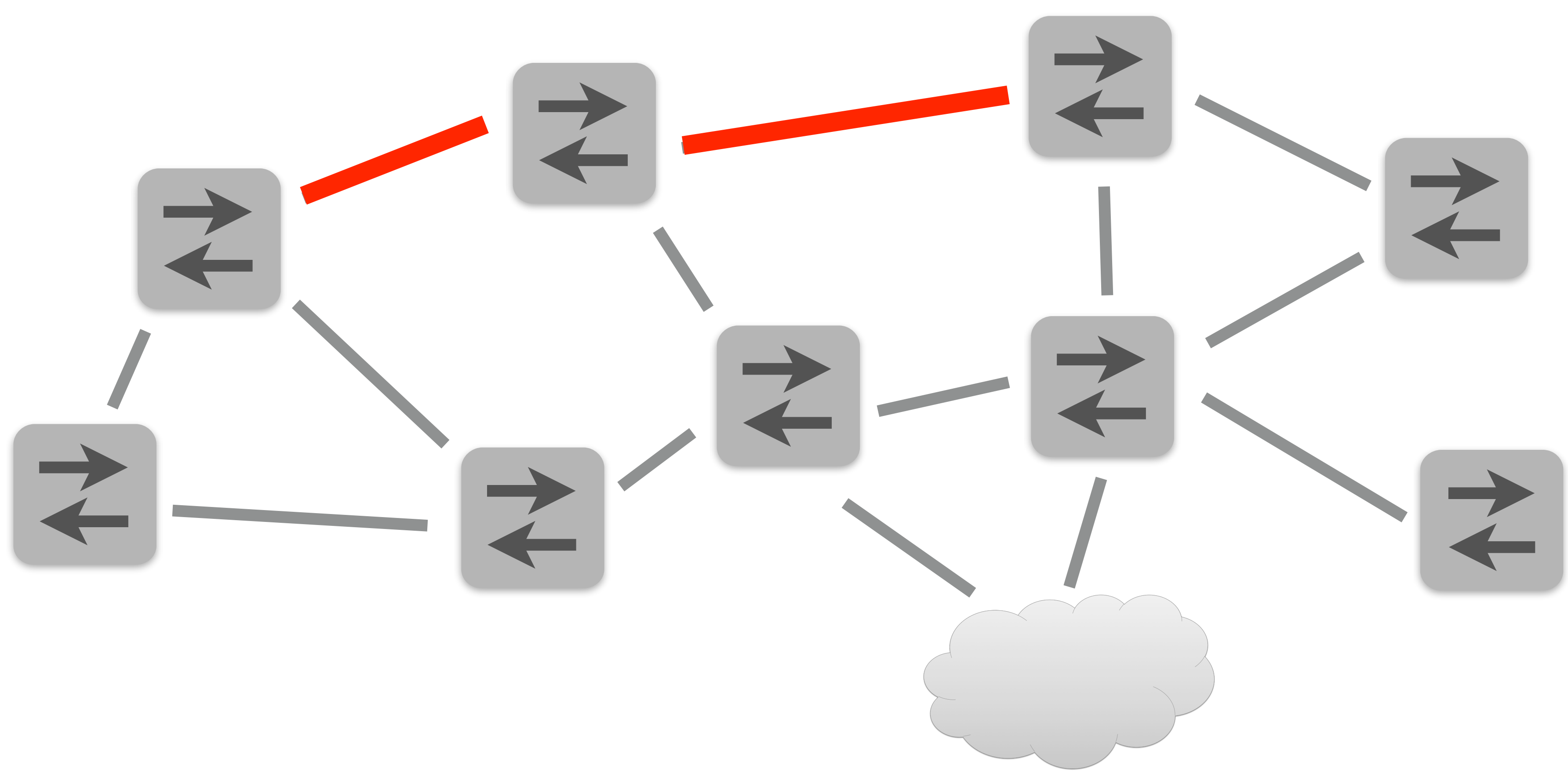
- Packet predicates
- Packet transformations
- Path construction



Network-Wide Programming

What features should an SDN language provide?

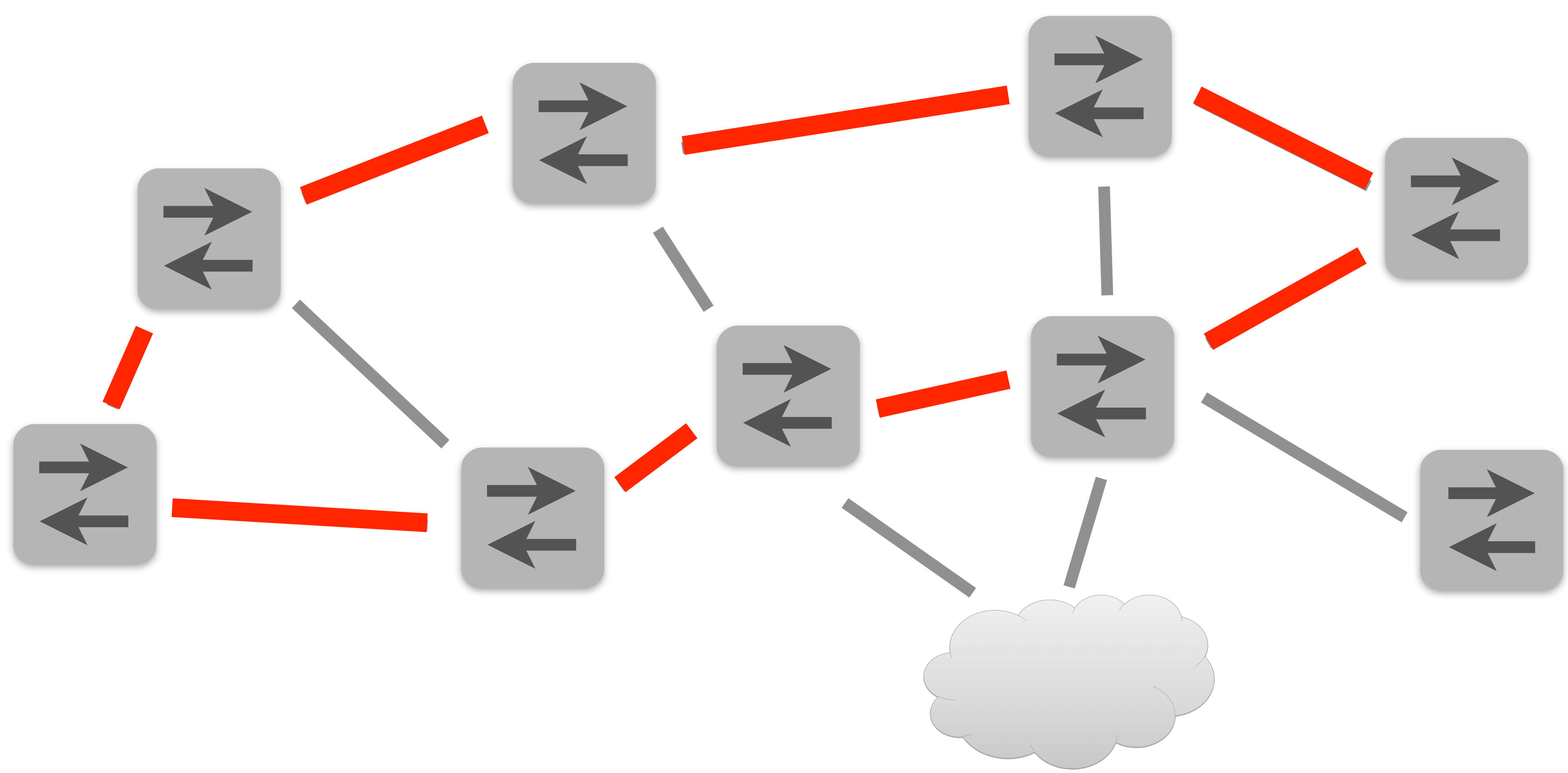
- Packet predicates
- Packet transformations
- Path construction
- Path concatenation



Network-Wide Programming

What features should an SDN language provide?

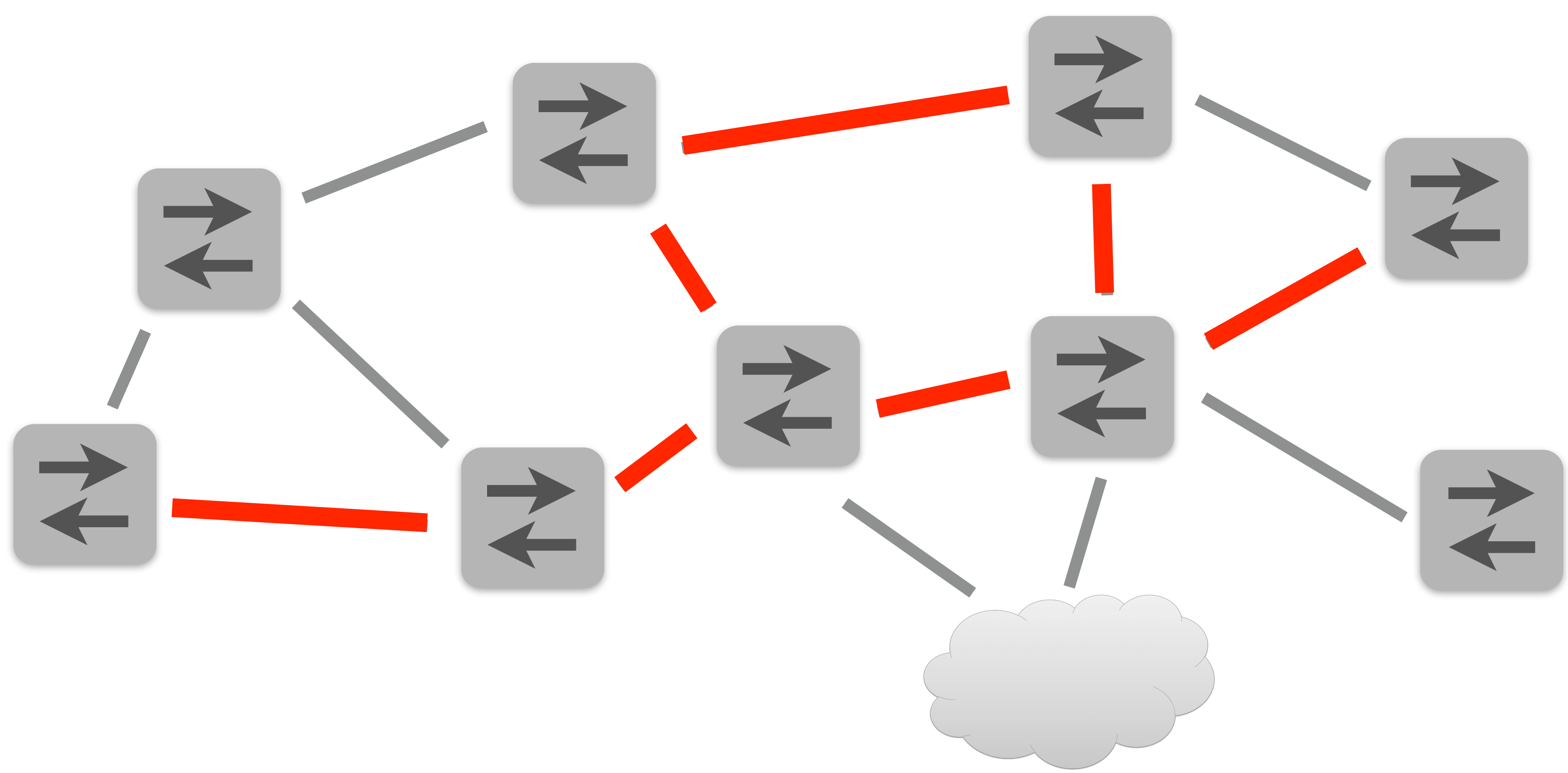
- Packet predicates
- Packet transformations
- Path construction
- Path concatenation
- Path union



Network-Wide Programming

What features should an SDN language provide?

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



NetKAT Language

f ::= **switch** | **port** | **ethSrc** | **ethDst** | ...

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

NetKAT Language

f ::= **switch** | **port** | **ethSrc** | **ethDst** | ...

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

if a **then** p_1 **else** p_2 ≜ (**filter** a**;** p_1) **+** (**filter !**a**;** p_2)

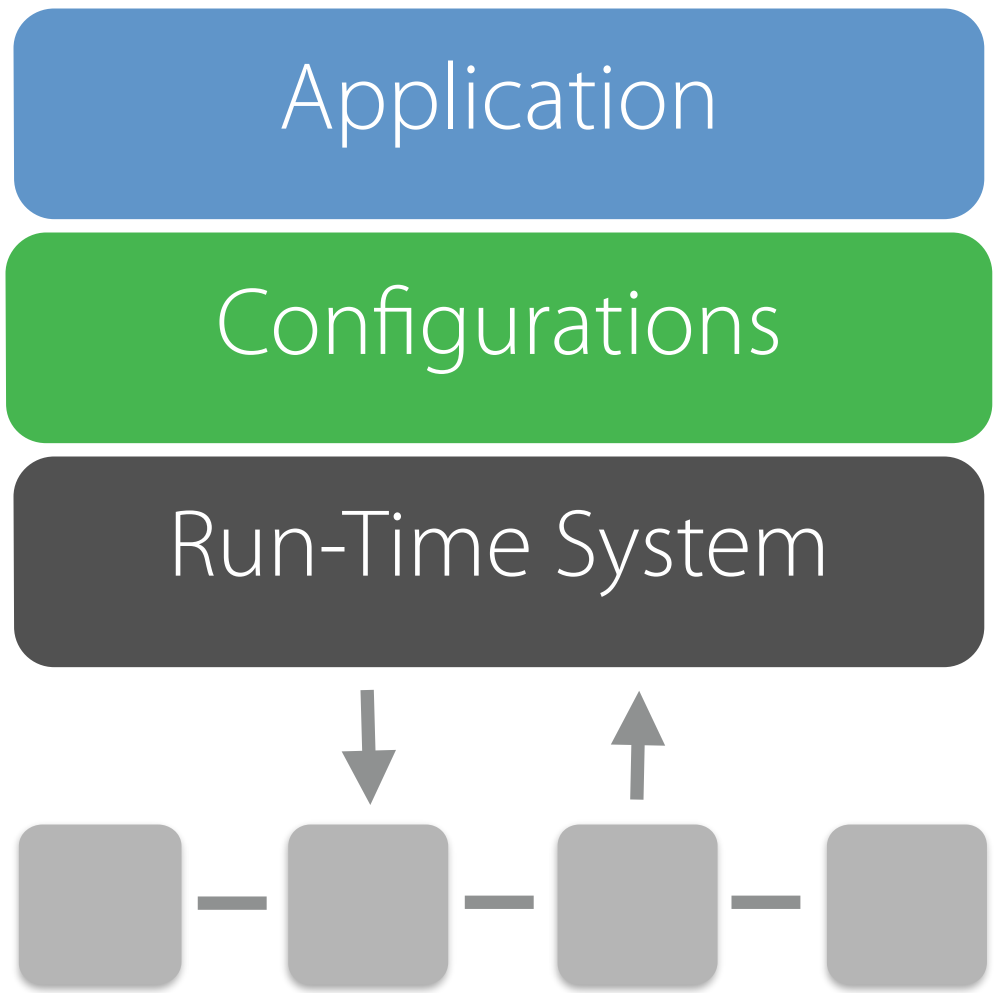
drop ≜ **filter false** **id** ≜ **filter true**

Demo: NetKAT Repeater

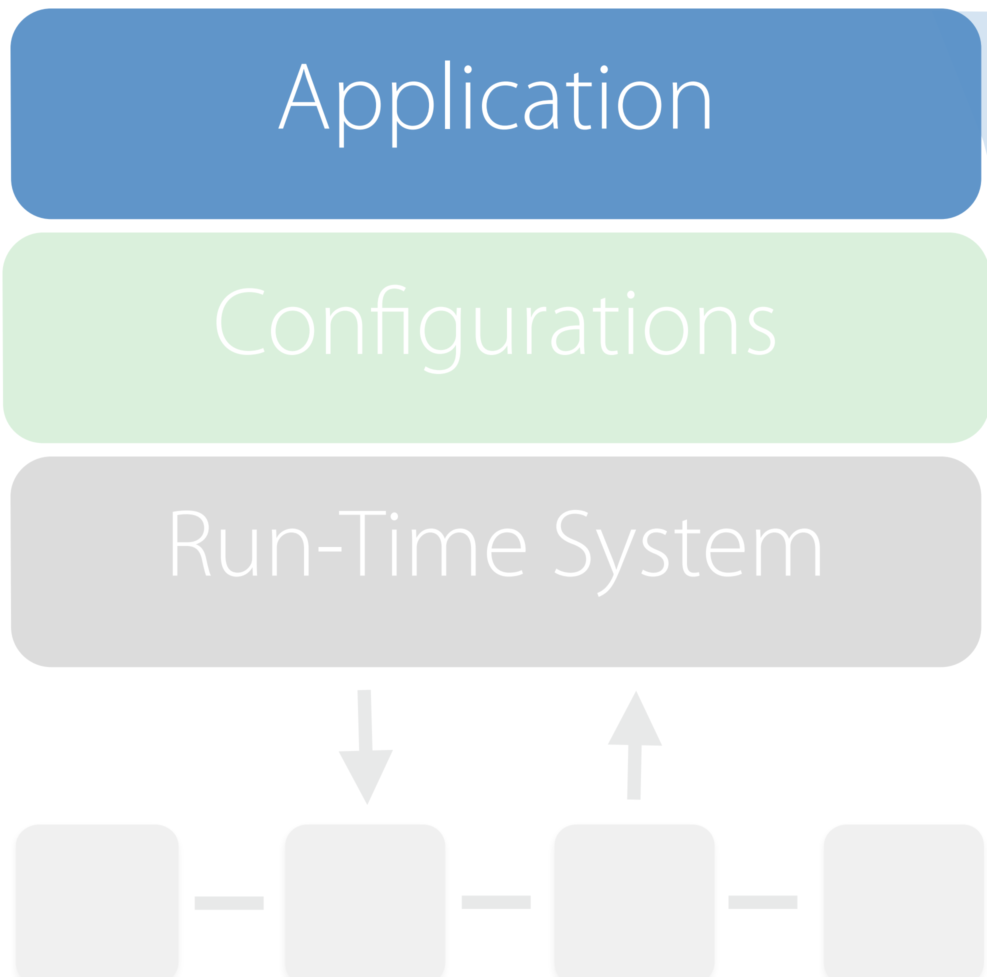
Demo: Ox Firewall

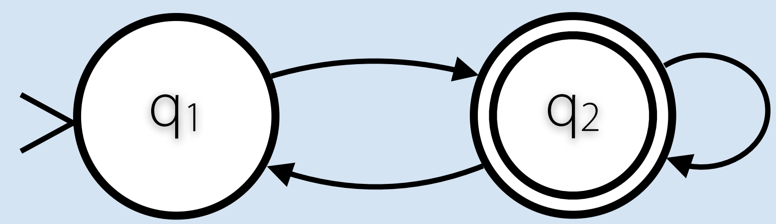
Demo: NetKAT Firewall

Dynamic Applications



Dynamic Applications

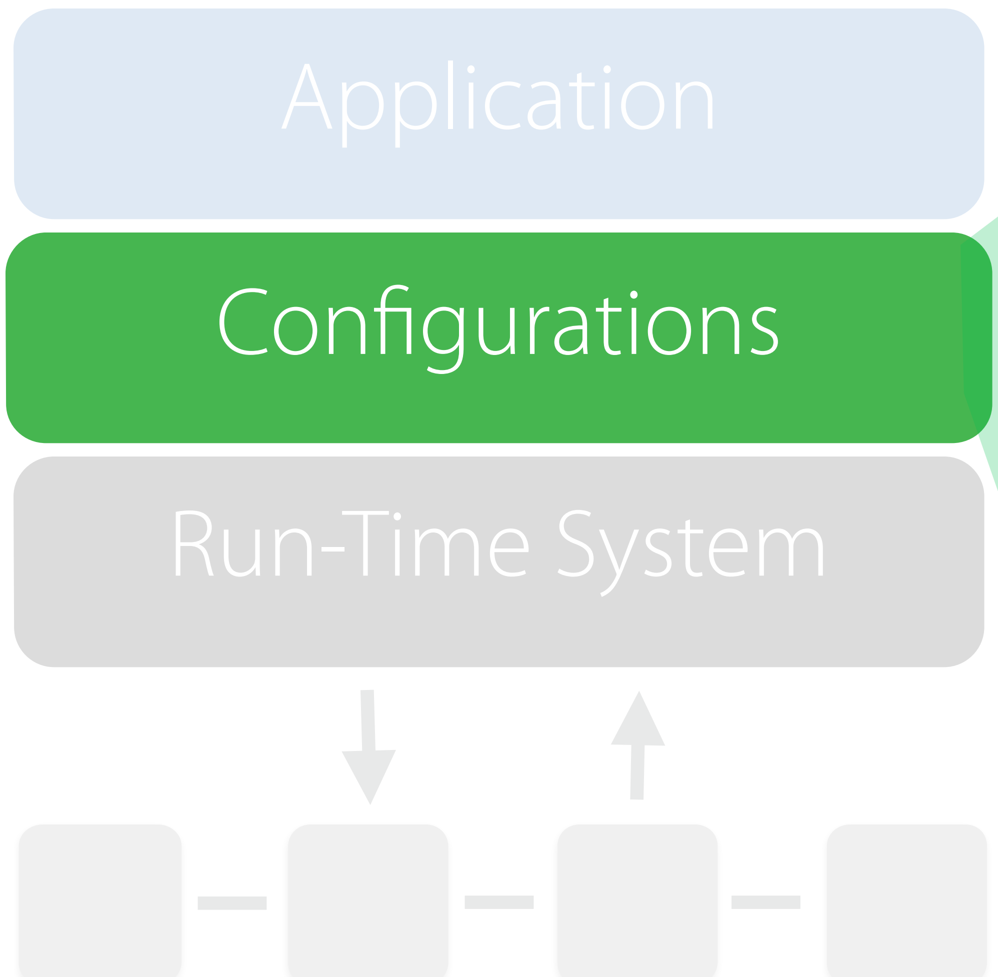




High-level application logic

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

Dynamic Applications

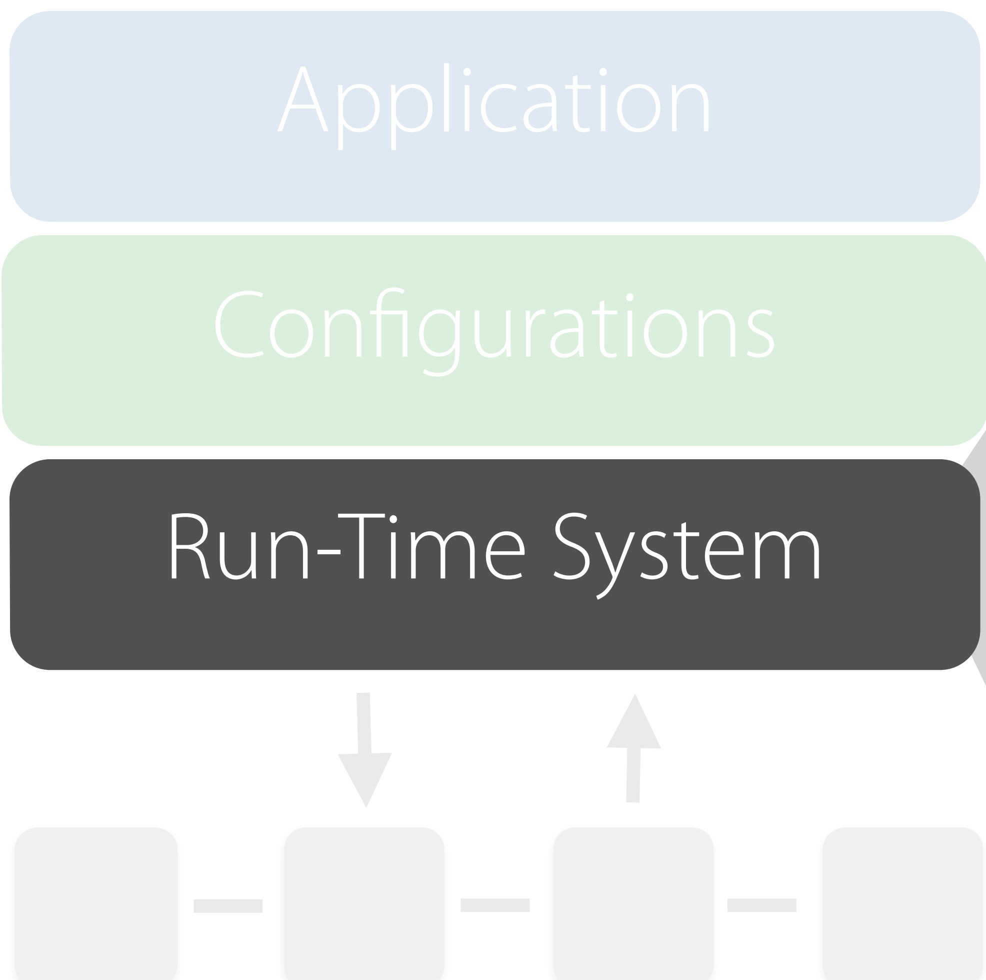


Pattern	Actions
srcip=1.2.3.4, tcpdst = 22	Count, Drop
srcip=1.2.3.4,	Forward 1, Count
srcip=1.2.3.4,	Forward 2, Count
srcip=1.2.3.4	Count
tcpdst = 22	Drop

Network-wide packet-processing function

Expressed in terms of a set of forwarding tables, one per switch in the network

Dynamic Applications

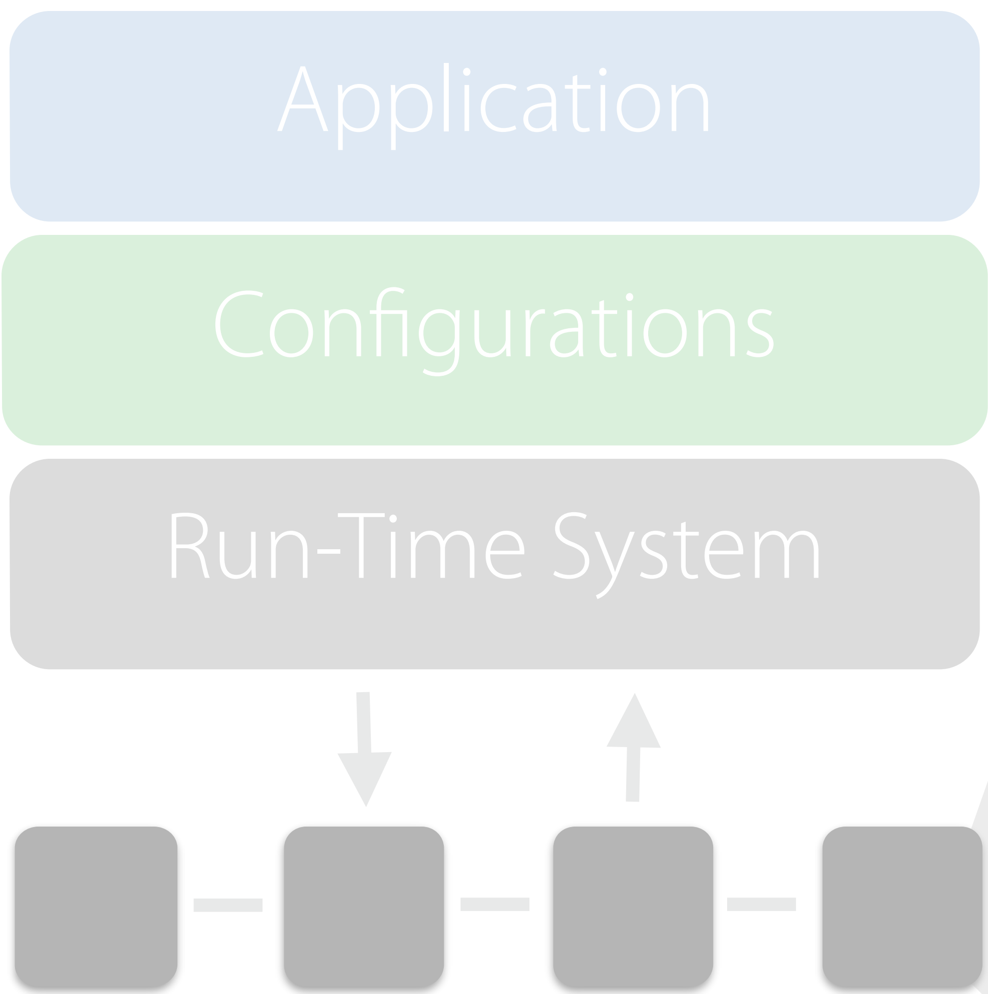



```
let swap_update_for (t : t) sw_id c_id new_table : unit Deferred.t =
   let max_priority = 65535 in
   let old_table = match SwitchMap.find t.edge sw_id with | Some ft -> ft | None -> [] in
   let (new_table, _) = List.fold new_table ~init:([], max_priority)
        ~f:(fun (acc,pri) x -> ((x,pri) :: 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

Dynamic Applications



Forwarding elements that implement packet-processing functionality efficiently in hardware

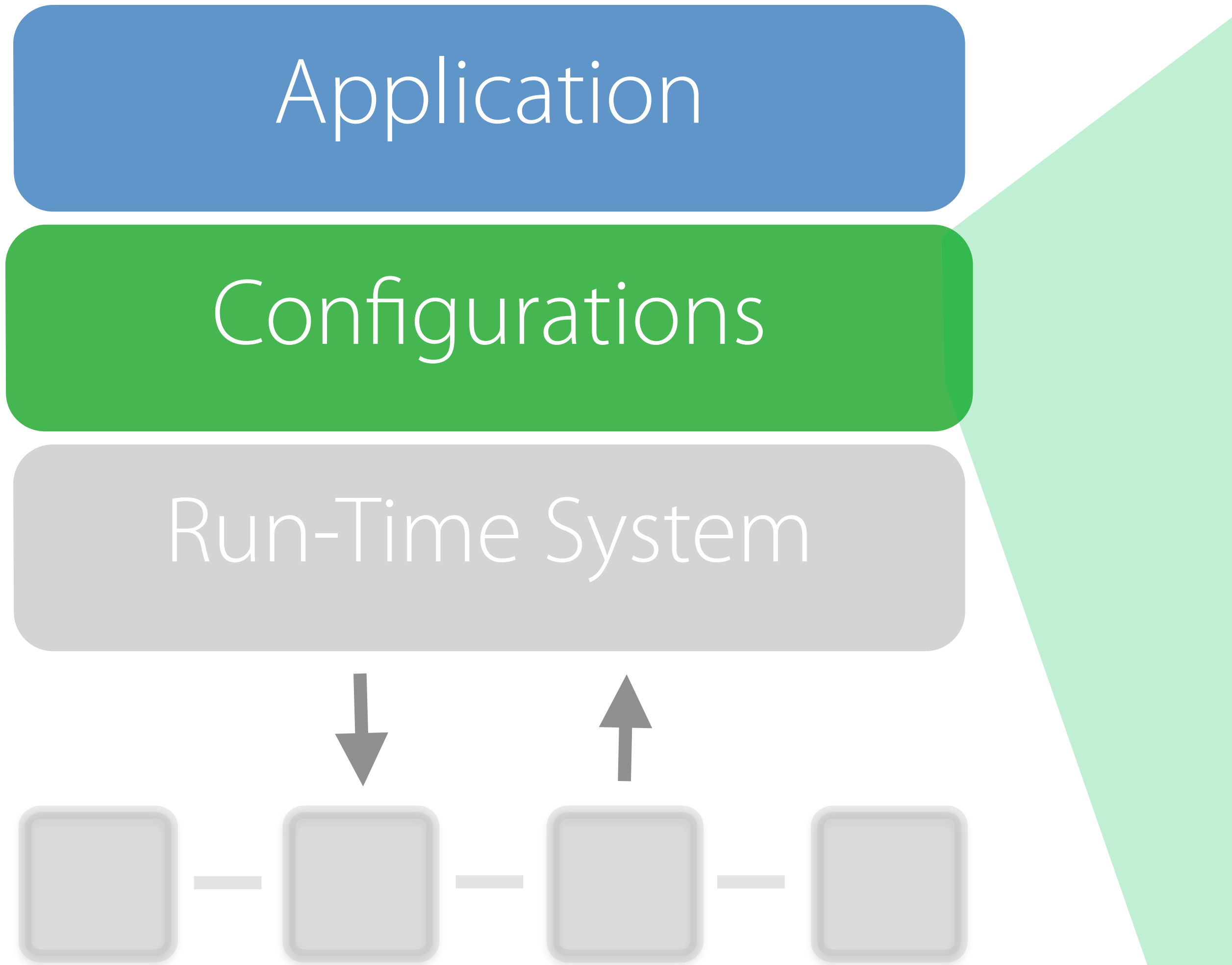
Demo: Ox Learning

Demo: NetKAT Learning

Reasoning in NetKAT

[POPL '14]

Language Model



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

Network-wide packet-processing function

Expressed in terms of a set of forwarding tables, one per switch in the network

Encoding Tables

Forwarding tables can be expressed as NetKAT policies

OpenFlow Normal Form (ONF)

fwd ::= f_1 **:=** n_1**;**…**;** f_k:=n_k **+** fwd
| **drop**

pat ::= f **=** n**;** pat
| **true**

tbl ::= **if** pat **then** fwd **else** tbl
| **drop**

Encoding Tables

Forwarding tables can be expressed as NetKAT policies

OpenFlow Normal Form (ONF)

fwd ::= f_1 **:=** n_1**;**…**;** f_k:=n_k **+** fwd
  | **drop**

pat ::= f **=** n**;** pat
  | **true**

tbl ::= **if** pat **then** fwd **else** tbl
  | **drop**

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

if dstport**=**22 **then drop**
else if srcip**=**10.0.0.1 **then** port **:=** 1
else if true then port **:=** 2
else drop

Encoding Tables

Forwarding tables can be expressed as NetKAT policies

OpenFlow Normal Form (ONF)

fwd ::= f_1 **:=** n_1**;**...**;** f_k:=n_k **+** fwd
  | **drop**

pat ::= f **=** n**;** pat
  | **true**

tbl ::= **if** pat **then** fwd **else** tbl
  | **drop**

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

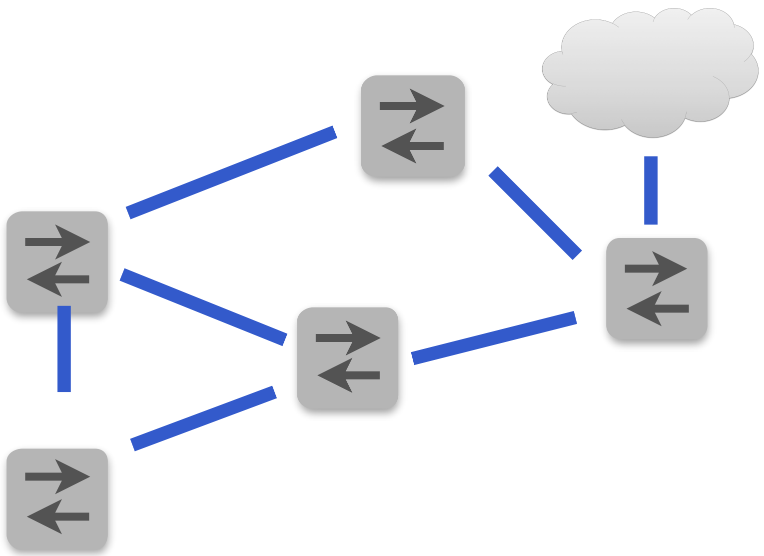
if dstport**=**22 **then drop**
else if srcip**=**10.0.0.1 **then** port **:=** 1
else if true then port **:=** 2
else drop

NetKAT compiler rewrites (local) policies into tables

This encoding also facilitates using NetKAT as the "composition substrate" for other platforms

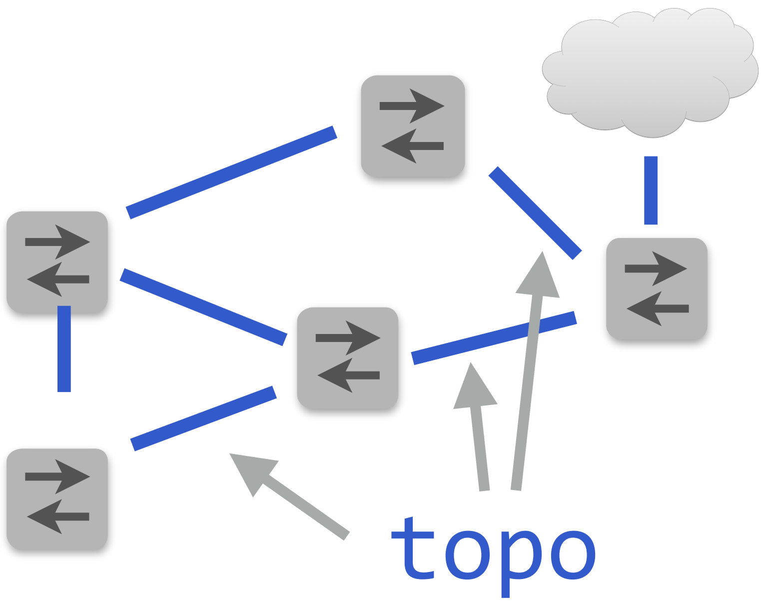
Encoding Topologies

Links can be modeled as simple policies that forward packets from one end to the other, and topologies as unions of links



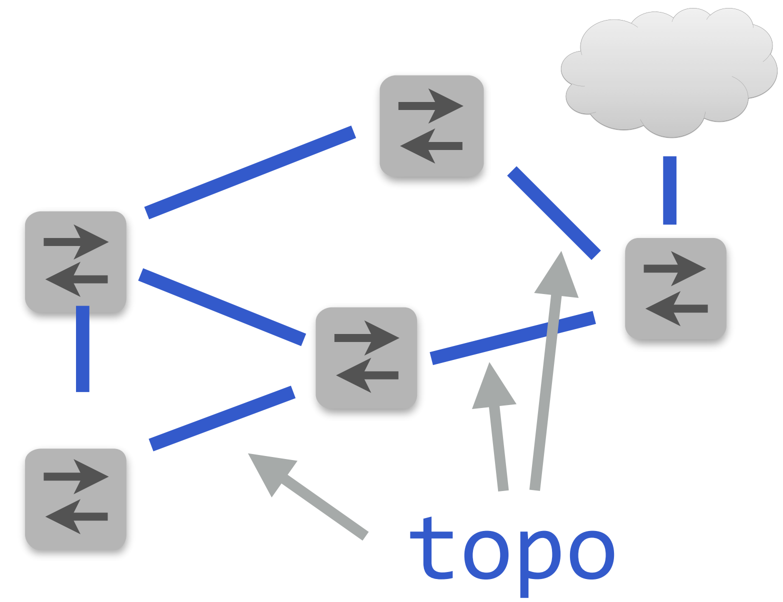
Encoding Topologies

Links can be modeled as simple policies that forward packets from one end to the other, and topologies as unions of links



Encoding Topologies

Links can be modeled as simple policies that forward packets from one end to the other, and topologies as unions of links



Topology Normal Form

lpred ::= **switch=**n**; port=**n

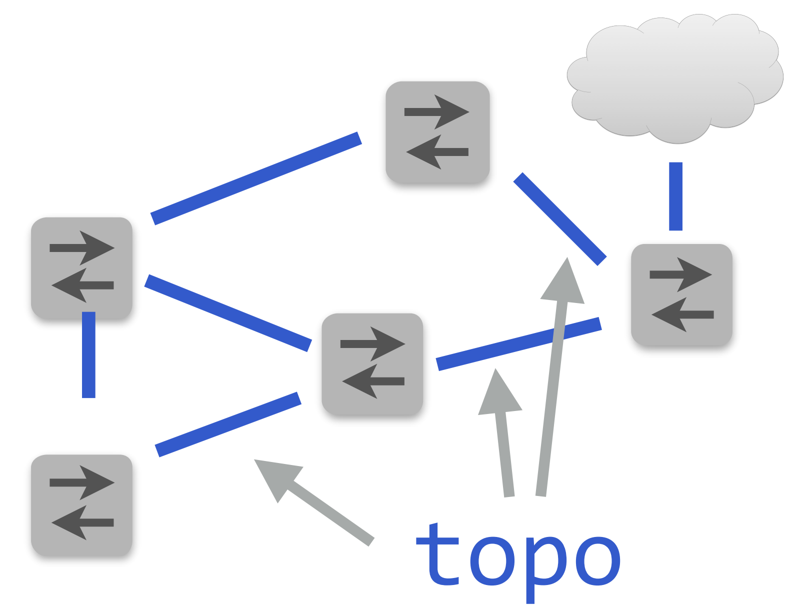
lpol ::= **switch:=**n**; port:=**n

link ::= lpred**;** lpol

topo ::= link **+** topo
| **drop**

Encoding Topologies

Links can be modeled as simple policies that forward packets from one end to the other, and topologies as unions of links



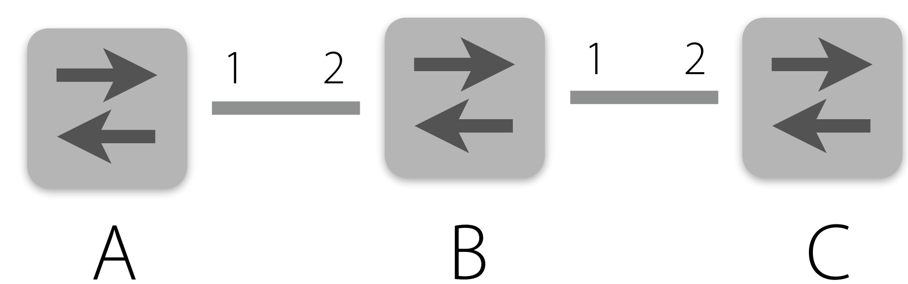
Topology Normal Form

lpred ::= **switch=**n**; port=**n

lpol ::= **switch:=**n**; port:=**n

link ::= lpred**;** lpol

topo ::= link **+** topo
| **drop**



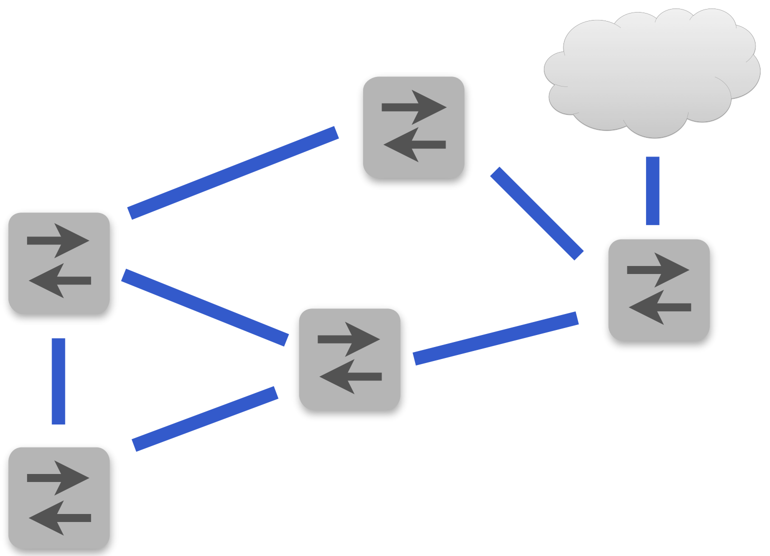
switch=A**; port=**1**; switch:=**B**; port:=**2 **+**
switch=B**; port=**2**; switch:=**A**; port:=**1 **+**
switch=B**; port=**1**; switch:=**C**; port:=**2 **+**
switch=C**; port=**2**; switch:=**B**; port:=**1 **+**
drop

Encoding Networks

Putting all these pieces together,
an entire network can be modeled
by interleaving policy and
topology processing steps

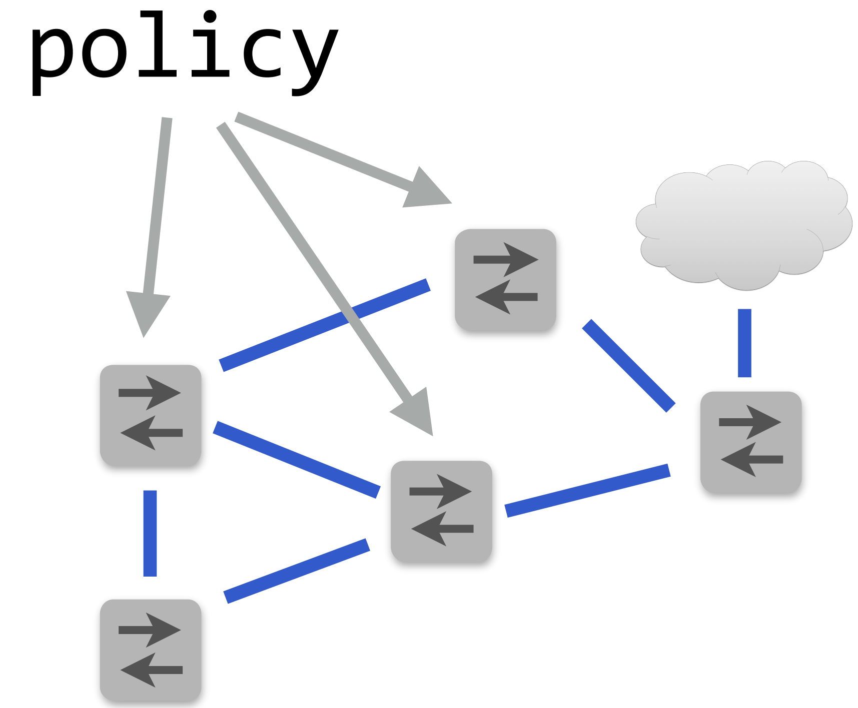
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps



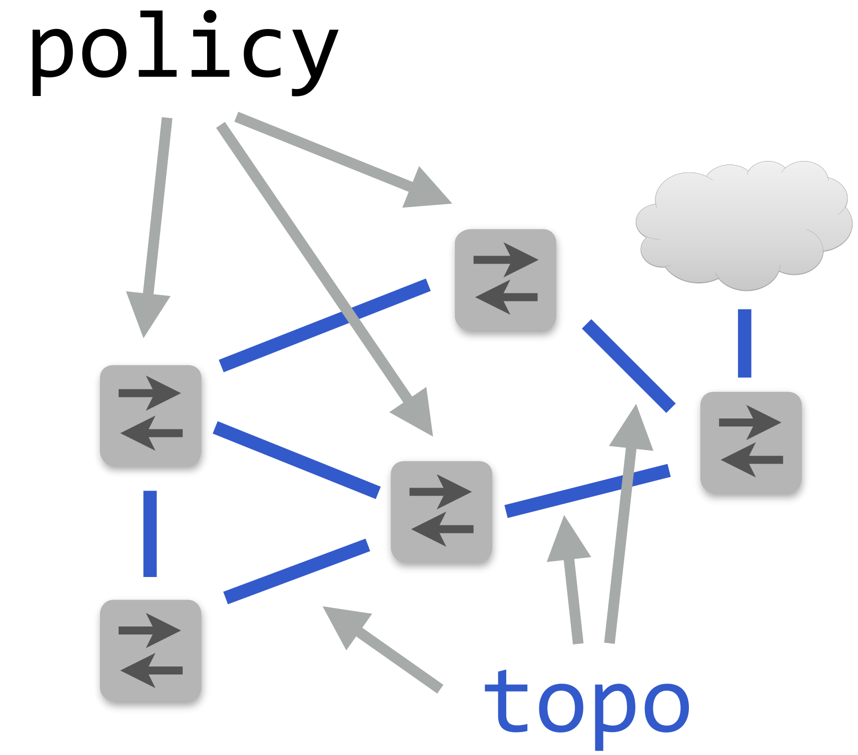
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps



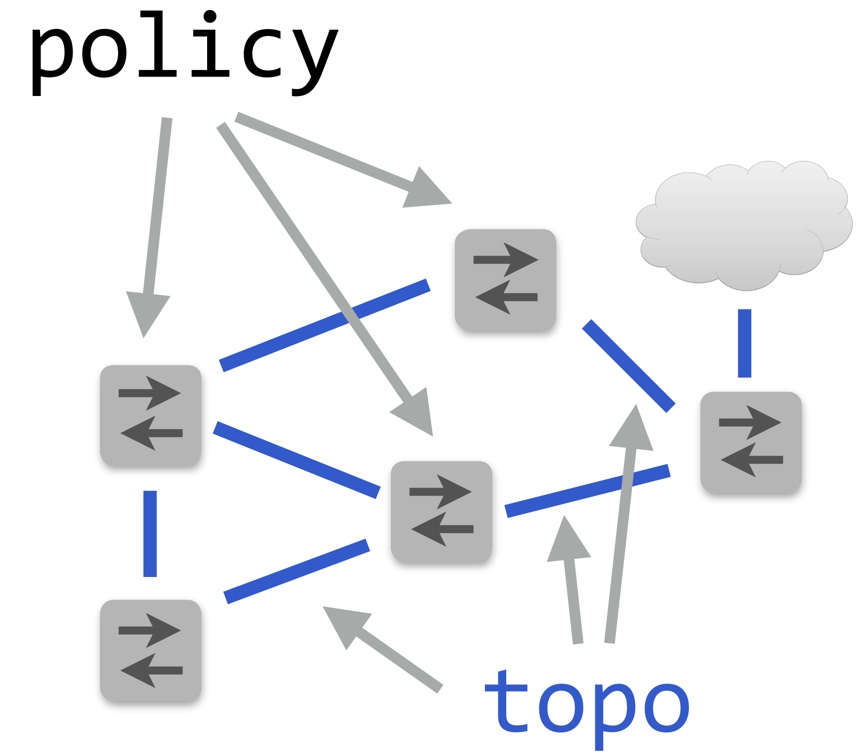
Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps



Encoding Networks

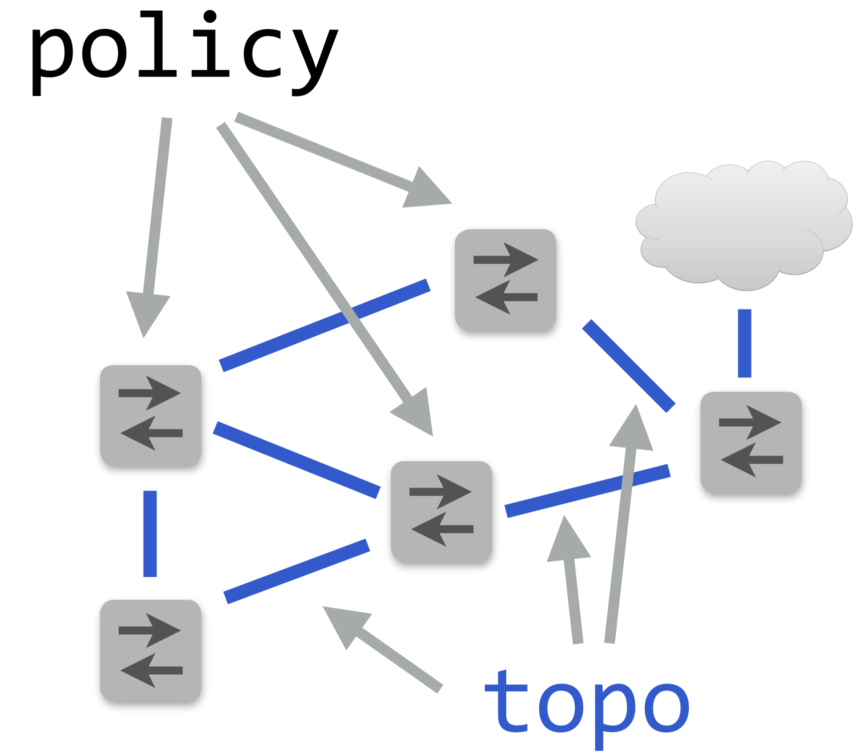
Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps



id

Encoding Networks

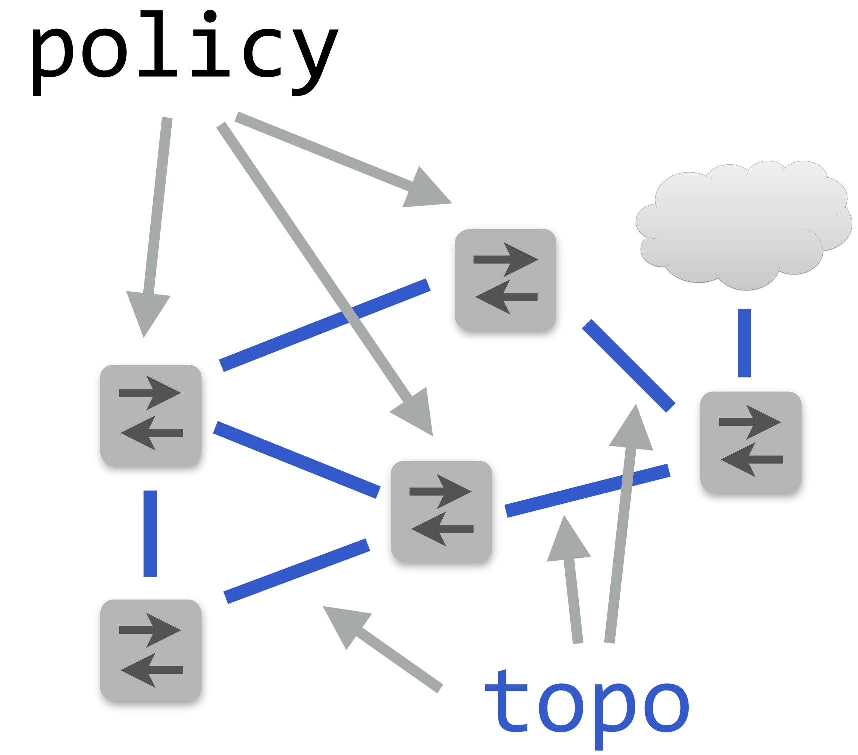
Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps




```
id
+
(policy; topo)
```

Encoding Networks

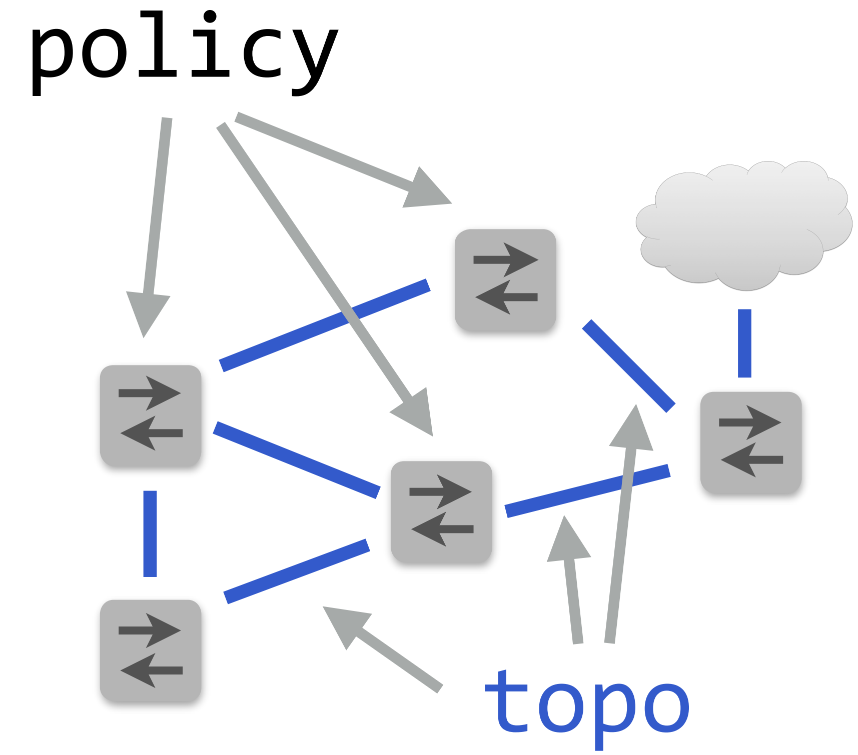
Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps




```
id
+
(policy; topo)
+
(policy; topo; policy; topo)
```

Encoding Networks

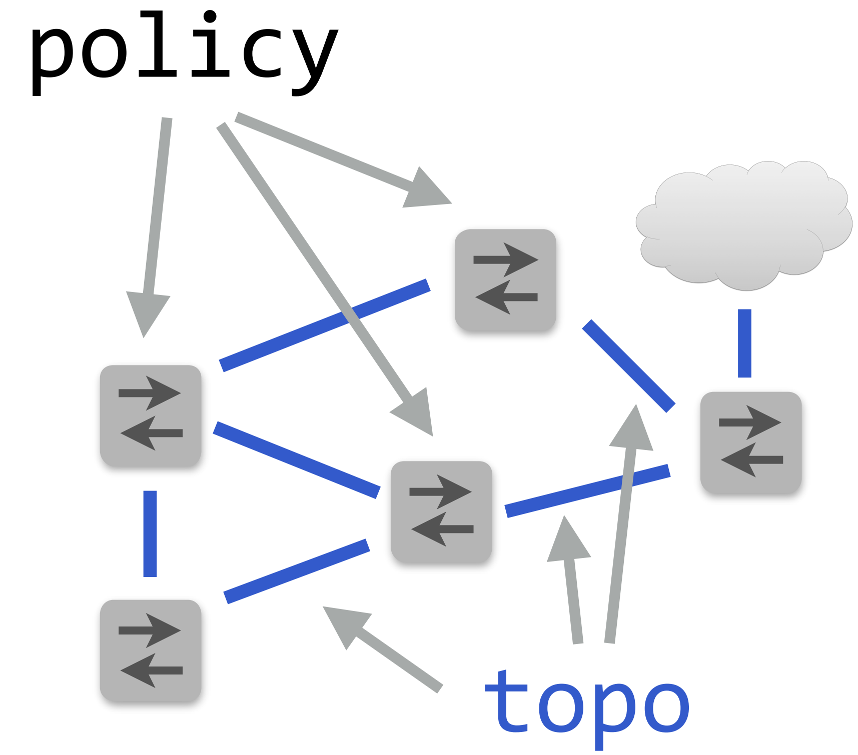
Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps




```
id
+
(policy; topo)
+
(policy; topo; policy; topo)
+
(policy; topo; policy; topo; policy; topo)
```

Encoding Networks

Putting all these pieces together, an entire network can be modeled by interleaving policy and topology processing steps




```
id
+
(policy; topo)
+
(policy; topo; policy; topo)
+
(policy; topo; policy; topo; policy; topo)
…
+
(policy; topo)*
```

Semantic Foundations

Unlike previous network programming languages, the design of NetKAT is not an accident!

Its foundations rest upon canonical mathematical structure:

- Regular operators (**+**, **;** , *****) encode paths through topology
- Boolean operators (**+**, **;** , **!**) encode forwarding tables

Such structures are called *Kleene Algebras with Tests (KAT)* [Kozen '96]

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

Many reasoning tasks can be reduced to checking equivalences between terms

NetKAT Proof System

Kleene Algebra Axioms

$p + (q + r) \sim (p + q) + r$

$p + q \sim q + p$

$p + \textbf{drop} \sim p$

$p + p \sim p$

$p; (q; r) \sim (p; q); r$

$p; (q + r) \sim p;\ q + p; r$

$(p + q);\ r \sim p;\ r + q;\ r$

$\textbf{id}; p \sim p$

$p \sim p; \textbf{id}$

$\textbf{drop}; p \sim \textbf{drop}$

$p; \textbf{drop} \sim \textbf{drop}$

$\textbf{id} + p; p^* \sim p^*$

$\textbf{id} + p^*; p \sim p^*$

$p + q; r + r \sim r \Rightarrow p^*; q + r \sim r$

$p + q; r + q \sim q \Rightarrow p; r^* + q \sim q$

Boolean Algebra Axioms

$a \,||\, (b \,\&\&\, c) \sim (a \,||\, b) \,\&\&\, (a \,||\, c)$

$a \,||\, \textbf{true} \sim \textbf{true}$

$a \,||\, !\, a \sim \textbf{true}$

$a \,\&\&\, b \sim b \,\&\&\, a$

$a \,\&\&\, !a \sim \textbf{false}$

$a \,\&\&\, a \sim a$

Packet Axioms

$f := n; f' := n' \sim f' := n'; f := n \qquad \text{if } f \neq f'$

$f := n; f' = n' \sim f' = n'; f := n \qquad \text{if } f \neq f'$

$f := n; f = n \sim f := n$

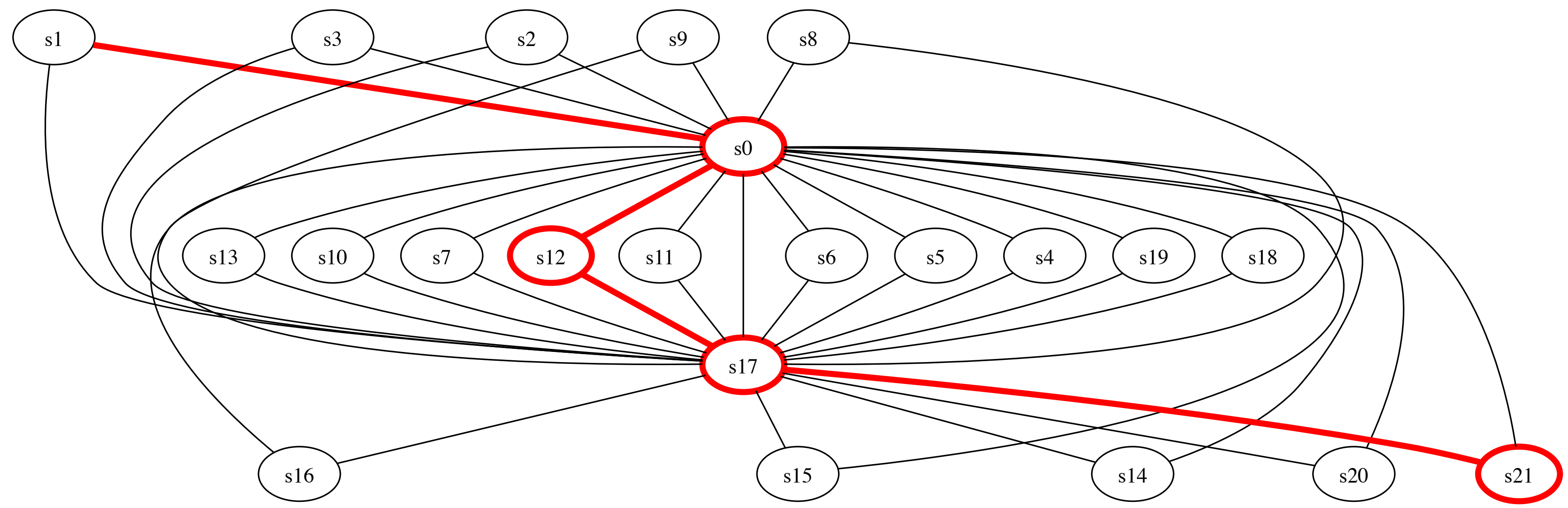
$f = n; f := n \sim f = n$

$f := n; f := n' \sim f := n'$

$f = n; f = n' \sim \textbf{drop} \qquad \text{if } n \neq n'$

$\textbf{dup}; f = n \sim f = n; \textbf{dup}$

Network-Wide Reachability



Given:

- Ingress predicate: switch = s_1
- Egress predicate: switch = s_{21}
- Topology: t
- Switch program: p

Check:

- switch = s_1**;** switch := s_{21} + (p**;** t)***** ~ (p**;** t)*****
- switch=s_1**;** (p**;** t)*****;** switch = s_{21} ~ **drop**

Metatheory

Soundness: If $\vdash p \sim q$, then $[\![p]\!] = [\![q]\!]$

Completeness: If $[\![p]\!] = [\![q]\!]$, then $\vdash p \sim q$

Metatheory

Soundness: If $\vdash p \sim q$, then $[\![p]\!] = [\![q]\!]$

Completeness: If $[\![p]\!] = [\![q]\!]$, then $\vdash p \sim q$

Established previously for KAT [Kozen & Smith '96]…

… but NetKAT's packet histories add extra structure

Metatheory

Soundness: If $\vdash p \sim q$, then $[\![p]\!] = [\![q]\!]$

Completeness: If $[\![p]\!] = [\![q]\!]$, then $\vdash p \sim q$

Established previously for KAT [Kozen & Smith '96]…
… but NetKAT's packet histories add extra structure

Idea: develop an alternate semantics based on a language model, and leverage completeness of Kleene Algebra over regular sets [Kozen '94]

Proof outline:

- Reduced NetKAT
- Regular interpretation
- Normal form

Completeness Proof

p and **q** such that $[\![\mathbf{p}]\!] = [\![\mathbf{q}]\!]$

Completeness Proof

$\mathbf{p}$ and $\mathbf{q}$ such that $[\![\mathbf{p}]\!] = [\![\mathbf{q}]\!]$

$\downarrow$ Reduce and Normalize

$\vdash \mathbf{p} \equiv \hat{p}$ and $\vdash \mathbf{q} \equiv \hat{q}$

$\downarrow$ Soundness

$[\![\hat{p}]\!] = [\![\hat{q}]\!]$

$\downarrow$ Language Model

$G(\hat{p}) = G(\hat{q})$

$\downarrow$ Normal Forms

$R(\hat{p}) = R(\hat{q})$

$\downarrow$ Kleene Algebra Completeness

$\vdash \hat{p} \equiv \hat{q}$

$\downarrow$ Transitivity

$\vdash \mathbf{p} \equiv \mathbf{q}$

NetKAT Automata

Can construct an automaton from a NetKAT program by generalizing the Brzozowski derivative

NetKAT Automata

Can construct an automaton from a NetKAT program by generalizing the Brzozowski derivative

Continuation Map:

$D_{\alpha\beta}(\mathbf{f = n}) = 0$

$D_{\alpha\beta}(\mathbf{dup}) = \alpha \bullet [\alpha{=}\beta]$

$D_{\alpha\beta}(\mathbf{f{:=}n}) = 0$

$D_{\alpha\beta}(p \mathbin{\mathbf{+}} q) = D_{\alpha\beta}(p) \mathbin{\mathbf{+}} D_{\alpha\beta}(q)$

$D_{\alpha\beta}(p \bullet q) = D_{\alpha\beta}(p) \bullet q \mathbin{\mathbf{+}} \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet D_{\gamma\beta}(q)$

$D_{\alpha\beta}(p^{\mathbf{*}}) = D_{\alpha\beta}(p) \bullet p^{\mathbf{*}} \mathbin{\mathbf{+}} \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet D_{\gamma\beta}(p^{\mathbf{*}})$

Observation Map:

$E_{\alpha\beta}(\mathbf{f = n}) = [\alpha{=}\beta \leq \mathbf{f{=}n}]$

$E_{\alpha\beta}(\mathbf{dup}) = \alpha \bullet [\alpha{=}\beta]$

$E_{\alpha\beta}(\mathbf{f{:=}n}) = [\mathbf{f{:=}n} = p_\beta]$

$E_{\alpha\beta}(p + q) = E_{\alpha\beta}(p) \mathbin{\mathbf{+}} E_{\alpha\beta}(q)$

$E_{\alpha\beta}(p \bullet q) = \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet E_{\gamma\beta}(q)$

$E_{\alpha\beta}(p^{\mathbf{*}}) = [\alpha{=}\beta] \mathbin{\mathbf{+}} \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet E_{\gamma\beta}(p^{\mathbf{*}})$

NetKAT Automata

Can construct an automaton from a NetKAT program by generalizing the Brzozowski derivative

Continuation Map:

$D_{\alpha\beta}(\mathbf{f = n}) = 0$

$D_{\alpha\beta}(\mathbf{dup}) = \alpha \bullet [\alpha=\beta]$

$D_{\alpha\beta}(\mathbf{f{:=}n}) = 0$

$D_{\alpha\beta}(p + q) = D_{\alpha\beta}(p) + D_{\alpha\beta}(q)$

$D_{\alpha\beta}(p \bullet q) = D_{\alpha\beta}(p) \bullet q + \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet D_{\gamma\beta}(q)$

$D_{\alpha\beta}(p^*) = D_{\alpha\beta}(p) \bullet p^* + \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet D_{\gamma\beta}(p^*)$

Observation Map:

$E_{\alpha\beta}(\mathbf{f = n}) = [\alpha=\beta \leq \mathbf{f{=}n}]$

$E_{\alpha\beta}(\mathbf{dup}) = \alpha \bullet [\alpha=\beta]$

$E_{\alpha\beta}(\mathbf{f{:=}n}) = [\mathbf{f{:=}n} = p_\beta]$

$E_{\alpha\beta}(p + q) = E_{\alpha\beta}(p) + E_{\alpha\beta}(q)$

$E_{\alpha\beta}(p \bullet q) = \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet E_{\gamma\beta}(q)$

$E_{\alpha\beta}(p^*) = [\alpha=\beta] + \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet E_{\gamma\beta}(p^*)$

Intuitively, these automata recognize the (guarded) strings denoted in NetKAT's language model

NetKAT Automata

Can construct an automaton from a NetKAT program by generalizing the Brzozowski derivative

Continuation Map:

$D_{\alpha\beta}(\mathbf{f = n}) = 0$

$D_{\alpha\beta}(\mathbf{dup}) = \alpha \bullet [\alpha{=}\beta]$

$D_{\alpha\beta}(\mathbf{f{:=}n}) = 0$

$D_{\alpha\beta}(p + q) = D_{\alpha\beta}(p) + D_{\alpha\beta}(q)$

$D_{\alpha\beta}(p \bullet q) = D_{\alpha\beta}(p) \bullet q + \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet D_{\gamma\beta}(q)$

$D_{\alpha\beta}(p^*) = D_{\alpha\beta}(p) \bullet p^* + \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet D_{\gamma\beta}(p^*)$

Observation Map:

$E_{\alpha\beta}(\mathbf{f = n}) = [\alpha{=}\beta \leq \mathbf{f{=}n}]$

$E_{\alpha\beta}(\mathbf{dup}) = \alpha \bullet [\alpha{=}\beta]$

$E_{\alpha\beta}(\mathbf{f{:=}n}) = [\mathbf{f{:=}n} = p_\beta]$

$E_{\alpha\beta}(p + q) = E_{\alpha\beta}(p) + E_{\alpha\beta}(q)$

$E_{\alpha\beta}(p \bullet q) = \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet E_{\gamma\beta}(q)$

$E_{\alpha\beta}(p^*) = [\alpha{=}\beta] + \Sigma_\gamma\, E_{\alpha\gamma}(p) \bullet E_{\gamma\beta}(p^*)$

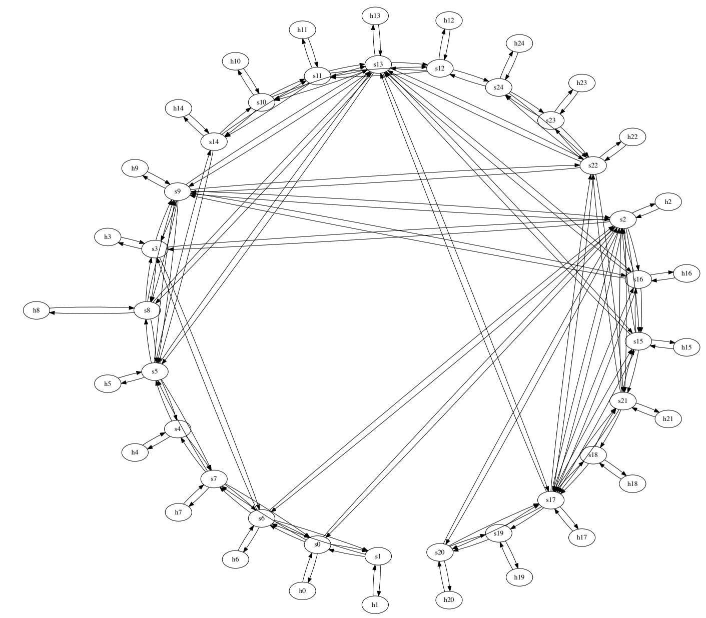
Intuitively, these automata recognize the (guarded) strings denoted in NetKAT's language model

Automata can be represented compactly using sparse matrices, yielding an efficient decision procedure based on bisimulation

Experiments

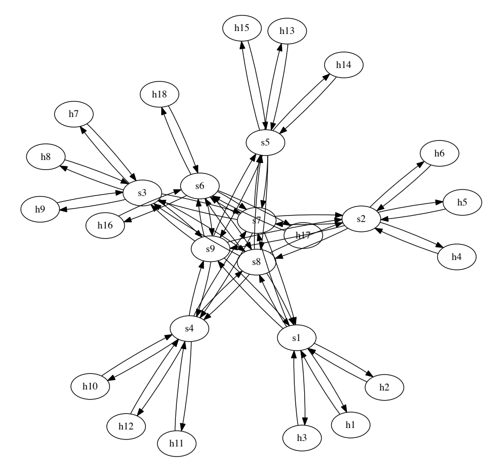
Networks:

- Topology Zoo
- FatTree
- Stanford Backbone



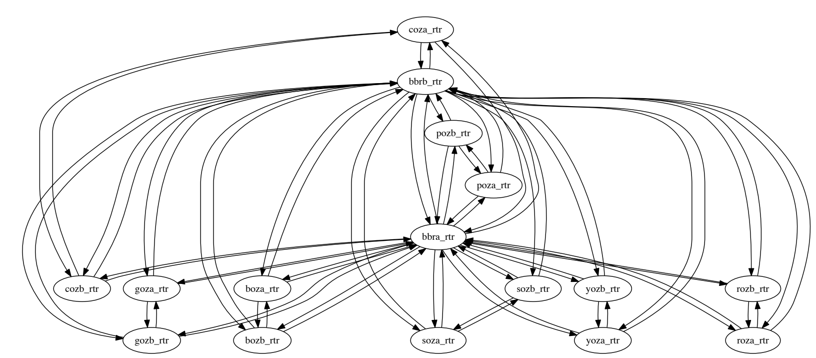
Programs:

- Shortest Paths
- Stanford Policy



Queries:

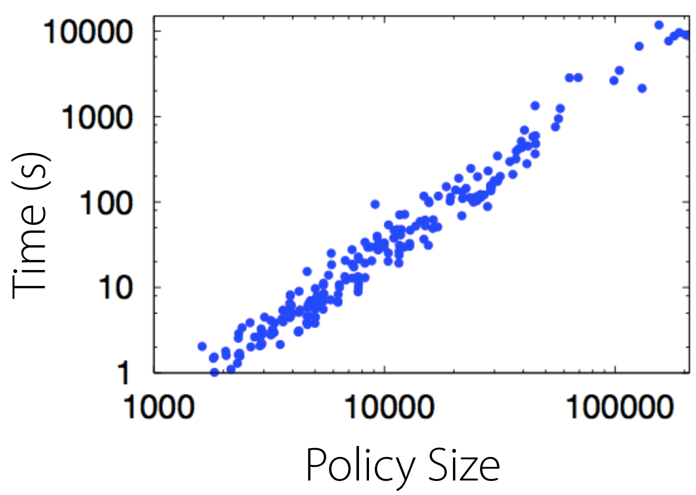
- Reachability
- All-Pairs Connectivity
- Loop Freedom
- Translation Validation



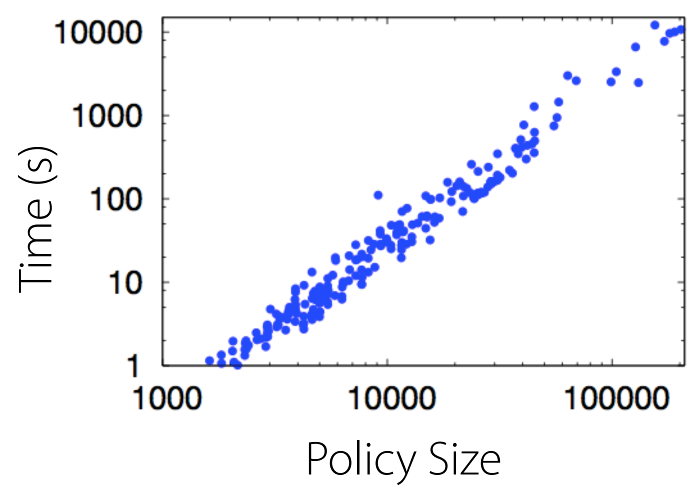
Results

Topology Zoo

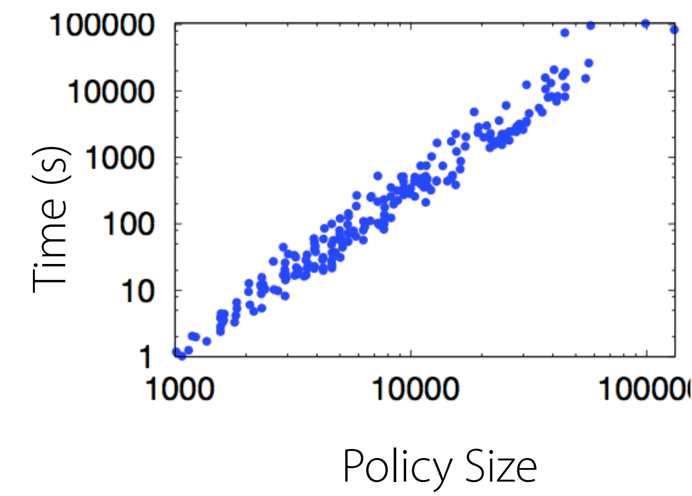
Connectivity



Loop Freedom

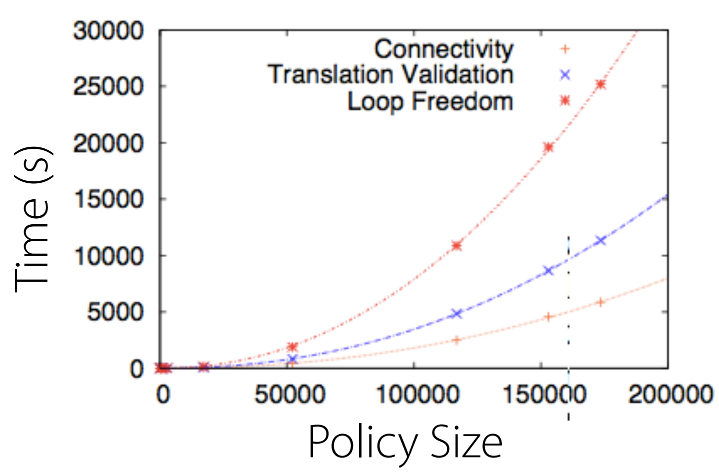


Translation Validation

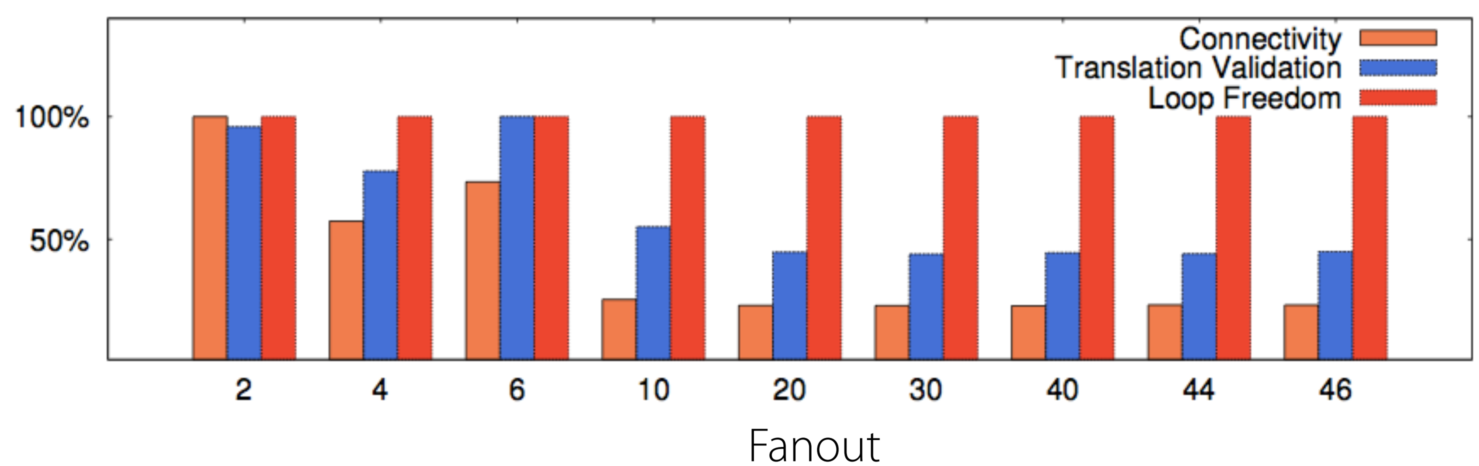


FatTree

Scalability



Relative Performance



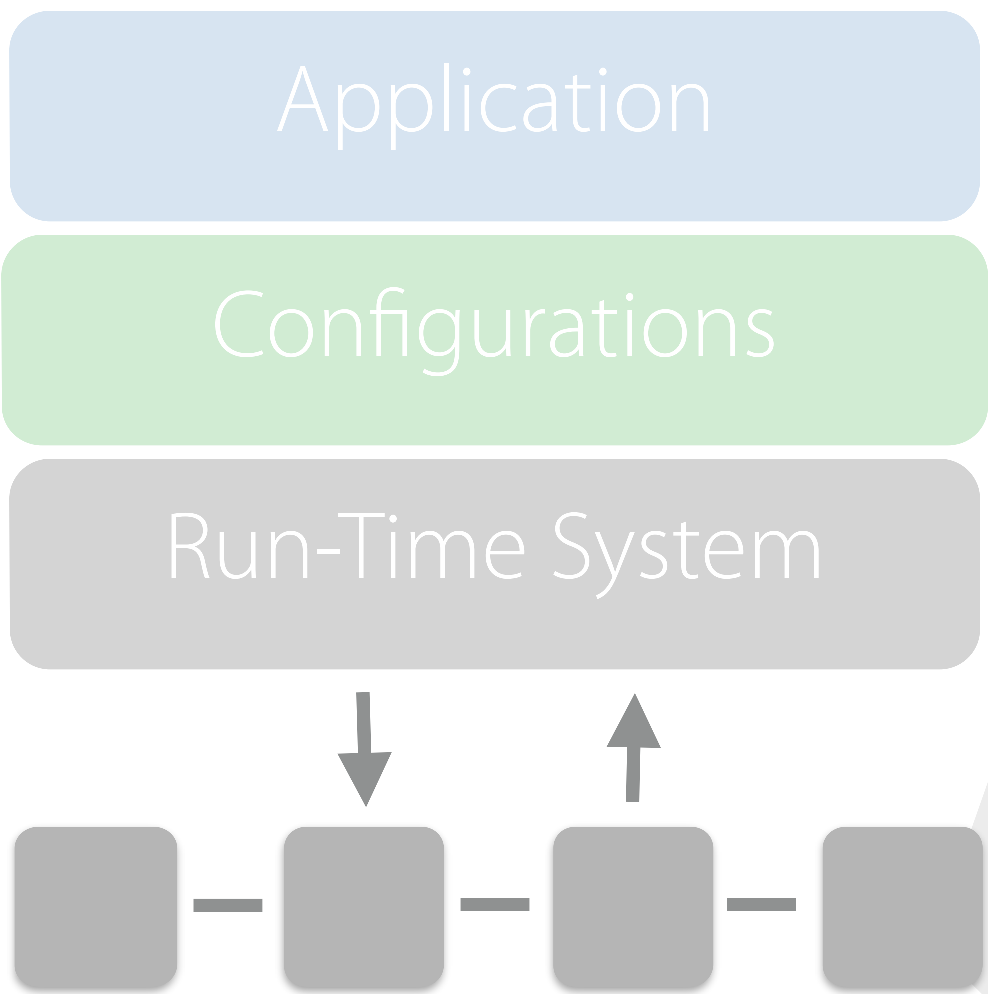
Stanford Backbone

Basic reachability in 0.67s (vs 13s for HSA)

Coq Implementation

[PLDI '13]

Machine Model

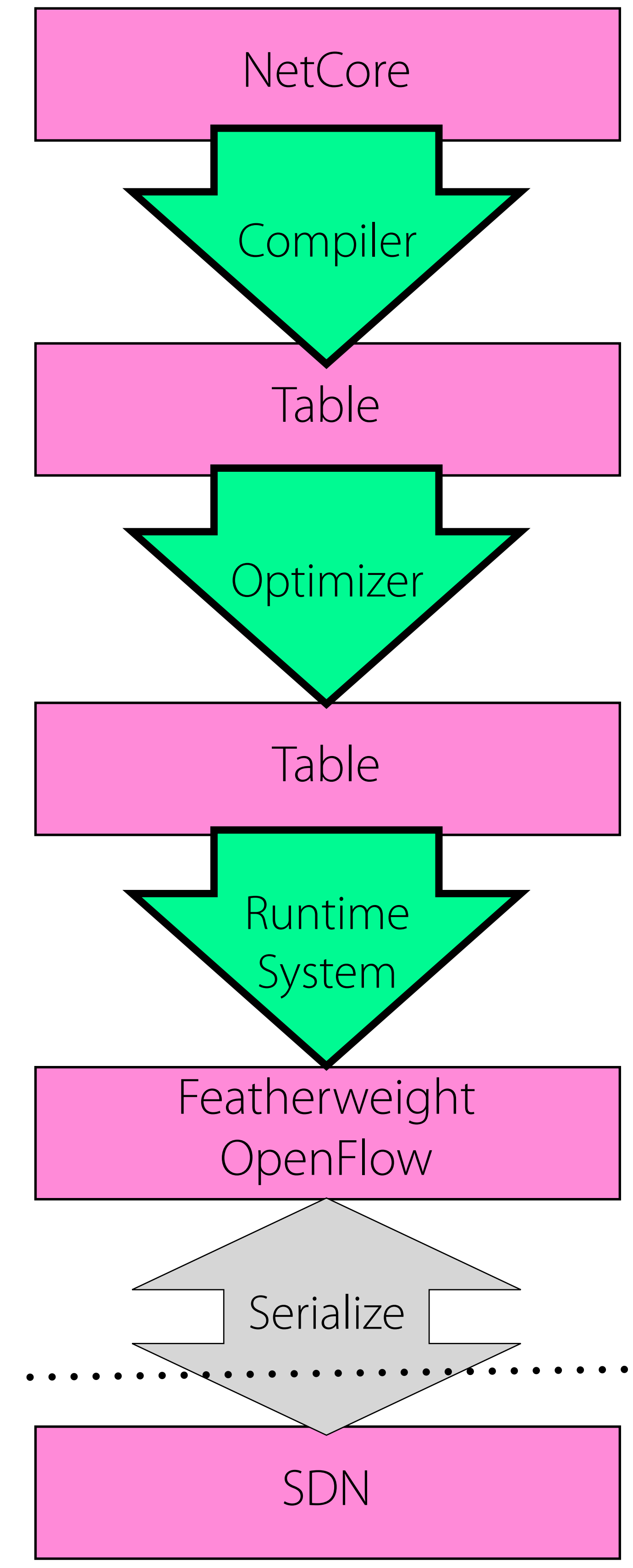


Forwarding elements that implement packet-processing functionality efficiently in hardware

Verified Software Stack

Formalized in Coq

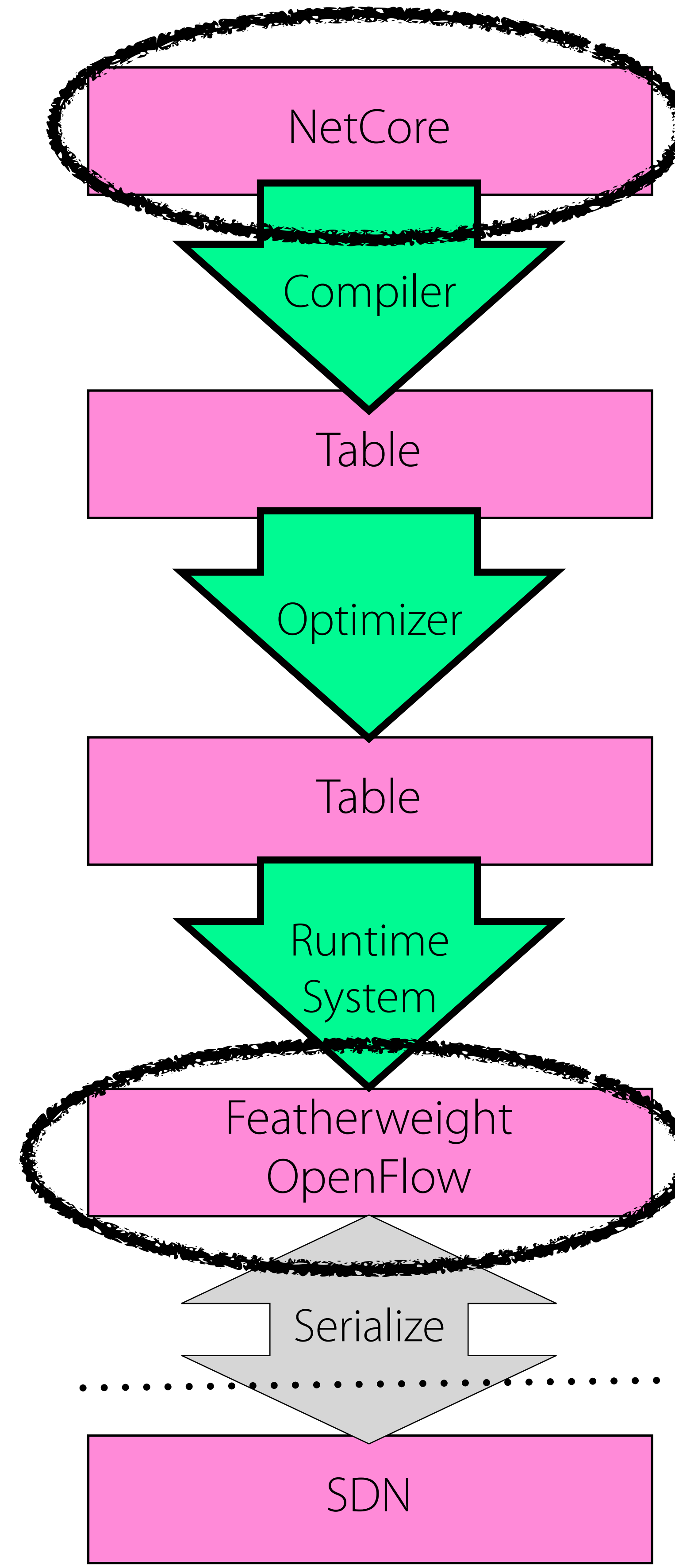
- Denotational semantics of NetCore (an earlier version of NetKAT)
- Operational semantics of OpenFlow
- Compiler
- Run-time system
- Correctness proofs



Verified Software Stack

Formalized in Coq

- Denotational semantics of NetCore (an earlier version of NetKAT)
- Operational semantics of OpenFlow
- Compiler
- Run-time system
- Correctness proofs



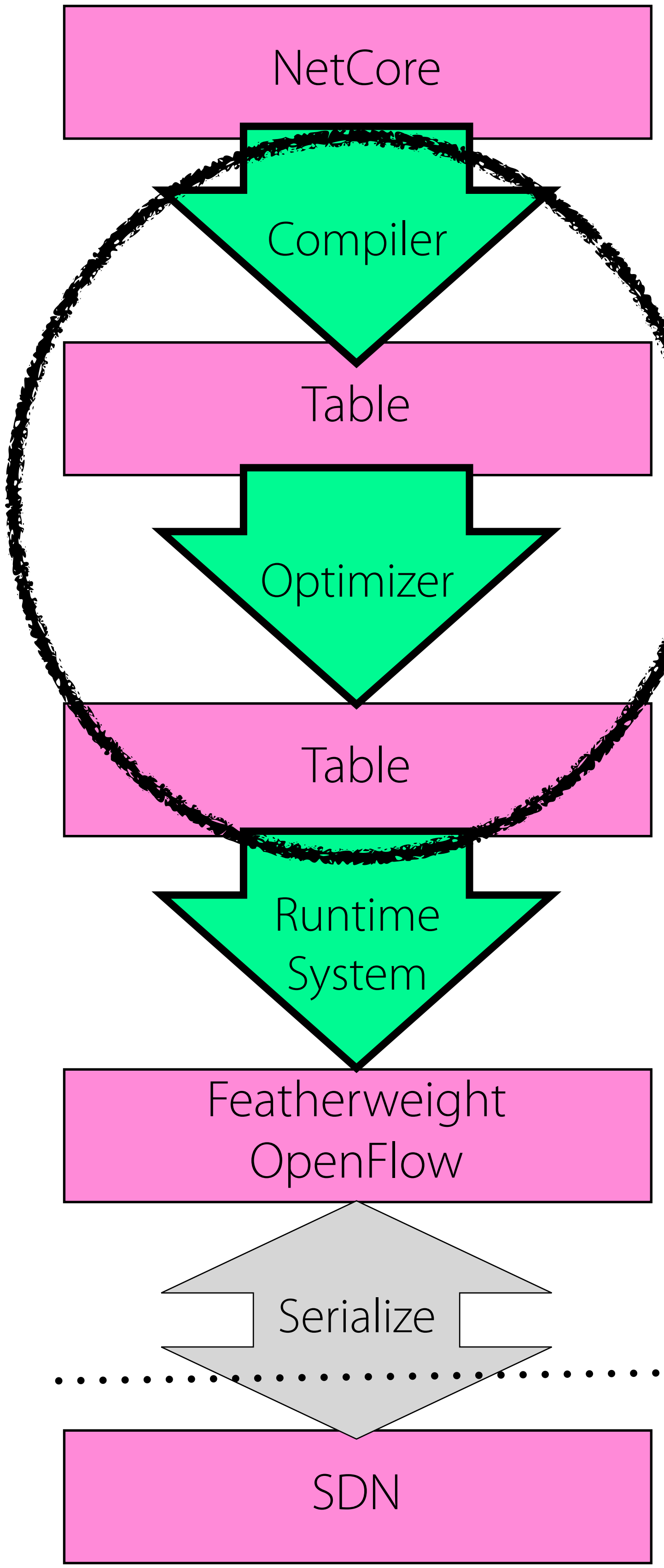
Compiler Correctness

Highlights

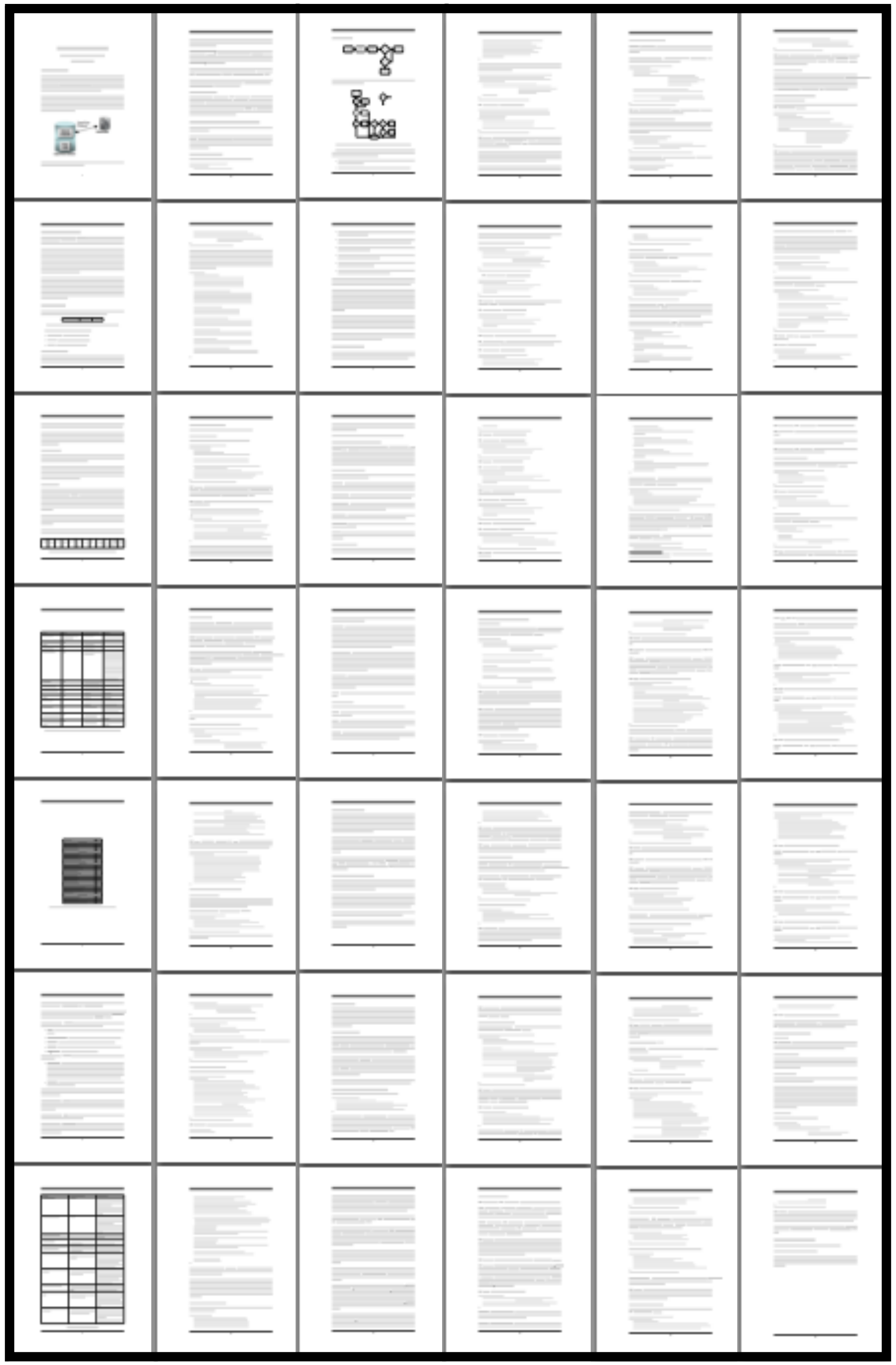
- Library of algebraic properties of tables
- New tactic for proving equalities on bags
- General-purpose table optimizer
- Key invariant: all synthesized predicates are well-formed (w.r.t. protocol types)

Theorem

```
Theorem compile_correct :
  forall pol sw pt pk,
    netcore_eval pol sw pt pk =
    table_eval (compile pol sw) pt pk.
```



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	$[\![RT]\!]$	$\in lp \to \{\mid lp_1 \cdots lp_n \mid\} \times \{\mid CM_1 \cdots C$
	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	$\in \{\mid lp_1 \cdots lp_n \mid\}$
	Produced packets	$outp$	$\in \{\mid lp_1 \cdots lp_n \mid\}$
	Messages from controller	inm	$\in \{\mid SM_1 \cdots SM_n \mid\}$
	Messages to controller	$outm$	$\in \{\mid CM_1 \cdots CM_n \mid\}$
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	$::= \textbf{FlowMod}\ \Delta RT \mid \textbf{PktOut}\ pt\ p$
	Message to controller	CM	$::= \textbf{PktIn}\ pt\ pk \mid \textbf{BarrierReply}\ n$

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

Semantics

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

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

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

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

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

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

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

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

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

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

Forwarding

```
/* Fields to match against flows */
struct ofp_match {
    uint32_t wildcards;        /* Wildcard fields. */
    uint16_t in_port;          /* Input switch port. */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
    uint16_t dl_vlan;          /* Input VLAN. */
    uint8_t dl_vlan_pcp;       /* Input VLAN priority. */
    uint8_t pad1[1];           /* Align to 64-bits. */
    uint16_5 dl_type;          /* Ethernet frame type. */
    uint8_t nw_tos;            /* IP ToS (DSCP field, 6 bits). */
    uint8_t nw_proto;          /* IP protocol or lower 8 bits of
                                  ARP opcode. */
    uint8_t pad2[2];           /* Align to 64-bits. */
    uint32_t nw_src;           /* IP source address. */
    uint32_t nw_dst;           /* IP destination address. */
    uint16_t tp_src;           /* TCP/UDP source port. */
    uint16_t tp_dst;           /* TCP/UDP destination port. */
};
OFP_ASSERT(sizeof(struct ofp_match) == 40);
```

Forwarding

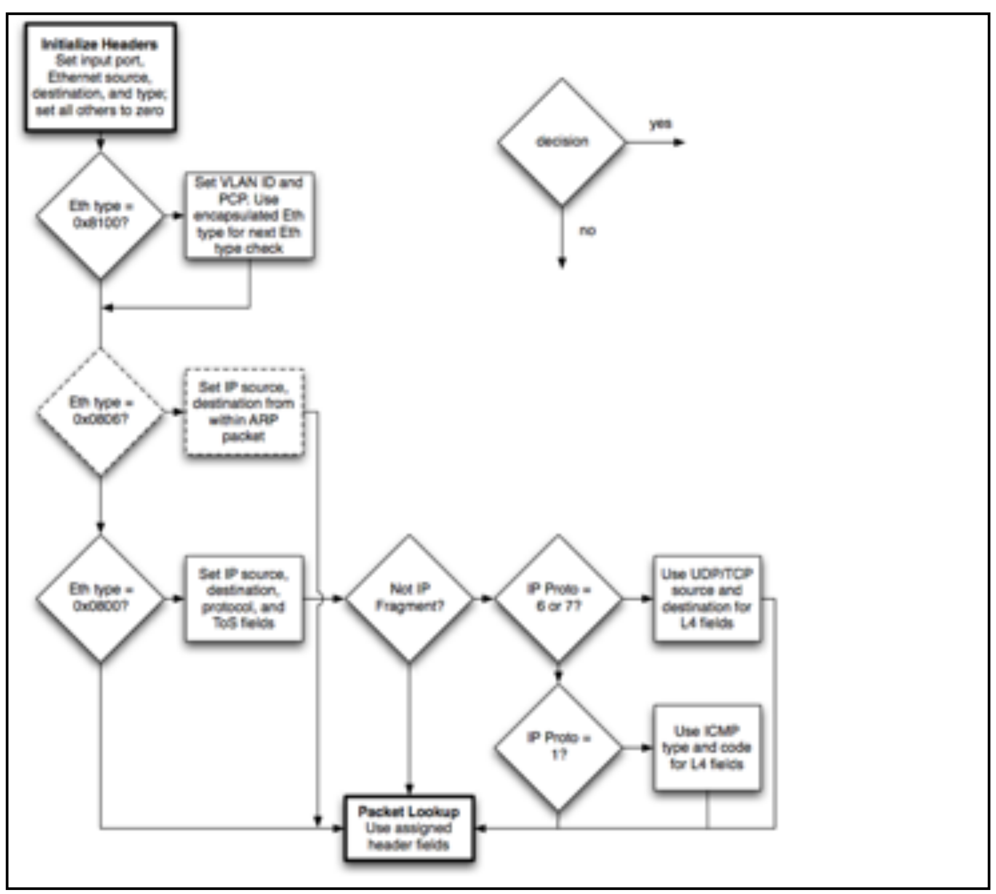
```c
/* Fields to match against flows */
struct ofp_match {
    uint32_t wildcards;        /* Wildcard fields. */
    uint16_t in_port;          /* Input switch port. */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
    uint16_t dl_vlan;          /* Input VLAN. */
    uint8_t dl_vlan_pcp;       /* Input VLAN priority. */
    uint8_t pad1[1];           /* Align to 64-bits. */
    uint16_5 dl_type;          /* Ethernet frame type. */
    uint8_t nw_tos;            /* IP ToS (DSCP field, 6 bits). */
    uint8_t nw_proto;          /* IP protocol or lower 8 bits of
                                  ARP opcode. */
    uint8_t pad2[2];           /* Align to 64-bits. */
    uint32_t nw_src;           /* IP source address. */
    uint32_t nw_dst;           /* IP destination address. */
    uint16_t tp_src;           /* TCP/UDP source port. */
    uint16_t tp_dst;           /* TCP/UDP destination port. */
};
OFP_ASSERT(sizeof(struct ofp_match) == 40);
```

```coq
Record Pattern : Type := MkPattern {
  dlSrc : Wildcard EthernetAddress;
  dlDst : Wildcard EthernetAddress;
  dlType : Wildcard EthernetType;
  dlVlan : Wildcard VLAN;
  dlVlanPcp : Wildcard VLANPriority;
  nwSrc : Wildcard IPAddress;
  nwDst : Wildcard IPAddress;
  nwProto : Wildcard IPProtocol;
  nwTos : Wildcard IPTypeOfService;
  tpSrc : Wildcard TransportPort;
  tpDst : Wildcard TransportPort;
  inPort : Wildcard Port
}.
```

Forwarding

```c
/* Fields to match against flows */
struct ofp_match {
    uint32_t wildcards;        /* Wildcard fields. */
    uint16_t in_port;          /* Input switch port. */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
    uint16_t dl_vlan;          /* Input VLAN. */
    uint8_t dl_vlan_pcp;       /* Input VLAN priority. */
    uint8_t pad1[1];           /* Align to 64-bits. */
    uint16_5 dl_type;          /* Ethernet frame type. */
    uint8_t nw_tos;            /* IP ToS (DSCP field, 6 bits). */
    uint8_t nw_proto;          /* IP protocol or lower 8 bits of
                                  ARP opcode. */
    uint8_t pad2[2];           /* Align to 64-bits. */
    uint32_t nw_src;           /* IP source address. */
    uint32_t nw_dst;           /* IP destination address. */
    uint16_t tp_src;           /* TCP/UDP source port. */
    uint16_t tp_dst;           /* TCP/UDP destination port. */
};
OFP_ASSERT(sizeof(struct ofp_match) == 40);
```

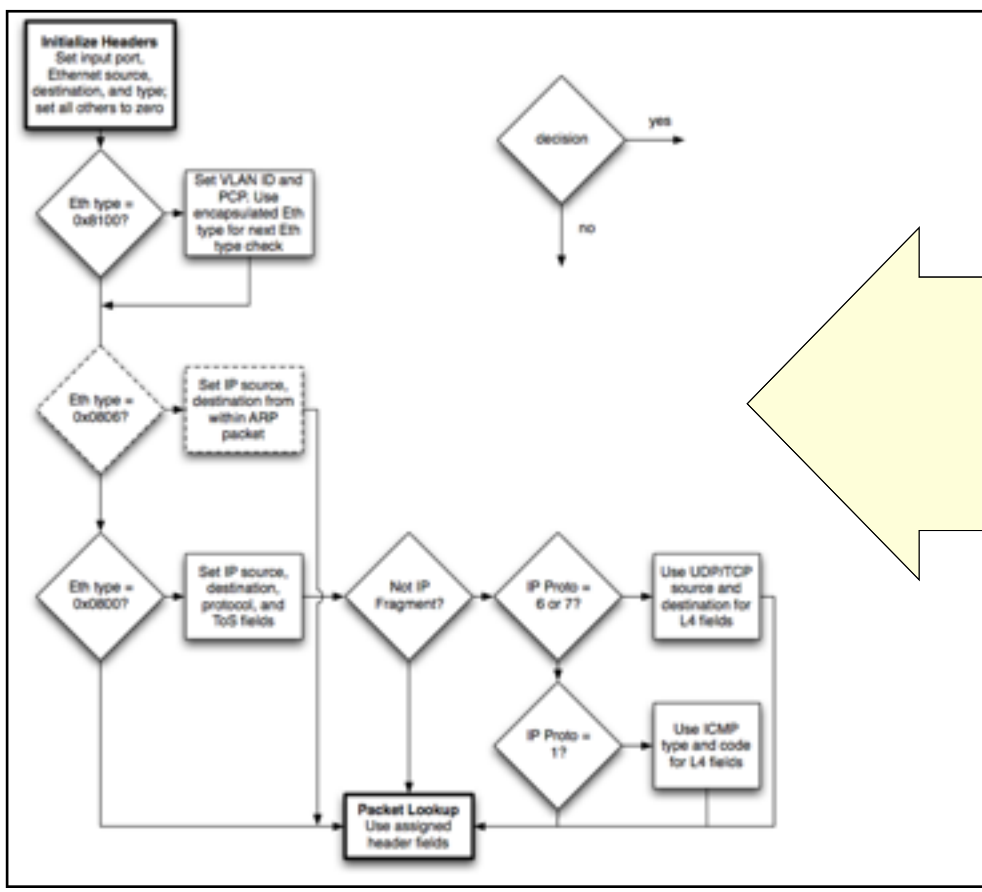
```coq
Record Pattern : Type := MkPattern {
  dlSrc : Wildcard EthernetAddress;
  dlDst : Wildcard EthernetAddress;
  dlType : Wildcard EthernetType;
  dlVlan : Wildcard VLAN;
  dlVlanPcp : Wildcard VLANPriority;
  nwSrc : Wildcard IPAddress;
  nwDst : Wildcard IPAddress;
  nwProto : Wildcard IPProtocol;
  nwTos : Wildcard IPTypeOfService;
  tpSrc : Wildcard TransportPort;
  tpDst : Wildcard TransportPort;
  inPort : Wildcard Port
}.
```



Forwarding

```c
/* Fields to match against flows */
struct ofp_match {
    uint32_t wildcards;        /* Wildcard fields. */
    uint16_t in_port;          /* Input switch port. */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
    uint16_t dl_vlan;          /* Input VLAN. */
    uint8_t dl_vlan_pcp;       /* Input VLAN priority. */
    uint8_t pad1[1];           /* Align to 64-bits. */
    uint16_5 dl_type;          /* Ethernet frame type. */
    uint8_t nw_tos;            /* IP ToS (DSCP field, 6 bits). */
    uint8_t nw_proto;          /* IP protocol or lower 8 bits of
                                  ARP opcode. */
    uint8_t pad2[2];           /* Align to 64-bits. */
    uint32_t nw_src;           /* IP source address. */
    uint32_t nw_dst;           /* IP destination address. */
    uint16_t tp_src;           /* TCP/UDP source port. */
    uint16_t tp_dst;           /* TCP/UDP destination port. */
};
OFP_ASSERT(sizeof(struct ofp_match) == 40);
```

```coq
Record Pattern : Type := MkPattern {
  dlSrc : Wildcard EthernetAddress;
  dlDst : Wildcard EthernetAddress;
  dlType : Wildcard EthernetType;
  dlVlan : Wildcard VLAN;
  dlVlanPcp : Wildcard VLANPriority;
  nwSrc : Wildcard IPAddress;
  nwDst : Wildcard IPAddress;
  nwProto : Wildcard IPProtocol;
  nwTos : Wildcard IPTypeOfService;
  tpSrc : Wildcard TransportPort;
  tpDst : Wildcard TransportPort;
  inPort : Wildcard Port
}.
```



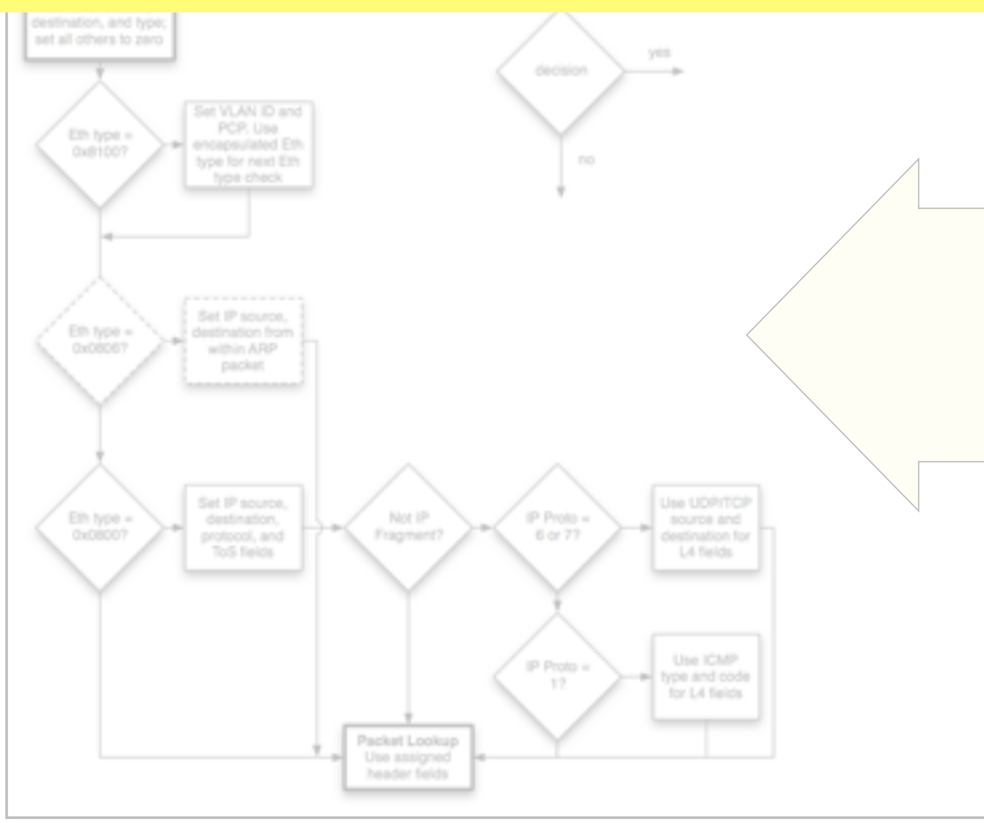

```coq
Definition Pattern_inter (p p':Pattern) :=
  let dlSrc := Wildcard_inter EthernetAddress.eqdec (ptrnDlSrc p) (ptrnDlSrc p') in
  let dlDst := Wildcard_inter EthernetAddress.eqdec (ptrnDlDst p) (ptrnDlDst p') in
  let dlType := Wildcard_inter Word16.eqdec (ptrnDlType p) (ptrnDlType p') in
  let dlVlan := Wildcard_inter Word16.eqdec (ptrnDlVlan p) (ptrnDlVlan p') in
  let dlVlanPcp := Wildcard_inter Word8.eqdec (ptrnDlVlanPcp p) (ptrnDlVlanPcp p') in
  let nwSrc := Wildcard_inter Word32.eqdec (ptrnNwSrc p) (ptrnNwSrc p') in
  let nwDst := Wildcard_inter Word32.eqdec (ptrnNwDst p) (ptrnNwDst p') in
  let nwProto := Wildcard_inter Word8.eqdec (ptrnNwProto p) (ptrnNwProto p') in
  let nwTos := Wildcard_inter Word8.eqdec (ptrnNwTos p) (ptrnNwTos p') in
  let tpSrc := Wildcard_inter Word16.eqdec (ptrnTpSrc p) (ptrnTpSrc p') in
  let tpDst := Wildcard_inter Word16.eqdec (ptrnTpDst p) (ptrnTpDst p') in
  let inPort := Wildcard_inter Word16.eqdec (ptrnInPort p) (ptrnInPort p') in
    MkPattern dlSrc dlDst dlType dlVlan dlVlanPcp
              nwSrc nwDst nwProto nwTos
              tpSrc tpDst
              inPort.
Definition exact_pattern (pk : Packet) (pt : Word16.T) : Pattern :=
  MkPattern
    (WildcardExact (pktDlSrc pk)) (WildcardExact (pktDlDst pk))
    (WildcardExact (pktDlTyp pk))
    (WildcardExact (pktDlVlan pk)) (WildcardExact (pktDlVlanPcp pk))
    (WildcardExact (pktNwSrc pk)) (WildcardExact (pktNwDst pk))
    (WildcardExact (pktNwProto pk)) (WildcardExact (pktNwTos pk))
    (Wildcard_of_option (pktTpSrc pk)) (Wildcard_of_option (pktTpDst pk))
    (WildcardExact pt).
Definition match_packet (pt : Word16.T) (pk : Packet) (pat : Pattern) : bool :=
  negb (Pattern_is_empty (Pattern_inter (exact_pattern pk pt) pat)).
```

Forwarding

```c
/* Fields to match against flows */
struct ofp_match {
    uint32_t wildcards;        /* Wildcard fields. */
    uint16_t in_port;          /* Input switch port. */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
    uint16_t dl_vlan;          /* Input VLAN. */
    uint8_t dl_vlan_pcp;       /* Input VLAN priority. */
    uint8_t pad1[1];           /* Align to 64-bits. */
    uint16_5 dl_type;          /* Ethernet frame type. */
    uint8_t nw_tos;            /* IP ToS (DSCP field, 6 bits). */
    uint8_t nw_proto;          /* IP protocol or lower 8 bits of
                                  ARP opcode. */
    uint8_t pad2[2];           /* Align to 64-bits. */
    uint32_t nw_src;           /* IP source address. */
    uint32_t nw_dst;           /* IP destination address. */
    uint16_t tp_src;           /* TCP/UDP source port. */
    uint16_t tp_dst;           /* TCP/UDP destination port. */
};
OFP_ASSERT(sizeof(struct ofp_match) == 40);
```

```coq
Record Pattern : Type := MkPattern {
  dlSrc : Wildcard EthernetAddress;
  dlDst : Wildcard EthernetAddress;
  dlType : Wildcard EthernetType;
  dlVlan : Wildcard VLAN;
  dlVlanPcp : Wildcard VLANPriority;
  nwSrc : Wildcard IPAddress;
  nwDst : Wildcard IPAddress;
  nwProto : Wildcard IPProtocol;
  nwTos : Wildcard IPTypeOfService;
  tpSrc : Wildcard TransportPort;
  tpDst : Wildcard TransportPort;
  inPort : Wildcard Port
}.
```

Detailed model of matching, forwarding, and flow table update




```coq
  let dlDst := Wildcard_inter EthernetAddress.eqdec (ptrnDlDst p) (ptrnDlDst p') in
  let dlType := Wildcard_inter Word16.eqdec (ptrnDlType p) (ptrnDlType p') in
  let dlVlan := Wildcard_inter Word16.eqdec (ptrnDlVlan p) (ptrnDlVlan p') in
  let dlVlanPcp := Wildcard_inter Word8.eqdec (ptrnDlVlanPcp p) (ptrnDlVlanPcp p') in
  let nwSrc := Wildcard_inter Word32.eqdec (ptrnNwSrc p) (ptrnNwSrc p') in
  let nwDst := Wildcard_inter Word32.eqdec (ptrnNwDst p) (ptrnNwDst p') in
  let nwProto := Wildcard_inter Word8.eqdec (ptrnNwProto p) (ptrnNwProto p') in
  let nwTos := Wildcard_inter Word8.eqdec (ptrnNwTos p) (ptrnNwTos p') in
  let tpSrc := Wildcard_inter Word16.eqdec (ptrnTpSrc p) (ptrnTpSrc p') in
  let tpDst := Wildcard_inter Word16.eqdec (ptrnTpDst p) (ptrnTpDst p') in
  let inPort := Wildcard_inter Word16.eqdec (ptrnInPort p) (ptrnInPort p') in
    MkPattern dlSrc dlDst dlType dlVlan dlVlanPcp
              nwSrc nwDst nwProto nwTos
              tpSrc tpDst
              inPort.
Definition exact_pattern (pk : Packet) (pt : Word16.T) : Pattern :=
  MkPattern
    (WildcardExact (pktDlSrc pk)) (WildcardExact (pktDlDst pk))
    (WildcardExact (pktDlTyp pk))
    (WildcardExact (pktDlVlan pk)) (WildcardExact (pktDlVlanPcp pk))
    (WildcardExact (pktNwSrc pk)) (WildcardExact (pktNwDst pk))
    (WildcardExact (pktNwProto pk)) (WildcardExact (pktNwTos pk))
    (Wildcard_of_option (pktTpSrc pk)) (Wildcard_of_option (pktTpDst pk))
    (WildcardExact pt).
Definition match_packet (pt : Word16.T) (pk : Packet) (pat : Pattern) : bool :=
  negb (Pattern_is_empty (Pattern_inter (exact_pattern pk pt) pat)).
```

Asynchrony

“In the absence of barrier messages, switches may arbitrarily reorder messages to maximize performance.”

“There is no packet output ordering guaranteed within a port.”

Asynchrony

"In the absence of barrier messages, switches may arbitrarily reorder messages to maximize performance."

"There is no packet output ordering guaranteed within a port."

```
Definition InBuf := Bag Packet.
Definition OutBuf := Bag Packet.
Definition OFInBuf := Bag SwitchMsg.
Definition OFOutBuf := Bag CtrlMsg.
```

Asynchrony

"In the absence of barrier messages, switches may arbitrarily reorder messages to maximize performance."

"There is no packet output ordering guaranteed within a port."

Essential asynchrony: packet buffers, message reordering, and barriers

```
Definition InBuf := Bag Packet.
Definition OutBuf := Bag Packet.
Definition OFInBuf := Bag SwitchMsg.
Definition OFOutBuf := Bag CtrlMsg.
```

Weak Bisimulation

$(H_1,$  $)$

Weak Bisimulation

$(H_1,$  $) \longrightarrow (S_1, pt_1,$  $)$

Weak Bisimulation

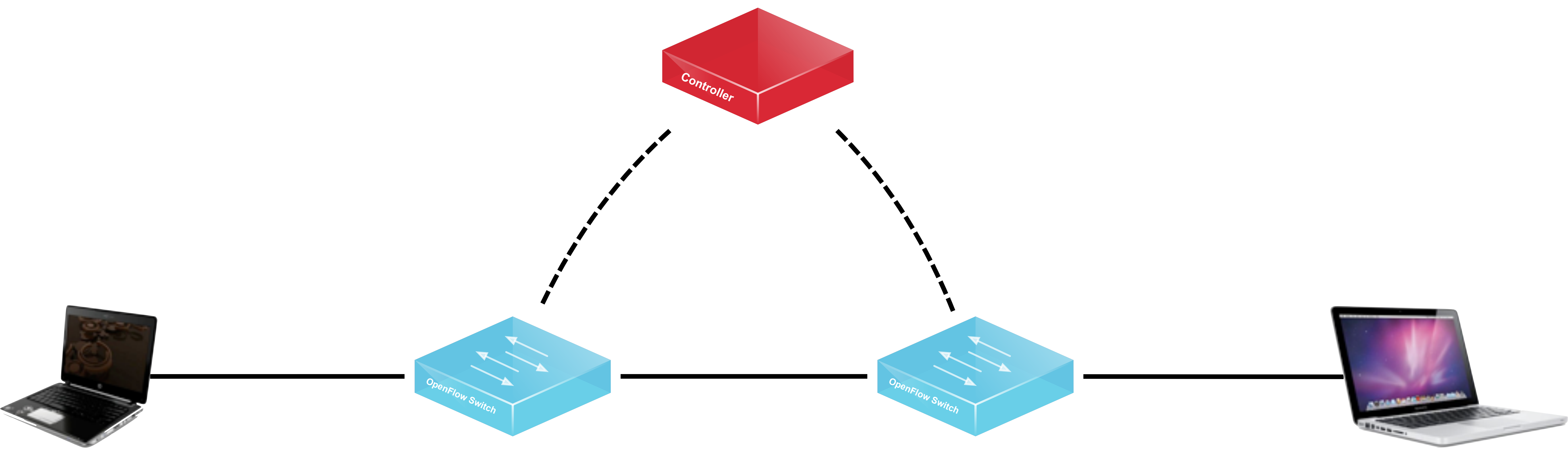
(H_1, ) $\longrightarrow$ (S_1, pt_1, ) $\longrightarrow$ (S_2, pt_1, )

Weak Bisimulation

$(H_1,$  $) \longrightarrow (S_1, pt_1,$  $) \longrightarrow (S_2, pt_1,$  $) \longrightarrow (H_2,$  $)$

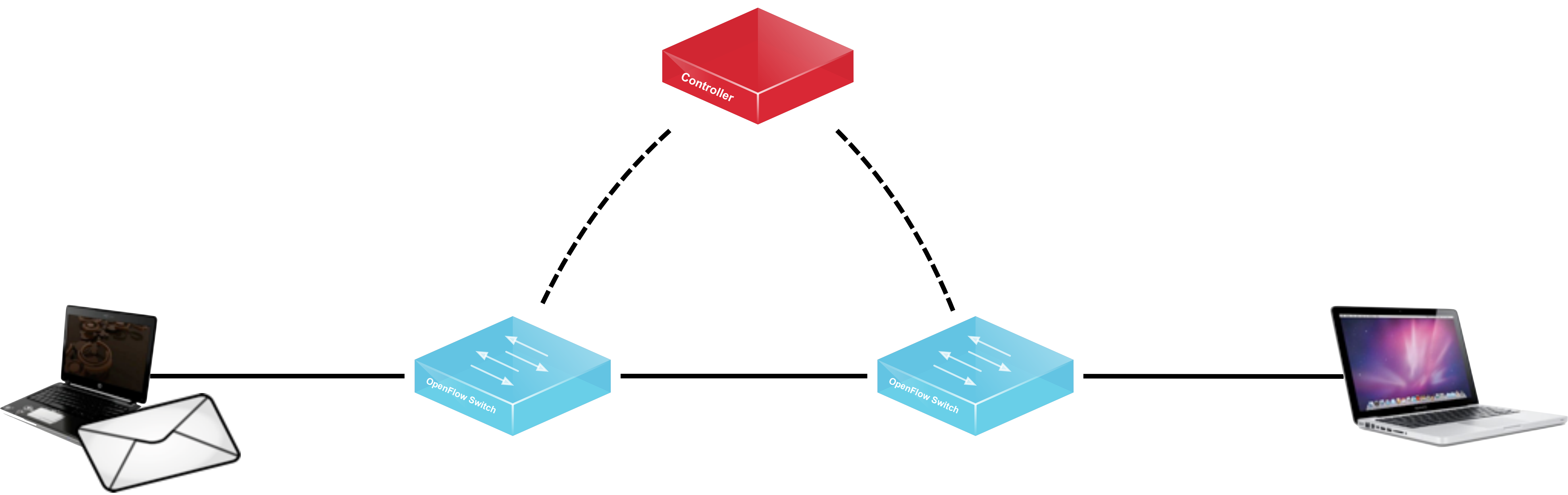
Weak Bisimulation

$(H_1,$  $) \longrightarrow (S_1, pt_1,$  $) \longrightarrow (S_2, pt_1,$  $) \longrightarrow (H_2,$  $)$



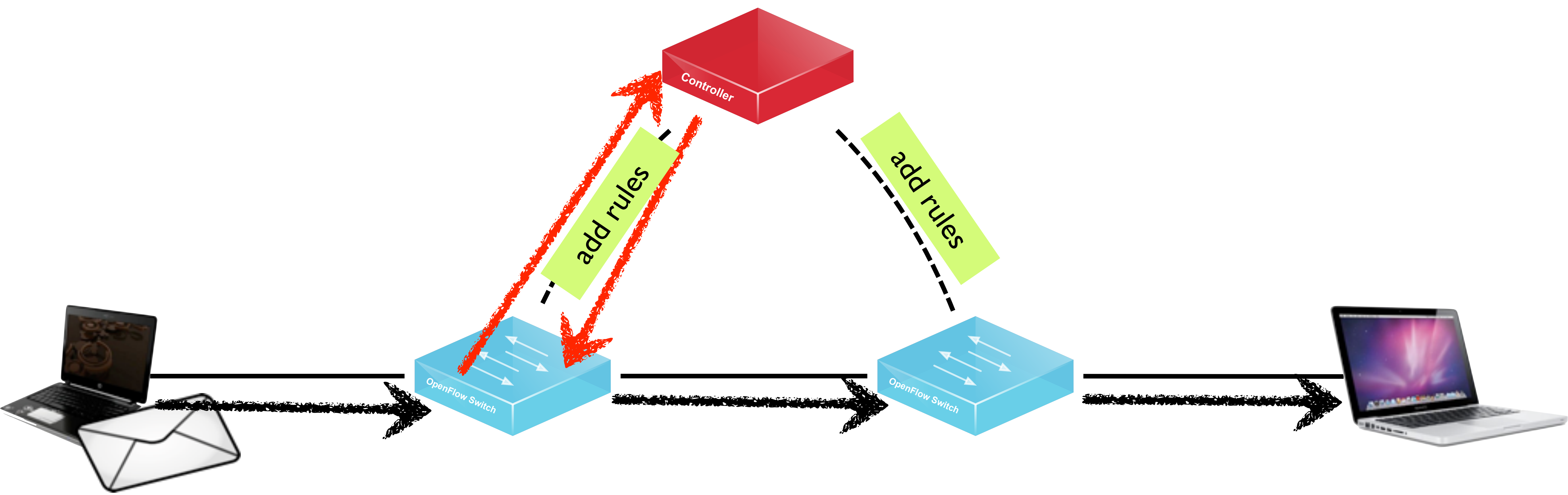
Weak Bisimulation

$(H_1, \ ) \longrightarrow (S_1, pt_1, \ ) \longrightarrow (S_2, pt_1, \ ) \longrightarrow (H_2, \ )$

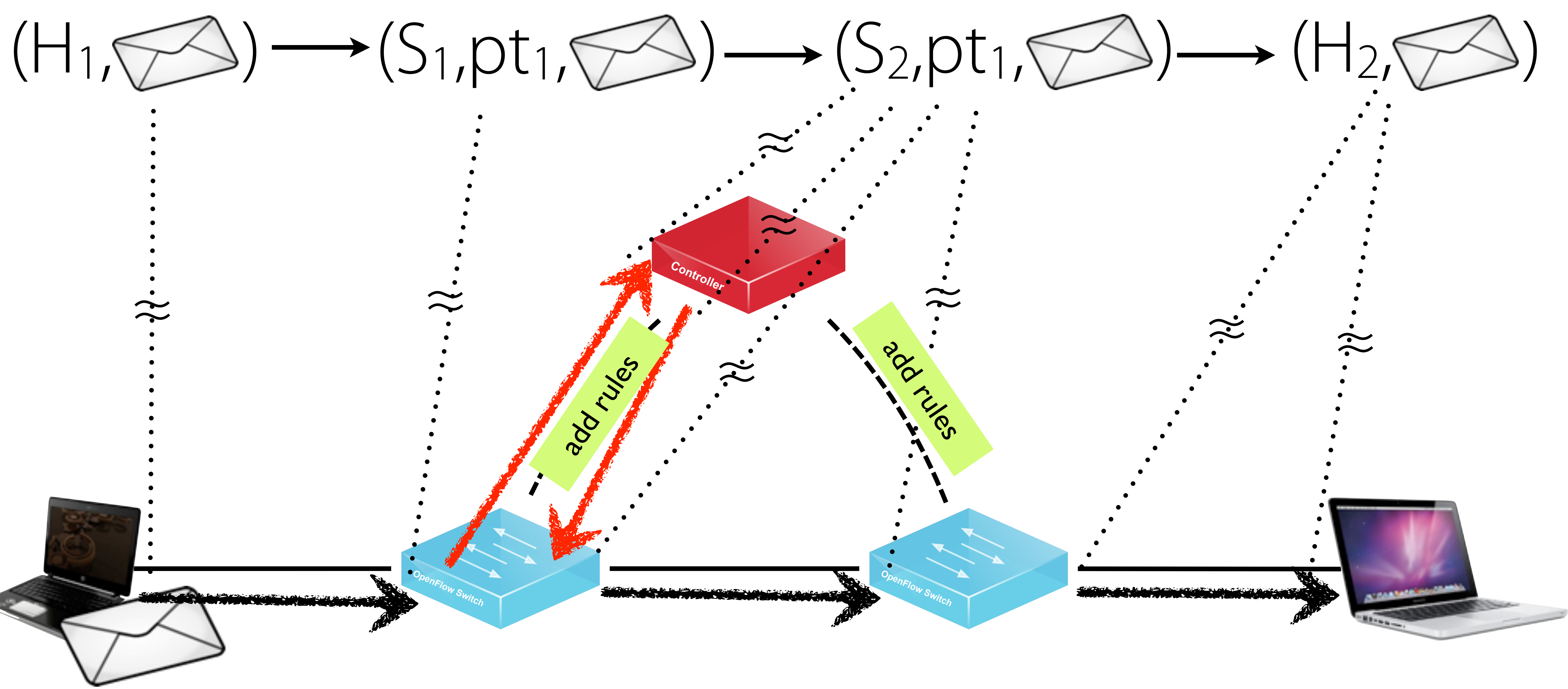


Weak Bisimulation

$(H_1,\ ) \longrightarrow (S_1, pt_1,\ ) \longrightarrow (S_2, pt_1,\ ) \longrightarrow (H_2,\ )$



Weak Bisimulation



Theorem: NetCore abstract semantics is weakly bisimilar to Featherweight OpenFlow + NetCore controller

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

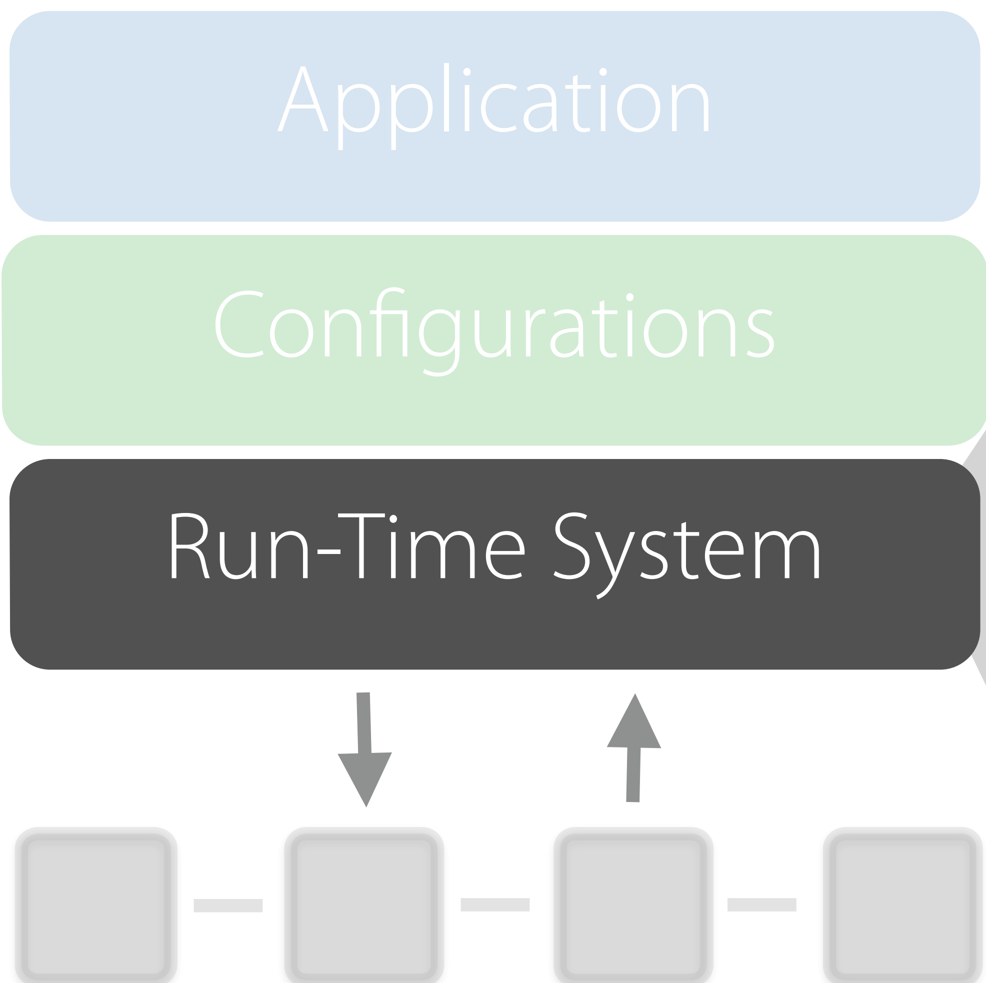
Theorem

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

Consistent Updates

[SIGCOMM '12]

Run-Time Model

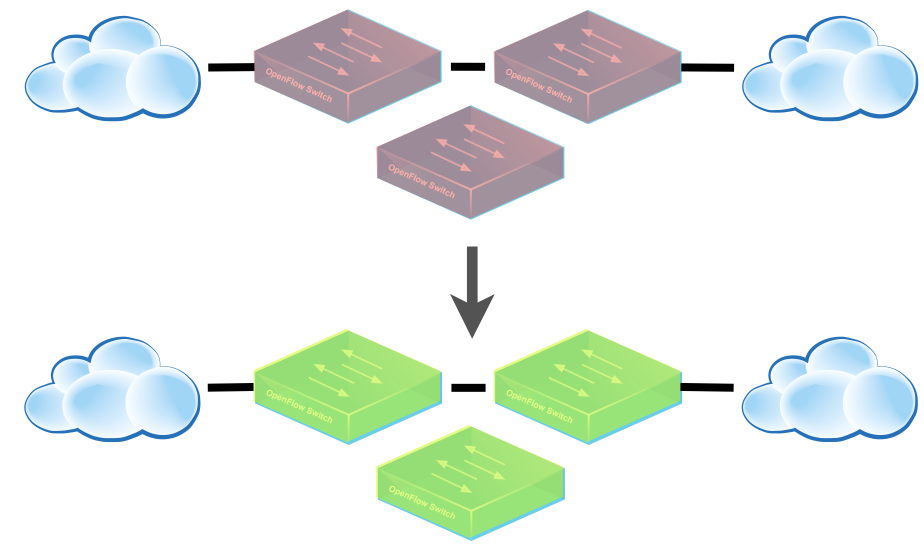



```
let swap_update_for (t : t) sw_id c_id new_table : unit Deferred.t =
   let max_priority = 65535 in
   let old_table = match SwitchMap.find t.edge sw_id with | Some ft -> ft | None -> [] in
   let (new_table, _) = List.fold new_table ~init:([], max_priority)
        ~f:(fun (acc,pri) x -> ((x,pri) :: 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



Challenges

- The network is a distributed system
- Can only update one element at a time

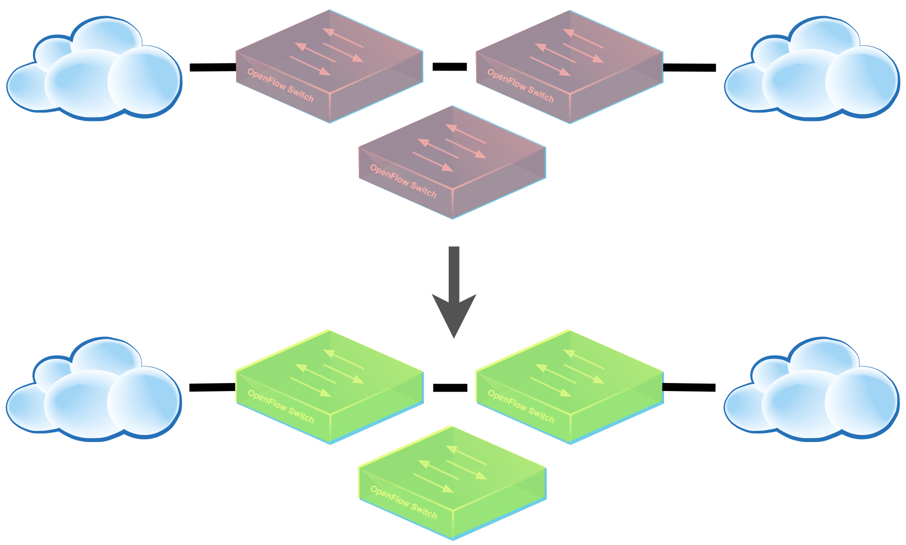
Our Approach

- Provide programmers with a construct for updating the entire network at once
- Semantics ensures "reasonable" behavior
- Engineer efficient implementations:
  - Compiler constructs update protocols
  - Optimizations applied automatically

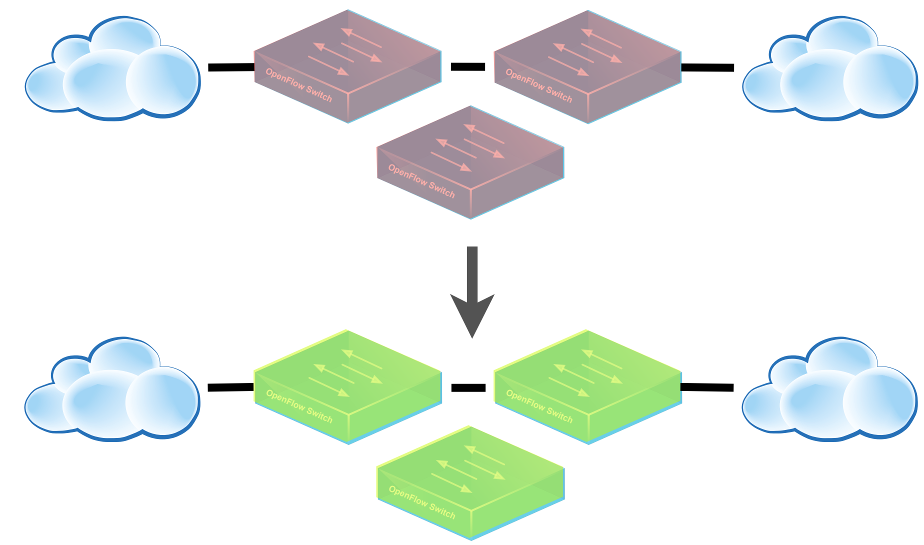
Update Semantics

Atomic Updates

- Seem sensible...
- but costly to implement...
- and difficult to reason about, due to behavior on in-flight packets



Update Semantics



Atomic Updates

- Seem sensible...
- but costly to implement...
- and difficult to reason about, due to behavior on in-flight packets

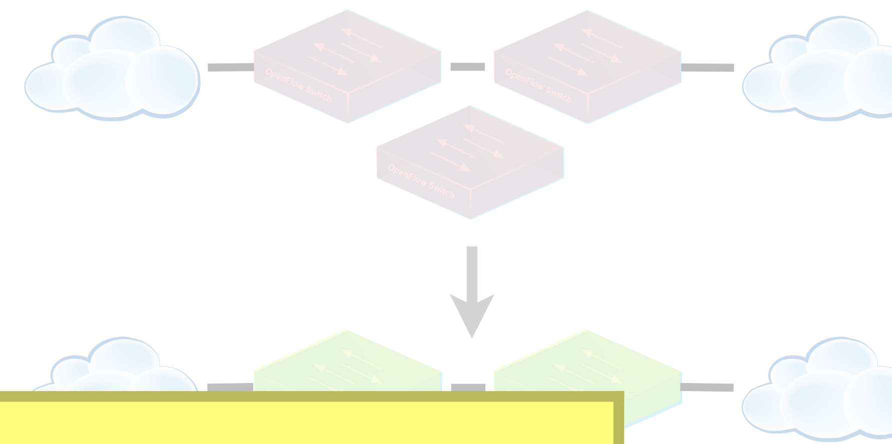
Per-Packet Consistent Updates

Every packet processed with old or new configuration, but not a mixture of the two

Per-Flow Consistent Updates

Every set of related packets processed with old or new configuration, but not a mixture of the two

Update Semantics



Atomic Updates

- Seem sensible...
- but costly to implement...
- and difficult to reason about, due to
  behavi

Per-Pac

Every pa
configu

Theorem (Universal Property Preservation)

An update is per-packet consistent if and only if it preserves all safety properties.

Per-Flow Consistent Updates

Every set of related packets processed with old or new configuration, but not a mixture of the two

Implementation

Two-phase commit

- Build versioned internal and edge switch configurations
- Phase 1: Install internal configuration
- Phase 2: Install edge configuration

Pure Extension

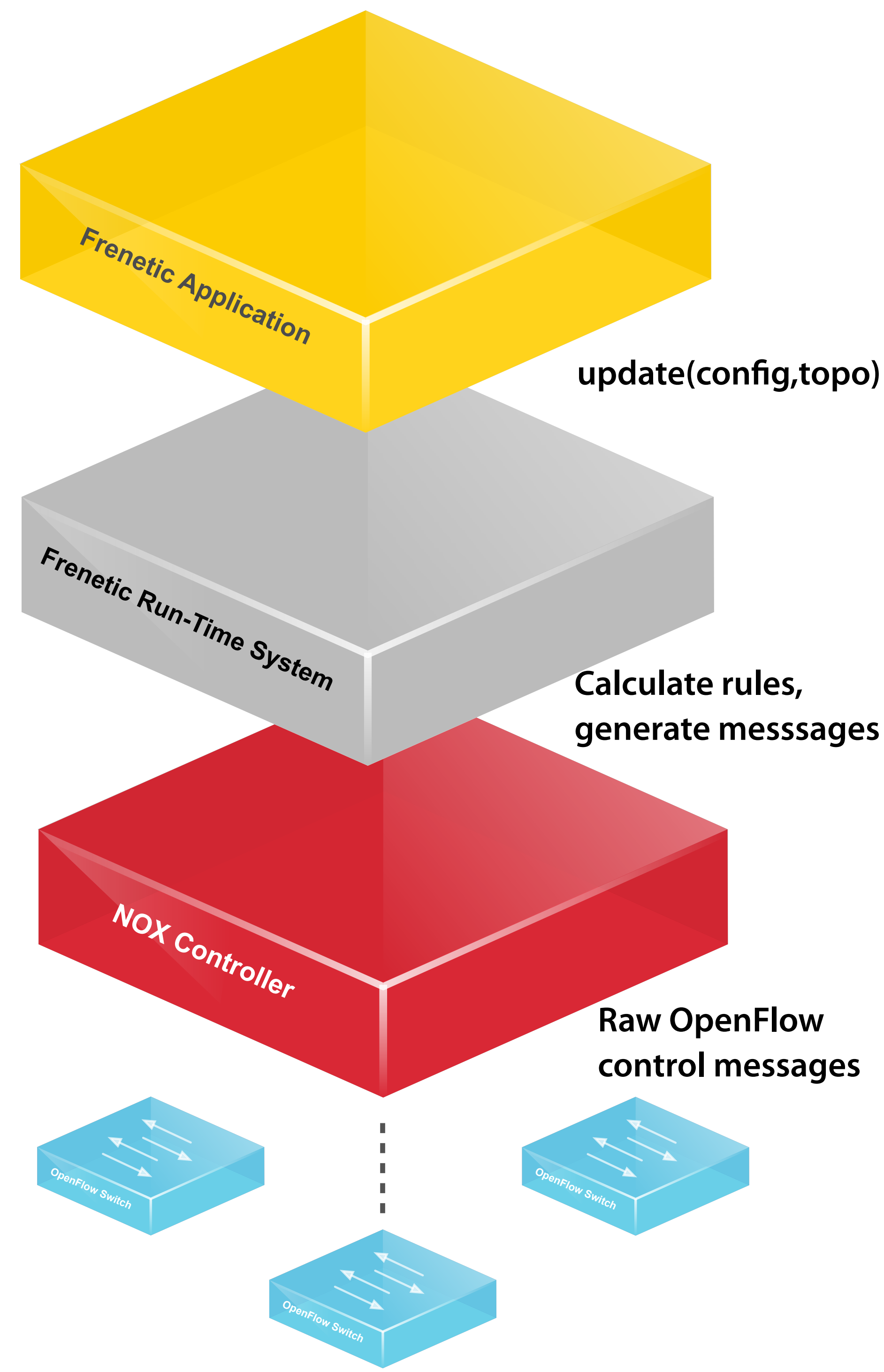
- Update strictly adds paths

Pure Retraction

- Update strictly removes paths

Sub-space updates

- Update modifies a small number of paths



Wrapping Up

Conclusion

- Lots of great PL problems in networking!
- SDN is an enabling technology for this kind of research
- Frenetic is a new platform for programming and reasoning about SDNs:
  - Automated formal reasoning in NetKAT [POPL '14]
  - Consistent updates [SIGCOMM '12]
  - Machine-verified controller [PLDI '13]
- Other work
  - Traffic isolation [HotSDN '12]
  - Joint host-network programming [SIGCOMM '13, HotNets '13]
  - Declarative fault tolerance [HotSDN '13]
  - Dynamic software updates [HotSDN '13]
  - Configuration synthesis [SYNT '13]
  - Tierless programming [HotSDN '13]

Frenetic @ Home

TP-Link TL-WR1043ND

$50

Open firmware:

`www.openwrt.com`

Frenetic @ Home

Thank you!

Collaborators

- Carolyn Anderson (Swarthmore)
- Spiridon Aristides Eliopoulos (Cornell)
- Jean-Baptiste Jeannin (CMU)
- Dexter Kozen (Cornell)
- Matthew Milano (Cornell)
- Jennifer Rexford (Princeton)
- Mark Reitblatt (Cornell)
- Cole Schlesinger (Princeton)
- Alexandra Silva (Nijmegen)
- Laure Thompson (Cornell)
- Dave Walker (Princeton)

frenetic

Papers, Code, etc.

`http://frenetic-lang.org/`