It's the last COS 326 class!

David Walker
COS 326
Princeton University

COS 326 Final Exam

Logistics:

- Friday Jan 26
- 1:30pm
- McCosh 46

Note: If you are doing study abroad, make sure that you email Chris Moretti so we can arrange the exam abroad. (Many of you have.)

COS 326 Final Exam

Contents:

- The entire semester
  - the lectures
  - the assignments
- There will be more emphasis on the 2nd half
- I will probably ask a question that is similar to something on the midterm
  - so make sure you know that stuff

Major Topics From 2nd Half

Modules

- signatures, structures, functors

Reasoning about modules

- representation invariants
- abstraction functions
- proofs of module equivalence

Laziness, memoization

Abstractions for parallel FP

- futures, sequences, map, reduce
- parallel functional algorithms, work, span

Precept this Week

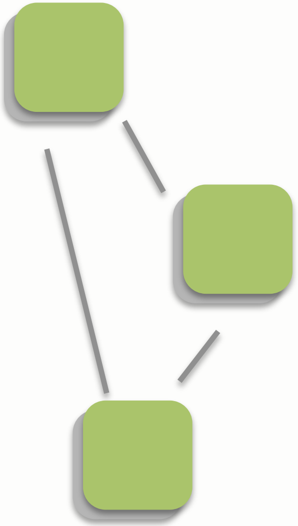
- A couple of questions from the 2015 exam

The Frenetic Project:
Adventures in Functional Networking

David Walker

COS 326

Princeton University



Course Themes

- Functional vs. imperative programming
  - a new way to think about the algorithms you write
- Modularity
- Abstraction
- Parallelism
- Equational reasoning

Useful on a day-to-day basis and in research to transform the way people think about solving programming problems:

frenetic

frenetic >>

http://frenetic-lang.org

Cornell:

- Faculty: Nate Foster, Dexter Kozen, Gun Sirer
- Students & Post Docs: Carolyn Anderson, Shrutarshi Basu, Mark Reitblatt, Robert Soule, Alec Story

Princeton:

- Faculty: Jen Rexford, Dave Walker
- Students & Post Docs: Ryan Beckett, Jennifer Gossels, Rob Harrison, Xin Jin, Naga Katta, Chris Monsanto, Srinivas Narayana, Josh Reich, Cole Schlesinger

UMass:

- Faculty: Arjun Guha

A Quick Story Circa 2009 @ Princeton

Dave:
Hey Jen, what's networking?

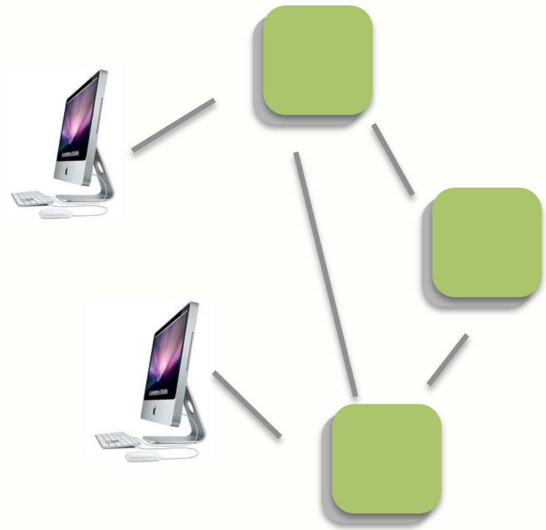
Jen:
Oooh, it's super-awesome.
No lambda calculus required!

Nate:
Too bad about the lambda calculus.
But fill us in.

What is Networking?

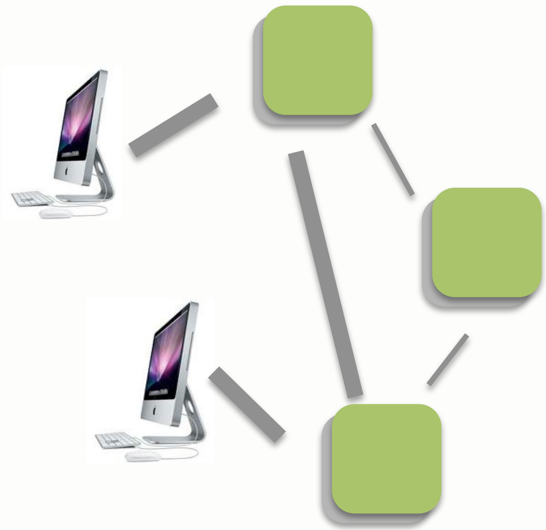
end-hosts need
to communicate

What is Networking?



Ethernet switches connect them

What is Networking?

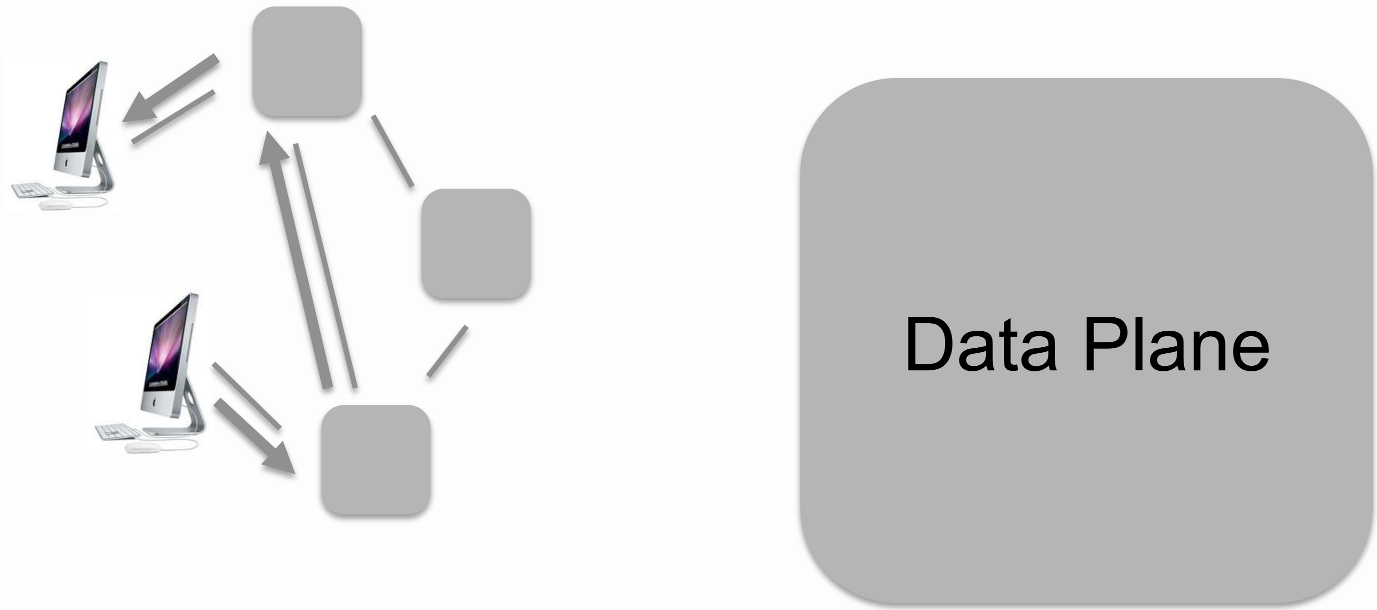


which decide how packets
should be forwarded

Control Plane

What is Networking?

and actually forward them



A Quick Story Circa 2009 @ Princeton

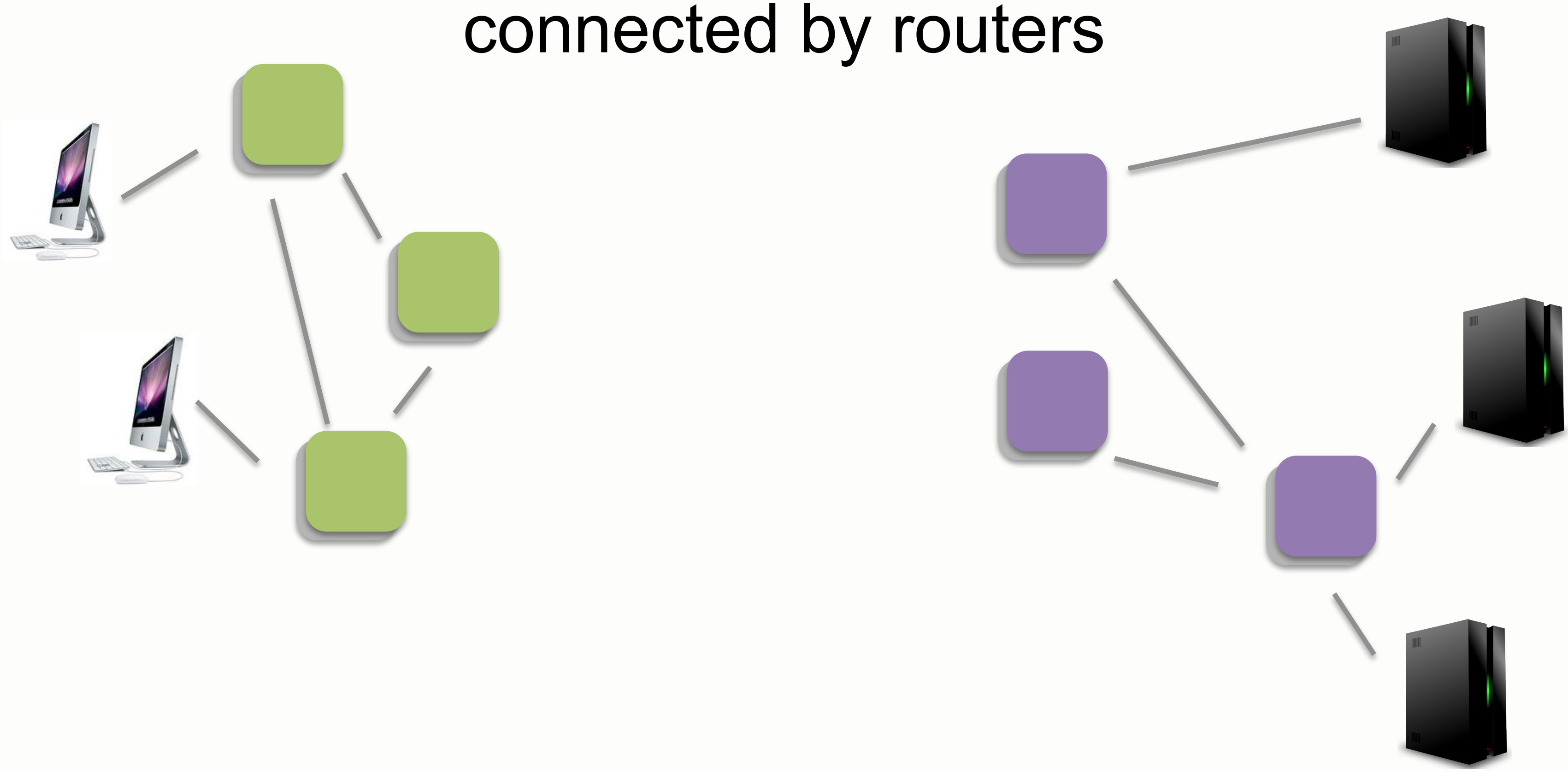
Nate:
  Sounds simple enough. Is that it?

Jen:
  There's a little more …
  Still no lambda calculus though.

Dave:
  Darn.

What is Networking?

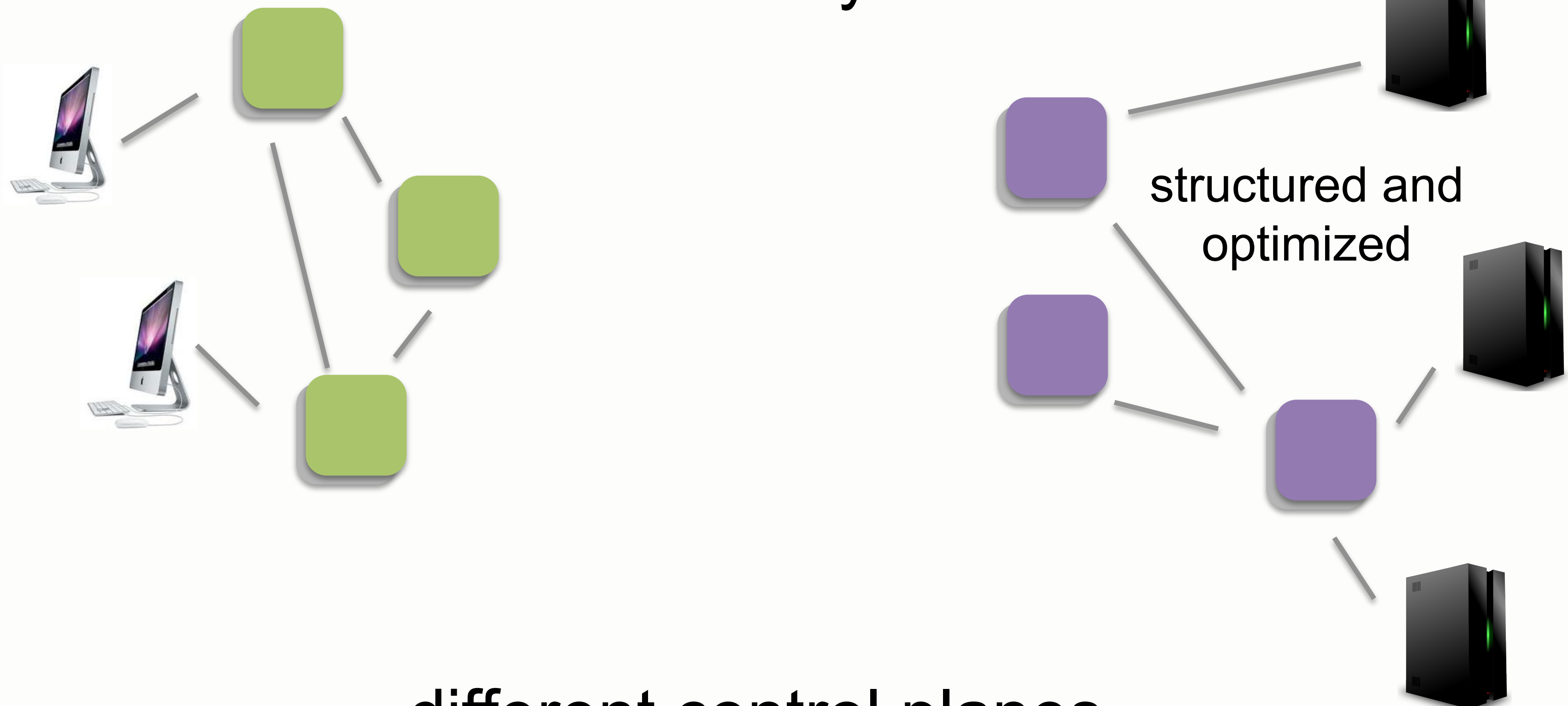
add servers ...
connected by routers



What is Networking?

plug-and-play

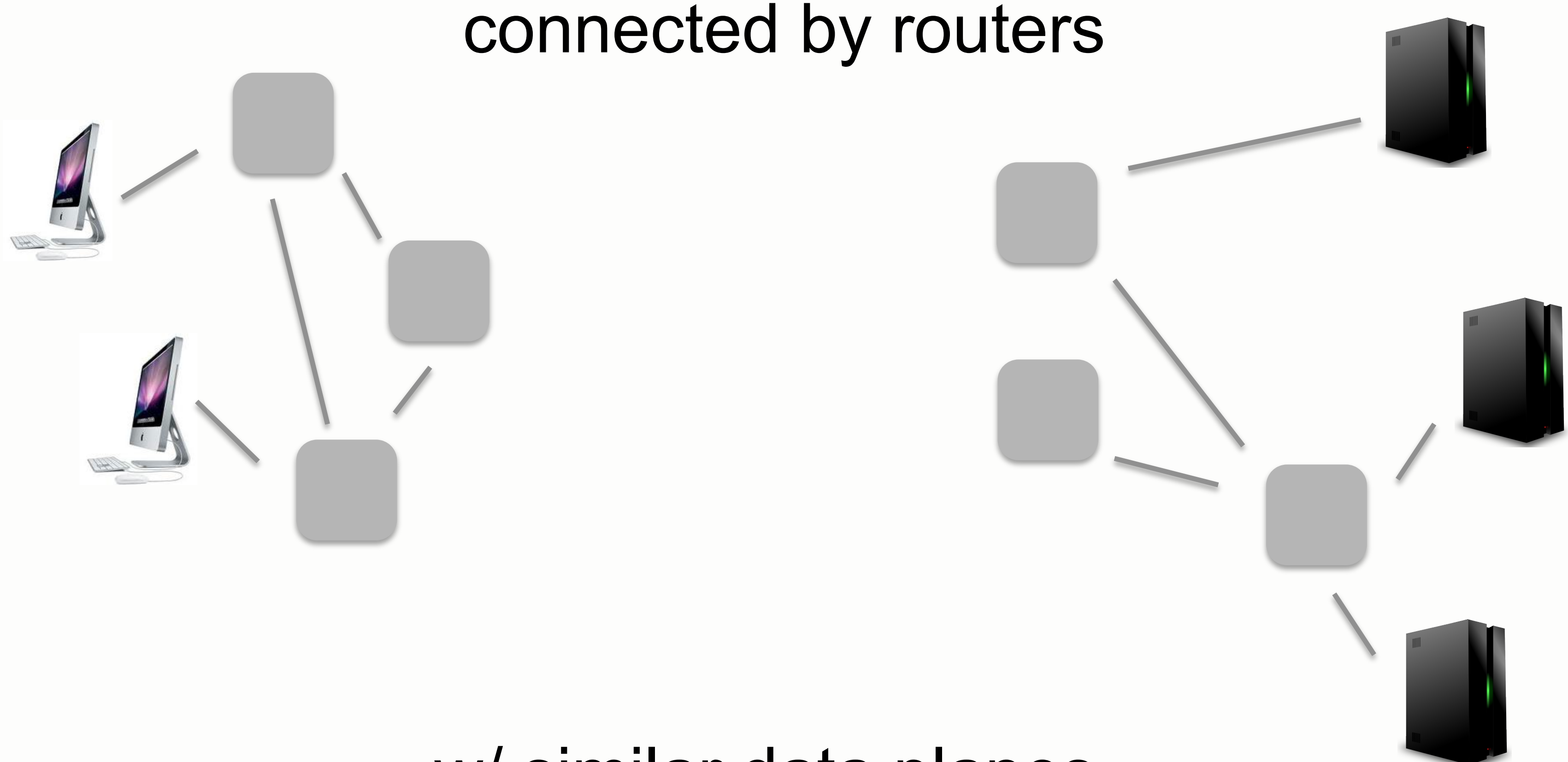
add servers ...
connected by routers



different control planes

What is Networking?

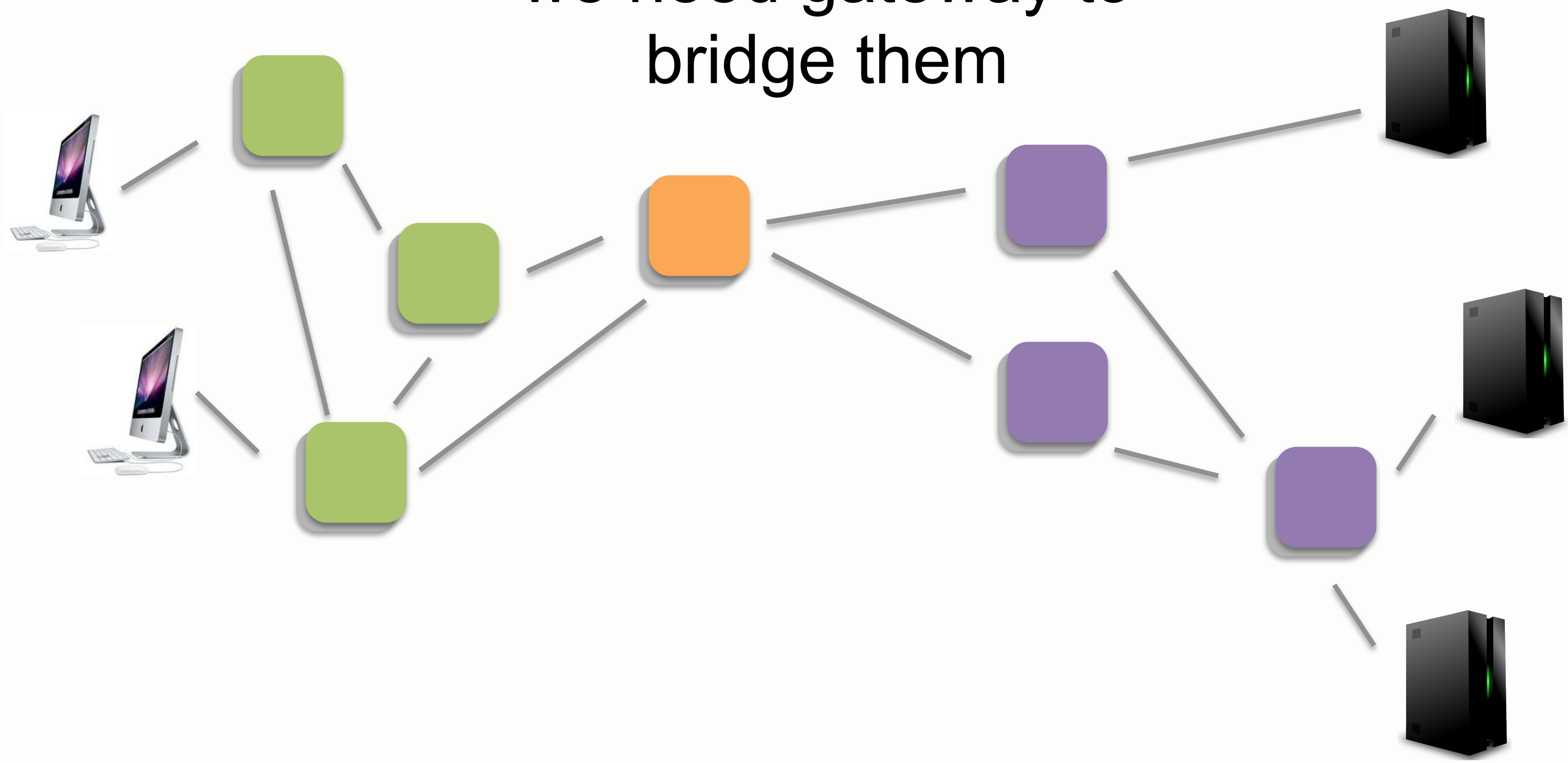
add servers ...
connected by routers



w/ similar data planes

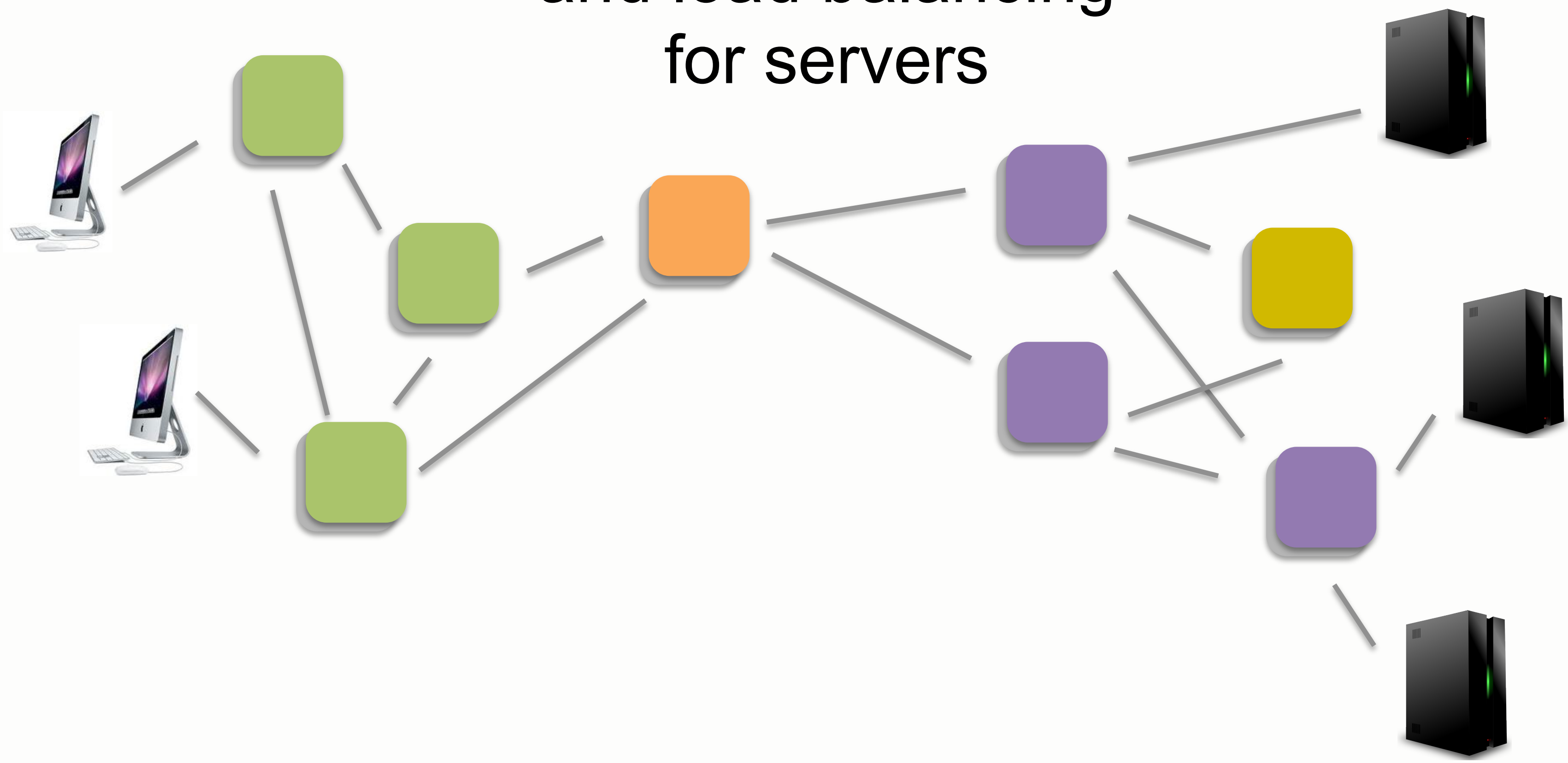
What is Networking?

we need gateway to
bridge them



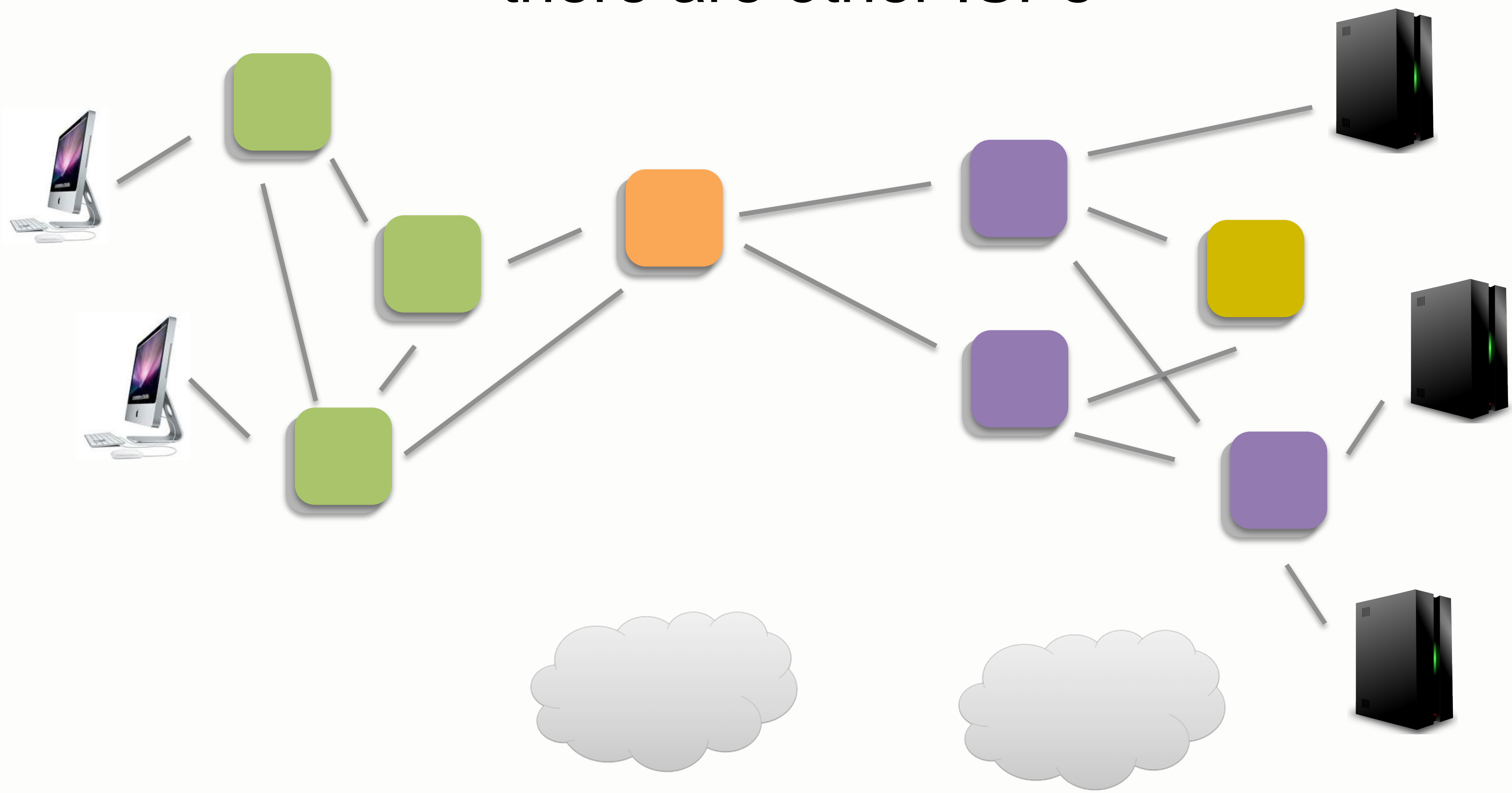
What is Networking?

and load balancing
for servers



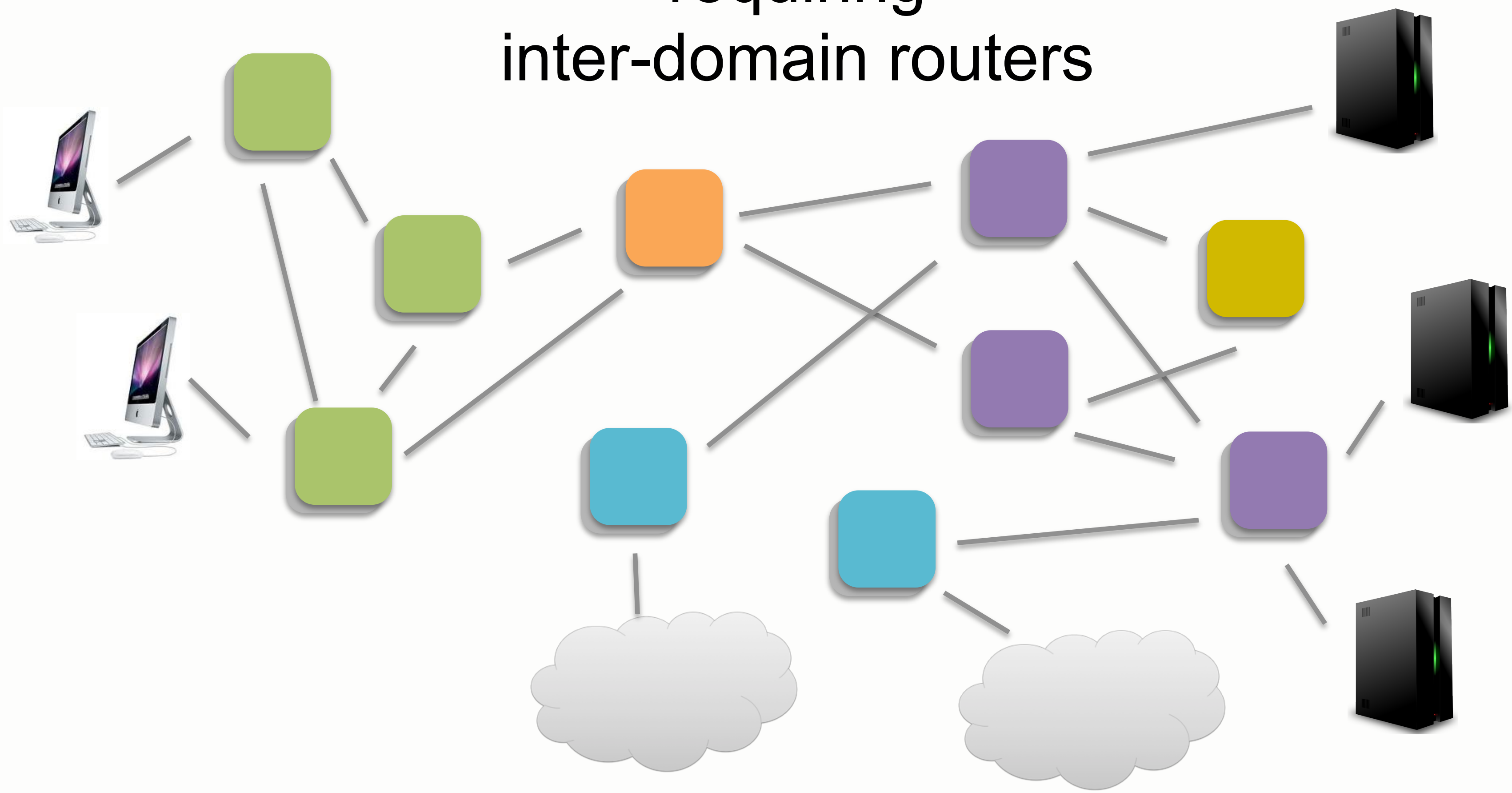
What is Networking?

there are other ISPs



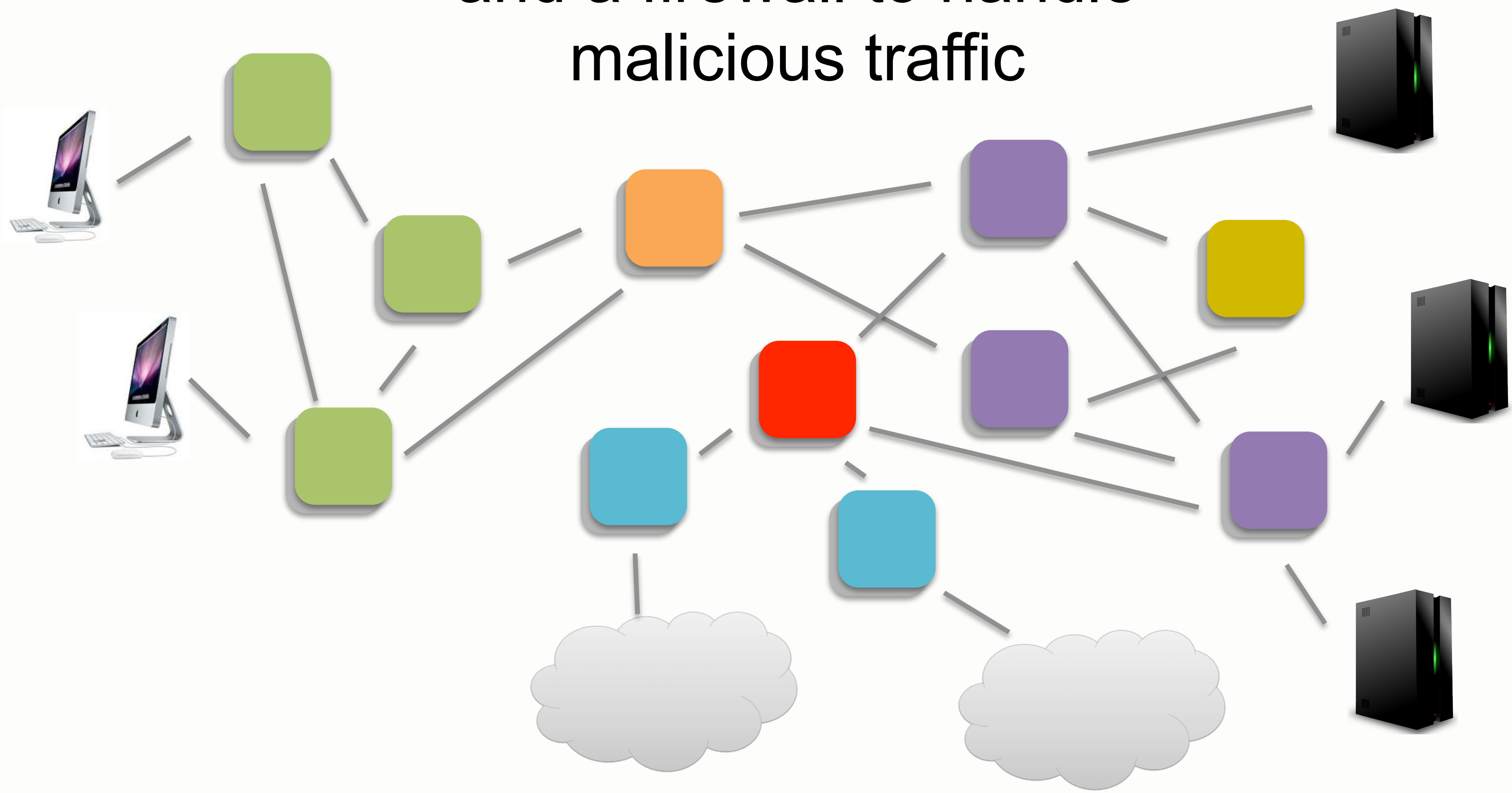
What is Networking?

requiring
inter-domain routers



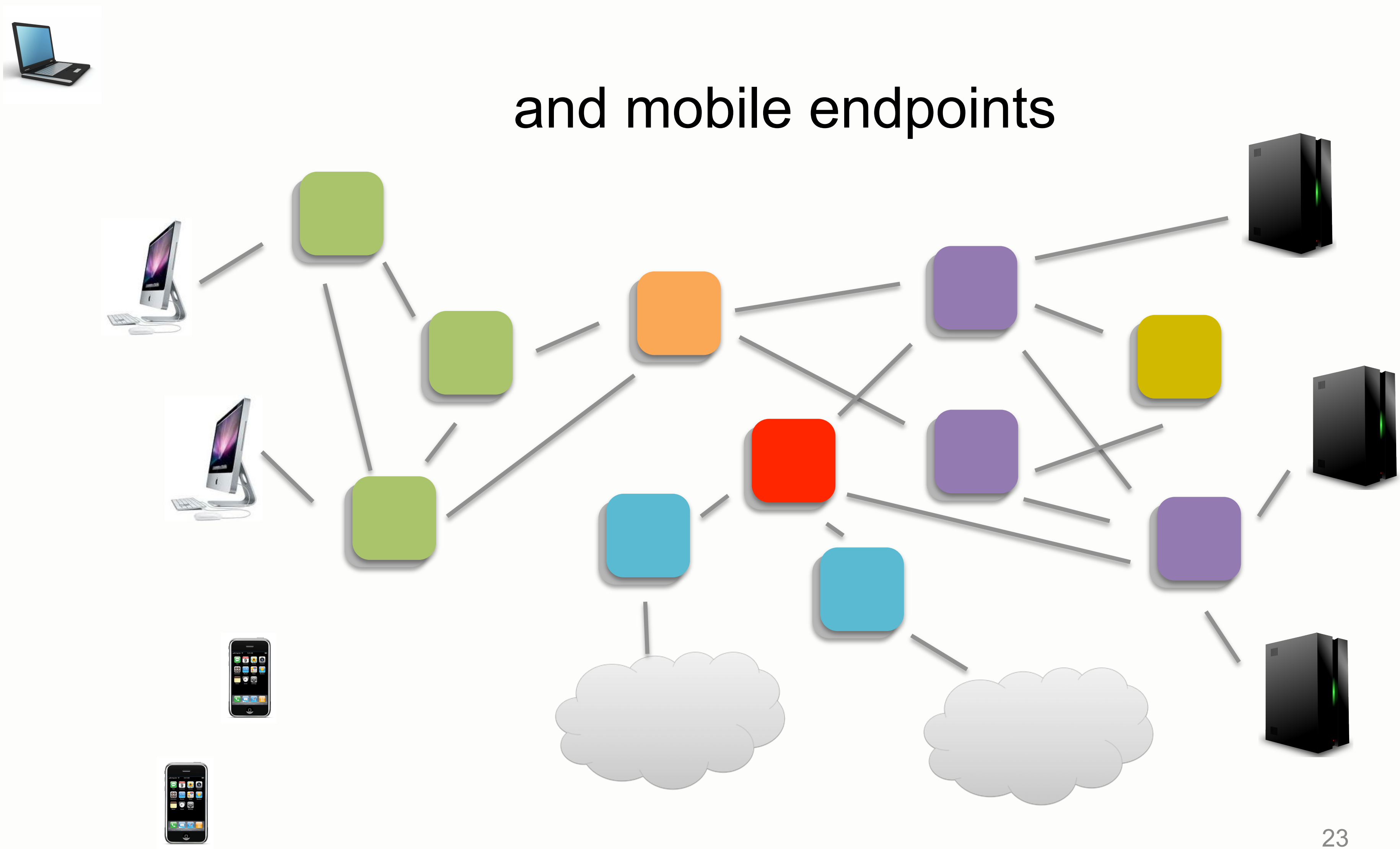
What is Networking?

and a firewall to handle
malicious traffic

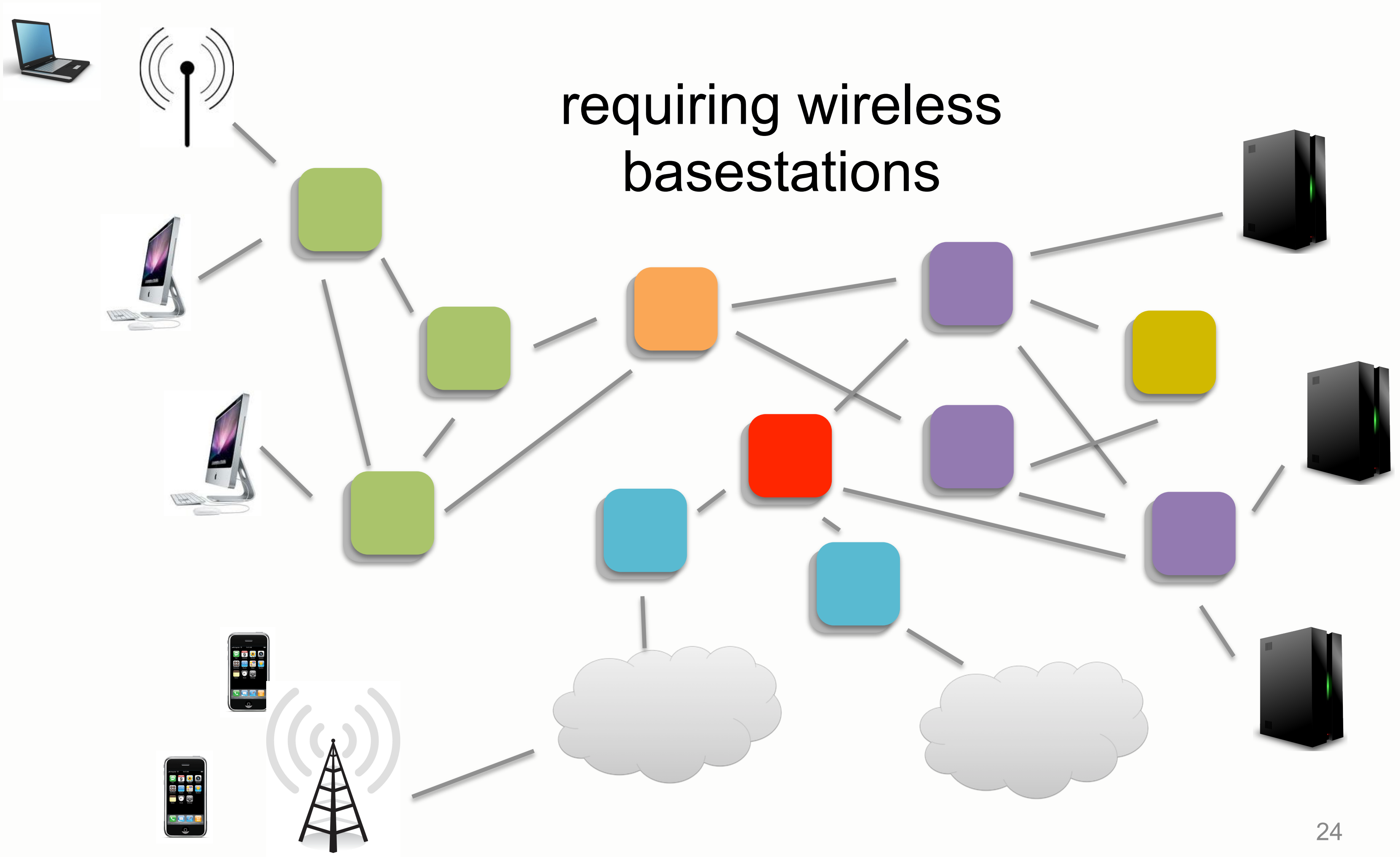


What is Networking?

and mobile endpoints

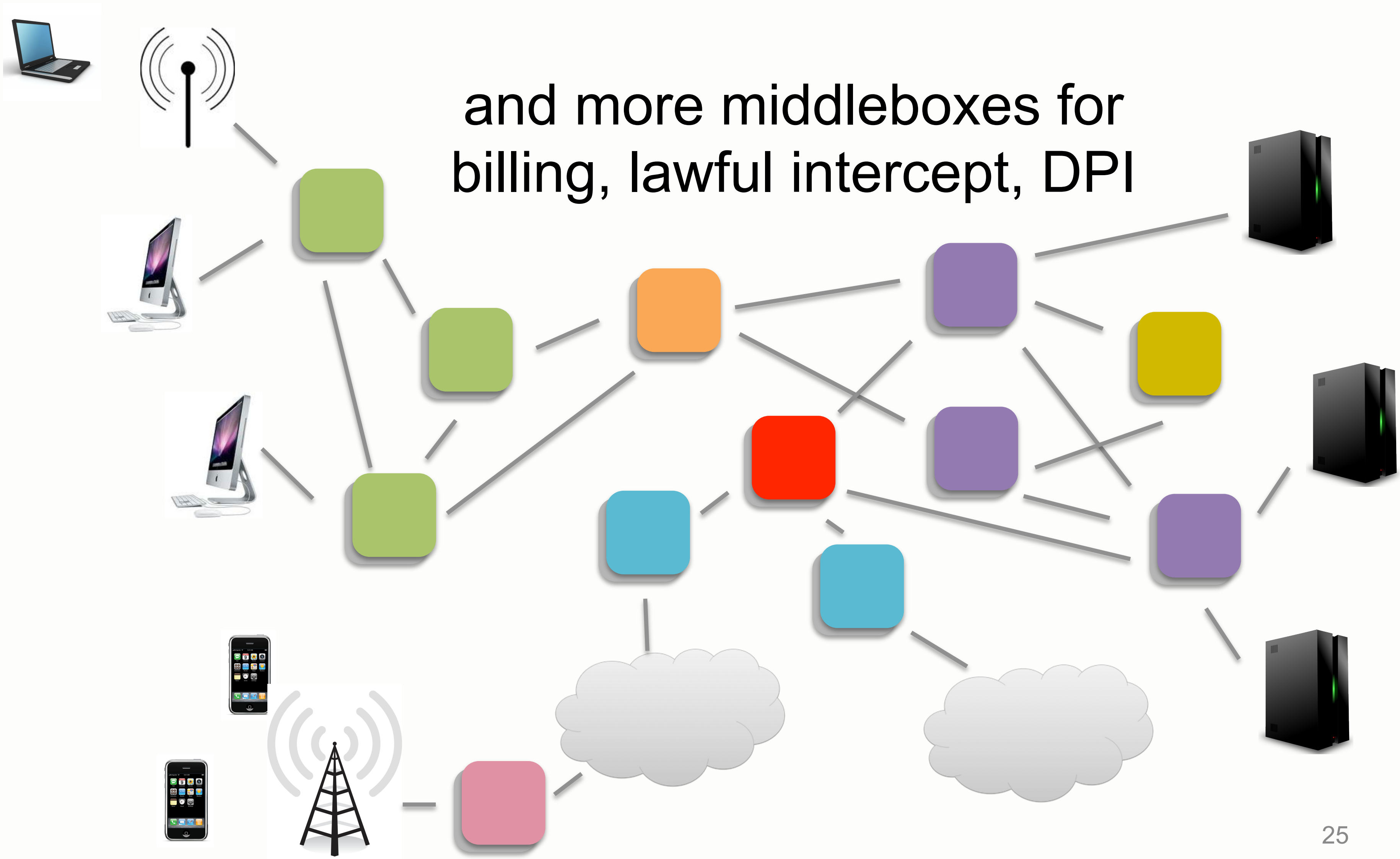


What is Networking?



What is Networking?

and more middleboxes for billing, lawful intercept, DPI



A Quick Story Circa 2009 @ Princeton

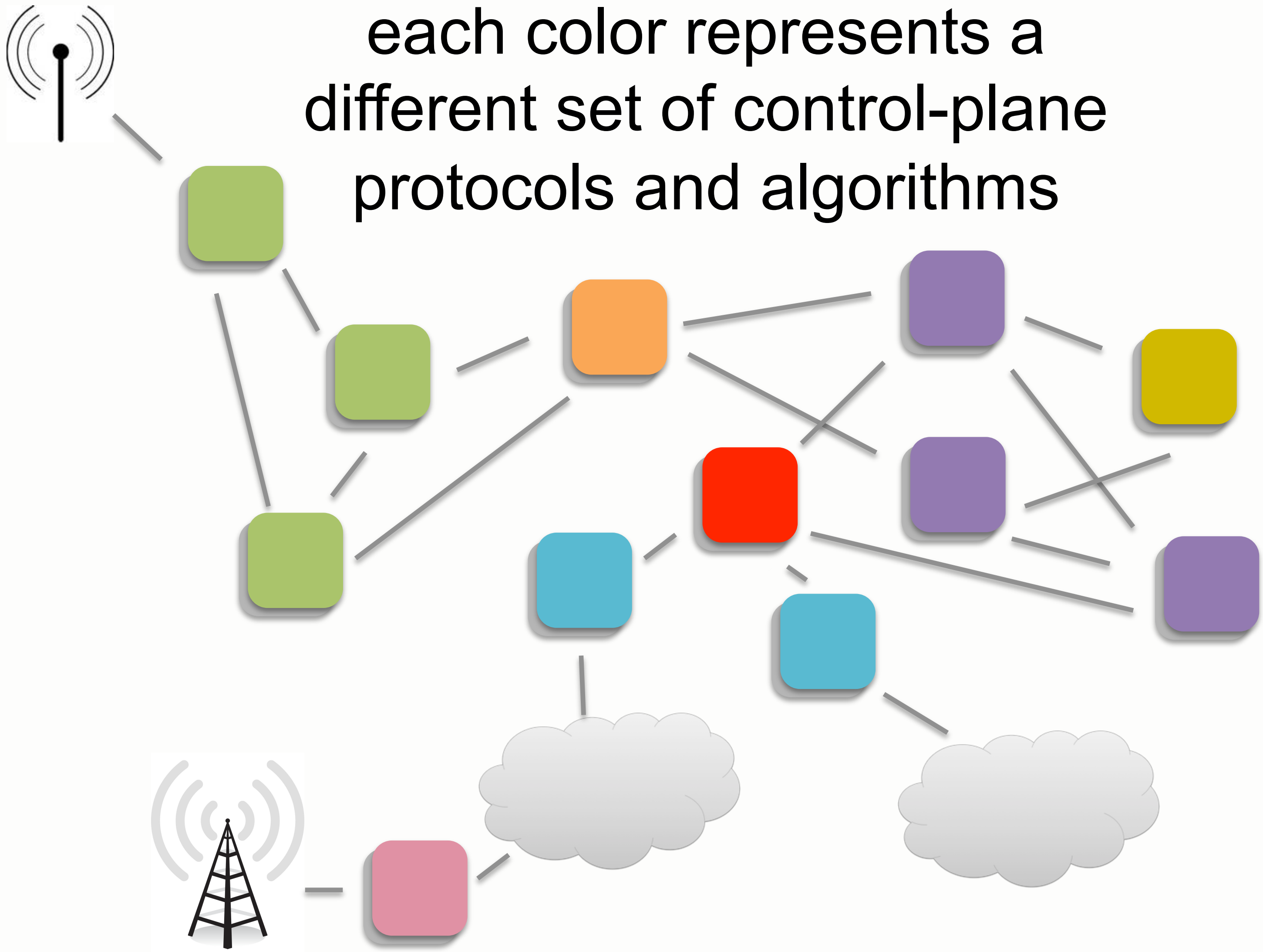
Dave:
??? Lambda calculus is easier.

Jen:
:-) Big mess, eh?

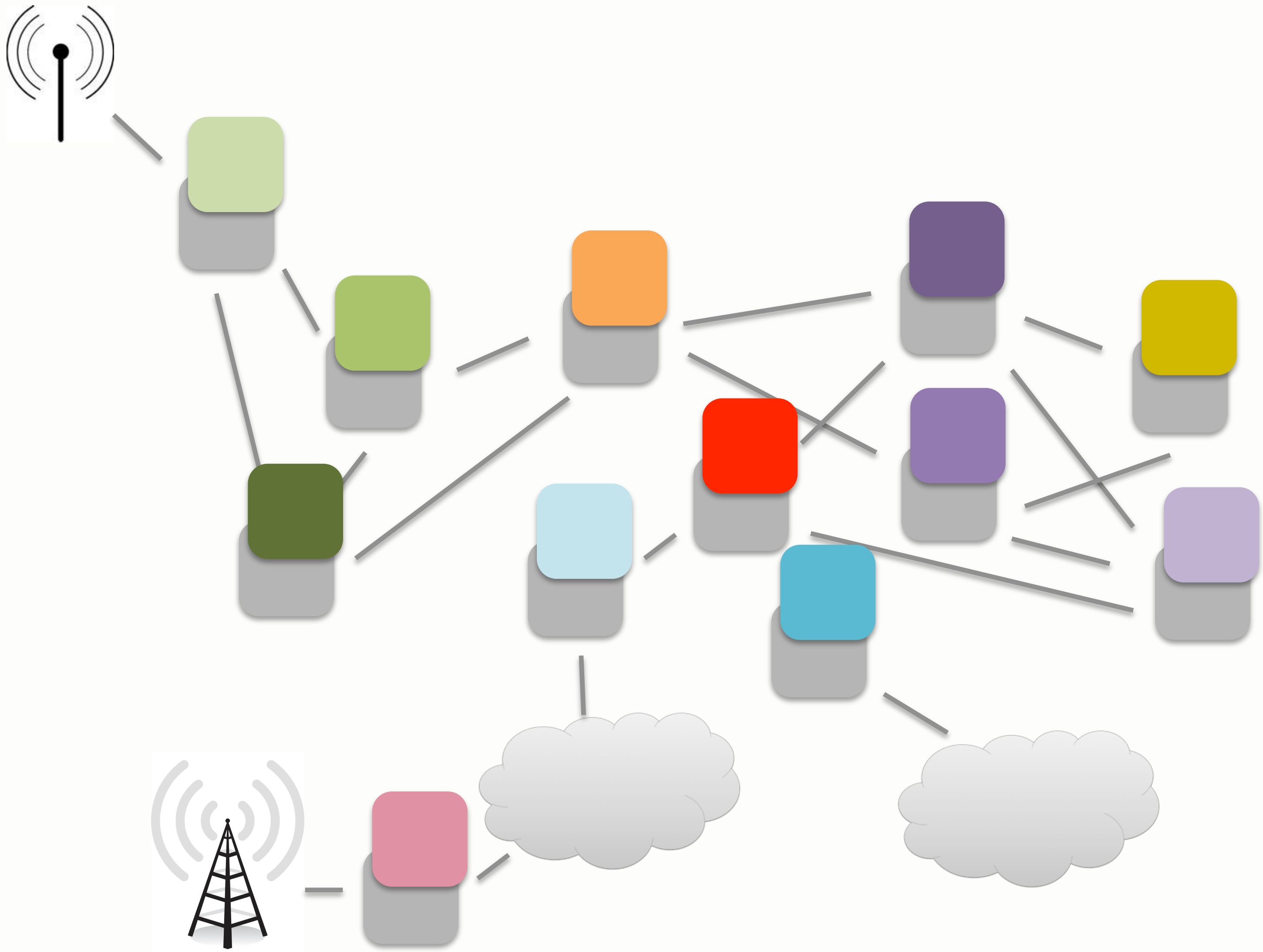
… but there is a new way to do things …

This is a Control Plane Issue

each color represents a different set of control-plane protocols and algorithms

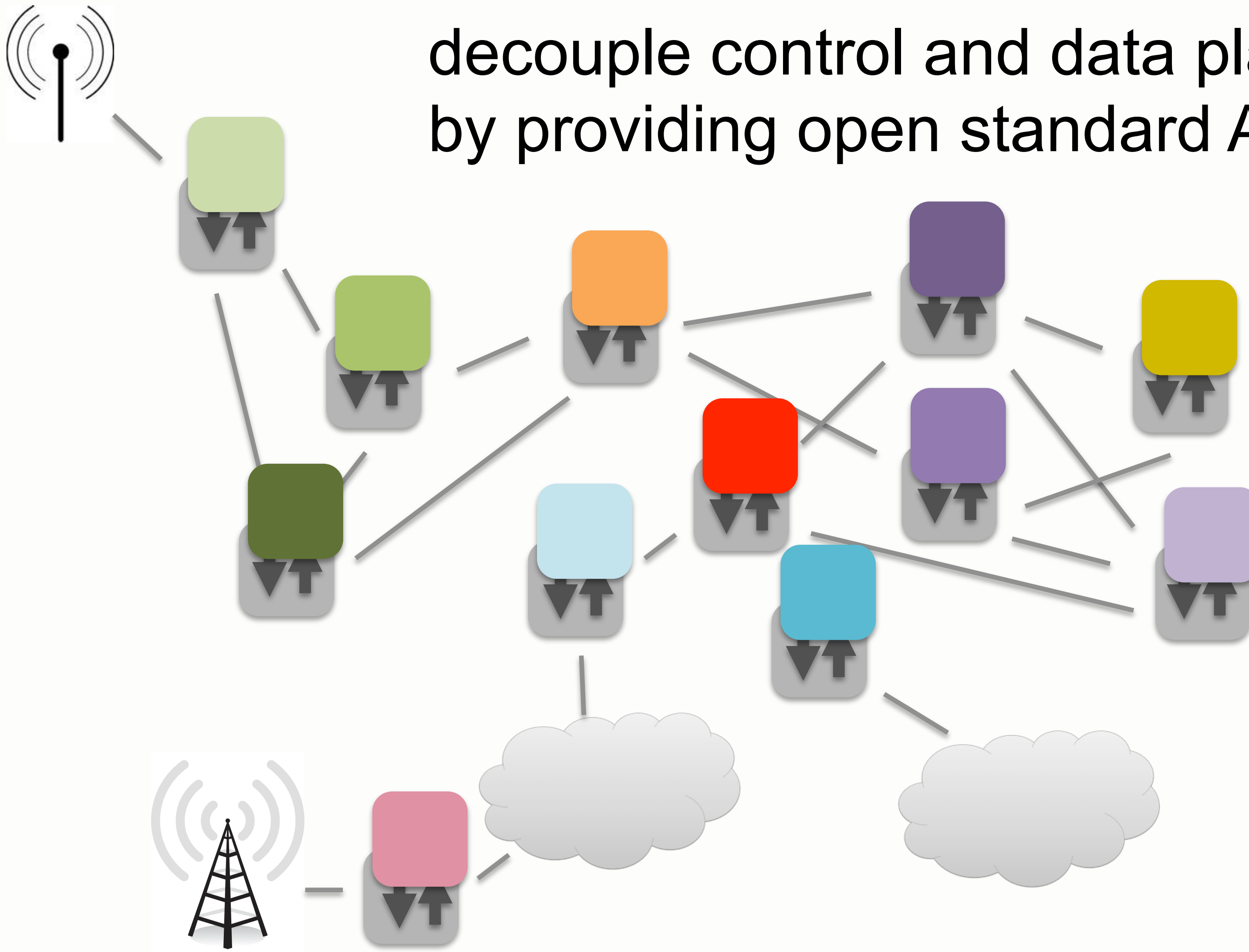


The Data Planes are Similar

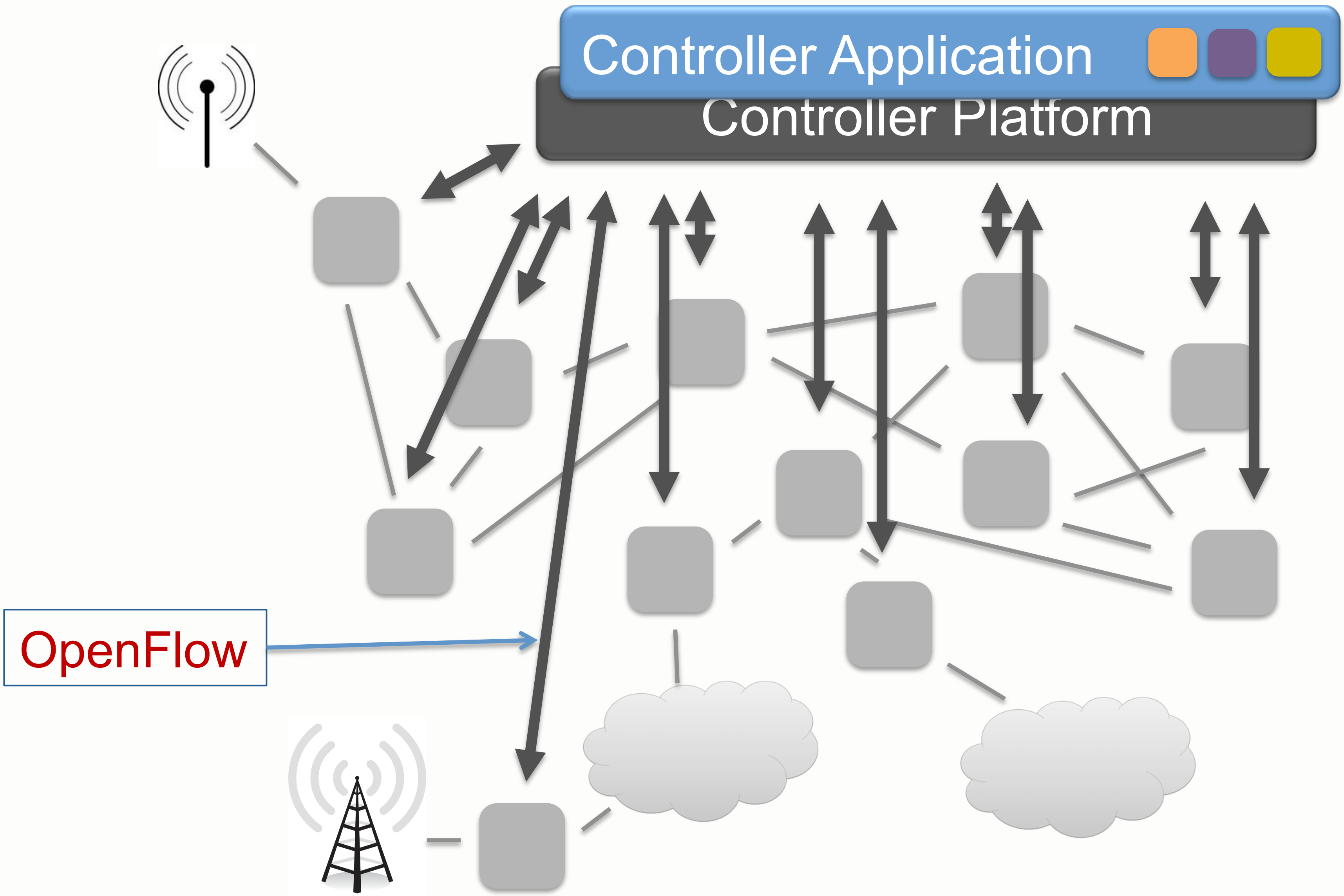


Software Defined Networks

decouple control and data planes
by providing open standard API

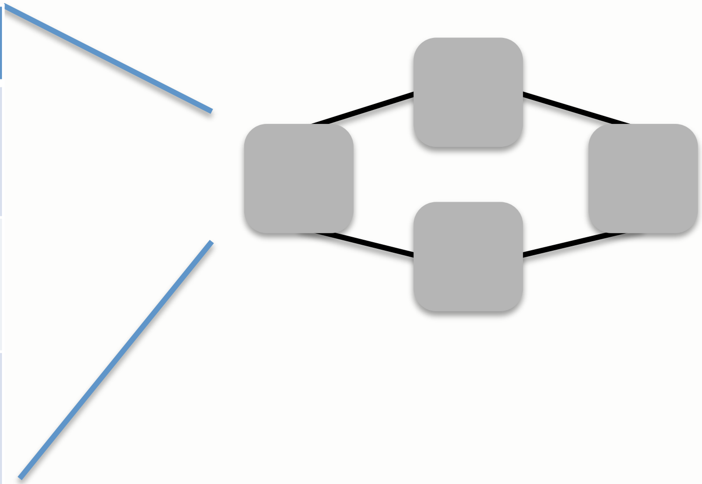


Centralize Control



OpenFlow Data Plane Abstraction

Pattern	Action	Priority	Counters
srcip = 1.2.*, dstip = 3.4.5.*	drop	1	76
srcip = *.*.*.* dstip = 3.4.5.*	fwd 2	2	13
srcip = *.*.*.* dstip = *.*.*.*	controller	3	22



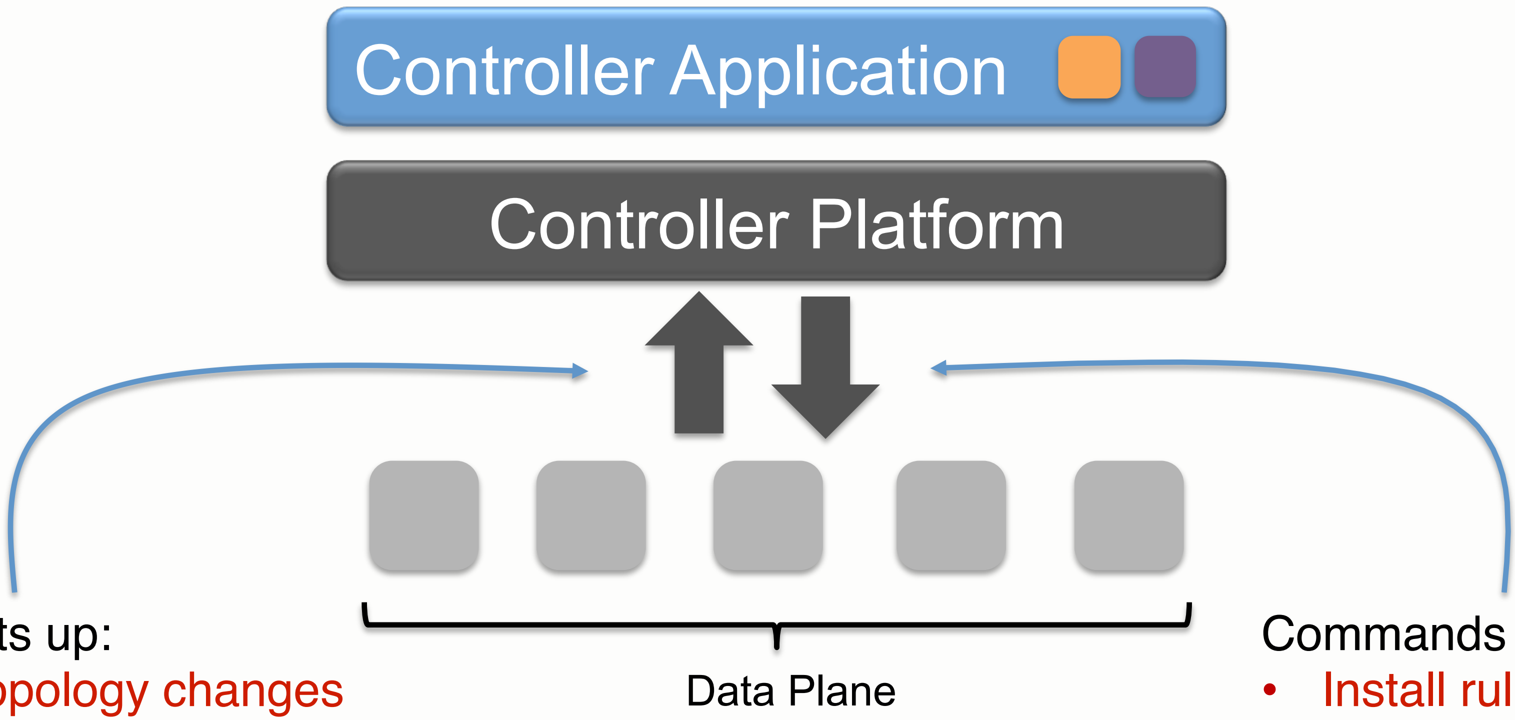
Operations:

– Install rule
– Uninstall rule
– Ask for counter values

The Payoff:

– Simplicity
– Generality

OpenFlow



Events up:

- Topology changes
- Traffic statistics
- Unprocessed arriving packets

Commands down:

- Install rule
- Uninstall rule
- Query statistics
- Send packets

The Payoff

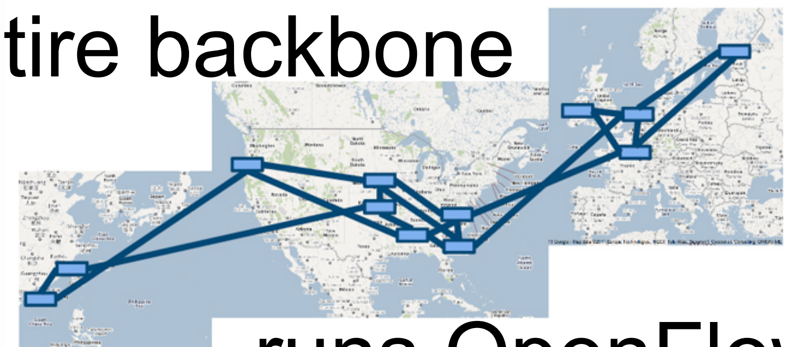
Simple, open interface:

- Easy to learn: *Even I can do it!*
- Enables rapid innovation by academics and industry
- Everything in the data center can be optimized
  - The network no longer "gets in the way”
- Commoditize the hardware

Huge Momentum in Industry

OPEN NETWORKING FOUNDATION

Entire backbone



runs OpenFlow

Bought for **$1.2 x 10^9** (mostly cash)

A Quick Story Circa 2009 @ Princeton

Jen:
So … SDN is a big deal.

Dave:
Cool. Let's get this party started.

The PL Perspective:

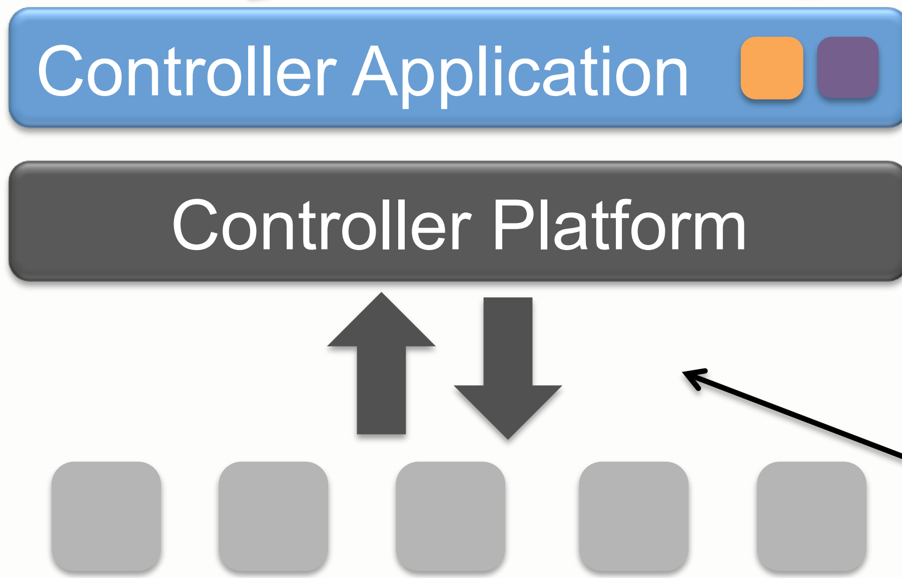
A new piece of our critical infrastructure is now available for programming

24-7 availability:

- correct-by-construction abstractions
- defect detection
- verification
- testing
- fault tolerance

multi-component applications:

- modularity
- composition
- abstraction
- information hiding



A new kind of heterogeneous distributed system

shared/used by multiple entities

- security

simple, clean, narrow interface:

- a new assembly language
- … needing domain-specific abstractions

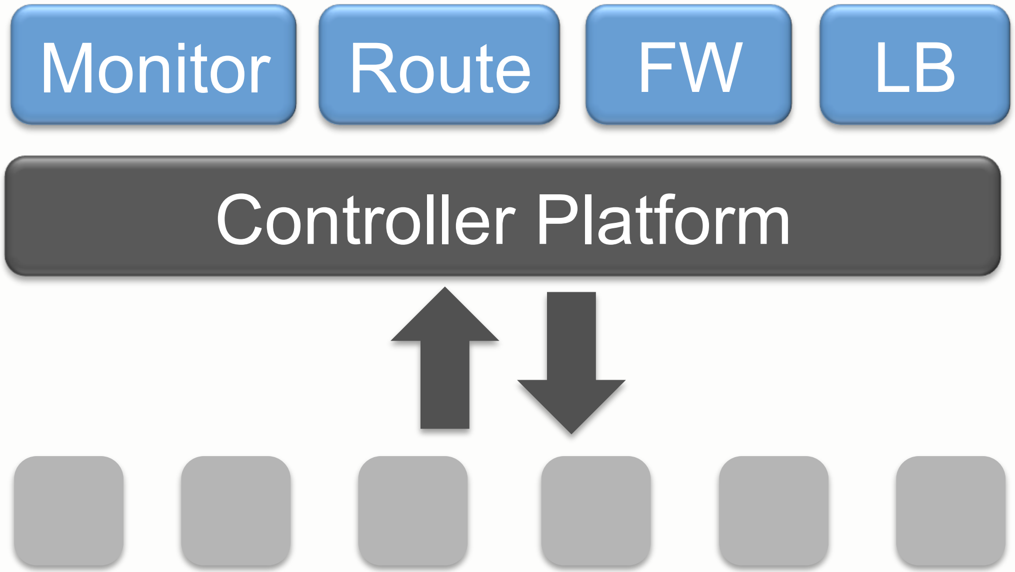
resource constraints:

- optimization problems

www.frenetic-lang.org

A DSL for modular network configuration [ICFP 11, POPL 12, NSDI 13, POPL 14, NSDI 15]

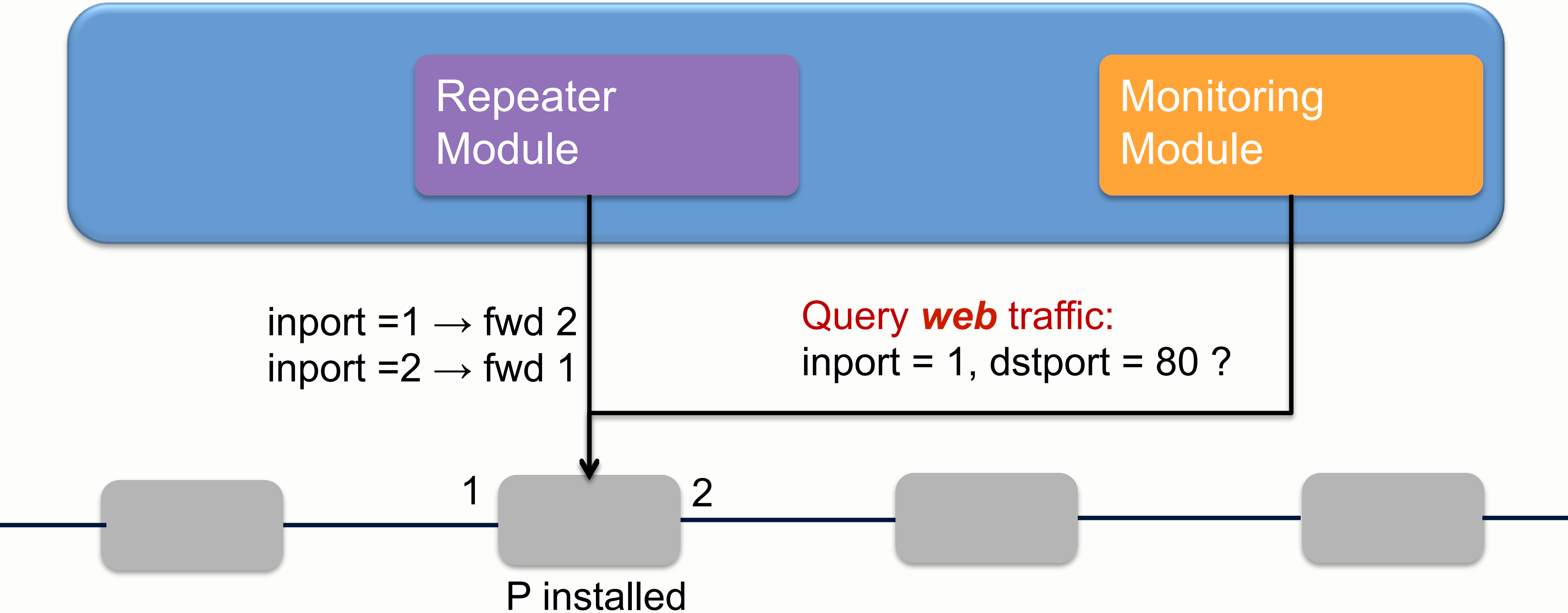
The Biggest Problem: Modularity



We still need all the functionality of old networks:
The only way to engineer it is through modular design.

OpenFlow is Anti-Modular

Controller Application



Bottom Line: It doesn't work:

- repeater rules are too coarse-grained for desired monitoring
- installing new monitoring rules will clobber the repeater actions

Anti-Modularity: A Closer Look

Repeater

```
def switch_join(switch):
  repeater(switch)

def repeater(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  install(switch,pat1,DEFAULT,None,[output(2)])
  install(switch,pat2,DEFAULT,None,[output(1)])
```

Web Monitor

```
def monitor(switch):
  pat = {in_port:2,tp_src:80}
  install(switch, pat, DEFAULT, None, [])
  query_stats(switch, pat)

def stats_in(switch, xid, pattern, packets, bytes):
  print bytes
  sleep(30)
  query_stats(switch, pattern)
```

Repeater/Monitor

```
def switch_join(switch)
  repeater_monitor(switch)

def repeater_monitor(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  pat2web = {in_port:2, tp_src:80}
  Install(switch, pat1, DEFAULT, None, [output(2)])
  install(switch, pat2web, HIGH, None, [output(1)])
  install(switch, pat2, DEFAULT, None, [output(1)])
  query_stats(switch, pat2web)

def stats_in(switch, xid, pattern, packets, bytes):
  print bytes
  sleep(30)
  query_stats(switch, pattern)
```

blue = from repeater
red = from web monitor
green = from neither

OpenFlow is Anti-Modular

You can't (easily and reliably) compose:

- a billing service with a repeater
- a firewall with a switch
- a load balancer with a router
- one broadcast service with another
- policy for one data center client with another

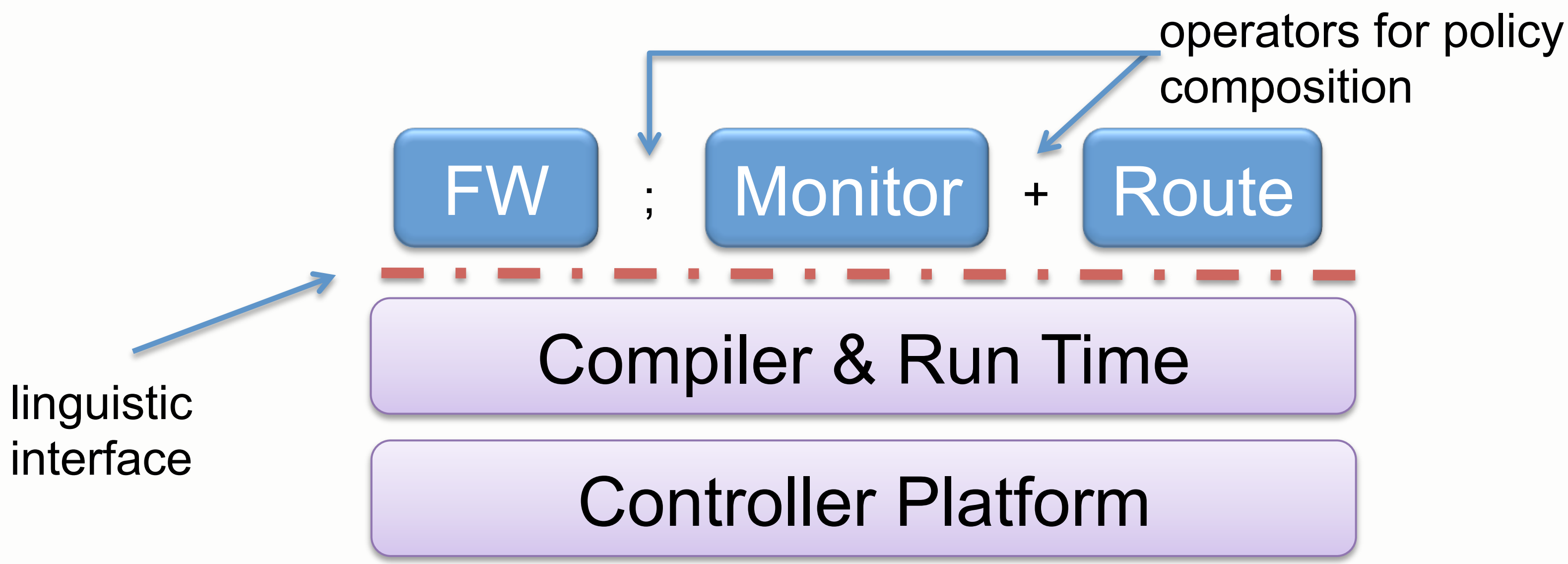
Solution: Functional Programming!

Stop thinking imperatively:

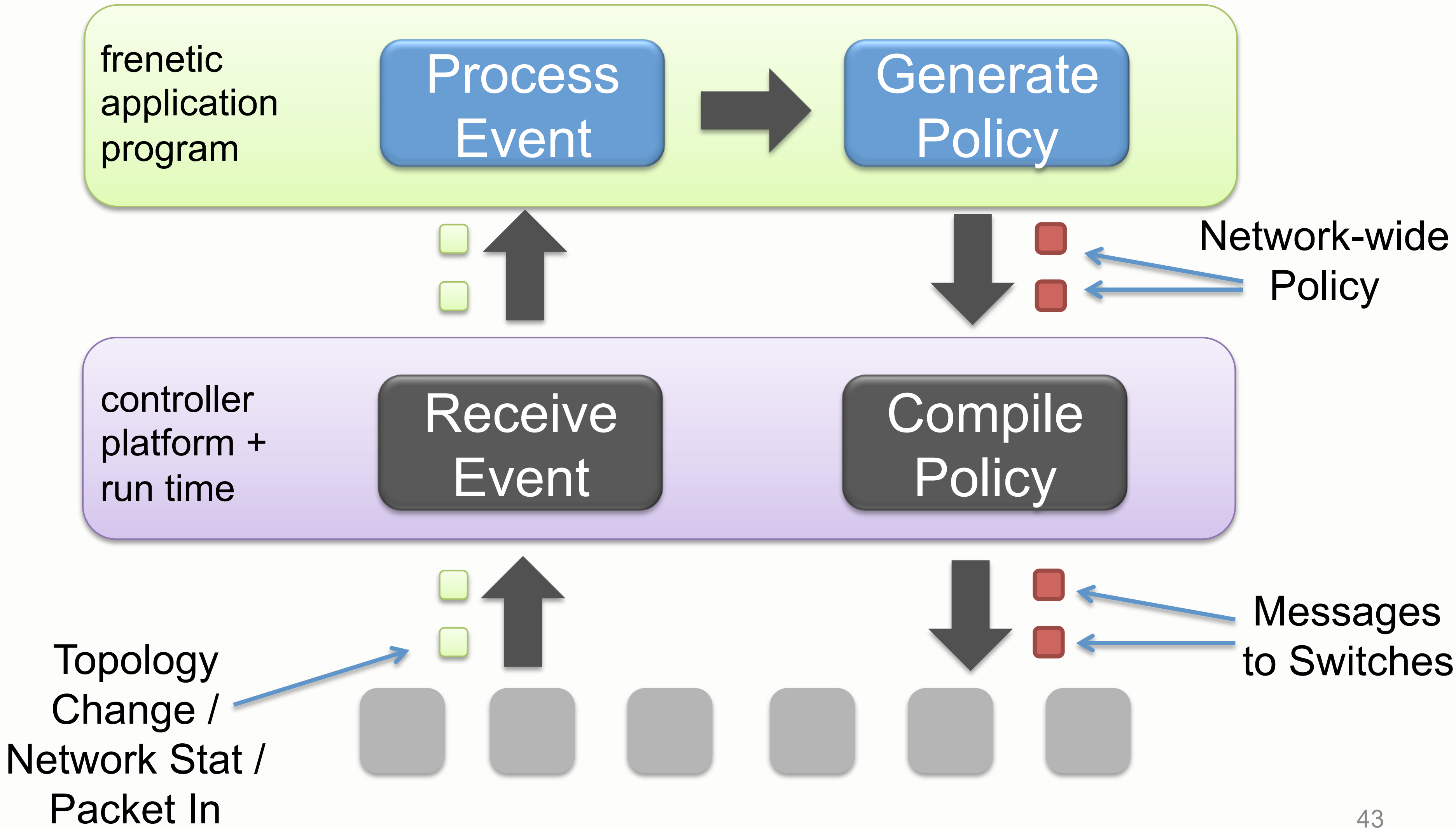
- Don't program with update/delete commands for *concrete* rules

And lift the level of abstraction:

- Use *pure functions as data structures* that describe network forwarding policy
- Provide primitives to build complex policies from simple ones
- Let a compiler and run-time do rule synthesis & installation



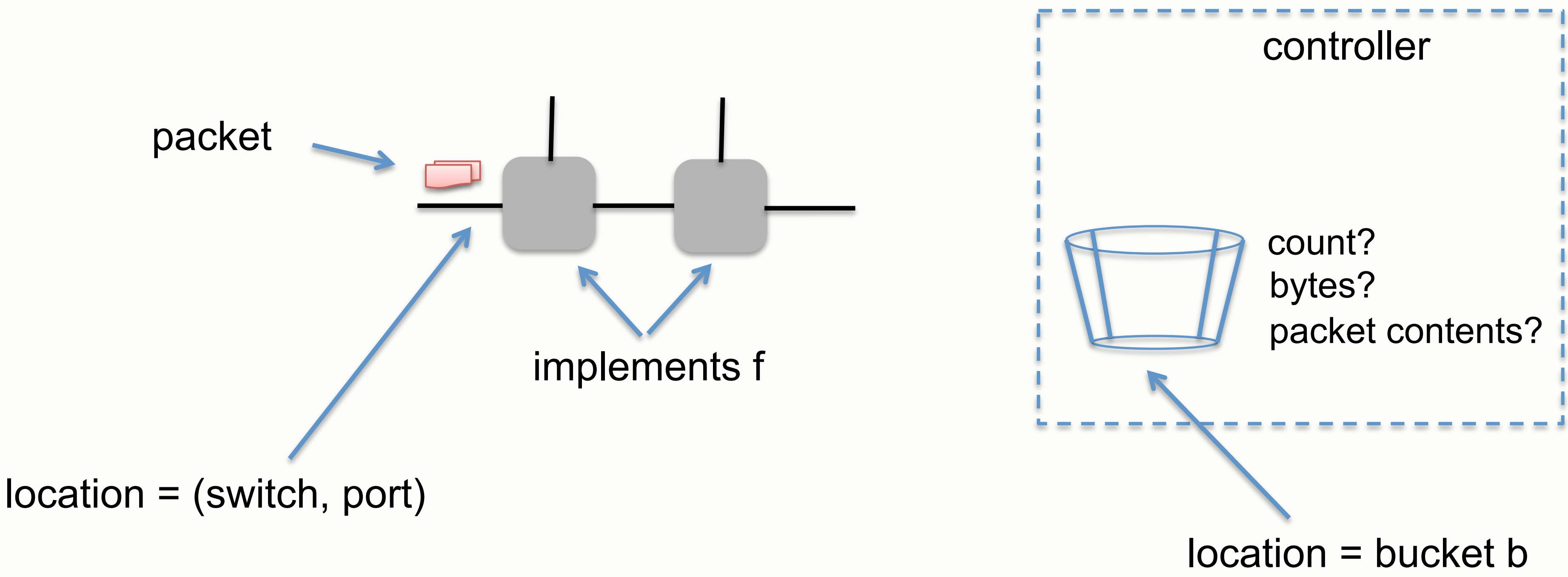
Frenetic Architecture



Frenetic Policy Language [Phase 1]

Rather than managing (un)installation of concrete rules, programmers specify what a network does using *pure functions.*

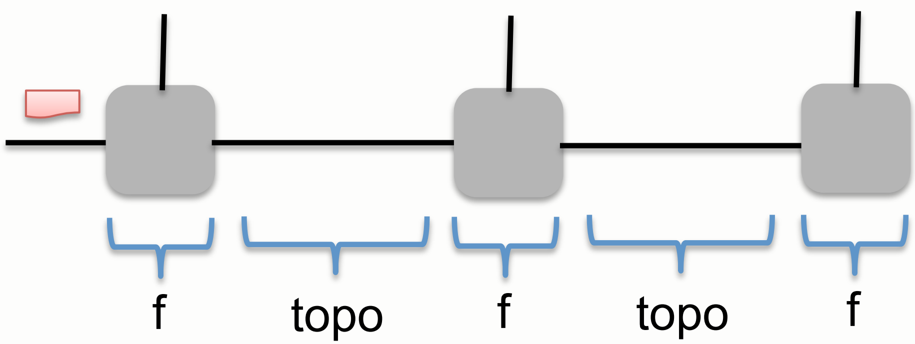
f : located_packet → located_packet set



Frenetic Policy Language [Phase 1]

Rather than managing (un)installation of concrete rules, programmers specify what a network does using *pure functions.*

f : located_packet → located_packet set



network execution

Firewalls: The Simplest Policies

Policy	Explanation	Function
false	drops all packets	fun p -> { }
true	admits all packets	fun p -> { p }

srcIP=10.0.0.1	admits packets with srcIP = 10.0.0.1
	drops others
		fun p ->
		  if p.srcIP = 10.0.0.1 then
		    { p }
		  else
		    { }

q1 /\ q2,
q1 \/ q2,
~q
	admits packets satisfying
	q1 /\ q2,
	q1 \/ q2,
	~q
		fun p -> (q1 p) U (q2 p)
		fun p -> (q1 p) Π (q2 p)
		fun p ->
		  match (q1 p) with
		    | { } -> { p }
		    | _ -> { }

Firewalls: The Simplest Policies

Example: Block all packets from source IP 10.0.0.1 and 10.0.0.2 and except those for web servers

Solution: ~(srcIP=10.0.0.1 /\ srcIP=10.0.0.2) \/ tcp_src_port = 80

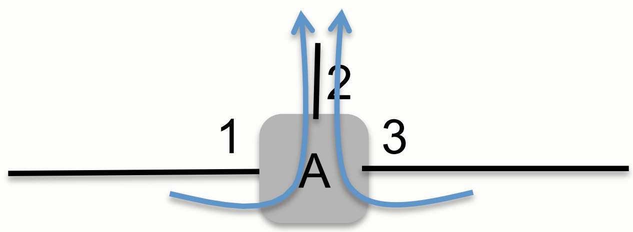
web traffic sent here

Firewalls: The Simplest Policies

Example: Allow traffic coming in to switches A, port 1 and switch B, port 2 to enter our network. Block others.

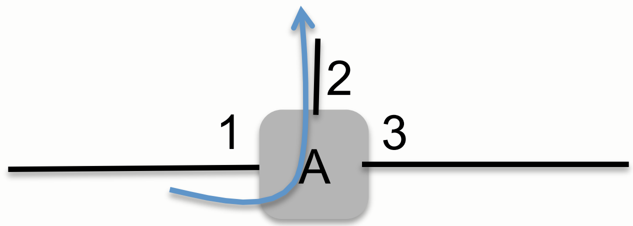
Solution: (switch=A /\ inport=1) \/ (switch=B & inport=2)

Moving Packets from Place to Place



Policy	Explanation	Function
fwd 2	forward all packets out port 2	fun p -> { p[port:= 2] }

Combining Policies



Policy	Explanation
port=1; fwd 2	only consider packets with port = 1 then forward all such packets out port 2

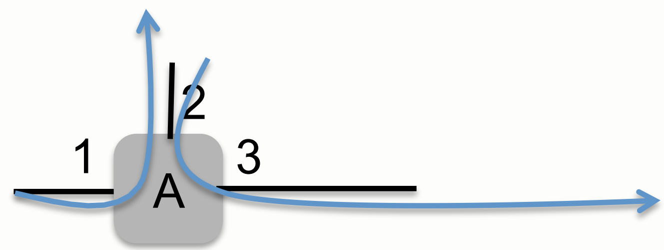
Function

```
let filter_port x p = if p.port = x then { p } else { } in
let fwd x p = p.port <- x in
(filter_port 1) <> (fwd 2)
```

where:

```
a <> b = fun packet ->
  let s = a packet in
  Set.Union (Set.map b s)
```

Multiple Flows



<u>Policy</u>

(port=1; fwd 2) +
(port=2; fwd 3)

<u>Explanation</u>

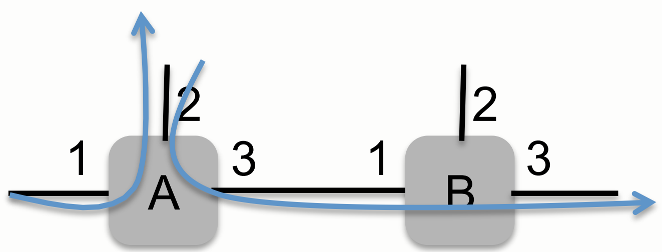
(if port = 1 then forward out port 2) and also
(if port = 1 then forward out port 2)

<u>Function</u>

(filter_port 1 <> fwd 2) +
(filter_port 2 <> fwd 3)

where:
(+) a b = fun packet ->
  Set.Union
    {(a packet),
     (b packet)}

Composing Policies



Policy	Explanation
let policyA =	
(port=1; fwd 2) +	(if port = 1 then forward out port 2) and also
(port=2; fwd 3)	(if port = 1 then forward out port 3)
let policyB =	
port=2; fwd 3	(if port = 1 then forward out port 3)
(switch = A; policyA) +	(if switch=A then policyA) and also
(switch = B; policyB)	(if port = 1 then policyB)

More Composition: Routing & Monitoring

```
router =
    dstip = 1.2.* ; fwd 1
  + dstip = 3.4.* ; fwd 2
```

```
monitor =
    srcip = 5.6.7.8 ; bucket b1
  + srcip = 5.6.7.9 ; bucket b2
```

Route on dest prefix

Monitor on source IP

```
app = monitor + router
```

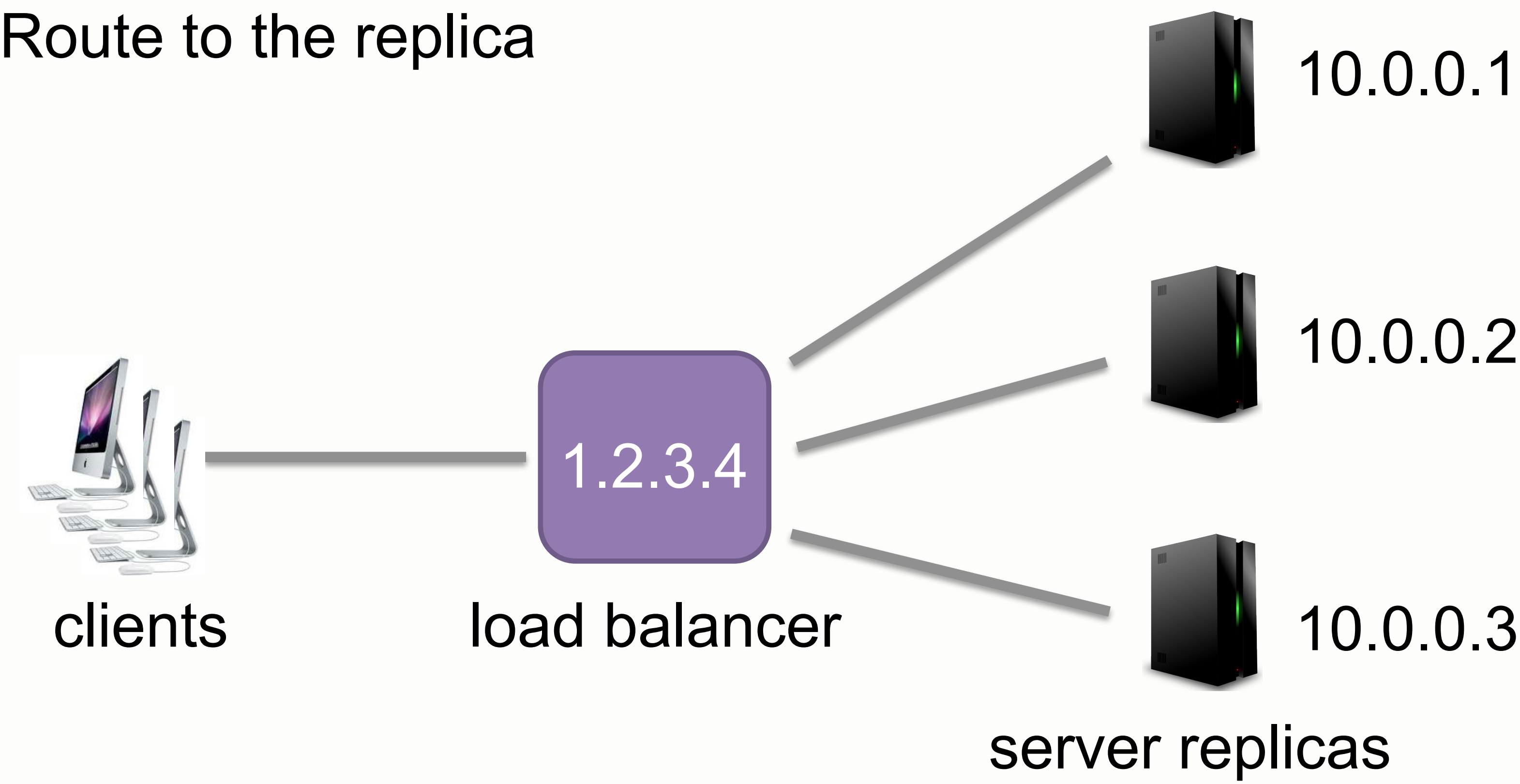
Server Load Balancing

Goal: Spread client traffic over server replicas

Setup: Advertise public IP address for the service

First: Split traffic on client IP & rewrite the server IP address

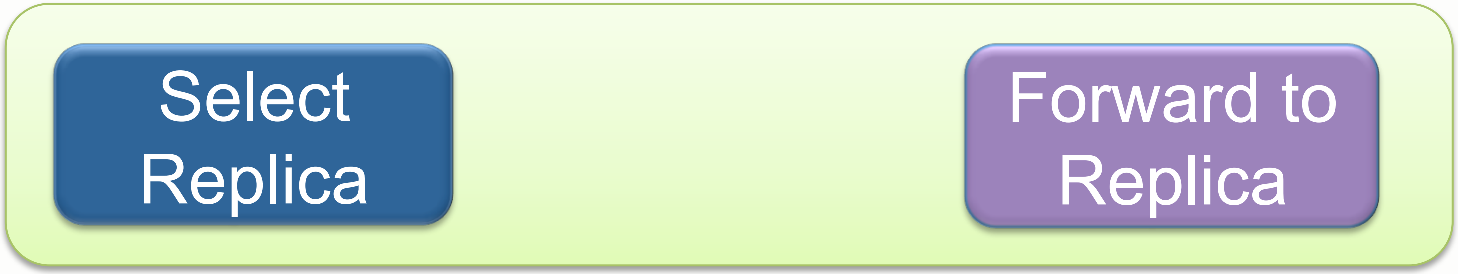
Then: Route to the replica



Sequential Composition

```
selector =
   srcip = 0* /\ dstip=1.2.3.4;
     dstip <- 10.0.0.1
+
   srcip = 1* /\ dstip=1.2.3.4;
     dstip <- 10.0.0.2
```

```
forwarder =
   dstip = 10.0.0.1; fwd 1
+
   dstip = 10.0.0.0; fwd 2
```




```
lb = selector ; forwarder
```

Summary So Far

predicates:

```
q ::=  f = pattern
     | true
     | false
     | q1 /\ q2
     | q1 \/ q2
     | ~q
```

simple actions:

```
a ::=  fwd n
     | f <- v
     | bucket b
```

network policies:

```
p ::=  a              (action)
     | q              (filter)
     | p1 + p2        (parallel comp.)
     | p1 ; p2        (sequential comp.)
```

abbreviations:

```
if q then p1 else p2  == (q; p1) + (~q; p2)

id     == true
drop == false
fwd p == port <- p
```

Equational Theory

A sign of a well-conceived language == a simple equational theory

P + Q	==	Q + P	(+ commutative)
(P + Q) + R	==	P + (Q + R)	(+ associative)
P + drop	==	P	(+ drop unit)
(P ; Q) ; R	==	P ; (Q ; R)	(; associative)
id ; P	==	P	(; id left unit)
P ; id	==	P	(; id right unit)
drop ; P	==	drop	(; drop left zero)
P ; drop	==	drop	(; drop right zero)
(if q then P else Q) ; R	==	if q then (P ; R) else (Q ; R)	(if commutes ;)

A Simple Use Case

(Modular Reasoning)

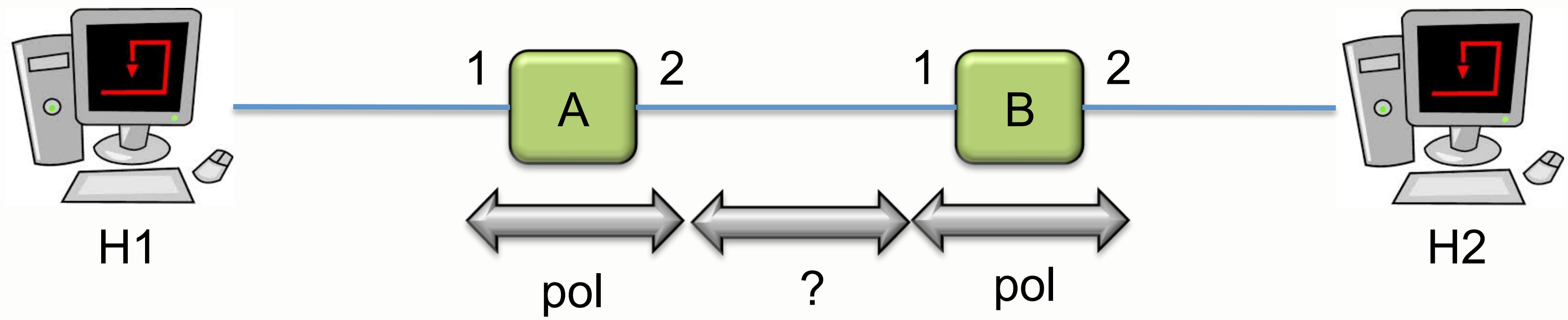
```
firewall =
  if srcip = 1.1.1.1 then
     drop
  else
     id
```

```
router = ...
```

```
app = firewall ; router
```

```
app == firewall ; router
     == (if srcip = 1.1.1.1 then drop else id) ; router
     == if srcip = 1.1.1.1 then (drop ; router) else (id ; router)
     == if srcip = 1.1.1.1 then drop else (id ; router)
     == if srcip = 1.1.1.1 then drop else router
```

But what if we want to reason about entire networks?



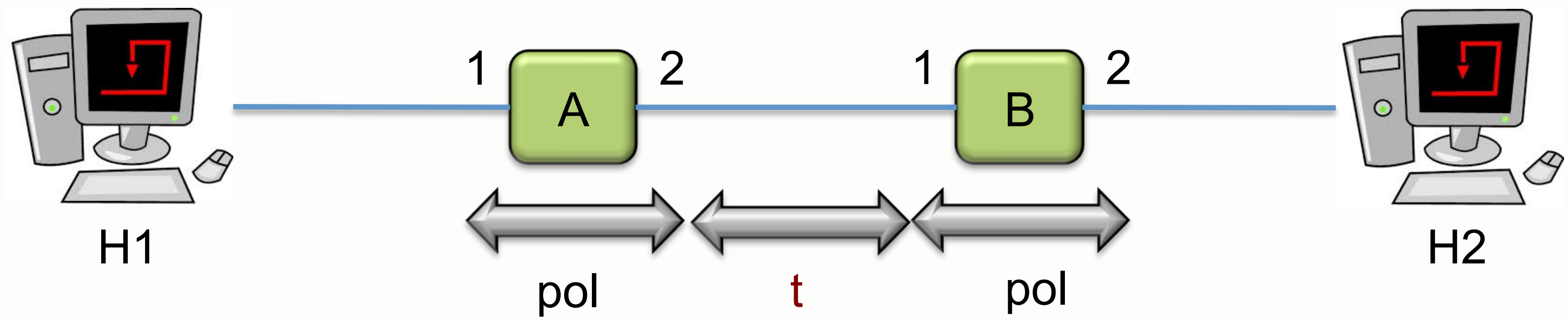

```
polA = ...
polB = ...
pol = switch=A; polA +
      switch=B; polB
```

Are all SSH packets dropped at some point along their path?

Do all non-SSH packets sent from H1 arrive at H2?

Are the optimized policies equivalent to the unoptimized one?

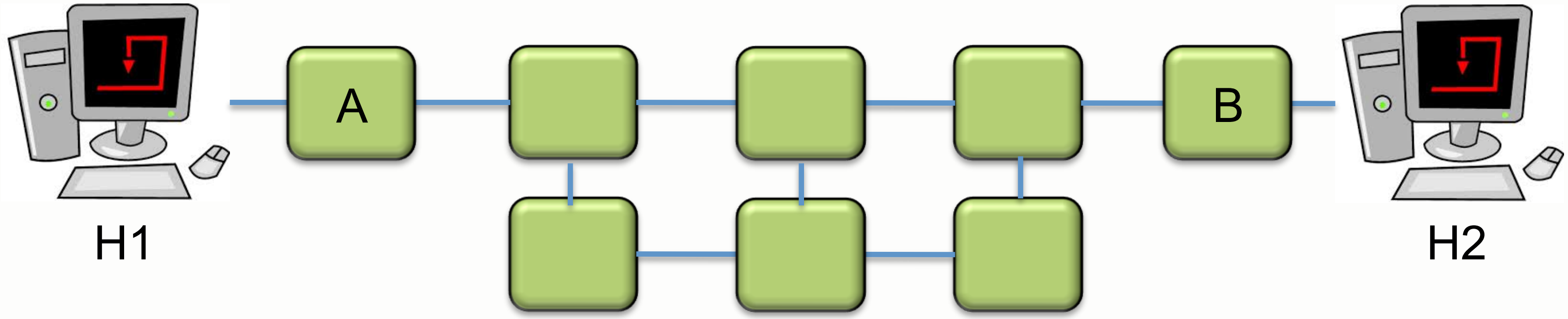
Encoding Topologies




```
t =
   (sw = A /\ pt = 2; sw <- B; pt <- 1)
+
   (sw = B /\ pt = 1; sw <- A; pt <- 2)
```

```
net = pol; t; pol
```

Encoding Topologies

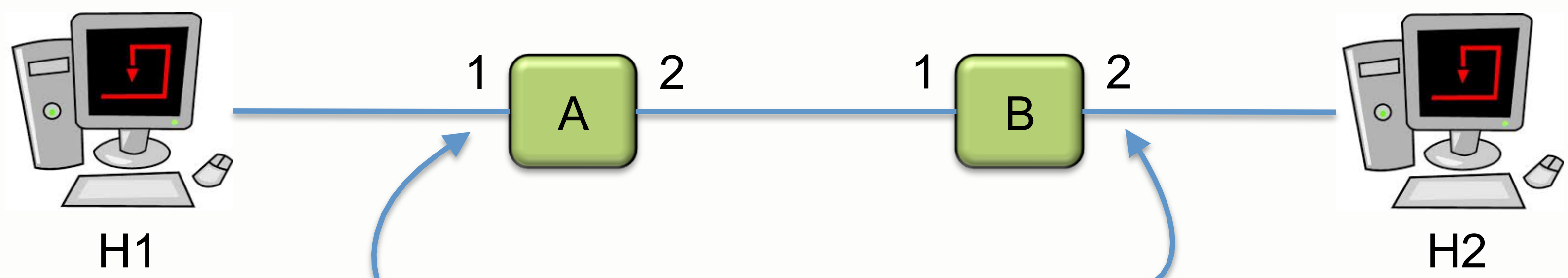


t = ...

net = (pol; t)*; pol

Kleene iteration:
p* = id + p + p;p + ...

Encoding Networks




```
pol  = ...
t     = ...
net = (pol; t)*; pol
```

net is a function that moves packets:

A1 ==> B2

B2 ==> A1

```
edge = sw=A & pt=1
         || sw=B & pt=2

net = edge; (ac; t)*; ac; edge
```

and also moves packets:

A1 ==> A2

A2 ==> A1

B1 ==> B2

B2 ==> B1

Summary So Far

Policies

```
p, q, r ::=
  a            // filter according to a
| f <- v       // update field f to v
| p ; q        // do p then q
| p + q        // do p and q in parallel
| p*           // do p zero or more times
```

Predicates

```
a, b, c ::=
  drop         // drop all packets
| id           // accept all packets
| f = v        // field f matches v
| ~a           // negation
| a & b        // conjunction
| a || b       // disjunction
```

Network Encoding in; (policy; topology)*; policy; out

Summary So Fa

Kleene Algebra

Boolean Algebra

p, q, r ::=

a	// filter according to a
| f <- v	// update field f to v
| p ; q	// do p then q
| p + q	// do p and q in parallel
| p*	// do p zero or more times

Predicates

a, b, c ::=

drop	// drop all packets
| id	// accept all packets
| f = v	// field f matches v
| ~a	// negation
| a & b	// conjunction
| a || b	// disjunction

Netwo

Boolean Algebra + Kleene Algebra
= Kleene Algebra with Tests

Equational Theory

net1 ≈ net2

For programmers:

- a system for reasoning about programs as they are written

For compiler writers:

- a means to prove their transformations correct

For verifiers:

- sound and complete with a PSPACE decision procedure

Equational Theory

Boolean Algebra: a & b ≈ b & a   a & ~a ≈ drop   a || ~a ≈ id   ...

Kleene Algebra: (a; b); c ≈ a; (b; c)   a; (b + c) ≈ (a; b) + (a; c)

p* ≈ id + p; p*   ...

Packet Algebra: f <- n; f = n ≈ f <- n   f = n; f <- n ≈ f = n

f <- n; f <- m ≈ f <- m

if f ≠ g: f = n; g <- m ≈ g <- m; f = n   f <- n; g <- m ≈ g <- m; f <- n

if m ≠ n: f = n; f = m ≈ drop

f = 0 + ... + f = n ≈ id (finite set of possible values in f)

Using the Theory

forward = (dst = H1; pt <- 1)
+ (dst = H2; pt <- 2)

ac = ~(typ = SSH); forward

t = ...

edge = ...

net = edge; (ac; t)*; ac; edge

Are all SSH packets dropped?

typ = SSH; net ≈ drop

Do all non-SSH packets sent from H1 arrive at H2?

~typ = SSH; sw= ... 1; p ...
≈
~typ = SSH; sw= ... 1; s ... ; pt <- 2

Using the Theory


```
forward = (dst = H1; pt <- 1)
        + (dst = H2; pt <- 2)

ac = ~(typ = SSH); forward

t = ...

edge = ...

net = edge; (ac; t)*; ac; edge
```

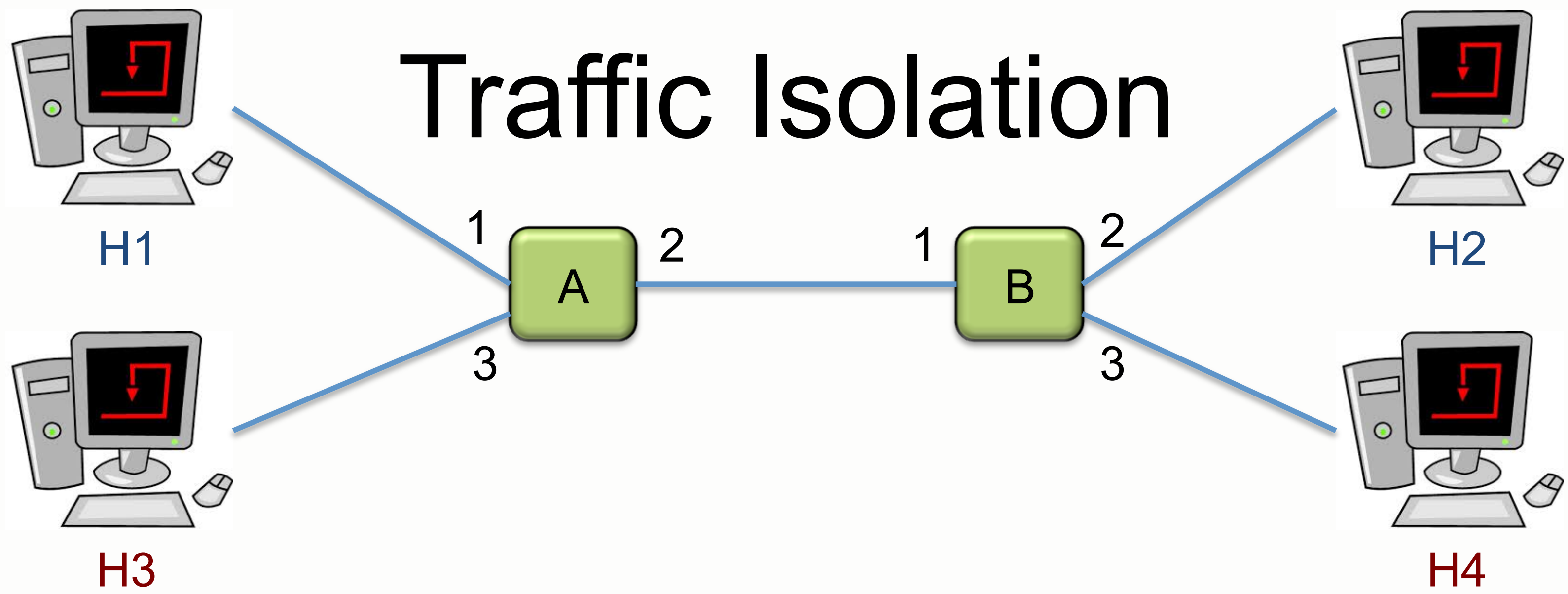
Are all SSH packets dropped?

```
typ = SSH; net  ≈  drop
```

Do all non-SSH packets destined for H2, sent from H1 arrive at H2?

```
~typ = SSH; dst = H2; sw=A; pt=1; net
≈
~typ = SSH; dst = H2; sw=A; pt=1; sw <- B; pt <- 2
```

Traffic Isolation



Programmer 1 connects H1 and H2:

```
polA1 = sw = A; (
            pt = 1; pt <- 2  +
            pt = 2; pt <- 1  )

polB1 = sw = B; ( ... )

pol1 = polA1 + polB1

net1 = (po
```

Programmer 2 connects H3 and H4:

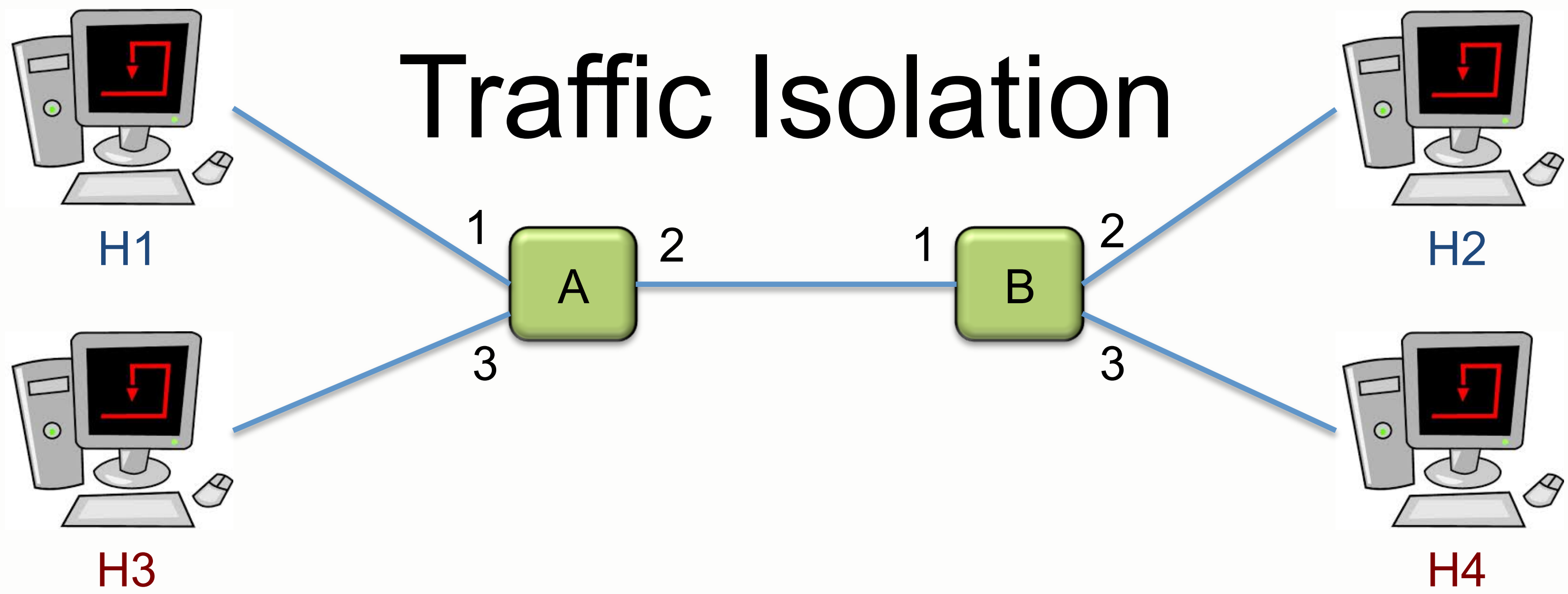
```
polA2 = sw = B; (
            pt = 3; pt <- 2  +
            pt = 1; pt <- 3  )

polB2 = sw = A; ( ... )

pol2 = polA2 + polB2
```

```
net3 = ((pol1 + pol2); t)*     // traffic from H2 goes to H1 and H4!
```

Traffic Isolation



A *network slice* is a light-weight abstraction designed for traffic isolation:

{ in } policy { out }

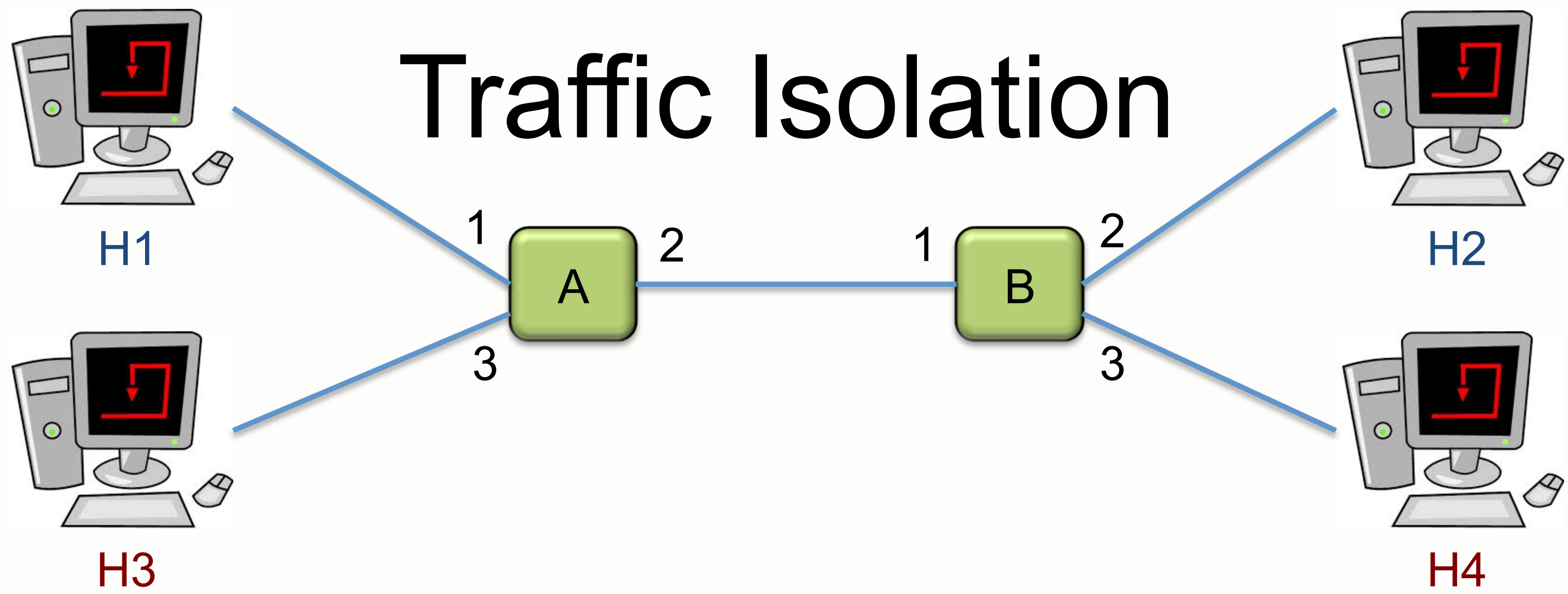
traffic outside the slice satisfying in enters the slice

traffic inside the slice obeys the policy

traffic inside the slice satis

slices are just a little syntactic sugar on top of NetKAT

Traffic Isolation



A *network slice* is a light-weight abstraction designed for traffic isolation:

edge1 = sw = A ∧ pt = 1 ∨ sw = B ∧ pt = 2

slice1 = {edge1} pol1 {edge1}

edge2 = sw = A ∧ pt = 3 ∨ sw = B ∧ pt = 3

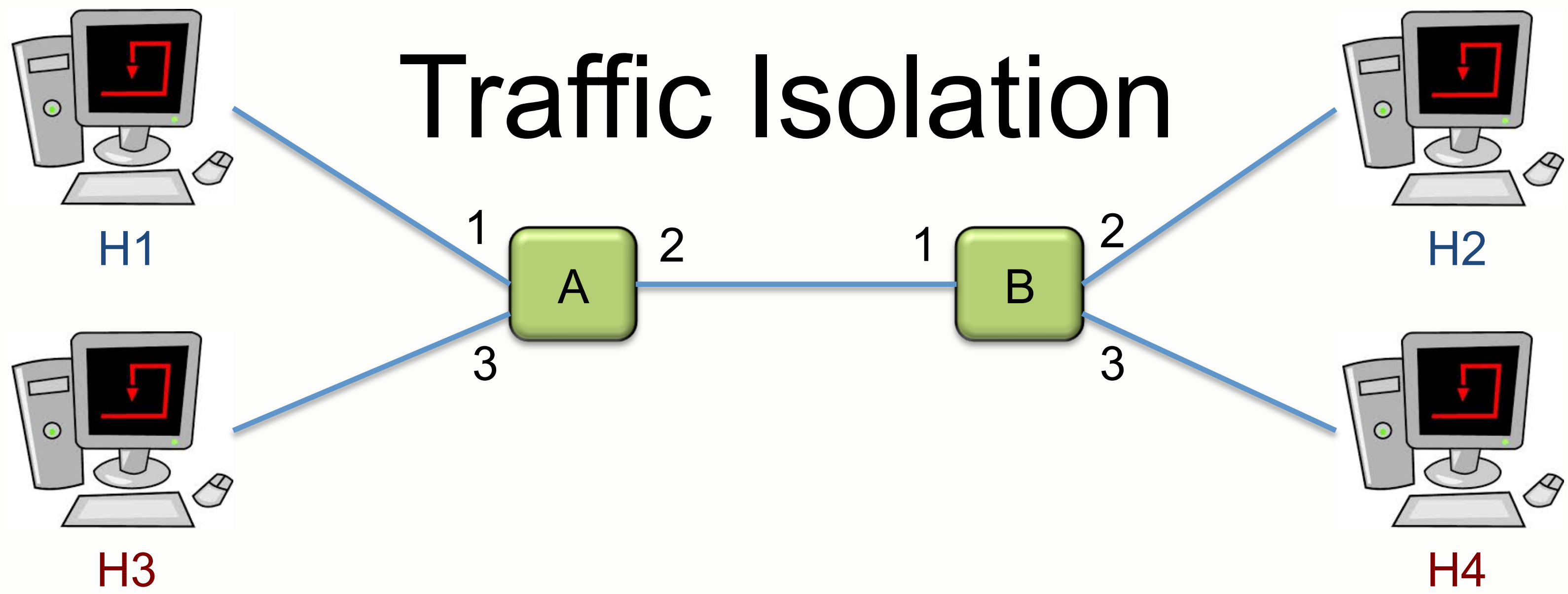
slice2 = {edge2} pol2 {edge2}

Theorem: (slice1; t)* + (slice2;t)* ≈ ((slice1 + slice2); t)*

packet copied and sent through slice1 and slice2 networks *separately*

packet runs through network that *combines* slice1 and slice2

Traffic Isolation



A *network slice* is a light-weight abstraction designed for traffic isolation:

edge1 = sw = A /\ pt = 1 \/ sw = B /\ pt = 2

slice1 = {edge1} pol1 {edge1}

edge2 = sw = A /\ pt = 3 \/ sw = B /\ pt = 3

slice2 = {edge2} pol2 {edge2}

Theorem: edge1; (slice1; t)* ≈ edge1; ((slice1 + slice2); t)*

consider those packets at the *edge1 of the slice*

can't tell the difference between *slice1 alone and slice1 + slice2*

NetKAT can be implemented with OpenFlow

```
forward =
   (dst = H1; pt <- 1)
+ (dst = H2; pt <- 2)

ac =
   ~(typ = SSH); forward
```

compile →

Flow Table for Switch 1:

Pattern	Actions
typ = SSH	drop
dst=H1	fwd 1
dst=H2	fwd 2

Flow Table for Switch 2:

Pattern	Actions
typ = SSH	drop
dst=H1	fwd 1
dst=H2	fwd 2

Theorem: Any NetKAT policy p that does not modify the switch field can be compiled in to an equivalent policy in "OpenFlow Normal Form."

Moving Forward

Multiple implementations:

- In OCaml:
  - Nate Foster, Arjun Guha, Mark Reitblatt, and others!
  - https://github.com/frenetic-lang/frenetic

See www.frenetic-lang.org

Moving Forward

Propane [SIGCOMM 2016, best paper]

- a language for configuring BGP routers
- similar abstractions to NetKAT; different compilation strategies

Synthesizing Protocols [in progress]

- abstractions for load-sensitive routing
- synthesis of load-sensitive distributed protocols

Concern	Assembly Languages		Programming Languages	
	x86	**NOX**	**ML**	**Frenetic**
Resource Management	Move values to/ from register		Declare/use variables	
Modularity	Unregulated calling conventions		Calling conventions managed automatically	
Consistency	Inconsistent memory model		Consistent (?) memory model	
Portability	Hardware dependent		Hardware independent	

Concern	Assembly Languages		Programming Languages	
	x86	NOX	Java/ML	Frenetic
Resource Management	Move values to/ from register	(Un)Install policy rule-by-rule	Declare/use variables	Declare network policy
Modularity	Unregulated calling conventions	Unregulated use of network flow space	Calling conventions managed automatically	Flow space managed automatically
Consistency	Inconsistent memory model	Inconsistent global policies	Consistent (?) memory model	Consistent global policies
Portability	Hardware dependent	Hardware dependent	Hardware independent	Hardware Independent

Summary

FUNCTIONAL NETWORK PROGRAMMERS: 326

OTHER NETWORK PROGRAMMERS: 0