Header Space Analysis
Part I

Peyman Kazemian
With James Zeng, George Varghese, Nick McKeown

Summer School on Formal Methods and Networks
Cornell University
June 2013

Recap of the last session

- Network Troubleshooting Today:
  - Manual.
  - Primitive tools.
  - Ad hoc heuristics.
  - Relies on wisdom of network admins.
- Other fields
  - Lots of intellectual ideas.
  - Multi-billion dollar tool industry.

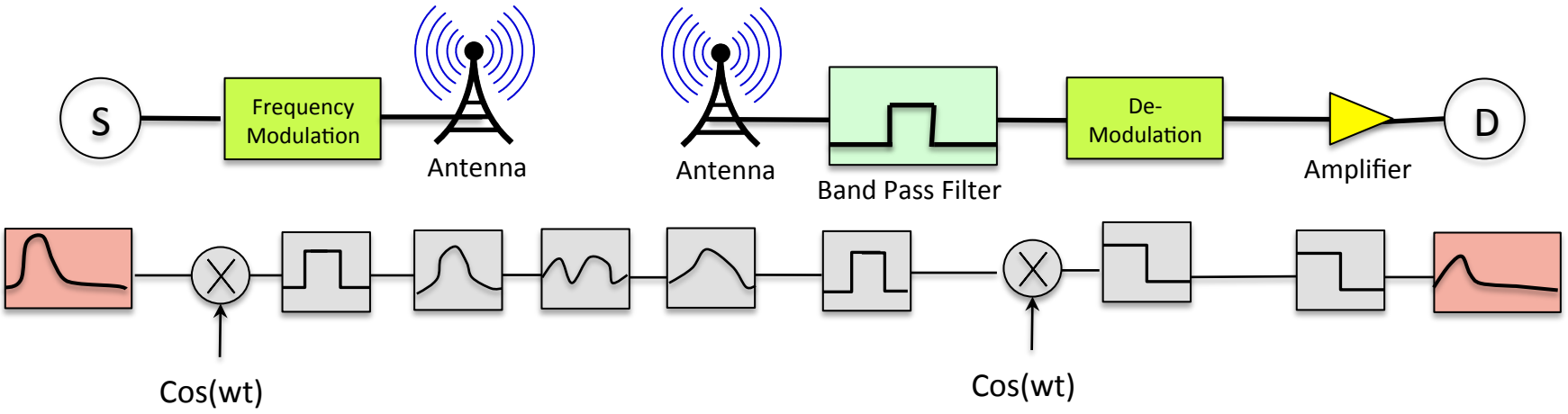
Systematic troubleshooting is hard!

- Forwarding state NOT constructed in a way that lend itself well to checking and verification!
  1. Distributed across multiple tables and boxes.
  2. Written to network by multiple independent writers (different protocols, network admins)
  3. Presented in different formats by vendors.
  4. Not directly observable or controllable.
- State-of-art: try to indirectly observe and verify network state using ping!

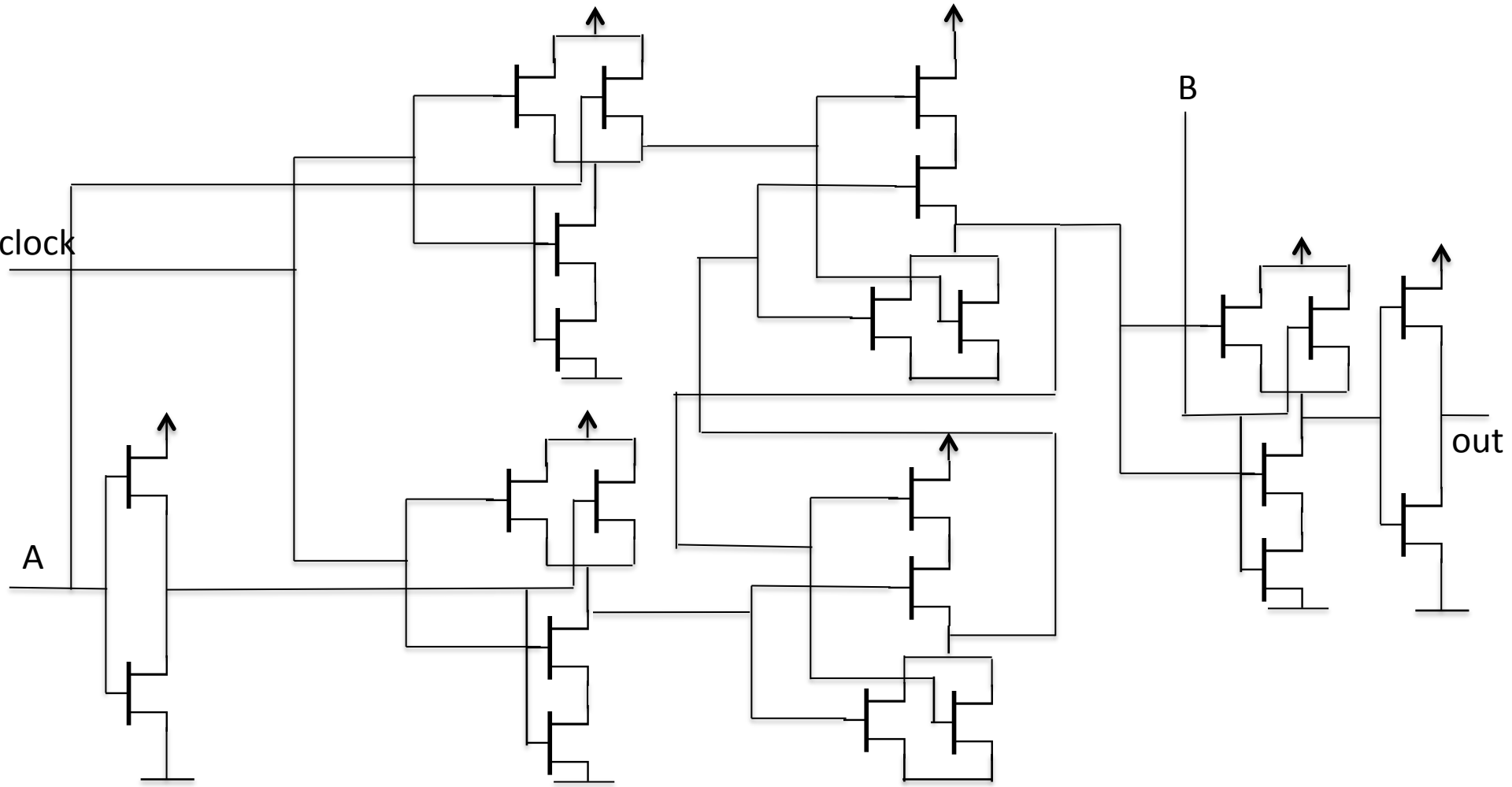
Q: How other fields overcome similar challenges and verify their systems?

A: By modeling their systems based on higher level abstractions.

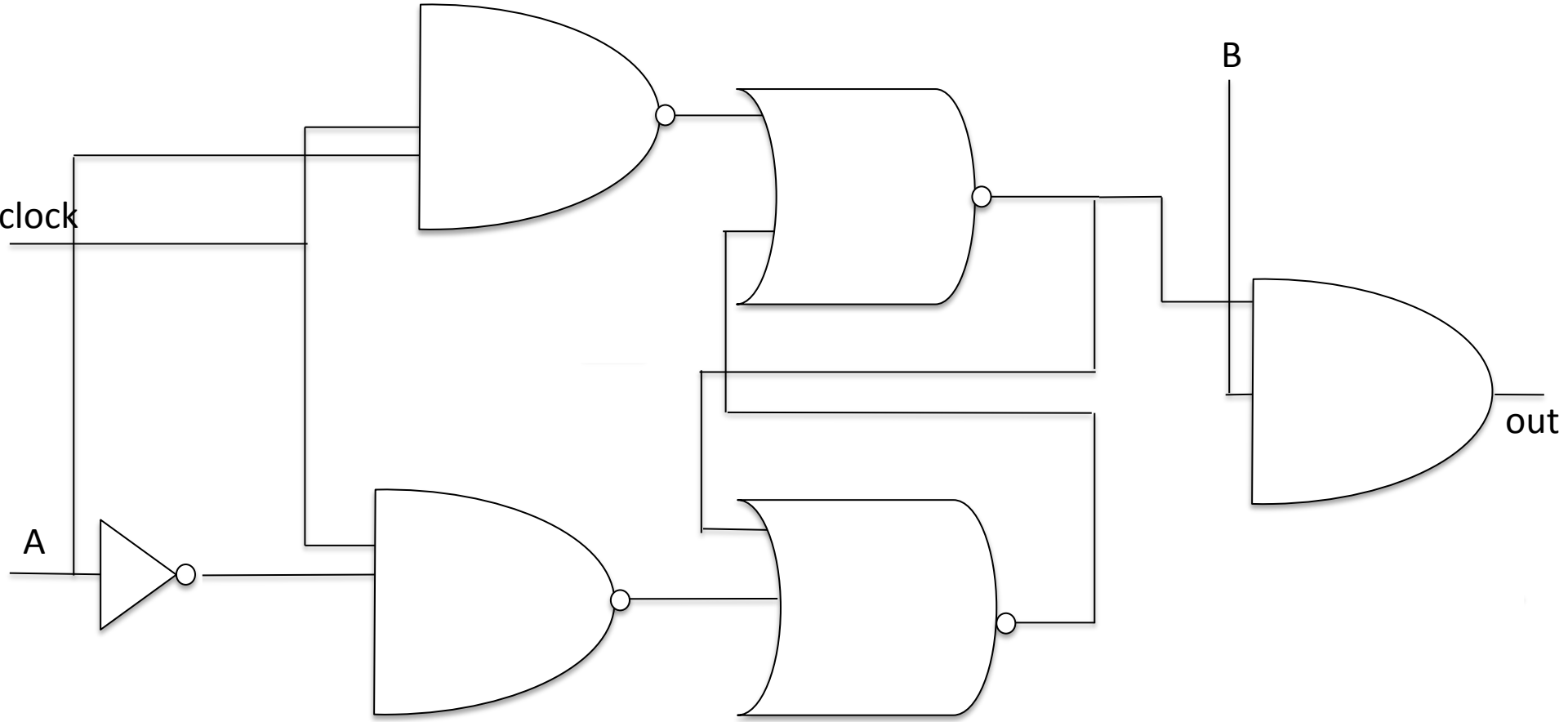
Communication Engineering



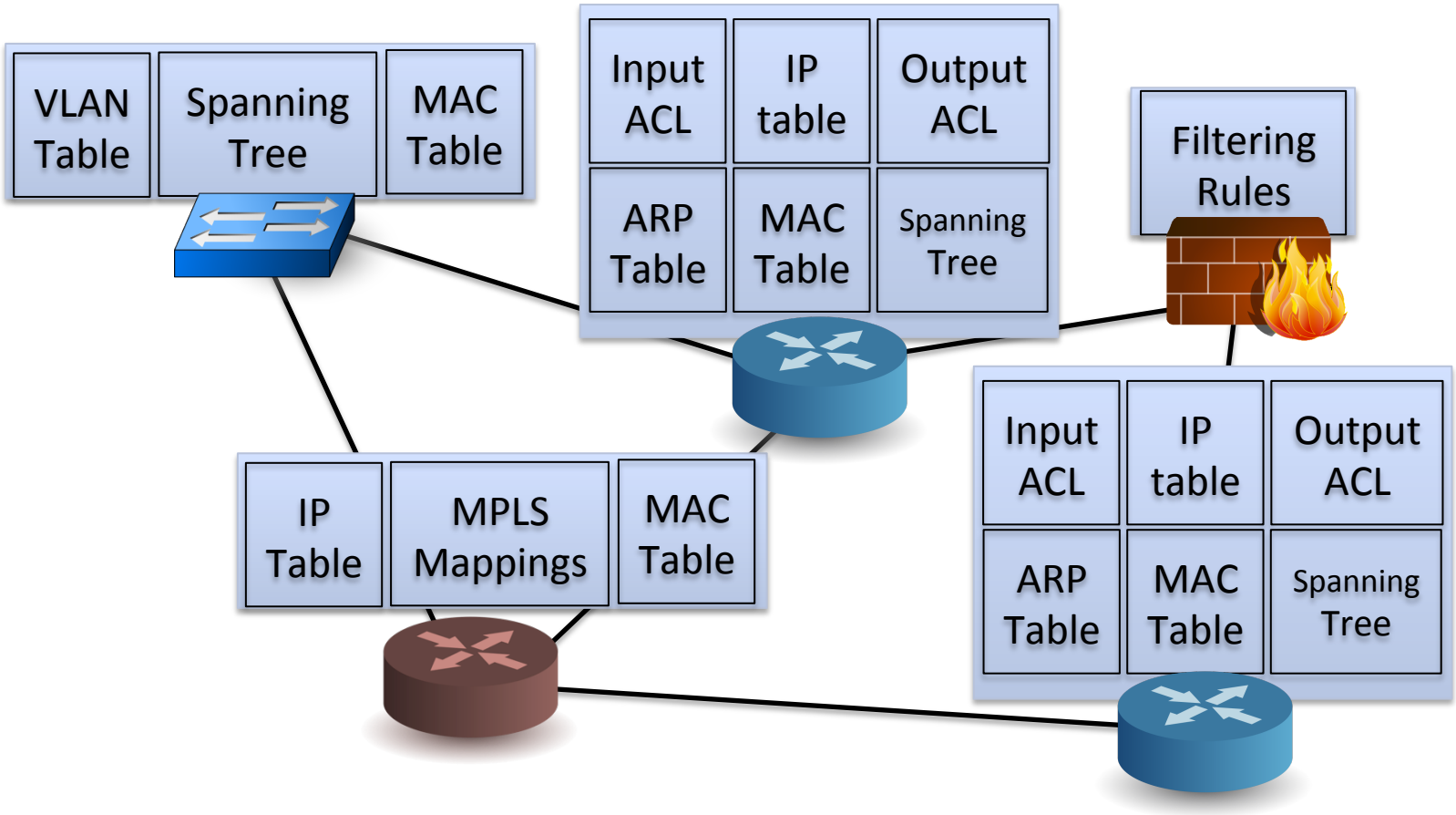
Digital Hardware Design



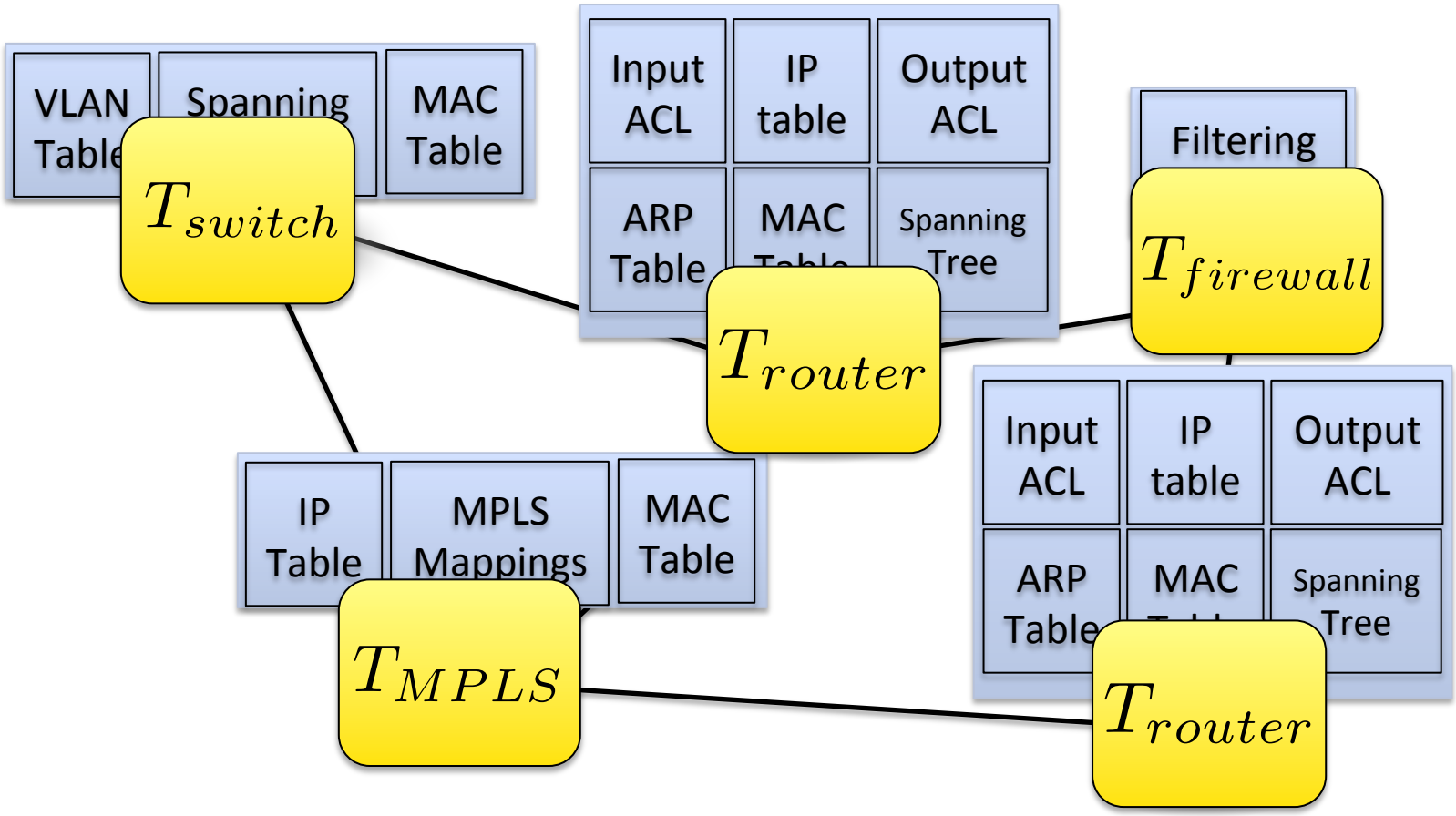
Digital Hardware Design



Vision for Network Verification



Vision for Network Verification



Outline

- Header Space Analysis (HSA)
  - Header Space
  - Transfer Function
  - HSA Set Algebra
- Algorithms
  - Reachability
  - Loop Detection
  - Slice Isolation
- Tools
  - Hassel: Header Space Library (+Demo)

Tomorrow

  - NetPlumber: Real Time Policy and Invariant Checking.
  - ATPG: Automatic Test Packet Generation.

Outline

- Header Space Analysis (HSA)
  - Header Space
  - Transfer Function
  - HSA Set Algebra
- Algorithms
  - Reachability
  - Loop Detection
  - Slice Isolation
- Tools
  - Hassel: Header Space Library (+Demo)
  - NetPlumber: Real Time Policy and Invariant Checking.
  - ATPG: Automatic Test Packet Generation.

How to achieve our vision?

- Define a simple and protocol-agnostic model for how a network forwards packets.
- Observation: A network consists of a set of boxes that process and forward packets.
- Header Space Analysis:
  1) Header Space: A model for packets in the network.
  2) Transfer Function: A model for the forwarding behavior of network boxes.
  3) Header Space Set Algebra: An algebra for working with header spaces.

Header Space Framework

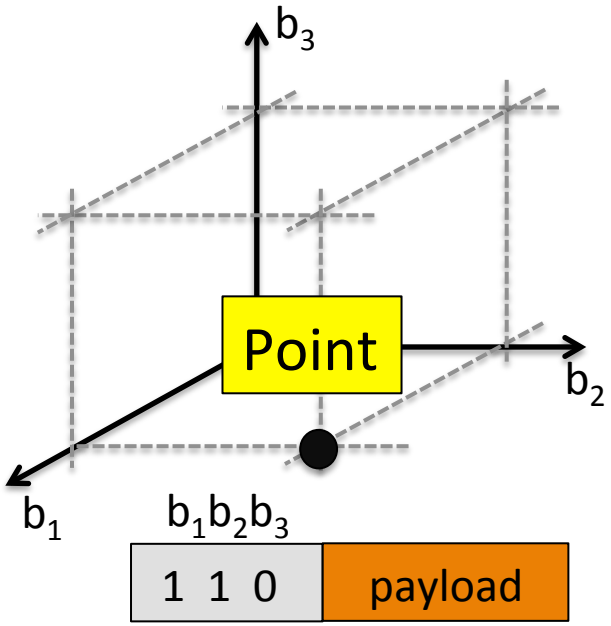
- Step One: model packets in the network.
- Observation 1: as far as processing of packets is concerned, two things matter (in most cases):
  - Header bits and
  - Location of packet (a.k.a. physical port)

Header Space Framework

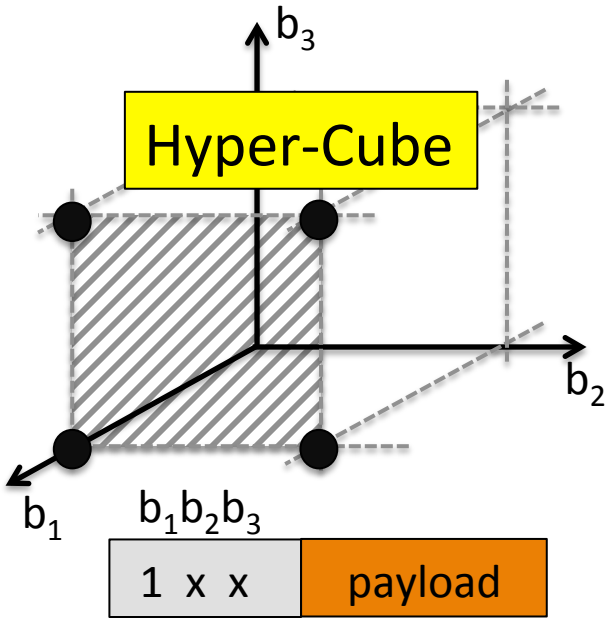
- Step One: model packets in the network.
- Observation 2: packet headers are only a sequence of 0s and 1s.
  - Protocol fields are interpretation of these bits by packet processing elements.

Header Space Framework

- Step One: model packets in the network.
- Header Space: packets, based on their header bits modeled as a point in $\{0,1\}^L$ space.
- Example:



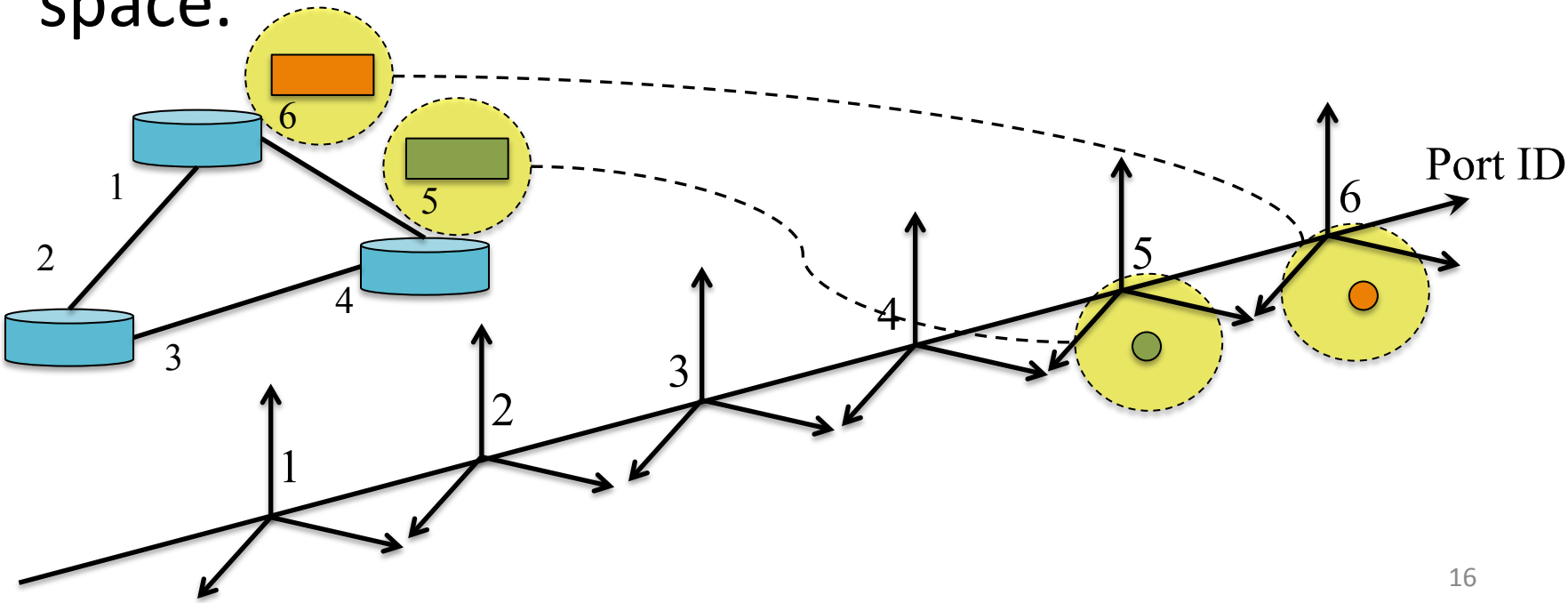
Packet with header 110



Flow with header 1xx

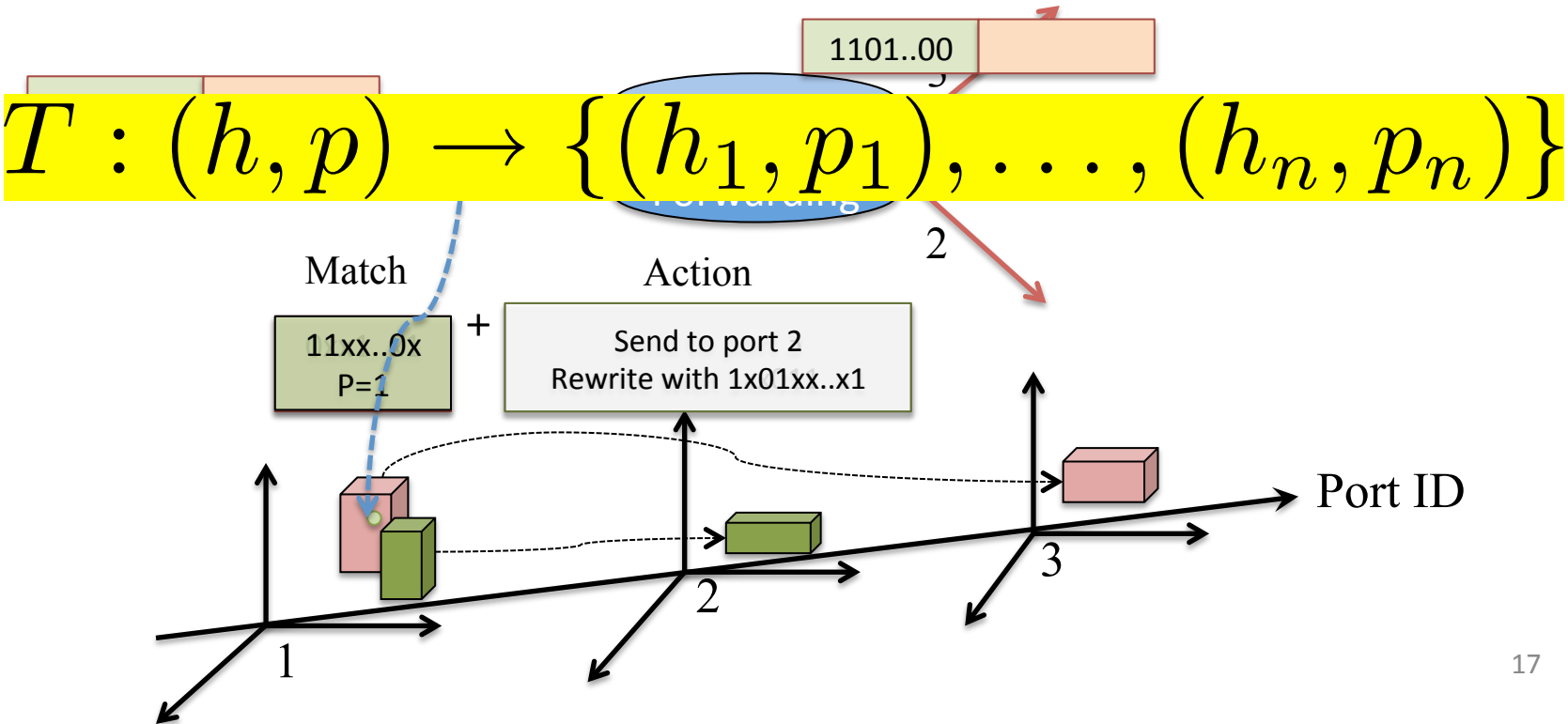
Header Space Framework

- Step One: model packets in the network.
- Network Space: packets localized on a physical port is modeled as a point in $\{0,1\}^L x\{1,\ldots,P\}$ space.



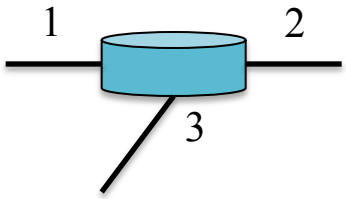
Header Space Framework

- Step Two: model the forwarding behavior of networking boxes.
- Transfer Function:



Transfer Function Example

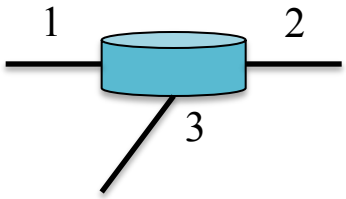
- IPv4 Router – Forwarding Behavior
  - 172.24.74.0 255.255.255.0 Port1
  - 172.24.128.0 255.255.255.0 Port2
  - 171.67.0.0 255.255.0.0 Port3



$$T(h, p) = \begin{cases} (h,1) & \text{if dst\_ip}(h) = 172.24.74.x \\ (h,2) & \text{if dst\_ip}(h) = 172.24.128.x \\ (h,3) & \text{if dst\_ip}(h) = 171.67.x.x \end{cases}$$

Transfer Function Example

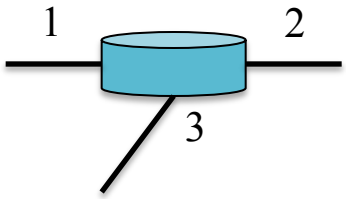
- IPv4 Router – forwarding + TTL
  - 172.24.74.0 255.255.255.0 Port1
  - 172.24.128.0 255.255.255.0 Port2
  - 171.67.0.0 255.255.0.0 Port3



$$T(h, p) = \begin{cases} (\text{dec\_ttl}(h), 1) & \text{if dst\_ip}(h) = 172.24.74.x \\ (\text{dec\_ttl}(h), 2) & \text{if dst\_ip}(h) = 172.24.128.x \\ (\text{dec\_ttl}(h), 3) & \text{if dst\_ip}(h) = 171.67.x.x \end{cases}$$

Transfer Function Example

- IPv4 Router – forwarding + TTL + MAC rewrite
  - 172.24.74.0 255.255.255.0 Port1
  - 172.24.128.0 255.255.255.0 Port2
  - 171.67.0.0 255.255.0.0 Port3



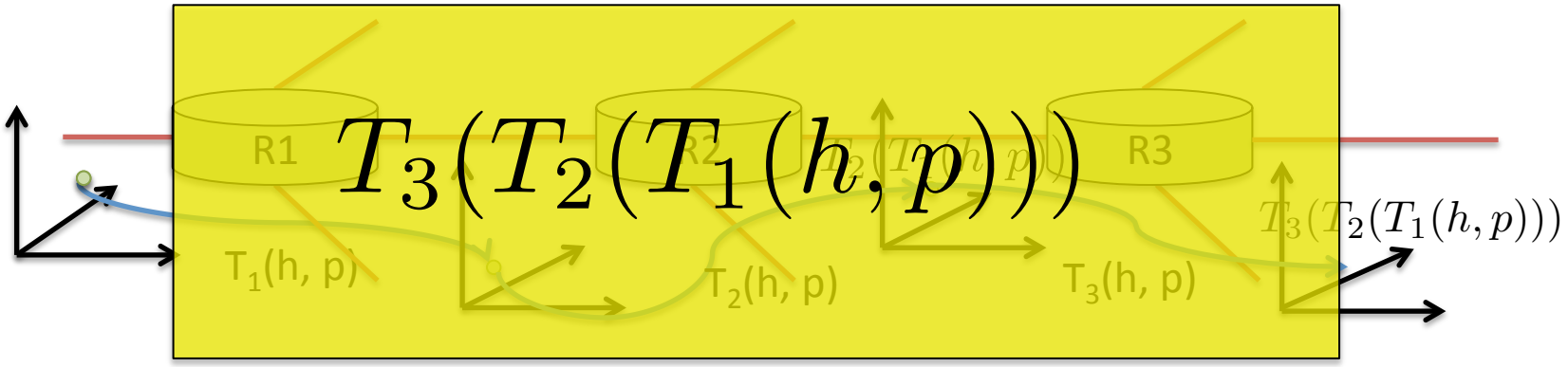
$$T(h, p) = \begin{cases} (\text{rw\_mac}(\text{dec\_ttl}(h), \text{next\_mac}), 1) & \text{if dst\_ip}(h) = 172.24.74.x \\ (\text{rw\_mac}(\text{dec\_ttl}(h), \text{next\_mac}), 2) & \text{if dst\_ip}(h) = 172.24.128.x \\ (\text{rw\_mac}(\text{dec\_ttl}(h), \text{next\_mac}), 3) & \text{if dst\_ip}(h) = 171.67.x.x \end{cases}$$

Example Actions:

- Rewrtie: rewrite bits 0-2 with value 101
  - (h & 000111…) | 101000…
- Encapsulation: encap packet in a 1010 header.
  - (h >> 4) | 1010….
- Decapsulation: decap 1010xxx… packets
  - (h << 4) | 000…xxxx
- TTL Decrement:
  - if ttl(h) == 0: Drop
  - if ttl(h) > 0: h – 0…000000010…0
- Load Balancing:
  - LB(h,p) = $\{(h,P_1),\ldots(h,P_n)\}$

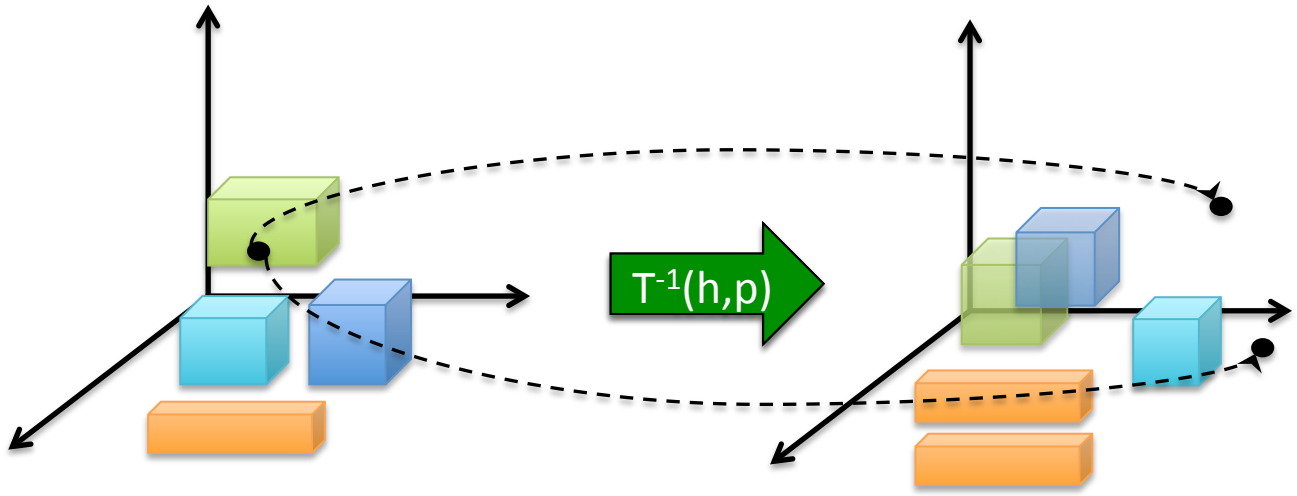
Composing Transfer Functions

- By composing transfer functions, we can find the end to end behavior of networks.



Inverse of Transfer Function

- Tell us all possible input packets that can generate an output packet.



Header Space Framework

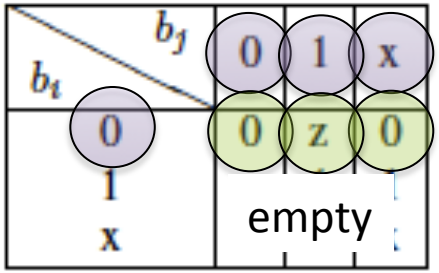
- Step Three: A set algebra for working with header spaces.
- Why? different applications, require performing set operations on header space regions.
  - Subset and Equality: checking reachability policy.
  - Intersection: checking isolation of network slices.

Header Space Framework

- Step Three: A set algebra for working with header spaces.
  - Intersection
  - Complementation and Difference
  - Checking subset and equality
- Observation: Every region of Header Space, can be described by union of wildcard expressions (WEs).
- Goal: Define set operations on WEs.

HS Set Algebra- Intersection

- Bit by bit intersect using intersection table:
  - Example: $10xx \cap 1xx0 = 10x0$
  - If result has any 'z', then intersection is empty:
  - Example: $10xx \cap 0xx1 = z0x1 = \phi$



HS Set Algebra- Complementation

- Flip every non-wildcard bit, and wild card every other bit.
- Result is union of all of these expressions
  - Example:

$$(010x)^c = 1xxx \cup x0xx \cup xx1x$$

HS Set Algebra- Difference

- $H_1 - H_2 = H_1 \cap H_2^c$
- Example:

$$01xx - 010x =$$
$$01xx \cap (1xxx \cup x0xx \cup xx1x) =$$
$$011x$$

HS Set Algebra- Subset & Equality

- Subset:

$$H_1 \subset H_2 \Longleftrightarrow H_1 - H_2 = \phi$$

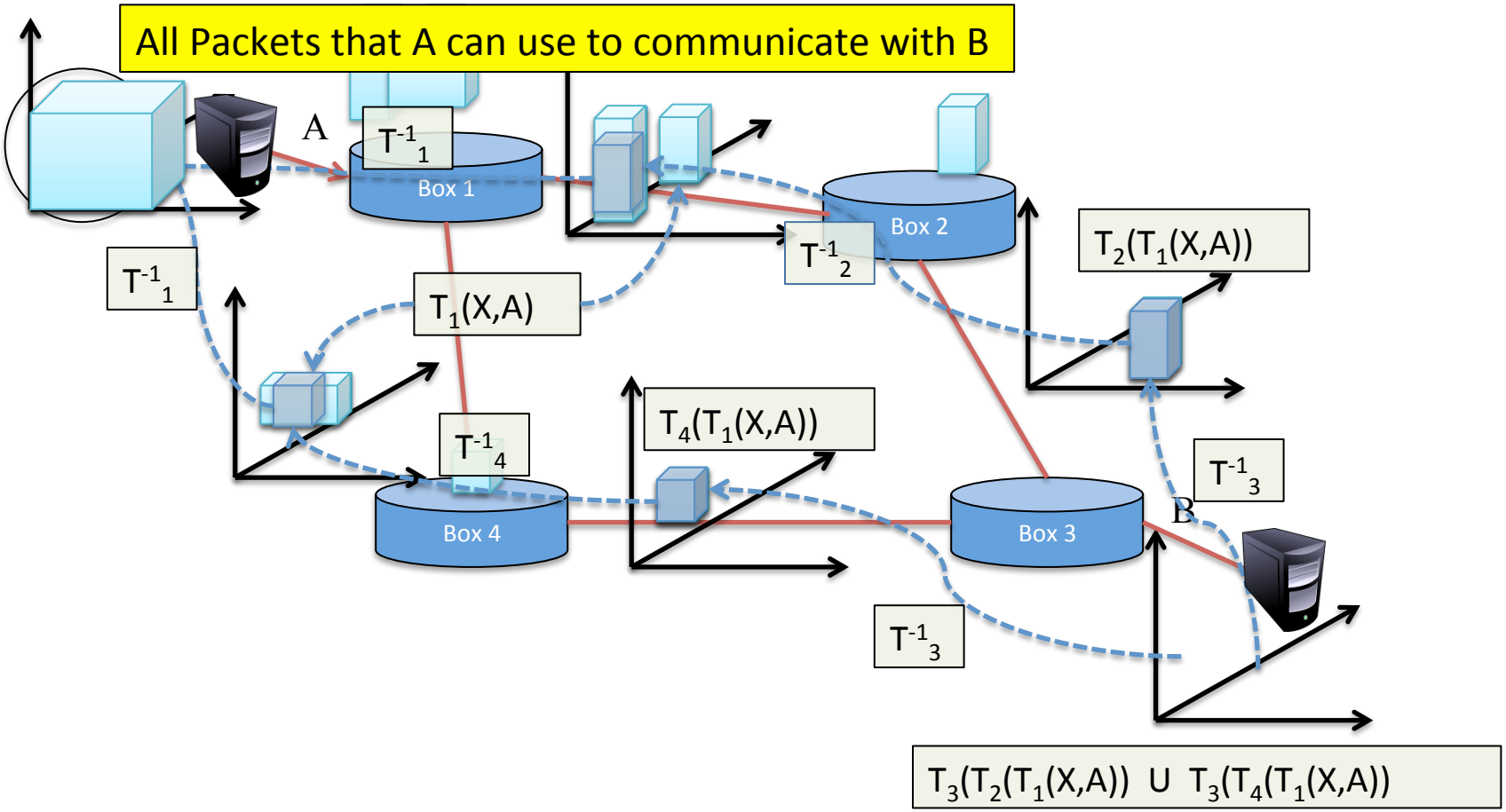
- Equality:

$$H_1 = H_2 \Longleftrightarrow H_1 - H_2 = \phi \ \ \& \ \ H_2 - H_1 = \phi$$

Outline

- Header Space Analysis (HSA)
  - Header Space
  - Transfer Function
  - HSA Set Algebra
- Algorithms
  - Reachability
  - Loop Detection
  - Slice Isolation
- Tools
  - Hassel: Header Space Library. (+Demo)
  - NetPlumber: Real Time Policy and Invariant Checking.
  - ATPG: Automatic Test Packet Generation.

Finding Reachability

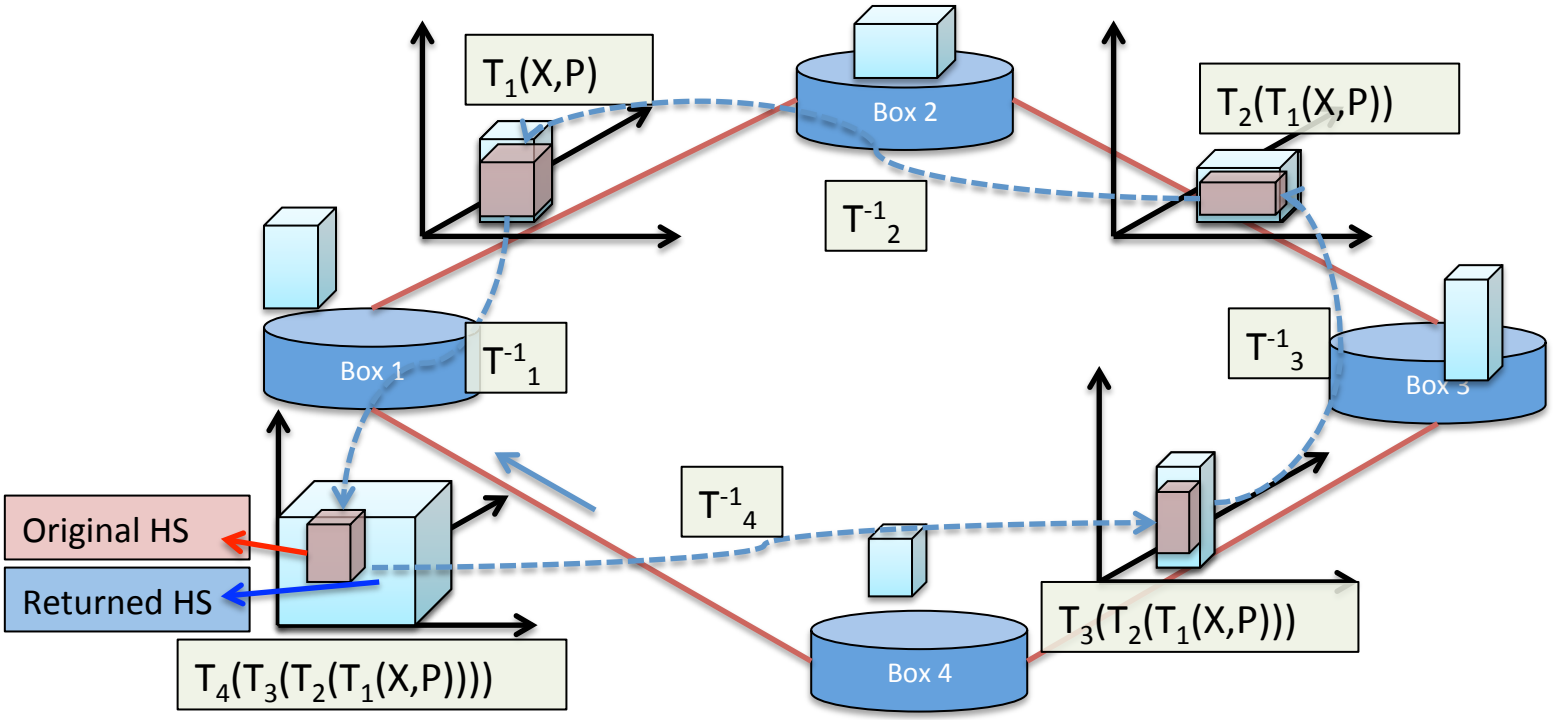


Checking Predicates on Paths

- Once we do reachability, we can generalize to check *path predicates* such as:
  - Blackhole freedom (A → B and notice unexpected drop)
  - Communication via middle box. (A→B packets must pass through C)
  - Maximum hop count (length of path from A → B never exceeds L)
  - Isolation of paths (http and https traffic from A→B don't share the same path)
  - Loop freedom (No packet should be reachable from any port to itself)

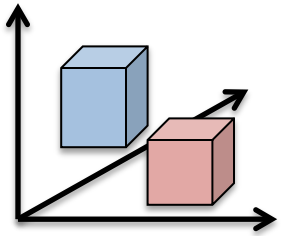
Finding Loops

- Is there a loop in the network?
  - Inject an all-x test packet from every switch-port
  - Follow the packet until it comes back to injection port

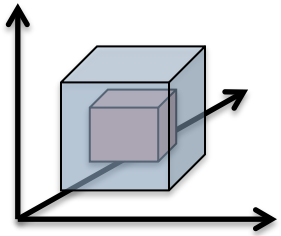


Finding Loops

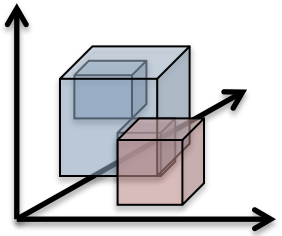
- Is the loop infinite?



Finite Loop



Infinite Loop



?

Network Slices

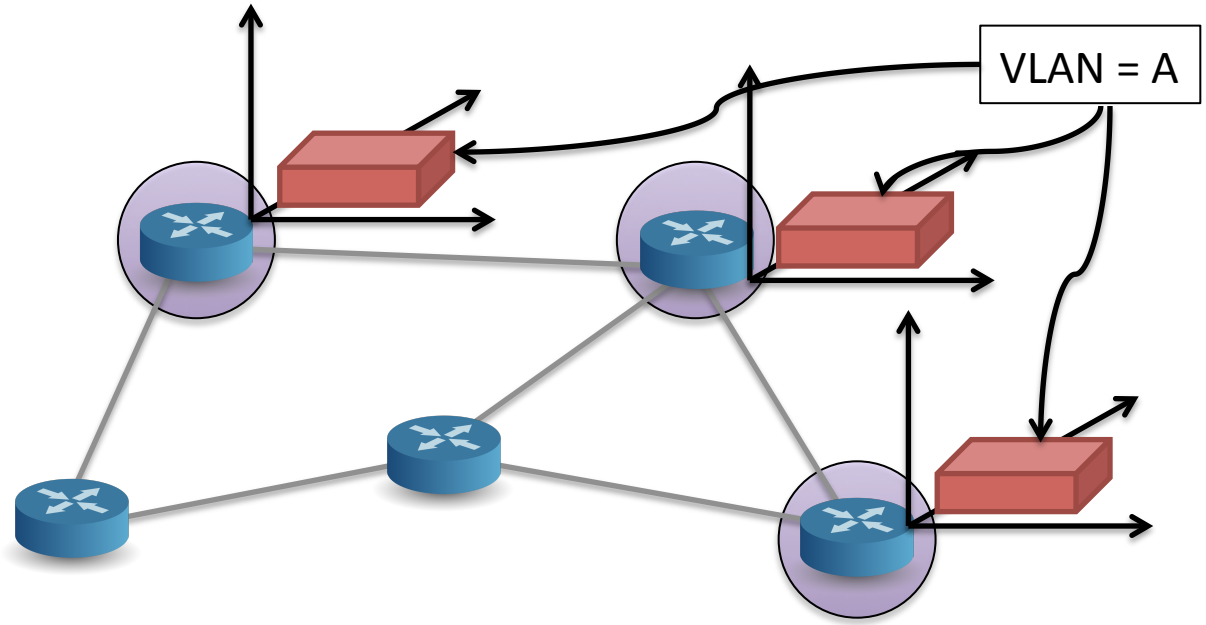
- Slicing network allow us to share network resources. (e.g. Bank of America and Citi share the same infrastructure in a financial center).
- Like VM, we need to ensure no interaction between slices. (independence, security).

→ We need to ensure isolation of slices.

Definition of Slice in HSA laguange

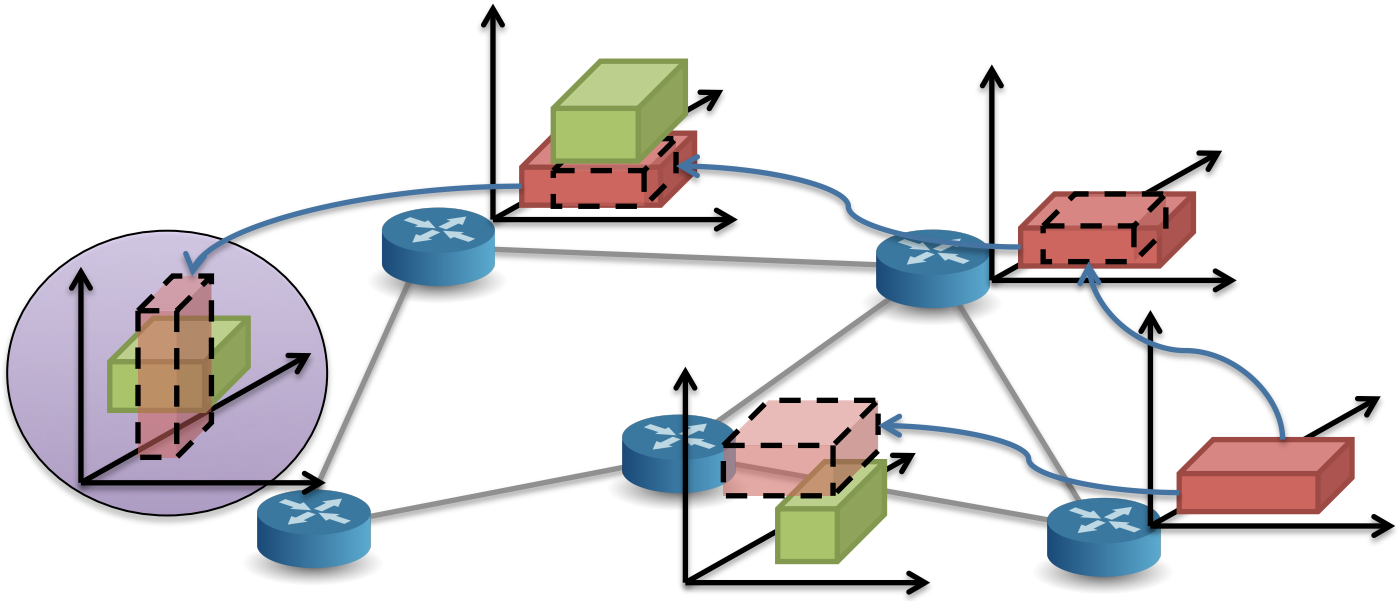
- Network slice: a piece of network resources defined by
  - A topology consisting of switches and ports.
  - A set of predicates on packet headers.

=> Defined by a region in the network space



Checking Isolation of Slices

- How to check if two slices are isolated?
  - Slice definitions don't intersect.
  - Packets don't leak after forwarding.



Outline

- Header Space Analysis (HSA)
  - Header Space
  - Transfer Function
  - HSA Set Algebra
- Algorithms
  - Reachability
  - Loop Detection
  - Slice Isolation
- Tools
  - Hassel: Header Space Library. (+Demo)
  - NetPlumber: Real Time Policy and Invariant Checking.
  - ATPG: Automatic Test Packet Generation.

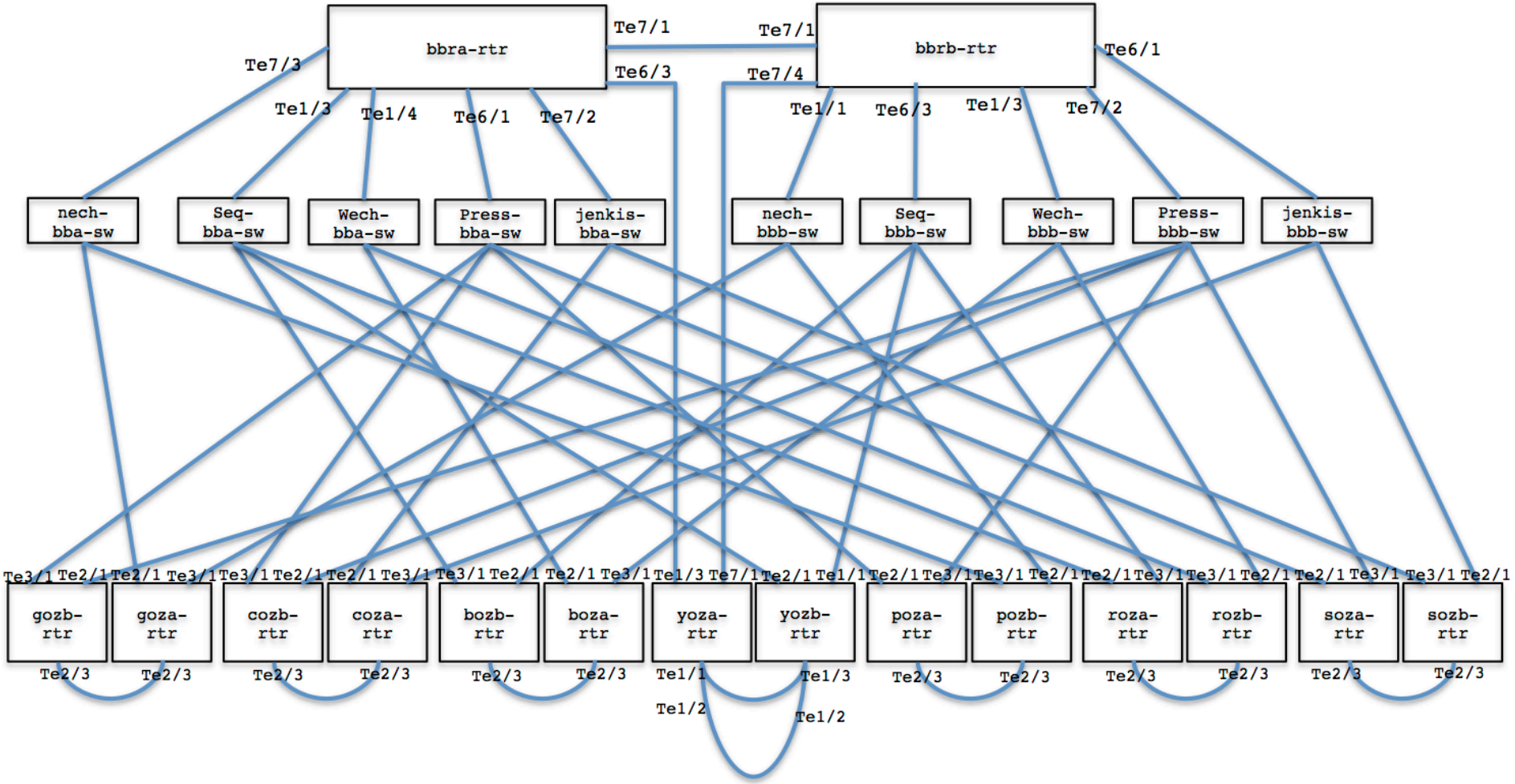
Header Space Library (Hassel)

- Two versions – Python and C.
- Foundation Layer
  - Implements Header Space and Transfer Function objects.
- Application Layer
  - Reachability, Loop Detection and Slice Isolation checks.
  - < 100 LoC for these checks.
- Parser (only available in Python)
  - CLI Parsing tool for Cisco IOS, Juniper Junos and OpenFlow table dump.
  - Keeps a mapping from TF Rule to CLI line number.
- https://bitbucket.org/peymank/hassel-public.git

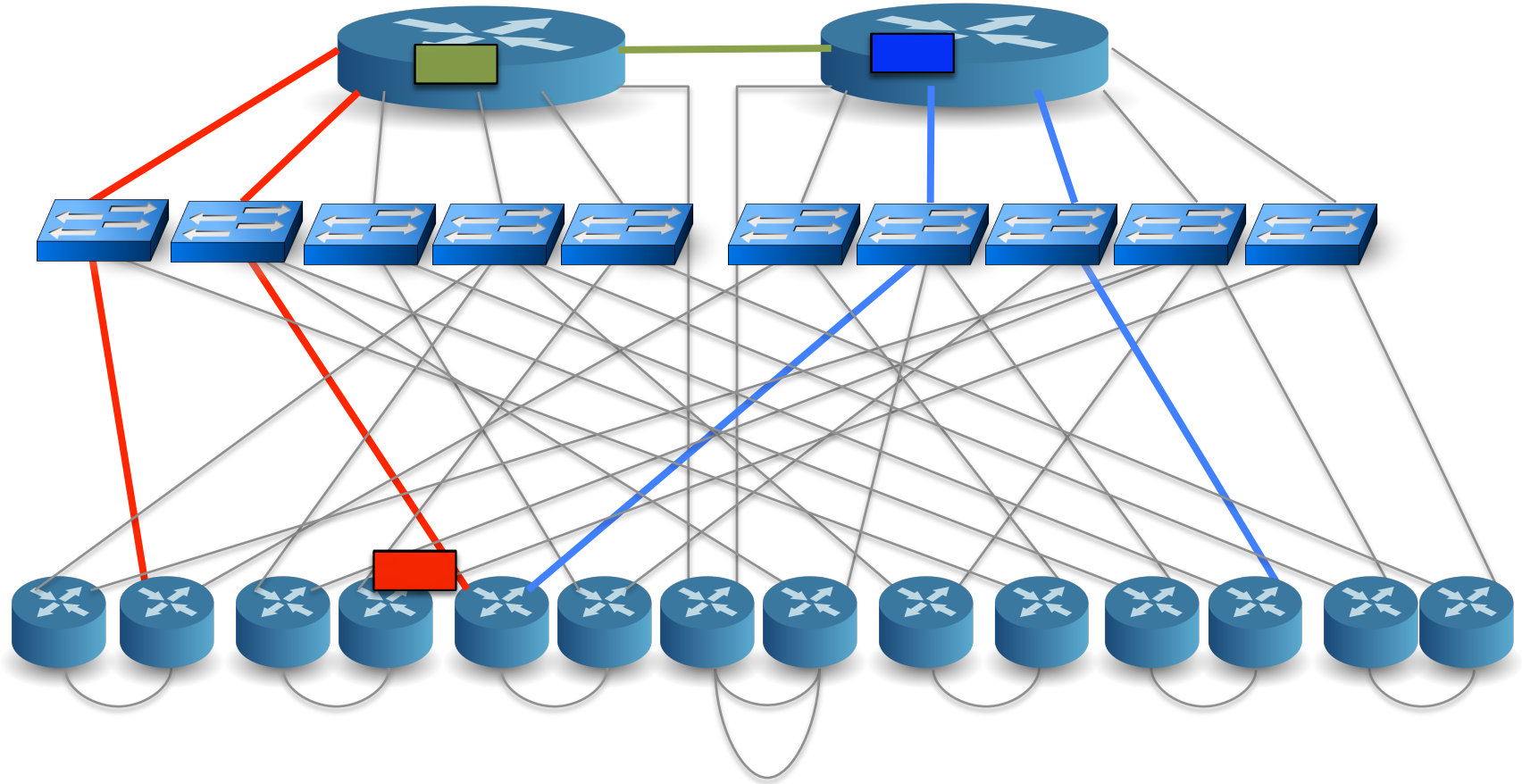
Typical Workflow with Hassel

- Get a dump of the network forwarding state.
  - E.g., IP table, ARP table, MAC table, Spanning tree output and Configuration file.
- Use the parser to generate transfer function for each box in the network.
- Use the transfer functions to compute reachability, find loops, etc.

Demo: Debugging in Stanford Network



Stanford backbone network



Tomorrow…

- We will see how we can use HSA to
  - Verify properties (policy and invariants) about a network in real time.
  - Maximally test networks with minimum test packets.

Header Space Analysis
Part II

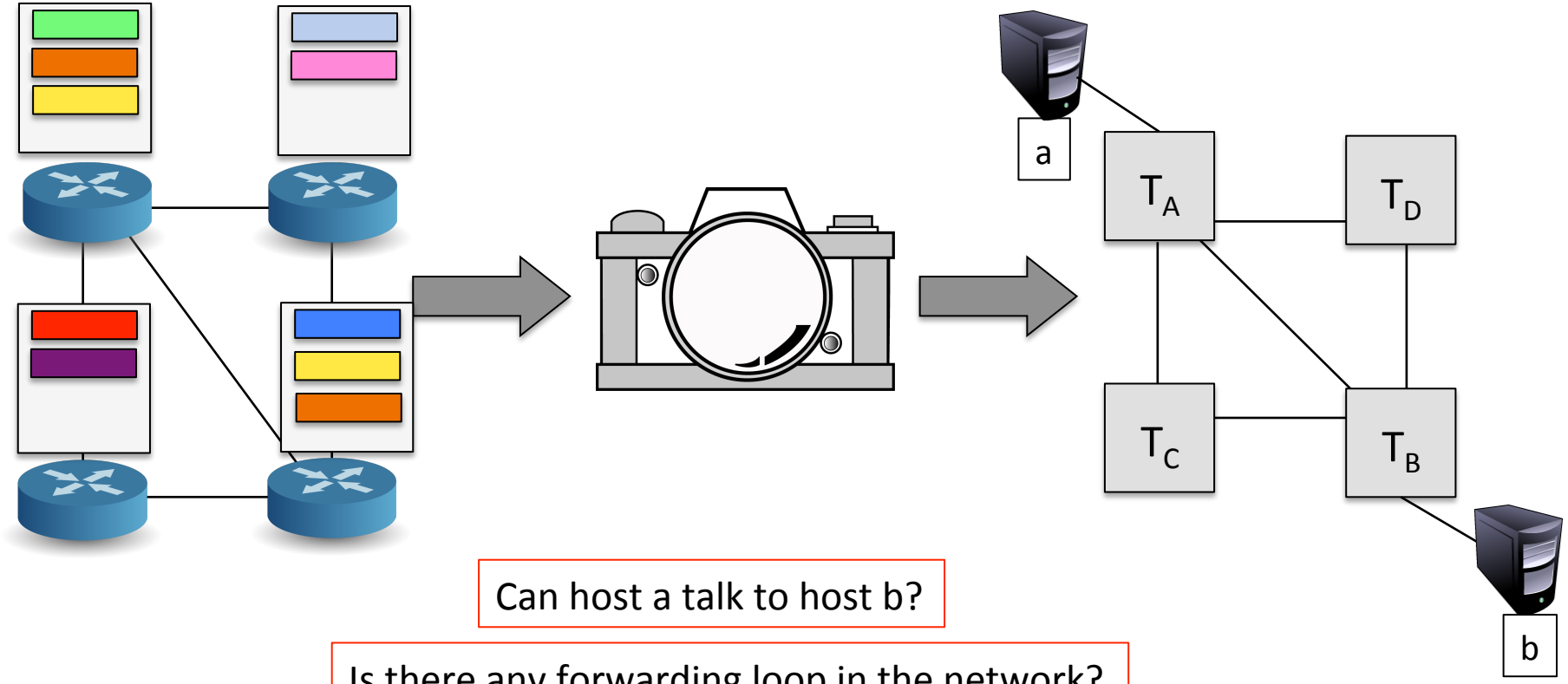
Peyman Kazemian
With James Zeng, George Varghese, Nick McKeown

Summer School on Formal Methods and Networks
Cornell University
June 2013

Outline

- NetPlumber: Real time policy checker.
  - How it works?
  - How to check policy with it?
  - Evaluation on Google WAN.
- Automatic Test Packet Generation.
  - How to pick test packets optimally?
  - Run through a toy example.
  - Deployment in CS/EE buildings in Stanford.
- Conclusion and Ideas for Future Work.

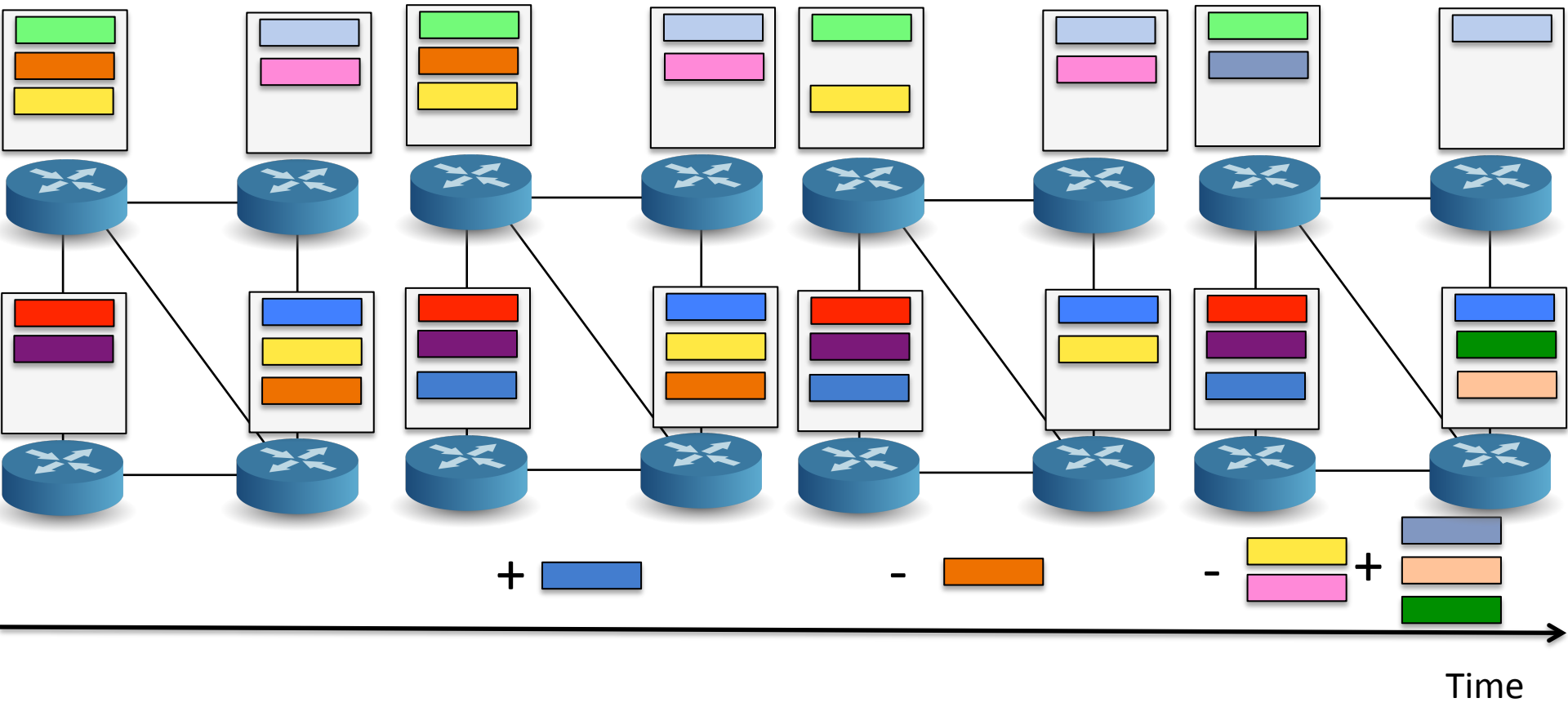
Hassel: Snapshot-based Checking



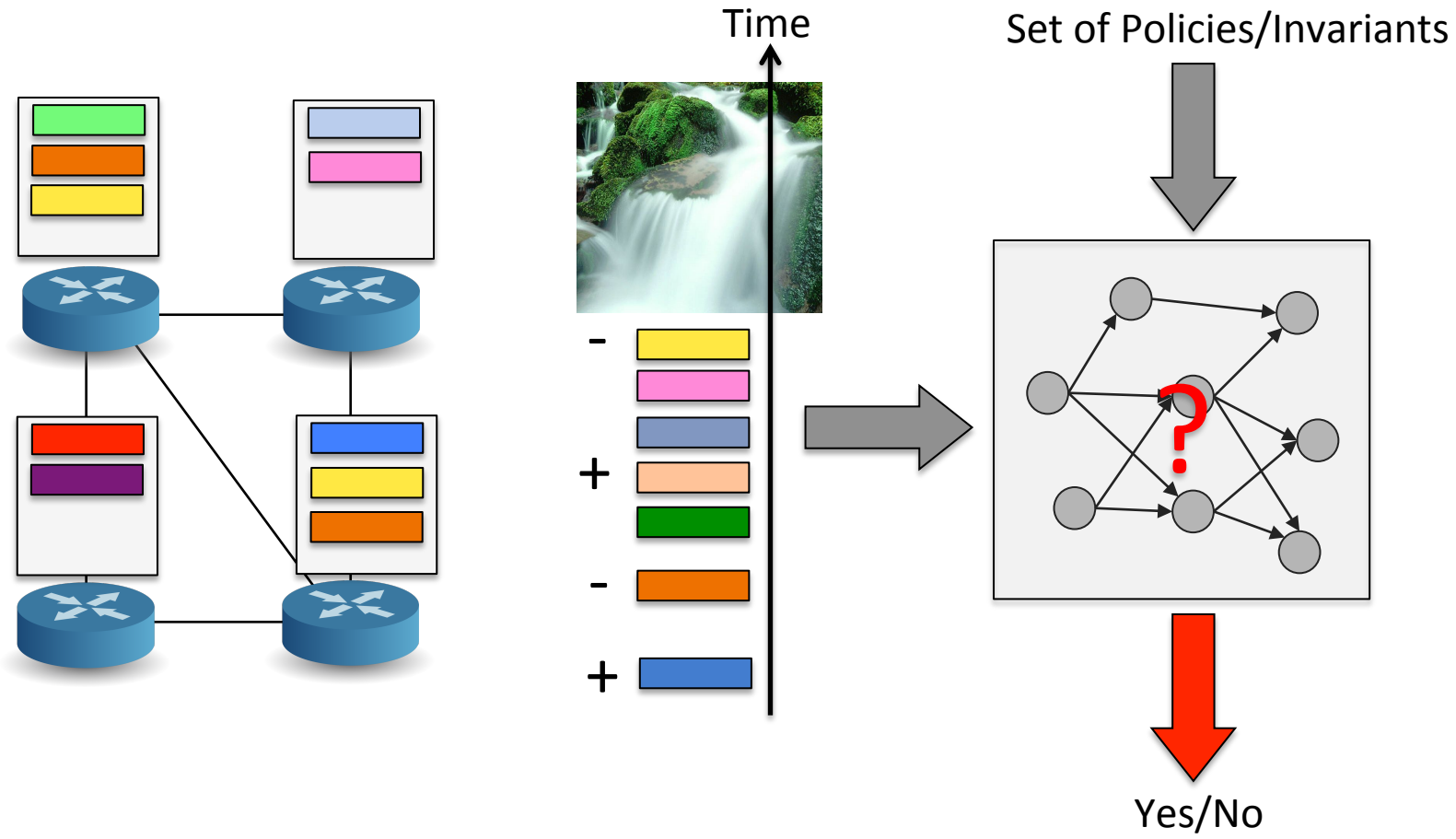
Can host a talk to host b?

Is there any forwarding loop in the network?

Steam-based Checking



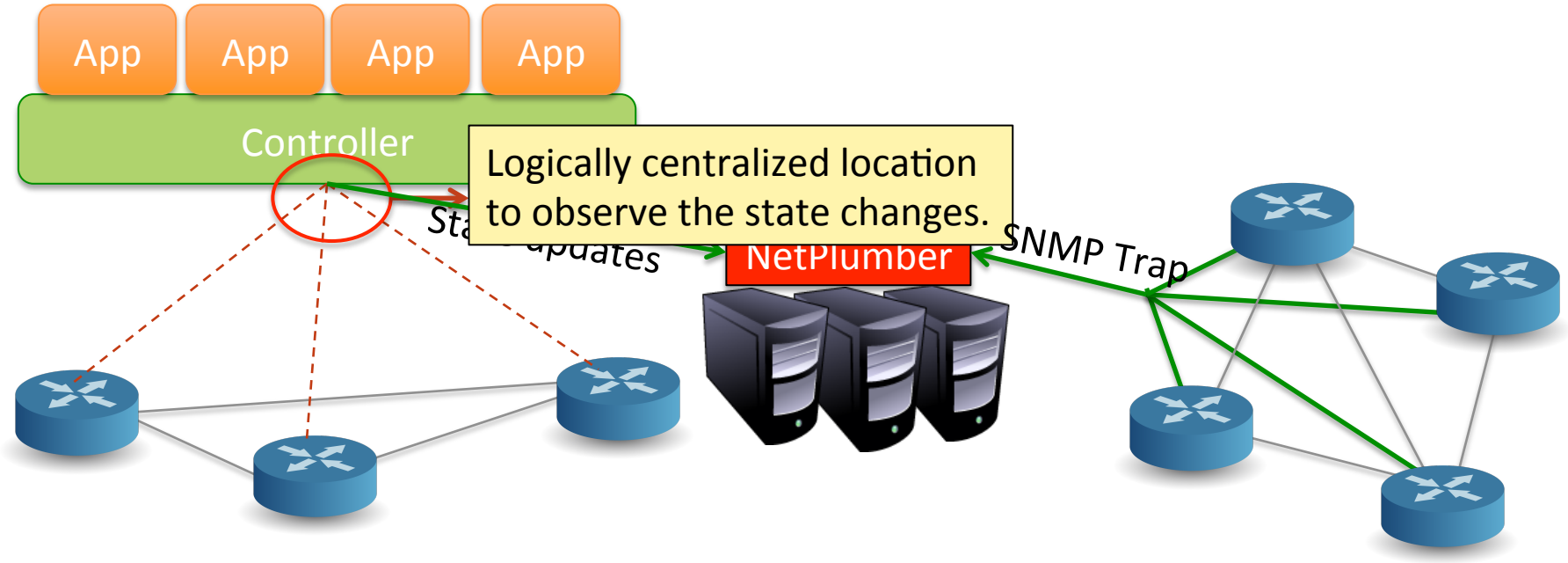
Steam-based Checking



Prevent errors before they hit network.
Report a violation as soon as it happens.

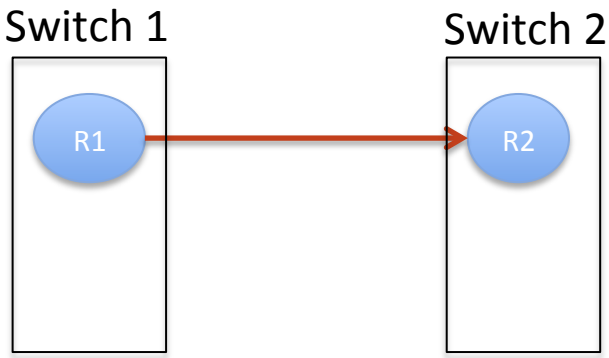
NetPlumber

- The System we build for real time policy checking is called NetPlumber.

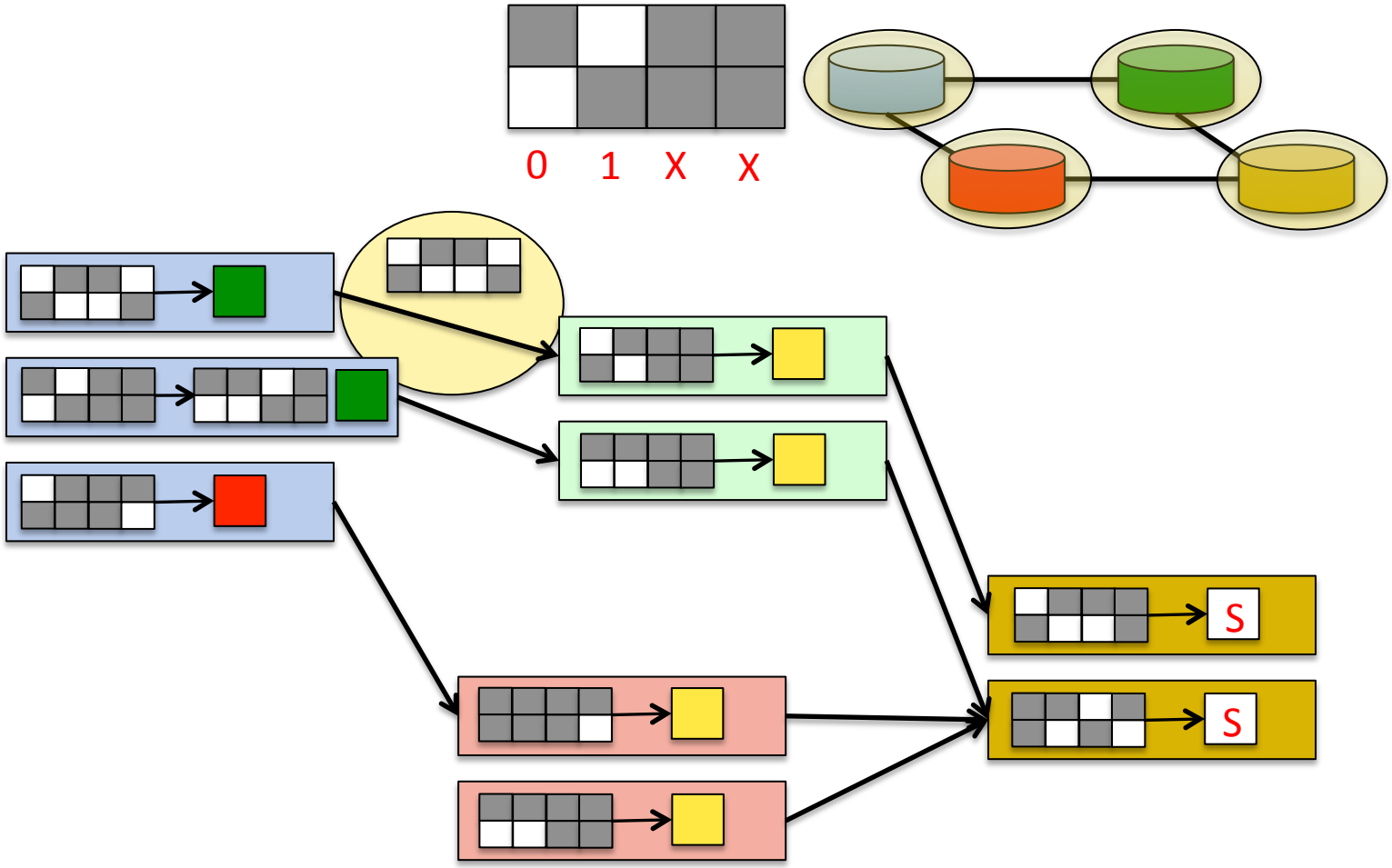


NetPlumber

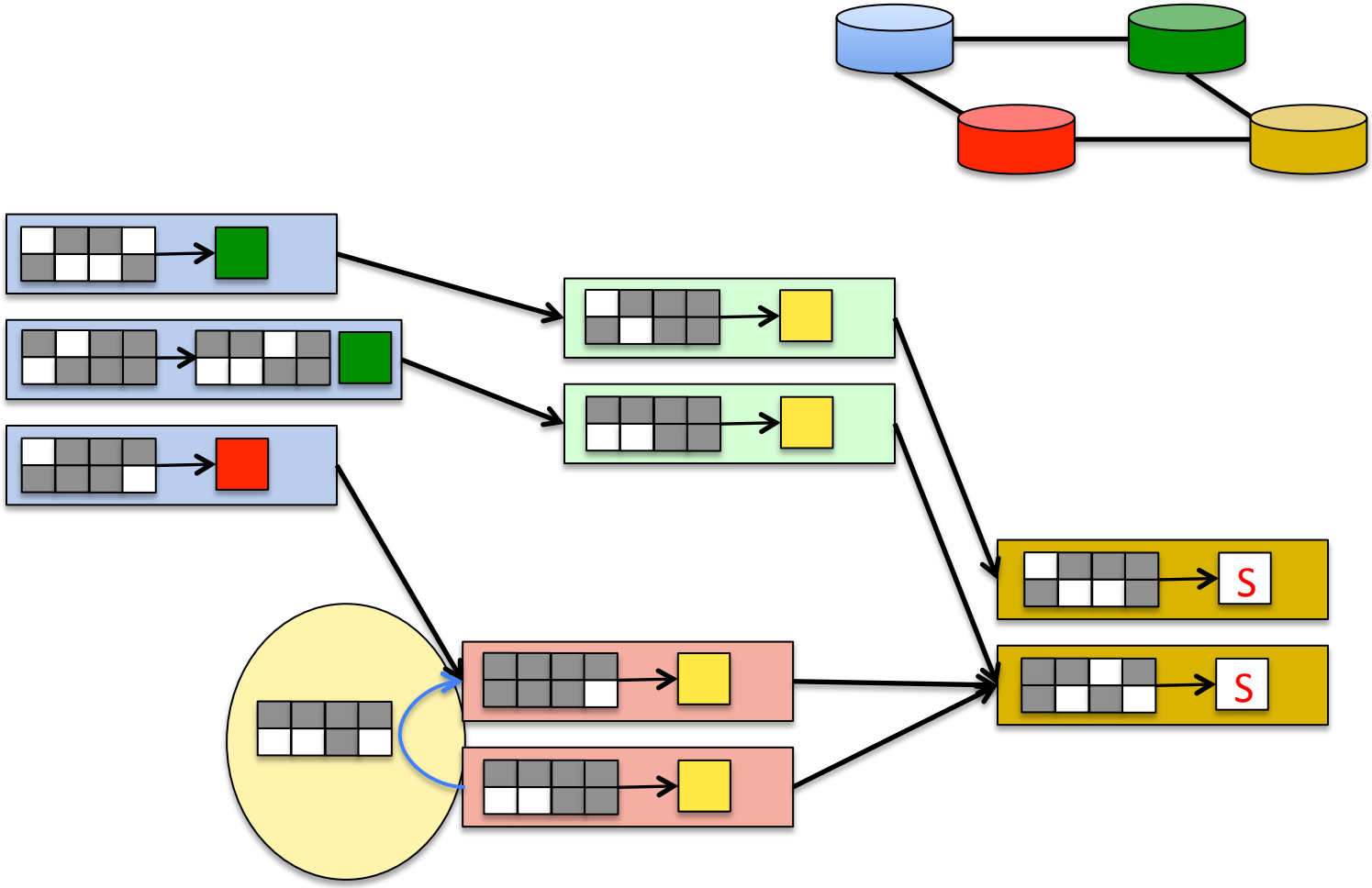
- The System we build for real time policy checking is called NetPlumber.
  - Creates a dependency graph of all forwarding rules in the network and uses it to verify policy.
  - Nodes: forwarding rules in the network.
  - Directed Edges: next hop dependency of rules.



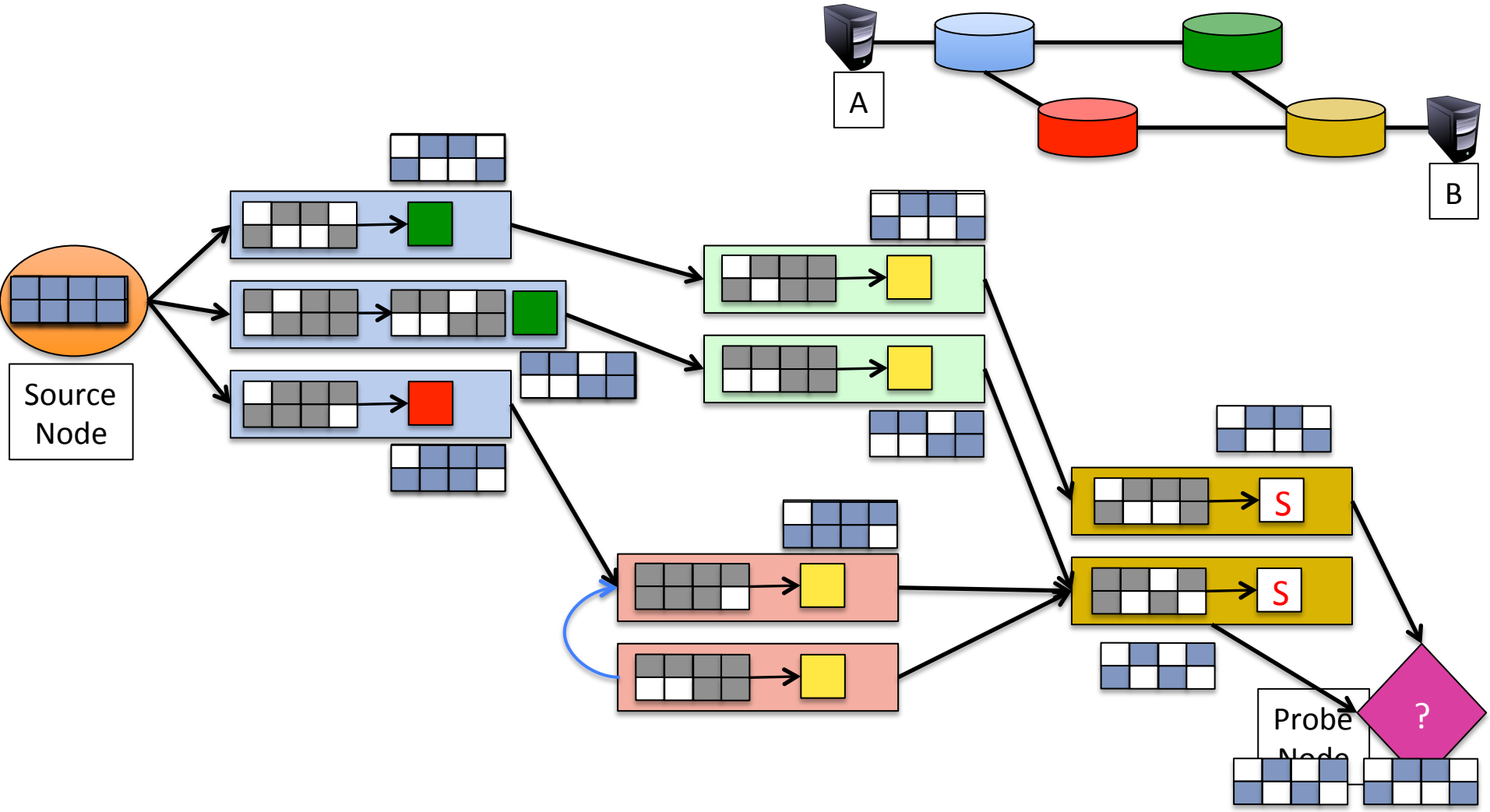
NetPlumber – Nodes and Edges



NetPlumber – Intra table dependency

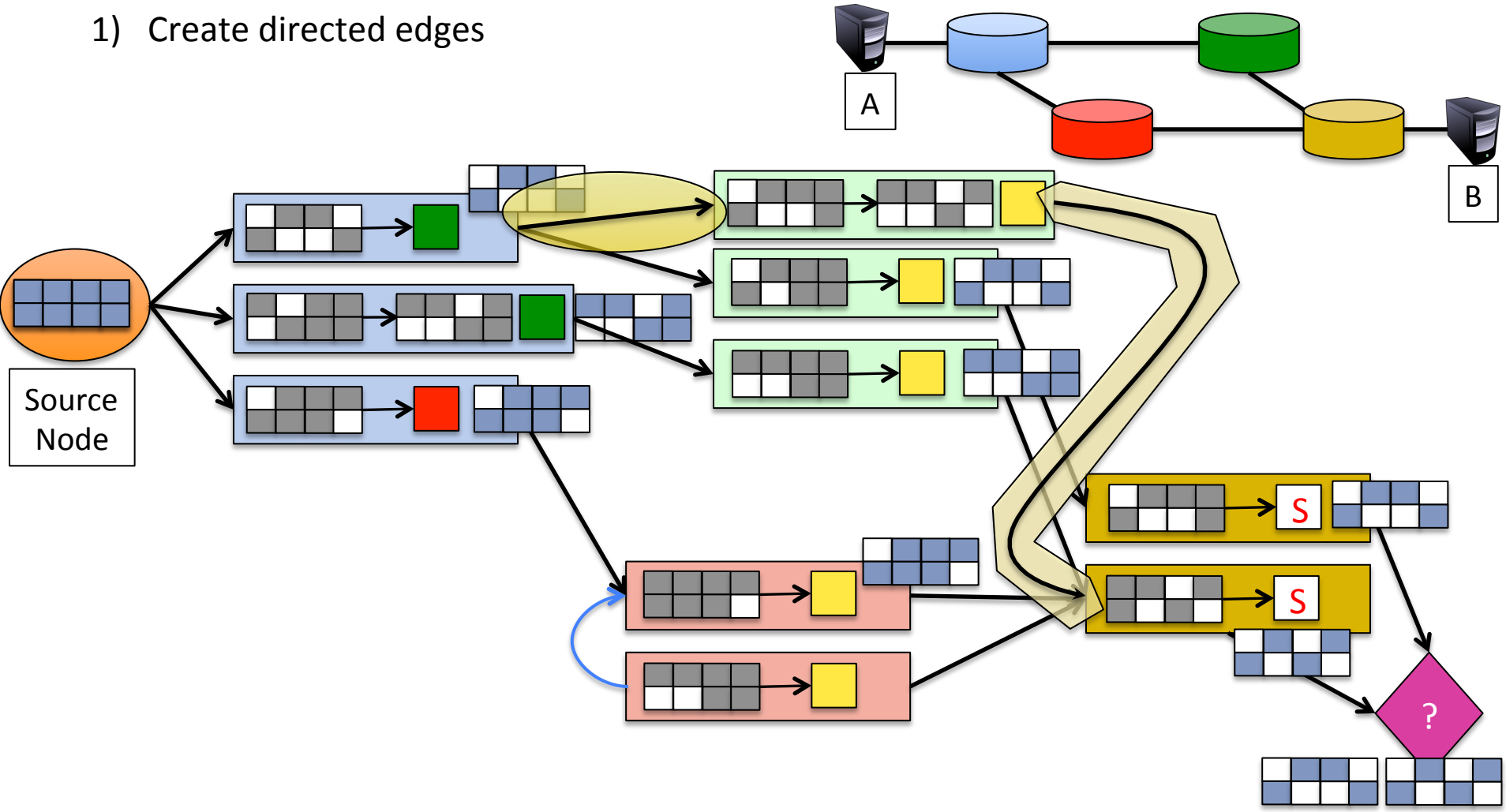


NetPlumber – Computing Reachability



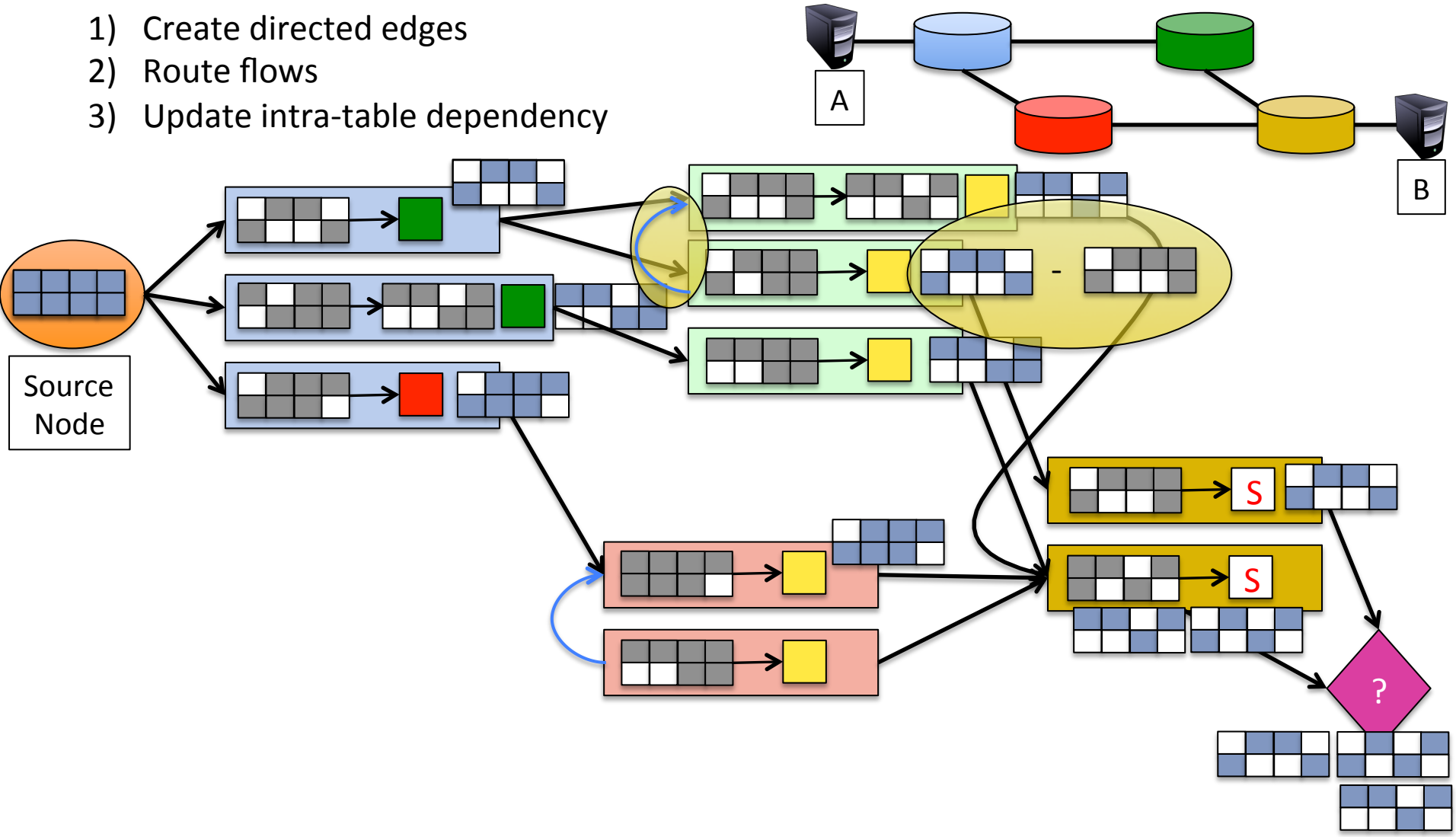
NetPlumber – Computing Reachability

1) Create directed edges

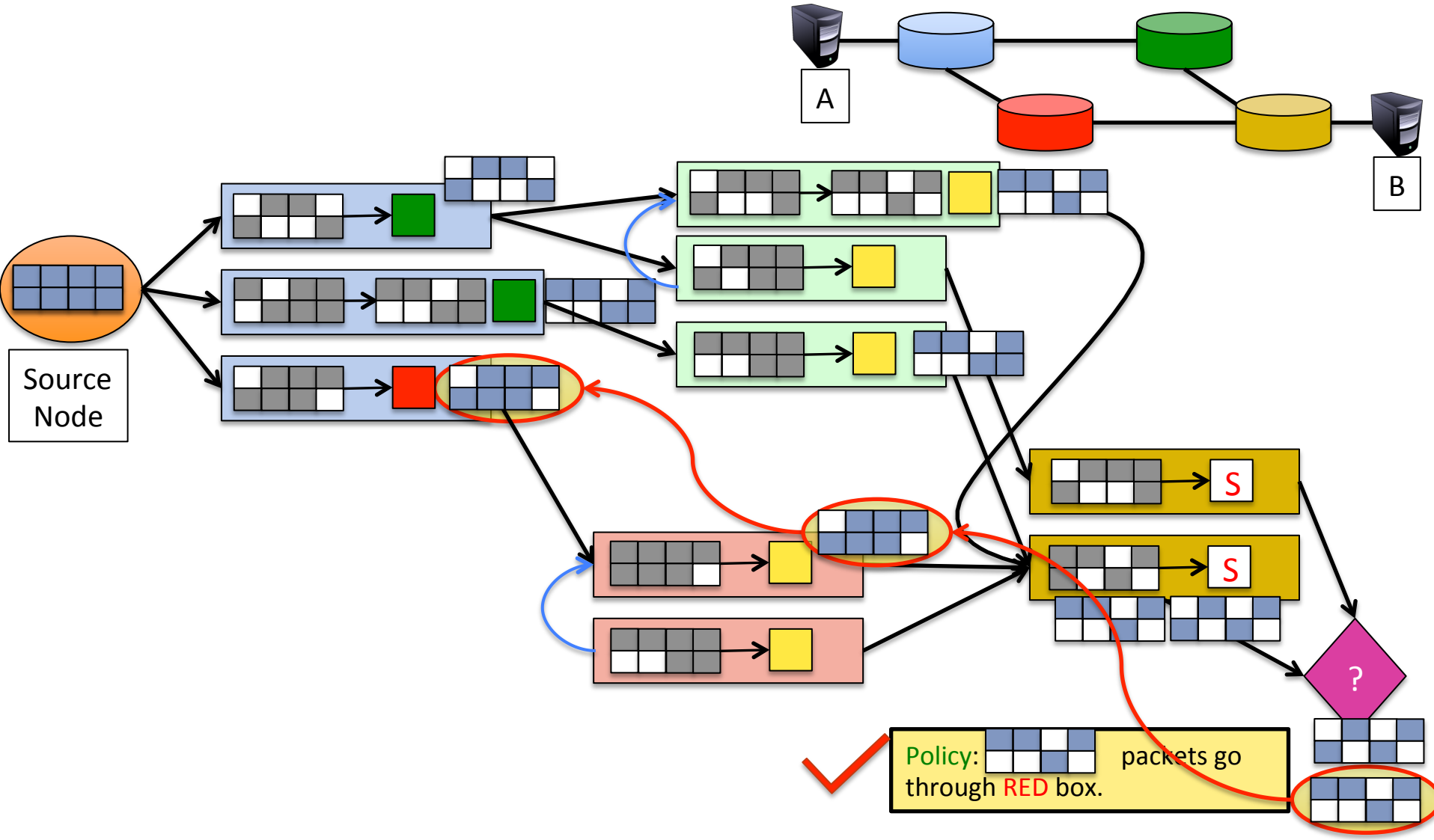


NetPlumber – Computing Reachability

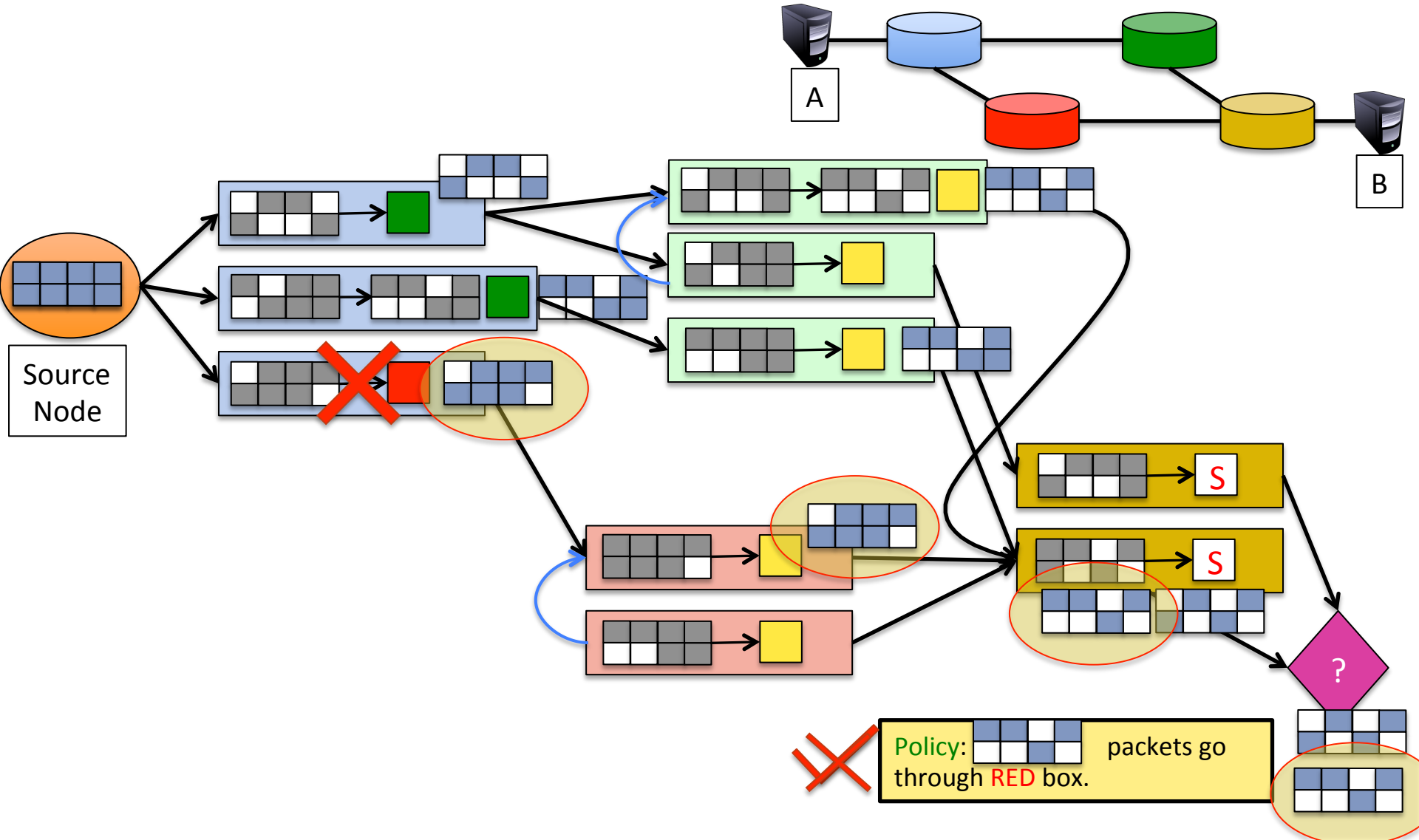
1) Create directed edges
2) Route flows
3) Update intra-table dependency



NetPlumber – Checking Policy

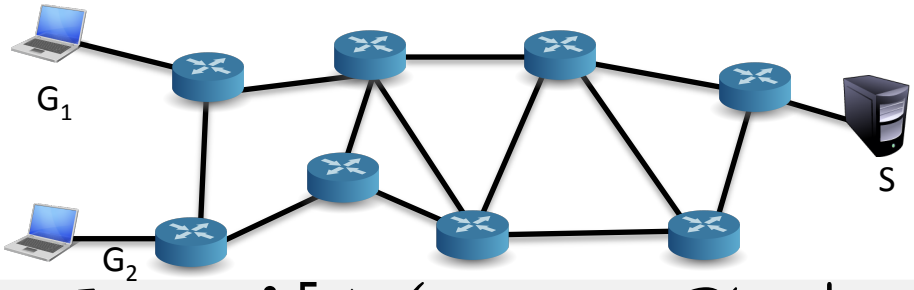


NetPlumber – Checking Policy

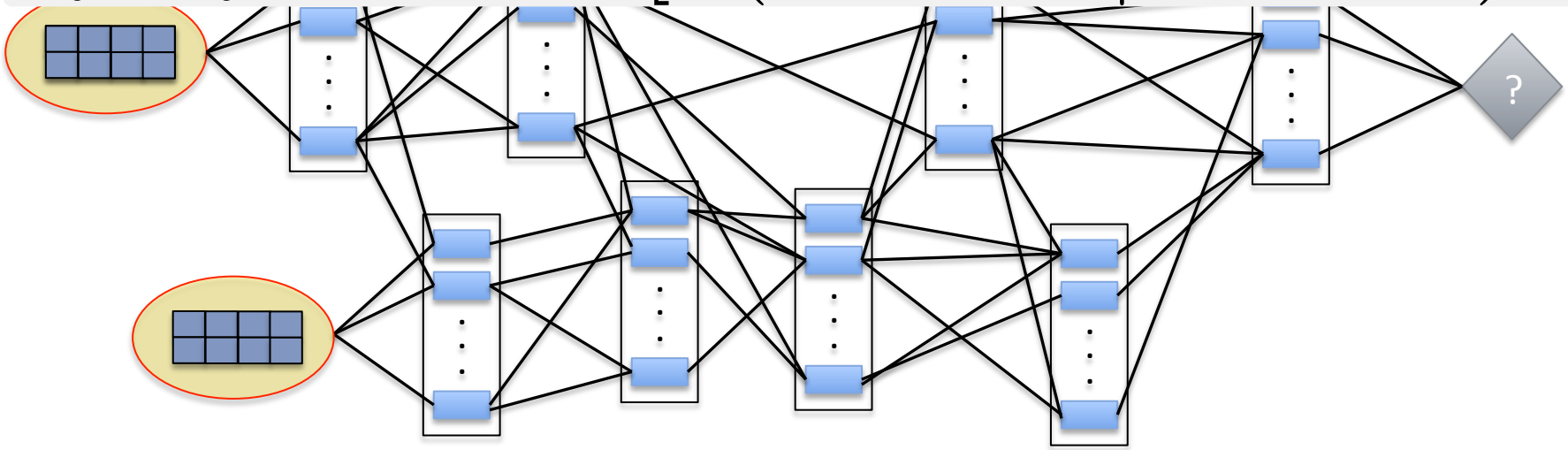


Checking Policy with NetPlumber

Policy: Guests can not access Server S.

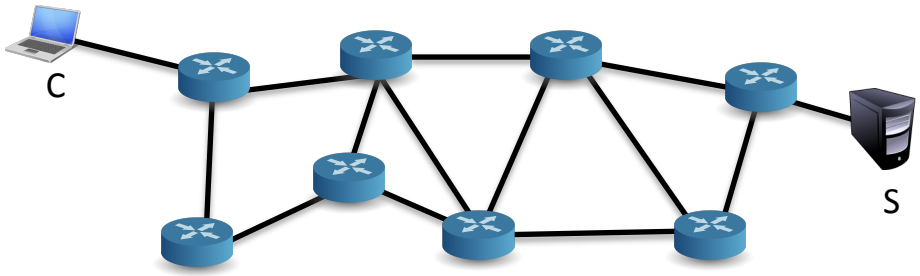


$$\forall f : f.path \sim ![\hat{}(p = G_1 \mid p = G_2).^*]$$

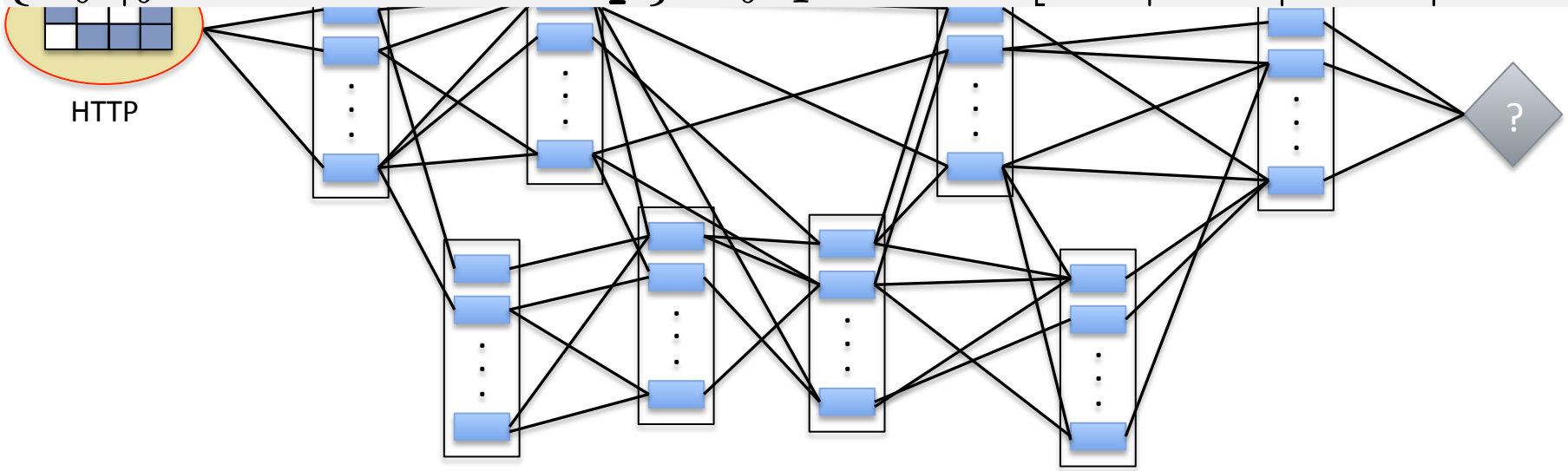


Checking Policy with NetPlumber

Policy: http traffic from client C to server S doesn't go through more than 4 hops.

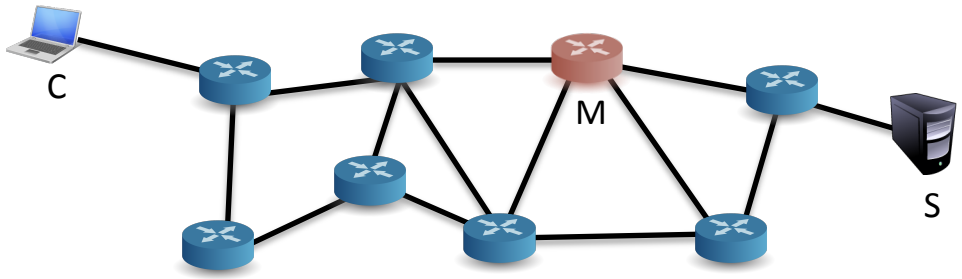


$$\{\forall f | f.header \sim \texttt{http}\} : f.path \sim [\hat{}.\$|\hat{}..\$|\hat{}...\$|\hat{}....\$]$$

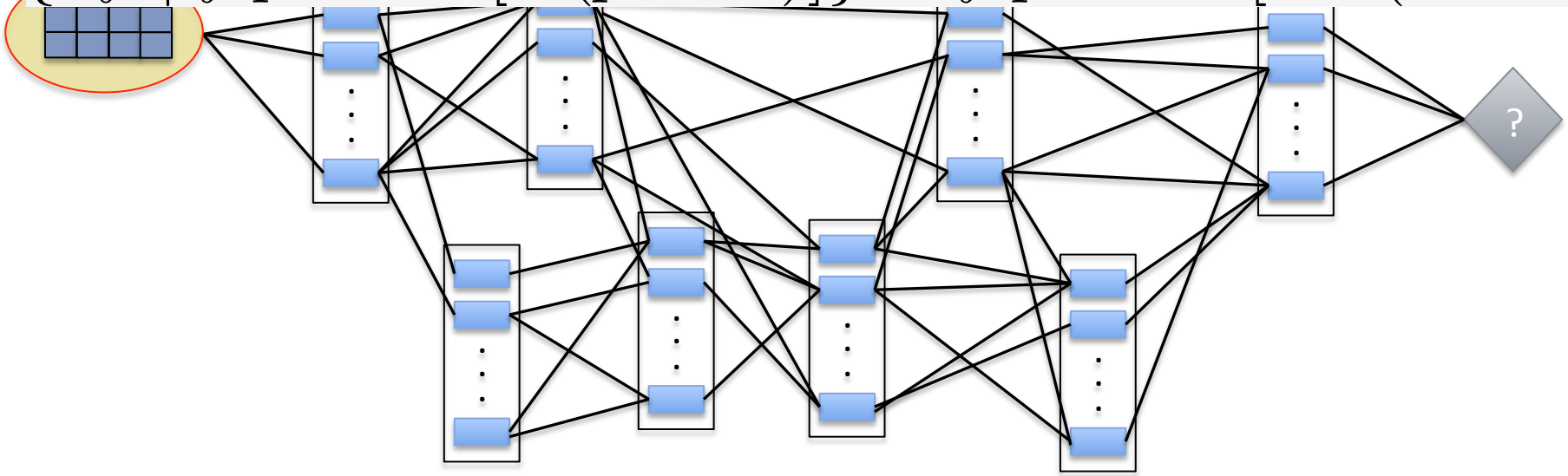


Checking Policy with NetPlumber

Policy: traffic from client C to server S should go through middle box M.



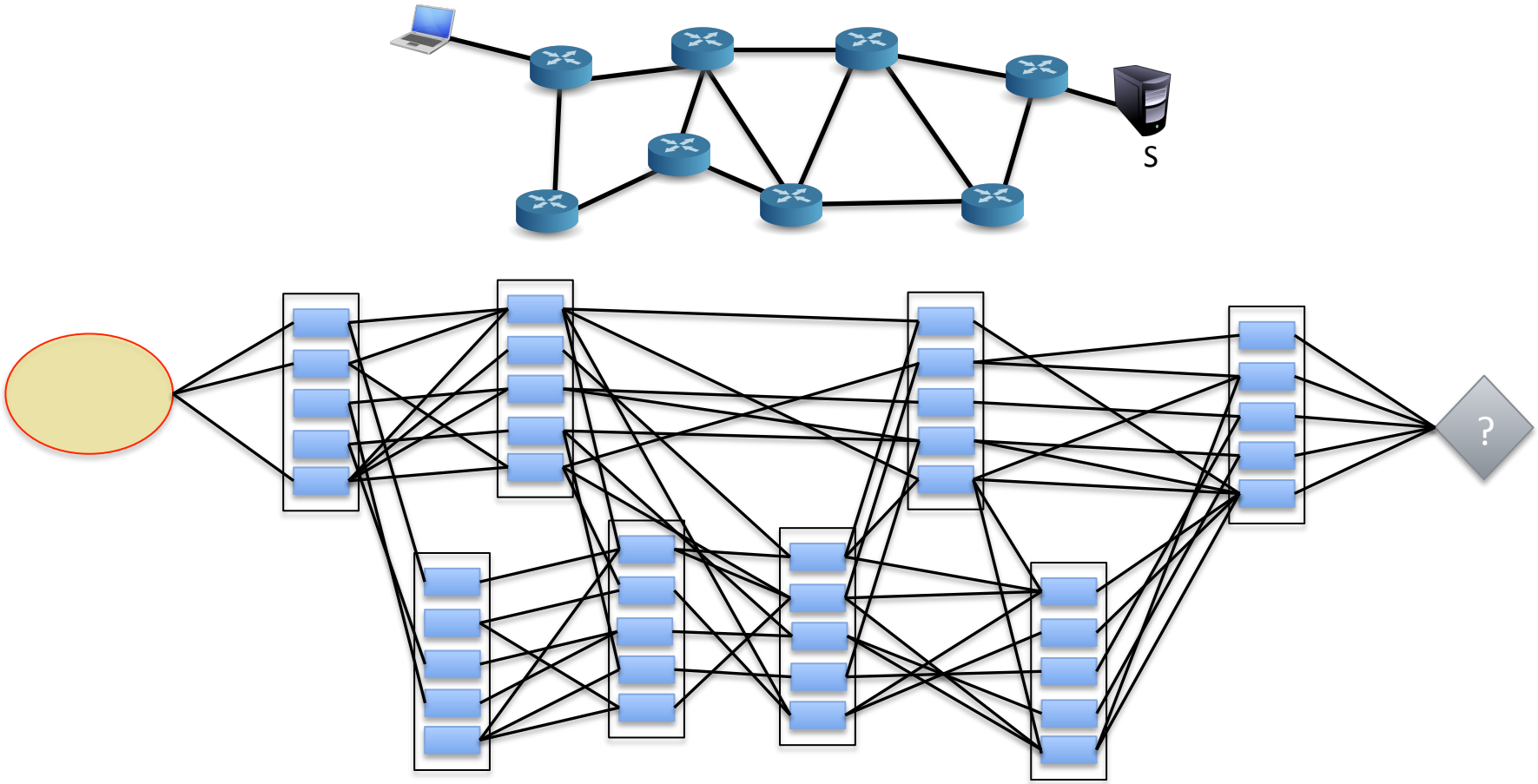
$$\{\forall f \mid f.path \sim [\hat{}\,(p = C)]\} \; : \; f.path \sim [\hat{}\,.*(t = M)]$$



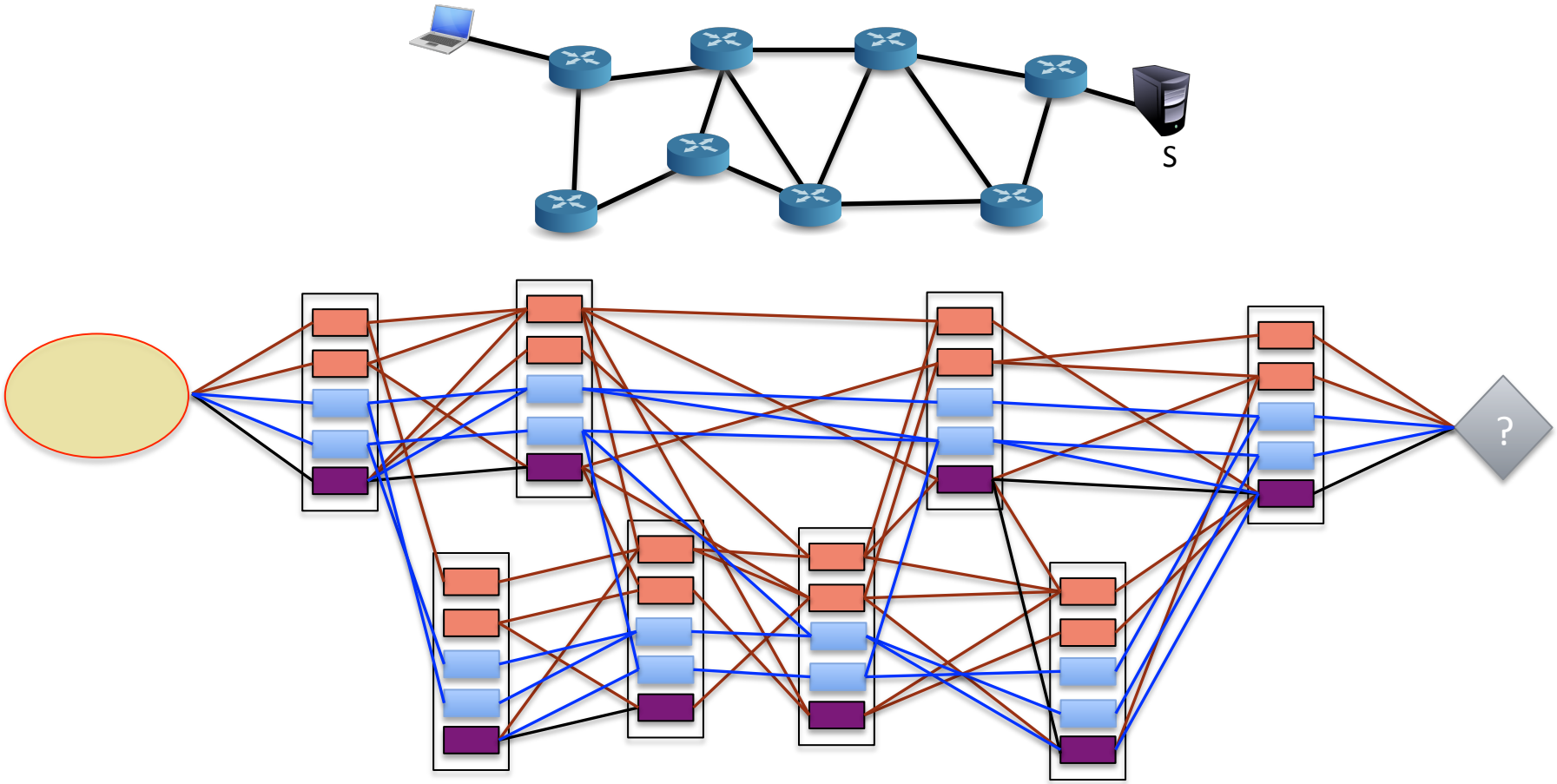
Why the dependency graph helps

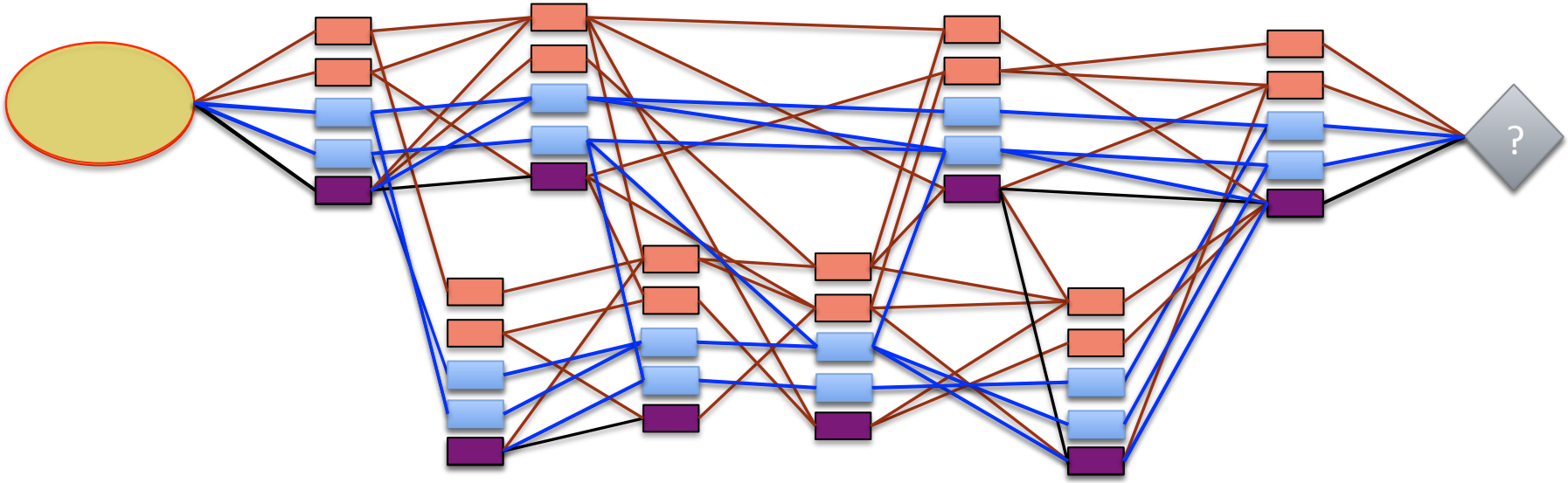
- Incremental update.
  - Only have to trace through dependency sub-graph affected by an update.
- Flexible policy expression.
  - Probe and source nodes are flexible to place and configure.
- Parallelization.
  - Can partition dependency graph into clusters to minimize inter-cluster dependences.

Distributed NetPlumber



Dependency Graph Clustering



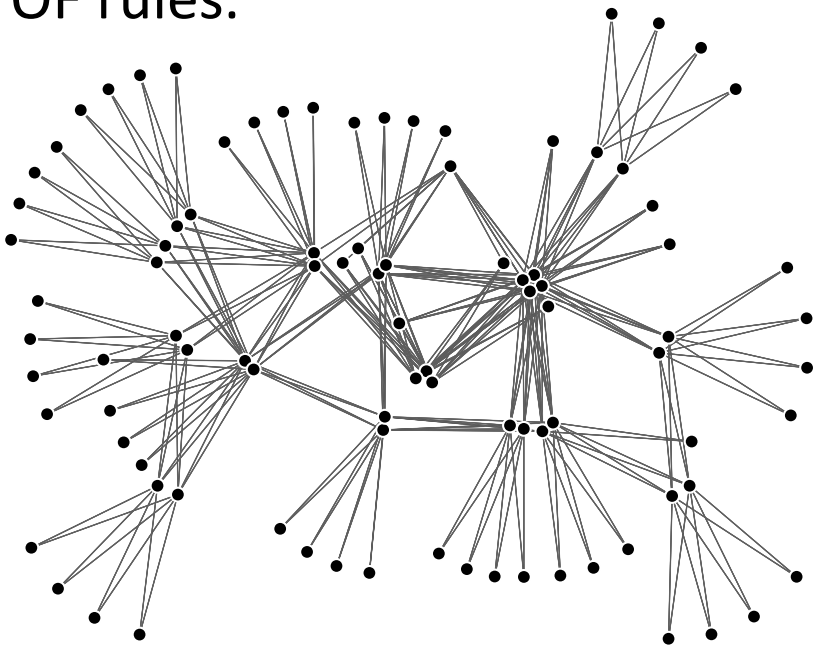


Outline

- NetPlumber: Real time policy checker.
  - How it works?
  - How to check policy with it?
  - Evaluation on Google WAN.
- Automatic Test Packet Generation.
  - How to pick test packets optimally?
  - Run through a toy example.
  - Deployment in CS/EE buildings in Stanford.
- Conclusion and Ideas for Future Work.

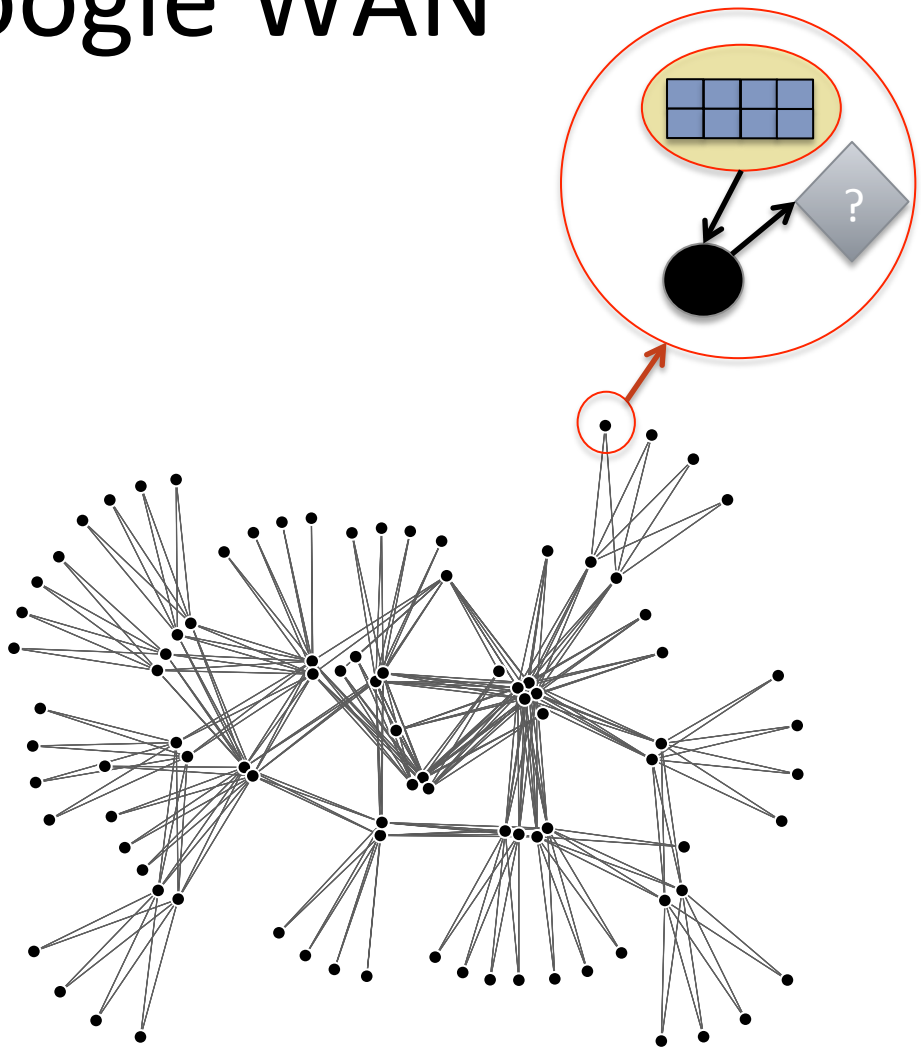
Experiment On Google WAN

- Google Inter-datacenter WAN.
  - Largest deployed SDN, running OpenFlow.
  - ~143,000 OF rules.



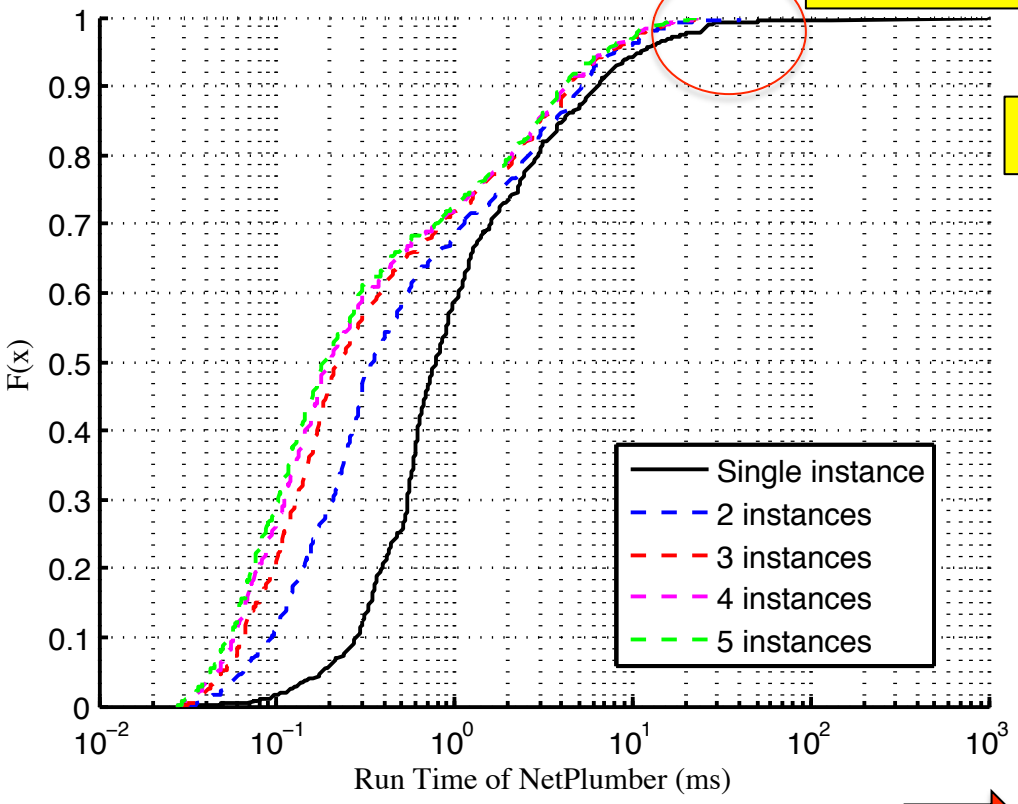
Experiment On Google WAN

- Policy check: all 52 edge switches can talk to each other.
- More than 2500 pairwise reachability check.
- Used two snapshots taken 6 weeks apart.
- Used the first snapshot to create initial NetPlumber state and used the diff as a sequential update.



Experiment On Google WAN

Default/Aggregate Rules.



Run time with Hassel > 100s

Not much more benefit!

#instances:	1	2	3	4	5	8
median (ms)	0.77	0.35	0.23	0.2	0.185	0.180
mean (ms)	5.74	1.81	1.52	1.44	1.39	1.32

Outline

- NetPlumber: Real time policy checker.
  - How it works?
  - How to check policy with it?
  - Evaluation on Google WAN.
- Automatic Test Packet Generation.
  - How to pick test packets optimally?
  - Deployment in CS/EE buildings in Stanford.
- Conclusion and Ideas for Future Work.

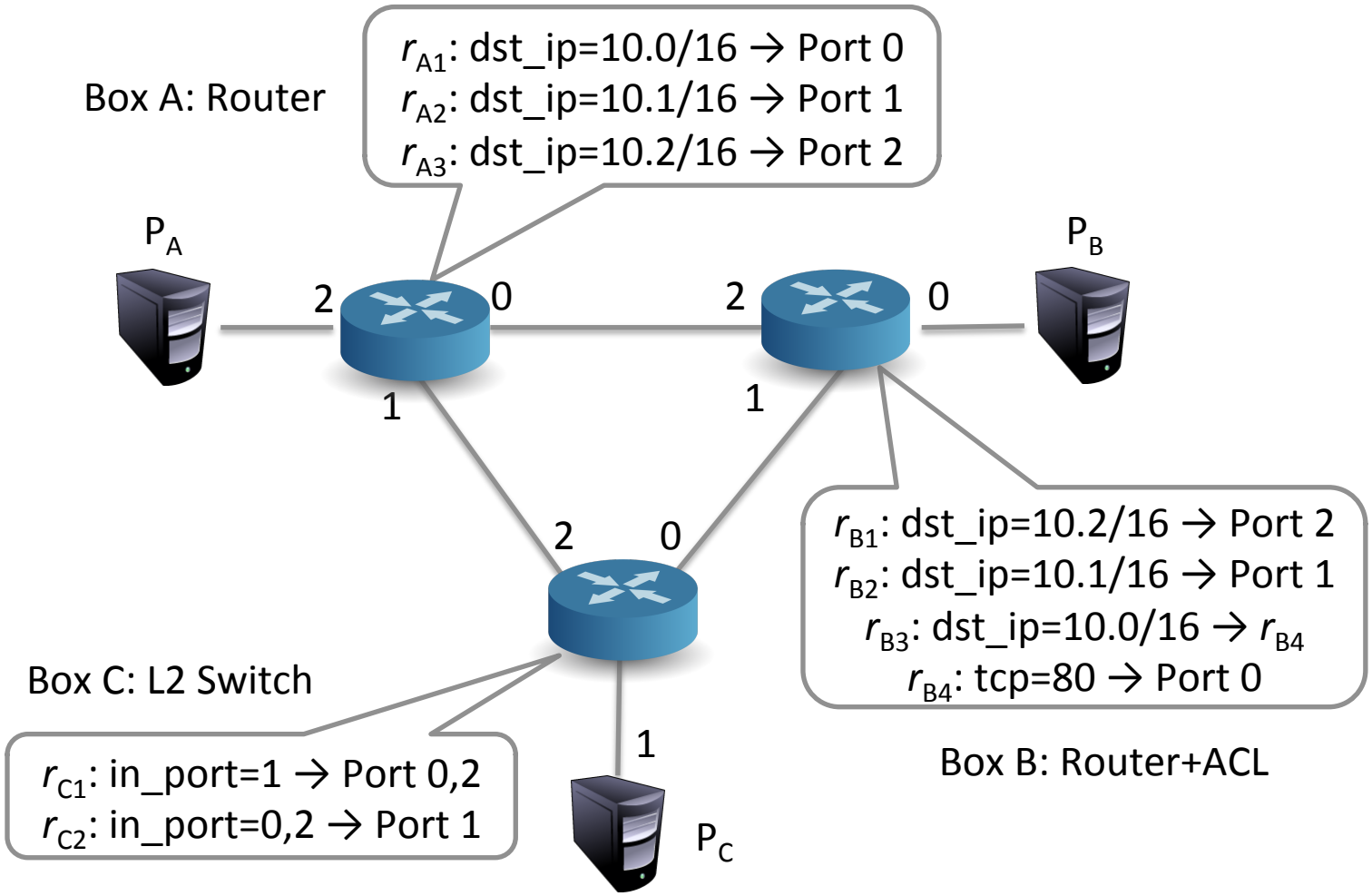
Why testing is still needed?

- NetPlumber and Hassel both designed to find problems in the control plane.
- What if
  - Switch/link is down.
  - Link is congested.
  - ASIC is broken.
- The only ways to detect these sort of data plane problems is at run time!
  - Active testing.
  - Passive monitoring.

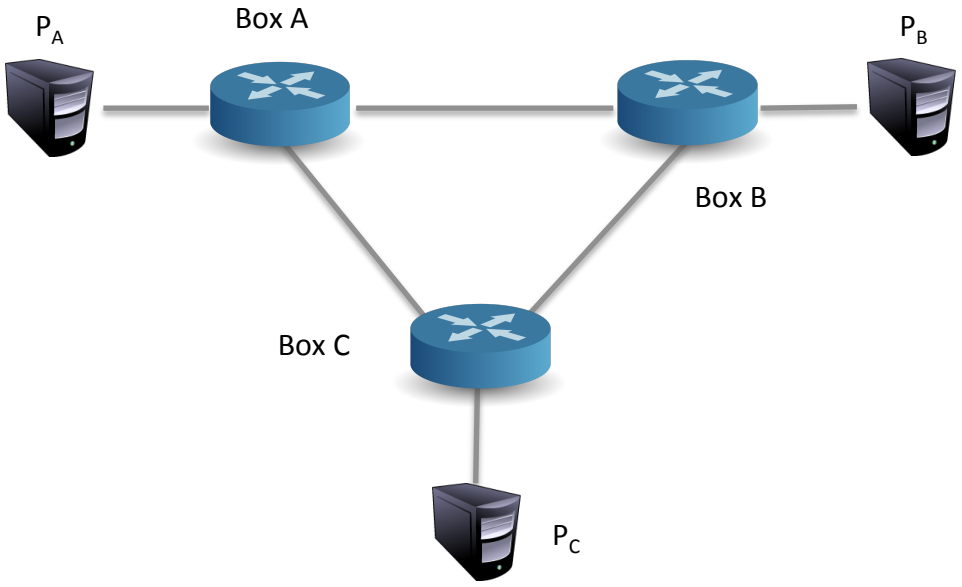
ATPG Goals

- Monitor the data plane by sending test packets.
  - Maximum rule/link/queue coverage.
  - Minimum number of packets required.
  - Constraints on terminal ports and headers of test packets.
- Once error is detected, localize it.

Example

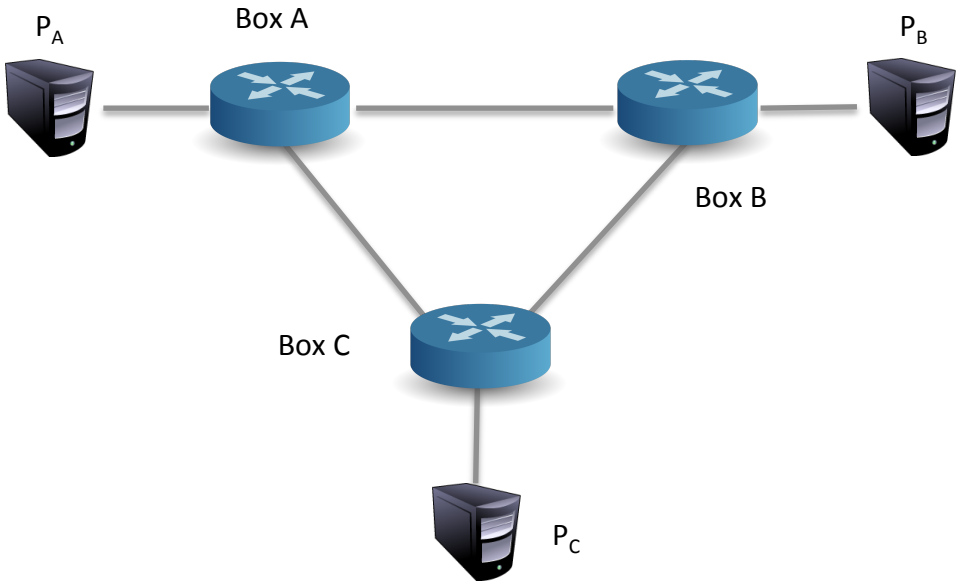

Box A: Router
r_{A1}: dst_ip=10.0/16 → Port 0
r_{A2}: dst_ip=10.1/16 → Port 1
r_{A3}: dst_ip=10.2/16 → Port 2
P_A
P_B
2
0
2
0
1
1
2
0
Box C: L2 Switch
r_{C1}: in_port=1 → Port 0,2
r_{C2}: in_port=0,2 → Port 1
1
P_C
r_{B1}: dst_ip=10.2/16 → Port 2
r_{B2}: dst_ip=10.1/16 → Port 1
r_{B3}: dst_ip=10.0/16 → r_{B4}
r_{B4}: tcp=80 → Port 0
Box B: Router+ACL

Step1: Find all-pairs reachability



	Header	Ingress Port	Egress Port	Rule History
p_1	`dst_ip=10.0/16, tcp=80`	P_A	P_B	r_{A1}, r_{B3}, r_{B4}, link AB
p_2	`dst_ip=10.1/16`	P_A	P_C	r_{A2}, r_{C2}, link AC
p_3	`dst_ip=10.2/16`	P_B	P_A	r_{B2}, r_{A3}, link AB
p_4	`dst_ip=10.1/16`	P_B	P_C	r_{B2}, r_{C2}, link BC
p_5	`dst_ip=10.2/16`	P_C	P_A	r_{C1}, r_{A3}, link BC
(p_6)	`dst_ip=10.2/16, tcp=80`	P_C	P_B	r_{C1}, r_{B3}, r_{B4}, link BC

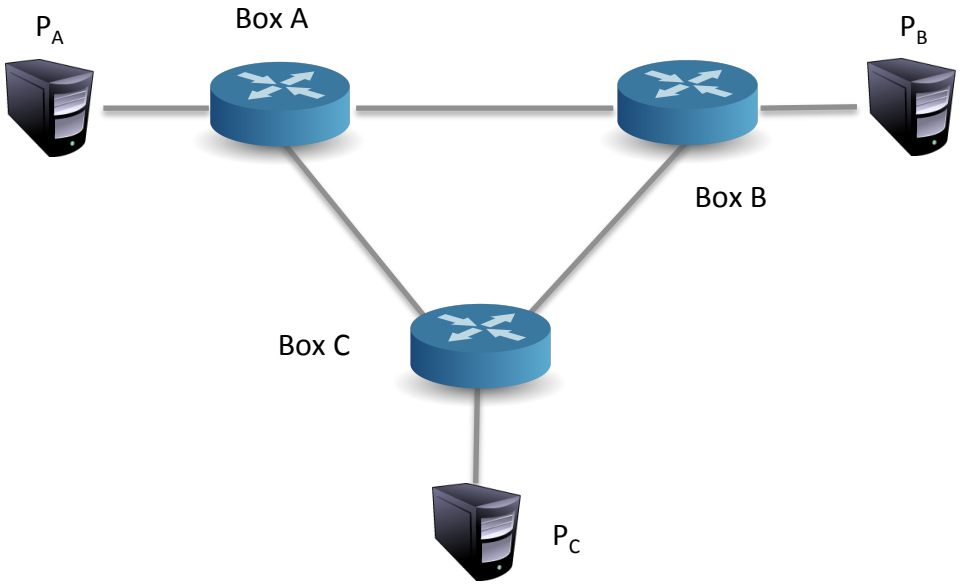
Step2: Find min covers (min set cover)



Cover All Rules

	Header	Ingress Port	Egress Port	Rule History
p_1	dst_ip=10.0/16, tcp=80	P_A	P_B	r_{A1}, r_{B3}, r_{B4}, link AB
p_2	dst_ip=10.1/16	P_A	P_C	r_{A2}, r_{C2}, link AC
p_3	dst_ip=10.2/16	P_B	P_A	r_{B2}, r_{A3}, link AB
p_4	dst_ip=10.1/16	P_B	P_C	r_{B2}, r_{C2}, link BC
p_5	dst_ip=10.2/16	P_C	P_A	r_{C1}, r_{A3}, link AC

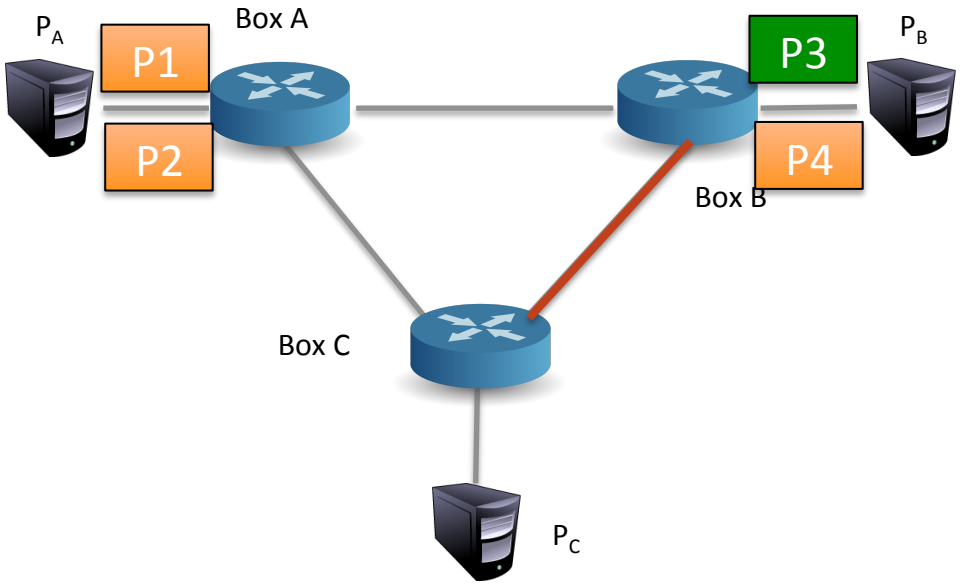
Step2: Find min covers (min set cover)



Cover All Links

	Header	Ingress Port	Egress Port	Rule History
p_1	`dst_ip=10.0/16, tcp=80`	P_A	P_B	r_{A1}, r_{B3}, r_{B4}, link AB
p_2	`dst_ip=10.1/16`	P_A	P_C	r_{A2}, r_{C2}, link AC
p_4	`dst_ip=10.1/16`	P_B	P_C	r_{B2}, r_{C2}, link BC

Step3: Localize fault.



Cover All Links

	Header	Ingress Port	Egress Port	Rule History
p_1	dst_ip=10.0/16, tcp=80	P_A	P_B	r_{A1}, r_{B3}, r_{B4}, link AB
p_2	dst_ip=10.1/16	P_A	P_C	r_{A2}, r_{C2}, link AC
p_3	dst_ip=10.2/16	P_B	P_A	r_{B2}, r_{A3}, link AB
p_4	dst_ip=10.1/16	P_B	P_C	r_{B2}, r_{C2}, link BC
p_5	dst_ip=10.2/16	P_C	P_A	r_{C1}, r_{A3}, link AC
(p_6)	dst_ip=10.2/16, tcp=80	P_C	P_B	r_{C1}, r_{B3}, r_{B4}, link BC

How many packets is enough?

	Stanford	Internet2
Total Packets	621,402	3,037,335
Regular Packets	3,871	35,462
Min-Set-Cover Reduction	160x	85x
Packets/Port (Avg)	12.99	102.8

<1% Link Utilization
when testing 10 times per second!

Using ATPG for Performance Testing

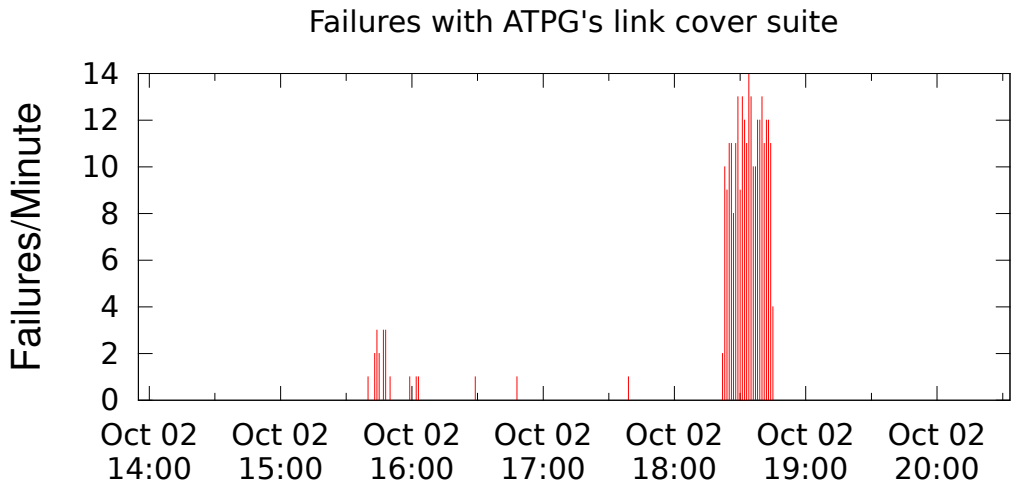
- Beyond correctness, ATPG can also be used for detecting and localizing performance problems.
- Intuition: generalize results of a test from success/failure to performance (e.g. latency or bandwidth).
- Track test packet latency/inferred bandwidth. Raise an error when changed significantly.

ATPG on Gates/Packard/CIS Network

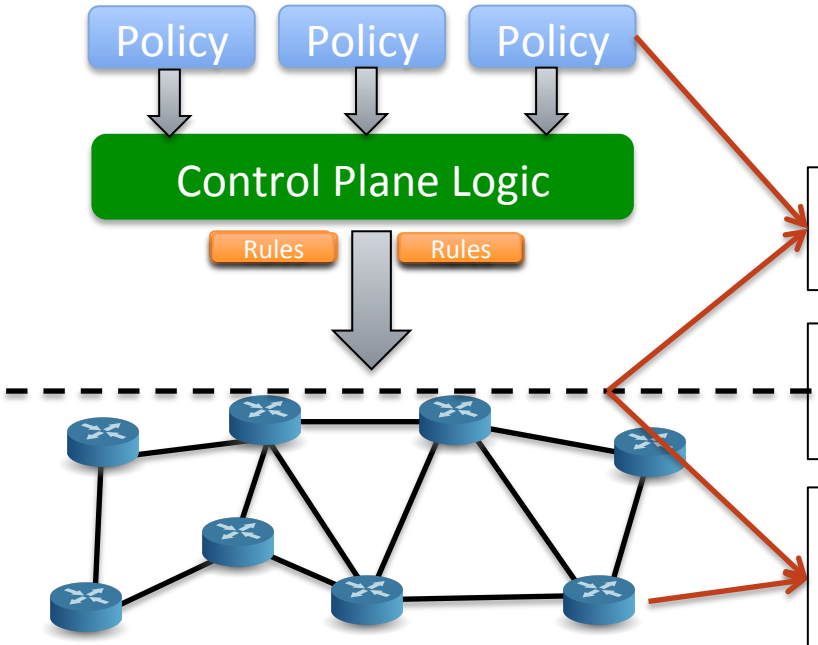
From: Miles Davis <miles@cs.stanford.edu>
Subject: Accidental Gates network outage, back online
Date: October 2, 2012 7:54:48 PM PDT
To: action@cs.stanford.edu

Between 18:20-19:00 tonight (Oct 2) we experienced a complete network outage in the building when a loop was accidentally created by CSD-CF staff. We're investigating the exact circumstances to understand why this caused a problem, since automatic protections are supposed to be in place to prevent loops from disabling the network.

--
// Miles Davis - miles@cs.stanford.edu
// Computer Science Department - Computer Facilities
// Stanford University



Bigger Picture



Hassel,NetPlumber: verify equivalence of policy and expected end-to-end network behavior.

Header Space Analysis: models forwarding behavior of network based on these rules.

ATPG: verify correctness of actual data plane behavior by generating test packets based on network rules.

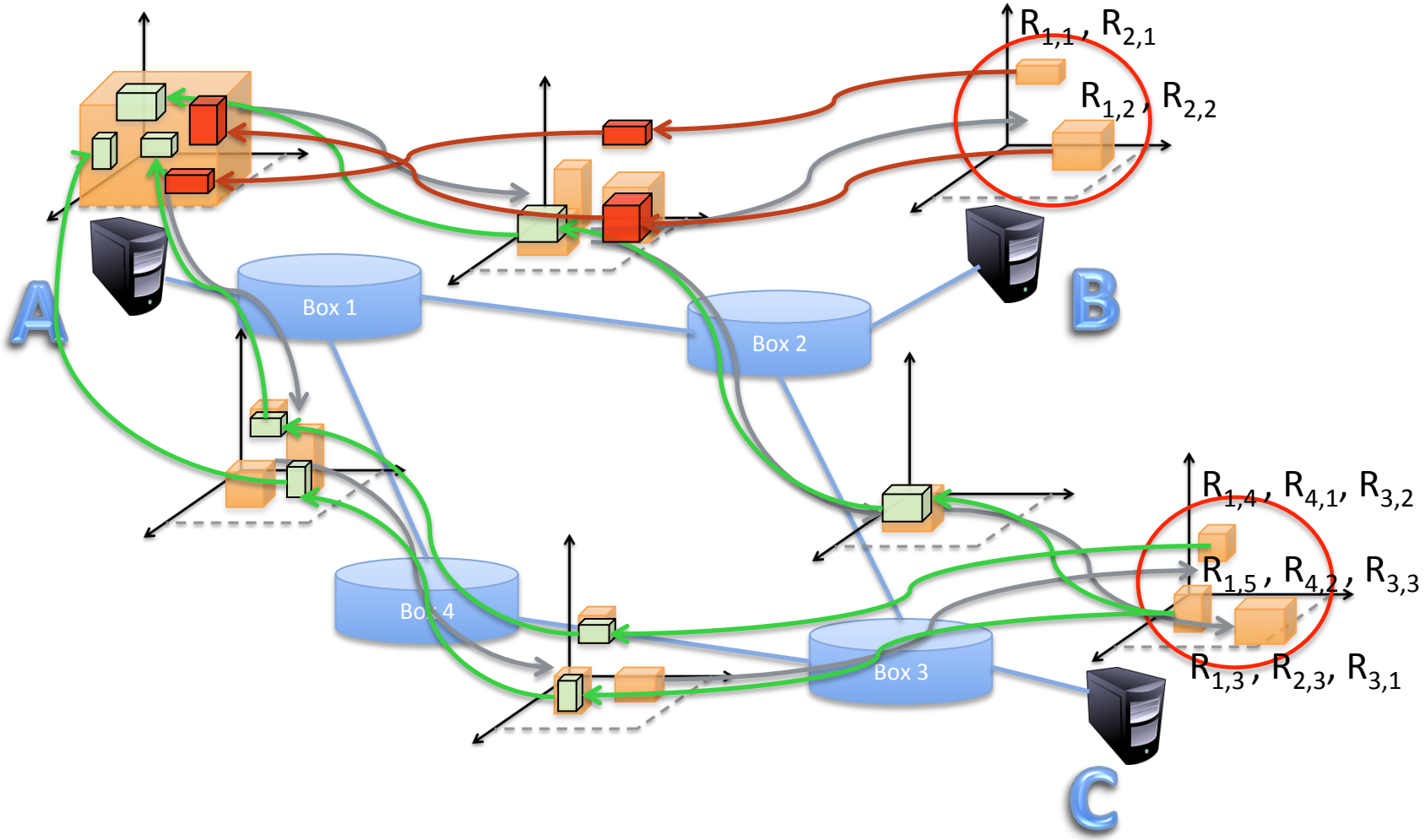
Main Takeaway

By defining the right model for how networks process packets, we can create techniques for systematic testing and debugging networks.

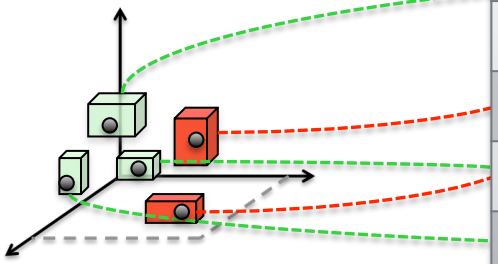
Ideas for Future Works

- Verification.
- Policy Extraction.
- Stateful HSA.

1. Choosing Test Packets



1. Choosing Test Packets

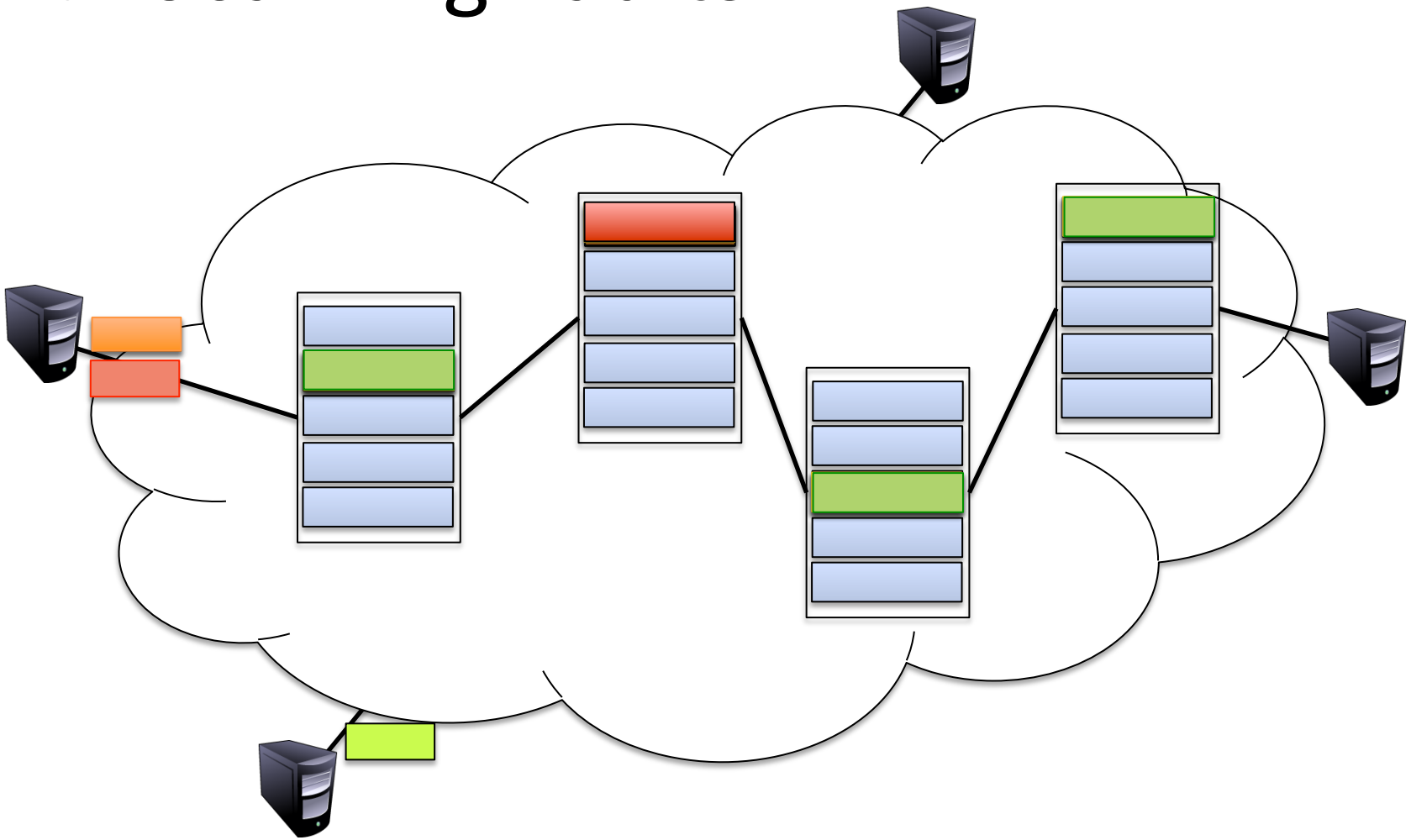


Packet	Rules
101100110	$R_{1,1}$, $R_{2,1}$
001001001	$R_{1,2}$, $R_{2,2}$
111000100	$R_{1,4}$, $R_{4,1}$, $R_{3,2}$
001100110	$R_{1,5}$, $R_{4,2}$, $R_{3,3}$
010101000	$R_{1,3}$, $R_{2,3}$, $R_{3,1}$

Choose a set of test packets to cover every rule.
(Regular vs. Reserved Test Packets)

Min Set Cover Problem

2. Localizing Faults



Complexity of Reachability and Loop Detection Tests

- Run time

  Reachability ~ dR^2

  Loop Detection ~ dPR^2

  - R: maximum number of rules per box.
  - d: diameter of network.
  - P: number of ports to be tested

  Assumption: Linear Fragmentation