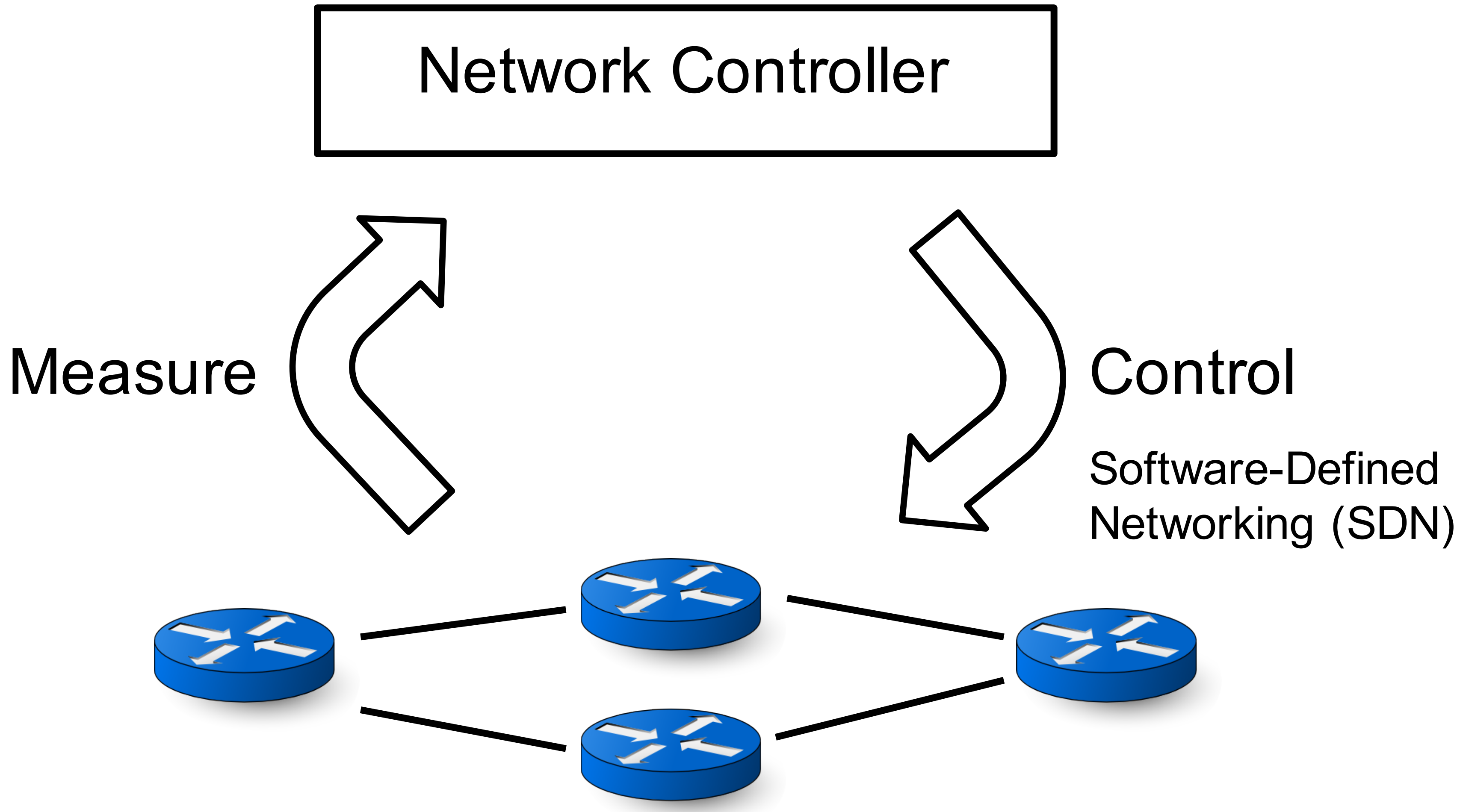
Declarative Network Path Queries

Srinivas Narayana

May 13, 2016

Advisor: Prof. Jennifer Rexford

Management = Measure + Control

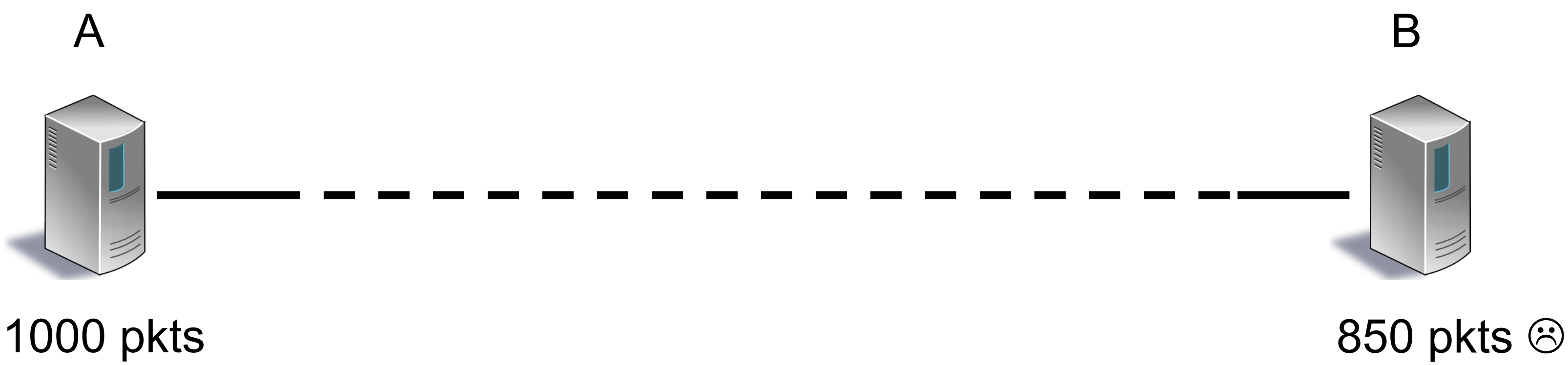


Enabling Easier Measurement Matters

- Networks are asked to do a lot!
  - Partition-aggregate applications
  - Growth in traffic demands
  - Stringent performance requirements
  - Avoid expensive outages
- Difficult to know *where* things go wrong!
  - Humans are slow in troubleshooting
  - Human time is expensive
- Can we build *programmatic tools* to help?

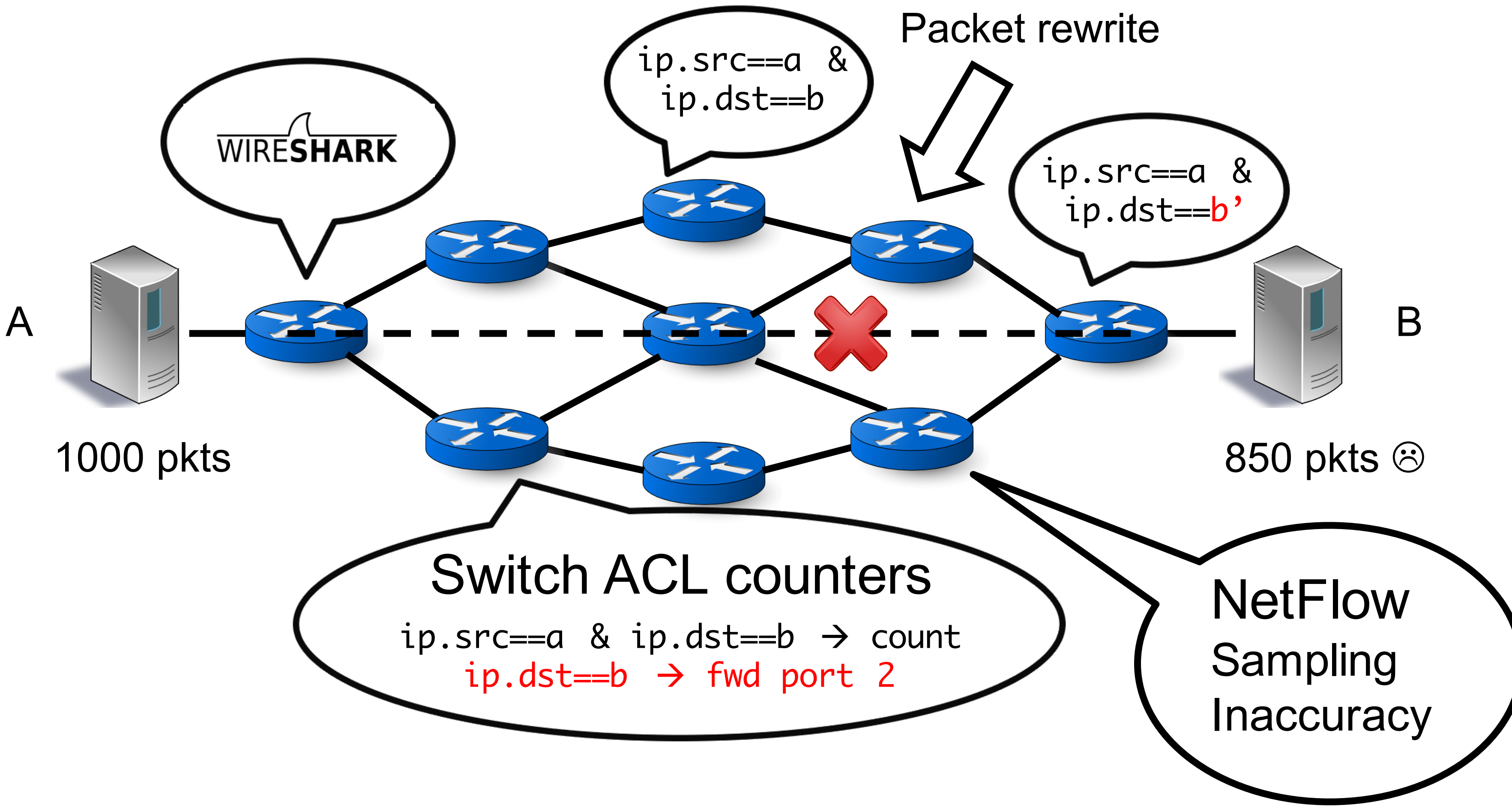
Example: Where's the Packet Loss?

Suspect: Faulty network device(s) along the way.



Example: Where's the Packet Loss?

Idea: "Follow" the path of packets through the network.



Example: Where's the Packet Loss?

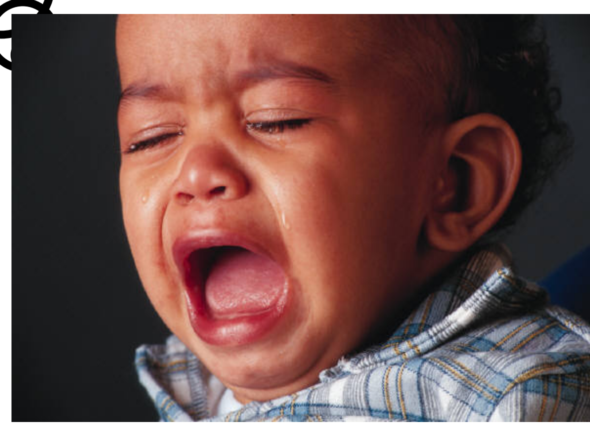
Complex & Inaccurate Join with multiple datasets: traffic, forwarding, topology

High Overhead of collecting (unnecessary) data to answer a given question

Example: Where’s the Packet Loss?

Complex & Inaccurate Join with multiple datasets: traffic, forwarding, topology

High Overhead of collecting (unnecessary) data to answer a given question



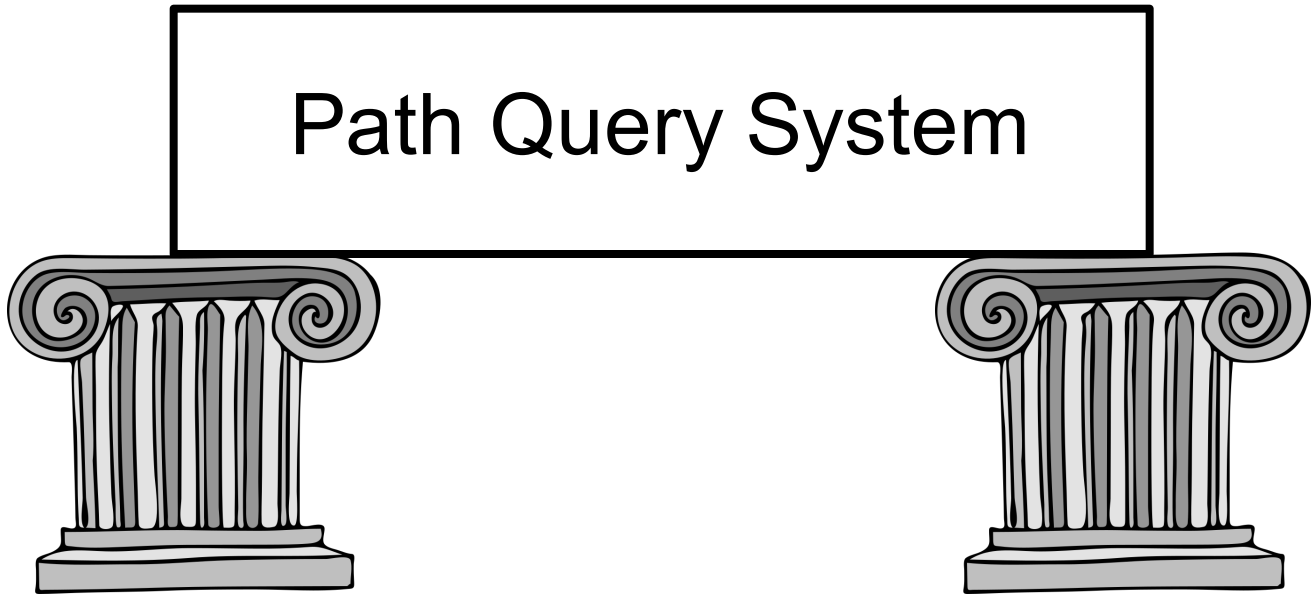
Pattern: Combining Traffic & Forwarding

- Traffic matrix
- Uneven load balancing
- DDoS source identification
- Port-level traffic matrix
- Congested link diagnosis
- Slice isolation
- Loop detection
- Middlebox traversal order
- Incorrect NAT rewrite
- Firewall evasion
- ...

} Resource management
Policy enforcement
Problem diagnosis

Sources: Feldman et al 2001, Patel et al 2013, Savage et al 2000, Varghese and Estan 2004, Duffield and Grossglauser 2001, Kazemian et al 2012, Fayazbakhsh et al 2014, Handigol et al 2014, Zhu et al 2015, and conversations with network operators at Microsoft and Amazon

Approach



Declarative Query Specification

Independent of Forwarding
Independent of Other Measurements
Independent of Hardware Details

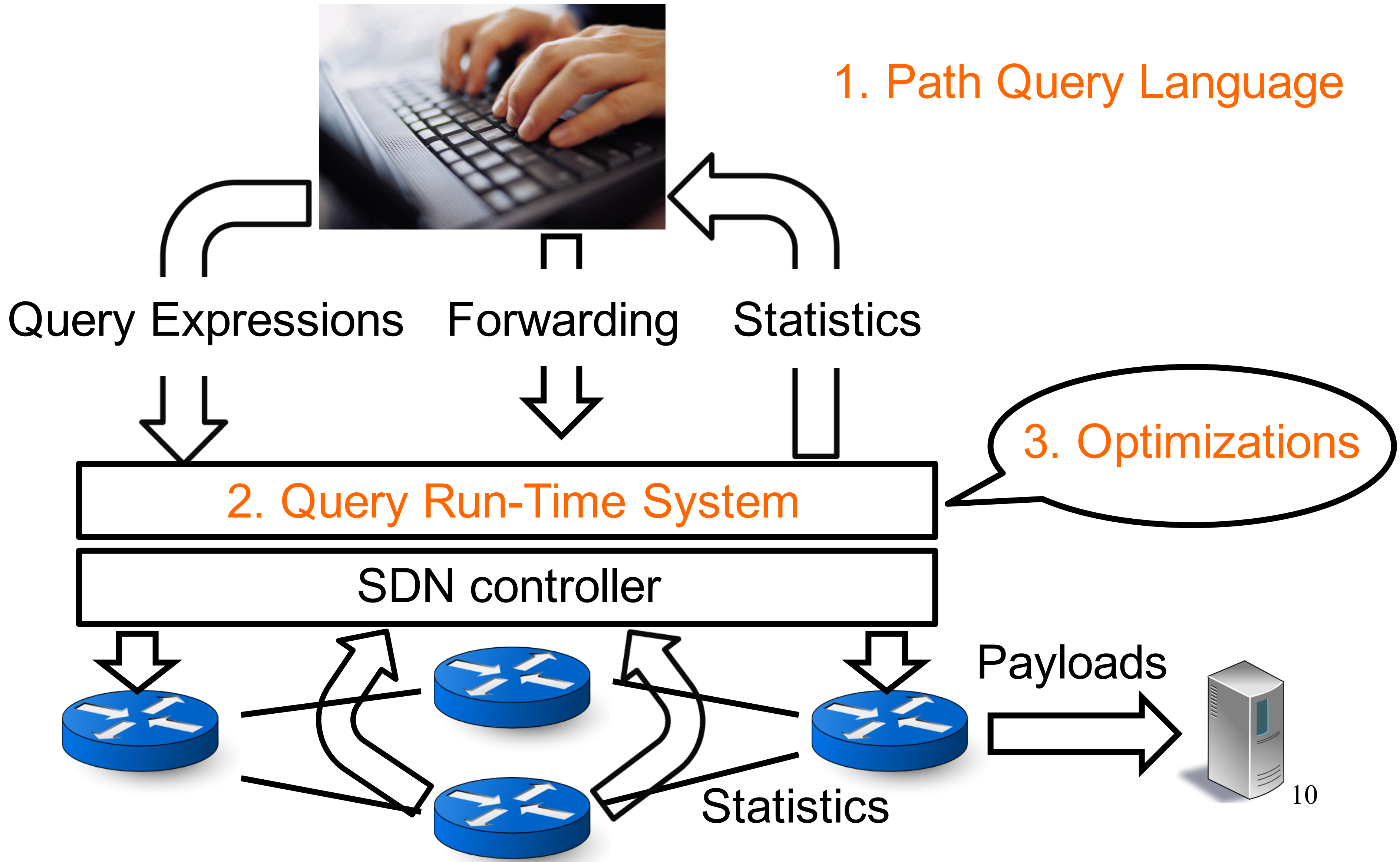
Query-Driven Measurement

Accurate Answers
Pay Exactly For What You Query
Commodity (“Match-Action”) Hardware

Path Query Language

Query Run-Time System

Approach



1. Path Query Language
Query Expressions
Forwarding
Statistics
3. Optimizations
2. Query Run-Time System
SDN controller
Payloads
Statistics

Approach

1. Path Query Language

Expressive measurement specification

2. Query Run-Time System

Accurate data plane measurement

3. Optimizations

Efficient measurement

Contributions

- Regular-expression-based language for traffic monitoring
  - With SQL-like aggregation and capture locations
- Run-Time: Deterministic finite state automata on packets using match-action switch rules
  - Collect *exactly* those packets that satisfy queries
- Compiler optimizations: to speed up or completely remove expensive overlapping actions on packets
- Result: Debug networks with practical overheads

How to design *general* measurement primitives

… that are *efficiently* implemented in the network?

Measurement Use Cases

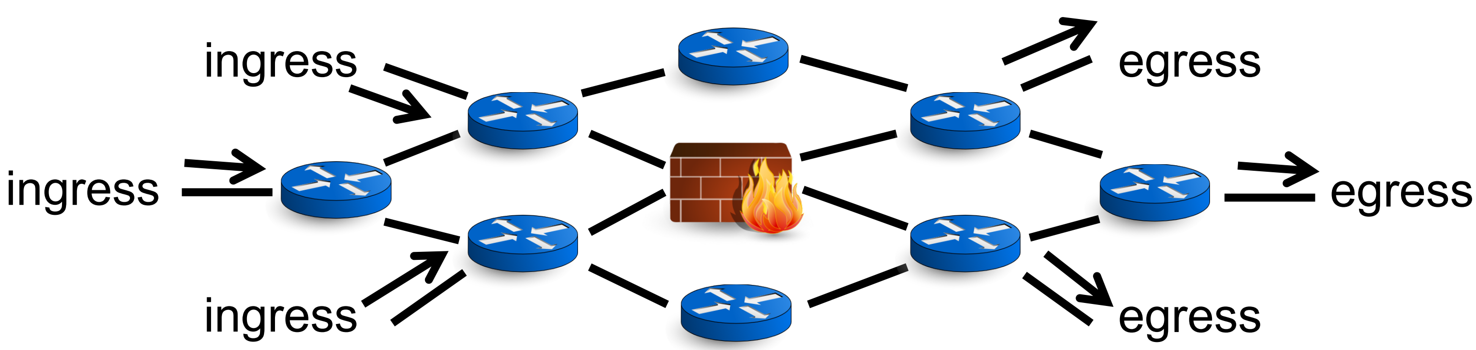
- Traffic matrix
- Uneven load balancing
- DDoS source identification
- Port-level traffic matrix
- Congested link diagnosis
- Slice isolation
- Loop detection
- Middlebox traversal order
- Incorrect NAT rewrite
- Firewall evasion
- ...

What are the common patterns?

Sources: Feldman et al 2001, Patel et al 2013, Savage et al 2000, Varghese and Estan 2004, Duffield and Grossglauser 2001, Kazemian et al 2012, Fayazbakhsh et al 2014, Handigol et al 2014, Zhu et al 2015, and conversations with network operators at Microsoft and Amazon

(I) Path Query Language

- *Test* predicates on packets at single locations:
  ```
  srcip=10.0.0.1
  port=3 & dstip=10.0.1.10
  ```
- *Combine* tests with regular expression operators!
  ```
  sw=1 ^ sw=4
  srcip=A ^ true* ^ sw=3
  ingress() ^ ~(sw=firewall)* ^ egress()
  ```



(I) Path Query Language

- *Aggregate* results with SQL-like grouping operators

```
in_group(ingress(), [sw])
  ^ true*
  ^ out_group(egress(), [sw])
```

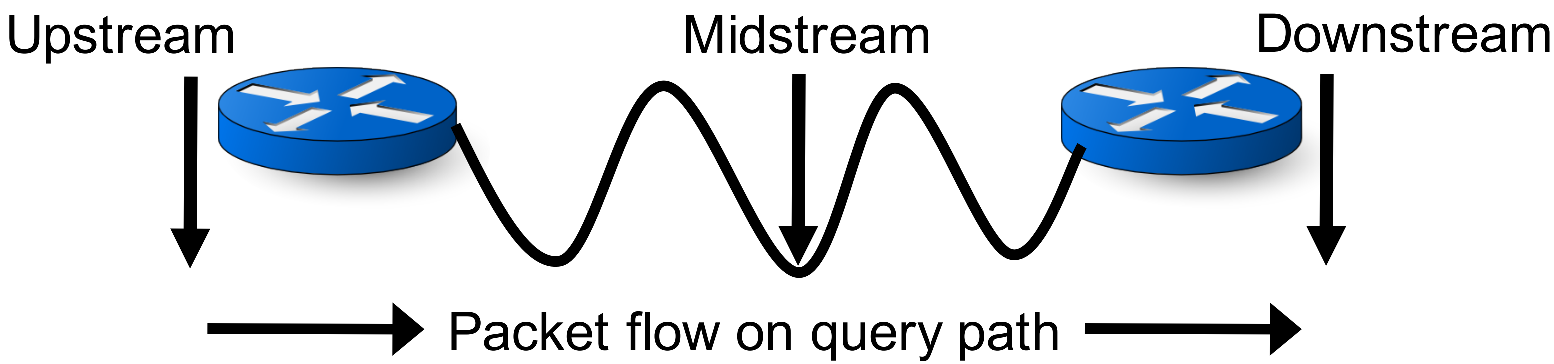
ingress() switch	#pkts
S1	1000
S2	500
S5	700
...	...

(ingress(), egress()) switch pairs	**#pkts**
(S1, S2)	800
(S1, S5)	200
(S2, S5)	300
...	...

- *Return* packets, counters, or samples (NetFlow/sFlow)

(I) Path Query Language

- *Capture* upstream, downstream or midstream



- *Match* predicates at switch ingress, egress or both

```
in_atom(dstip=128.1.2.3)
in_out_atom(dstip=128.1.2.3, dstip=10.1.2.3)
```

(I) Evaluation: Query Examples

Example	Query code	Description
A simple path	in_atom(switch=S1) ^ in_atom(switch=S4)	Packets going from switch S1 to S4 in the network.
Slice isolation	true* ^ (in_out_atom(slice1, slice2) | in_out_atom(slice2, slice1))	Packets going from network slice slice 1 to slice2, or vice versa, when crossing a switch.
Firewall evasion	in_atom(ingress()) ^ (in_atom(~switch=FW))* ^ out_atom(egress())	Catch packets evading a firewall device FW when moving from any network ingress to egress interface.
DDoS sources	in_group(ingress(), [switch]) ^ true* ^ out_atom(egress(), switch=vic)	Determine traffic contribution by volume from all ingress switches reaching a DDoS victim switch vic.
Switch-level traffic matrix	in_group(ingress(), [switch]) ^ true* ^ out_group(egress(), [switch])	Count packets from any ingress to any egress switch, with results grouped by (ingress, egress) switch pair.
Congested link diagnosis	in_group(ingress(), [switch]) ^ true* ^ out_atom(switch=sc) ^ in_atom(switch=dc) ^ true* ^ out_group(egress(), [switch])	Determine flows (switch sources → sinks) utilizing a congested link (from switch sc to switch dc), to help reroute traffic around the congested link.
Port-to-port traffic matrix	in_out_group(switch=s, true, [inport], [outport])	Count traffic flowing between any two ports of switch s, grouping the results by the ingress and egress interface.
Packet loss localization	in_atom(srcip=H1) ^ in_group(true, [switch]) ^ in_group(true, [switch]) ^ out_atom(dstip=H2)	Localize packet loss by measuring per-path traffic flow along each 4-hop path between hosts H1 and H2.
Loop detection	port = in_group(true, [switch, inport]); port ^ true* ^ port	Detect packets that visit any fixed switch and port twice in their trajectory.
Middlebox order	(true* ^ in_atom(switch=FW) ^ true*) & (true* ^ in_atom(switch=P) ^ true*) & (true* ^ in_atom(switch=IDS) ^ true*) & ~(in_atom(ingress()) ** in_atom(switch=FW) ** in_atom(switch=P) ** in_atom(switch=IDS) ** out_atom(egress()))	Packets that traverse a firewall FW, proxy P and intrusion detection device IDS, but do so in an undesirable order [51].
NAT debugging	in_out_atom(switch=NAT & dstip=192.168.1.10, dstip=10.0.1.10)	Catch packets entering a NAT with destination IP 192.168.1.10 and leaving with the (modified) destination IP 10.0.1.10.
ECMP debugging	in_out_group(switch=S1 & ecmp_pred,	Measure ECMP traffic splitting on switch S1 for a small

Sources: Feldman et al 2001, Patel et al 2013, Savage et al 2000, Varghese and Estan 2004, Duffield and Grossglauser 2001, Kazemian et al 2012, Fayazbakhsh et al 2014, Handigol et al 2014, Zhu et al 2015, and conversations with network operators at Microsoft and Amazon

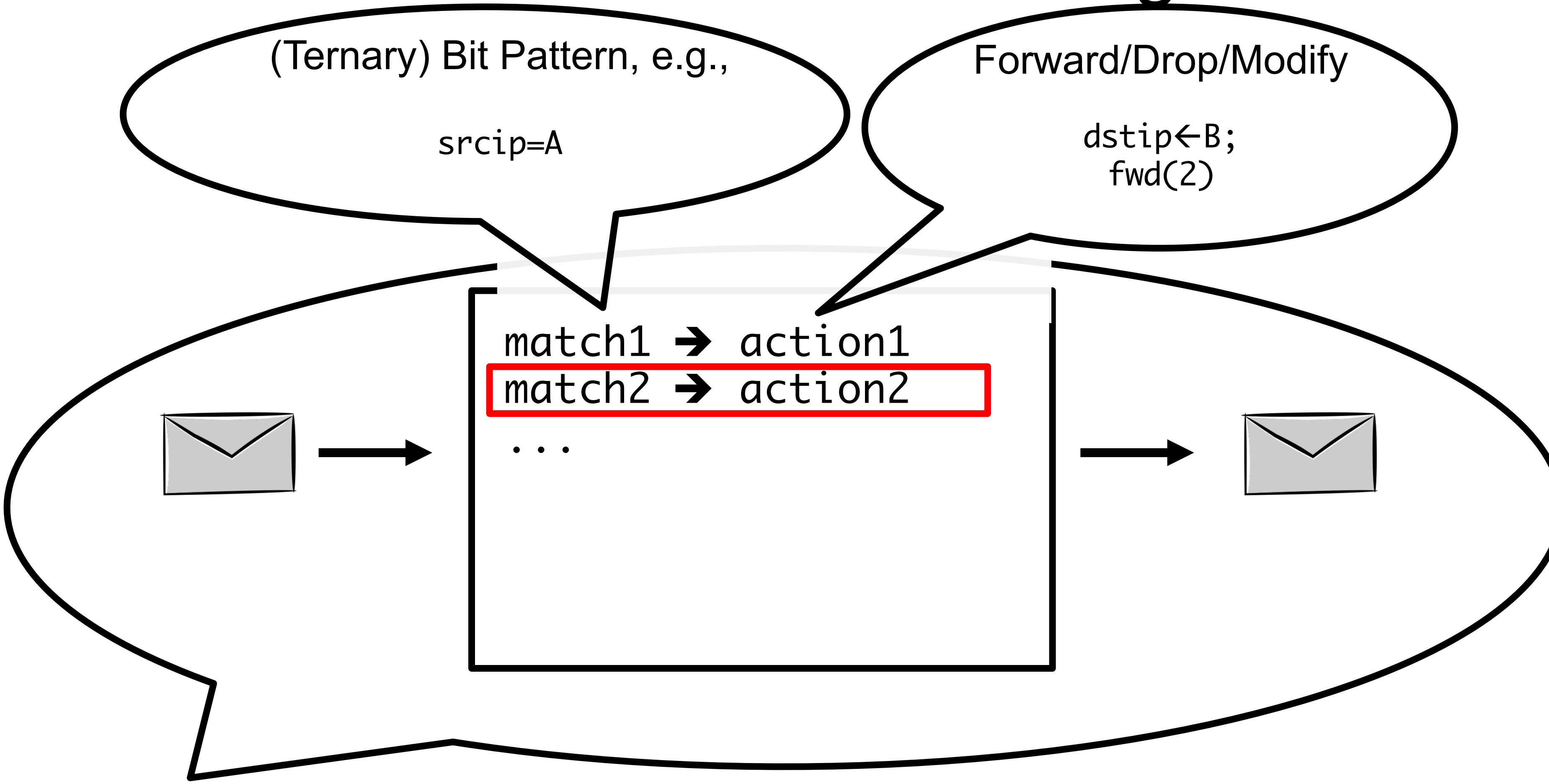
(I) Language: Related Work

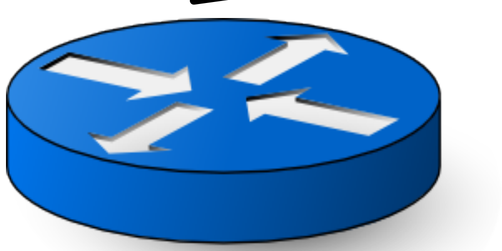
Primitive	Description	Prior Work	Our Extensions
Atomic Predicates	Boolean tests on located packets	[Foster11] [Monsanto13]	Switch input and output differentiation
Packet Trajectories	Regular expressions on atomic predicates	[Tarjan79], [Handigol14]	Additional regex operators (&, ~)
Result Aggregation	Group results by location or header fields	SQL groupby, [Foster11]	Group anywhere along a path
Capture Location	Get packets before or after queried path	--	N/A
Capture Result	Actions on packets satisfying queries	[Monsanto13]	Sampling (sFlow); path-based forwarding

How do we implement path queries efficiently?

In general, switches don't know prior or future packet *paths*.

Match-Action Packet Processing



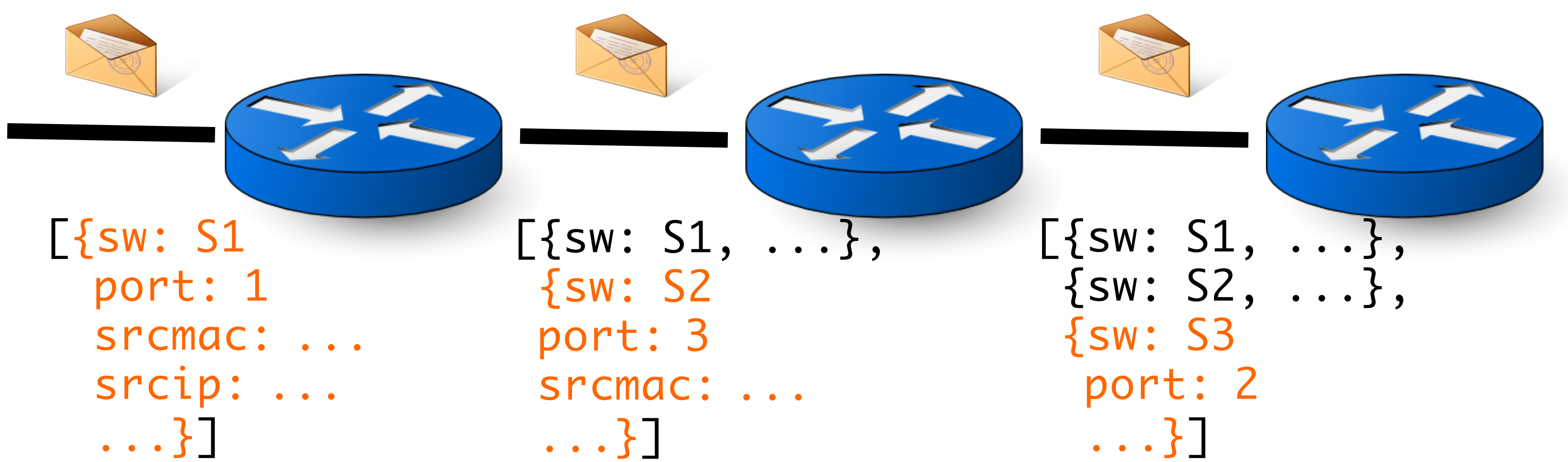


Multiple but limited # stages (e.g., 16)

Limited # rules per stage (e.g., 2K)

How to observe pkt paths downstream?

- Analyze packet paths *in the data plane* itself
    - Write path information into packets!



- Pros: accurate path information ☺
- Cons: too much per-packet information ☹
- Cons: can't match regular expressions on switches

Reducing Path Information on Packets

- Observation 1: Queries already tell us what’s needed!
  - Only record path state needed by queries
- Observation 2: Queries are regular expressions
  - Regular expressions ➔ Finite automaton (DFA)
  - Distinguish only paths corresponding to DFA states

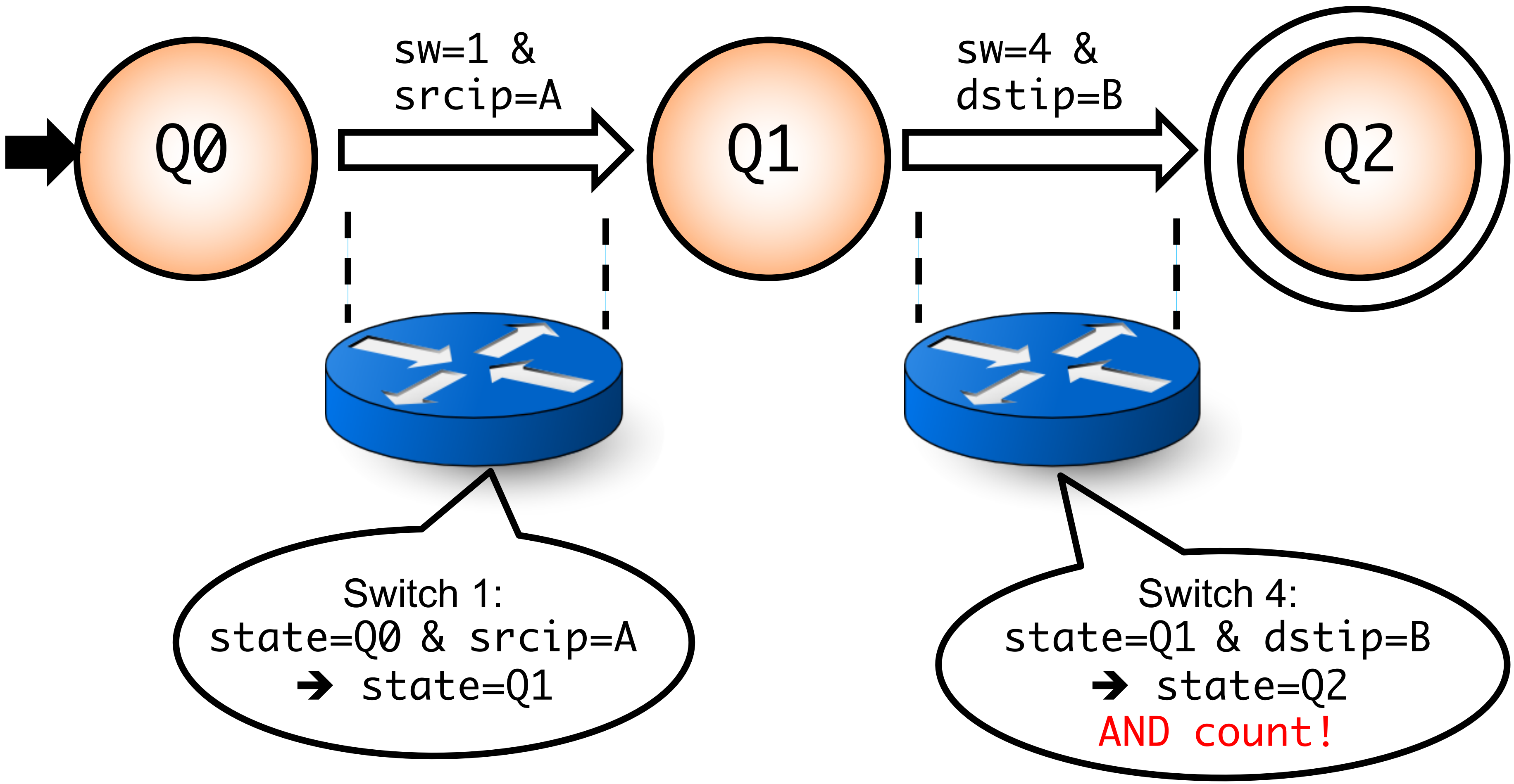
Reducing Path Information on Packets

Record only DFA state on packets (1-2 bytes)

Use existing “tag” fields! (e.g., VLAN)

(II) Query Run-Time System

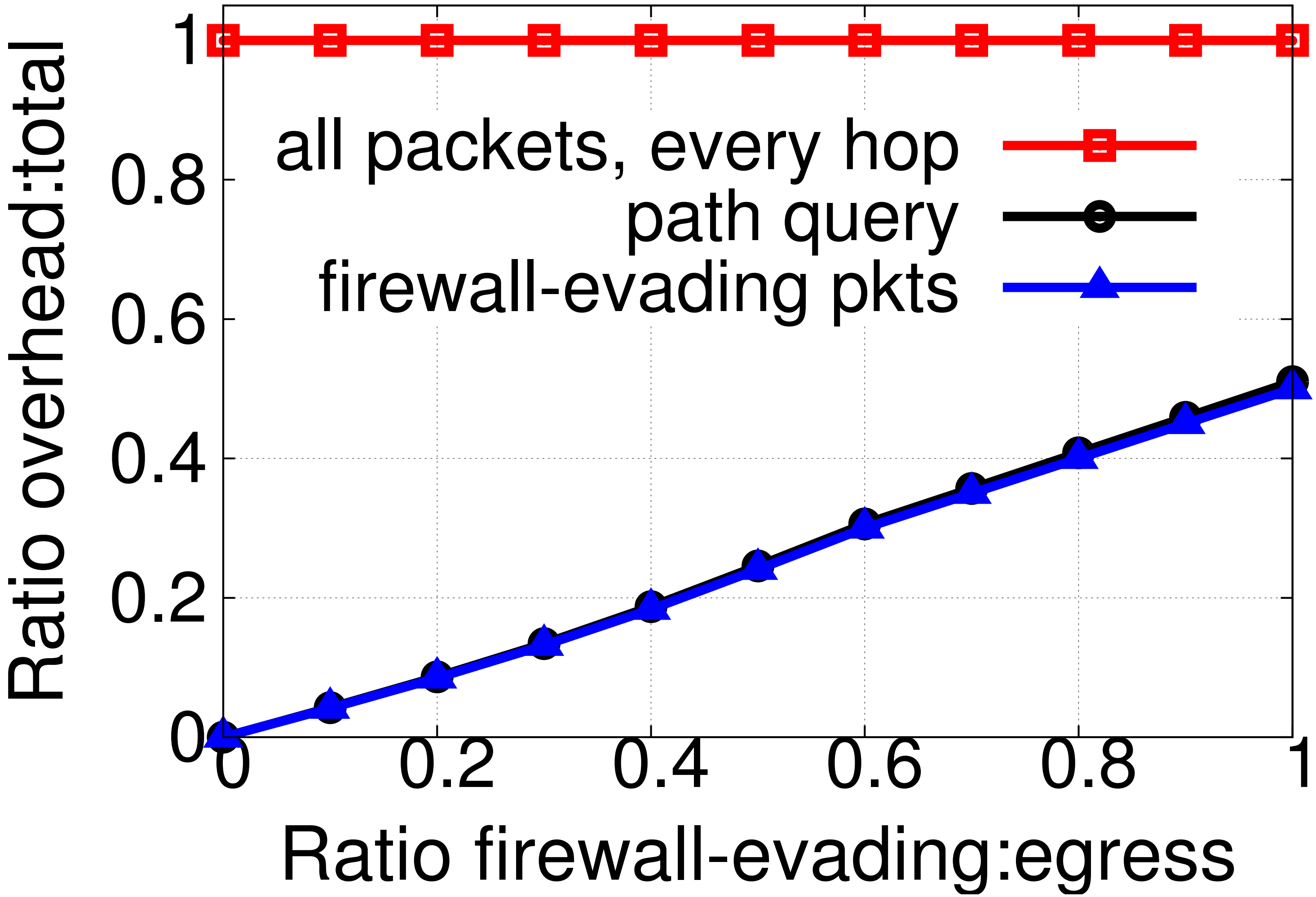
- (sw=1 & srcip=A) ^ (sw=4 & dstip=B)



(II) Query Run-Time System

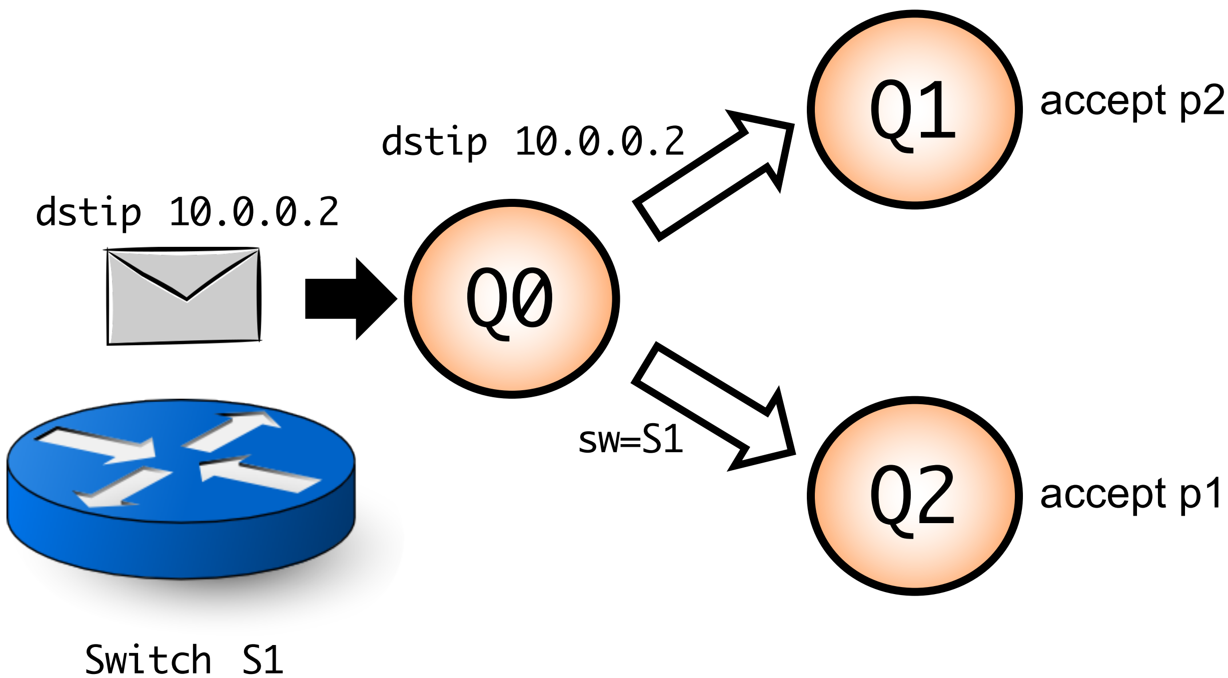
- Each packet carries its own DFA state
- Query DFA transitions *distributed* to switches
  - … as *match-action* rules!
- Packet satisfies query iff it reaches accepting states
  - "Pay for what you query"

(II) You Pay For What You Query



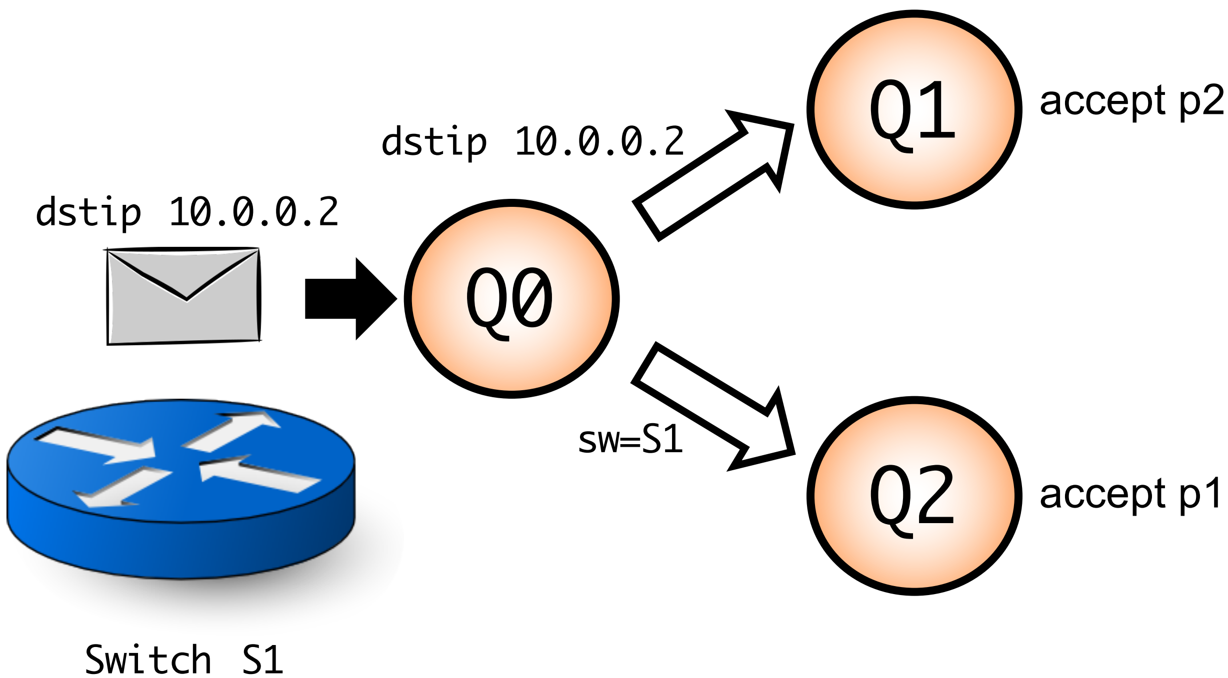
(II) Run-Time: Deterministic Transitions

- p1: `sw=S1`
- p2: `dstip=10.0.0.2`



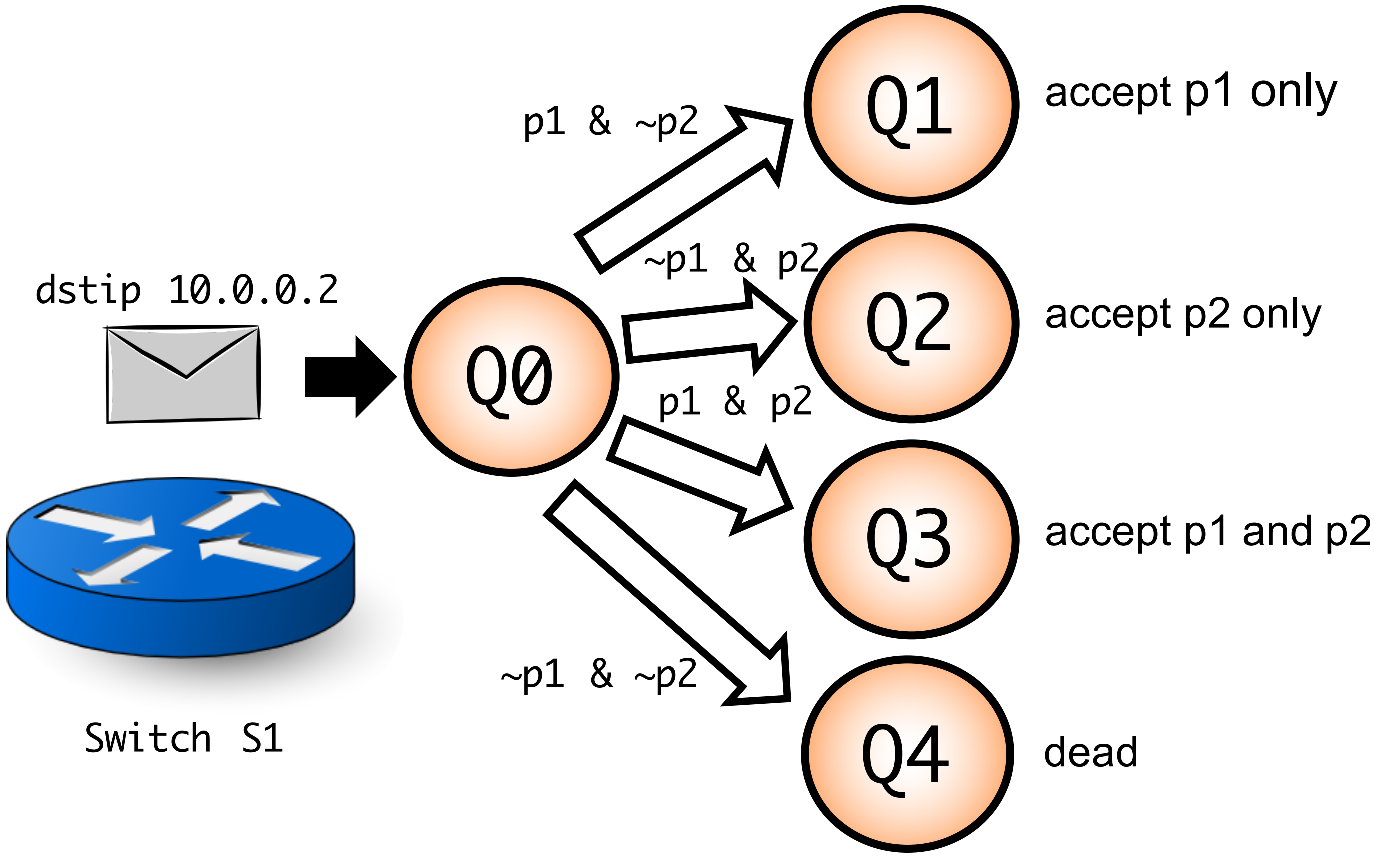
(II) Run-Time: Deterministic Transitions

- p1: `sw=S1`
- p2: `dstip=10.0.0.2`
- Trouble: Packet should only be in one automaton state!



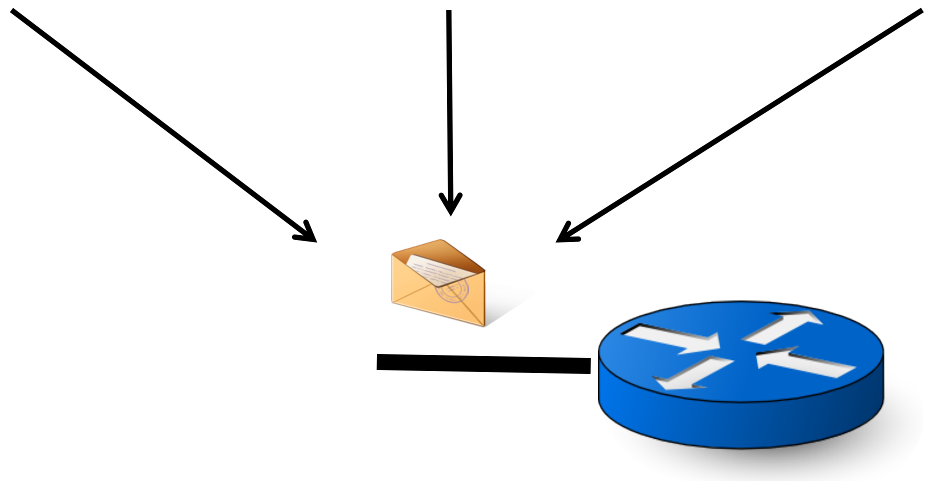
(II) Run-Time: Deterministic Transitions

- p1: `sw=S1`
- p2: `dstip=10.0.0.2`
- Solution: Split predicates into disjoint parts



(II) Run-Time: Composition

(DFA-Transitioning >> Forwarding) + DFA-Accepting



All acting on the same data plane packets!

Use policy composition operators and compiler

Composing software-defined networks. Monsanto et al., 2013
A fast compiler for NetKAT. Smolka et al., 2015

(II) Run-Time: Composition

(DFA-Transitioning >> Forwarding) + DFA-Accepting

```
state=Q0 & switch=S1 & srcip=A → state←Q1
state=Q1 & switch=S4 & dstip=A → state←Q2
```

\>>

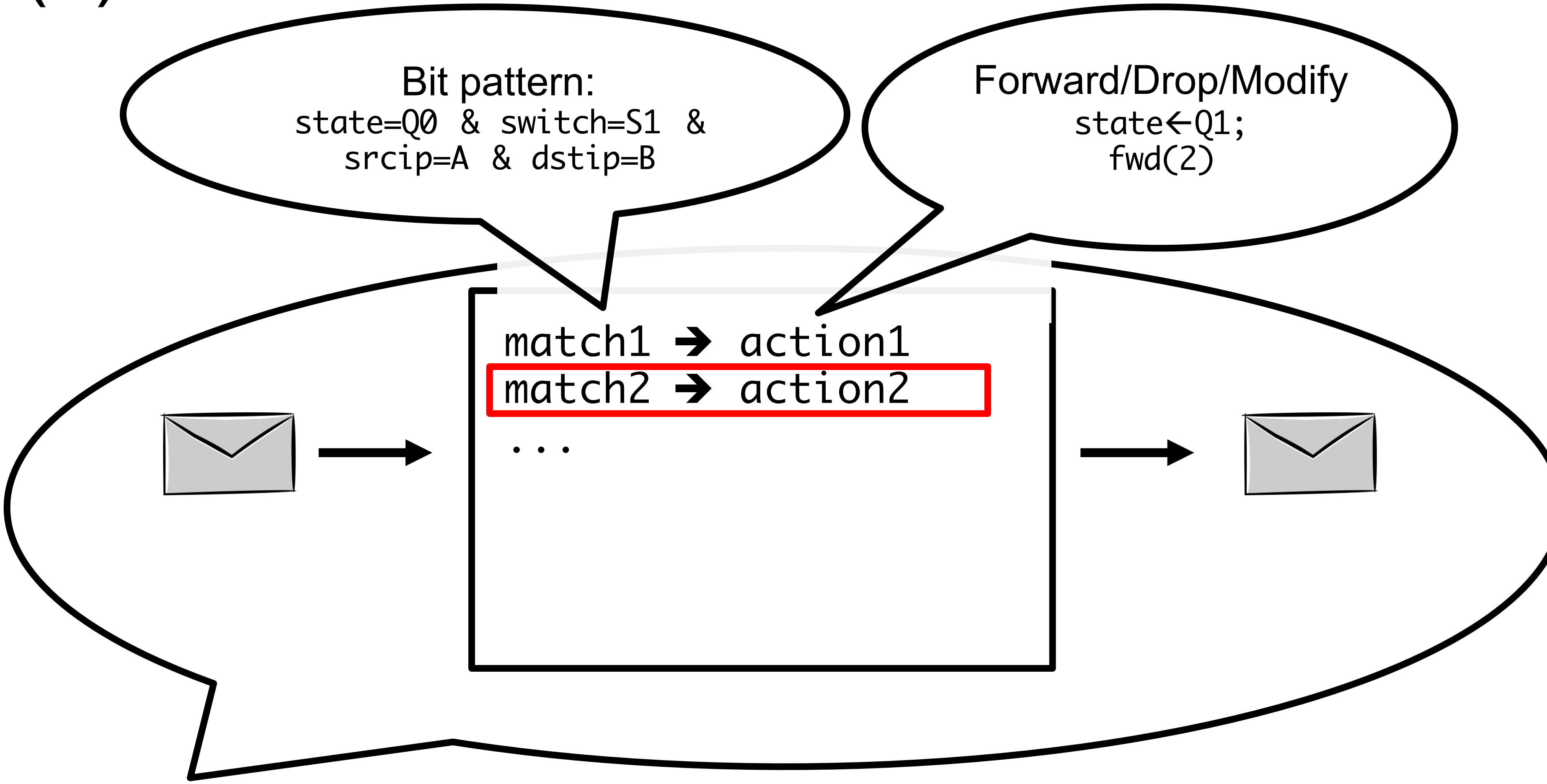
```
dstip=A → fwd(1)
dstip=B → fwd(2)
dstip=C → fwd(3)
...
```

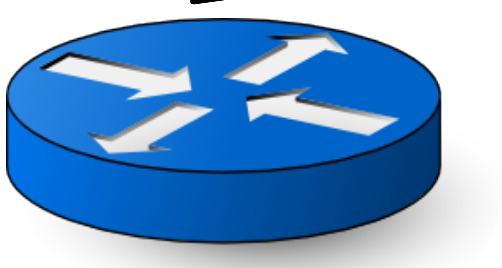
```
state=Q0 & switch=S1 & srcip=A & dstip=B
→ state←Q1, fwd(2)
```

Openflow 1.0 (for example)

Composing software-defined networks. Monsanto et al., 2013
A fast compiler for NetKAT. Smolka et al., 2015

(II) Run-Time: Generate Switch Rules





Result: unified switch rules for forwarding *and* measurement

(II) Run-Time: Other details in paper…

- Handle `groupby` aggregation
- Testing predicates before and after forwarding
- Upstream query compilation

(II) Run-Time: Related Work

Approach	Expressiveness	Sources of inaccuracy	Sources of overhead
Policy checking (§1.5.1)			
Header space analysis [52, 53]	Locations and headers	No actual packets Only control plane view	Policy analysis
Out-of-band approaches (§1.5.2)			
Infer using traffic matrix [32, 119]	Switch-level paths	Forwarding dynamism Downstream packet drop Opaque multipath routing	Load collection [21] Traffic collection [1, 14, 87]
Upstream inference [53, 121]	Locations and headers	Ambiguous upstream path Packet modification	Traffic collection [1, 14, 87] Policy analysis
Join per-hop info [27, 40, 96, 122]	Locations and headers	Ambiguous packet joins	Packet digests (every hop) Topological sort
In-band approaches (§1.5.3)			
Record interfaces [83, 90]	Interface-level paths	Record few interfaces	Packet space for interfaces
Path tracing [102, 118]	Interface-level paths	Strong assumptions	Packet space for interfaces Data plane rules
Our approach (§1.6)			
DFA on packet state [65, 66]	Locations and headers	*None*	Packet space for DFA state Data plane rules Query compile time

How well does it work?

Evaluation of initial prototype

- Prototype on Pyretic + NetKAT + OpenVSwitch
  - Publicly available: http://frenetic-lang.org/pyretic/
- Queries: traffic matrix, DDoS detection, per-hop packet loss, firewall evasion, slice isolation, congested link
- Run *all queries together* on Stanford backbone
  - Compile time: > 2 hours
  - Switch rules: (estimated per switch) 1M
  - Packet state: 10 bits

Problem: Cross-Products

(DFA-Transitioning >> Forwarding) + DFA-Accepting

```
state=Q0 & switch=S1 & srcip=A ➔ state←Q1
state=Q1 & switch=S4 & dstip=A ➔ state←Q2
```

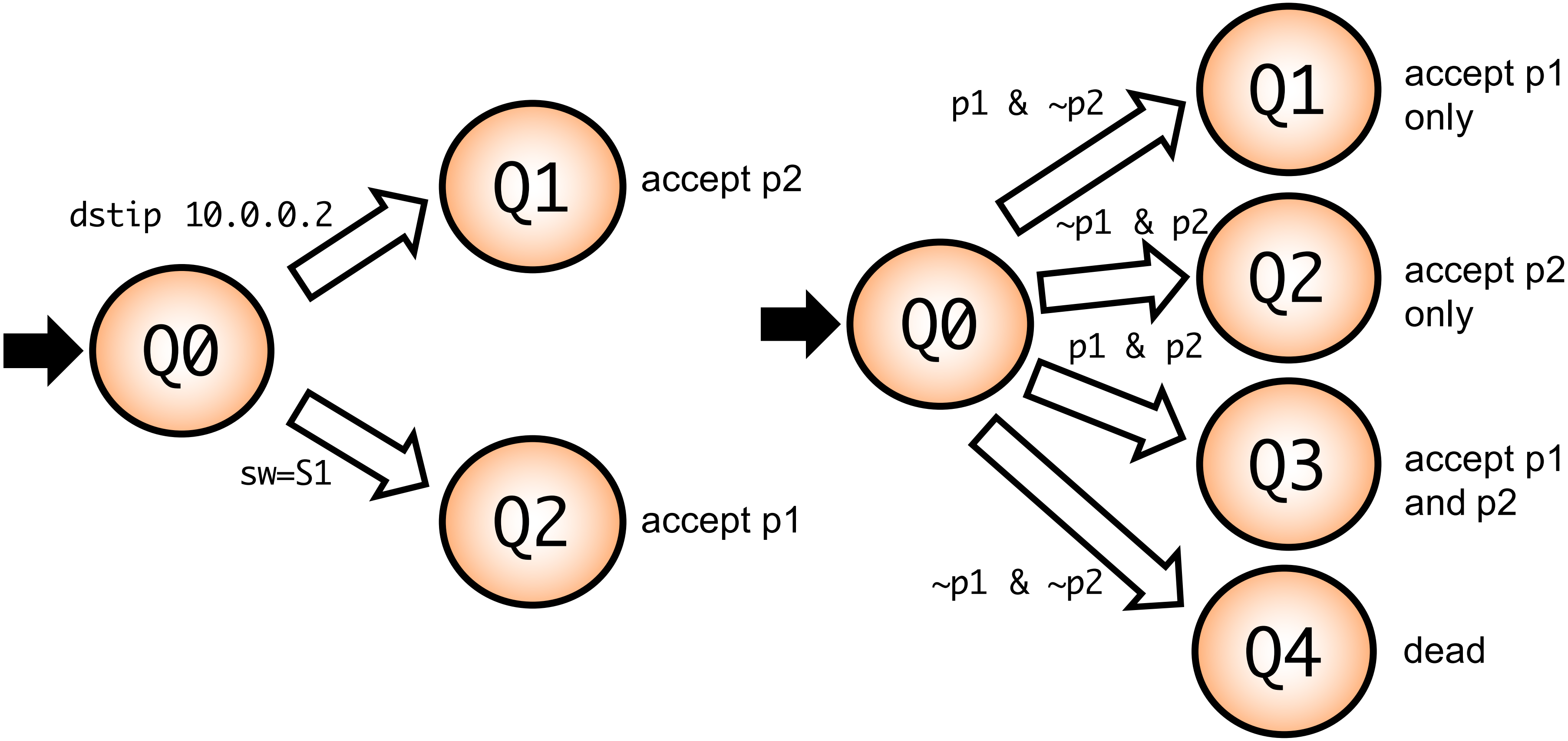
>>

```
dstip=A ➔ fwd(1)
dstip=B ➔ fwd(2)
dstip=C ➔ fwd(3)
...
```

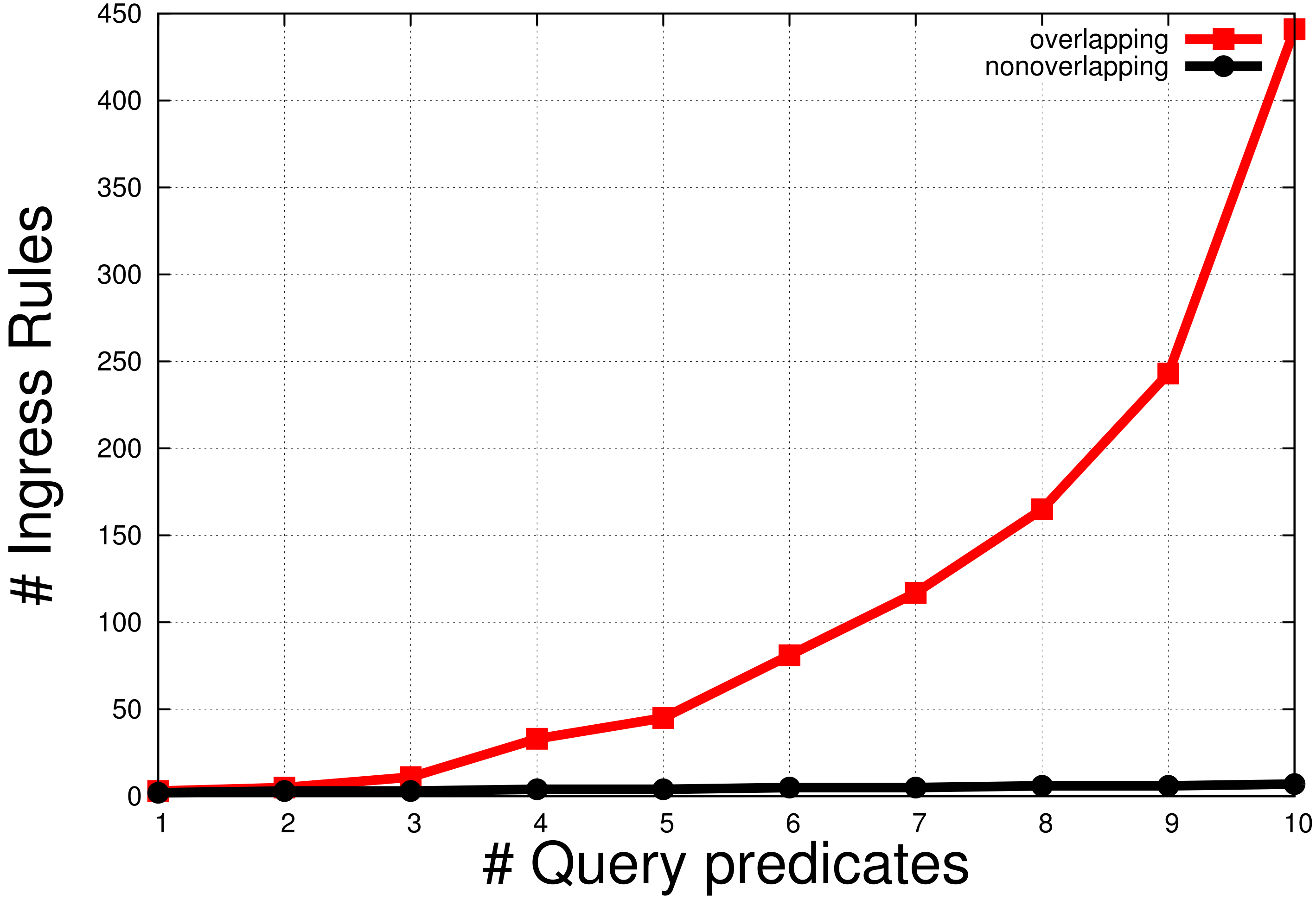
```
state=Q0 & switch=S1 & srcip=A & dstip=B
➔ state←Q1, fwd(2)
```

Problem: Cross-Products

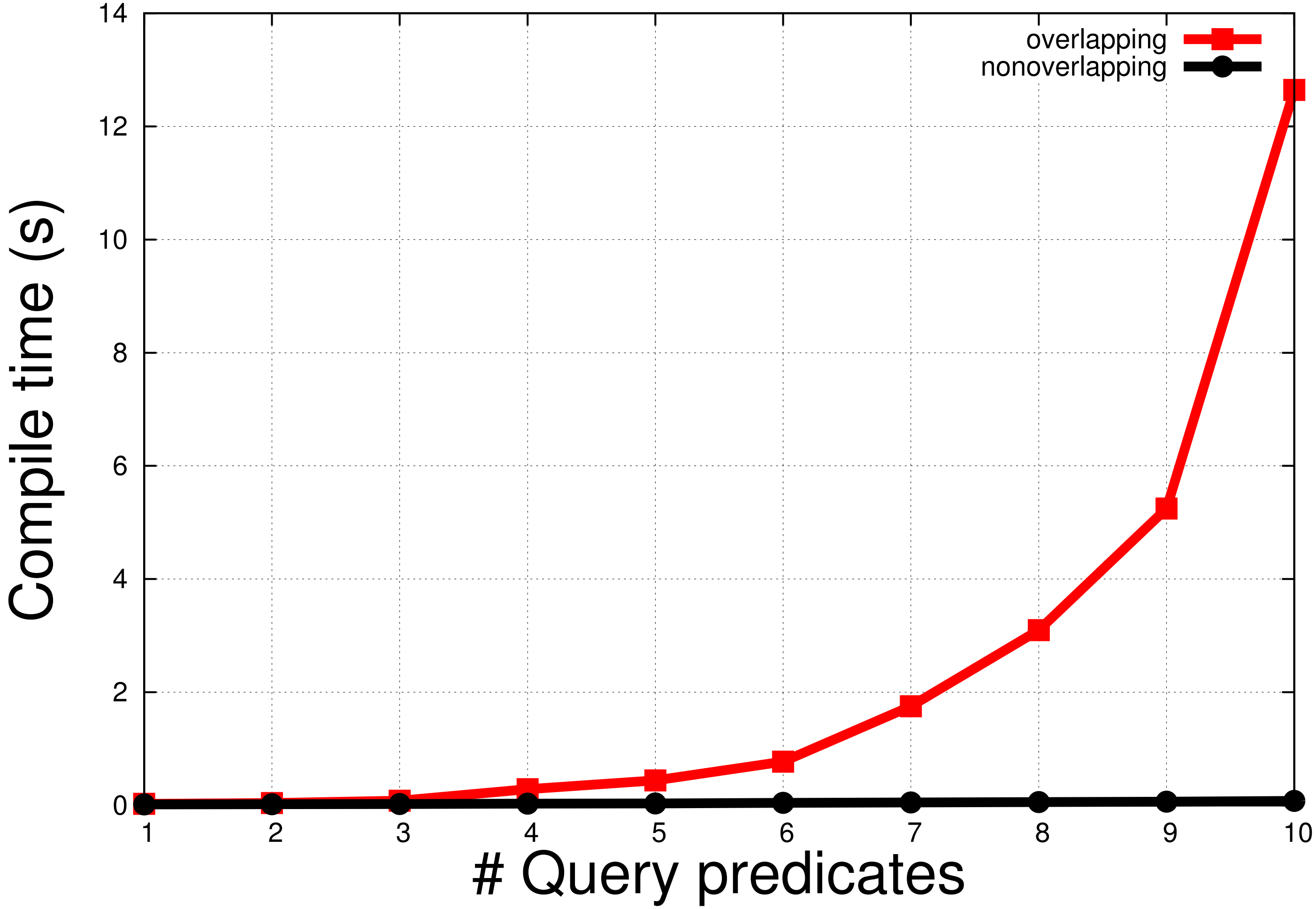
- p1: `sw=S1`
- p2: `dstip=10.0.0.2`



Complexity From Overlaps



Complexity From Overlaps



(III) Optimizations: Reduce Pkt Overlap

- *Construct* non-overlapping policies
  - Use structure of generated Pyretic policies

- *Remove* overlapping actions on packets
  - Use pipelined packet processing

- *Speed up* detection of overlapping actions
  - Use better data structures & caching

(III) Optimizations: Summary

Optimization	# Rules?	Time?	# States?
Separate query & forwarding actions into separate stages	↓	↓	
Optimize conditional policy compilation	↓	↓	
Integrate tagging and capture policies		↓	
Pre-partition predicates by flow space	↓	↓	
Cache predicate overlap decisions		↓	
Decompose query predicates into multiple stages	↓	↓	↑
Detect predicate overlaps with Forwarding Decision Diagrams		↓	

(III) Optimizations: Summary

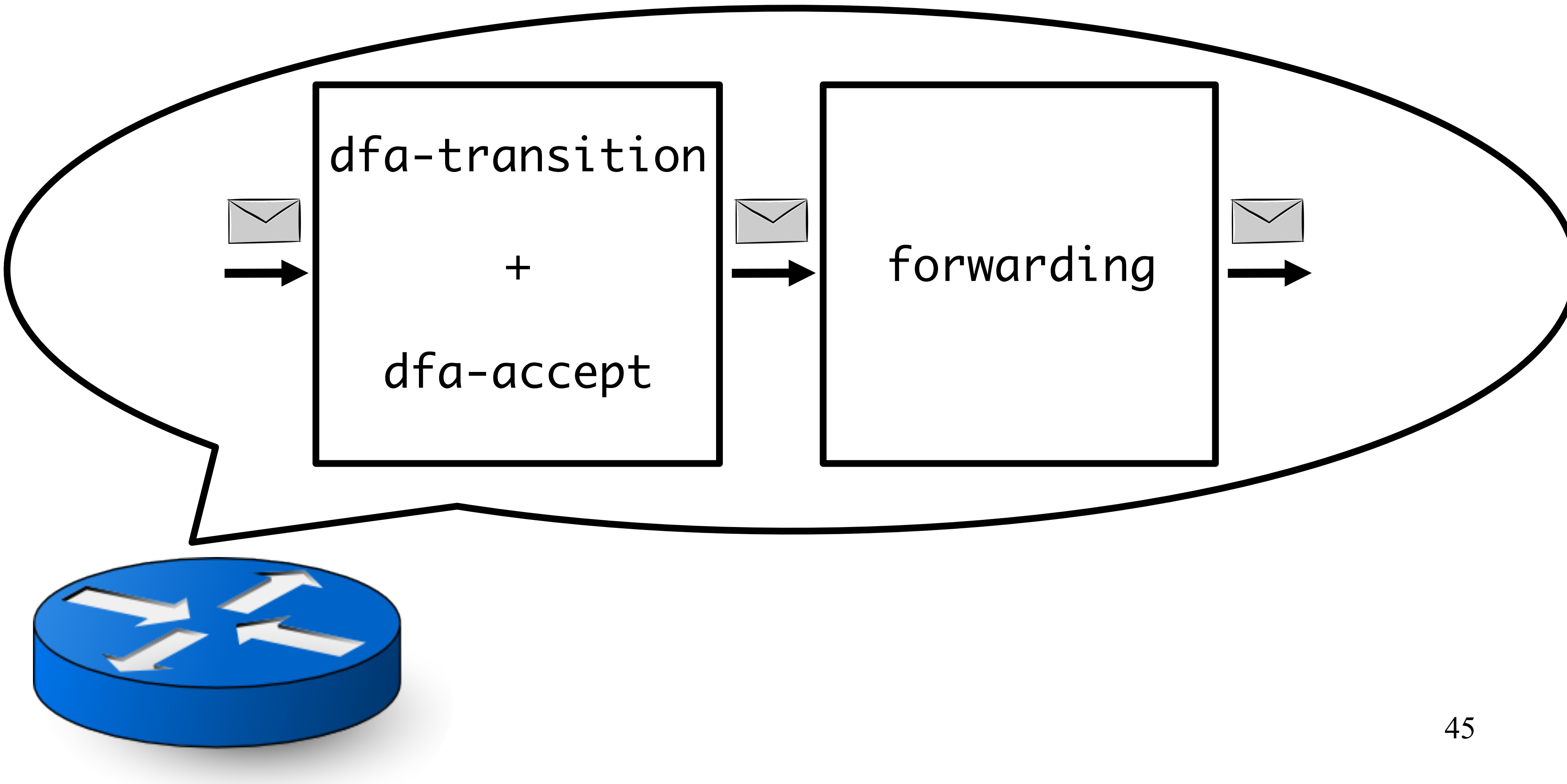
Optimization	# Rules?	Time?	# States?
Separate query & forwarding actions into separate stages	↓	↓	
Optimize conditional policy compilation	↓	↓	
Integrate tagging and capture policies		↓	
Pre-partition predicates by flow space	↓	↓	
Cache predicate overlap decisions		↓	
Decompose query predicates into multiple stages	↓	↓	↑
Detect predicate overlaps with Forwarding Decision Diagrams		↓	

(III) Separate Queries from Forwarding

(DFA-Transition >> Forwarding) + DFA-Accept

==

(DFA-Transition + DFA-Accept) >> Forwarding



(III) Separate Queries from Forwarding

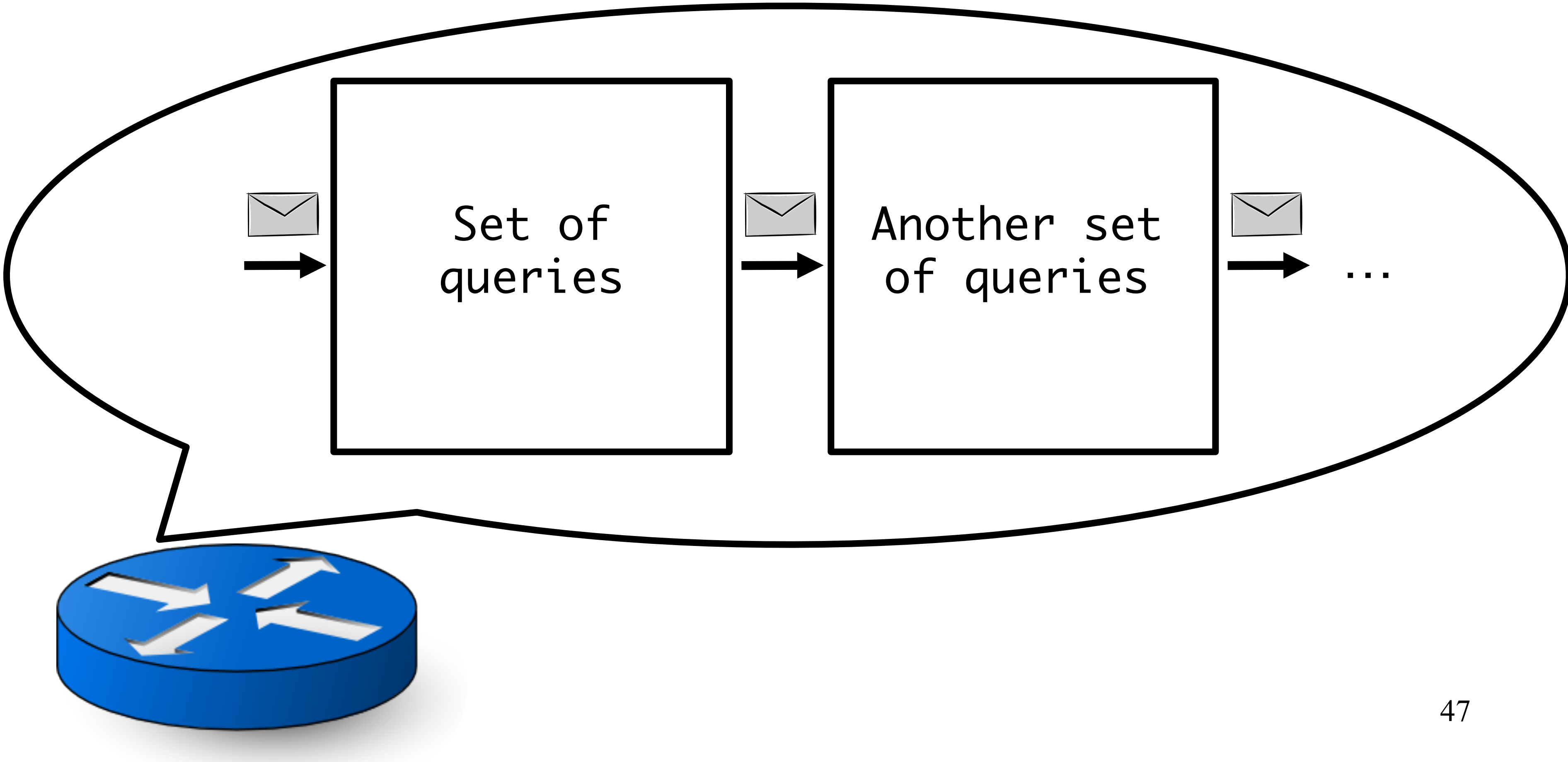
(DFA-Ingress-Transitioning >> Forwarding >> DFA-Egress-Transitioning)
+
(DFA-Ingress-Accepting)
+
(DFA-Ingress-Transitioning >> Forwarding >> DFA-Egress-Accepting)

==

(DFA-Ingress-Transitioning + DFA-Ingress-Accepting)
>>
Forwarding
>>
(DFA-Egress-Transitioning + DFA-Egress-Accepting)

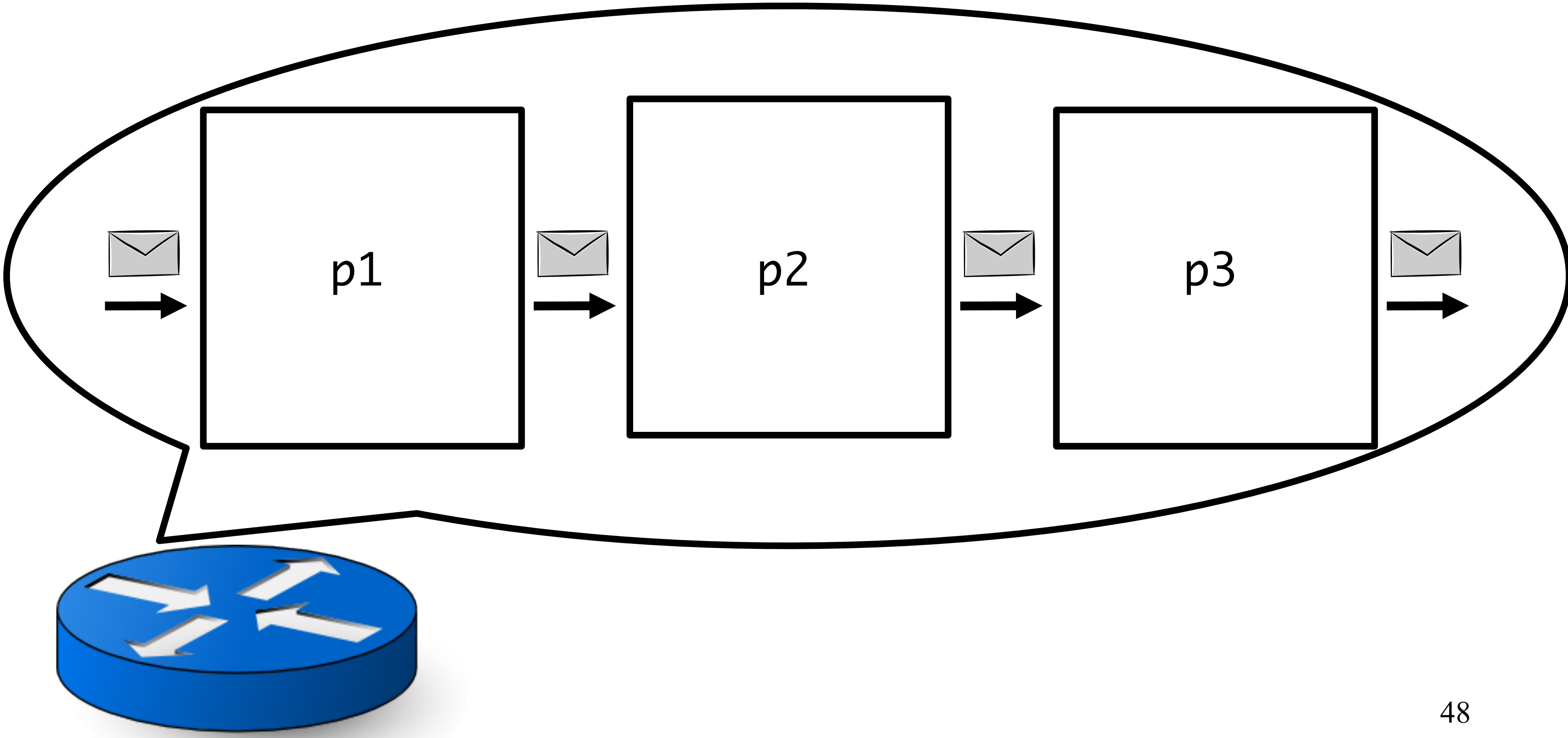
(III) Separating Queries

- Could we run queries in a pipelined fashion?



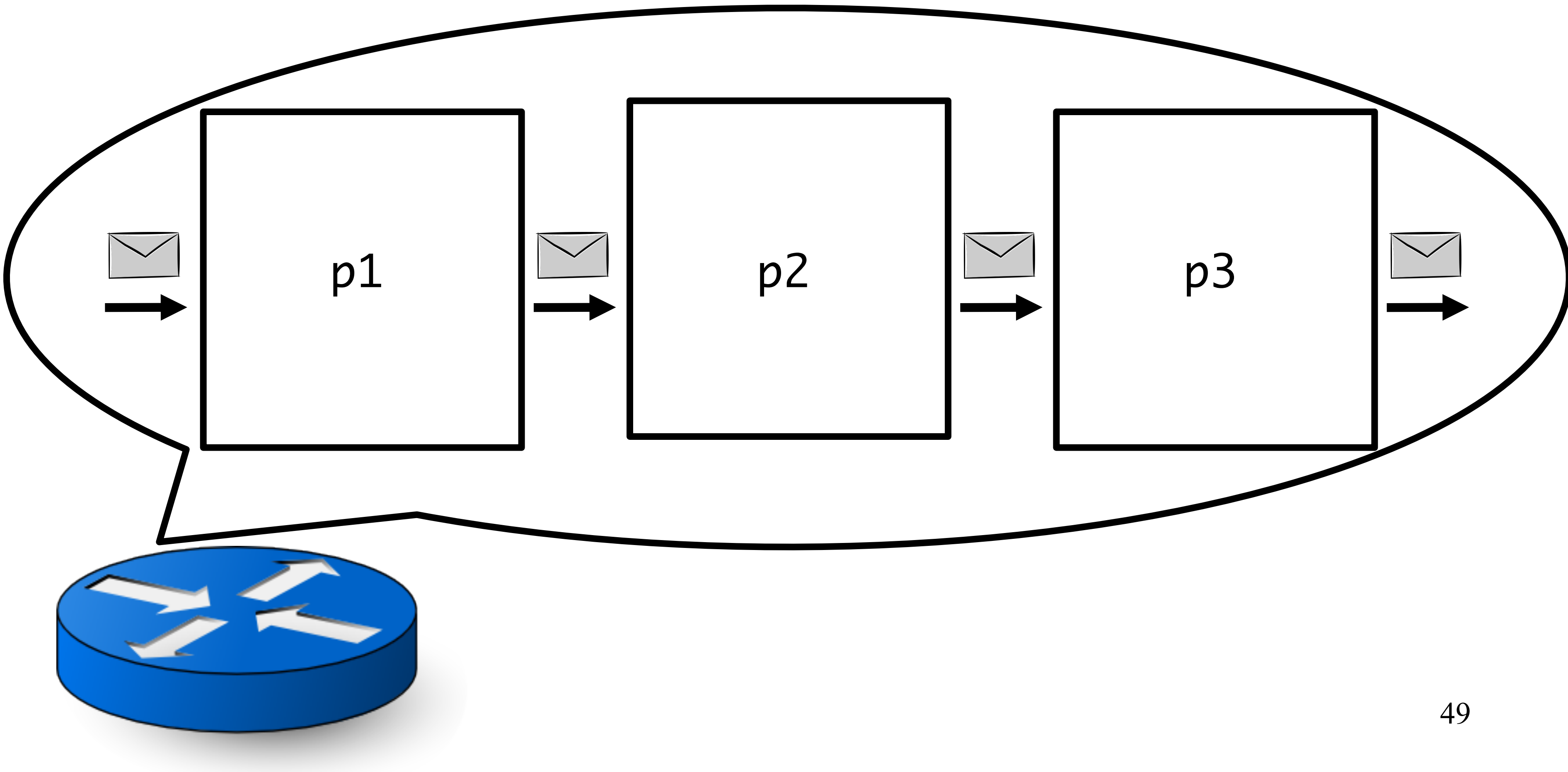
(III) Separating Queries

- p1: `sw=S1`; p2: `dstip=10.0.0.2;` p3: `dstip=10.0.0.3`



(III) Separating Queries

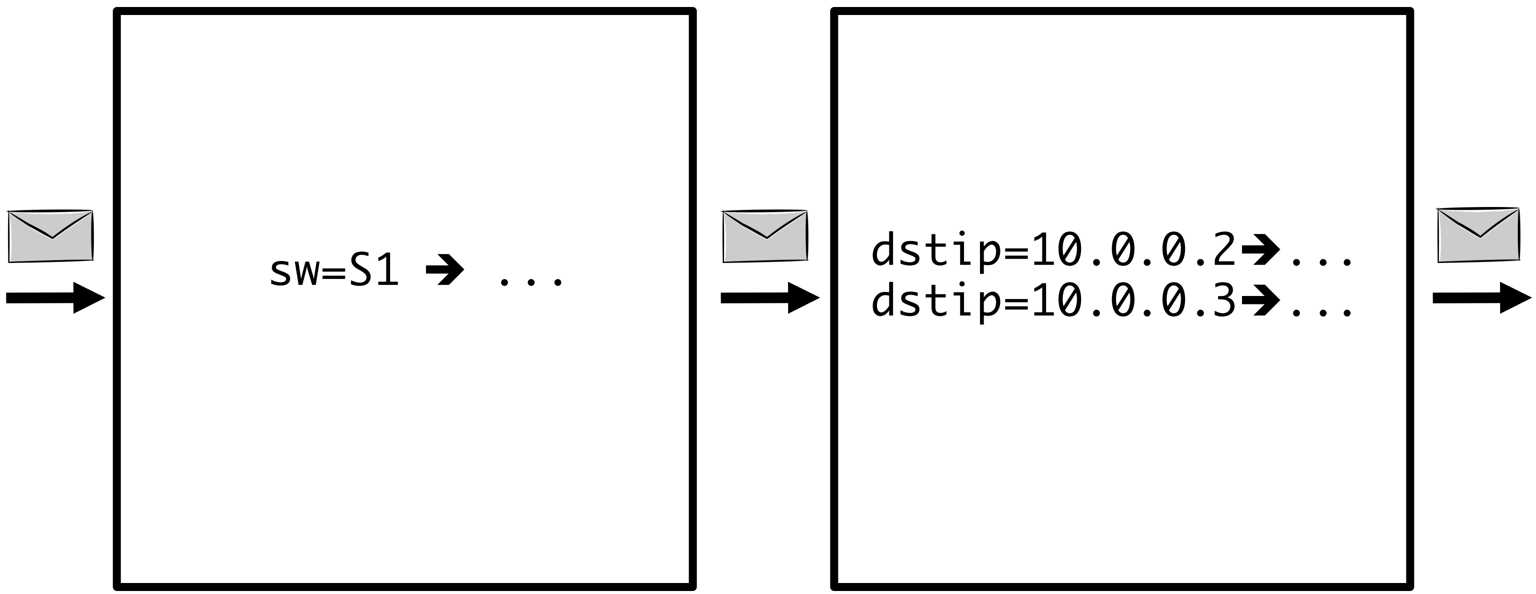
- p1: `sw=S1`; p2: `dstip=10.0.0.2;` p3: `dstip=10.0.0.3`
- Problem: Limited # table stages & rules per stage



(III) Separating Queries

- p1: `sw=S1`; p2: `dstip=10.0.0.2`; p3: `dstip=10.0.0.3`

- Idea: Group queries by their "similarity"
  - p1 in one stage, p2 and p3 in another



(III) Cost Function for Query Similarity

- Input: a set of queries
- Output: estimate # rules if queries in *same* table stage

```
cost ((type1, count1), (type2, count2)) :=
  case type1 == φ:
    count2 + 1
  case type1 == type2:
    count1 + count2
  case type1 ⊂ type2:
    count1 + count2
  case type1 ∩ type2 == φ:
    (count1 + 1) * (count2 + 1) - 1
  case default:
    (count1 + 1) * (count2 + 1) - 1
```

Predicate-similarity-aware rule space estimation

Concurrent NetCore: From policies to pipelines. Schlesinger et al., 2014
Compiling packet programs to reconfigurable swithces. Jose et al., 2015

(III) Cost-Aware Query Grouping

- Minimize total # stages $S = \sum_j y_j$
- Subject to:
  - Rule space per stage $\text{cost}(\{q_{ij} : q_{ij} = 1\}) \leq \text{rulelimit} * y_j$
  - Total number of stages $S \leq \text{stagelimit}$
  - One query ➔ one stage $\forall i : \sum_j q_{ij} = 1$
- Variables (binary integers)
  - Stage j assigned
  - Query i assigned to j

$$q_{ij} \in \{0,1\}, y_j \in \{0,1\}$$

Evaluation

- Prototype on Pyretic + NetKAT + OpenVSwitch
  - Publicly available: http://frenetic-lang.org/pyretic/
- Queries: traffic matrix, DDoS detection, per-hop packet loss, firewall evasion, slice isolation, congested link

- Run *all queries together* on Stanford backbone
  - Compile time: > 2 hours ➔ 5 seconds
  - Switch rules: (estimated) 1M ➔ (actual) ~1K
  - Packet state: 10 bits ➔ 16 bits

Benefit of Optimizations (Stanford)

Cumulative Optimization	Time (s)	# Rules	# State Bits
None	> 7900	DNF	DNF
Separate query & forwarding actions into separate stages	> 4920	DNF	DNF
Optimize conditional policy compilation	> 4080	DNF	DNF
Integrate tagging and capture policies	2991	2596	10
Pre-partition predicates by flow space	56.19	1846	10
Cache predicate overlap decisions	35.13	1846	10
Decompose query predicates into multiple stages	5.467	260	16

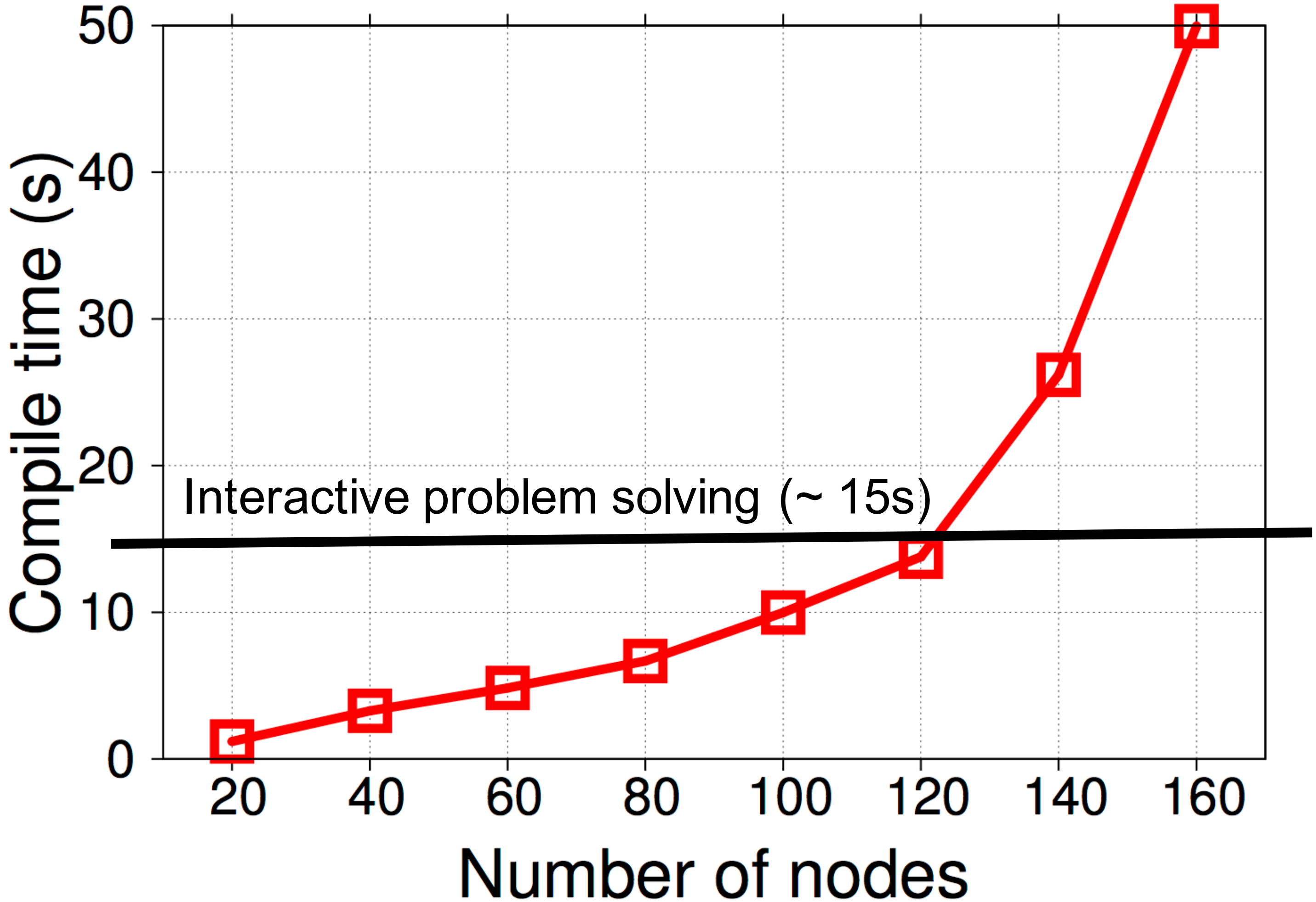
Scalability Trends

- Five synthetic ISP (Waxman) topologies at various network sizes

- At each network size, run mix of queries from before

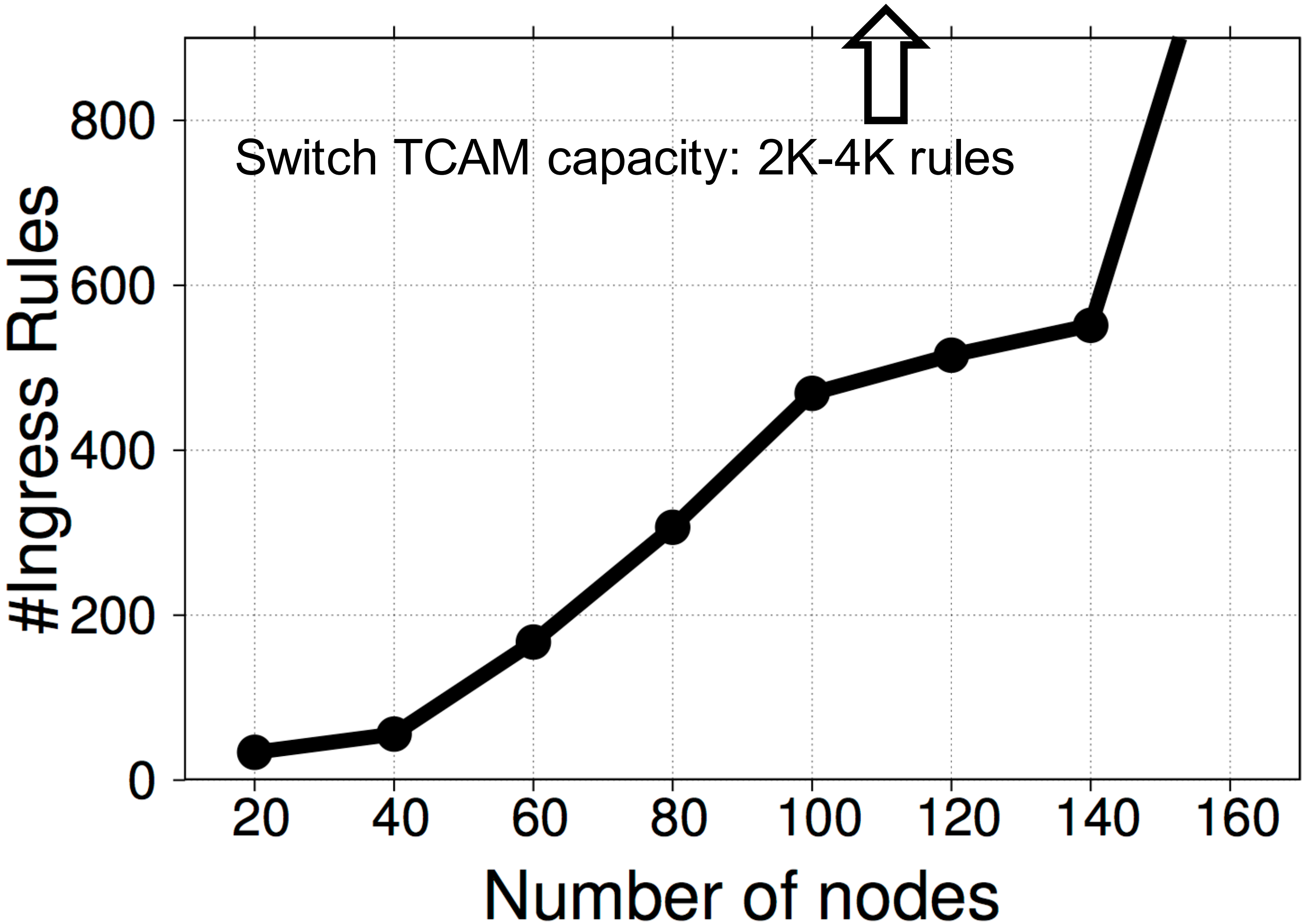
    - Averaged metrics across queries & topologies

Evaluation: Scaling

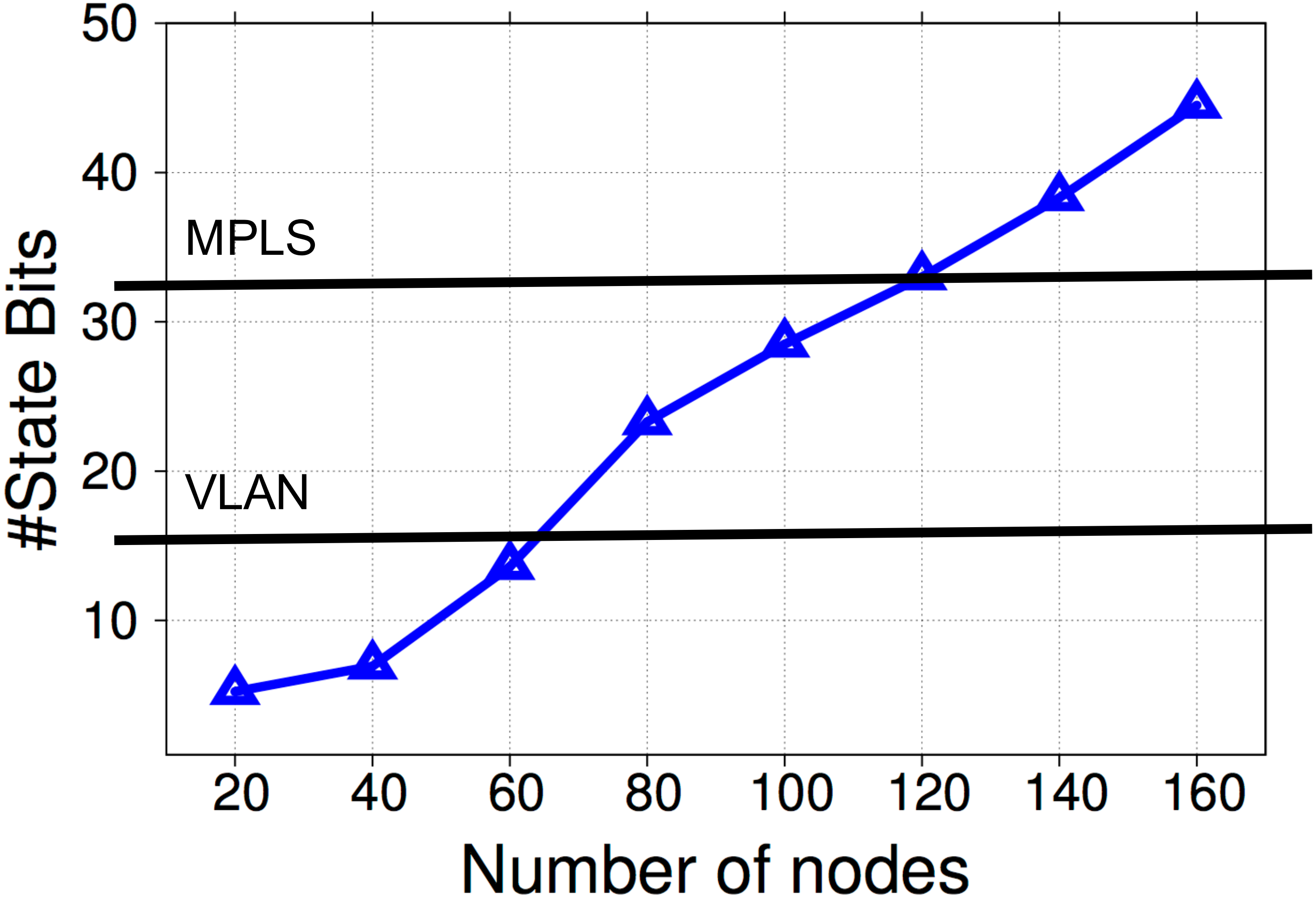


Miller, “Response time in man-computer conversational transactions”

II. Rule Count



III. Packet State Bits



Conclusions

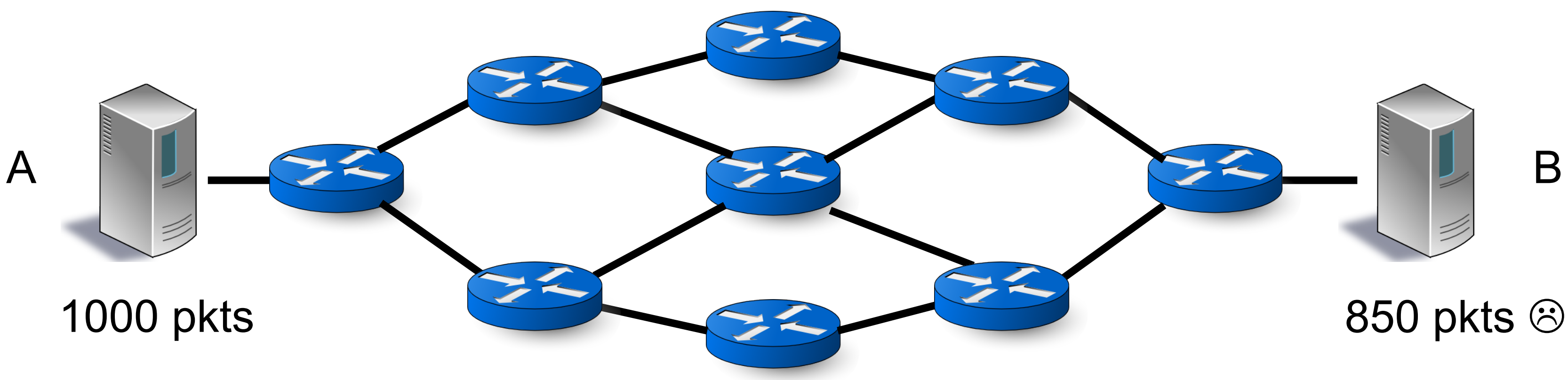
- We need good abstractions to measure networks
  - Abstractions must be efficiently implementable
- Query-driven measurement: a useful principle
  - Improves accuracy; *and*
  - Reduces overheads
- Challenge: finding sufficiently general *families* of questions with efficient solution techniques
- Path queries can simplify network management!

Thanks! ☺






Demo: Where's the Packet Loss?



Demo: Where's the Packet Loss?

https://youtu.be/VxOaN9iGPWc

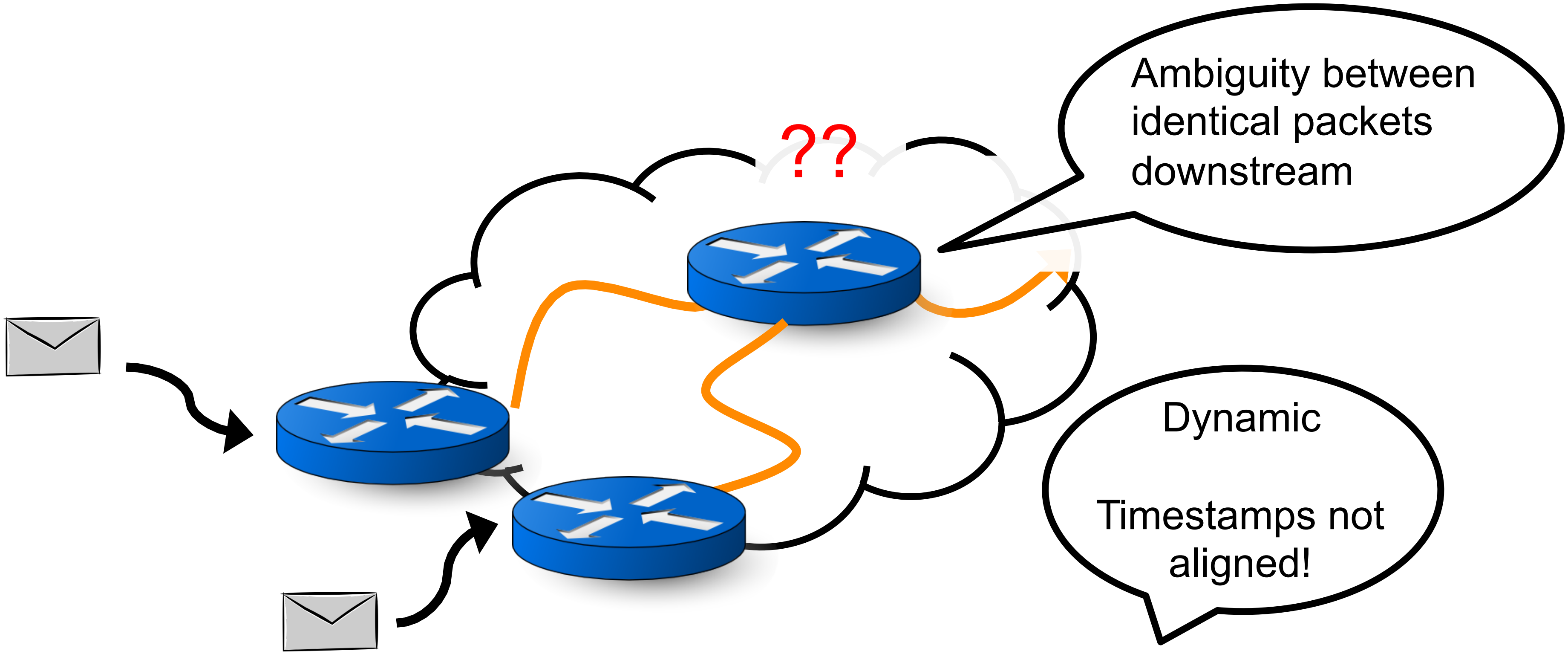
Discussion: Questions

- Control plane versus data plane checking
- Switch performance impact (throughput, delay…)
  - Table stages
  - Memory on the switch
  - Memory on the packet
- Comparison to existing SDN approaches
- System evaluation

Discussion: Extensions

- Multi-packet queries?
  - Performance, security, …
  - What language abstractions? What hardware?
- Post-facto queries
- Improving compiler performance

Approach 1: Join Traffic & Forwarding



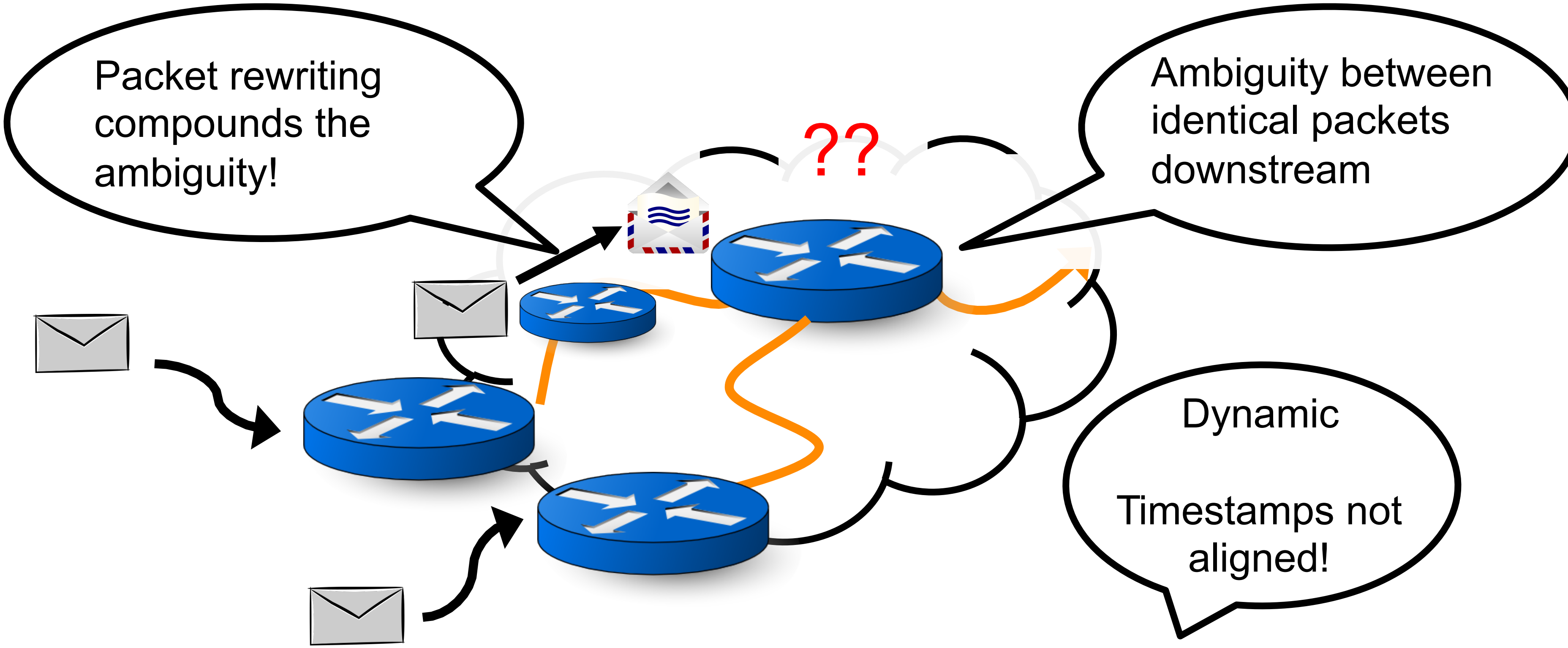
Traffic dataset X Forwarding updates

e.g., NetFlow, SNMP

e.g., OF/routing protocol updates

Packet traceback for software-defined networks. Zhang et al., 2015

Approach 1: Join Traffic & Forwarding



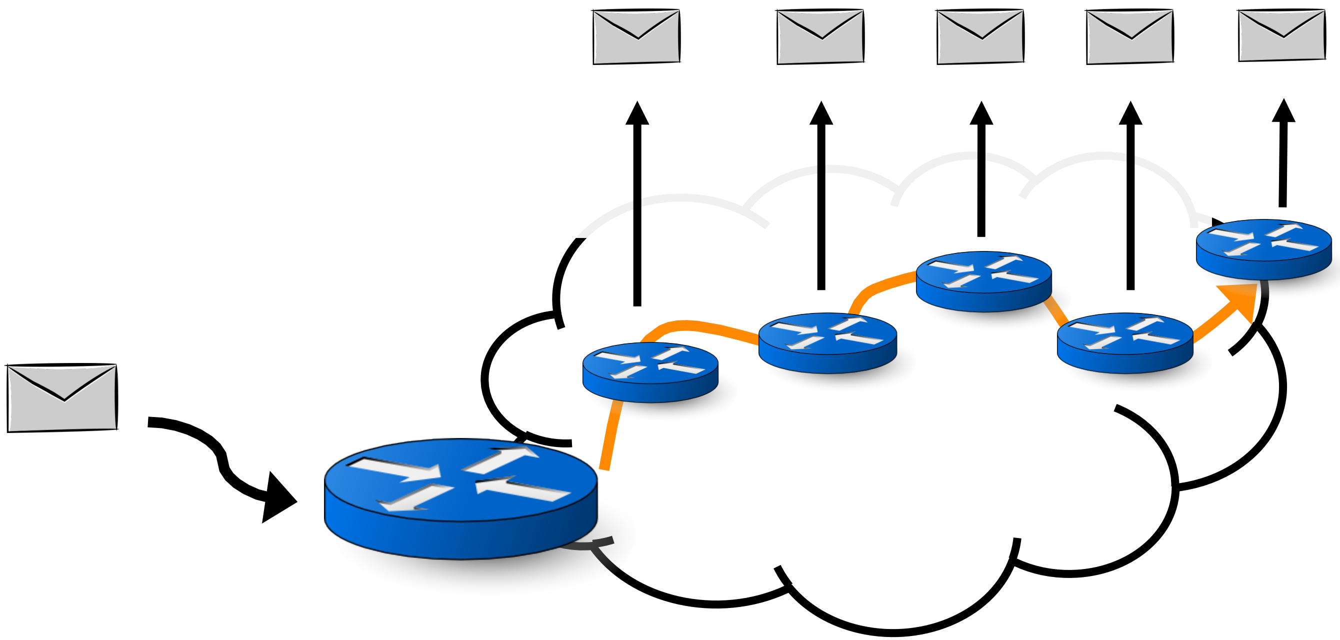
Traffic dataset X Forwarding updates

e.g., NetFlow, SNMP

e.g., OF/routing protocol updates

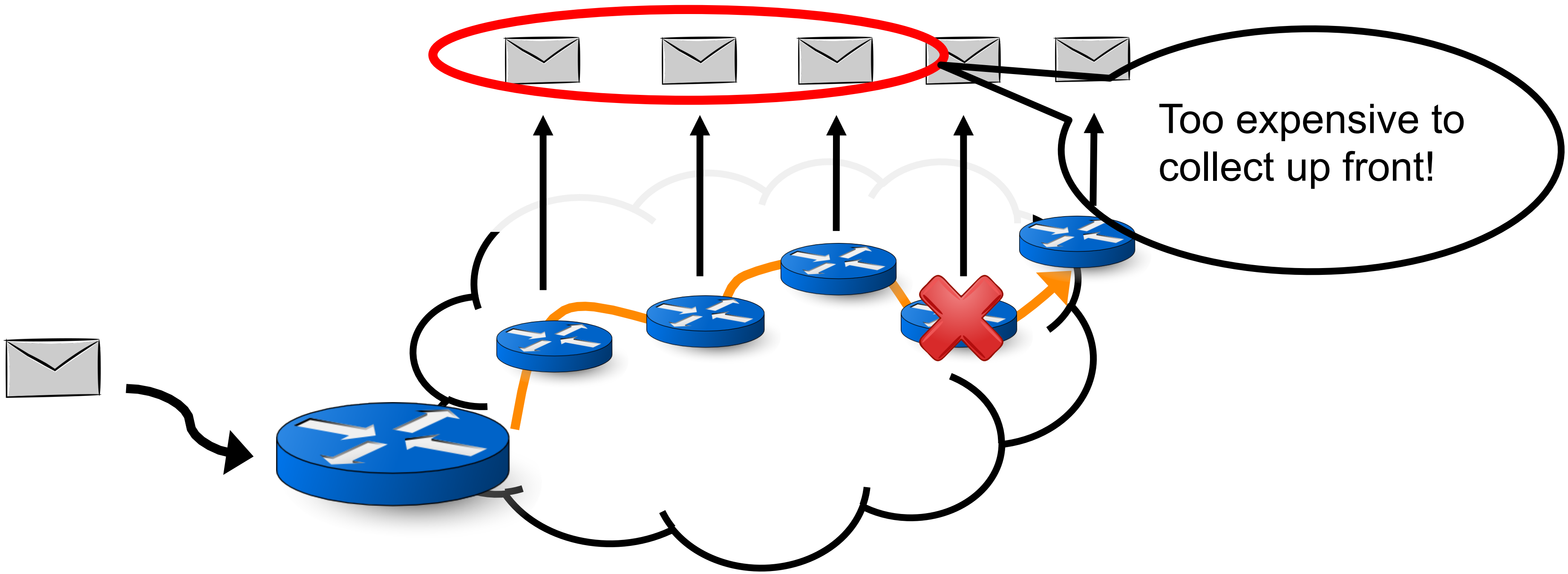
Trajectory sampling for direct traffic observation. Duffield et al., 2001

Approach 2: Collect at Every Hop



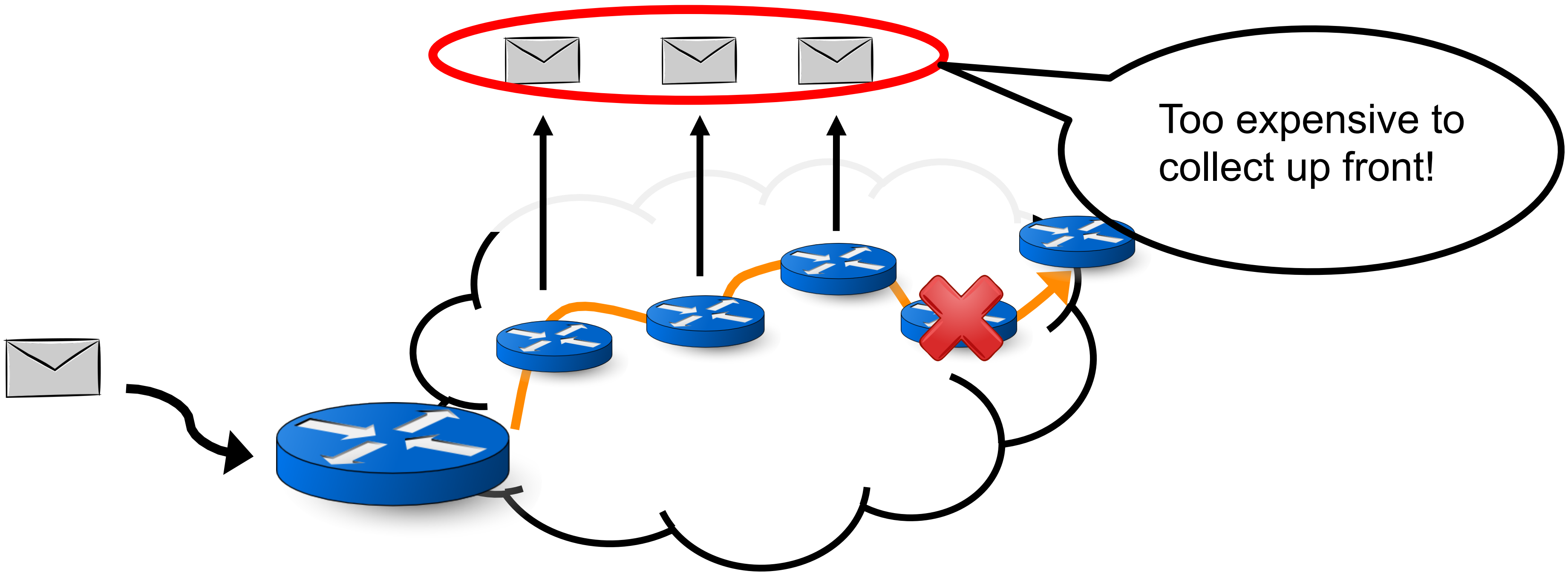
Using packet histories to troubleshoot networks. Handigol et al., 2014
Hash-based IP traceback. Snoeren et al., 2001
Packet-level telemetry in large data-center networks. Zhu et al., 2015

Approach 2: Collect at Every Hop



Using packet histories to troubleshoot networks. Handigol et al., 2014
Hash-based IP traceback. Snoeren et al., 2001
Packet-level telemetry in large data-center networks. Zhu et al., 2015

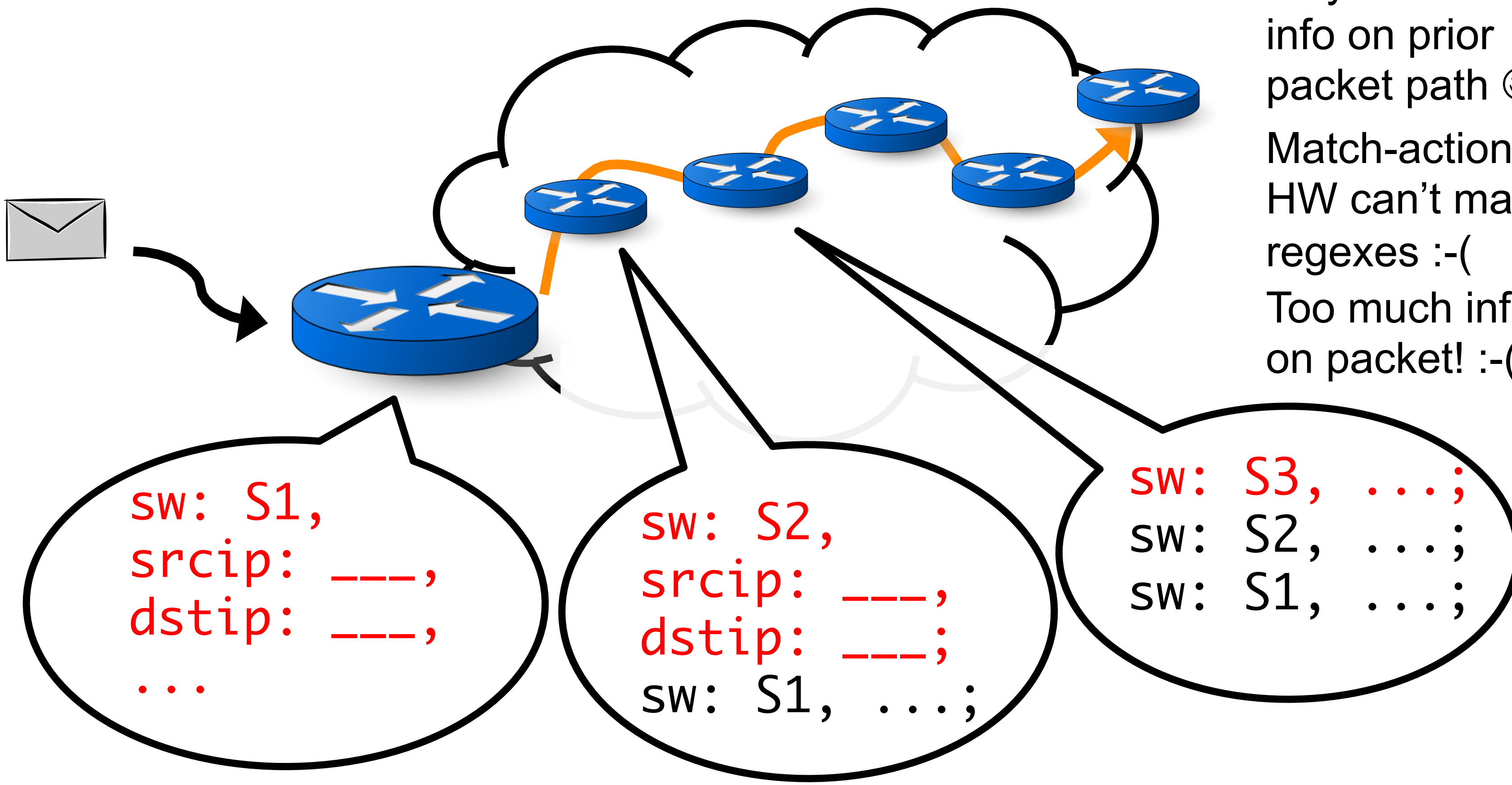
Approach 2: Collect at Every Hop



Sampling to reduce overhead may miss the packets you care about…

Trajectory sampling for direct traffic observation. Duffield et al., 2001

Approach 3: Write Path into Packet



Switches have very accurate info on prior packet path ☺

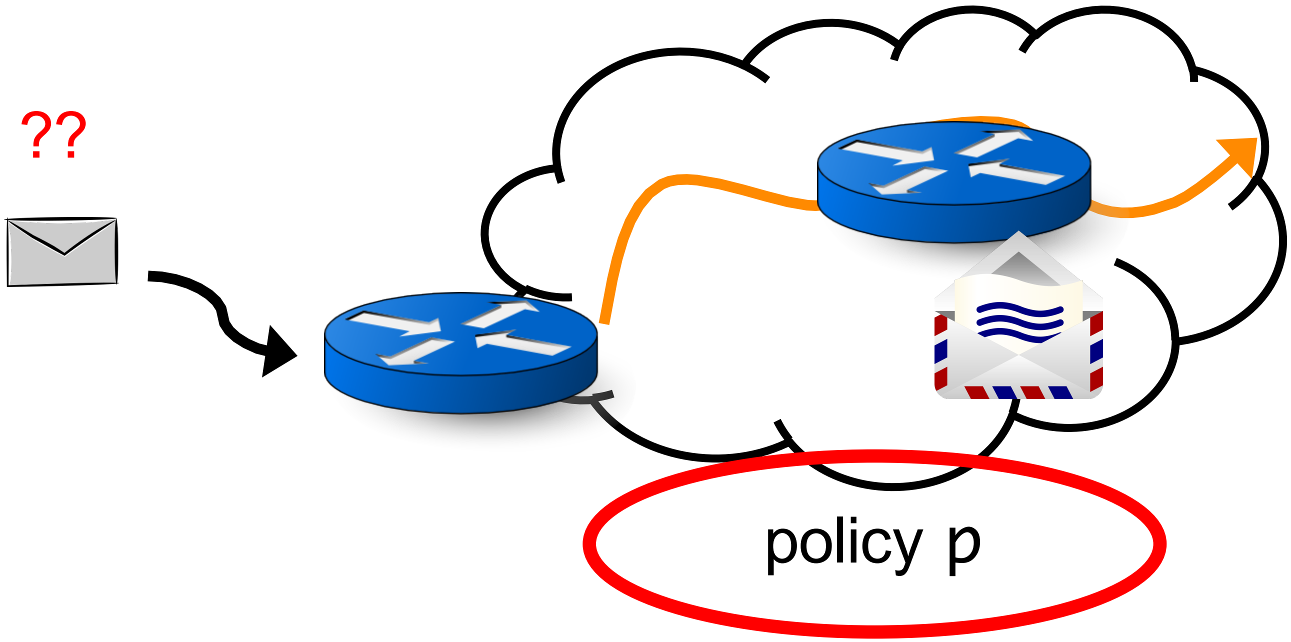
Match-action HW can't match regexes :-(

Too much info on packet! :-(

IP record route, RFC 791. Postel, 1981

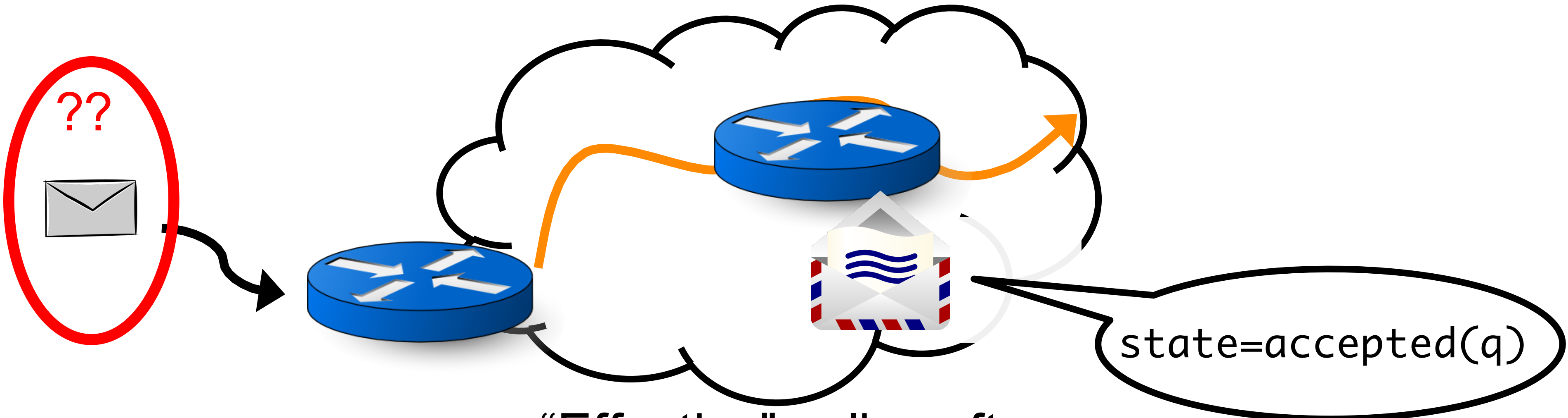
Tracing packet trajectory in data-center networks. Tammana et al., 2015

Reachability Testing for Accepted Pkts



Static checking for networks, Kazemian et al. NSDI ‘12

Reachability Testing for Accepted Pkts



"Effective" policy after downstream compilation of query *q*

Static checking for networks, Kazemian et al. NSDI '12

Complexity from Overlaps