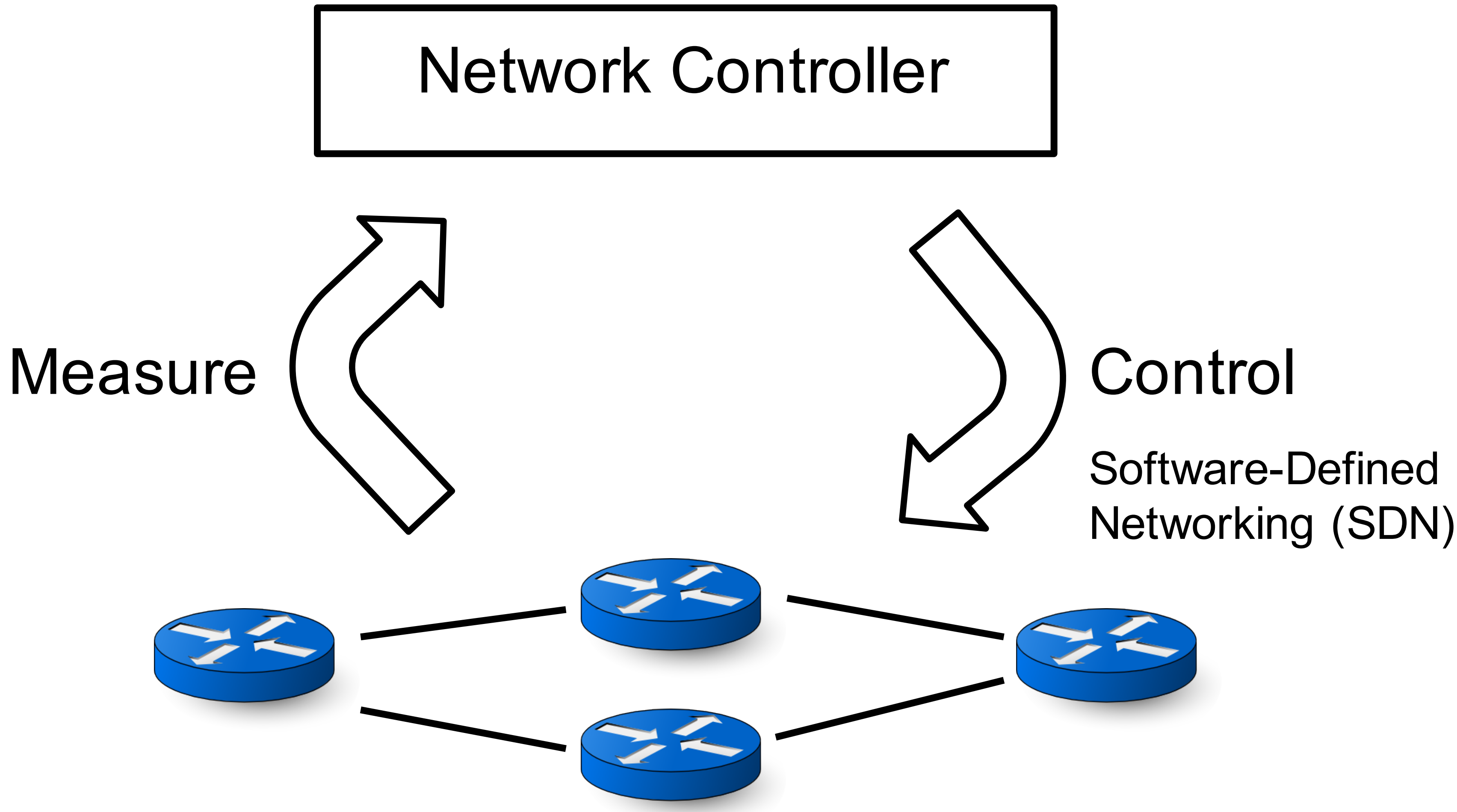
Compiling Path Queries

Princeton University

Srinivas Narayana Mina Tahmasbi Jen Rexford David Walker

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

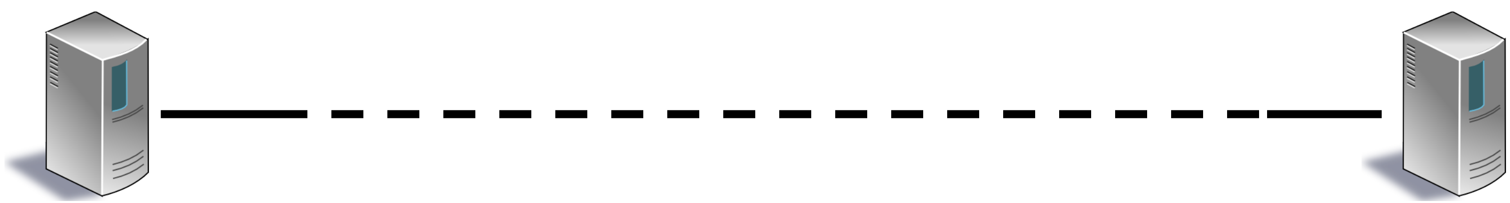
- Difficult to know *where* things go wrong!
  - Humans are slow in troubleshooting
  - Human time is expensive

- Can we build *programmatic tools* to help?

Example: Where's the Packet Loss?

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

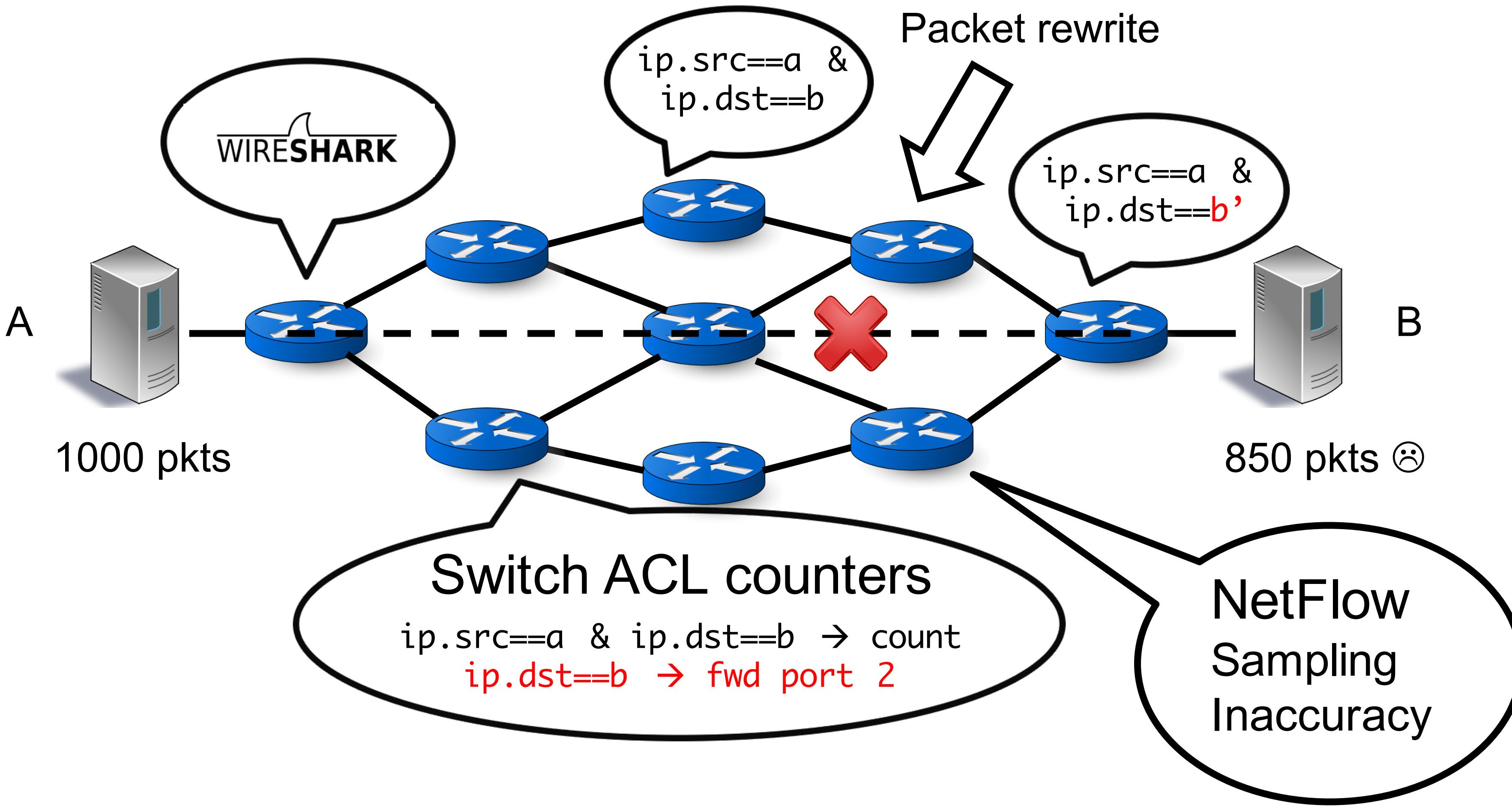
A B



1000 pkts 850 pkts ☹

Example: Where's the Packet Loss?

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



Example: Where's the Packet Loss?

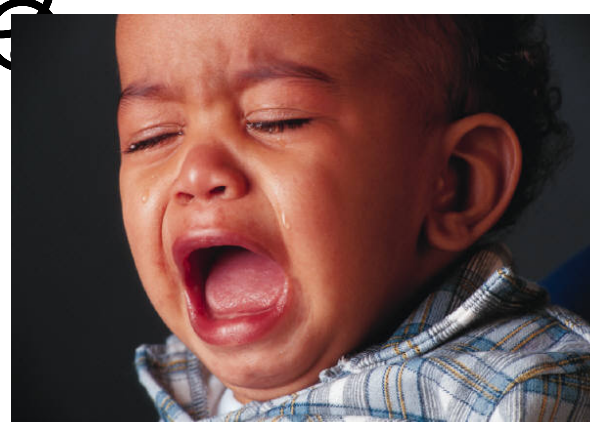
Complex & Inaccurate Join with multiple datasets: traffic, forwarding, topology

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

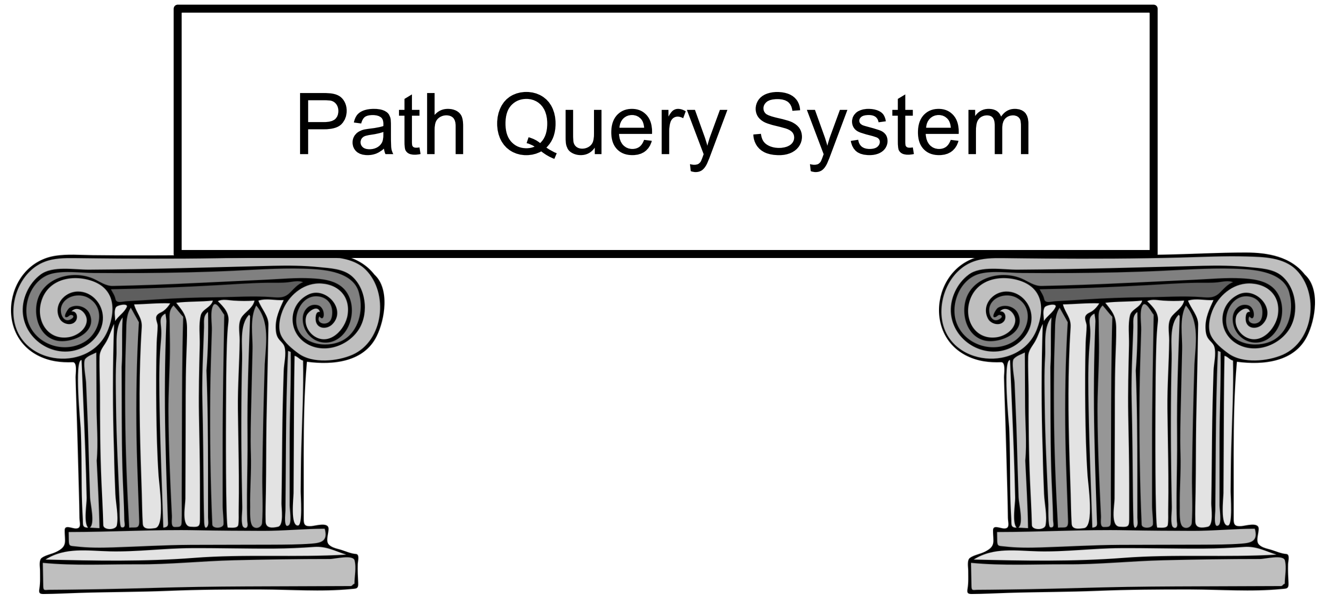


Pattern: Combining Traffic & Forwarding

- 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

Our Approach



Declarative Query Specification

Independent of Forwarding
Independent of Other Measurements
Independent of Hardware Details

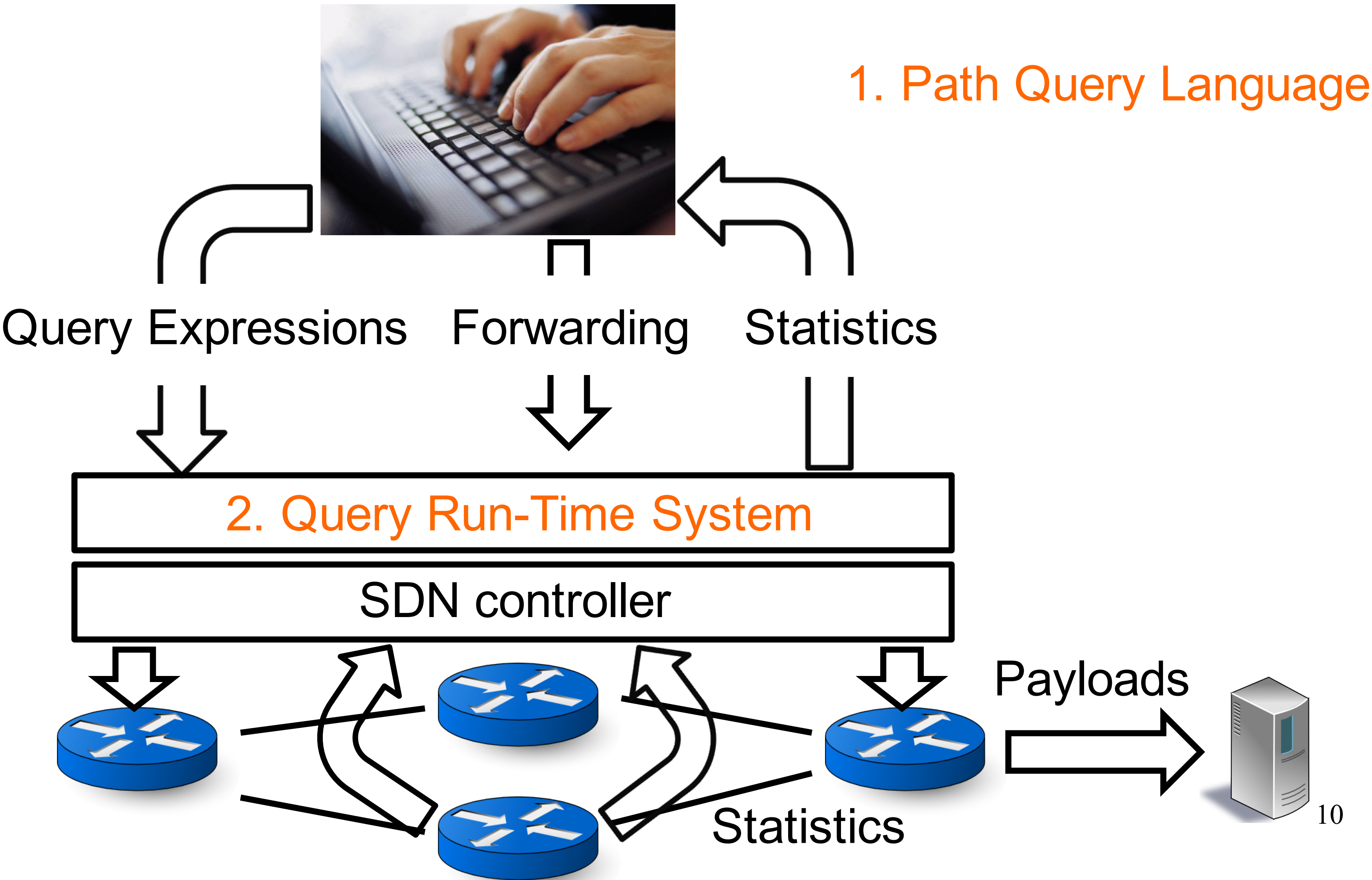
Path Query Language

Query-Driven Measurement

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

Query Run-Time System

Our Approach



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?

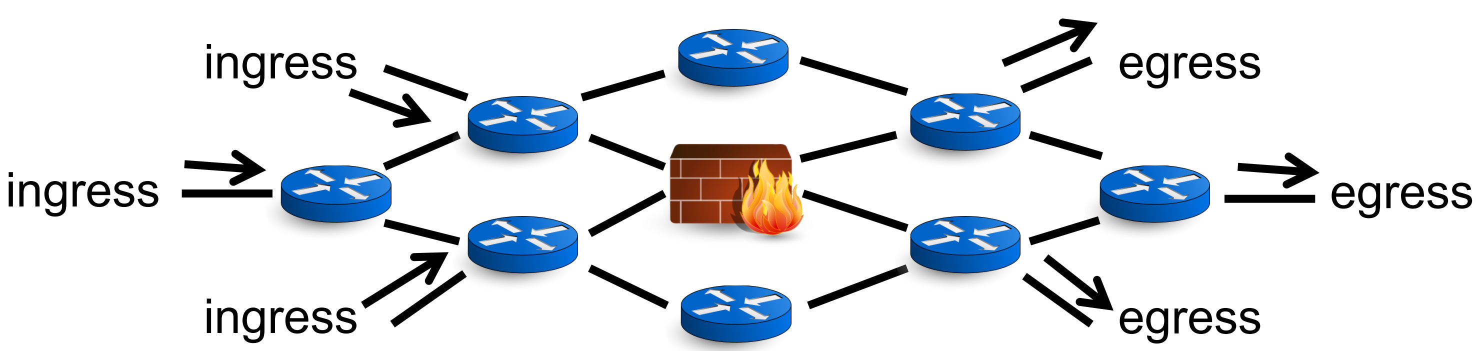
(I) Path Query Language

- *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)

Language: More examples in paper...

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].

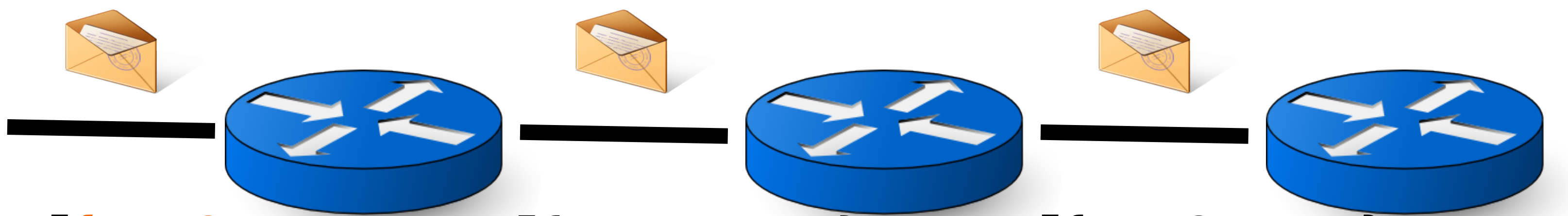
NAT debugging	`in_out_atom(switch=NAT & dstip=192.168.1.10,` `dstip=10.0.1.10)`	Catch packets entering a NAT with destination IP 192.168.1.10 and leaving with the (modified) destination IP 10.0.1.10.
ECMP debugging	`in_out_group(switch=S1 & ecmp_pred,`	Measure ECMP traffic splitting on switch S1 for a small

How do we implement path queries efficiently?

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

How to observe packet paths?

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



```
[{sw: S1
  port: 1
  srcmac: ...
  srcip: ...
  ...}]
```

```
[{sw: S1, ...},
 {sw: S2
 port: 3
 srcmac: ...
 ...}]
```

```
[{sw: S1, ...},
 {sw: S2, ...},
 {sw: S3
  port: 2
  ...}]
```

- Pros: accurate trajectory information ☺
- Cons: too much per-packet information ☹

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

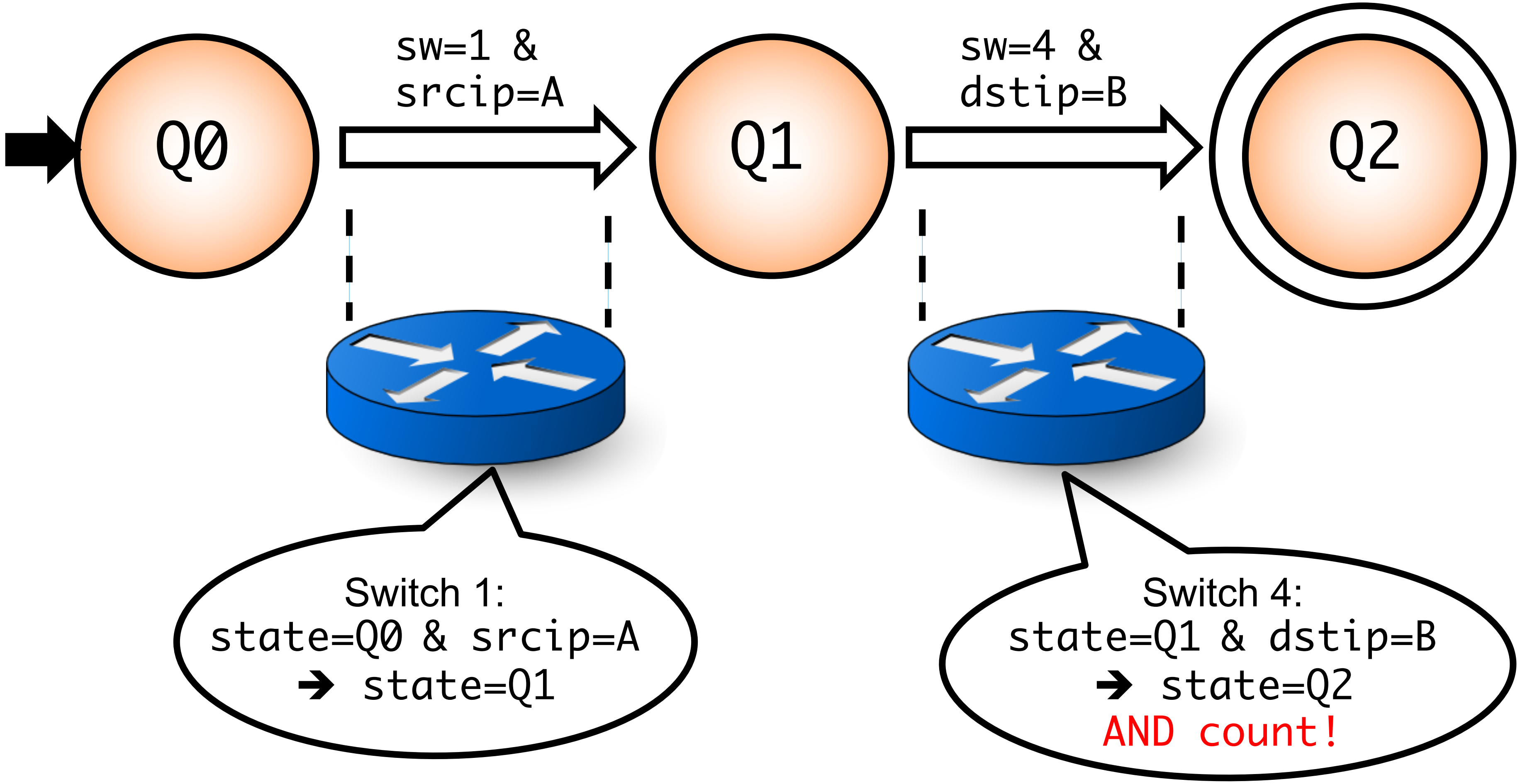
Reducing Path Information on Packets

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) Run-Time: Juicy details in paper…

- Packet forwarding shouldn't be affected by DFA rules
  - No unnecessary duplicate traffic should be created
- Handle query overlap
  - Predicates can also overlap

```
srcip=A ^ dstip=B
sw=1 ^ true* ^ sw=4
```

- Handle `groupby` aggregation
- Capture *upstream* or *downstream* of queried path
  - Test predicates before or after forwarding on switch
- Optimizations: to make the system practical

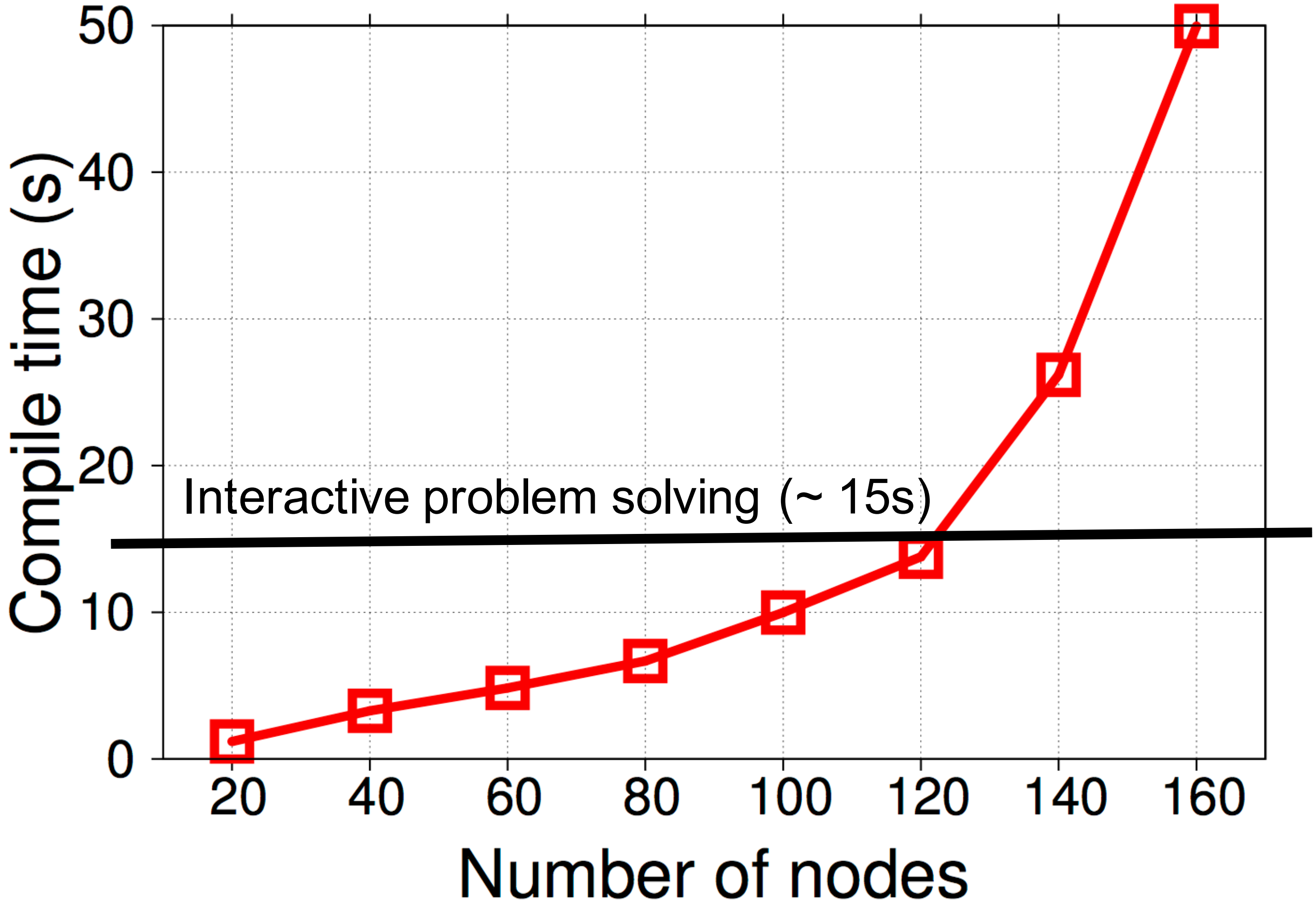
Evaluation

- 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

- Results on Stanford backbone (*all queries together*):
  - Compile time: 5 seconds (from > 2 hours)
  - # Rules: ~ 650
  - # State bytes: 2 bytes

Evaluation: Scaling



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

Summary

- We need good abstractions to measure networks
  - Abstractions must be efficiently implementable
- We implemented declarative queries on packet paths:
  - Packet state akin to a deterministic automaton
- Path queries can simplify network management!

Queries? ☺

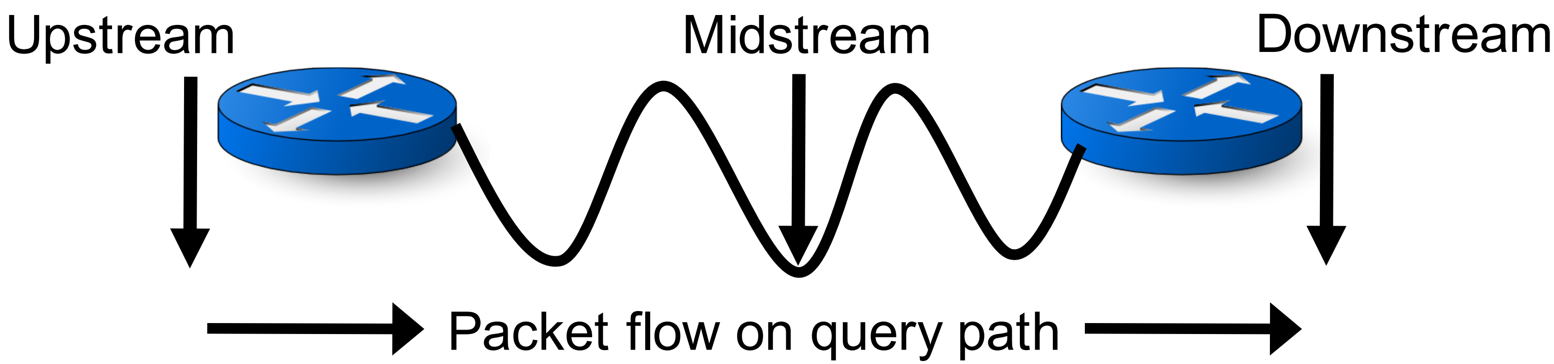
http://www.cs.princeton.edu/~narayana/pathqueries
https://youtu.be/VxOaN9iGPWc

(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)

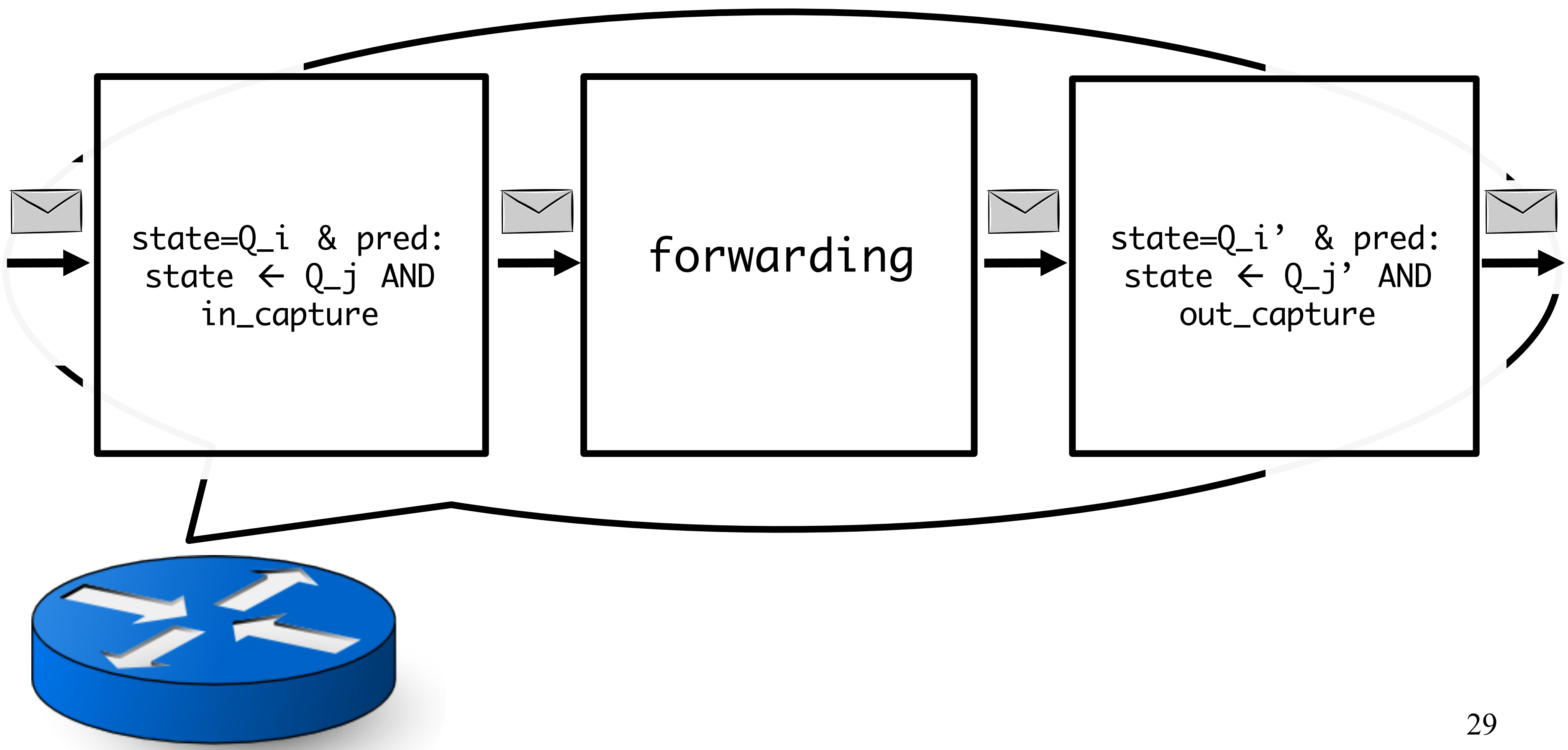
(I) Capture locations

- *Capture* upstream, downstream or midstream



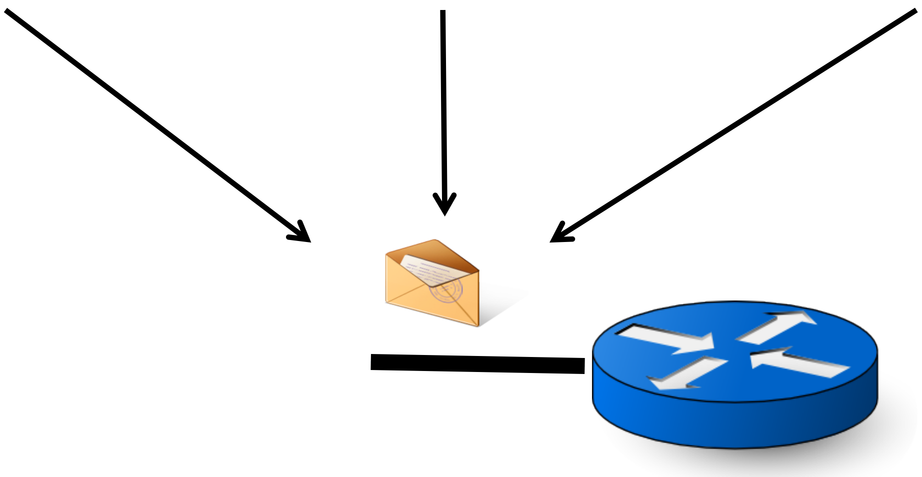
(II) Run-Time: Data Plane Rule Layout

"In" table >> Forwarding >> "Out" table



(II) Query Compilation

(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

(II) Query Compilation

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

```
state=Q0 & switch=S1 & srcip=10.0.0.1 ➔ state←Q1
state=Q1 & switch=S2 & dstip=10.0.0.3 ➔ state←Q2
```

>>

```
dstip=10.0.0.1 ➔ fwd(1)
dstip=10.0.0.2 ➔ fwd(2)
dstip=10.0.0.3 ➔ fwd(3)
...
```

```
state=Q0 & switch=S1 & srcip=10.0.0.1 & dstip=10.0.0.2
➔ state←Q1, fwd(2)
```


Composing software-defined networks. Monsanto et al., 2013

(II) Query Compilation

(DFA-Ingress-Transitioning **>>** Forwarding **>>** DFA-Egress-Transitioning)

+

(DFA-Ingress-Accepting)

+

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

(II) Detecting Query Overlaps

- Predicate overlaps:
  - q1: `srcip=10.0.0.1`; q2: `dstip=10.0.0.2`
  - Automaton can only have one state!

- Query overlaps:
  - q1: `sw=1 ^ sw=2`
  - q2: `srcip=10.0.0.1 ^ dstip=10.0.0.2`

  - q1: `in_atom(srcip=10.0.0.1)`
  - q2: `out_atom(srcip=10.0.0.1)`

  - Automaton states must distinguish all possibilities!

(II) Detecting Query Overlaps

- Predicate overlaps: Generate orthogonal predicates!
  - q1: `srcip=10.0.0.1`; q2: `dstip=10.0.0.2`

  - Generated predicates:
    - `srcip=10.0.0.1 & dstip=10.0.0.2`
    - `srcip=10.0.0.1 & ~dstip=10.0.0.2`
    - `~srcip=10.0.0.1 & dstip=10.0.0.2`

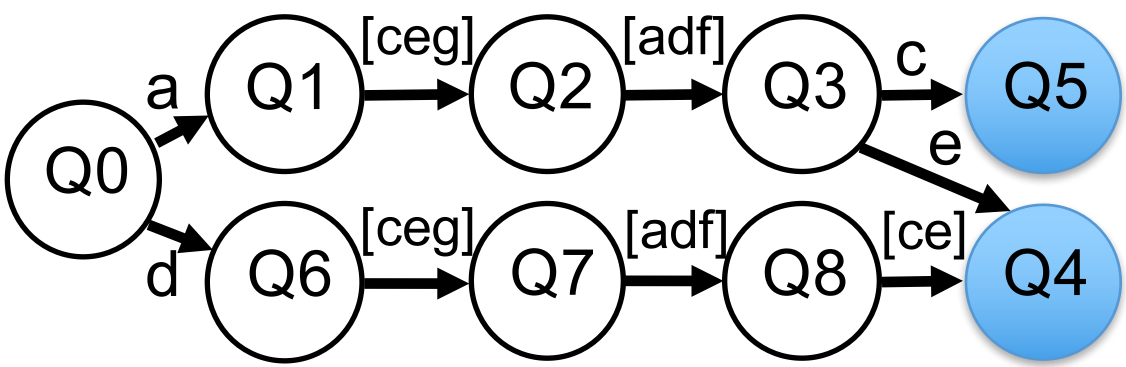
(II) Detecting Query Overlaps

- Query Overlaps:
  - Convert `in_` and `out_` atoms to `in_out_atoms`:
    - `in_atom(srcip=10.0.0.1)` ➔ `in_out_atom( srcip=10.0.0.1, true)`
    - `out_atom(dstip=10.0.0.1)` ➔ `in_out_atom(true, dstip=10.0.0.1)`

(II) Detecting Query Overlaps

- Query Overlaps:

  - Build one DFA for many expressions together
    - `in_atom(srcip=H1 & sw=1) ^ out_atom(sw=2 & dstip=H2)`
    - `in_atom(sw=1) ^ in_out_atom(true, sw=2)`



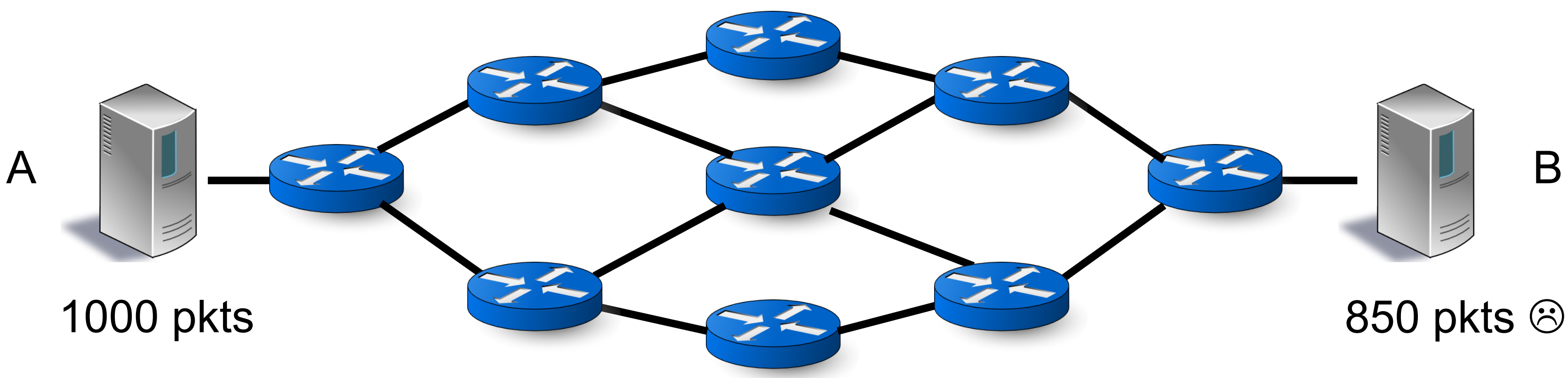
Optimizations: Summary

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		↓	

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

Demo: Where's the Packet Loss?

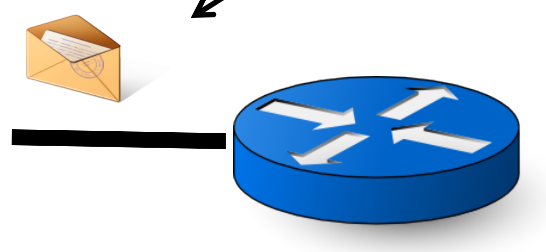


Demo: Where's the Packet Loss?

https://youtu.be/VxOaN9iGPWc

Downstream Query Compilation (3/3)

(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

Downstream Query Compilation (3/3)

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

```
state=Q0 & switch=S1 & srcip=10.0.0.1 ➔ state←Q1
state=Q1 & switch=S2 & dstip=10.0.0.3 ➔ state←Q2
```

>>

```
dstip=10.0.0.1 ➔ fwd(1)
dstip=10.0.0.2 ➔ fwd(2)
dstip=10.0.0.3 ➔ fwd(3)
...
```

```
state=Q0 & switch=S1 & srcip=10.0.0.1 & dstip=10.0.0.2
➔ state←Q1, fwd(2)
```


Composing software-defined networks. Monsanto et al., 2013

Downstream Query Compilation (3/3)

(DFA-Ingress-Transitioning **>>** Forwarding **>>** DFA-Egress-Transitioning)

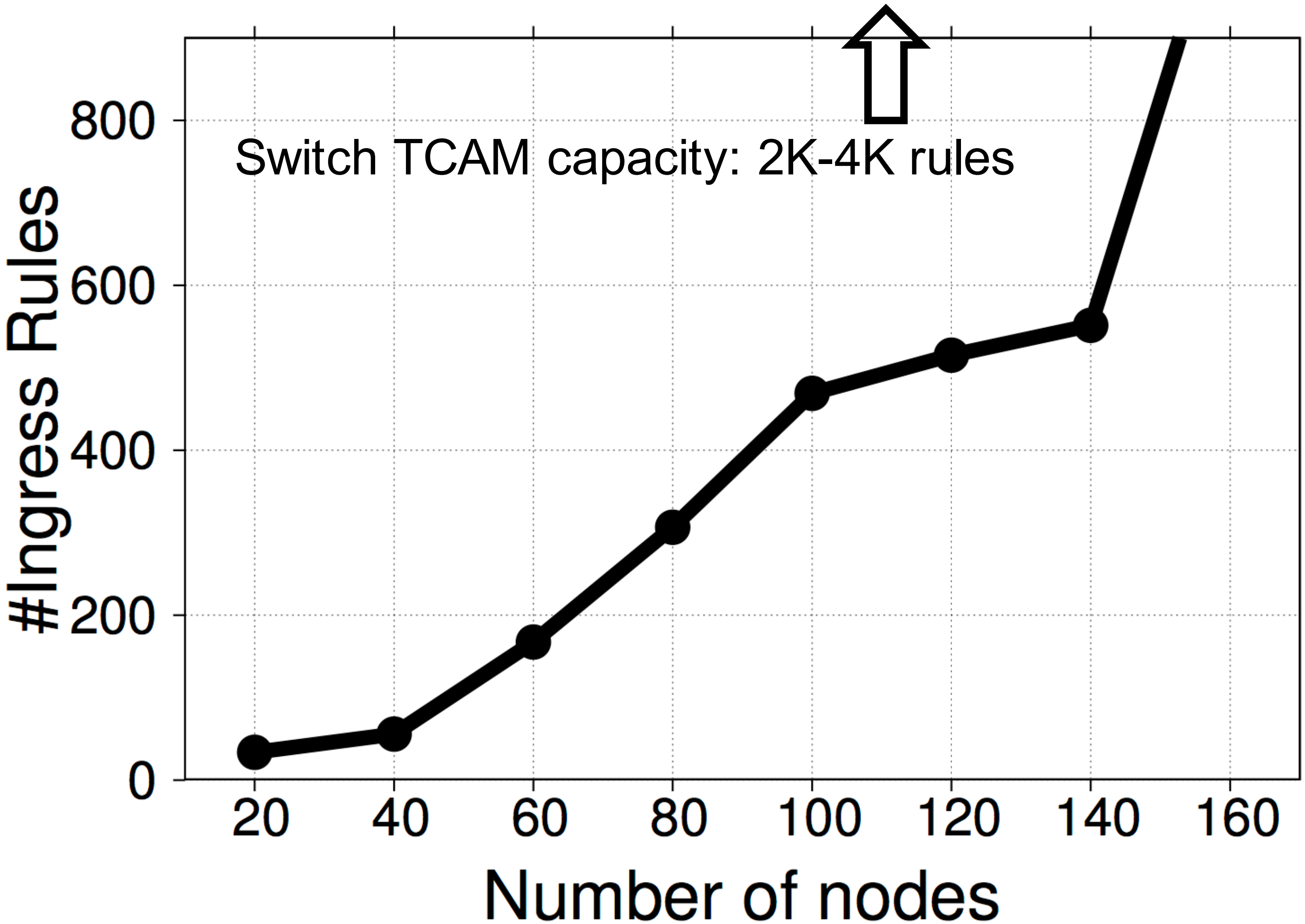
+

(DFA-Ingress-Accepting)

+

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

II. Rule Count



III. Packet State Bits