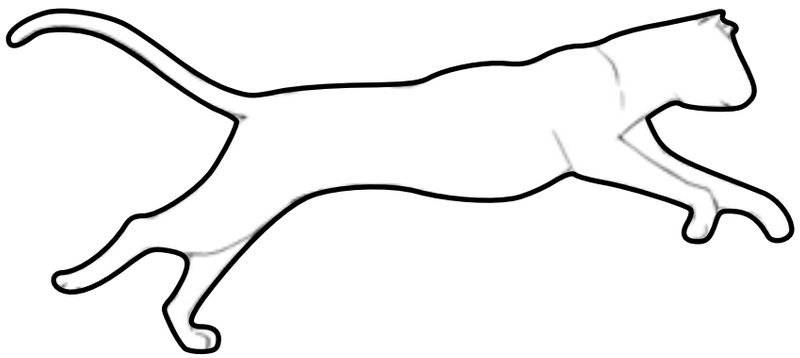
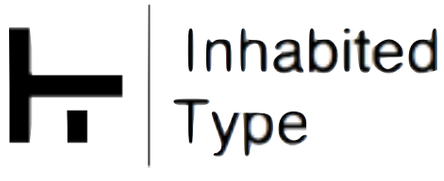


A Fast Compiler for NetKAT

Steffen Smolka
Nate Foster

Arjun Guha

Spiridon Eliopoulos



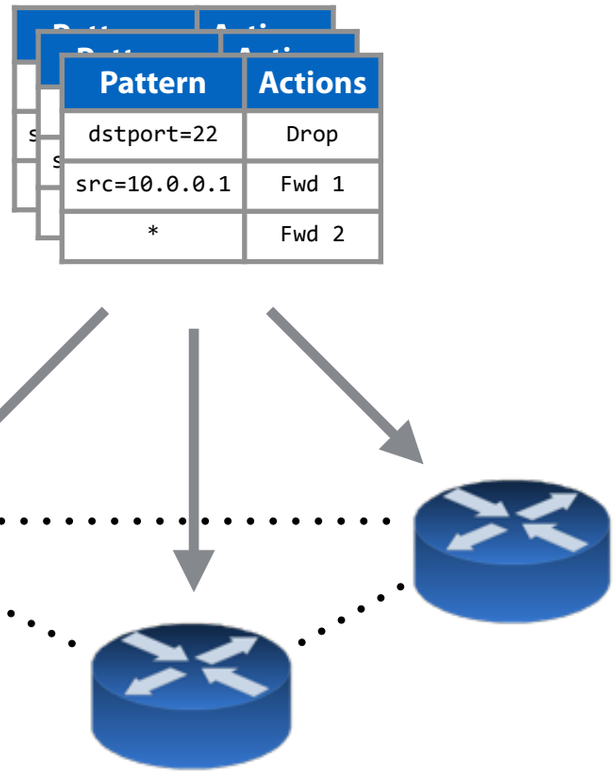
Networks have become programmable (just now!)…

Networks have become programmable (just now!)…

high-level program

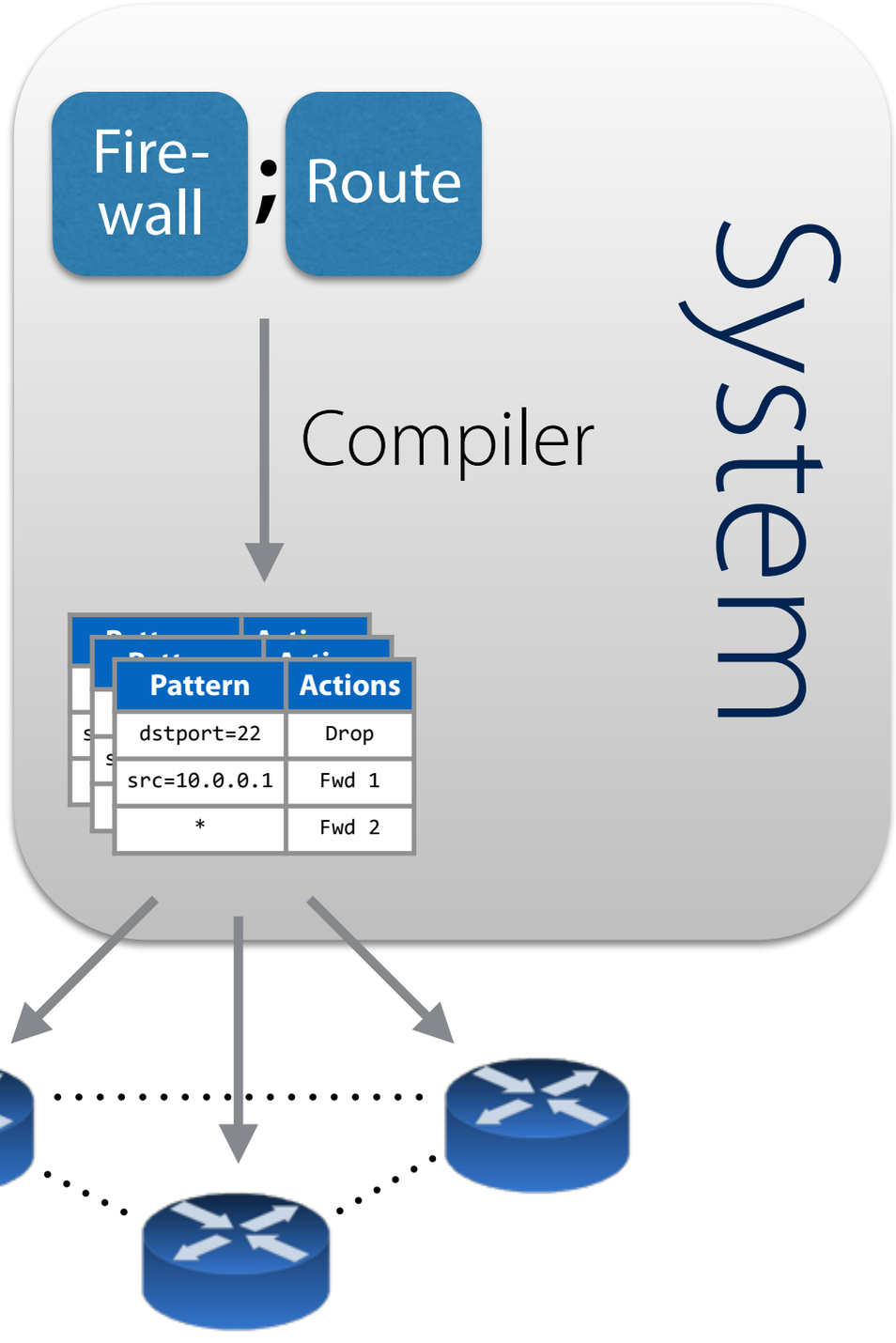
Compiler

low-level rule tables



Networks have become programmable (just now!)…

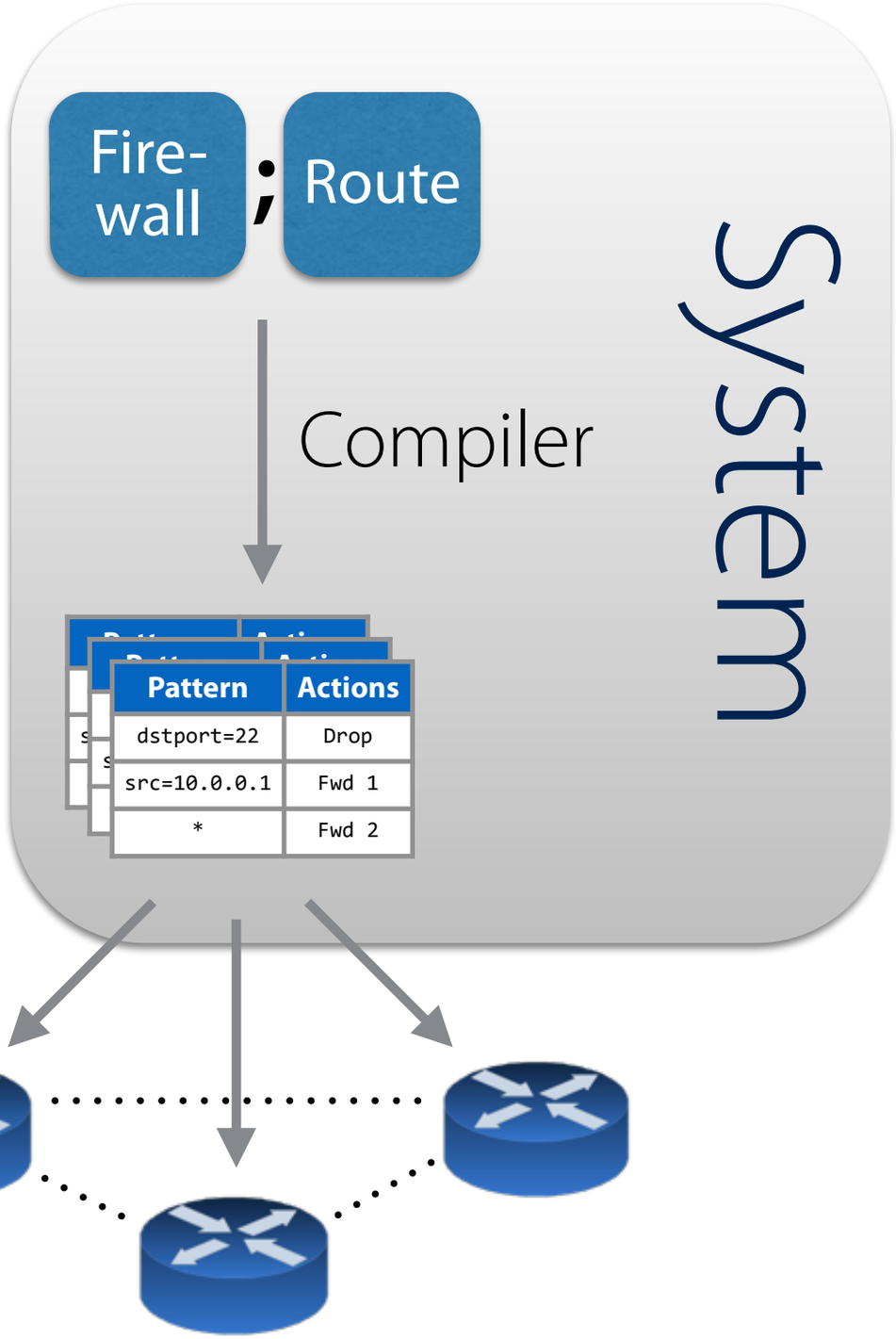
high-level program



low-level rule tables

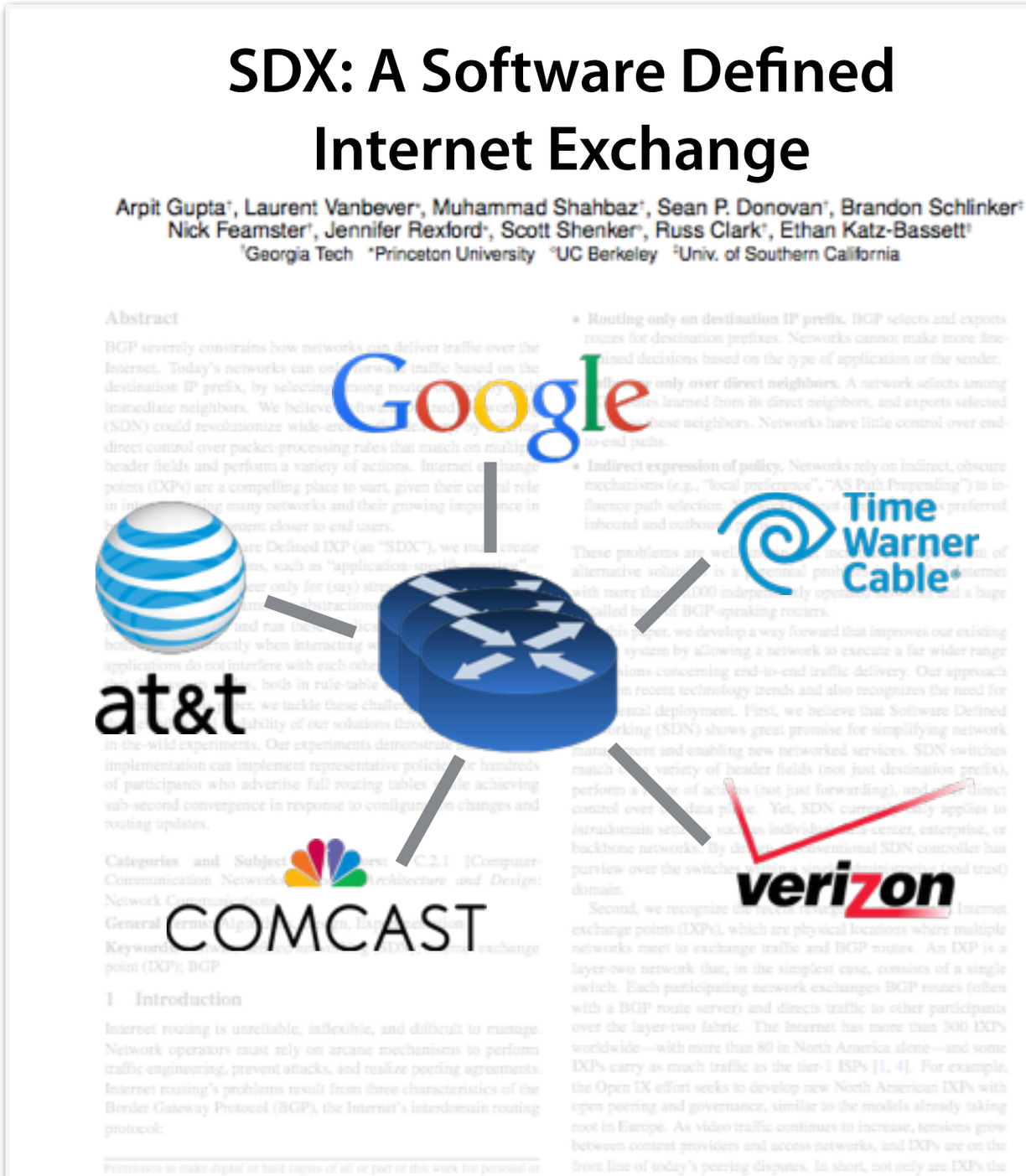
Networks have become programmable (just now!)…

high-level program



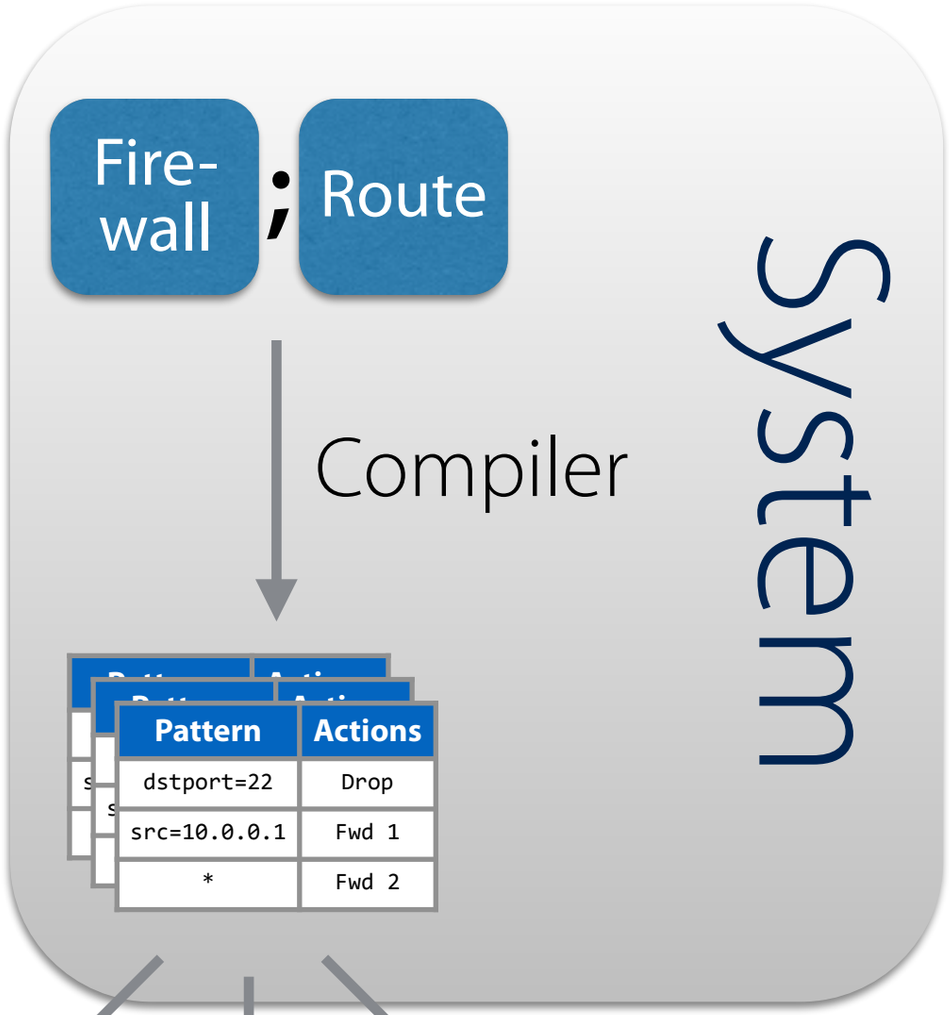
low-level rule tables

SIGCOMM 2014

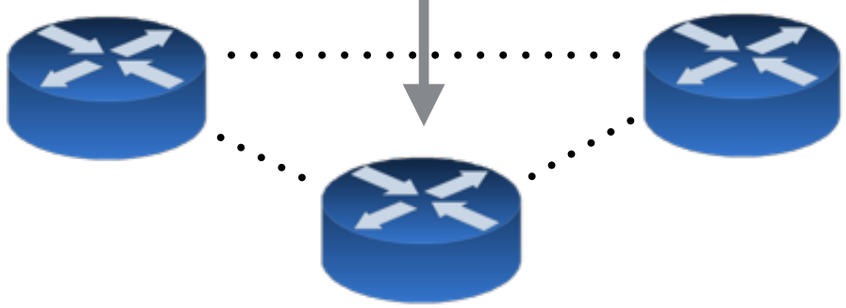


Networks have become programmable (just now!)…

high-level program



low-level rule tables



SIGCOMM 2014

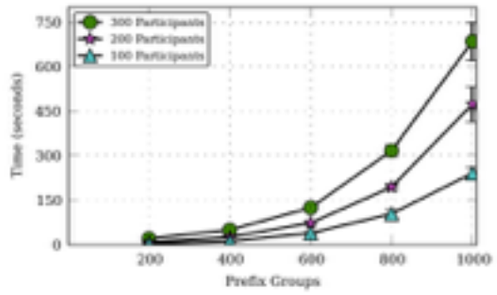


Figure 8: *Compilation time as a function of the number of prefix groups, for different numbers of participants.*

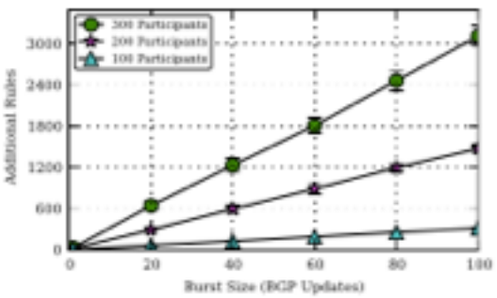


Figure 9: *Number of additional forwarding rules.*

policies at IXPs. The number of forwarding rules increases roughly linearly with the number of prefix groups. Because each prefix group operates on a disjoint portion of the flow space, the increase in forwarding rules is linear in the number of prefix groups.

6.3 Compilation Time

We measure the compilation time for two scenarios: (1) *initial compilation time*, which measures the time to compile the initial set of policies to the resulting forwarding rules; and (2) *incremental compilation time*, which measures how long it takes to recompute when changes occur.

Initial compilation time. Figure 8 shows how the time to compute low-level forwarding rules from higher-level policies varies as we increase both the number of prefix groups and IXP participants. The time to compute the forwarding rules is on the order of several minutes for typical numbers of prefix groups and participants. The results also show that compilation time increases roughly quadratically with the number of prefix groups. The compilation time increases more quickly than linearly because, as the number of prefix groups increases, the interactions between policies of pairs of participants at the SDX also increases. The time for the SDX to compute VNHs increases non-linearly as the number of participants and prefix groups increases. We observed that for 1,000 prefix groups and 100 participants, VNH computation took about five minutes.

As discussed in Section 4.3, the SDX controller achieves faster compilation by memoizing the results of partial policy compilations. Supporting caching for 300 participants at the SDX and 1,000 prefix groups could require a cache of about 4.5 GB. Although this requirement may seem large, it is on the order of the amount of memory required for a route server in a large operational IXP today.

Incremental compilation time. Recall that in addition to computing an initial set of forwarding table rules, the SDX controller must recompile them whenever the best BGP route for a prefix changes or when any participant updates its policy. We now evaluate the benefits of the optimizations that we discussed in Section 4.3 in terms of the savings in compilation time. When new BGP updates arrive at the controller, the controller must recompute VNH IP addresses

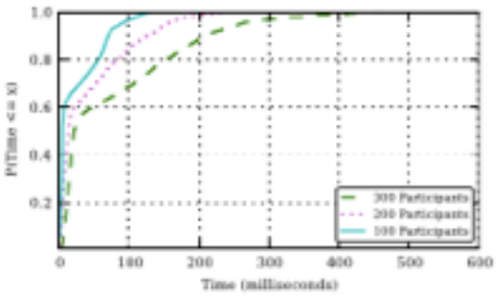


Figure 10: *Time to process a single BGP update for various participants.*

number of additional forwarding rules that depends on the number of participants with policies installed. In practice, as we discussed in Section 4.3, not every BGP update induces changes in forwarding table entries. When a BGP update arrives, the SDX controller installs additional flow table rules for the affected flows and computes a new optimized table *in the background* to ultimately coalesce these flows into the smaller, minimal forwarding tables. As shown in Figure 10, re-computing the tables takes less than 100 milliseconds most of the time.

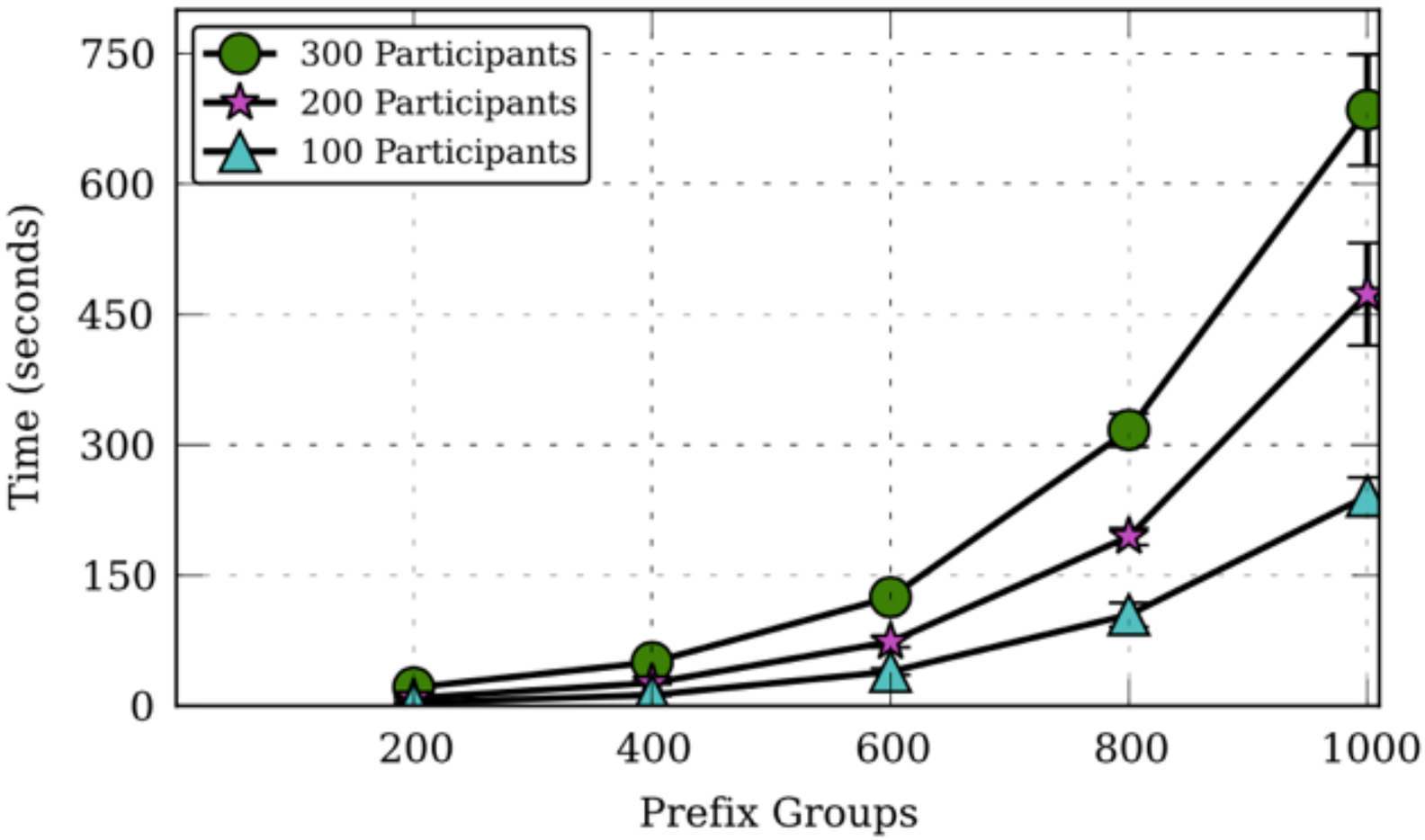
7 Related Work

We briefly describe related work in SDN exchange points, interdomain route control, and policy languages for SDNs.

SDN-based exchange points. The most closely related work is Google's Cardigan project [22], which shares our broad goal of using SDN to enable innovation at IXPs. Cardigan runs a route server based on RouteFlow [17] and uses an OpenFlow switch to enforce security and routing policies. The Cardigan project is developing a logical SDN-based exchange point that is physically distributed across multiple locations. Unlike the SDX in this paper, Cardigan does not provide a general controller for composing participant policies, offer a framework that allows IXP participants to write

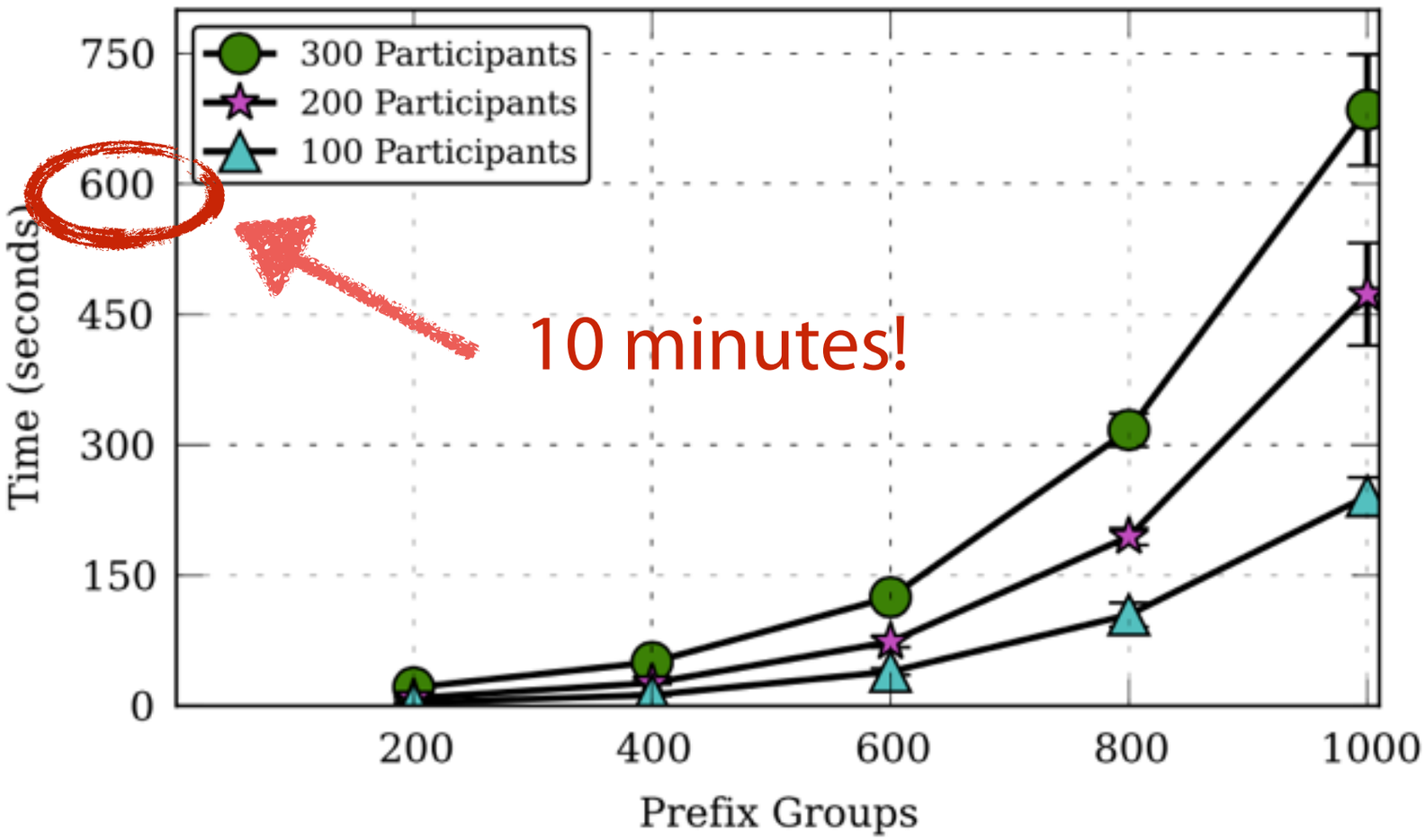
Networks have become programmable (just now!)…

Compilation Time



Networks have become programmable (just now!)…

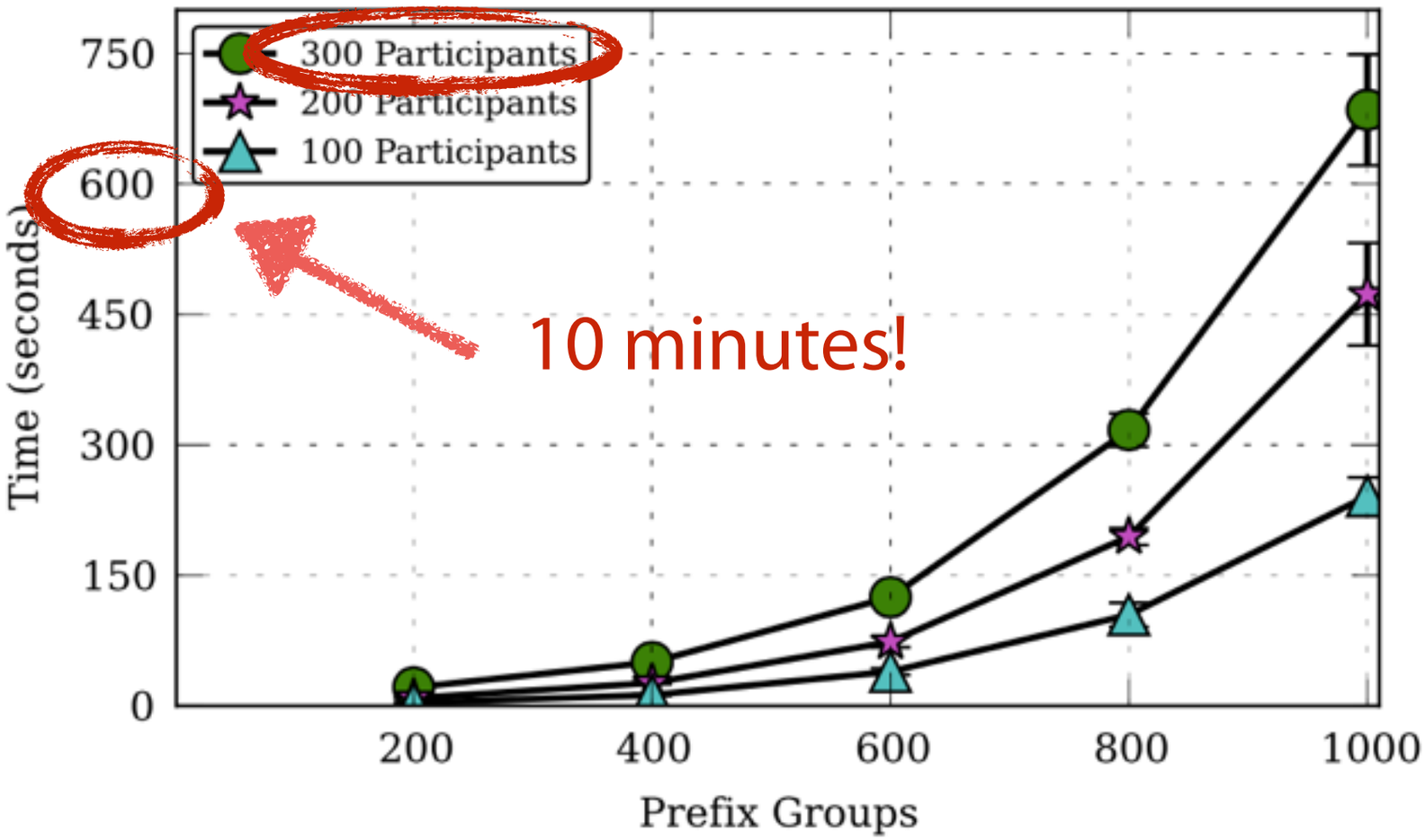
Compilation Time



…with ad hoc performance hacks
(some of which turned out to be unsound)

Networks have become programmable (just now!)…

Compilation Time



Top 5 IXPs

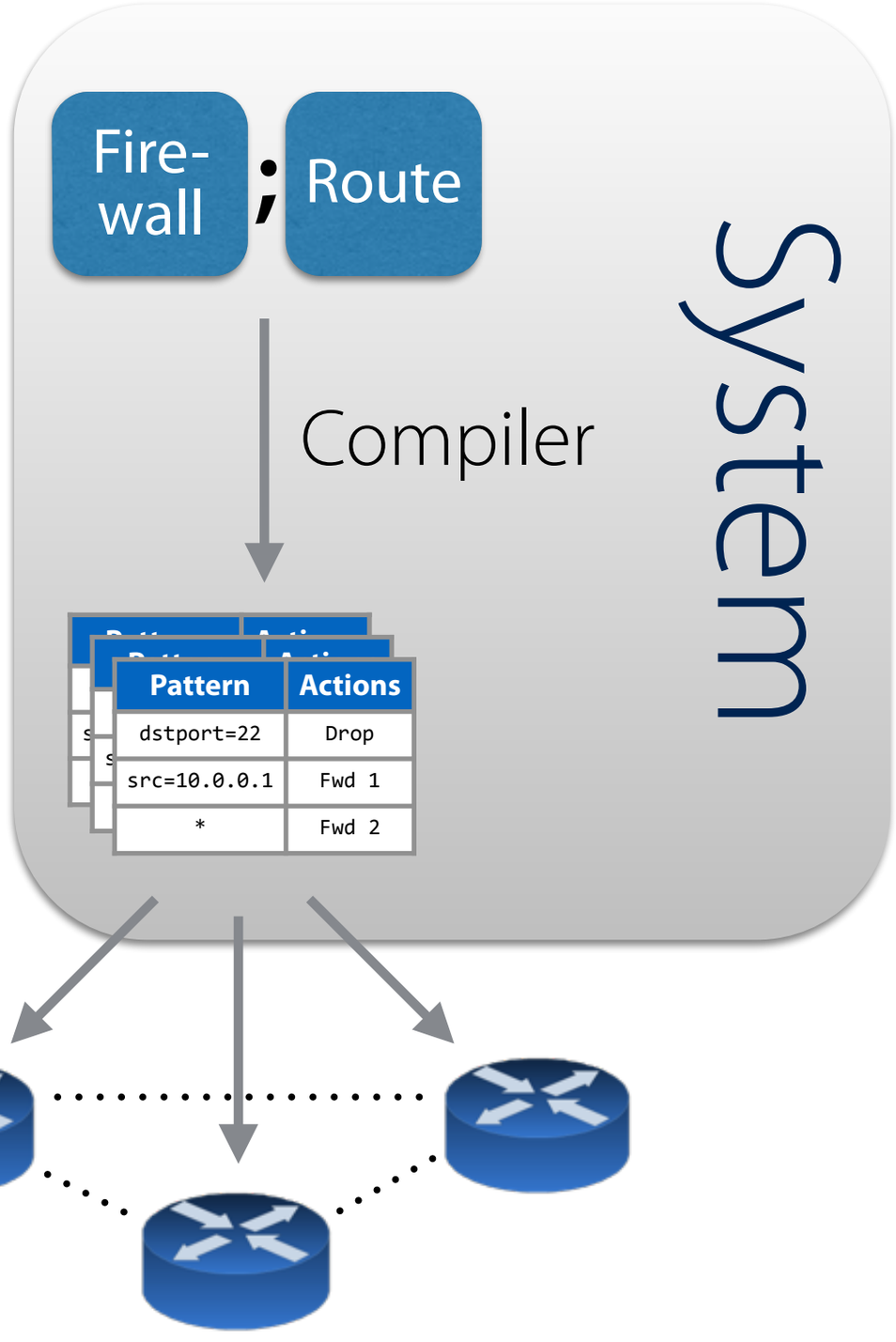
Name	Participants
IX.br	861
Equinix	768
AMS-IX	710
LINX	652
DE-CIX	610

Source: Wikipedia

…with ad hoc performance hacks
(some of which turned out to be unsound)

Networks have become programmable (just now!)…

high-level program



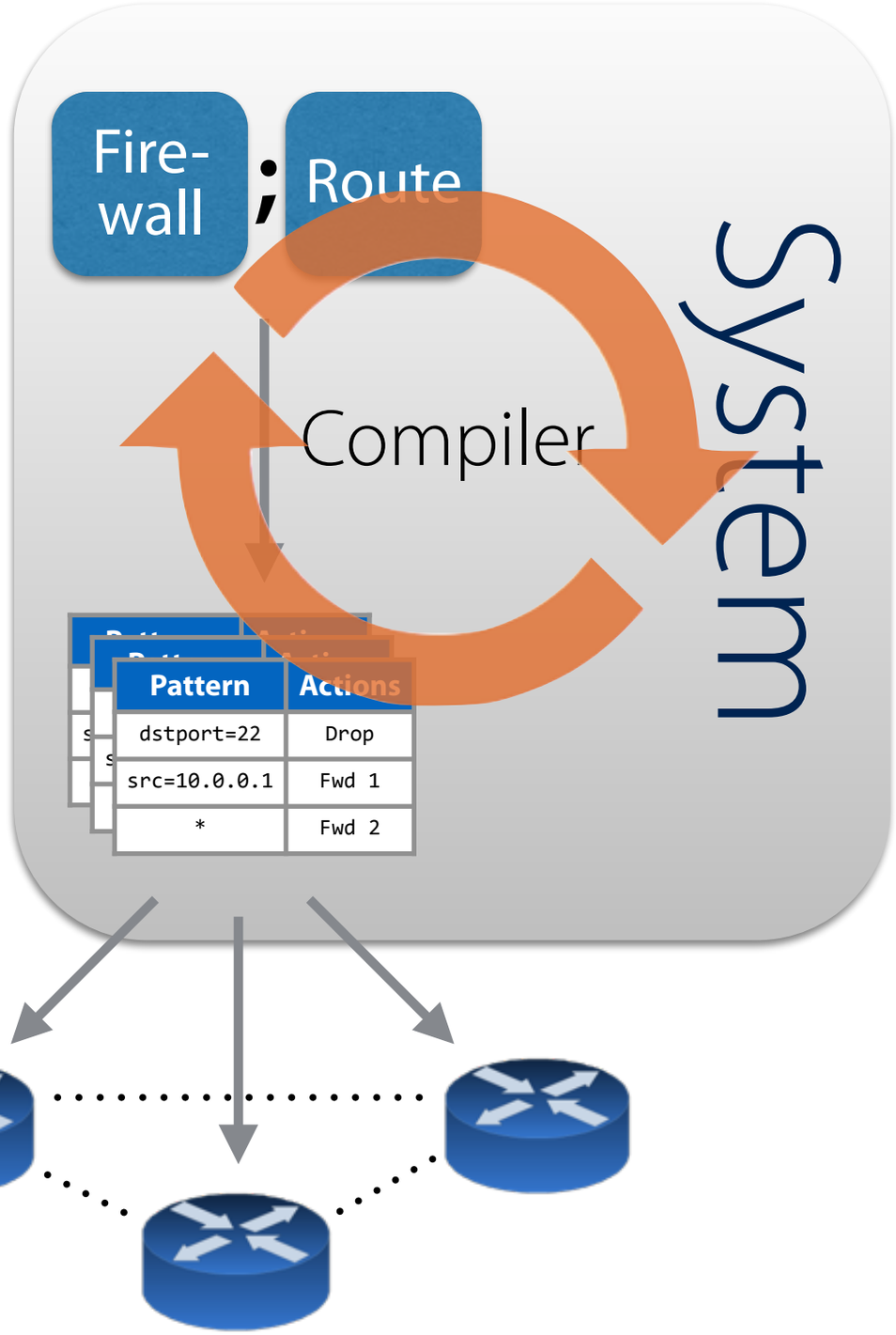
low-level rule tables

…but current compilers are

→ too slow

Networks have become programmable (just now!)…

high-level program



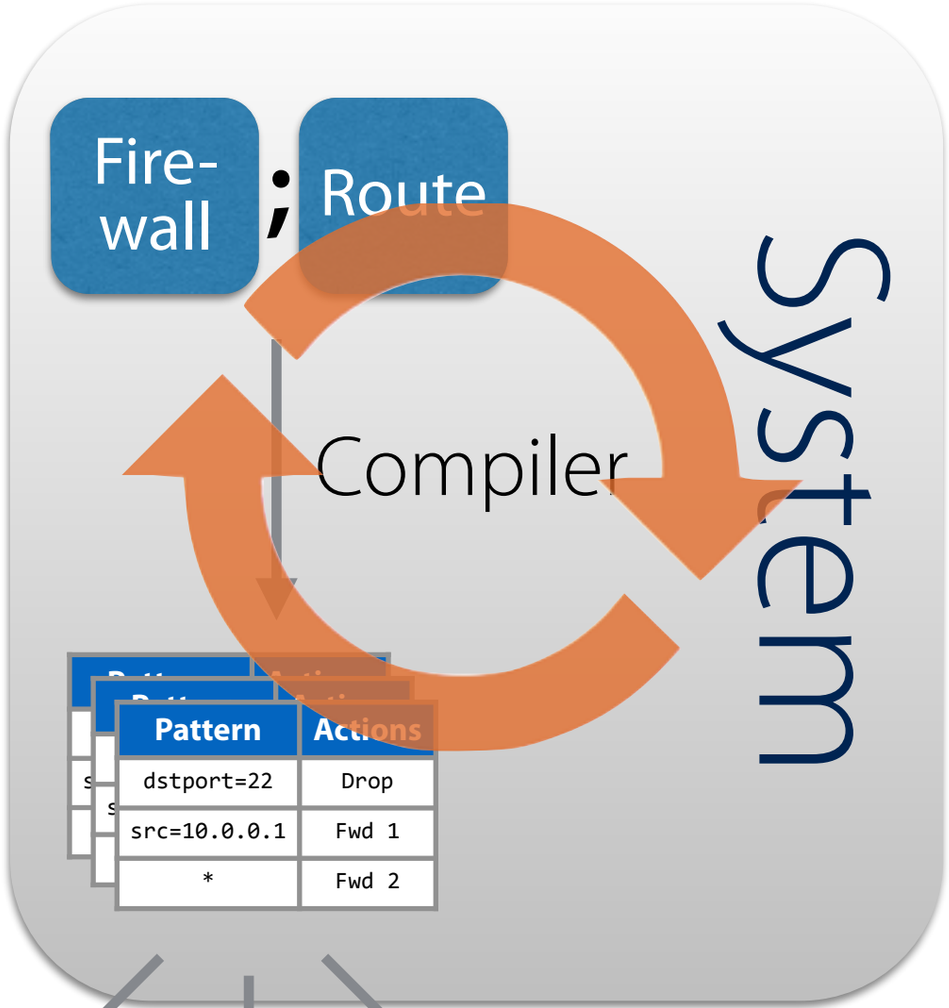
low-level rule tables

…but current compilers are

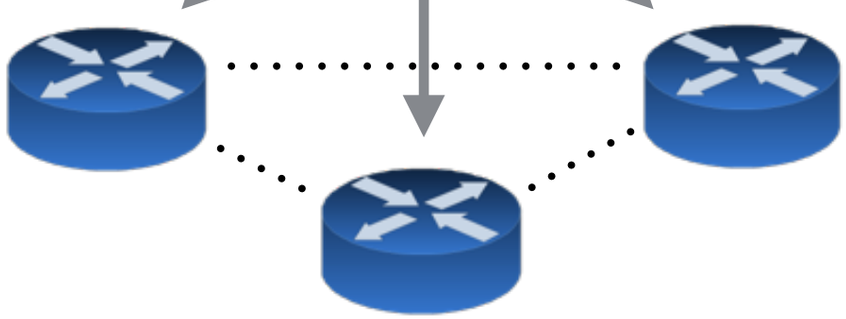
→ too slow

Networks have become programmable (just now!)…

high-level program



low-level rule tables



…but current compilers are

→ too slow

→ limited to "local" languages

Our Contribution

First ***complete*** compiler pipeline for NetKAT

Our Contribution

First ***complete*** compiler pipeline for NetKAT

drop-in replacement
~ 100x speedup

Our Contribution

First ***complete*** compiler pipeline for NetKAT

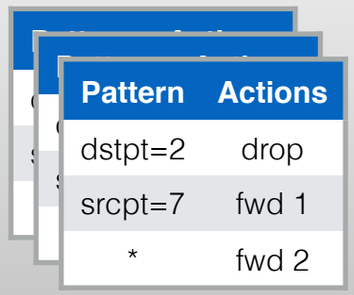
network-wide behavior

drop-in replacement
~ 100x speedup

Our Contribution

First ***complete*** compiler pipeline for NetKAT

virtual policy → **Virtual** Compiler → global policy → **Global** Compiler → local policy → **Local** Compiler



abstract topologies

network-wide behavior

drop-in replacement
~ 100x speedup

Our Contribution

First ***complete*** compiler pipeline for NetKAT

virtual policy

Virtual Compiler

global policy

Global Compiler

local policy

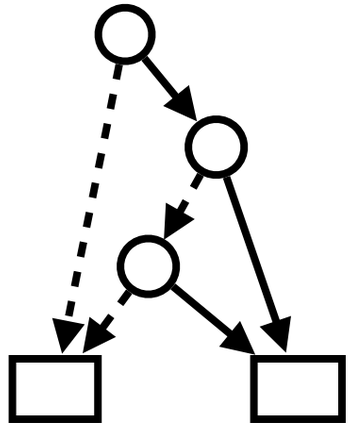
Local Compiler

Pattern	Actions
dstpt=2	drop
srcpt=7	fwd 1
*	fwd 2

abstract topologies

network-wide behavior

drop-in replacement
~ 100x speedup



Our Contribution

First ***complete*** compiler pipeline for NetKAT

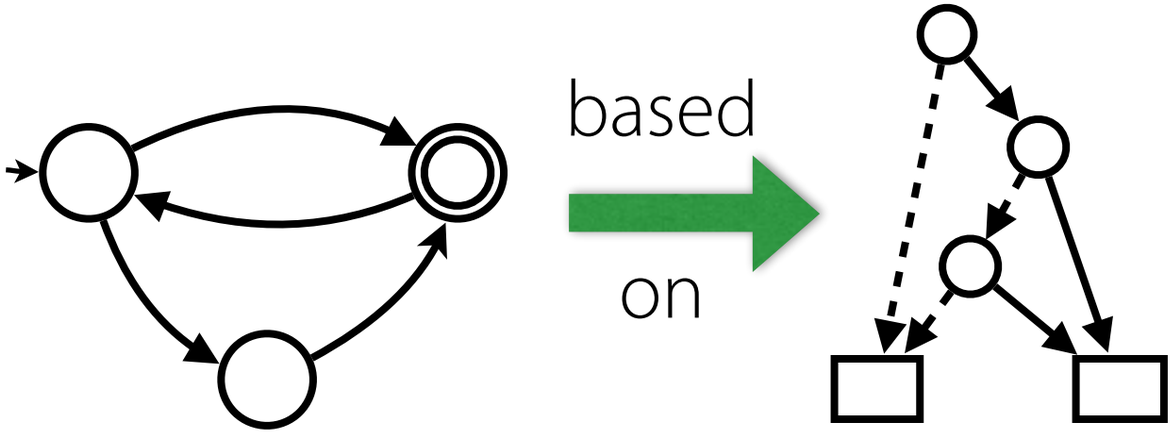
virtual policy → **Virtual** Compiler → global policy → **Global** Compiler → local policy → **Local** Compiler → flow tables

Pattern	Actions
dstpt=2	drop
srcpt=7	fwd 1
*	fwd 2

abstract topologies

network-wide behavior

drop-in replacement
~ 100x speedup



Our Contribution

First ***complete*** compiler pipeline for NetKAT

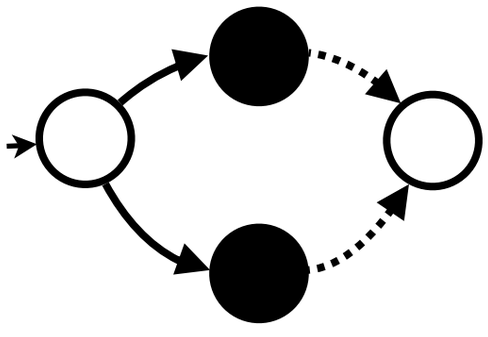
virtual policy → **Virtual** Compiler → global policy → **Global** Compiler → local policy → **Local** Compiler → flow tables

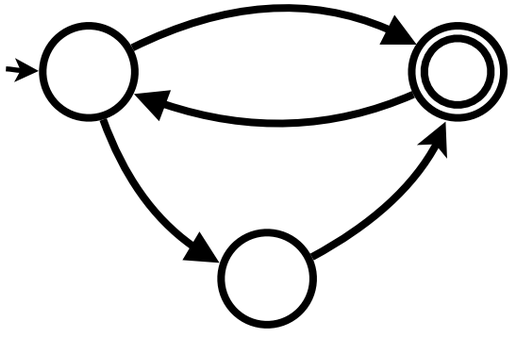
Pattern	Actions
dstpt=2	drop
srcpt=7	fwd 1
*	fwd 2

abstract topologies

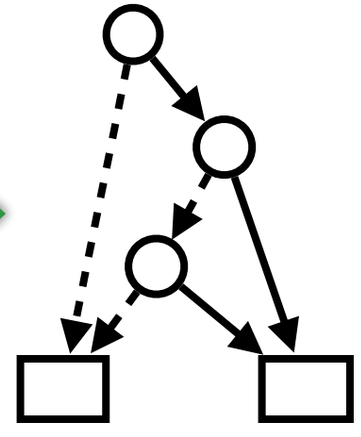
network-wide behavior

drop-in replacement
~ 100x speedup

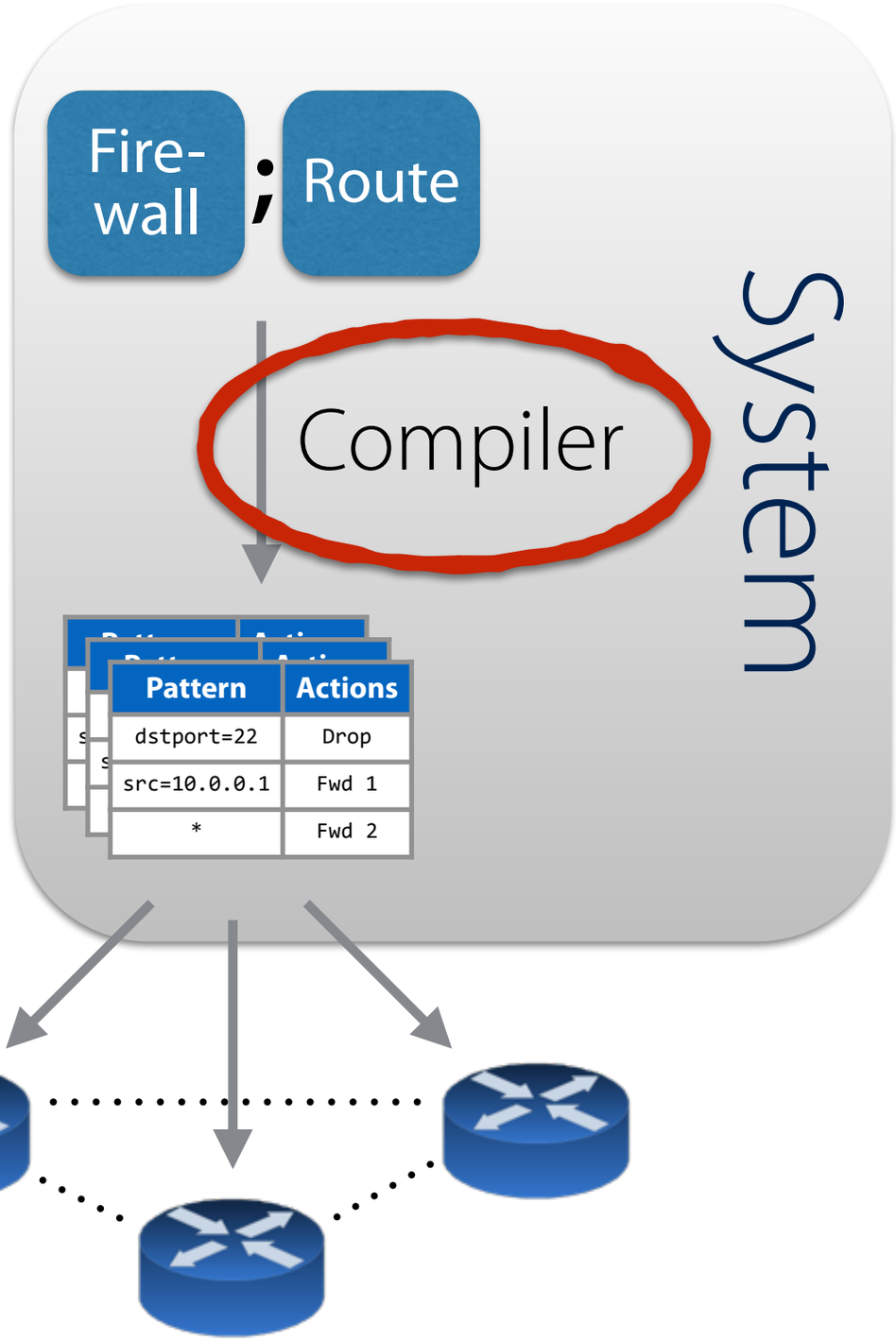




based on



high-level program



low-level rule tables

Source Language?

Target Language?

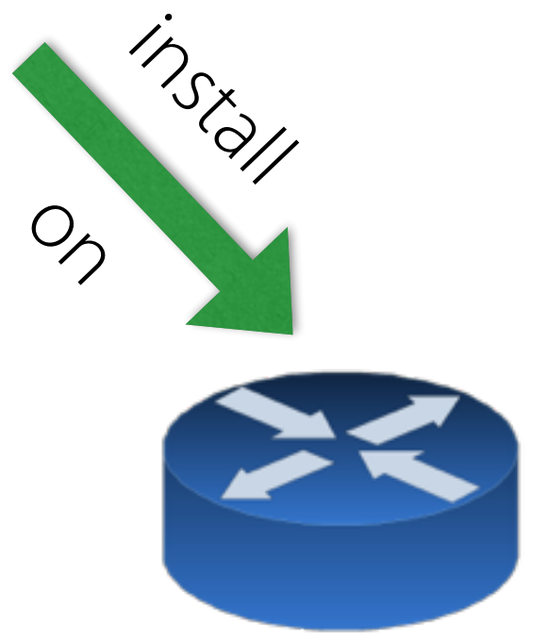
The Target: Match+Action Tables

Match	Actions
port=2, srcIP=10.0.0.1	Fwd 1
port=2	Drop
port=1	dstIP=10.0.0.2, Fwd 2
*	Fwd 1, Fwd 2

= "Ordered Lookup Table"

The Target: Match+Action Tables

Match	Actions
port=2, srcIP=10.0.0.1	Fwd 1
port=2	Drop
port=1	dstIP=10.0.0.2, Fwd 2
*	Fwd 1, Fwd 2

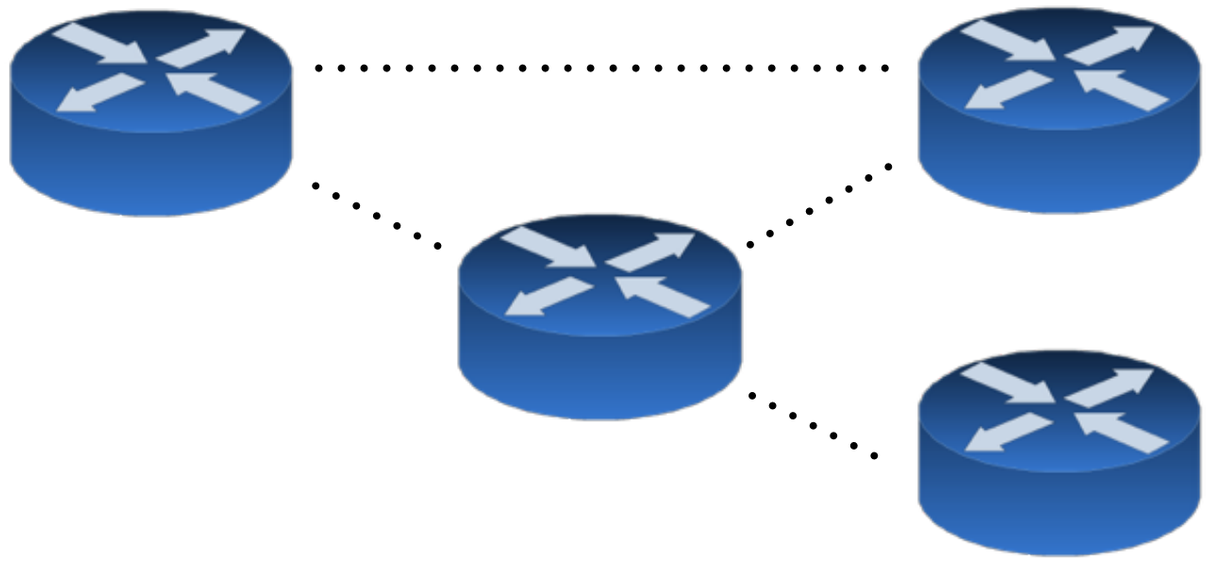


= "Ordered Lookup Table"

→ designed for efficient execution in hardware

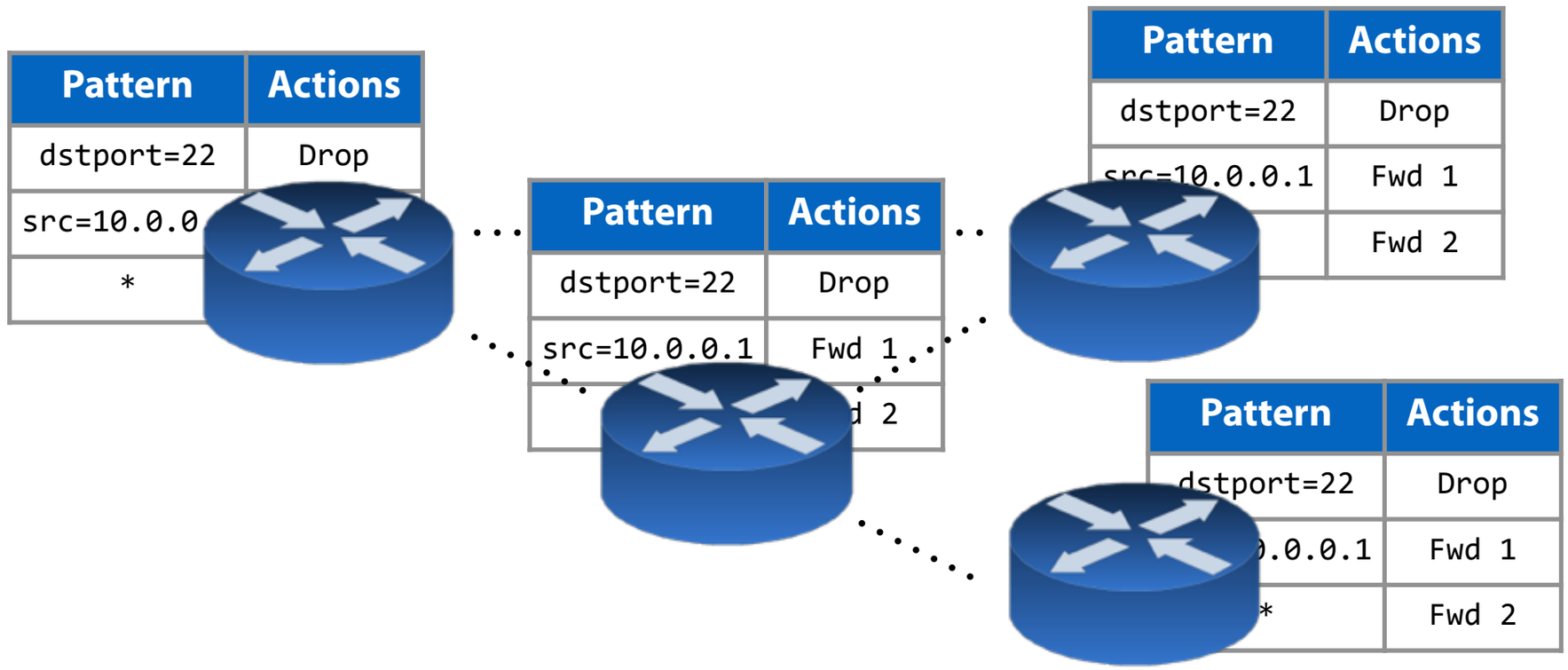
The Goal:

Create one table for each switch in the network…



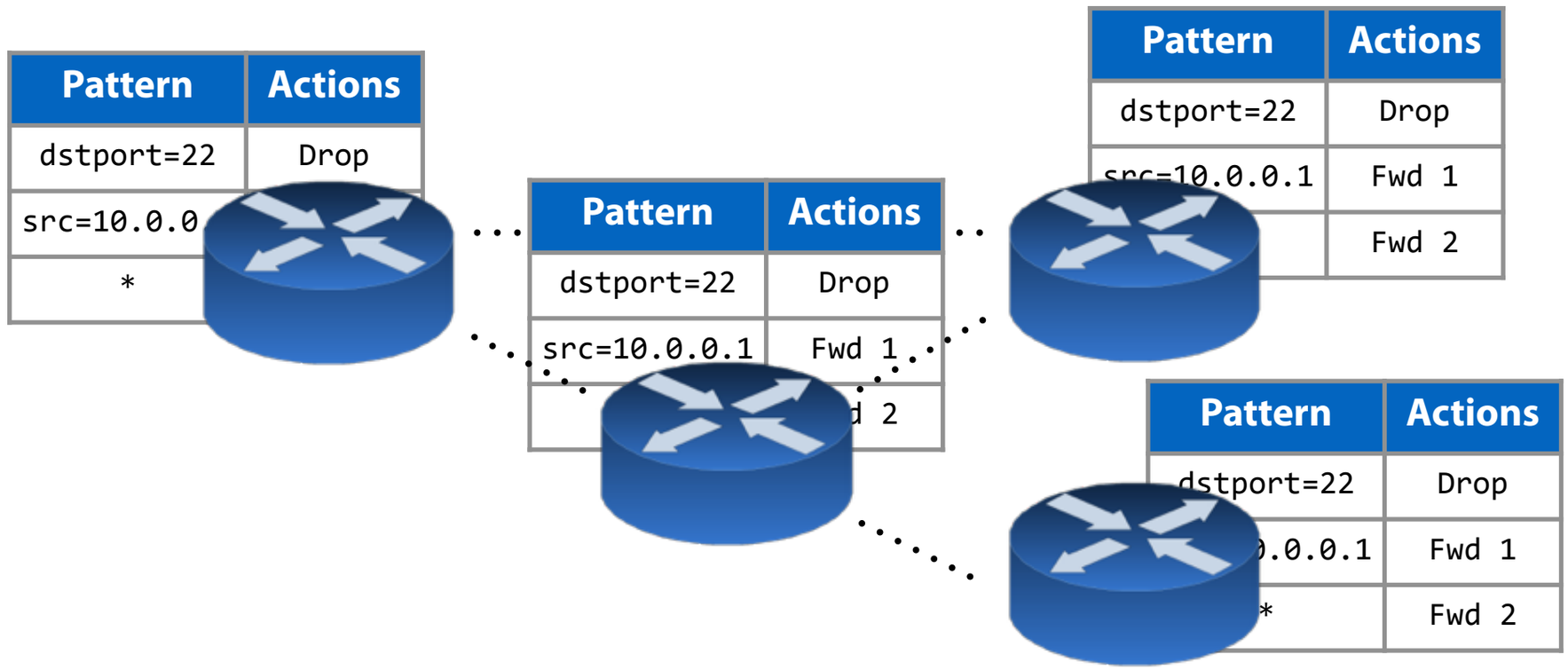
The Goal:

Create one table for each switch in the network…



The Goal:

Create one table for each switch in the network…



…given a high-level program in the source language

The Source: NetKAT Language

pol ::=
- | **false**
- | **true**
- | field **=** val
- | field **:=** val
- | pol_1 **+** pol_2
- | pol_1 **;** pol_2
- | !pol
- | pol*
- | S⟶S'

The Source: NetKAT Language

pol ::=

| **false**

| **true**

| field **=** val

| field **:=** val

| pol_1 **+** pol_2

| pol_1 **;** pol_2

| !pol

| pol*

| S→S'

Boolean Algebra

The Source: NetKAT Language

```
pol ::=
      | false
      | true
      | field = val
      | field := val
      | pol1 + pol2
      | pol1 ; pol2
      | !pol
      | pol*
      | S→S'
```

Boolean Algebra

+

Kleene Algebra
"Regular Expressions"

The Source: NetKAT Language

```
pol ::=
      | false
      | true
      | field = val
      | field := val
      | pol1 + pol2
      | pol1 ; pol2
      | !pol
      | pol*
      | S→S'
```

Boolean Algebra

+

Kleene Algebra
"Regular Expressions"

+

Packet Primitives

Semantics

pol ::=

| **false**

| **true**

| field **=** val

| field **:=** val

| pol_1 **+** pol_2

| pol_1 **;** pol_2

| !pol

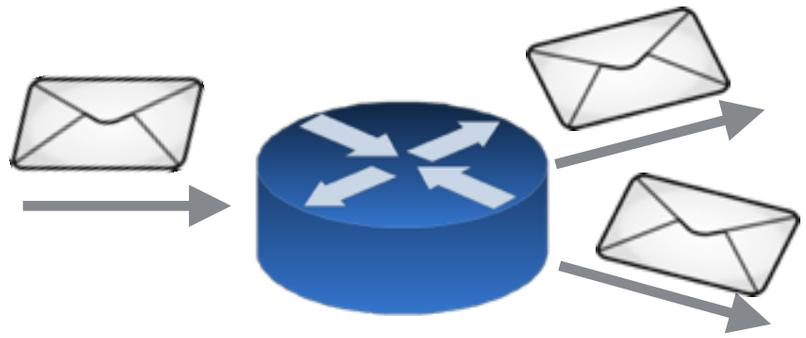
| pol*

| ~~S ↦ S'~~

Semantics

pol ::=
- | **false**
- | **true**
- | field **=** val
- | field **:=** val
- | pol_1 **+** pol_2
- | pol_1 **;** pol_2
- | !pol
- | pol*
- | ~~S ⇝ S'~~

Local NetKAT: input-output behavior of switches

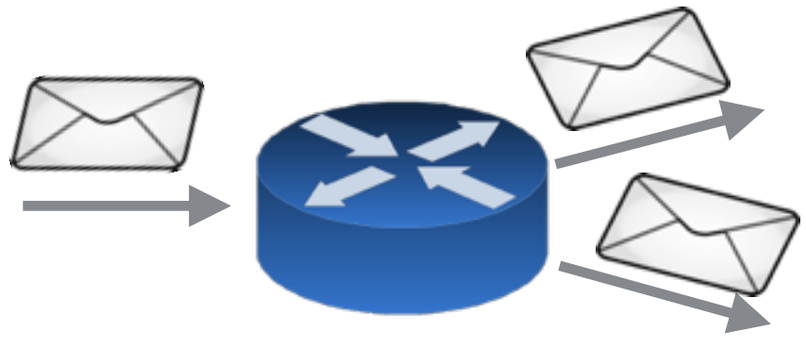


⟦pol⟧ ∈ Packet → Packet Set

Semantics

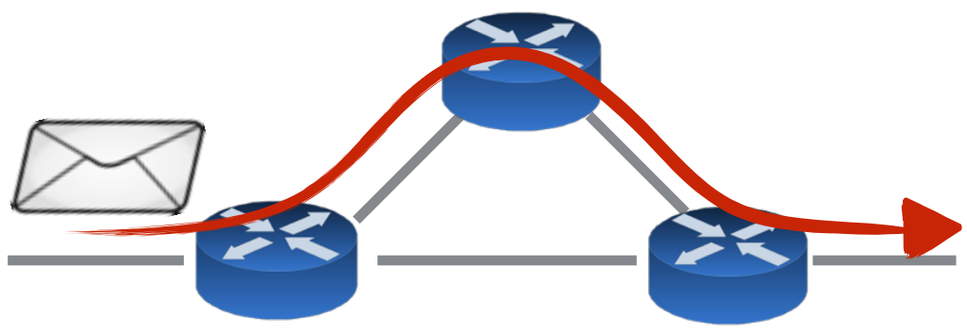
pol ::=
- | **false**
- | **true**
- | field **=** val
- | field **:=** val
- | pol_1 **+** pol_2
- | pol_1 **;** pol_2
- | !pol
- | pol*
- | S→S'

Local NetKAT: input-output behavior of switches



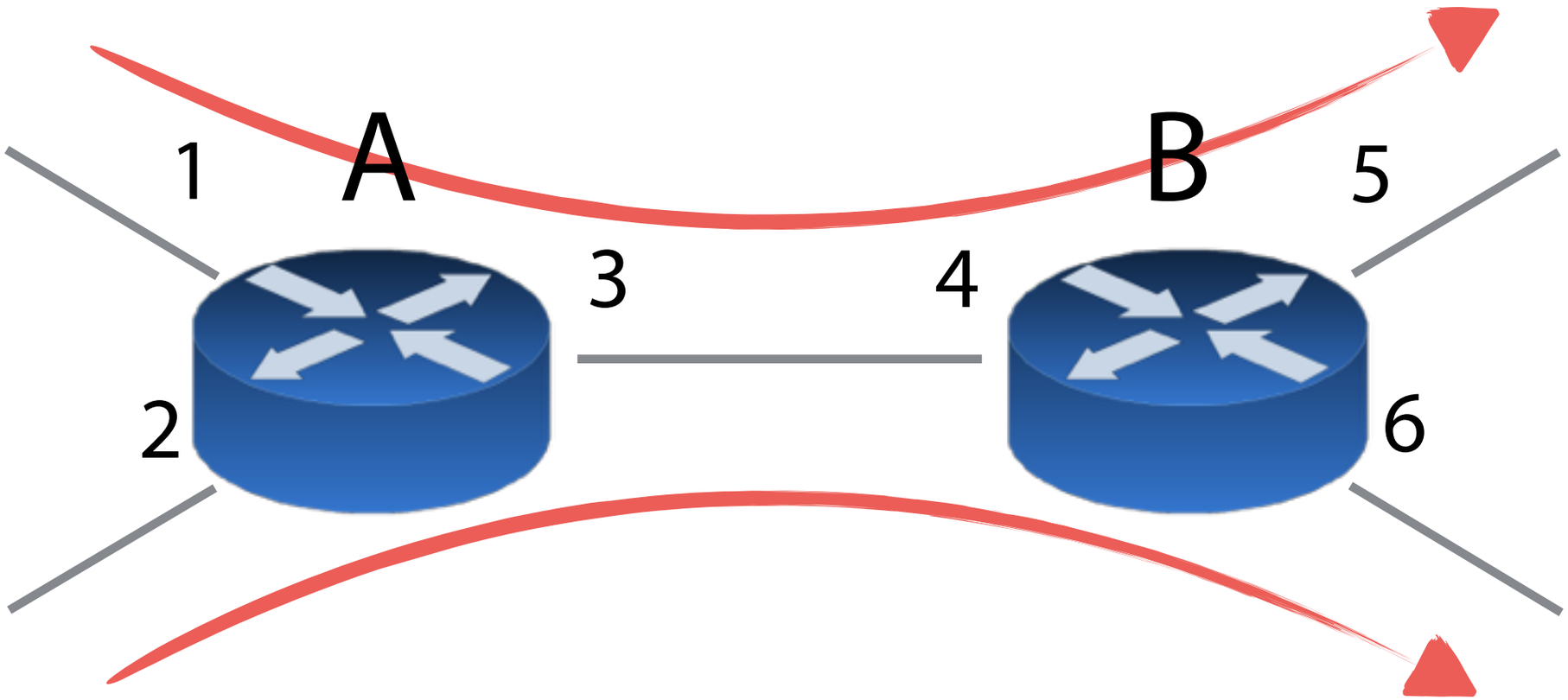
⟦pol⟧ ∈ Packet → Packet Set

Global NetKAT: network-wide behavior

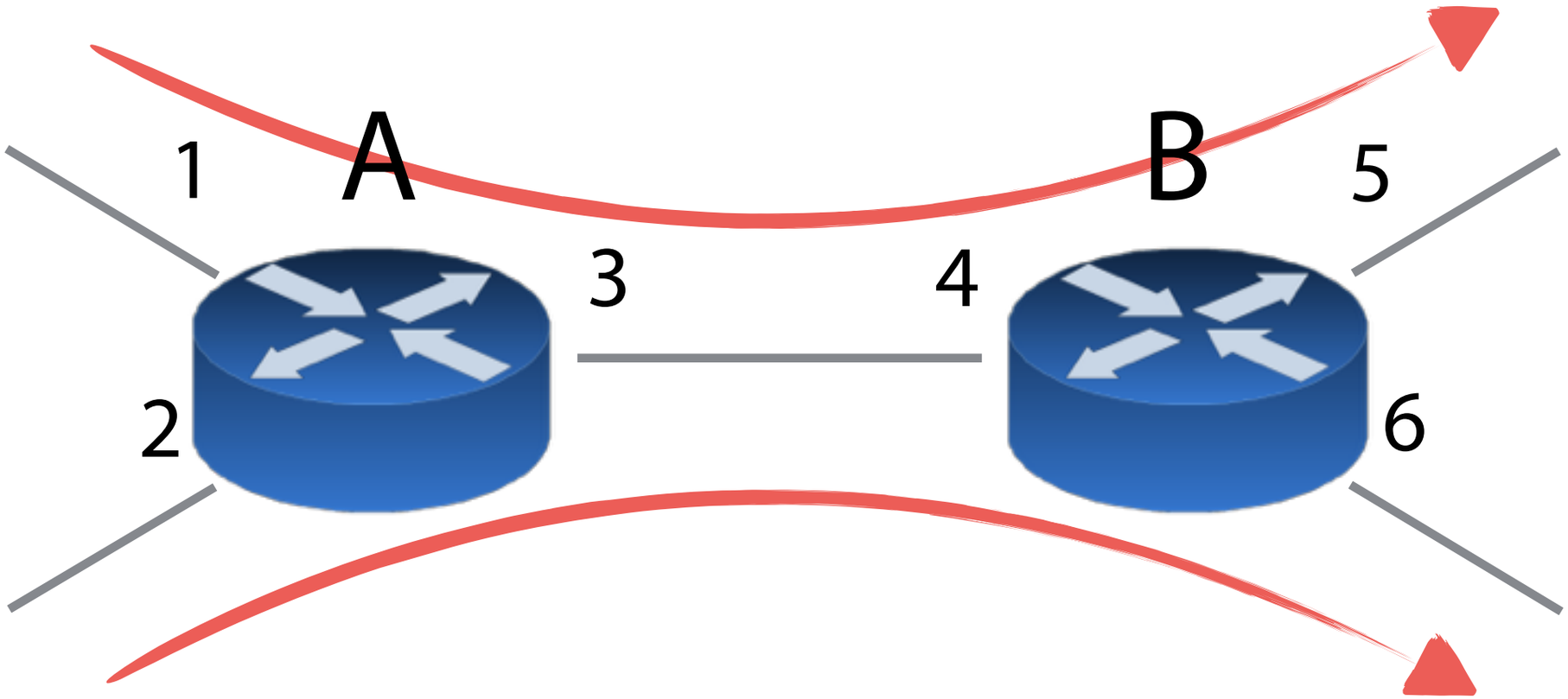


⟦pol⟧ ∈ Trace → Trace Set

Example



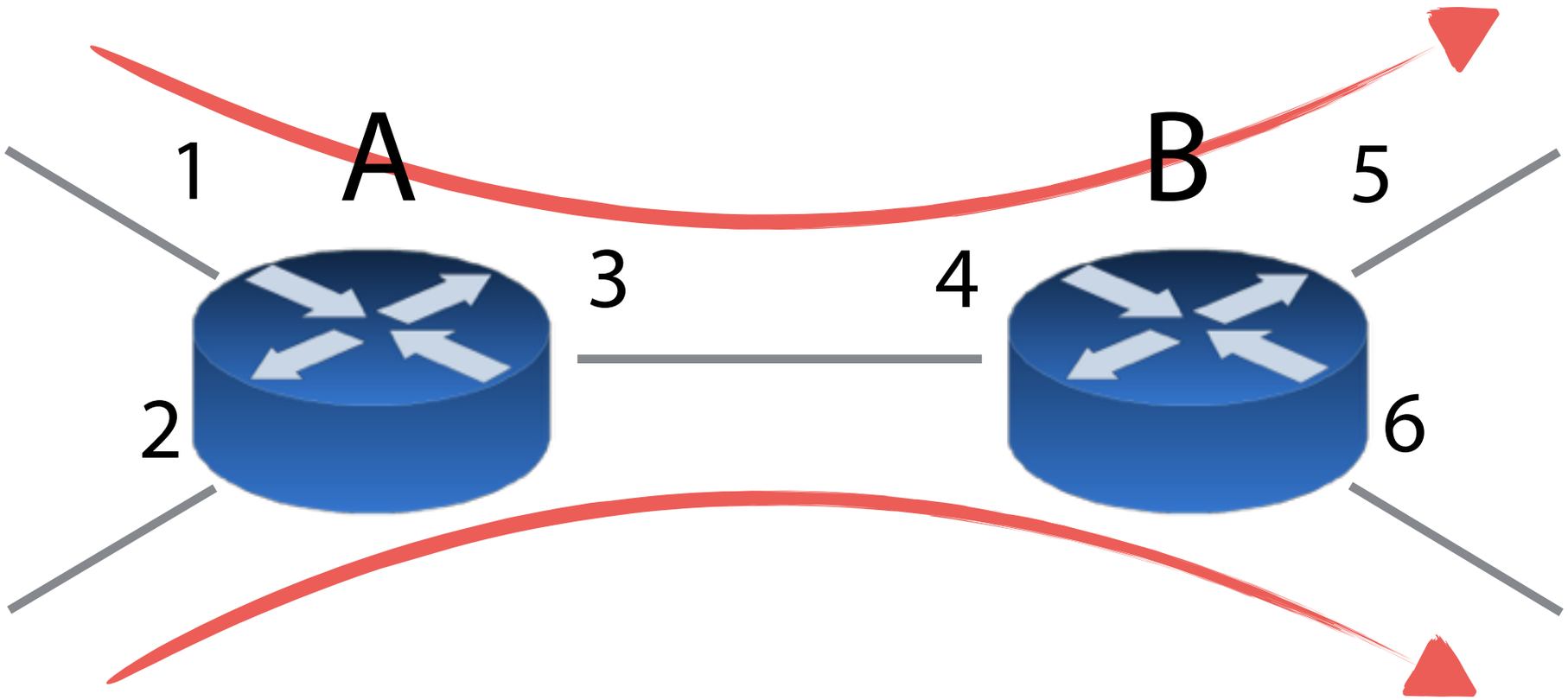
Local NetKAT Program



pol_A

pol_B

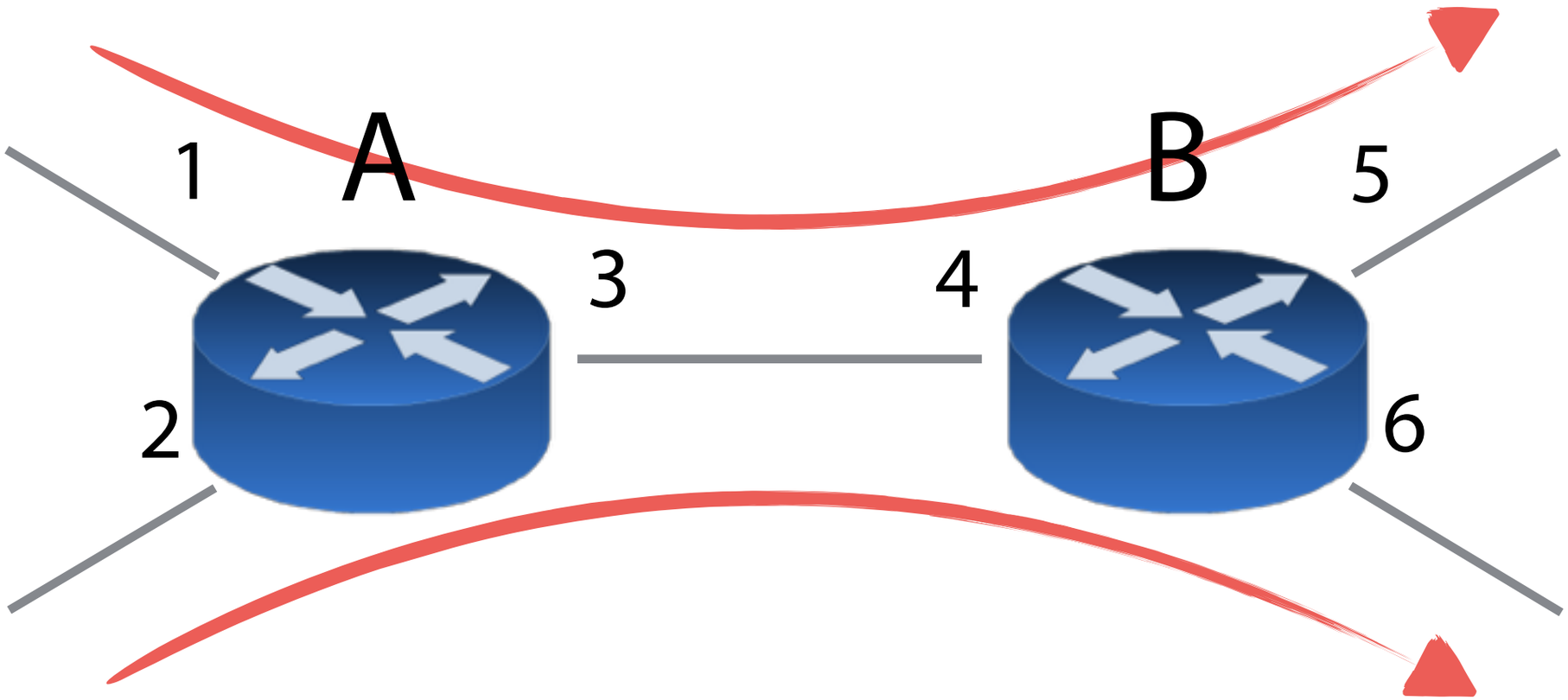
Local NetKAT Program



port:=3

???

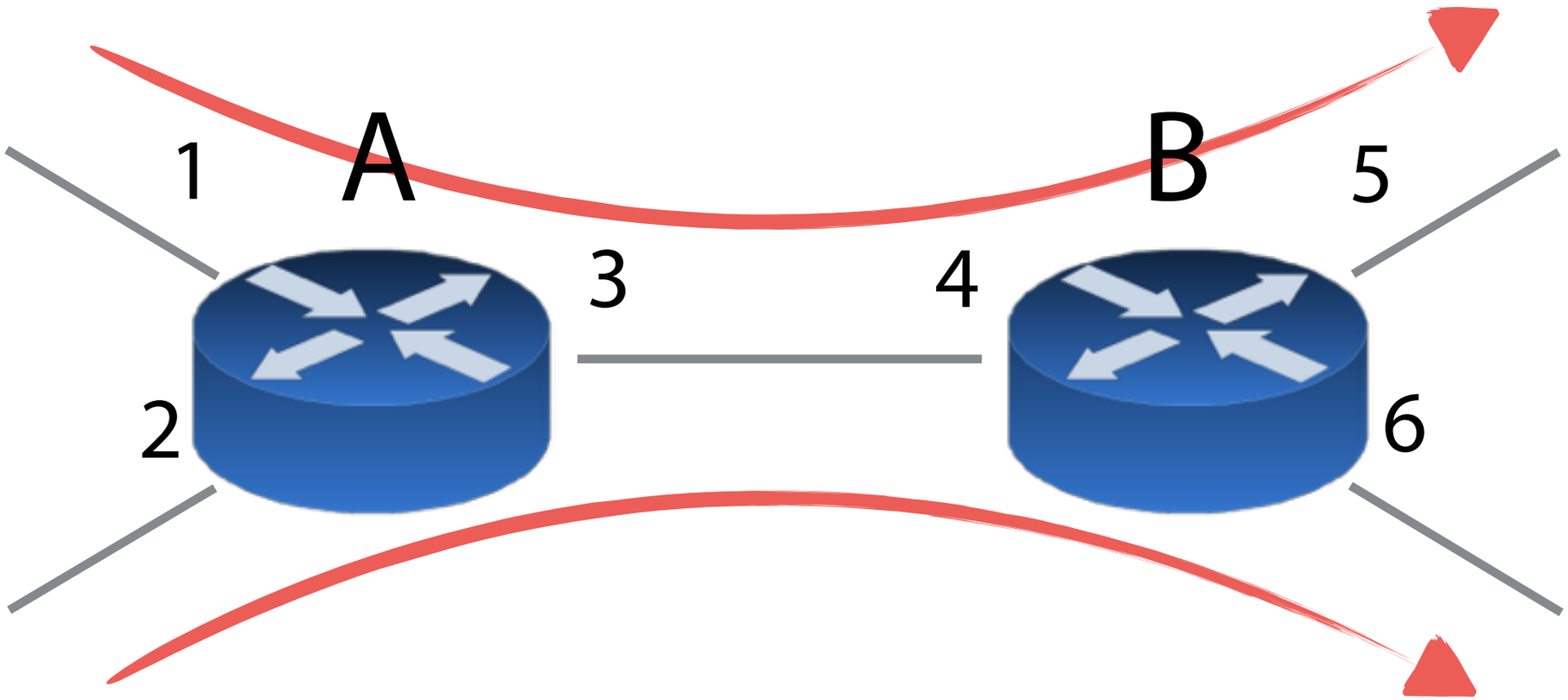
Local NetKAT Program




```
port=1; tag:=1; port:=3
        +
port=2; tag:=2; port:=3
```

???

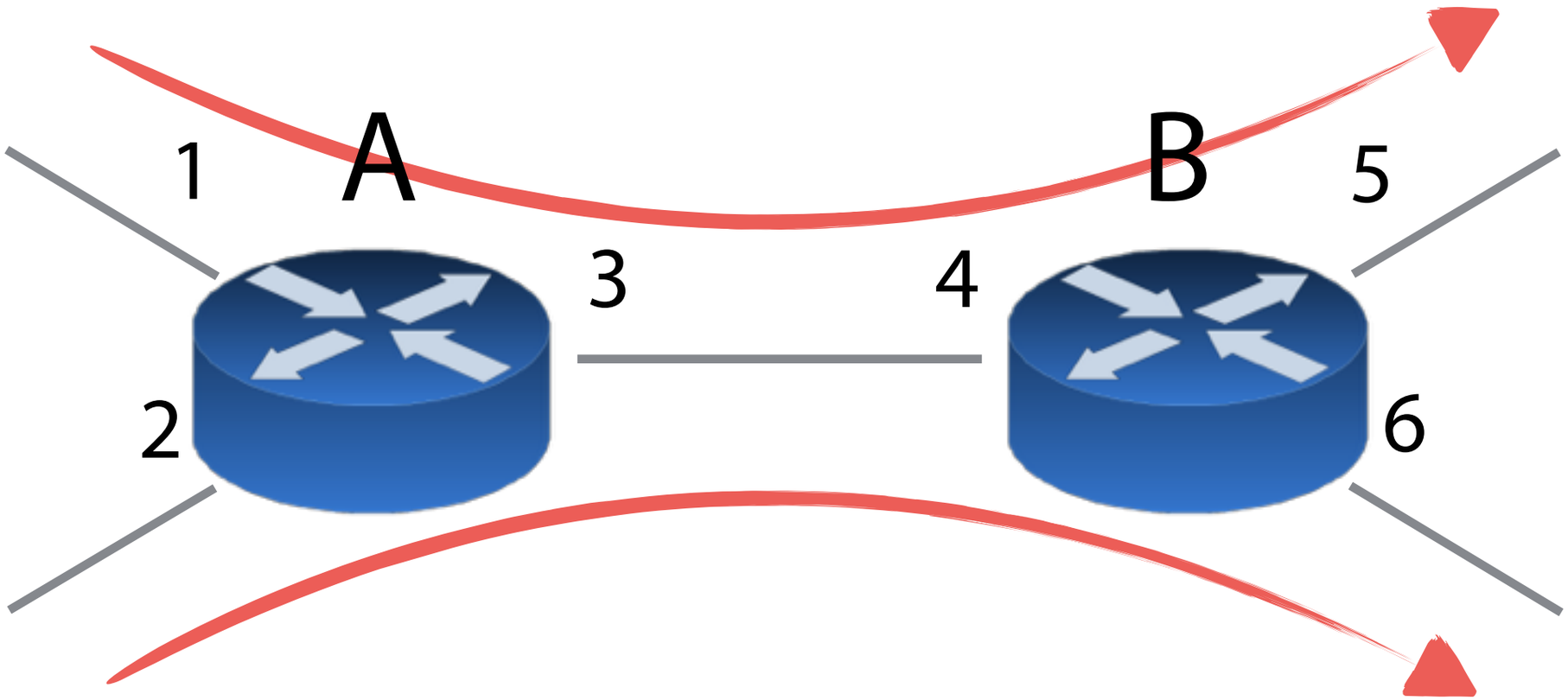
Local NetKAT Program



port=1; tag:=1; port:=3
+
port=2; tag:=2; port:=3

tag=1; port:=5
+
tag=2; port:=6

Local NetKAT Program

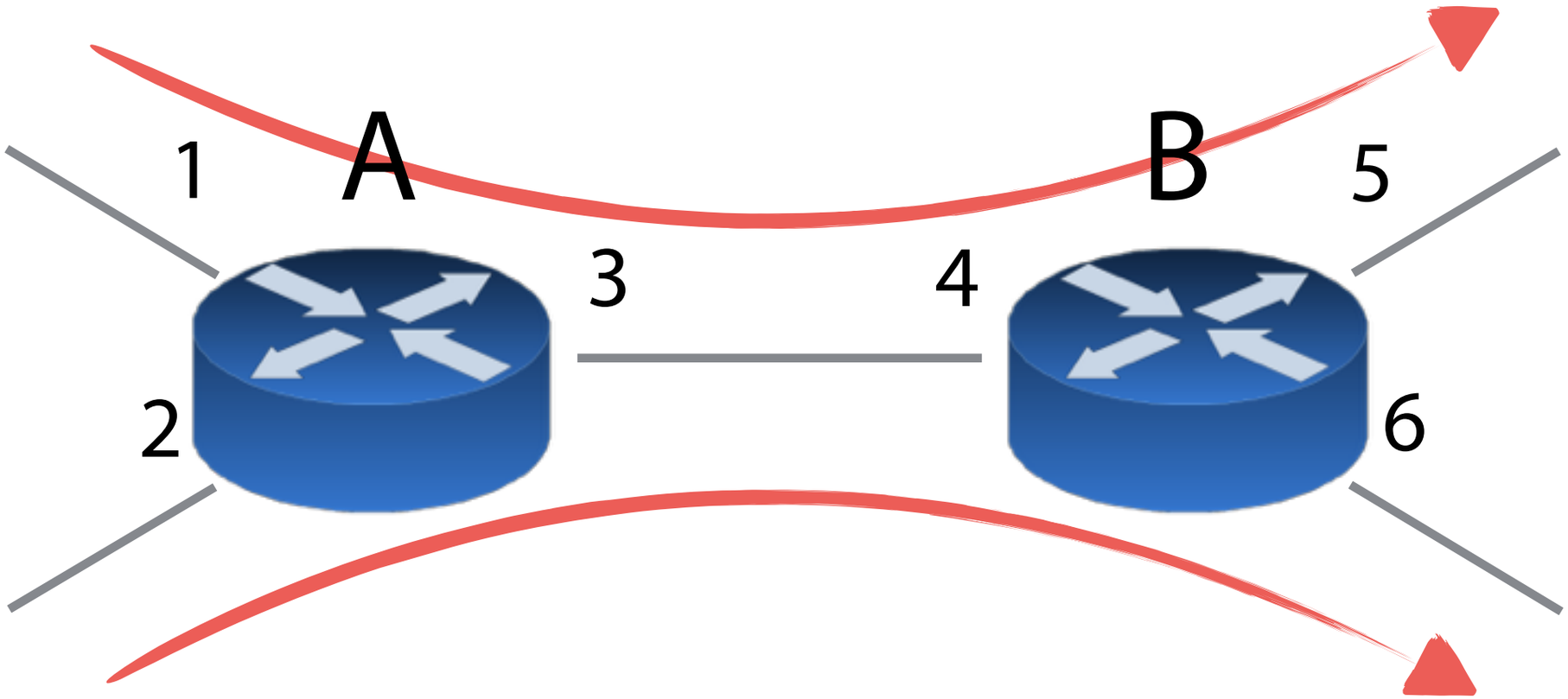


port=1; tag:=1; port:=3
+
port=2; tag:=2; port:=3

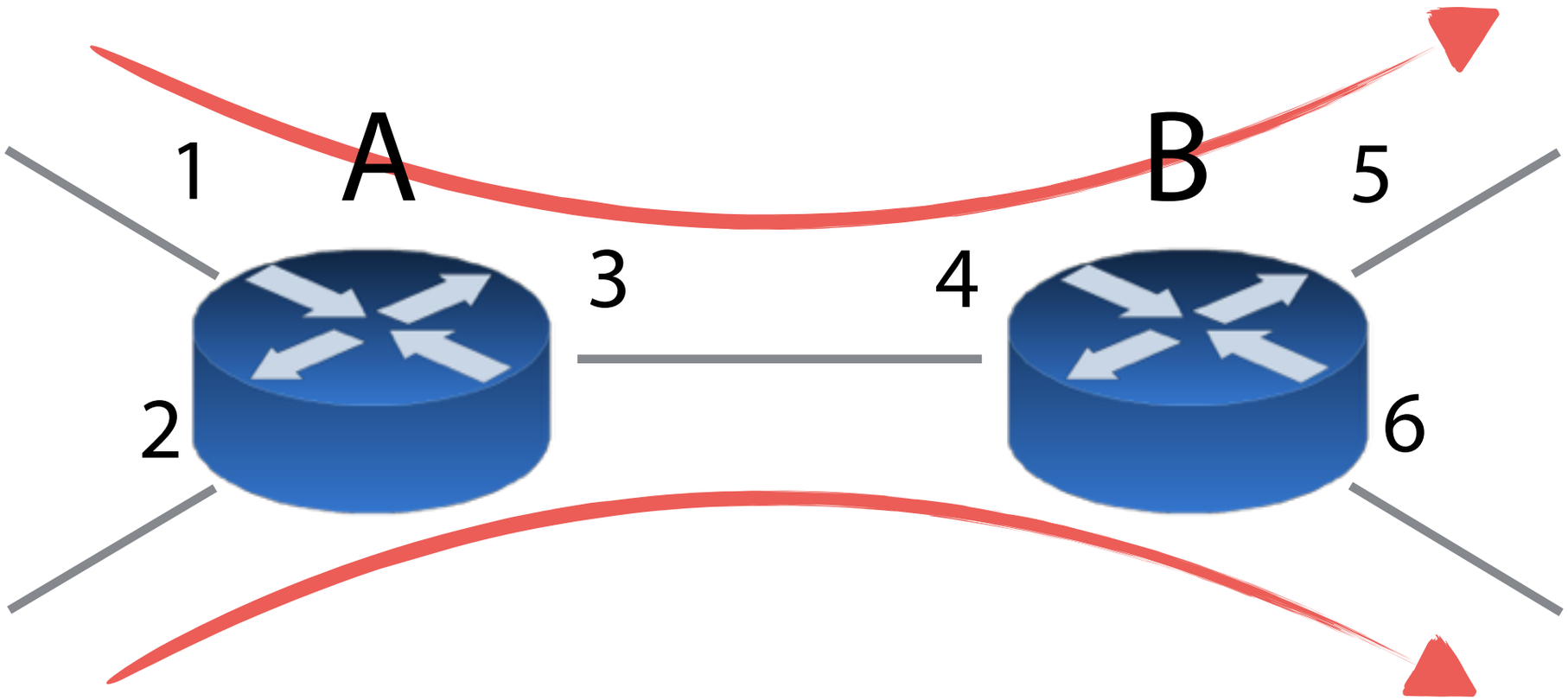
tag=1; port:=5
+
tag=2; port:=6

tedious for programmers… difficult to get right!

Global NetKAT Program

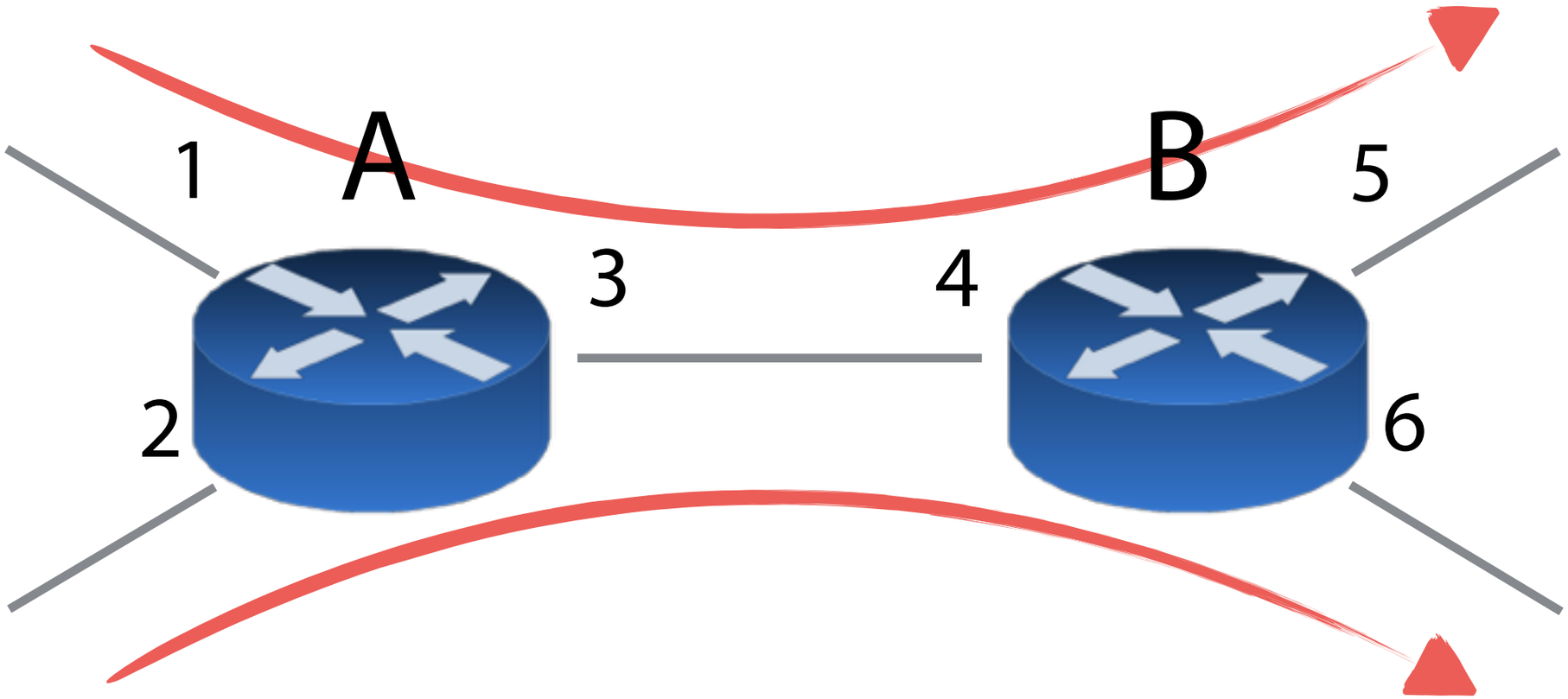


Global NetKAT Program



port=1; **A→B;** port**:=**5
+
port=2; **A→B;** port**:=**6

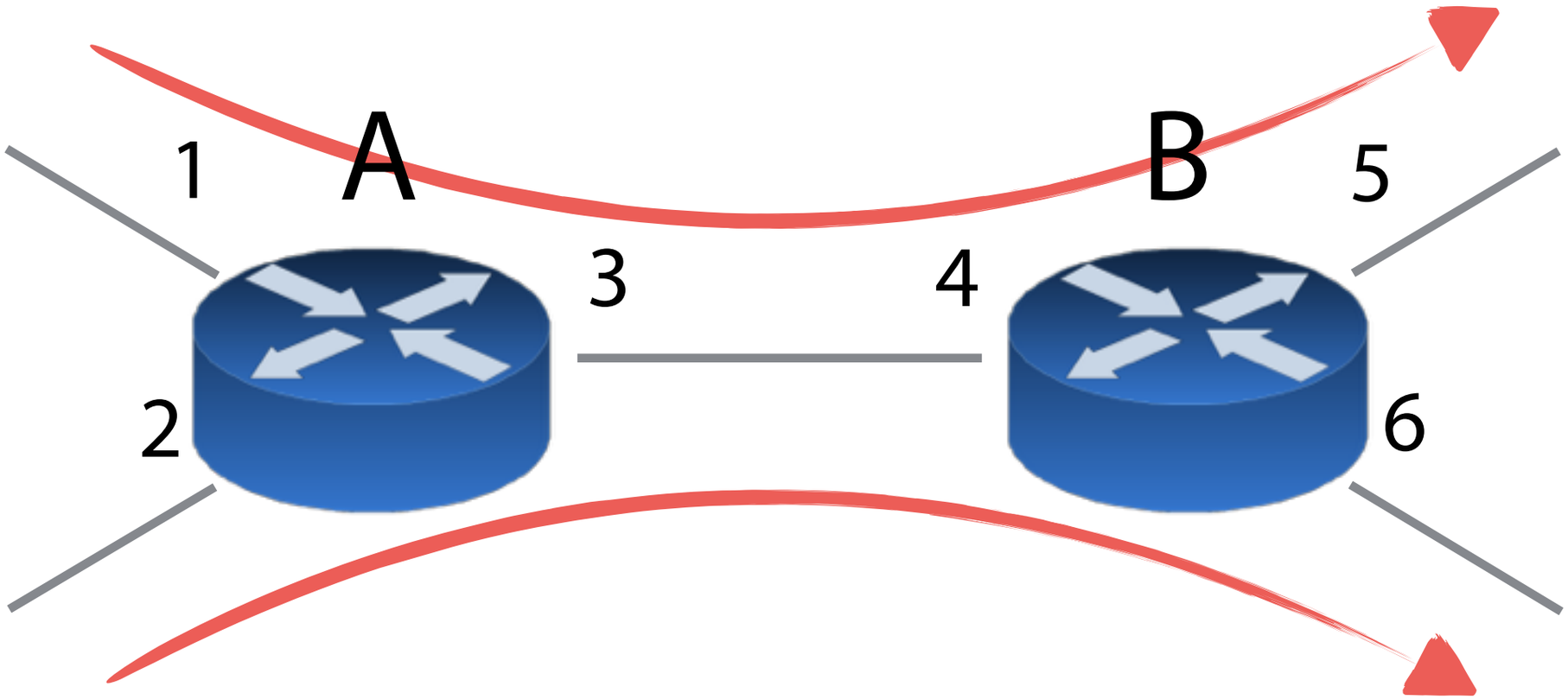
Global NetKAT Program



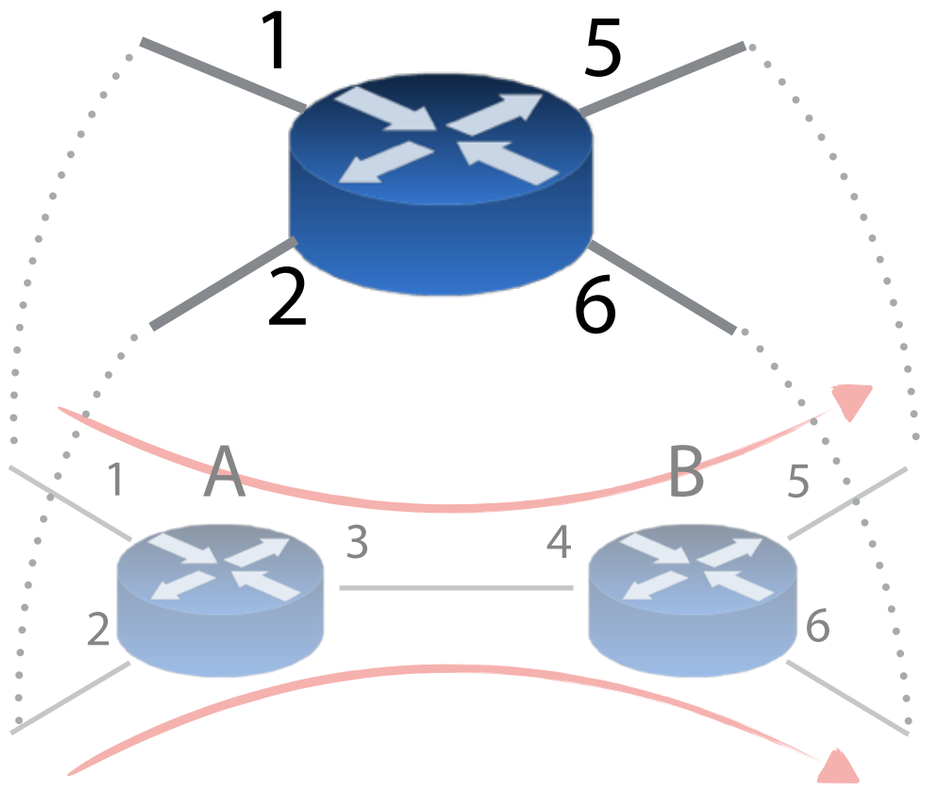
port=1; **A→B;** port:=5
+
port=2; **A→B;** port:=6

simple and elegant!

Virtual NetKAT Program

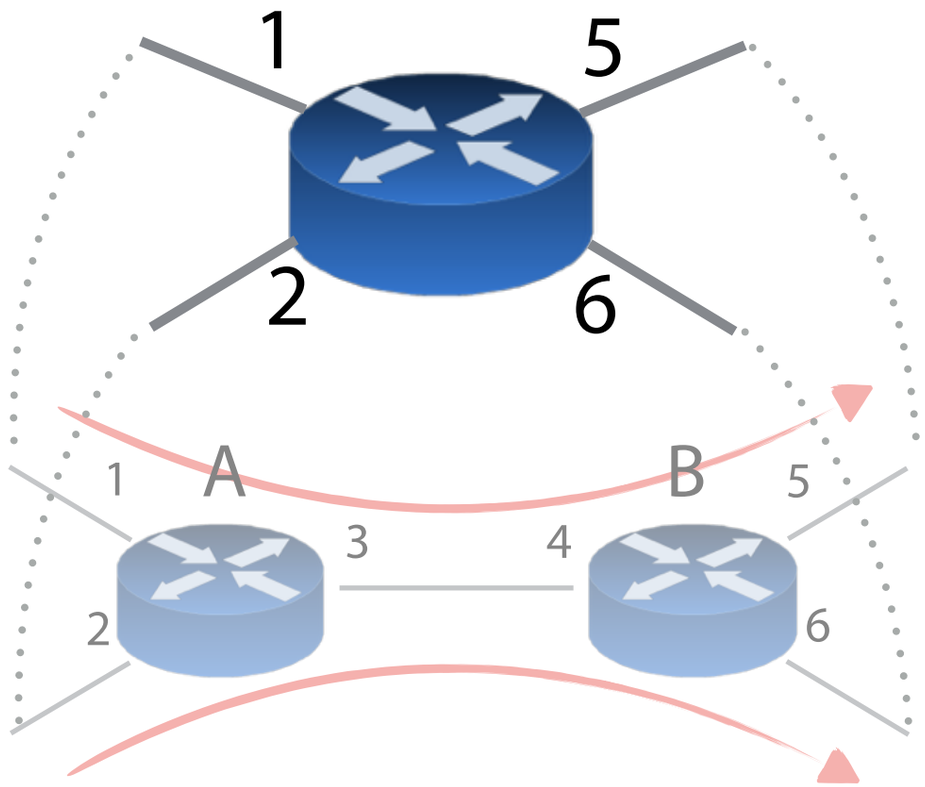


Virtual NetKAT Program



virtual "big switch"

Virtual NetKAT Program



virtual "big switch"

```
port=1; port:=5
  +
port=2; port:=6
```

even simpler!

Local Compilation

Input: local program

Output: collection of flow tables, one per switch

Challenges: efficiency and size of generated tables

Traditional Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

Traditional Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

Pattern	Actions
src=10.0.0.1	Fwd 1
src=10.0.0.2	Fwd 2
*	Controller

Pattern	Actions
tcpSrc=22	Controller
tcpDst=22	Controller
*	Drop

Traditional Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

Pattern	Actions
src=10.0.0.1	Fwd 1
src=10.0.0.2	Fwd 2
*	Controller

+

Pattern	Actions
tcpSrc=22	Controller
tcpDst=22	Controller
*	Drop

Traditional Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

Pattern	Actions
src=10.0.0.1	Fwd 1
src=10.0.0.2	Fwd 2
*	Controller

+

Pattern	Actions
tcpSrc=22	Controller
tcpDst=22	Controller
*	Drop

Inefficient!

Tables are a hardware abstraction,
not an efficient data structure!!

Our Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

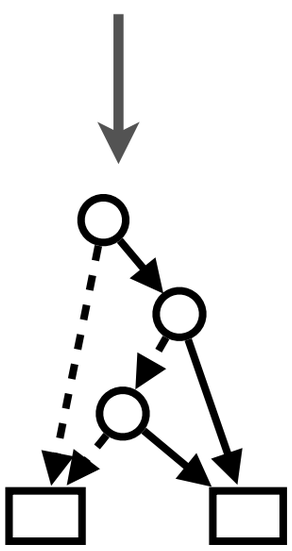
```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

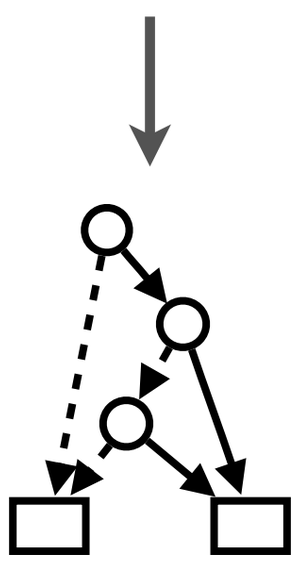
Our Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```



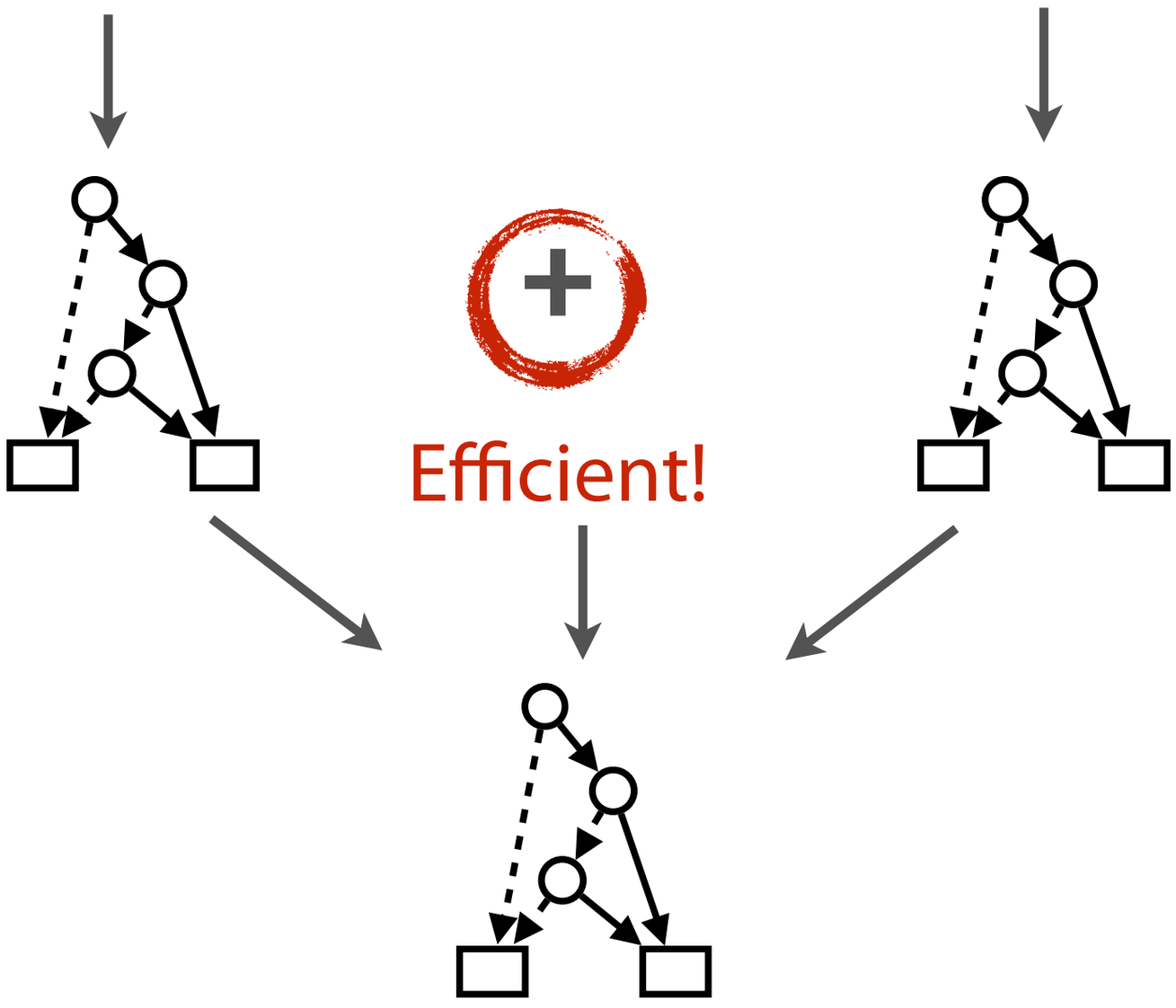


Our Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

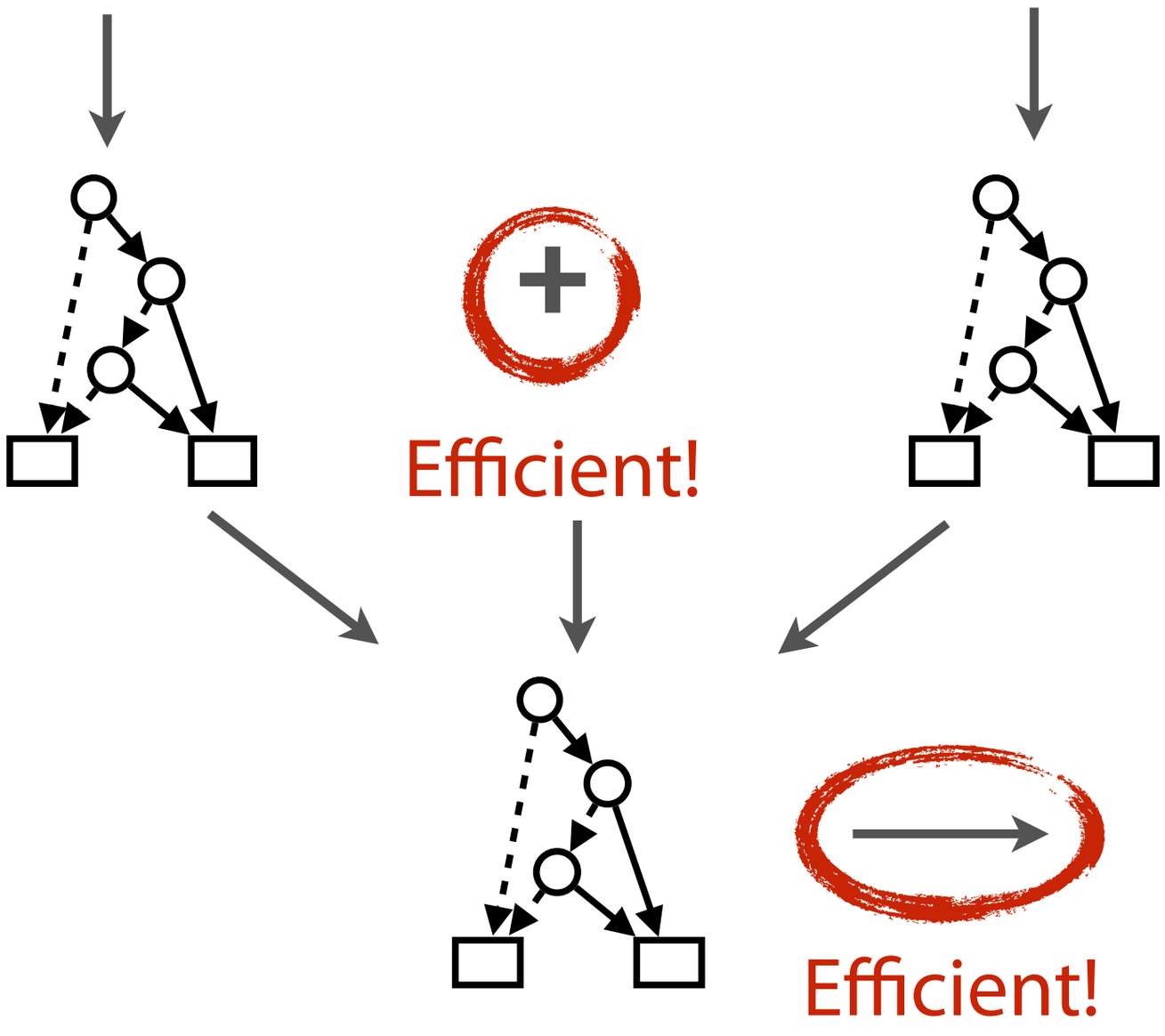


Our Approach

```
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
   port := learn
```

+

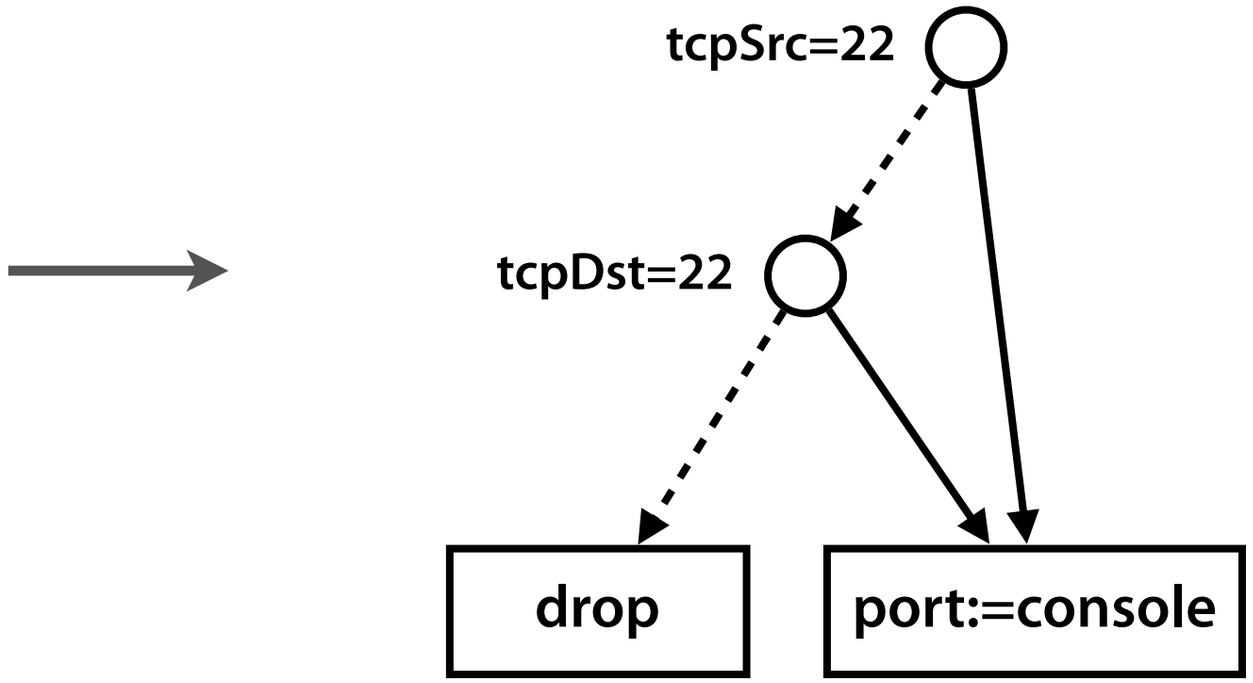
```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```



Pattern	Actions
ipDst=10.0.0.1, tcpSrc=22	Forward 1, Controller
ipDst=10.0.0.1, tcpDst=22	Forward 1, Controller
…	

IR: Forwarding Decision Diagrams

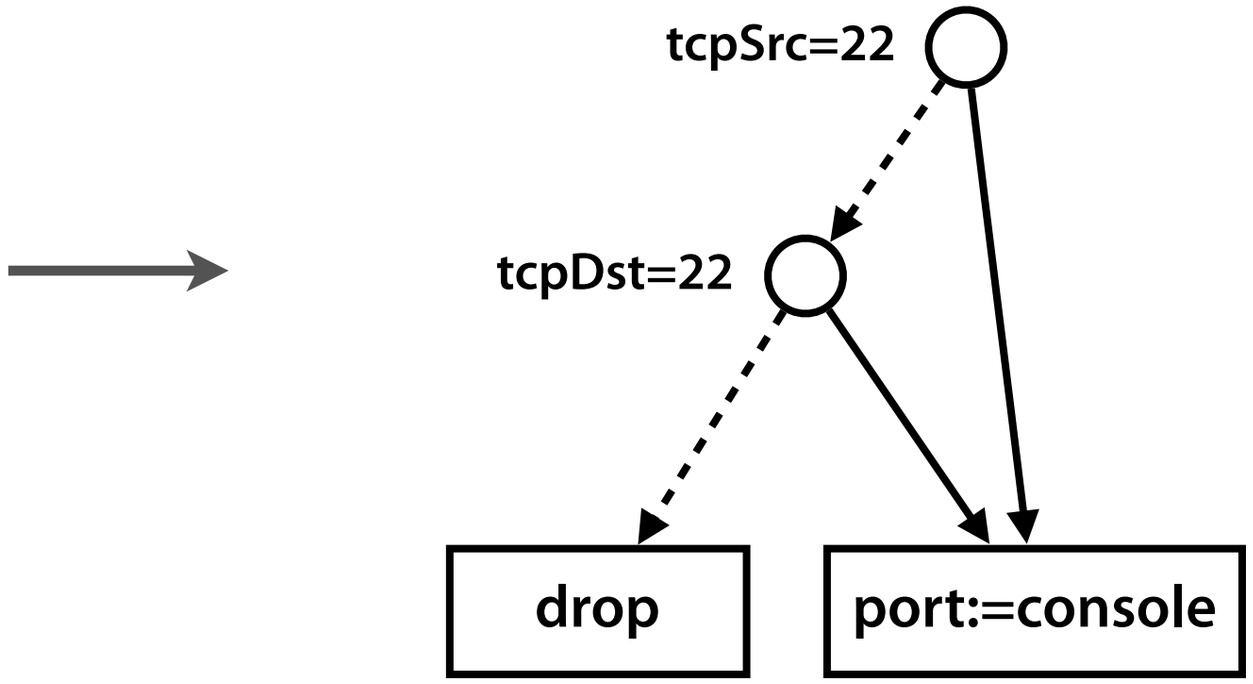
```
if (tcpSrc = 22
  + tcpDst = 22)
then
  port := console
else
  drop
```



Inspired by Binary Decision Diagrams

IR: Forwarding Decision Diagrams

```
if (tcpSrc = 22
  + tcpDst = 22)
then
  port := console
else
  drop
```

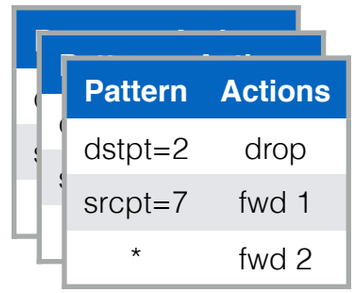


Inspired by Binary Decision Diagrams

NetKAT operators (**+**, **;** , *****, **!**) can be implemented efficiently on FDDs using standard BDD techniques

Global Compilation

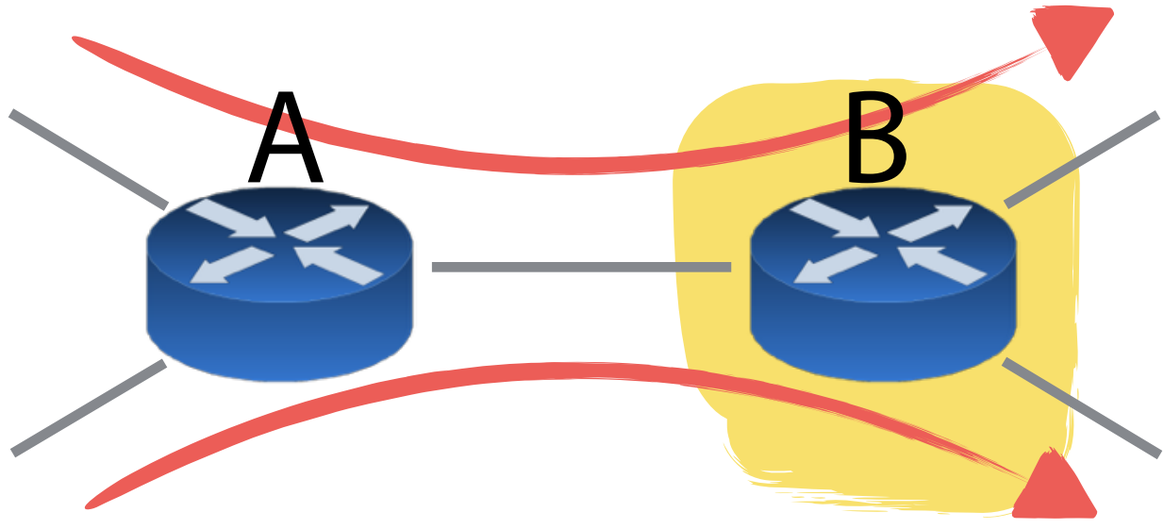
Virtual Compil → **Global** Compiler → Loca Compil



Input: NetKAT program (with links)

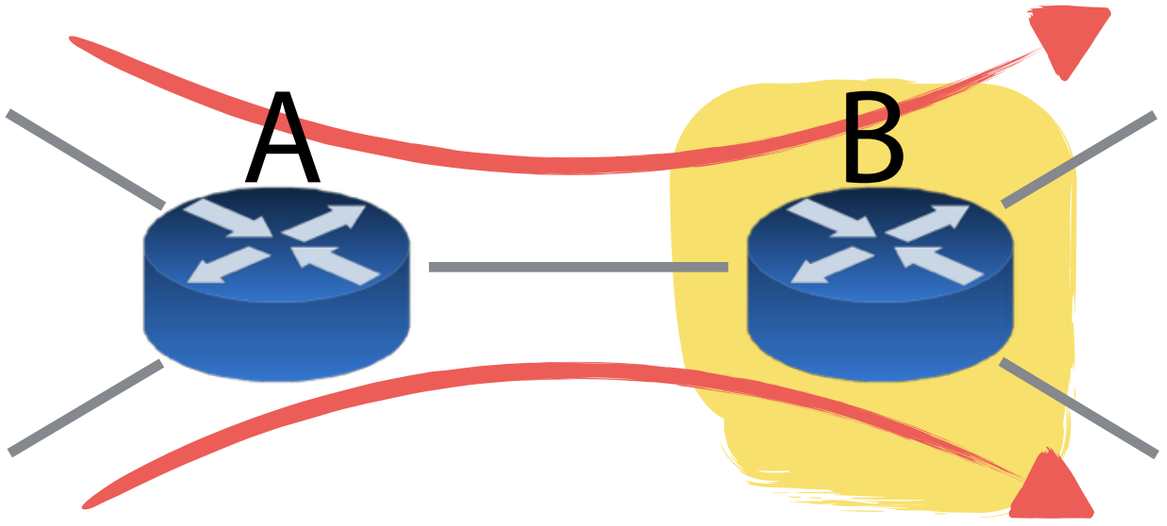
Output: equivalent local program (without links)

Main Challenges



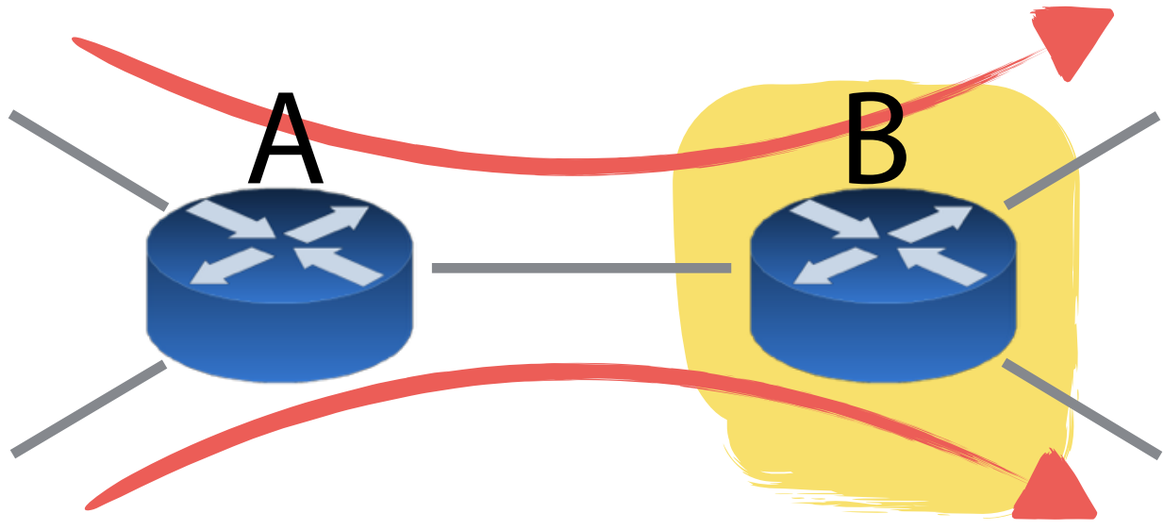
Main Challenges

1. Adding Extra State "Tagging"



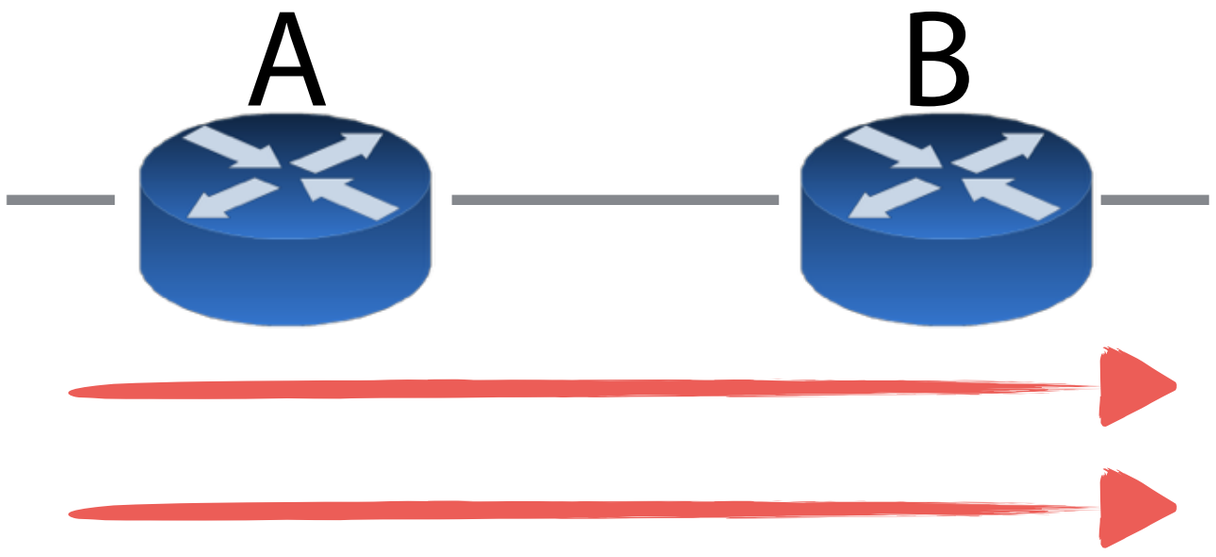
Main Challenges

1. Adding Extra State "Tagging"



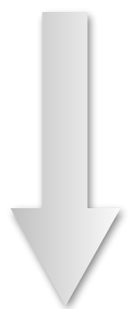
2. Avoiding Duplication

(naive tagging is unsound!)



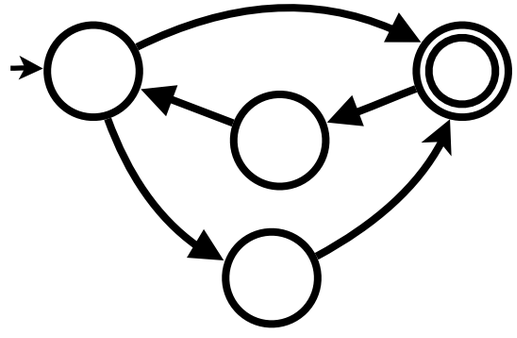
Our Solution

Our Solution

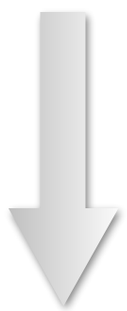


Adding Extra State
= Translation to Automaton

NetKAT NFA

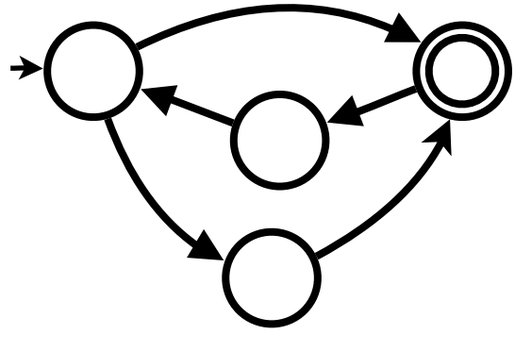


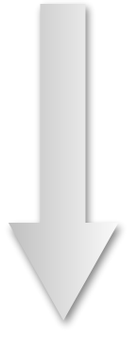
Our Solution



Adding Extra State
= Translation to Automaton

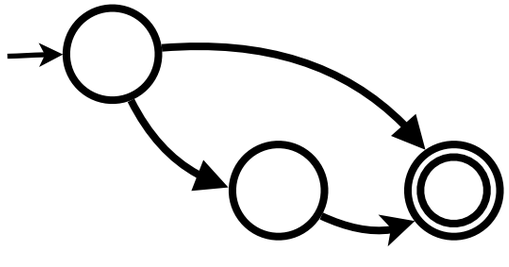
NetKAT NFA





Avoiding Duplication
= Determinization

NetKAT DFA

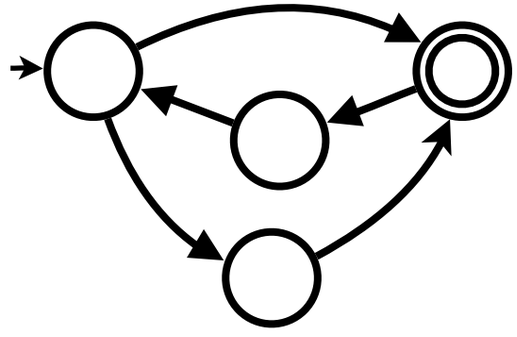


Our Solution

Global Program

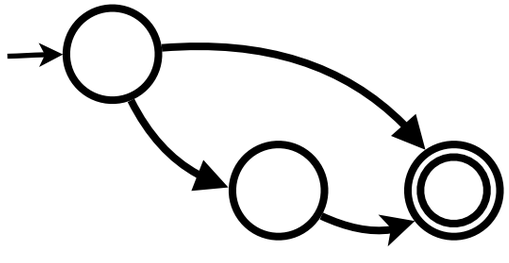
Adding Extra State
= Translation to Automaton

NetKAT NFA



Avoiding Duplication
= Determinization

NetKAT DFA



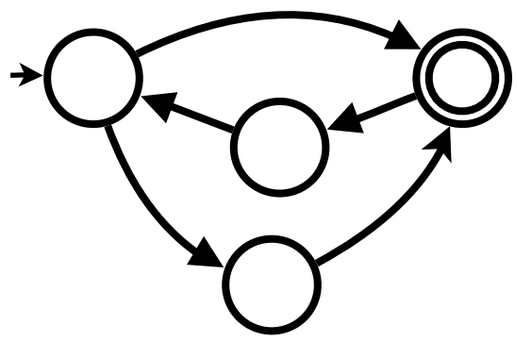
Local Program

Our Solution

Global Program

Adding Extra State
= Translation to Automaton

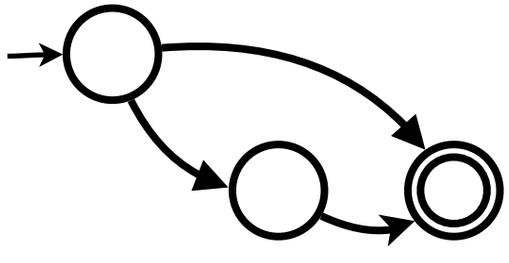
NetKAT NFA



Automaton Minimization
= Tag Elimination

Avoiding Duplication
= Determinization

NetKAT DFA



Local Program

NetKAT Automata [Foster et al, POPL '15]

Transition relation $\delta : Q \rightarrow \text{Packet} \rightarrow P(Q \times \text{Packet})$

NetKAT Automata [Foster et al, POPL '15]

Transition relation $\delta : Q \rightarrow$ Packet $\rightarrow P(Q \times \text{Packet})$

"Alphabet size": |Packet x Packet|

NetKAT Automata [Foster et al, POPL '15]

Transition relation $\delta : Q \rightarrow \text{Packet} \rightarrow P(Q \times \text{Packet})$

"Alphabet size": $|\text{Packet} \times \text{Packet}|$

Can represent δ **symbolically** using FDDs!

NetKAT Automata [Foster et al, POPL '15]

Transition relation $\delta : Q \rightarrow$ Packet $\rightarrow P(Q \times \text{Packet})$

"Alphabet size": |Packet x Packet|

Can represent δ **symbolically** using FDDs!

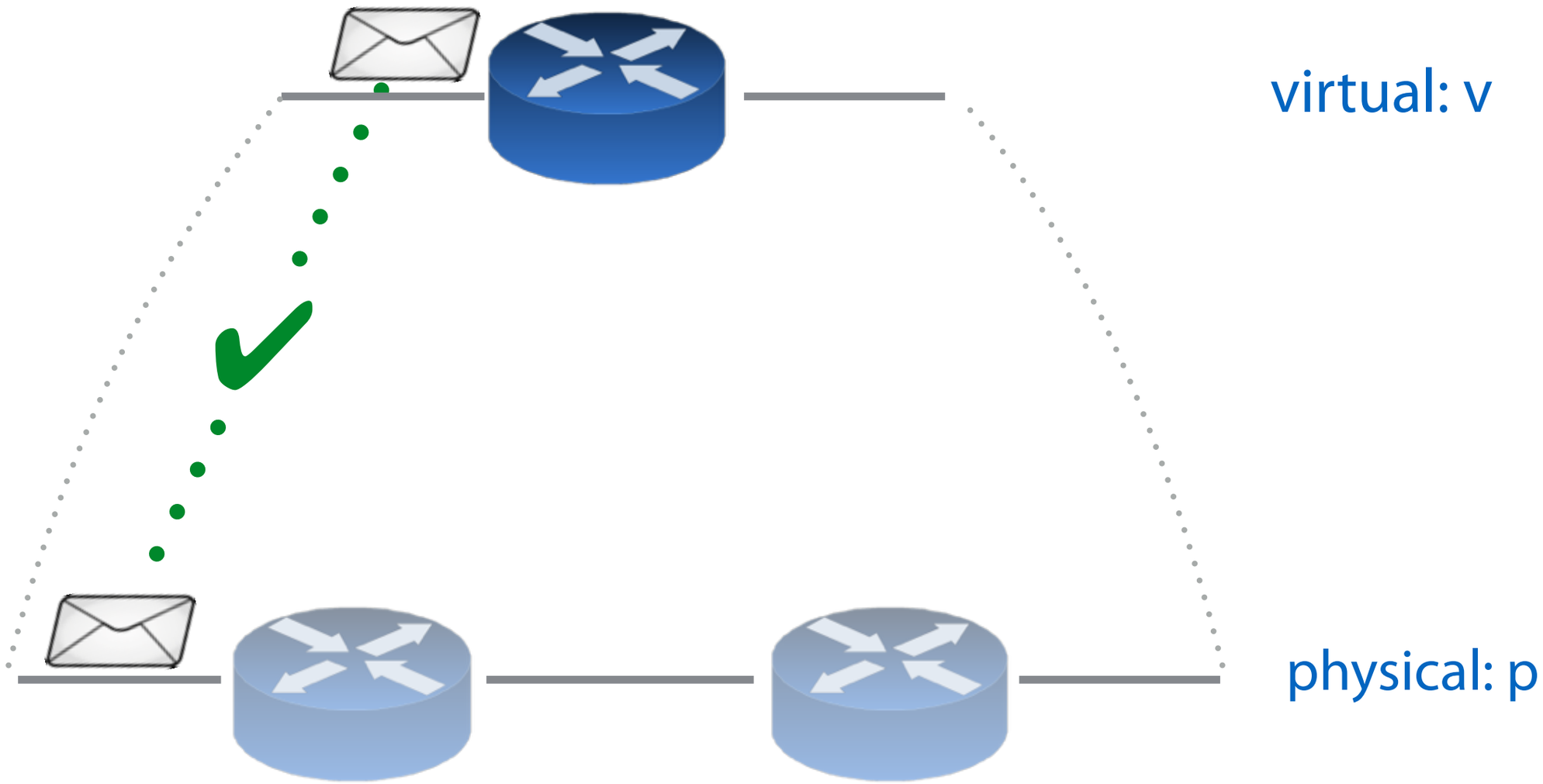
Automata construction:
Antimirov partial derivatives & **Position Automata**

Virtual Compilation

Input: program written against virtual topology

Output: global program that simulates virtual behavior

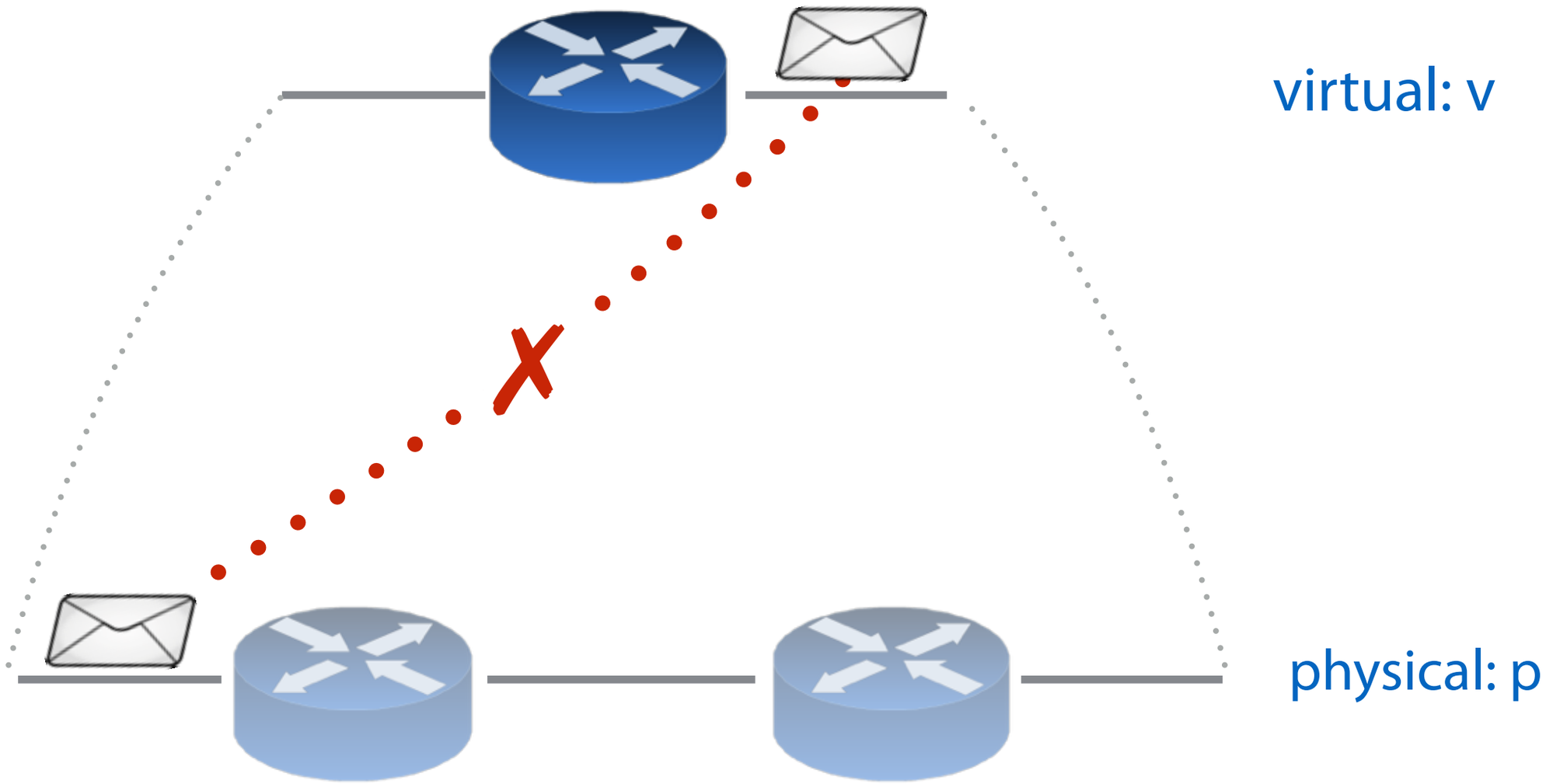
Virtualization



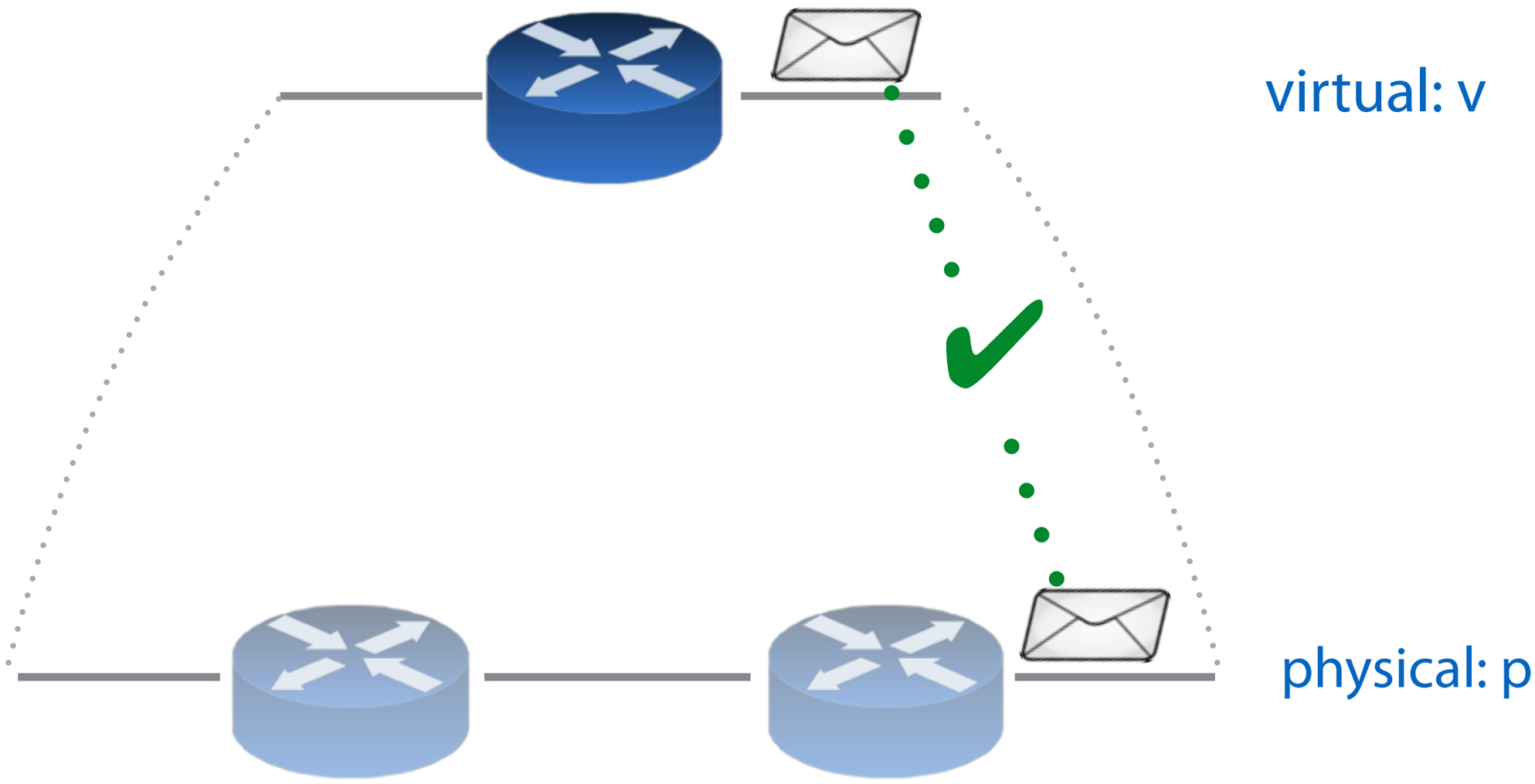
virtual: v

physical: p

Virtualization



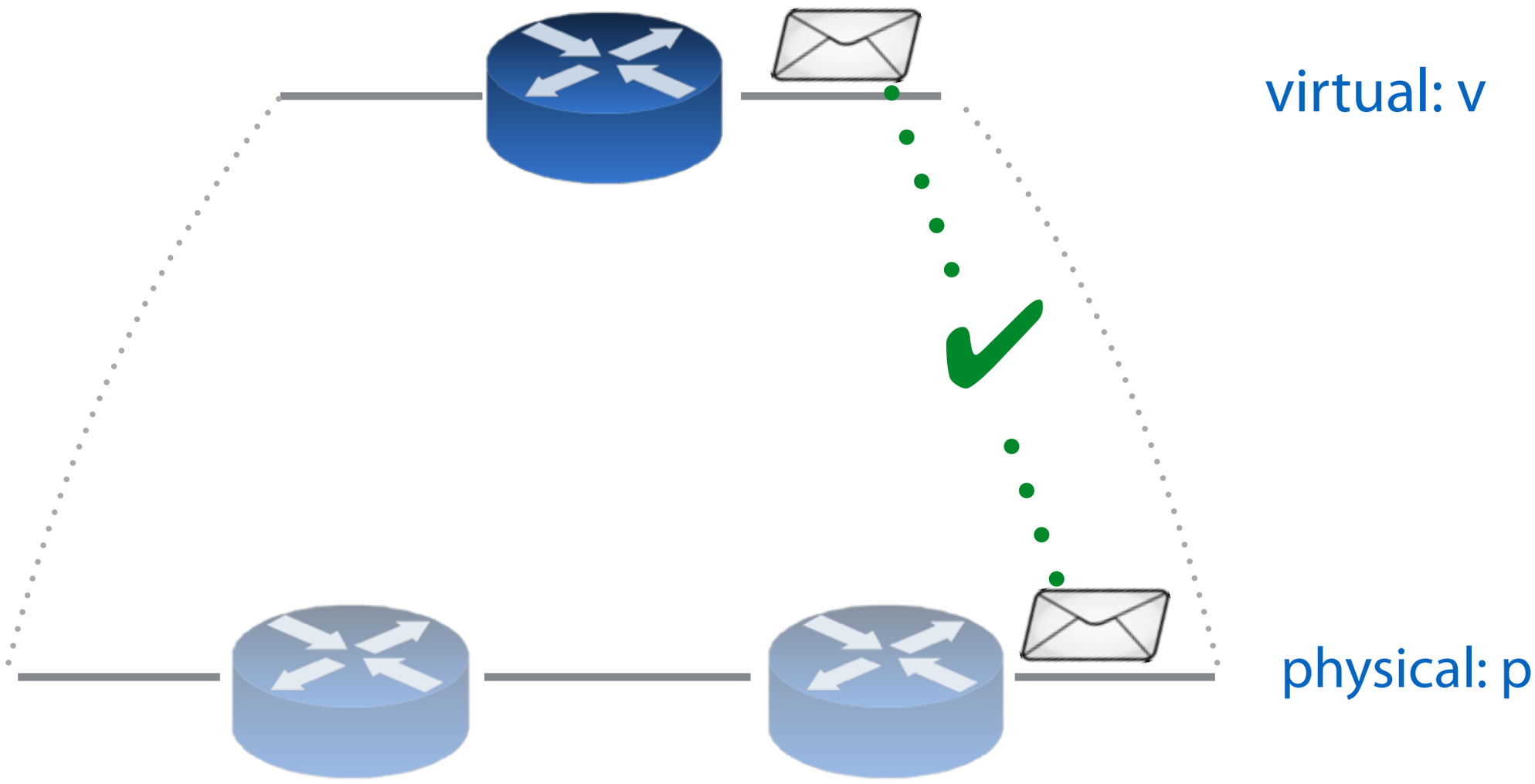
Virtualization



virtual: v

physical: p

Virtualization

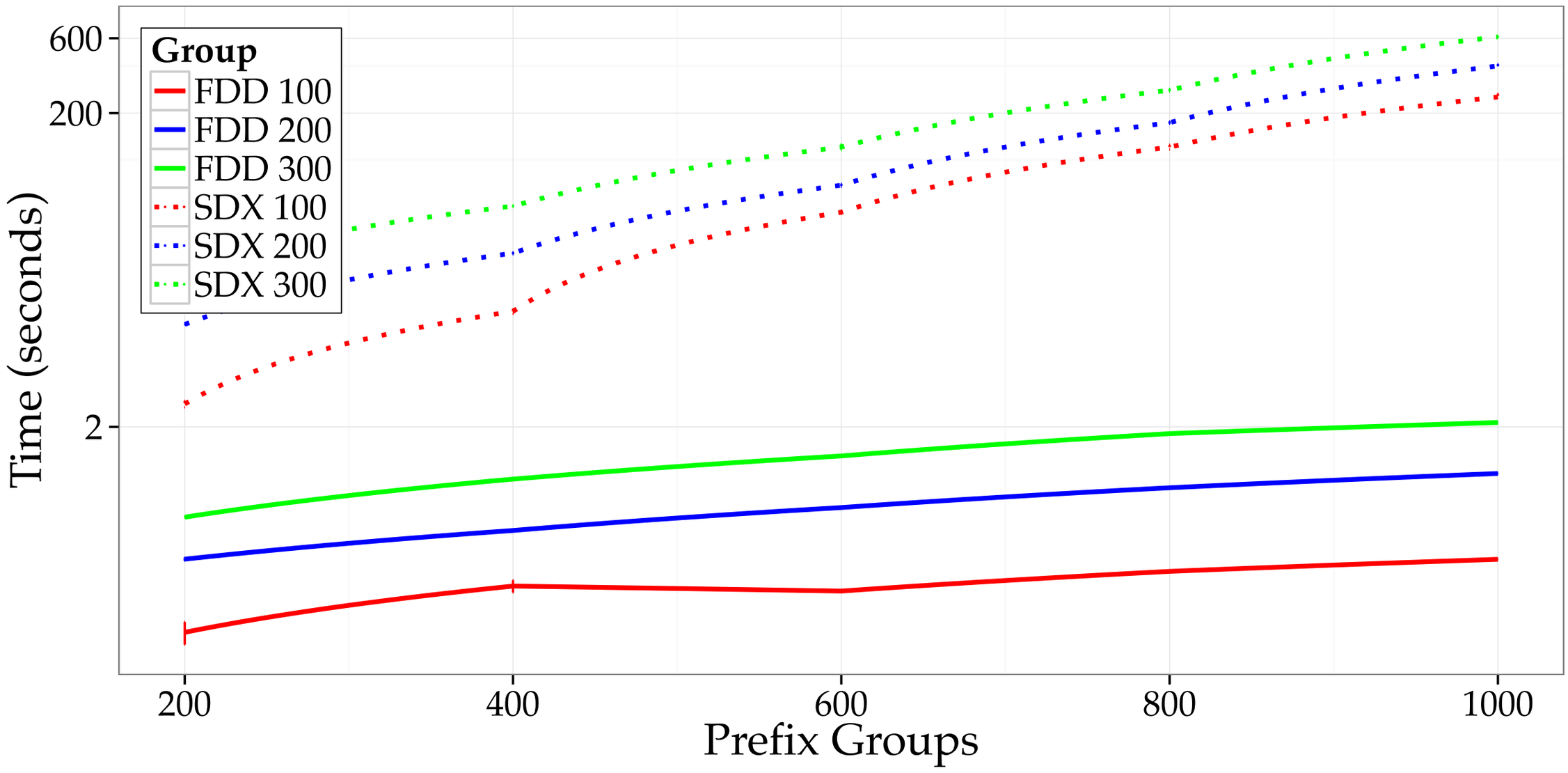


Observation: can formulate execution of a virtual program as a two-player game

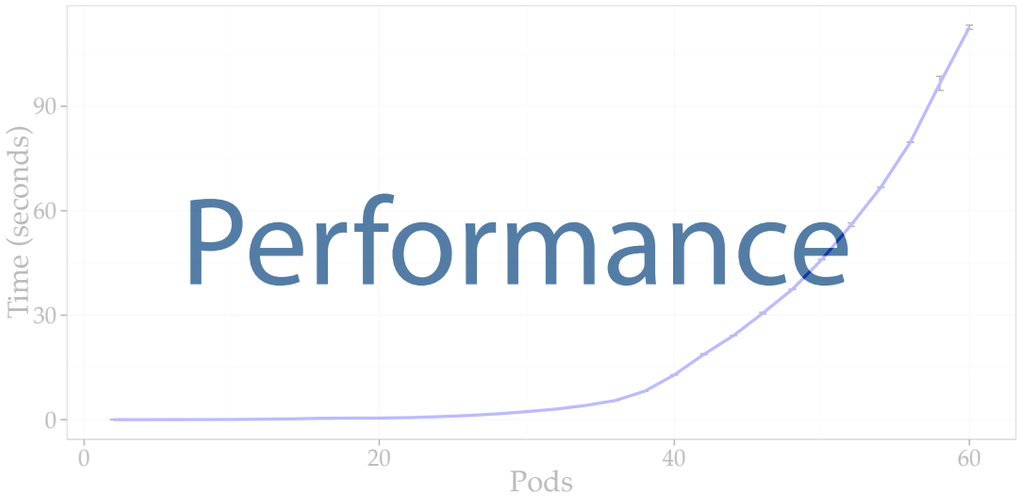
Compiler: synthesizes physical program p that encodes a winning strategy to all instances of that game

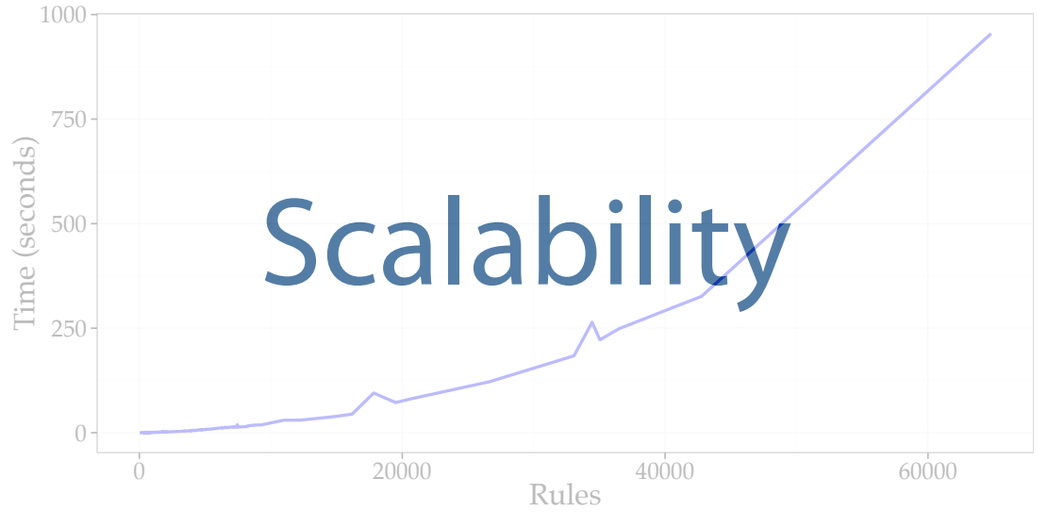
Evaluation

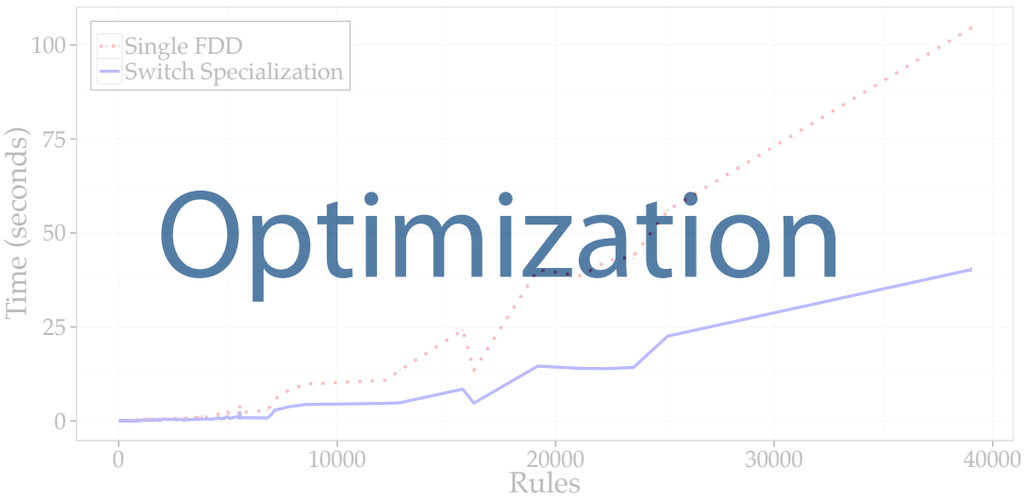
Local Compiler vs State of the Art

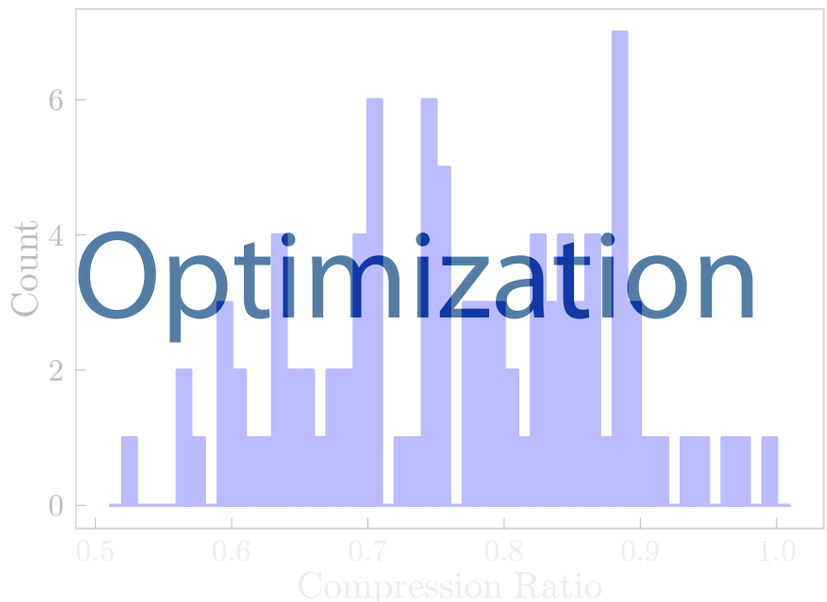


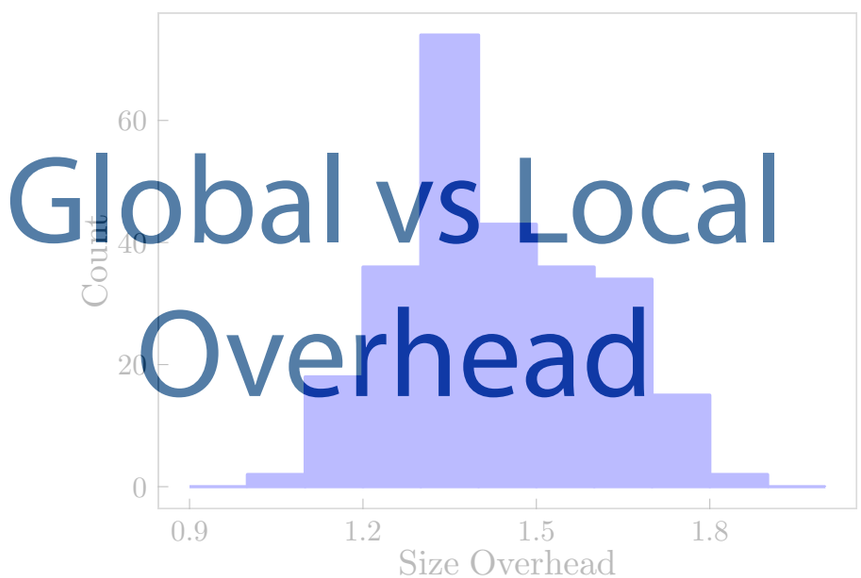
about 100x speedup

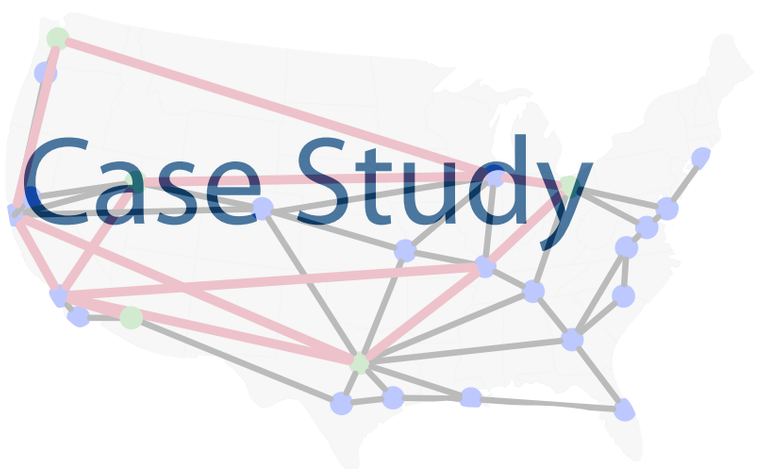

Performance
Time (seconds)
0
30
60
90
0
20
40
60
Pods


Scalability
Time (seconds)
0
250
500
750
1000
0
20000
40000
60000
Rules

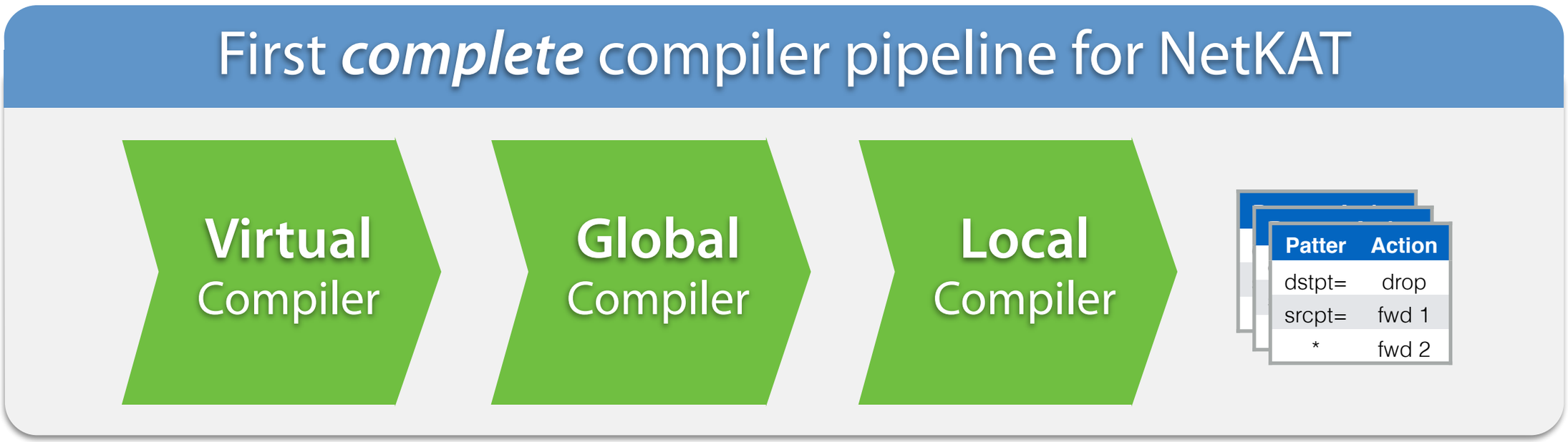

Optimization
Single FDD
Switch Specialization
Time (seconds)
0
25
50
75
100
0
10000
20000
30000
40000
Rules


Optimization
Count
0
2
4
6
0.5
0.6
0.7
0.8
0.9
1.0
Compression Ratio


Global vs Local
Overhead
Count
0
20
40
60
0.9
1.2
1.5
1.8
Size Overhead


Case Study

Conclusion



Fast, **Flexible**, and **Fully implemented** in OCaml:
http://github.com/frenetic-lang/frenetic/

Go ahead and use it!
(others are using it already)

FUJITSU SDX frenetic ≫ Pyretic

Thank you!

Papers, code, etc: http://frenetic-lang.org/