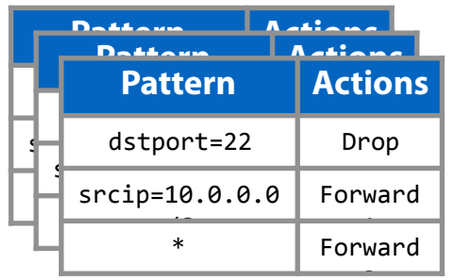
Efficient Programming Abstractions for SDN

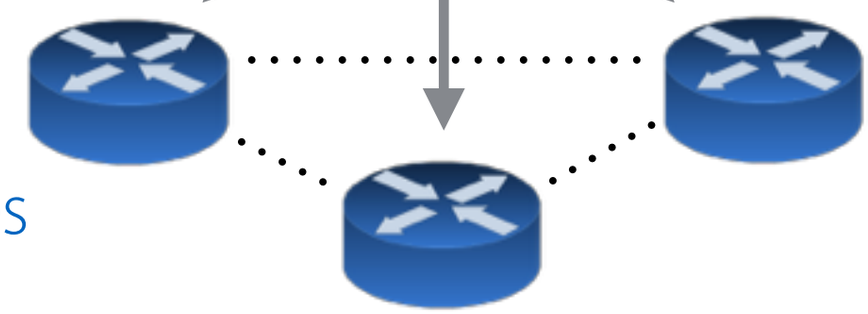
Steffen Smolka

Networks are becoming programmable

rule tables



SDN switches



machine abstraction

open interface (OpenFlow / P4)

Networks are becoming programmable

high-level program

Fire-wall ; Route

Compiler

This Talk

rule tables

Pattern	Actions
dstport=22	Drop
srcip=10.0.0.0	Forward
*	Forward

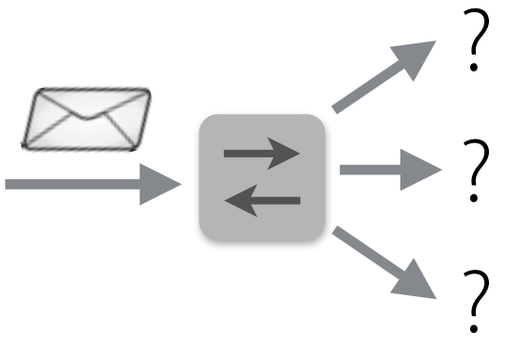
SDN switches

machine abstraction

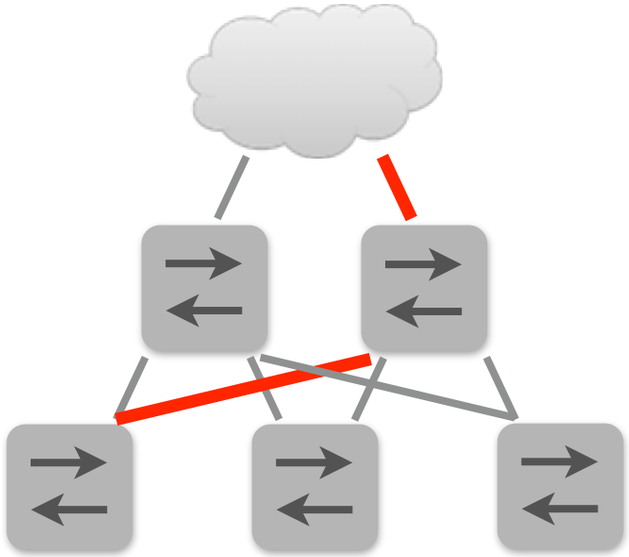
open interface (OpenFlow / P4)

Language Design

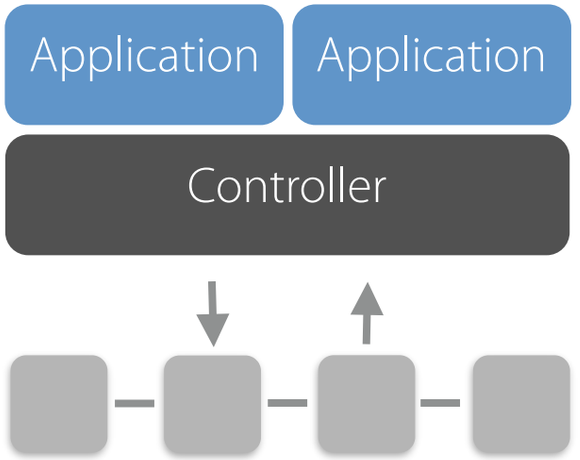
Language Design



Rich Packet Classification



Network-wide Abstractions



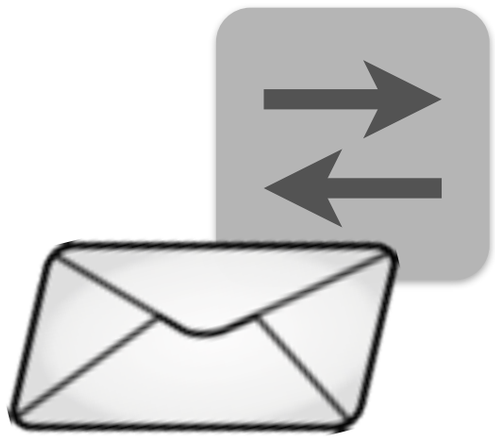
Modular Composition

NetKAT

Model

Packets are records of values.

Programs are functions on packets.



{ **switch** = A,
port = 3,
ethSrc = 8:8:::::8:8,
ethDst = 2:2:::::2:2,
vLan = 8,
ipSrc = 192.168.2.1,
ipDst = 127.0.0.1,
...
}

NetKAT Language

pol ::=

| **false**

| **true**

| field **=** val

| field **:=** val

| pol_1 **+** pol_2

| pol_1 **;** pol_2

| !pol

| pol*

| S→S'

NetKAT Language

pol ::=

| **false**

| **true**

| field **=** val

| field **:=** val

| pol_1 **+** pol_2

| pol_1 **;** pol_2

| !pol

| pol*

| S→S'

Boolean Algebra

NetKAT Language

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

Boolean Algebra

+

Kleene Algebra

"Regular Expressions"

NetKAT Language

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

Boolean Algebra

+

Kleene Algebra
"Regular Expressions"

+

Packet Primitives

NetKAT Language

pol ::=
| **false**
| pol$_1$ **;** pol$_2$
| !pol
| pol*
| S→S'

Boolean Algebra

if p **then** q **else** r ≜ p**;**q **+ !**p**;**r

while p **do** q ≜ p**;**q***;!**p

Packet Primitives

NetKAT Semantics

pol ::=

| **false**

| **true**

| field **=** val

| field **:=** val

| pol_1 **+** pol_2

| pol_1 **;** pol_2

| !pol

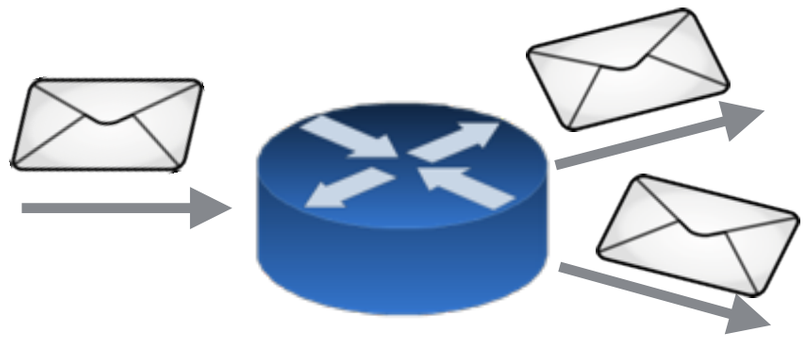
| pol*

| ~~S ⇝ S'~~

NetKAT 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

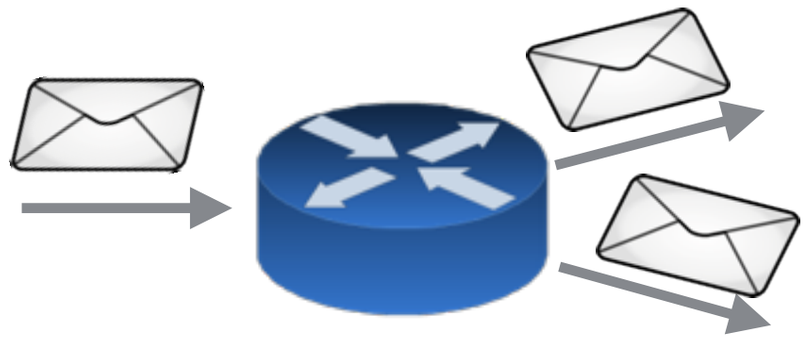


⟦pol⟧ ∈ Packet → Packet Set

NetKAT Semantics

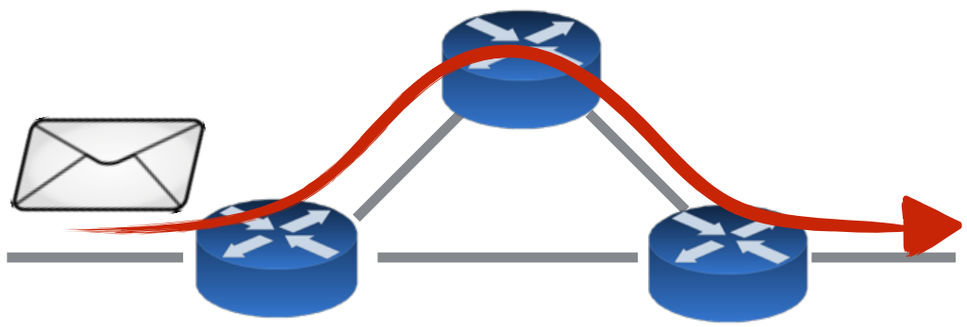
pol ::=
- | **false**
- | **true**
- | field **=** val
- | field **:=** val
- | pol_1 **+** pol_2
- | pol_1 **;** pol_2
- | !pol
- | pol*
- | S→S'

Local NetKAT: input-output behavior of switches



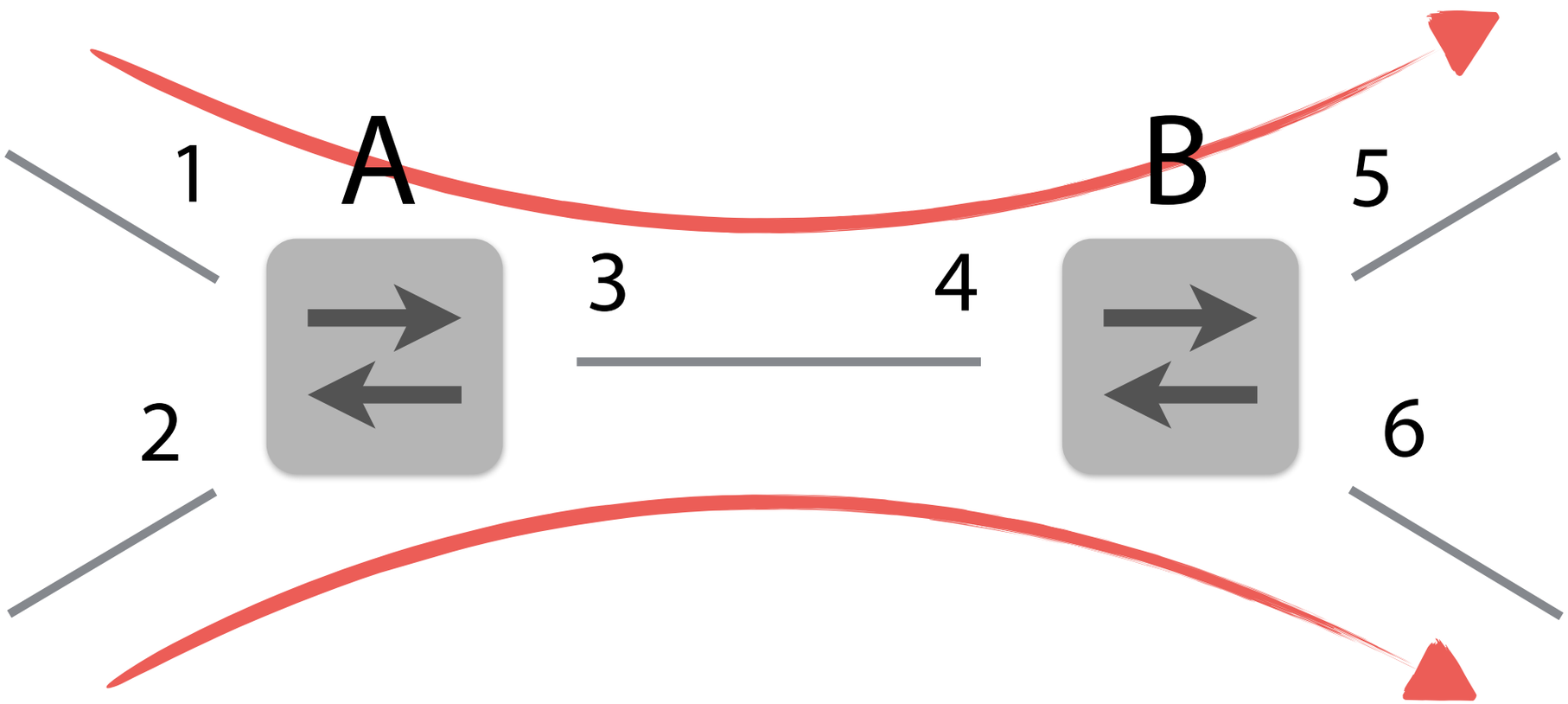
⟦pol⟧ ∈ Packet → Packet Set

Global NetKAT: network-wide behavior

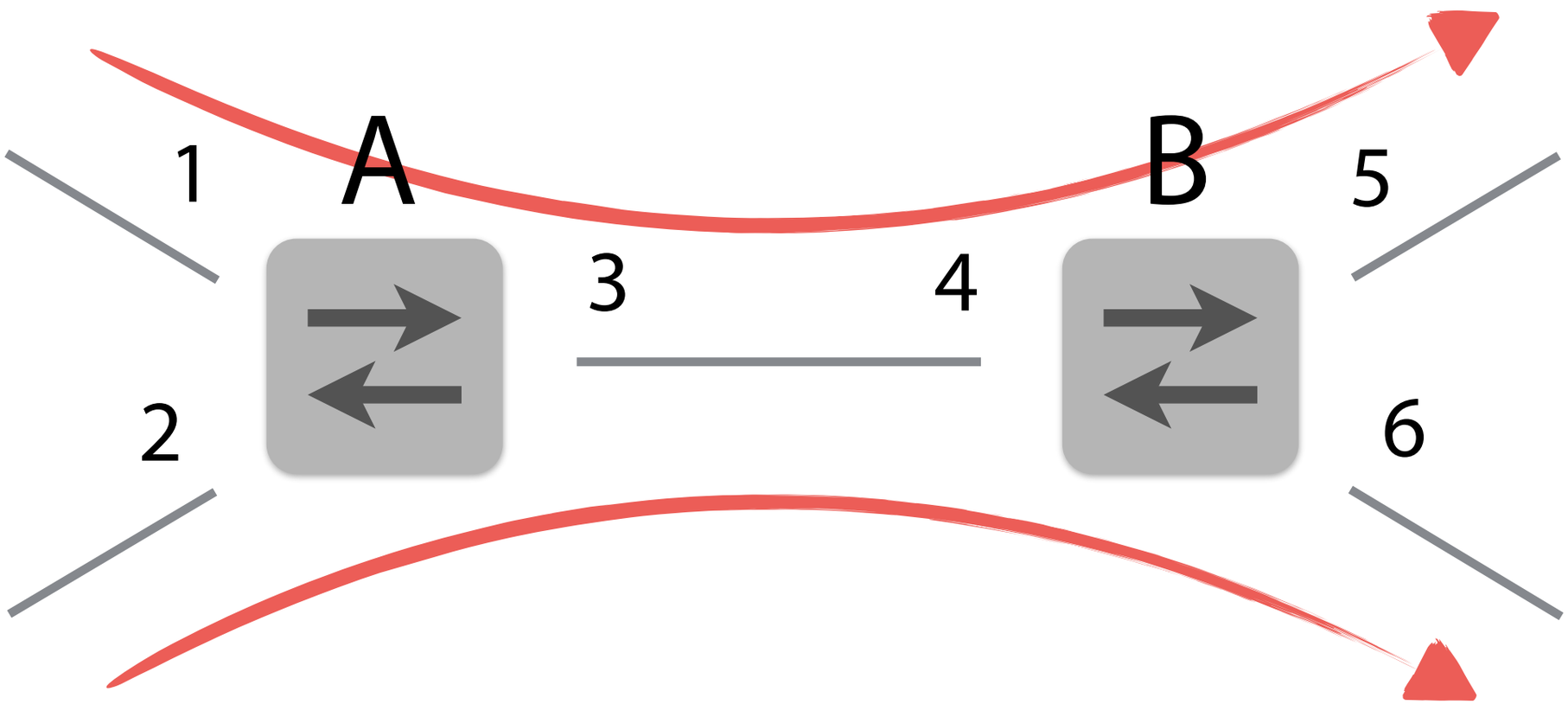


⟦pol⟧ ∈ History → History Set

Example



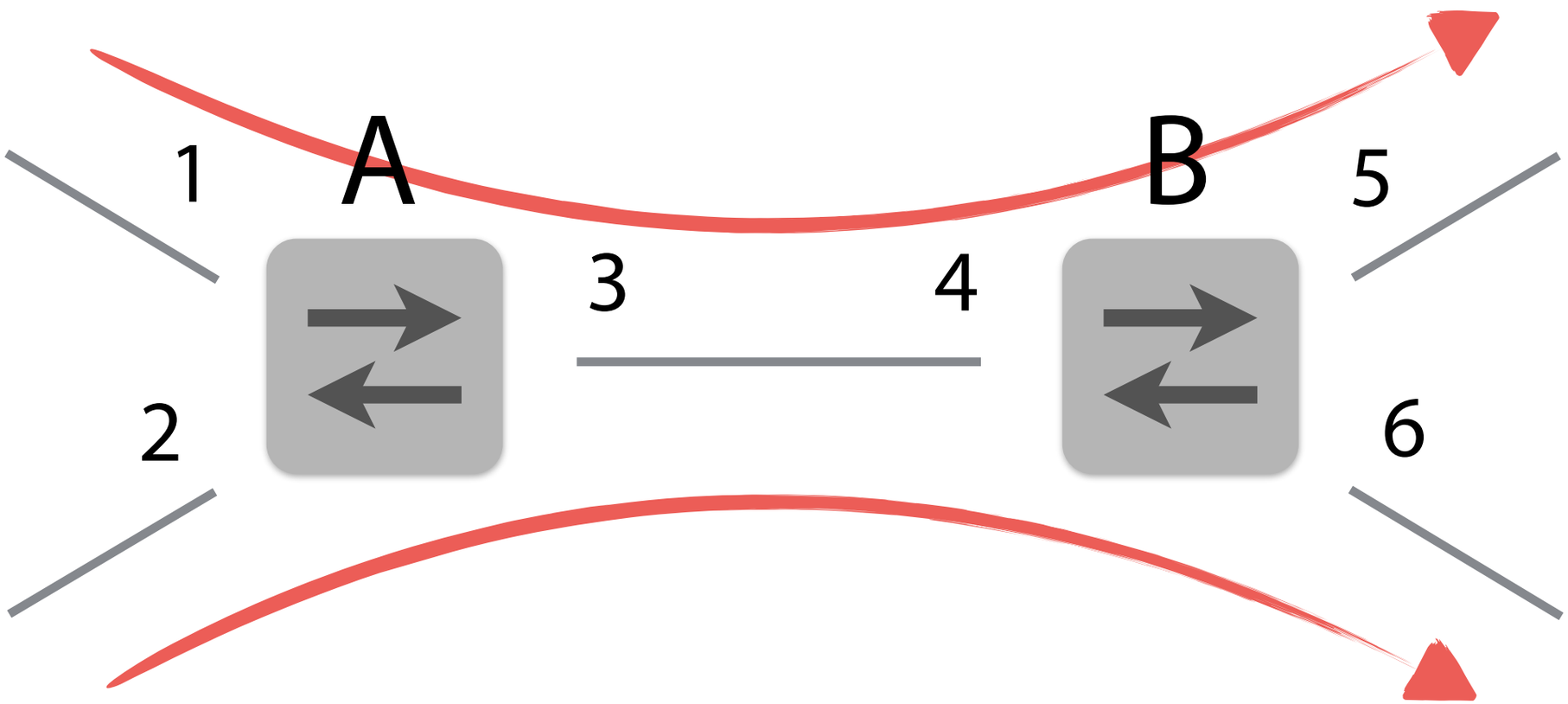
Local Program



pol_A

pol_B

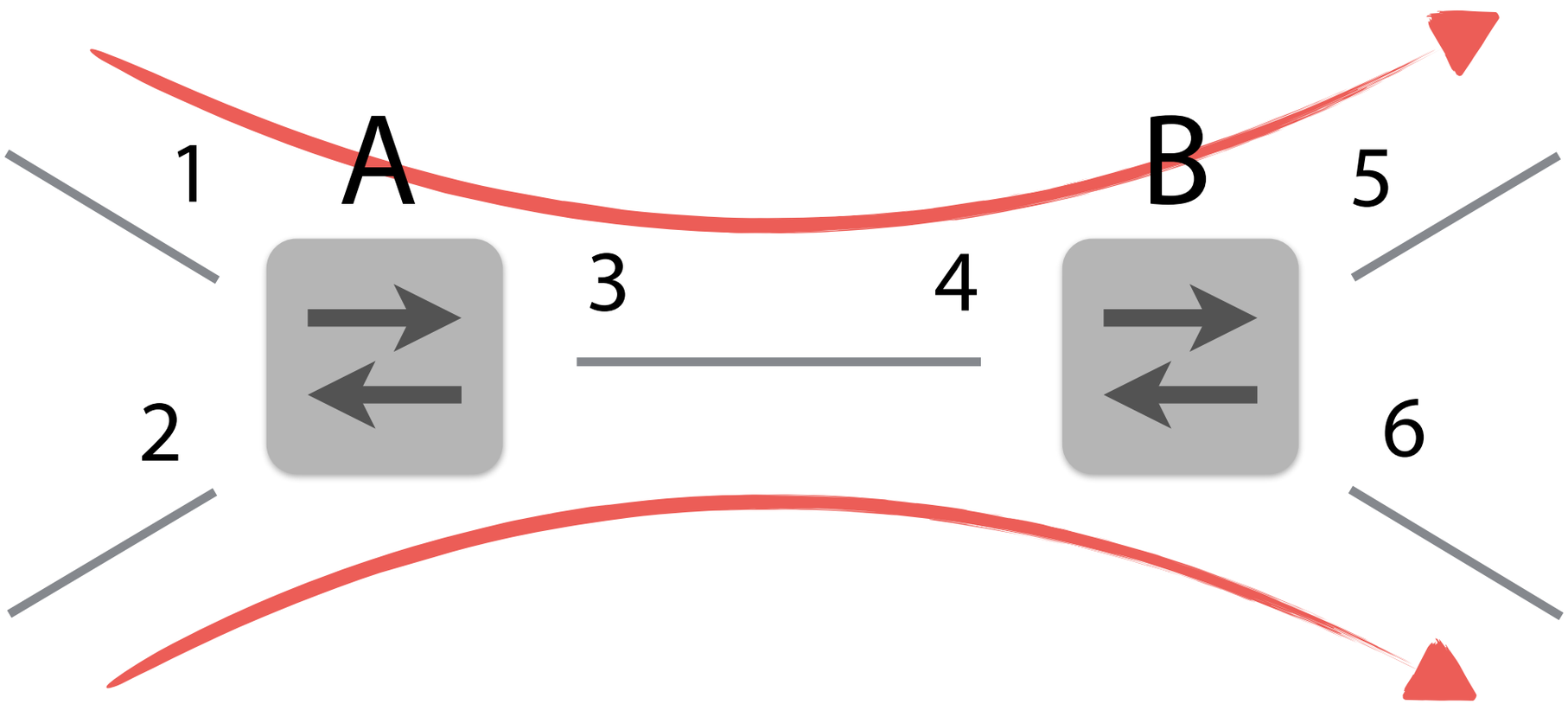
Local Program



port:=3

???

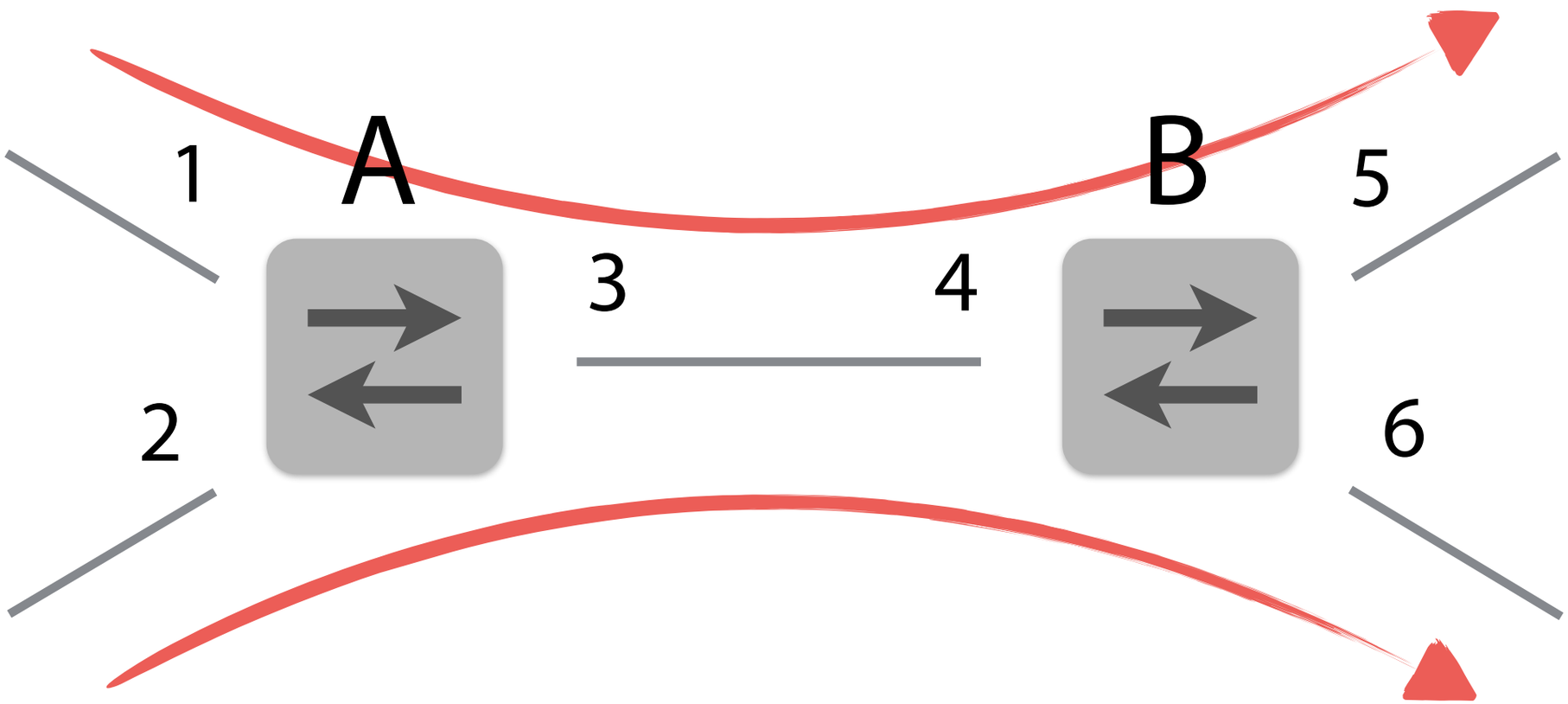
Local Program



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

???

Local Program

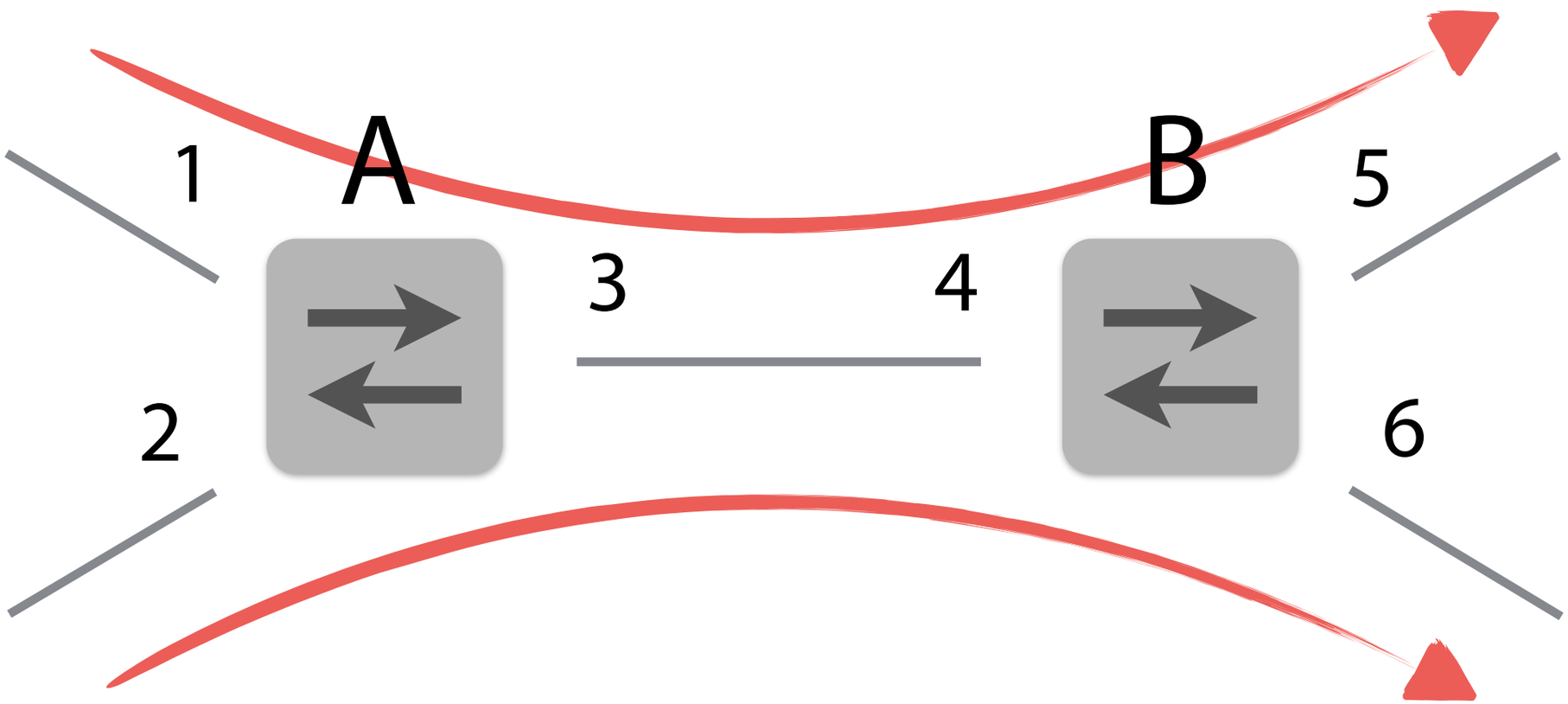


port=1; tag:=1; port:=3
+
port=2; tag:=2; port:=3

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

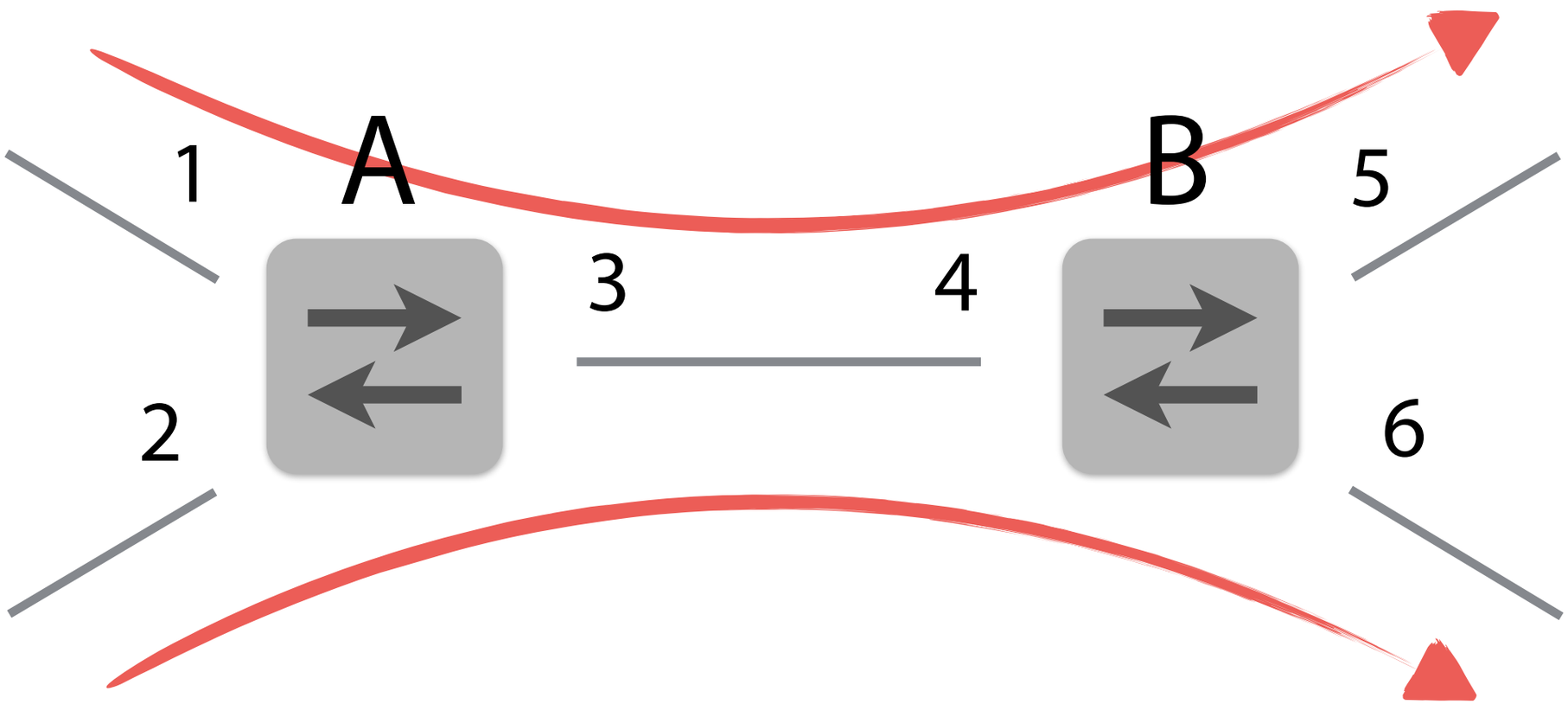
Tedious for programmers… difficult to get right!

Global Program



pol

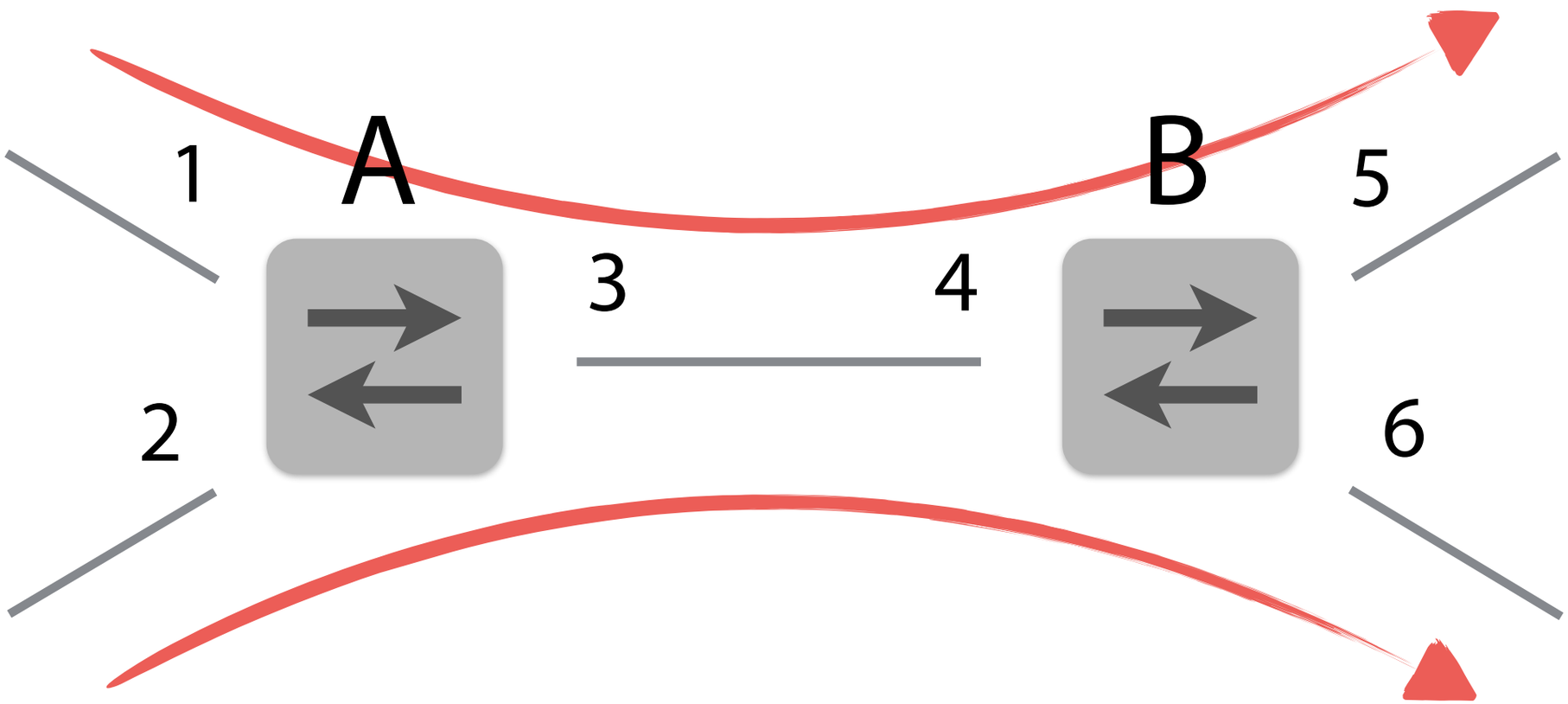
Global Program



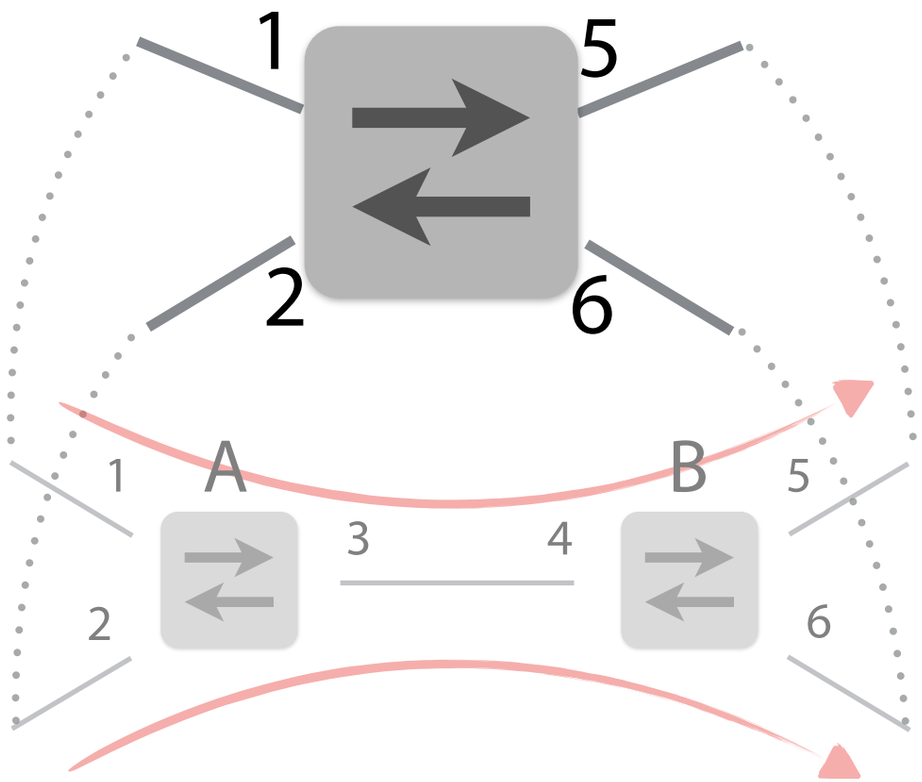
port=1; **A→B;** port**:=**5
+
port=2; **A→B;** port**:=**6

Simple and elegant!

Virtual Program

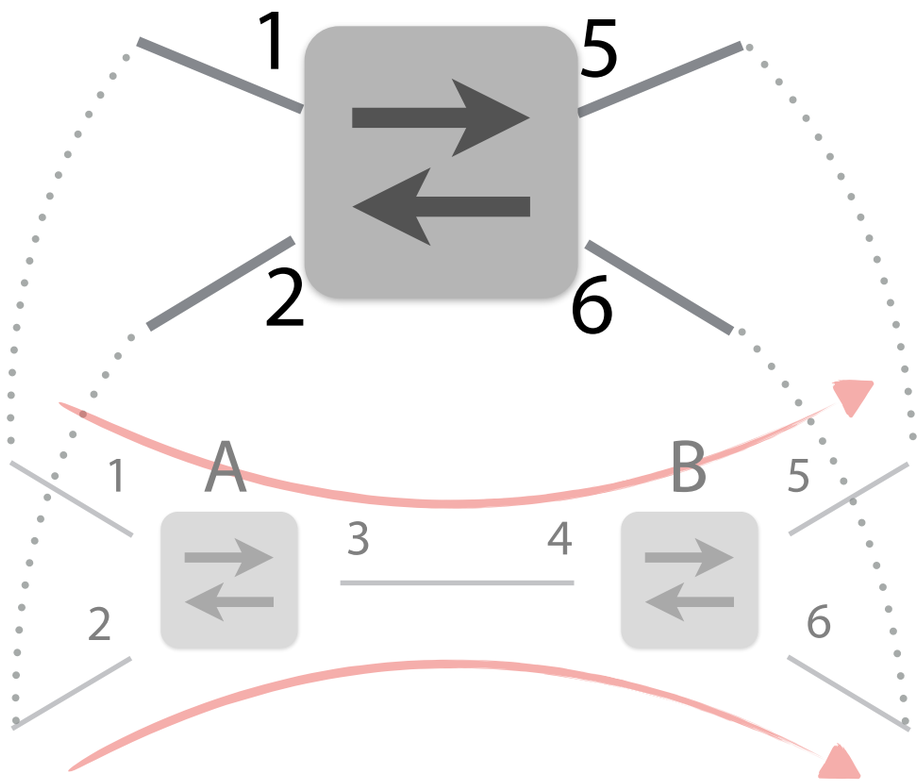


Virtual Program



virtual "big switch"

Virtual Program

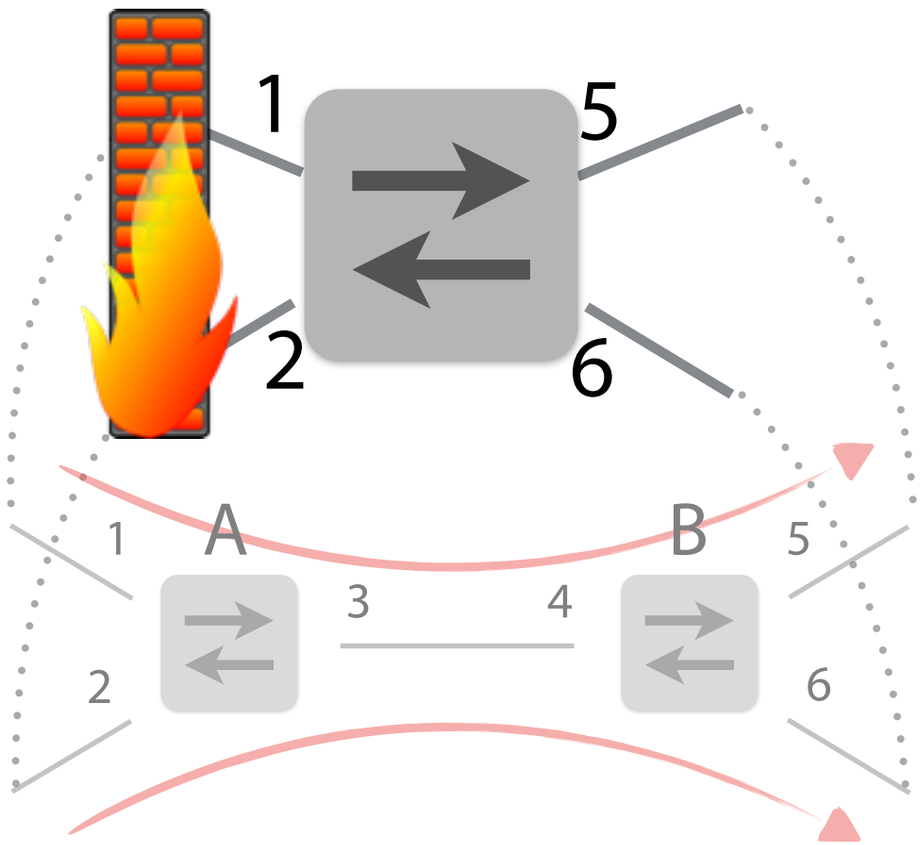


virtual "big switch"

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

Even simpler!

Virtual Program

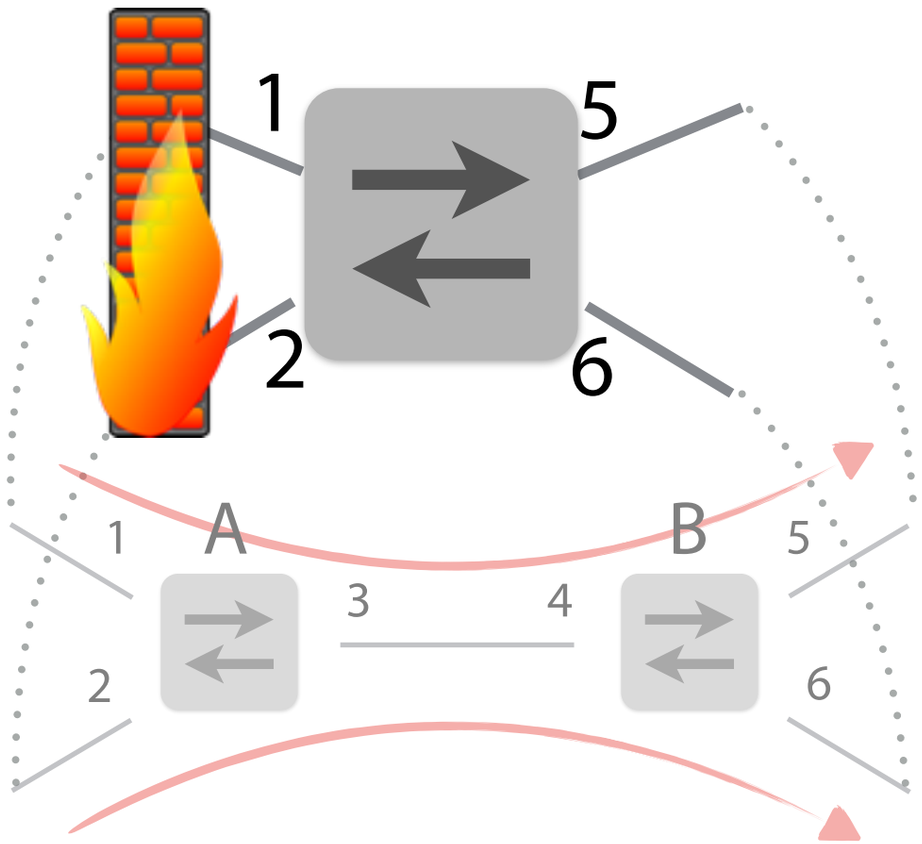


virtual "big switch"

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

Even simpler!

Virtual Program



virtual "big switch"

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

Even simpler!

Virtual Program

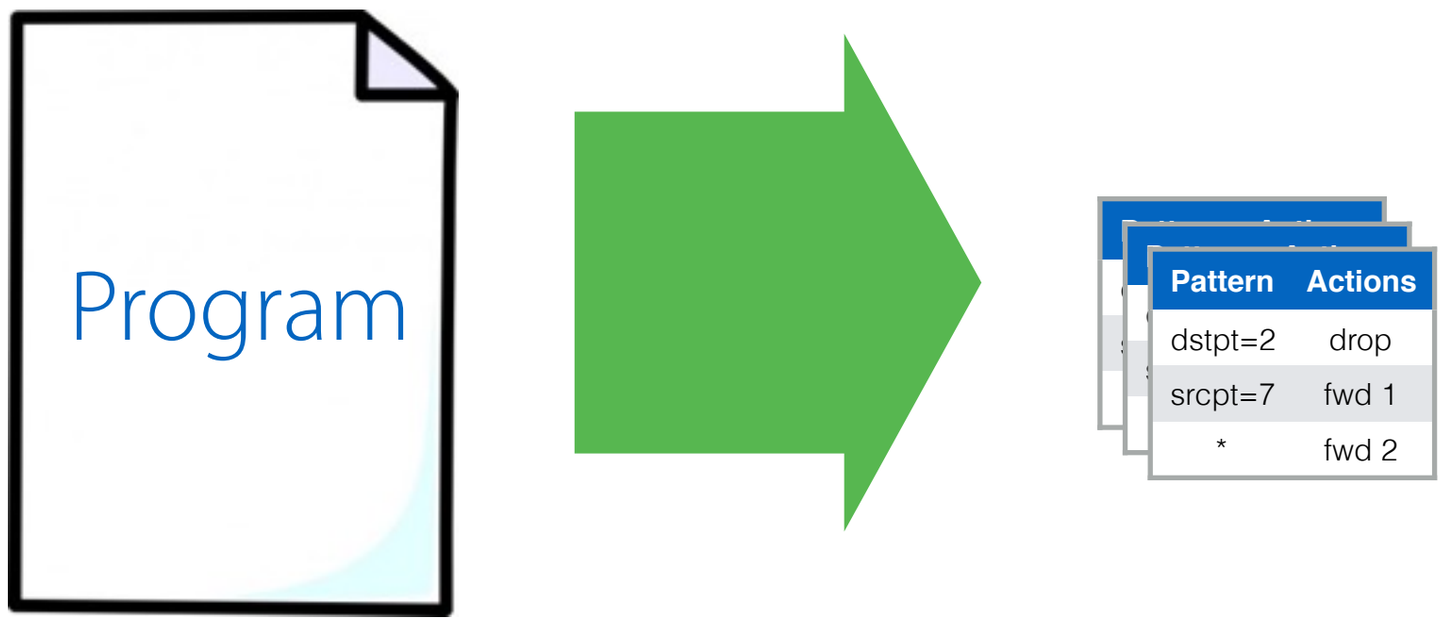
virtual "big switch"

Can implement **multiple** arbitrary **virtual networks**
on top of **single physical network**

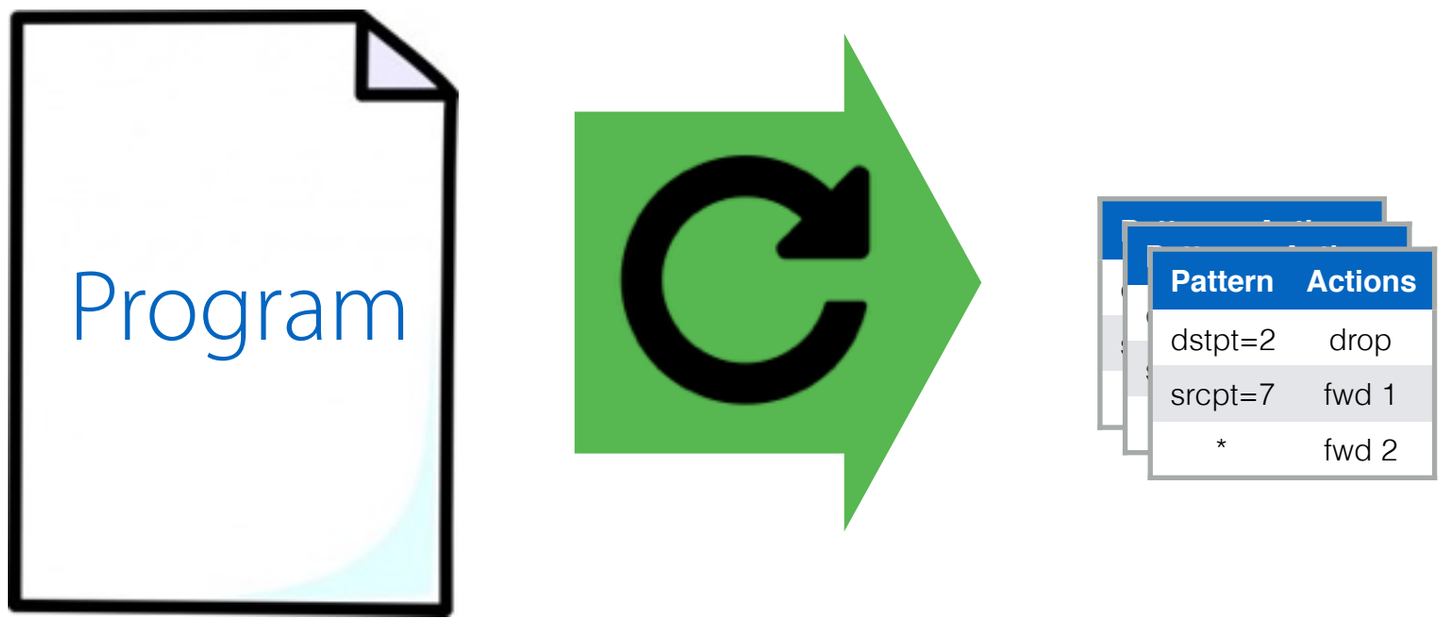
firewall ; port=1; port:=5 + port=2; port:=6

Even simpler!

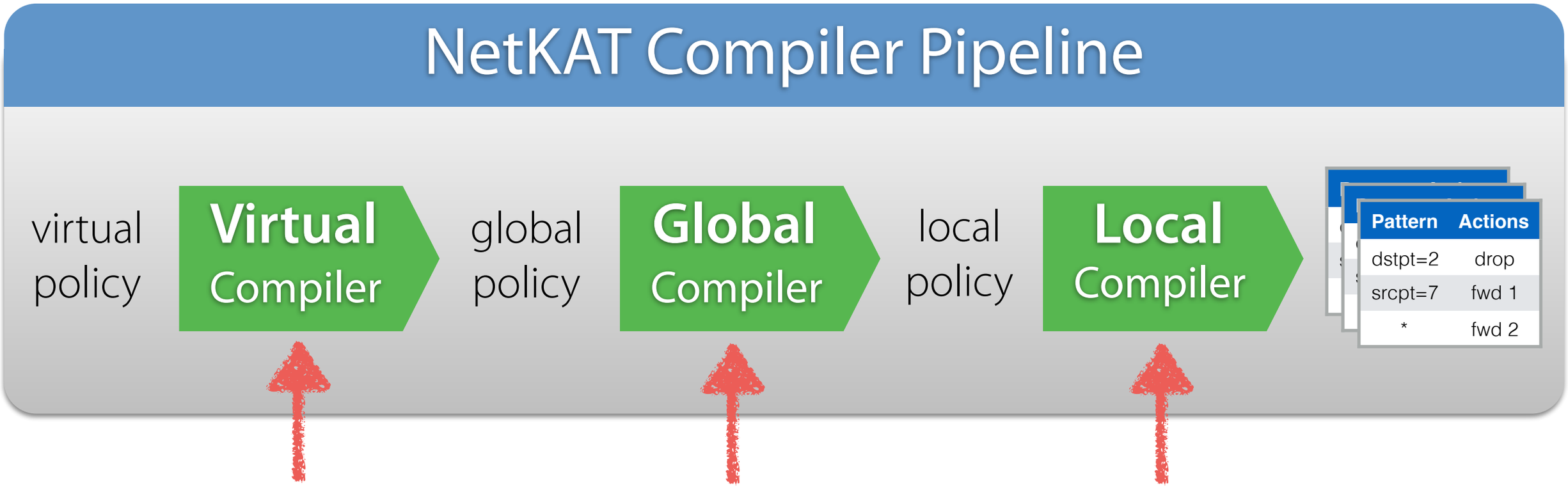
Compilation



Compilation



NetKAT Compiler



abstract topologies

network-wide behavior

~ 100x faster than competitors

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
```

+

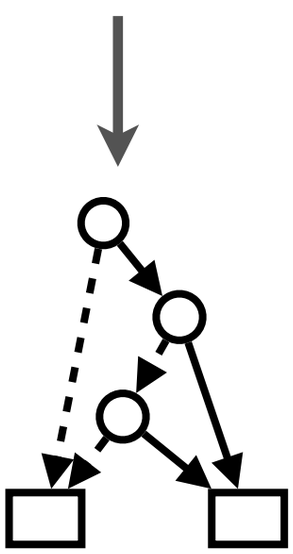
```
let monitor =
if (tcpSrc = 22 + tcpDst = 22) then
  port:=console
else
  false
```

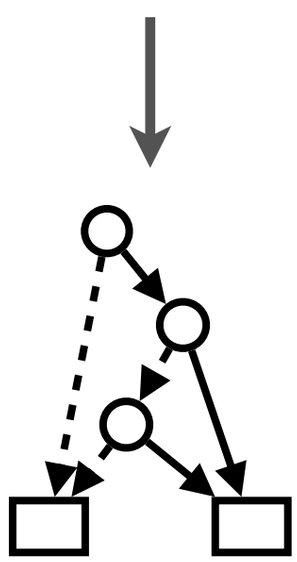
Our Approach

```
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
```



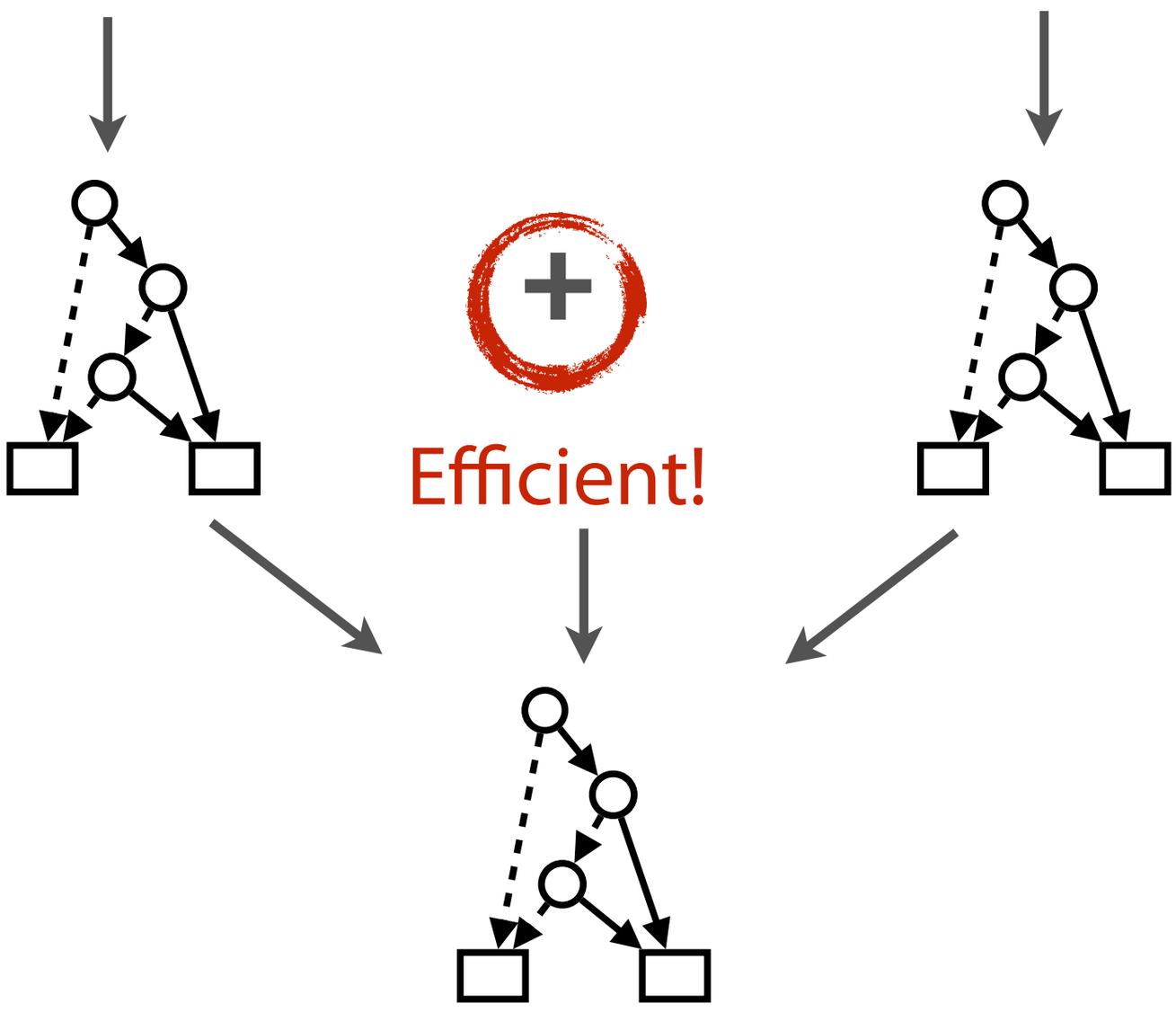


Our Approach

```
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
```

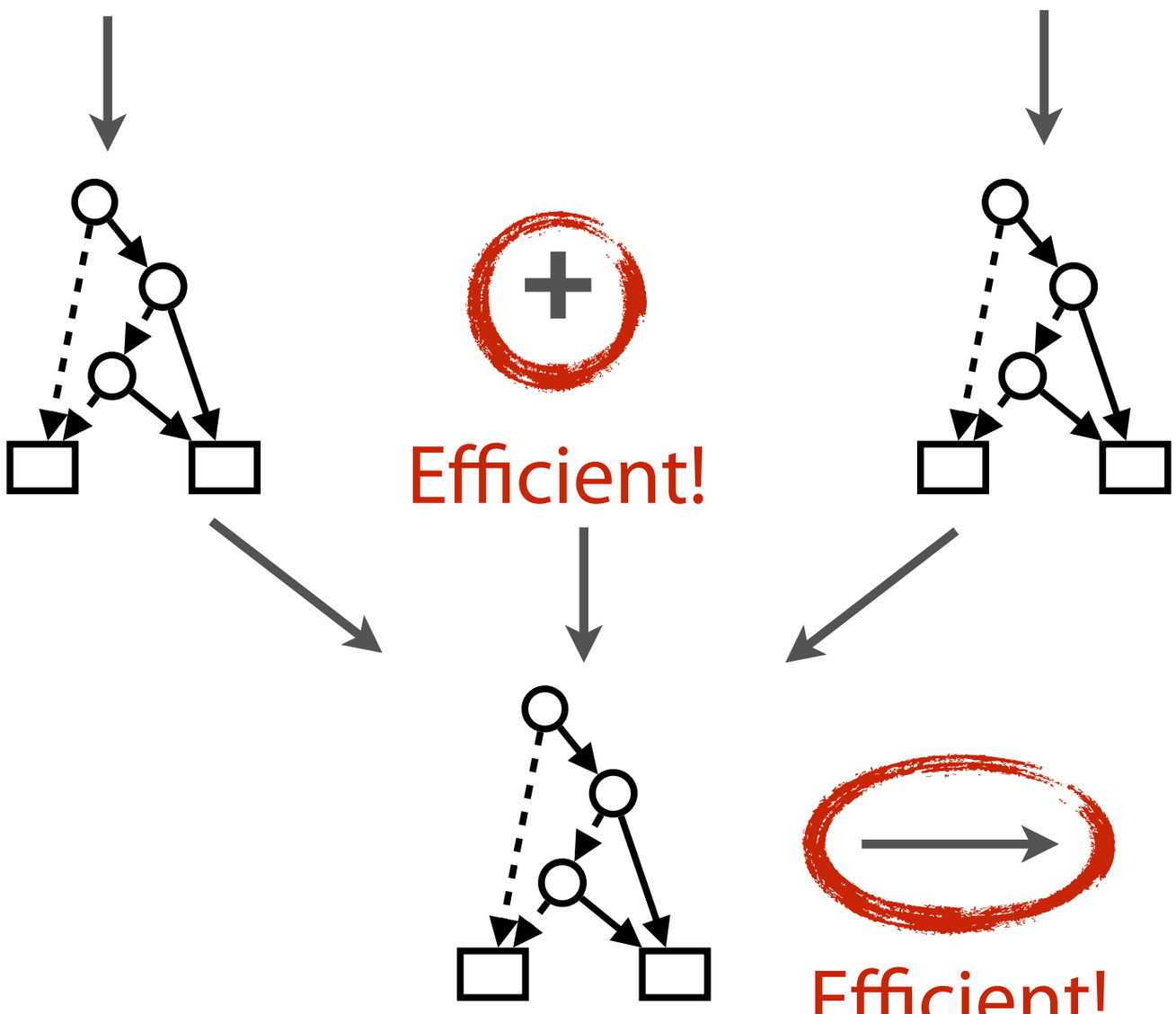


Our Approach

```
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
…	…

Our Approach

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

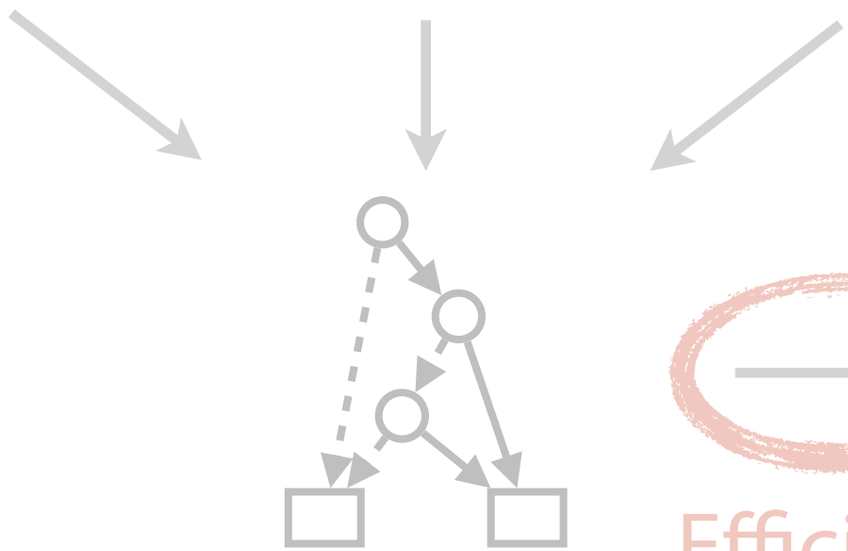
+

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

Key Data Structure:

Forwarding Decision Diagram

→ now widely adopted



Efficient!

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

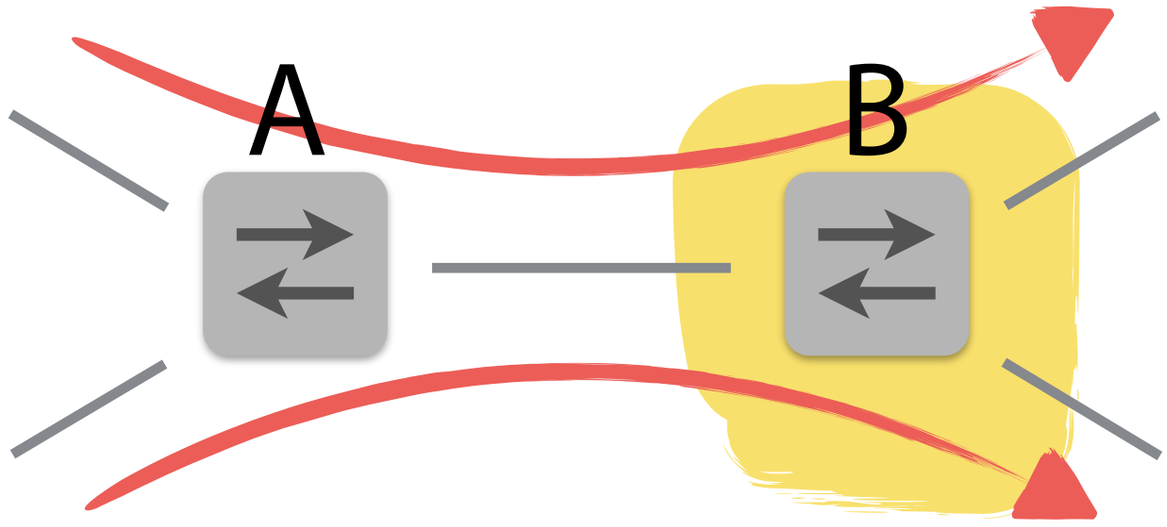
Global Compilation

Input: NetKAT program (with links)

Output: equivalent local program (without links)

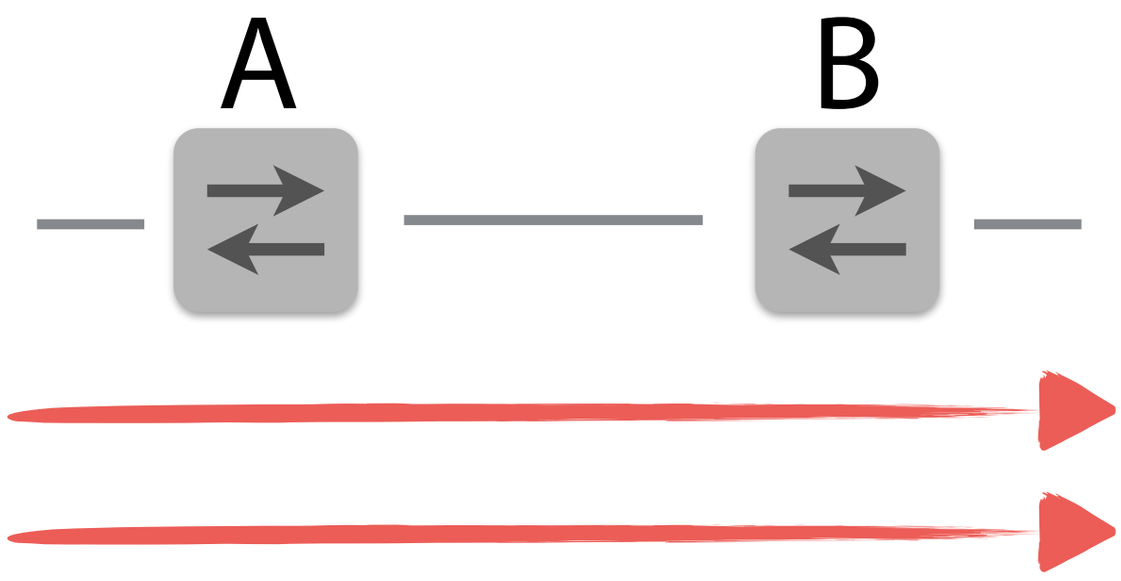
Main Challenges

1. Adding Extra State "Tagging"

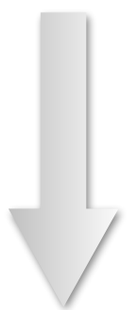


2. Avoiding Duplication

(naive tagging is unsound!)

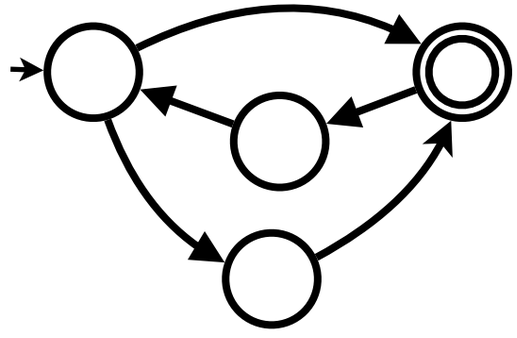


Our Solution

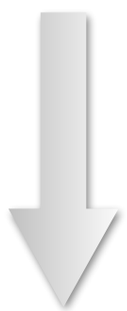


Adding Extra State
= Translation to Automaton

NetKAT NFA

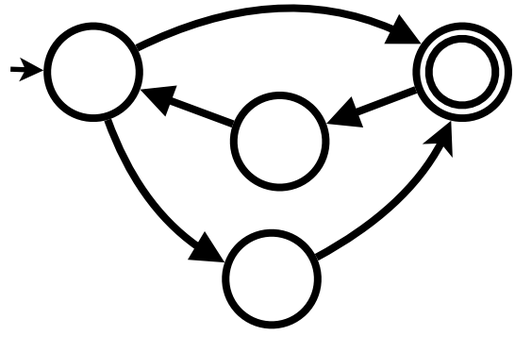


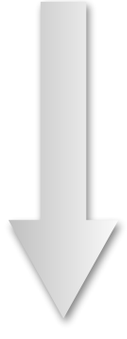
Our Solution



Adding Extra State
= Translation to Automaton

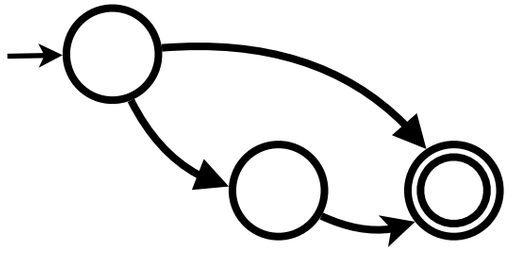
NetKAT NFA





Avoiding Duplication
= Determinization

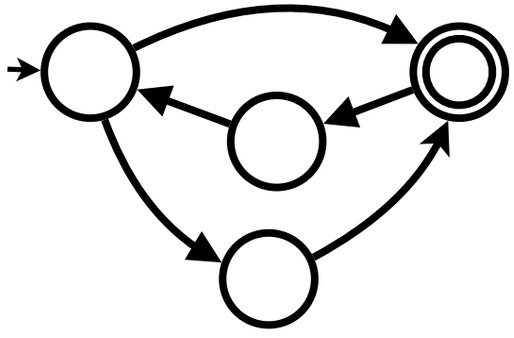
NetKAT DFA



Our Solution

Adding Extra State
= Translation to Automaton

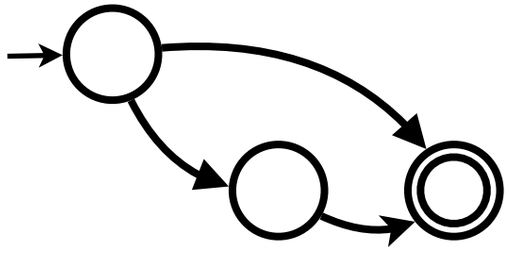
NetKAT NFA



Automaton Minimization
= Tag Elimination

Avoiding Duplication
= Determinization

NetKAT DFA

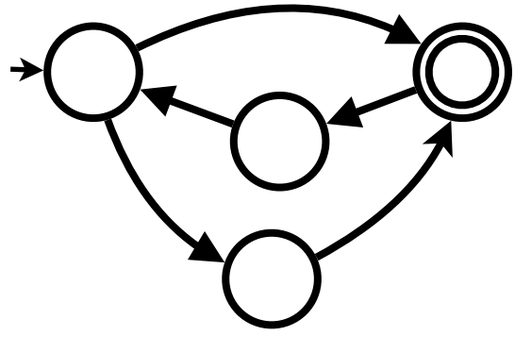


Our Solution

Global Program

Adding Extra State
= Translation to Automaton

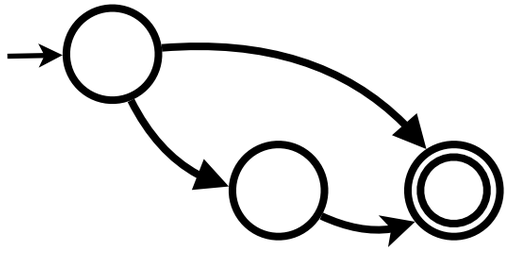
NetKAT NFA



Automaton Minimization
= Tag Elimination

Avoiding Duplication
= Determinization

NetKAT DFA



Local Program

NetKAT Automata [Foster et al, POPL '15]

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

"Alphabet size": |Packet x 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|

Can represent δ **symbolically** using FDDs!

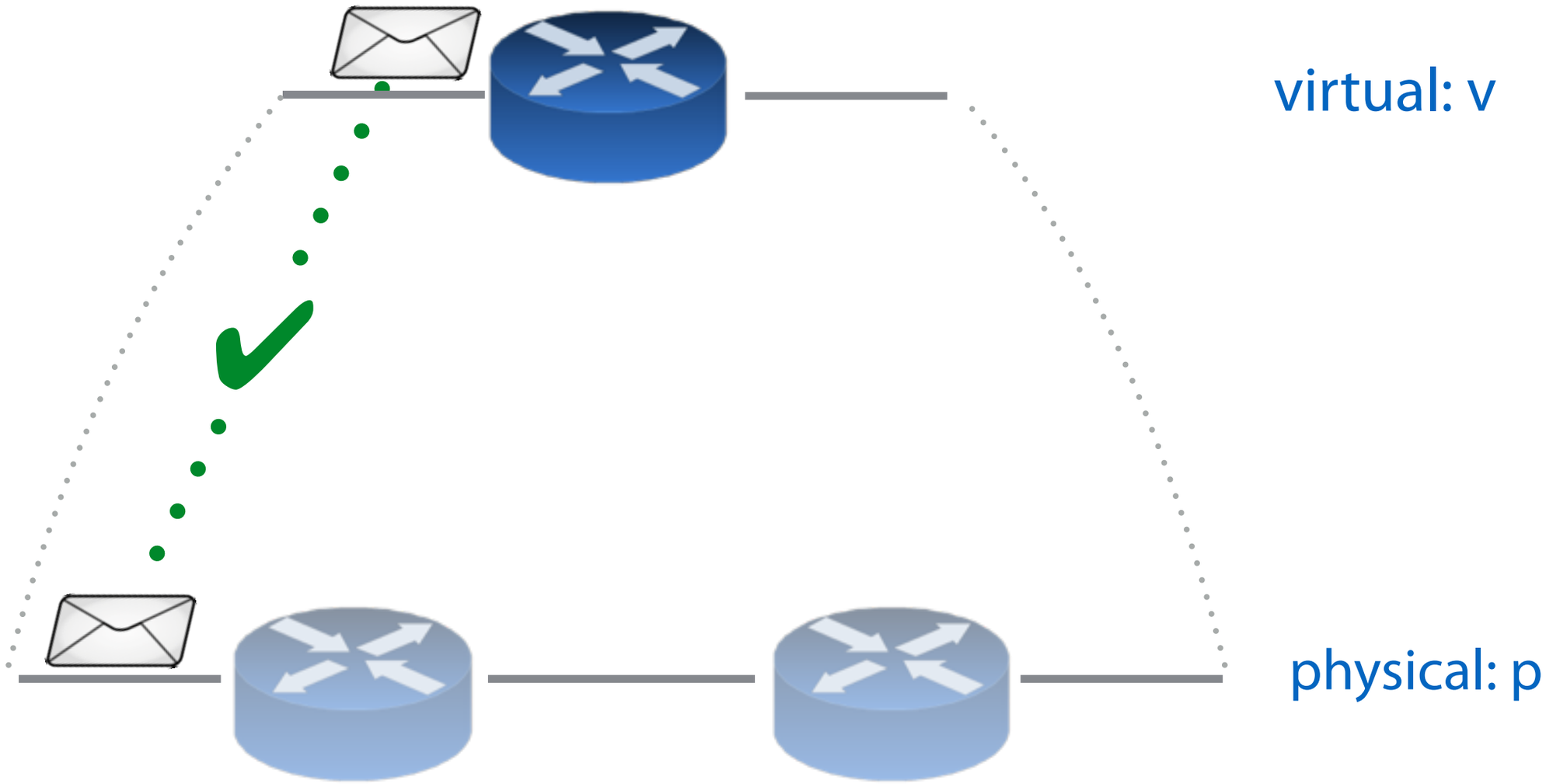
Automata construction:
Antimirov partial derivatives & **Position Automata**

Virtual Compilation

Input: program written against virtual topology

Output: global program that simulates virtual behavior

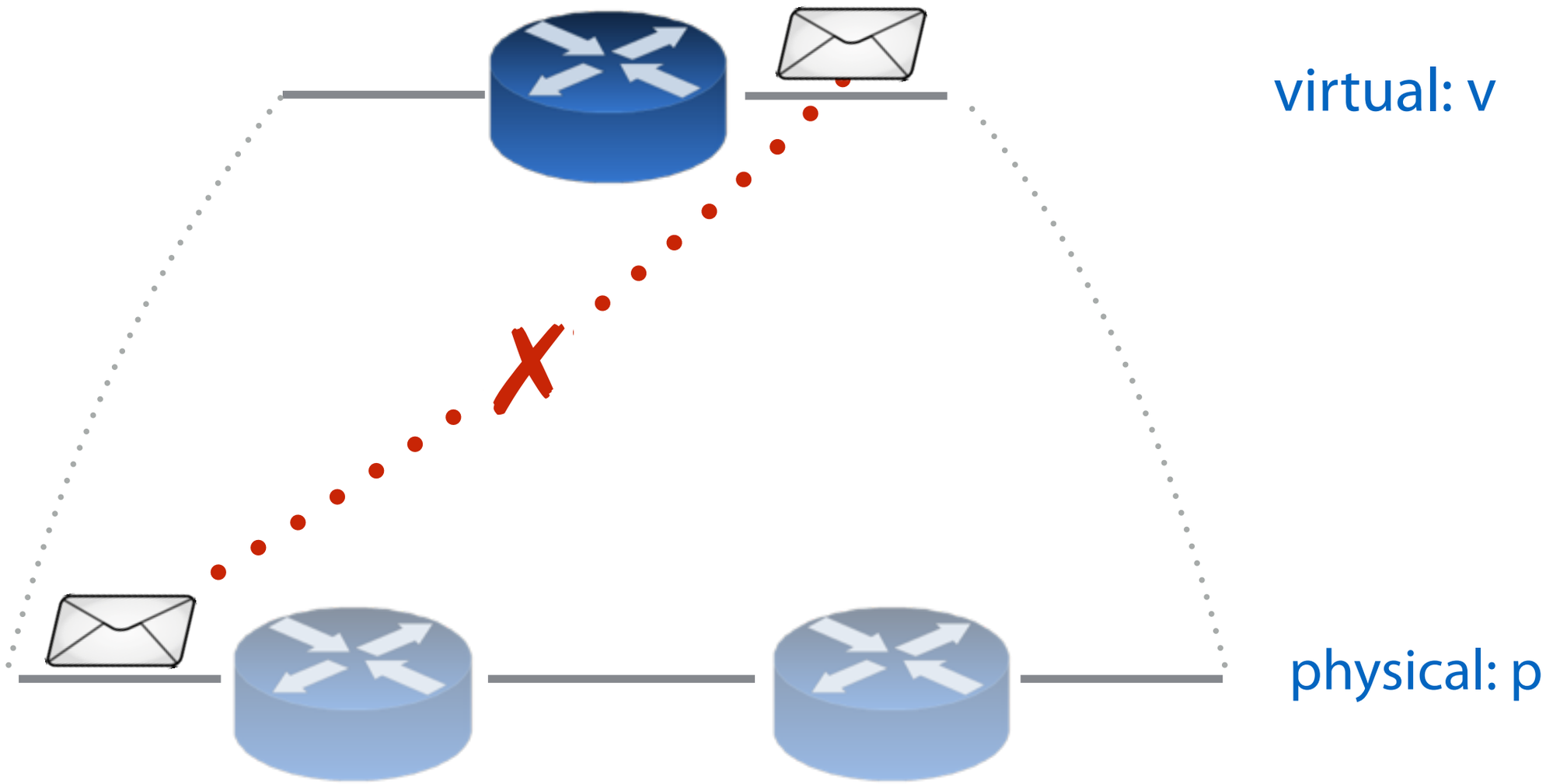
Virtualization



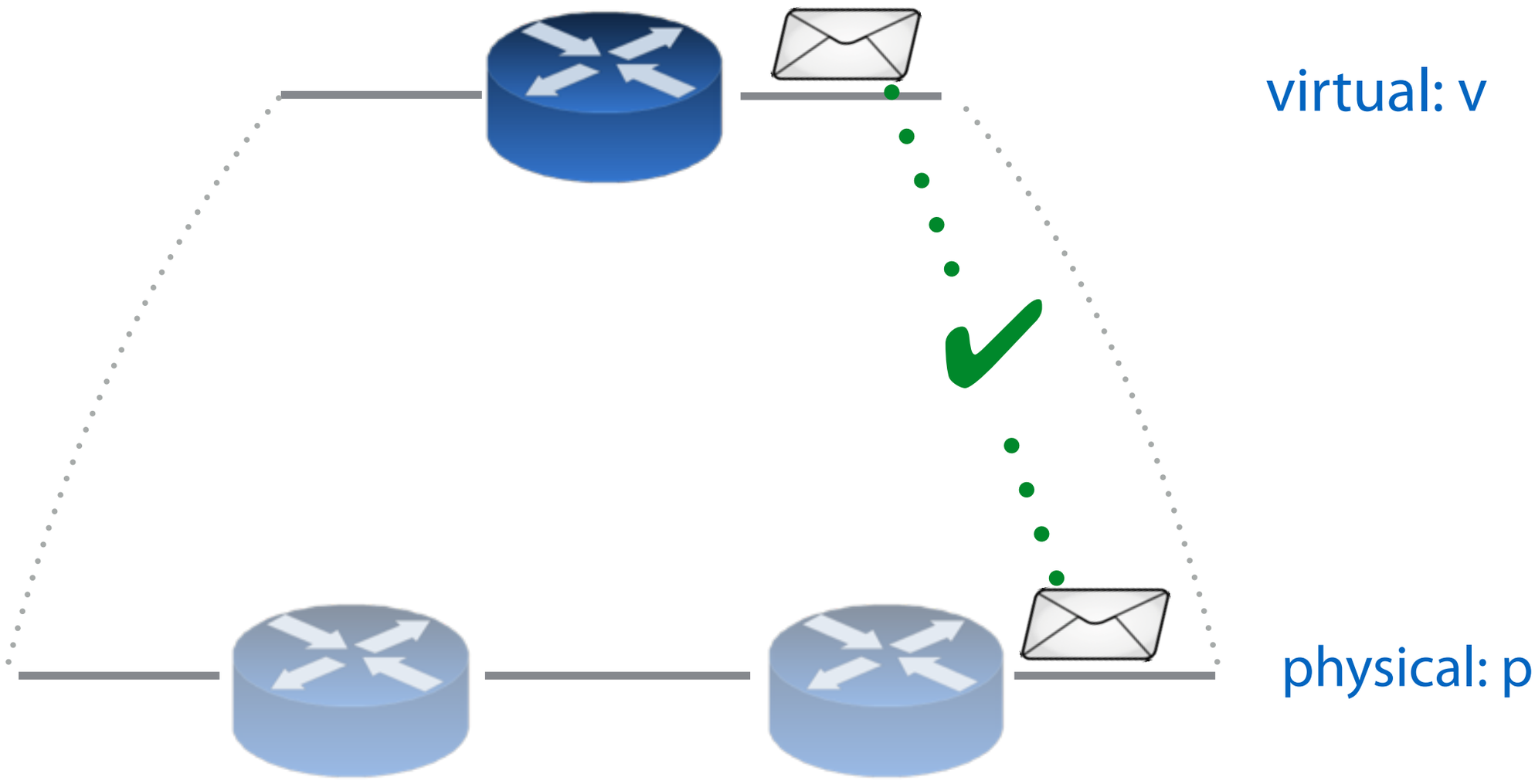
virtual: v

physical: p

Virtualization



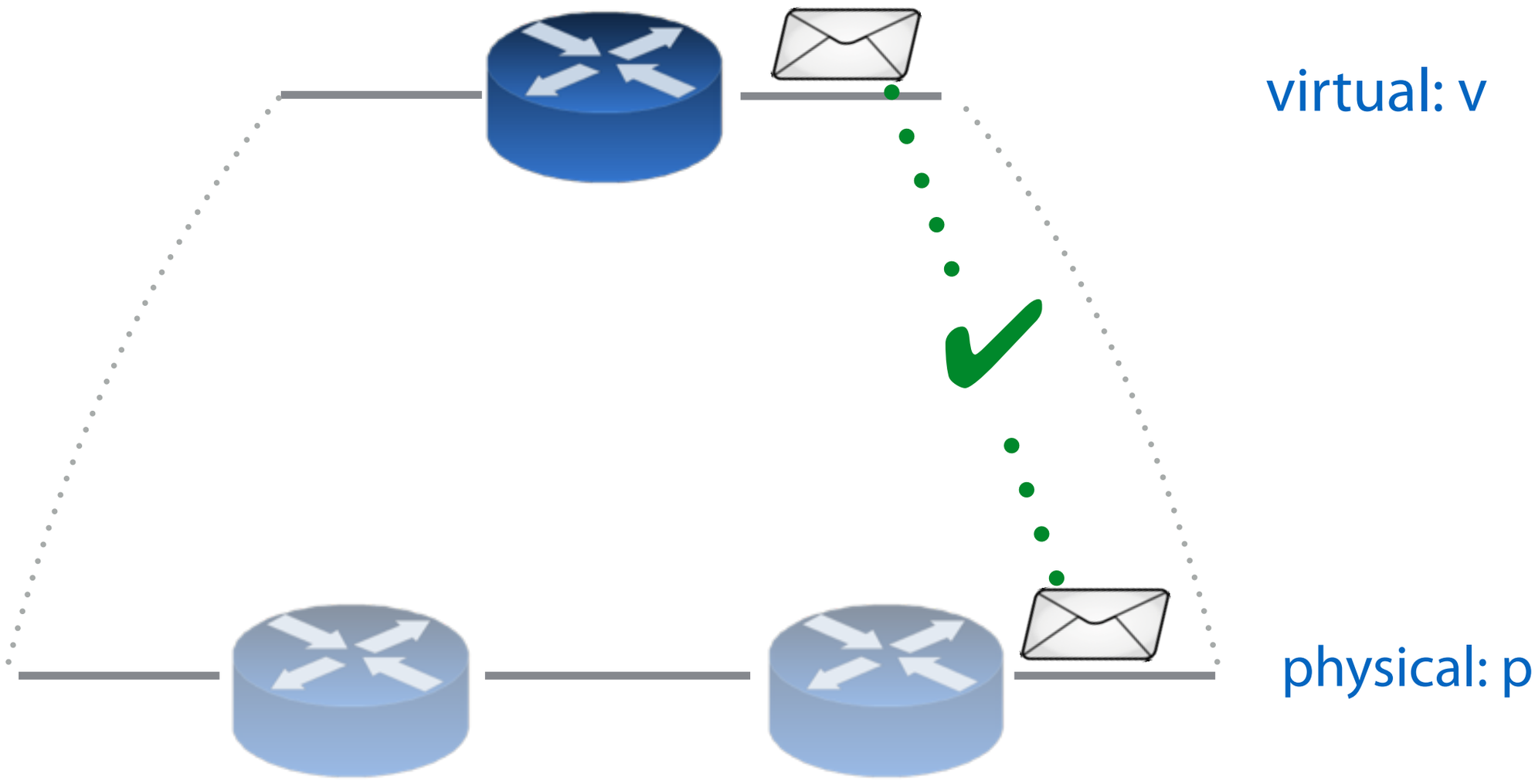
Virtualization



virtual: v

physical: p

Virtualization

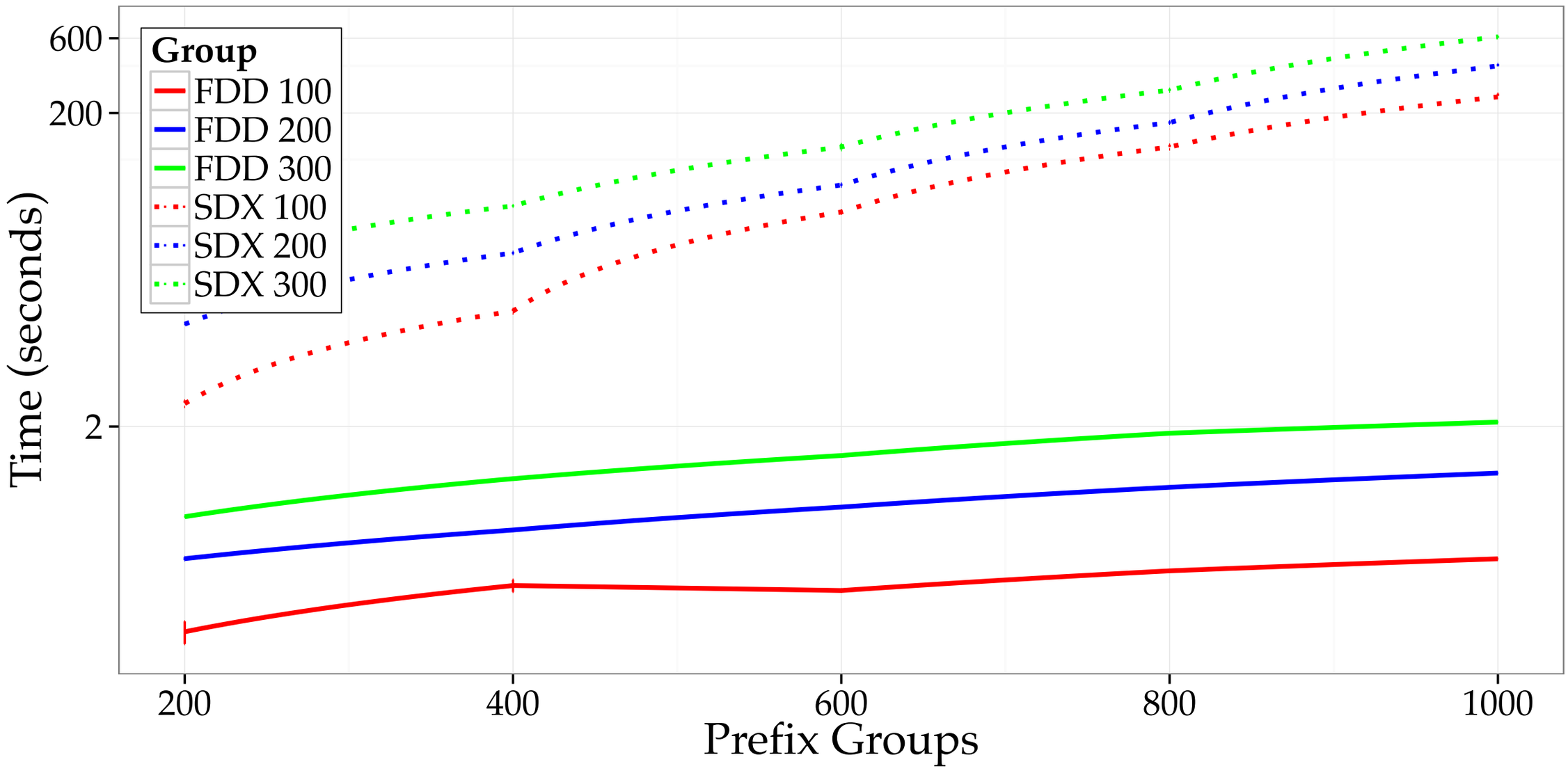


Observation: can formulate execution of a virtual program as a two-player game

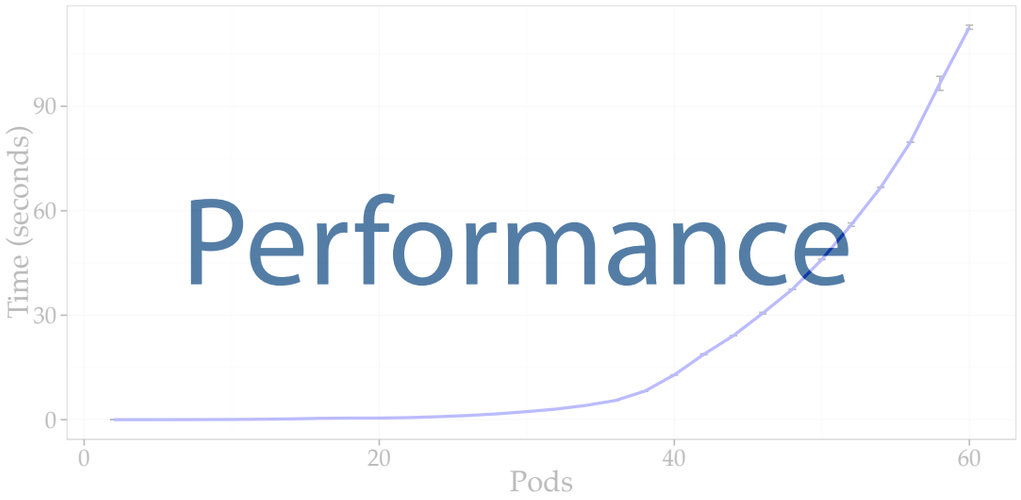
Compiler: synthesizes physical program p that encodes a winning strategy to all instances of that game

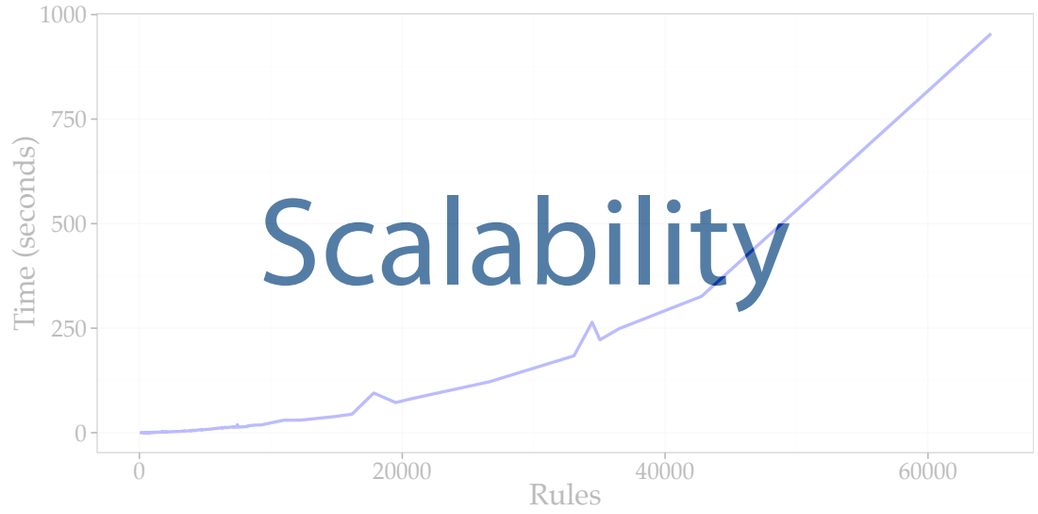
Evaluation

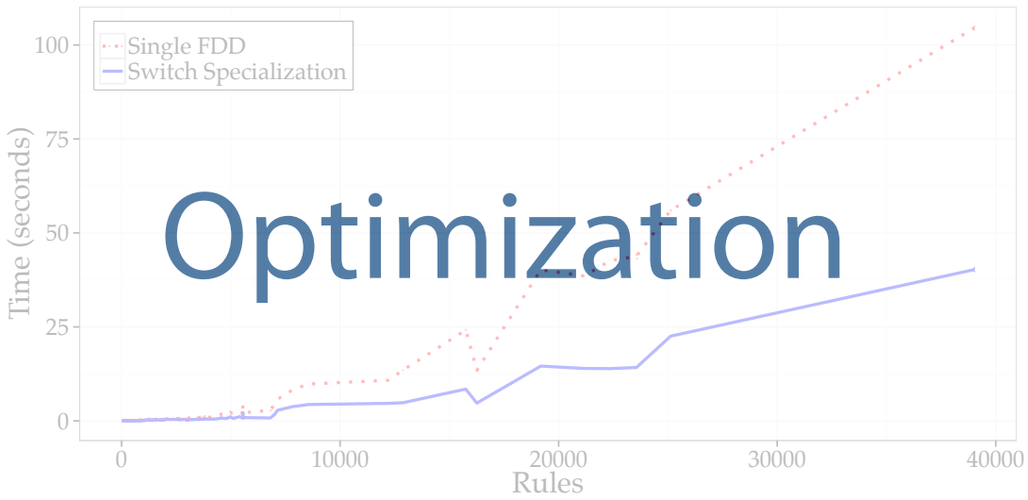
Local Compiler vs State of the Art

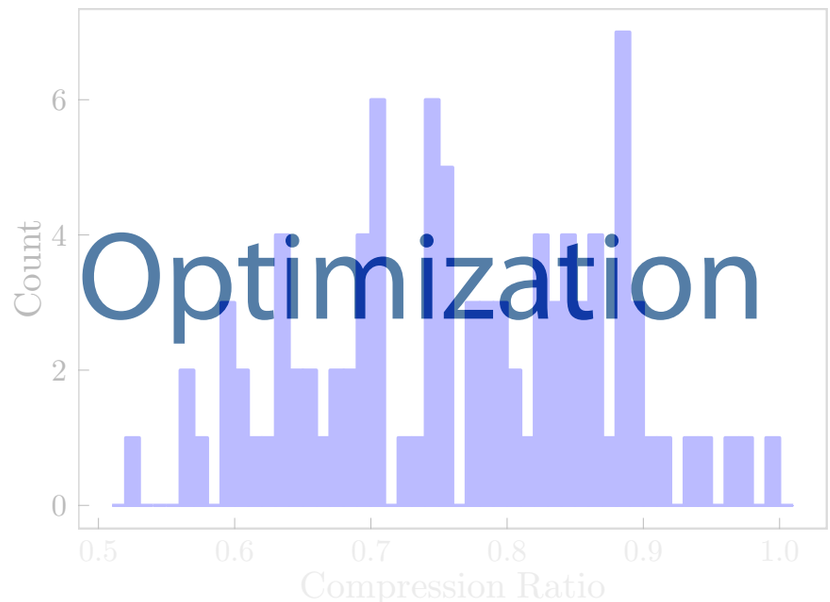


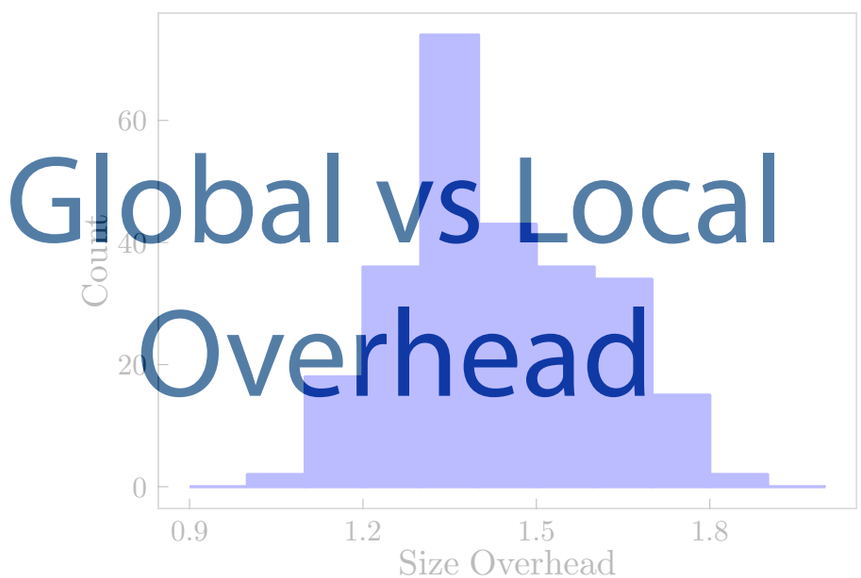
about 100x speedup

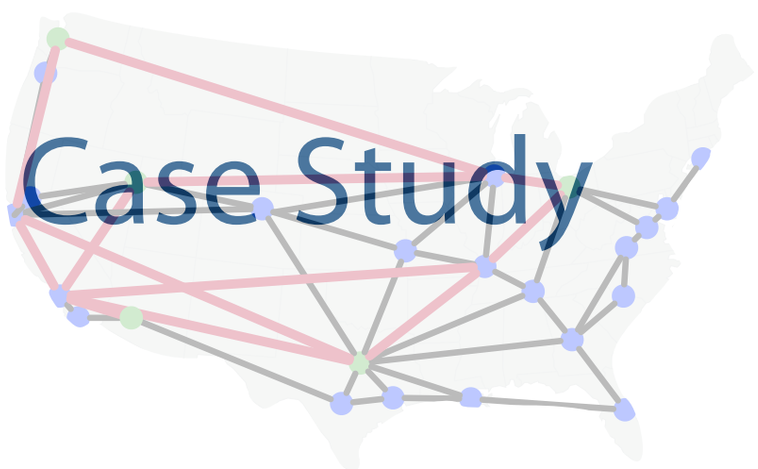

Performance
Time (seconds)
0
30
60
90
0
20
40
60
Pods


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

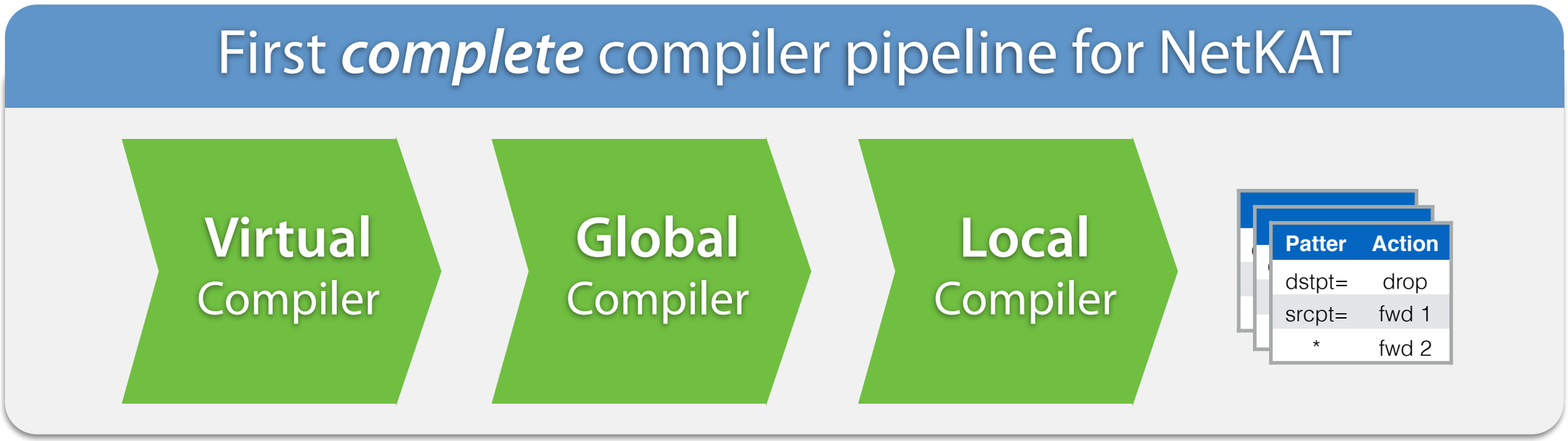

Optimization
Single FDD
Switch Specialization
Time (seconds)
0
25
50
75
100
0
10000
20000
30000
40000
Rules


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