Boosting existing networks with SDN

A bird in the hand is worth two in the bush

Laurent Vanbever

ETH Zürich (D-ITET)

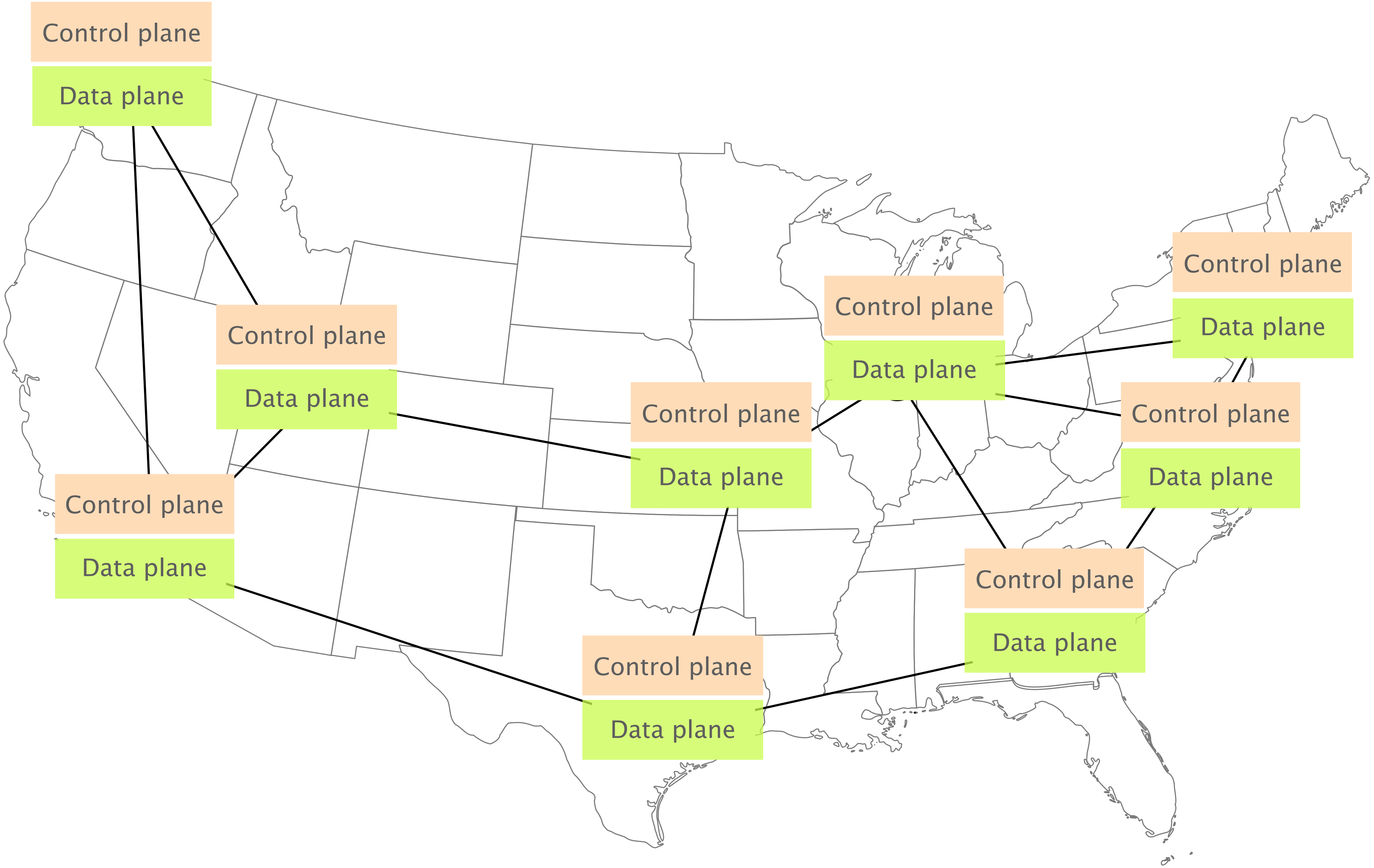
Hebrew U. net. seminar

June, 9 2015

Software-Defined Network

Why?!

A network is a distributed system whose behavior depends on each element configuration



Configuring each element is often done manually,
using arcane low-level, vendor-specific “languages”

Configuring each element is **often done manually,** using arcane **low-level, vendor-specific** “languages”

Cisco IOS

```
!
ip multicast-routing
!
interface Loopback0
 ip address 120.1.7.7 255.255.255.255
 ip ospf 1 area 0
!
!
interface Ethernet0/0
 no ip address
!
interface Ethernet0/0.17
 encapsulation dot1Q 17
 ip address 125.1.17.7 255.255.255.0
 ip pim bsr-border
 ip pim sparse-mode
!
!
router ospf 1
 router-id 120.1.7.7
 redistribute bgp 700 subnets
!
router bgp 700
 neighbor 125.1.17.1 remote-as 100
 !
 address-family ipv4
  redistribute ospf 1 match internal external 1 external 2
  neighbor 125.1.17.1 activate
 !
 address-family ipv4 multicast
  network 125.1.79.0 mask 255.255.255.0
  redistribute ospf 1 match internal external 1 external 2
```

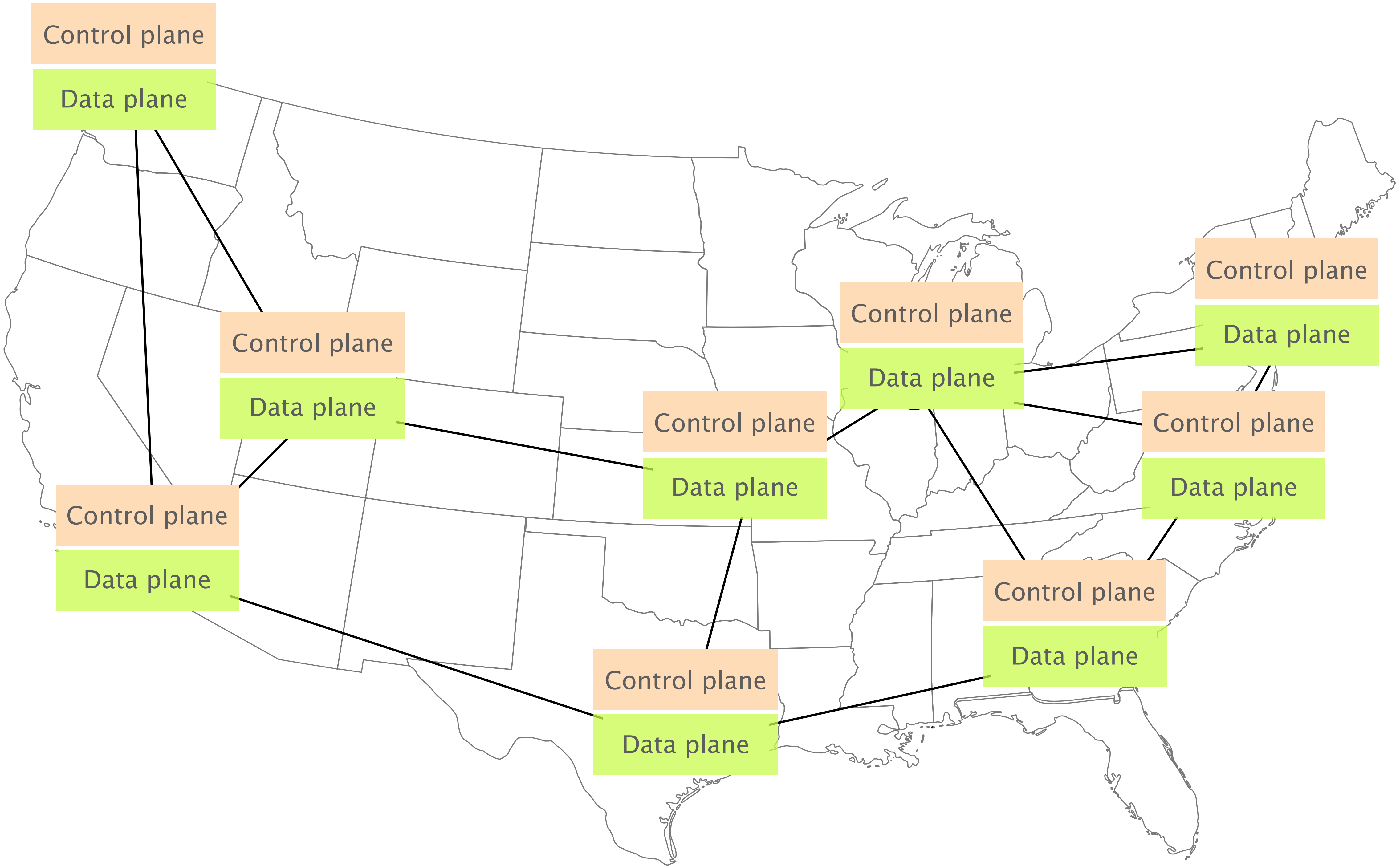
Juniper JunOS

```
interfaces {
   so-0/0/0 {
        unit 0 {
            family inet {
                address 10.12.1.2/24;
            }
            family mpls;
        }
    }
   ge-0/1/0 {
        vlan-tagging;
        unit 0 {
            vlan-id 100;
            family inet {
                address 10.108.1.1/24;
            }
            family mpls;
        }
        unit 1 {
            vlan-id 200;
            family inet {
                address 10.208.1.1/24;
            }
        }
    }
…
}
protocols {
    mpls {
        interface all;
    }
    bgp {
```

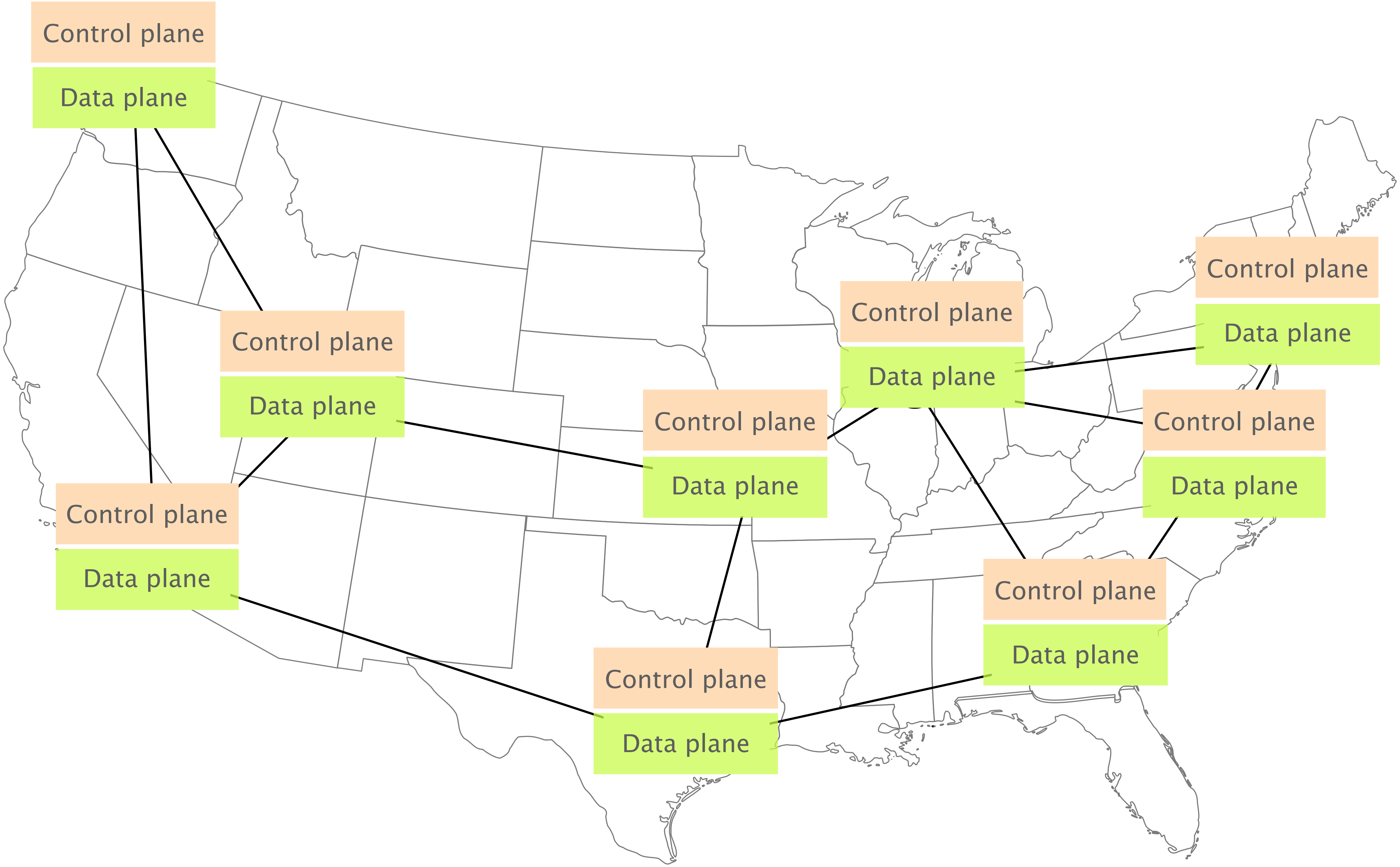
“Human factors are responsible for 50% to 80% of network outages”

Juniper Networks, *What’s Behind Network Downtime?*, 2008

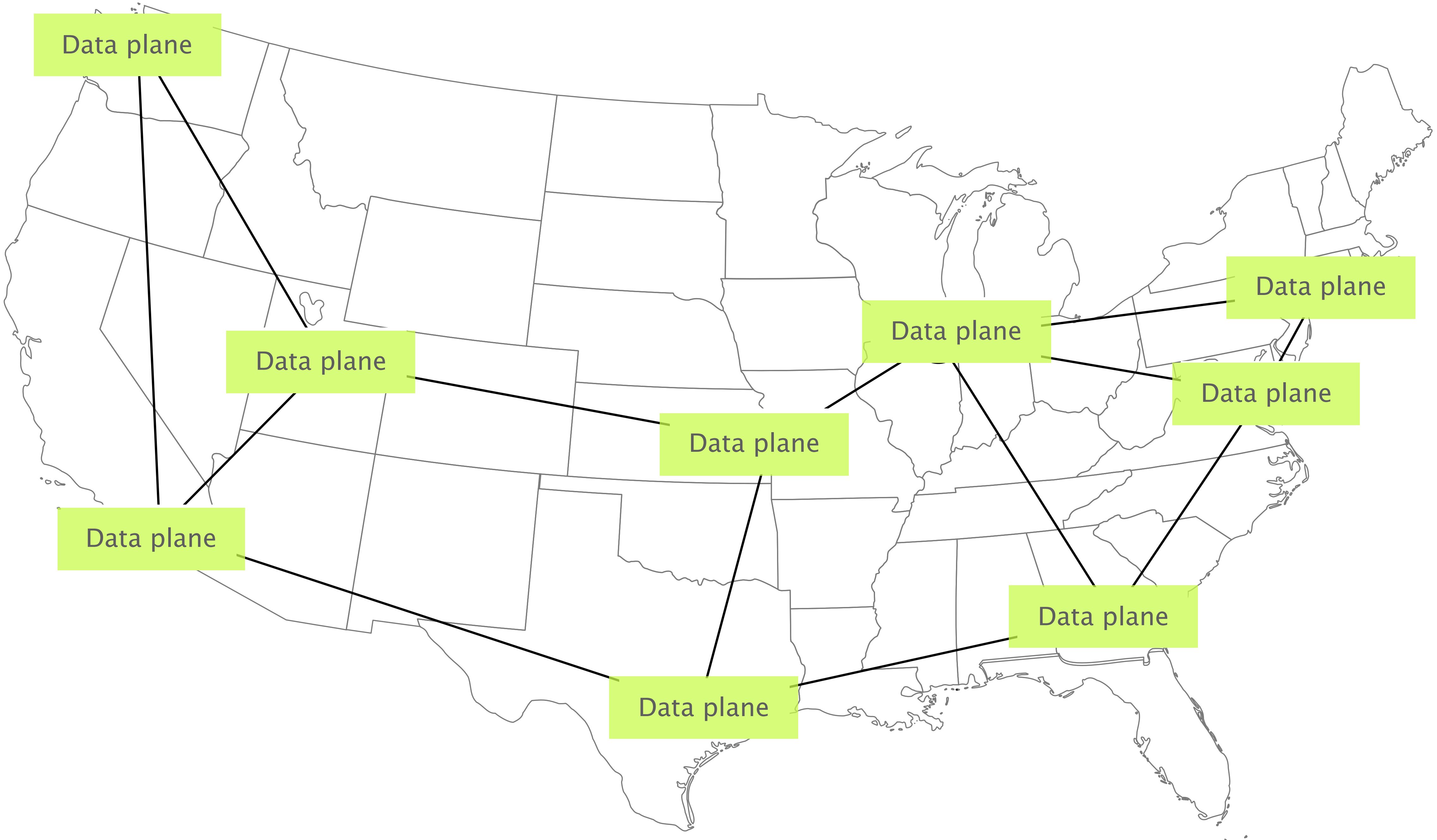
In contrast, SDN simplifies networks…



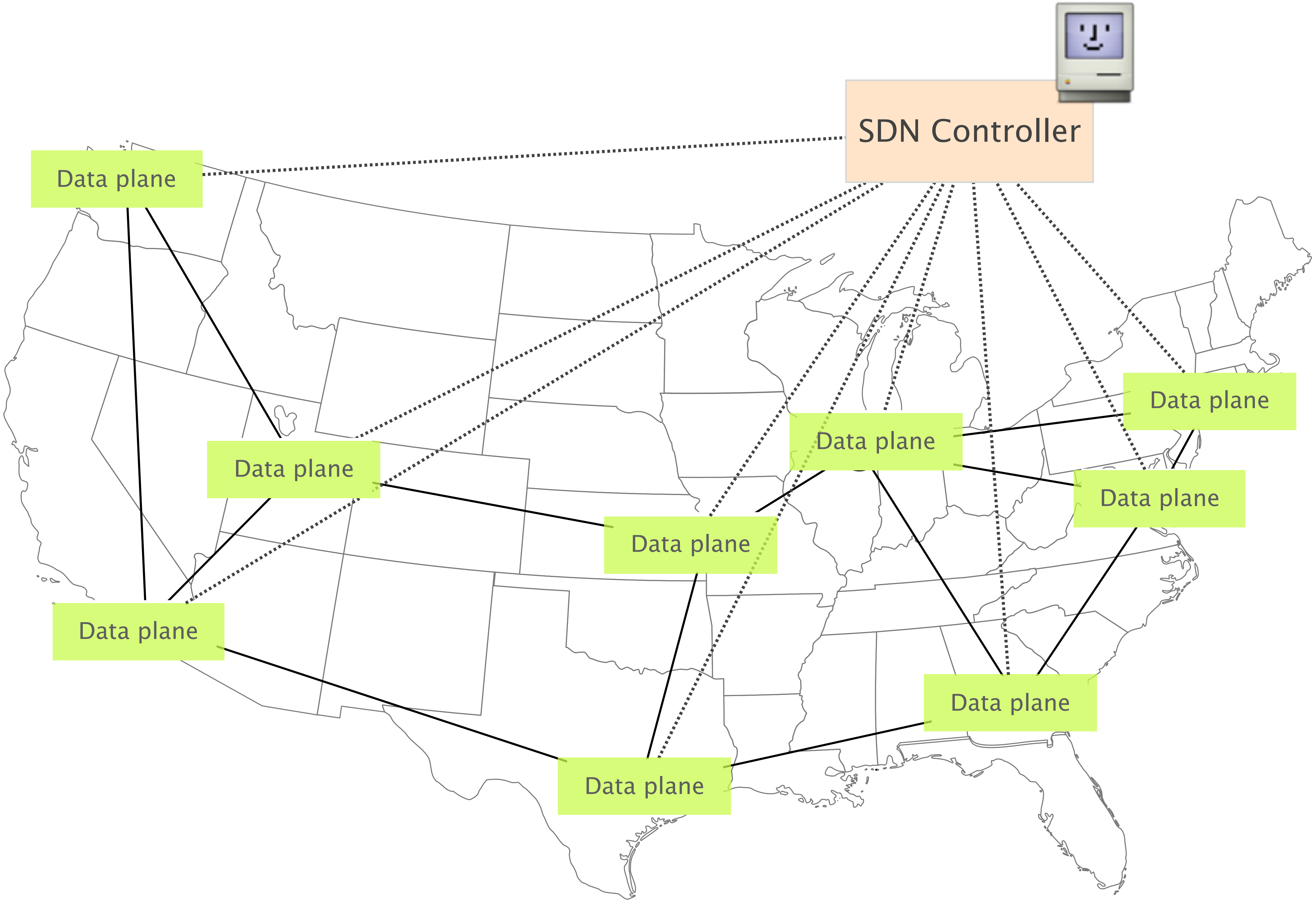
… by removing the intelligence from the equipments



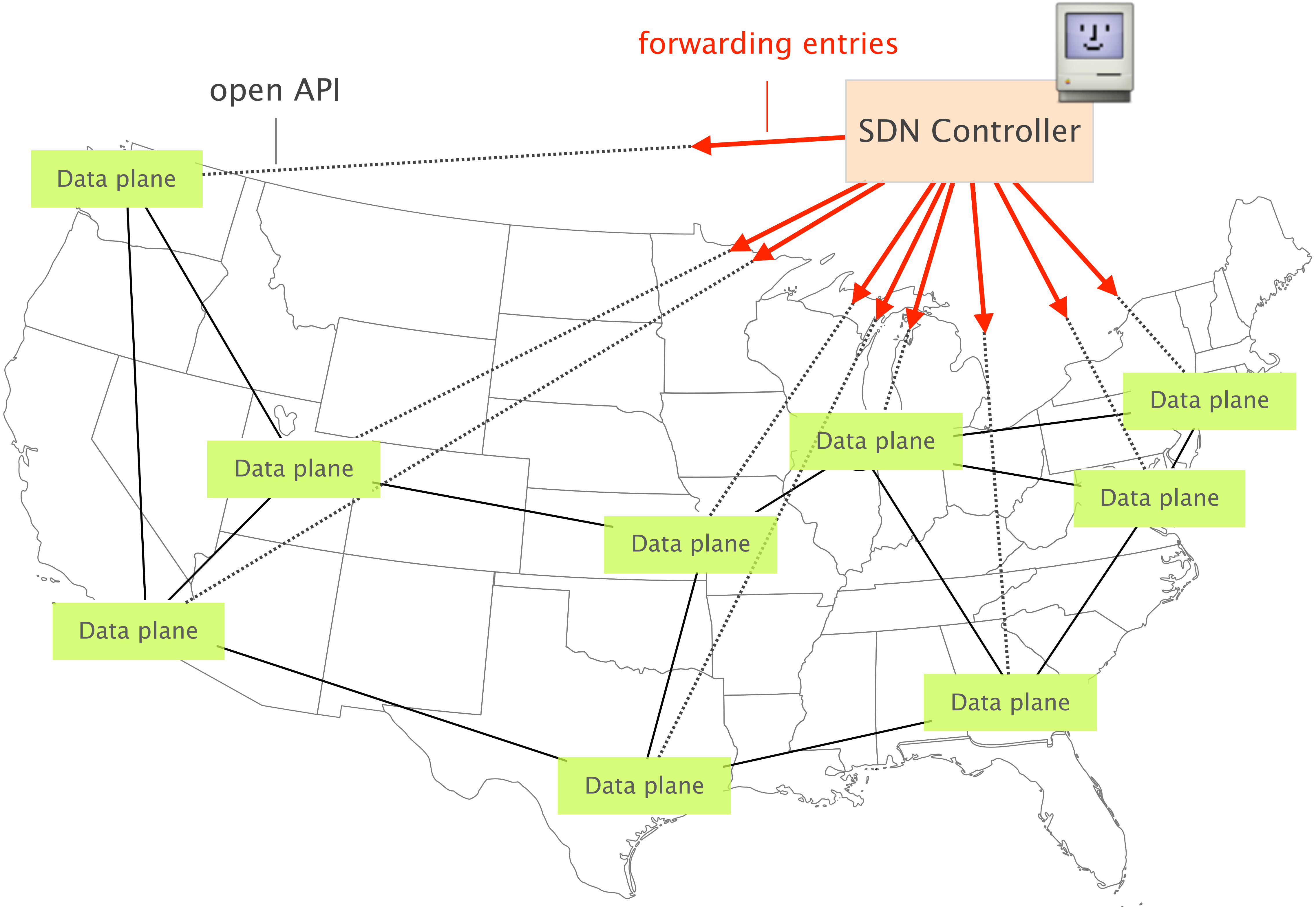
… by removing the intelligence from the equipments



… and **centralizing it** in a SDN controller that can run arbitrary programs



The SDN controller programs forwarding state in the devices using an open API (e.g., OpenFlow)



Sounds great

Sounds great, but…

How do you go from a traditional network to a SDN-enabled one?

Well… not easily

Deploying SDN requires to upgrade network …

- devices
- management systems
- operators

challenging, time-consuming and therefore **costly**

To succeed, SDN-based technologies should possess at least 3 characteristics

Small investment

Low risk

High return

To succeed, SDN-based technologies should possess at least 3 characteristics

Small investment

provide benefits
under partial deployment
(ideally, with a single switch)

Low risk

High return

To succeed, SDN-based technologies should possess at least 3 characteristics

Small investment

Low risk

- require minimum changes to operational practices
- be compatible with existing technologies

High return

To succeed, SDN-based technologies should possess at least 3 characteristics

Small investment

Low risk

High return

solve a timely problem

This talk is about two such SDN-based technologies

Fibbing

improved flexibility

Supercharged

performance boost

Fibbing

improved flexibility

Supercharged

performance boost

central control over distributed system

Fibbing

improved flexibility

Supercharged

performance boost

reduce convergence time
by 1000x

Fibbing

improved flexibility

central control over distributed system

Supercharged

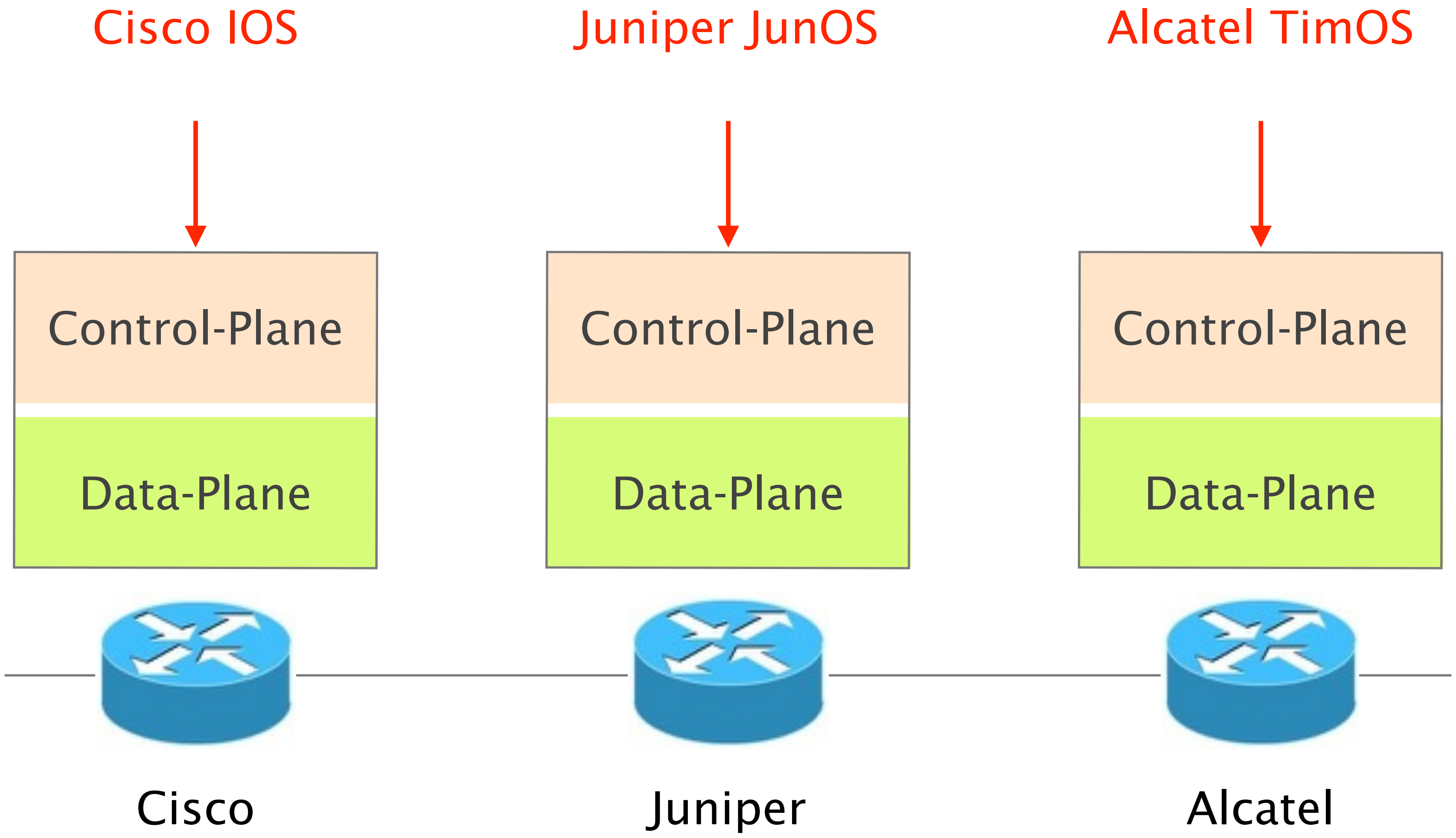
performance boost

Wouldn’t it be great to manage
an existing network “à la SDN”?

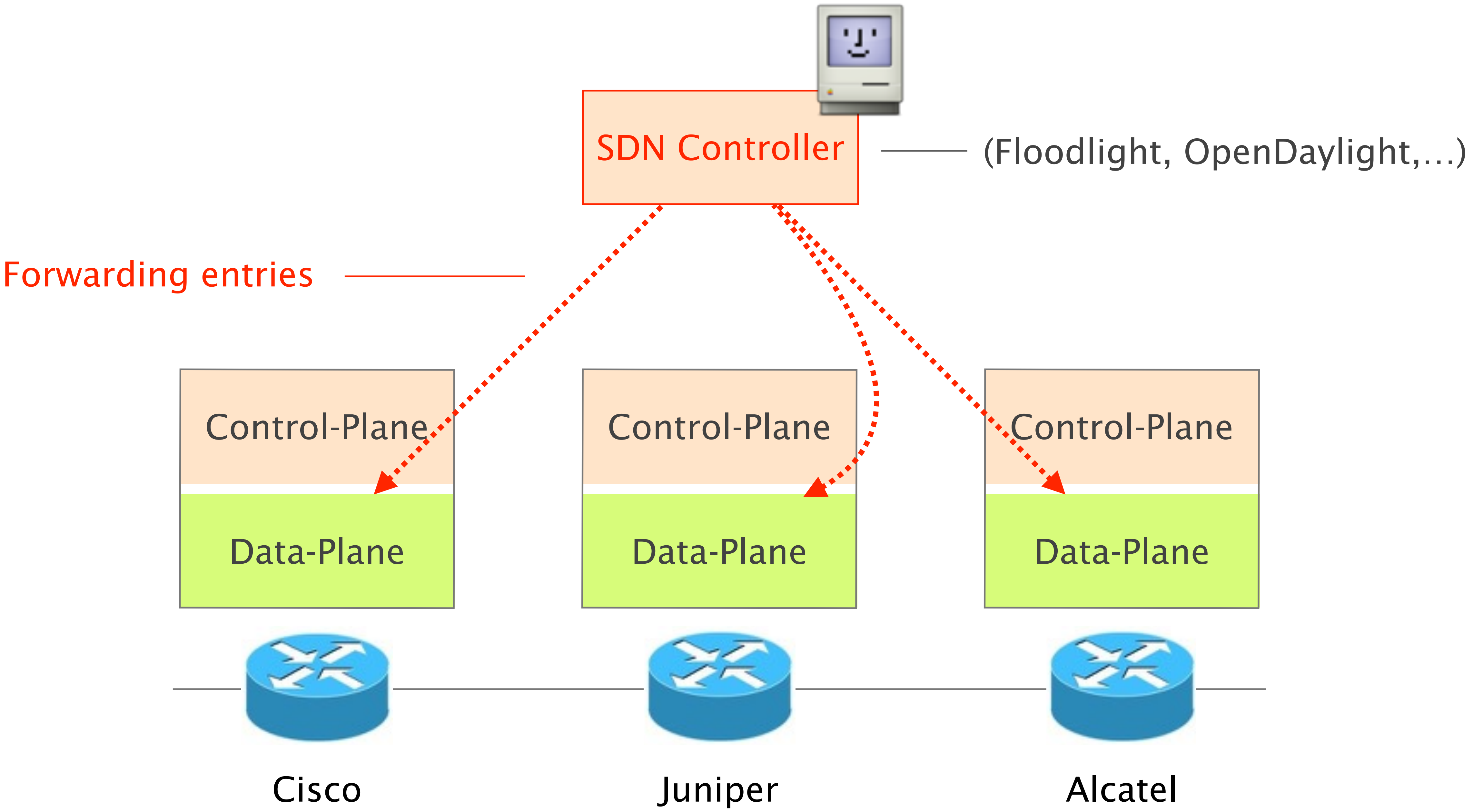
Wouldn’t it be great to manage an existing network “à la SDN”?

what does it mean?

Instead of configuring a network using configuration "languages" …



… program it from a central SDN controller



SDN Controller

(Floodlight, OpenDaylight,…)

Forwarding entries

Control-Plane

Data-Plane

Control-Plane

Data-Plane

Control-Plane

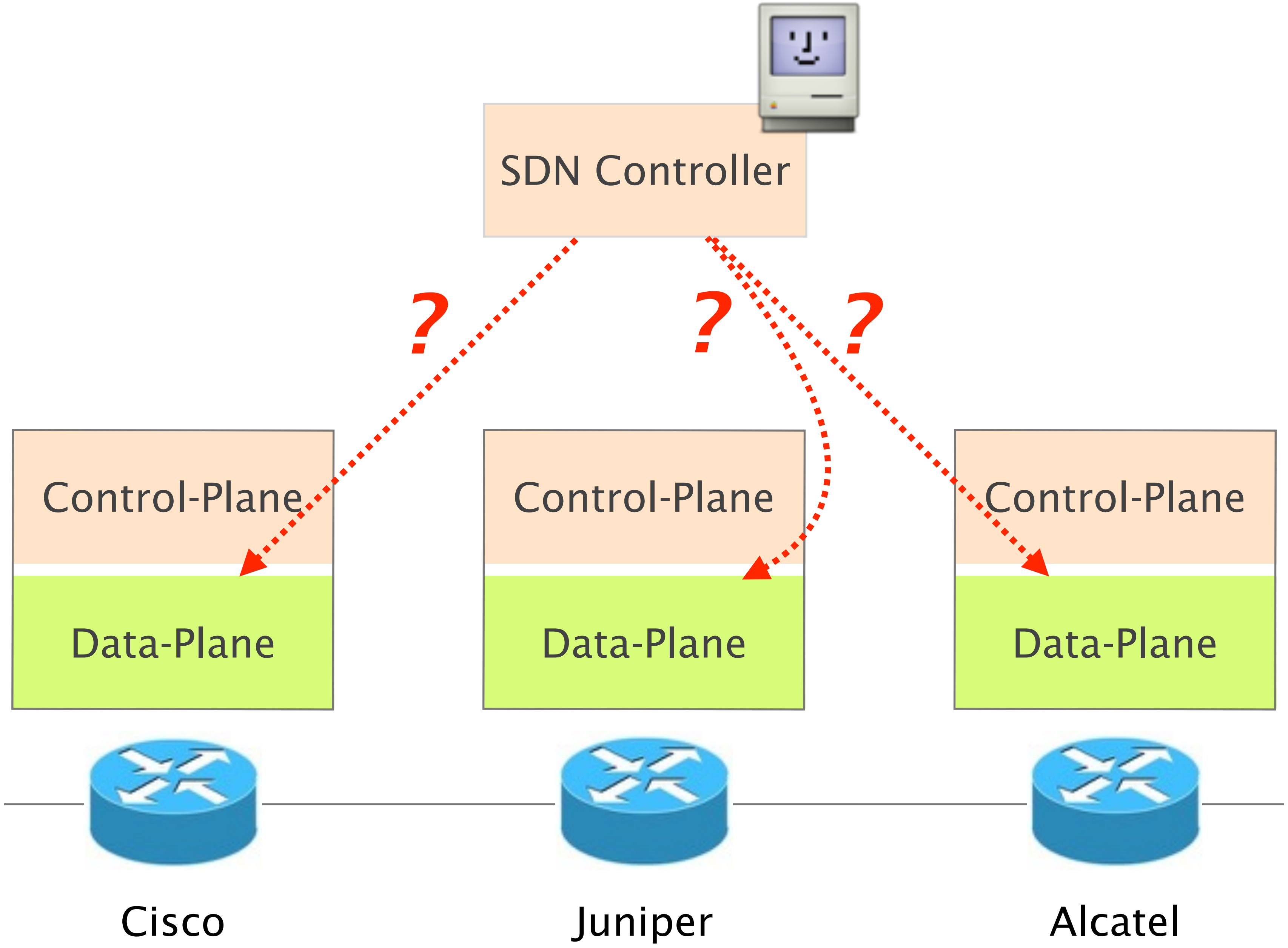
Data-Plane

Cisco

Juniper

Alcatel

For that, we need an API
that *any* router can understand



Routing protocols are perfect candidates to act as such API

- messages are standardized

  routers must speak the same language

- behaviors are well-defined

  e.g., shortest-path routing

- implementations are widely available

  nearly all routers support OSPF

@SIGCOMM'15

Fibbing

Fibbing

= lying

@SIGCOMM'15

Fibbing

to **control** router's forwarding table

Central Control Over Distributed Routing

Joint work with: Stefano Vissicchio, Olivier Tilmans and Jennifer Rexford

1 **Fibbing**

lying made useful

2 **Expressivity**

any path, anywhere

3 **Scalability**

1 lie is better than 2

Central Control Over Distributed Routing

1 **Fibbing**

lying made useful

Expressivity

any path, anywhere

Scalability

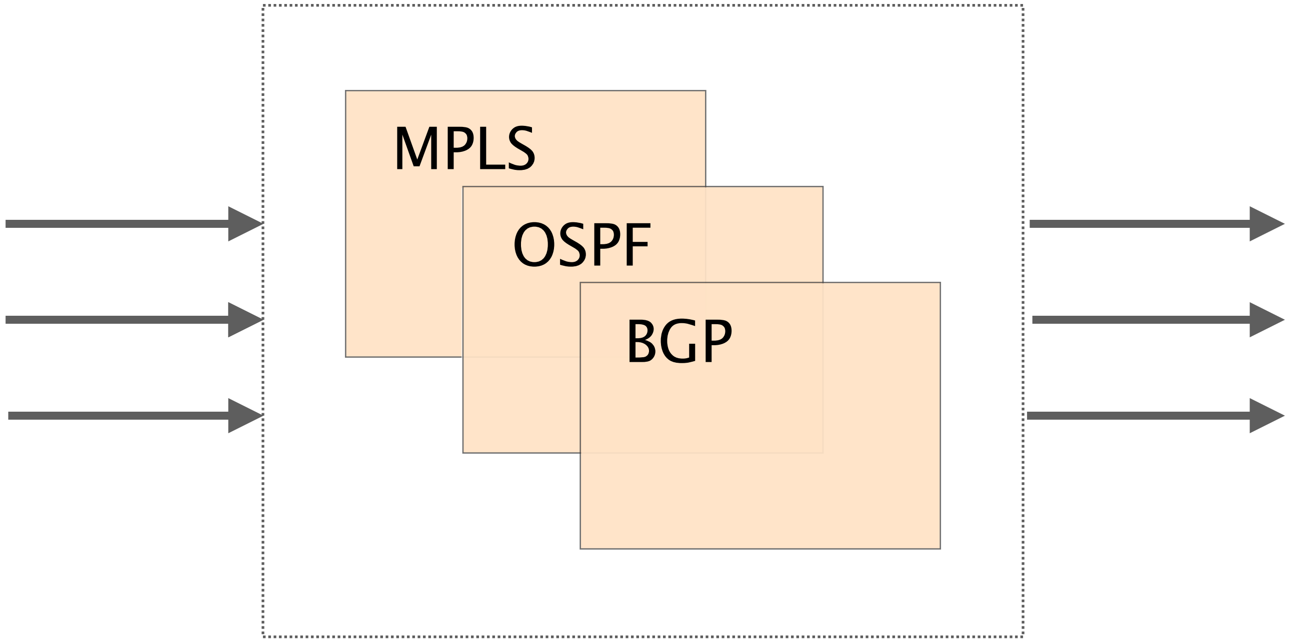
1 lie is better than 2

A router implements a function from **routing messages** to **forwarding paths**

input

function

output



Routing

Messages

MPLS

OSPF

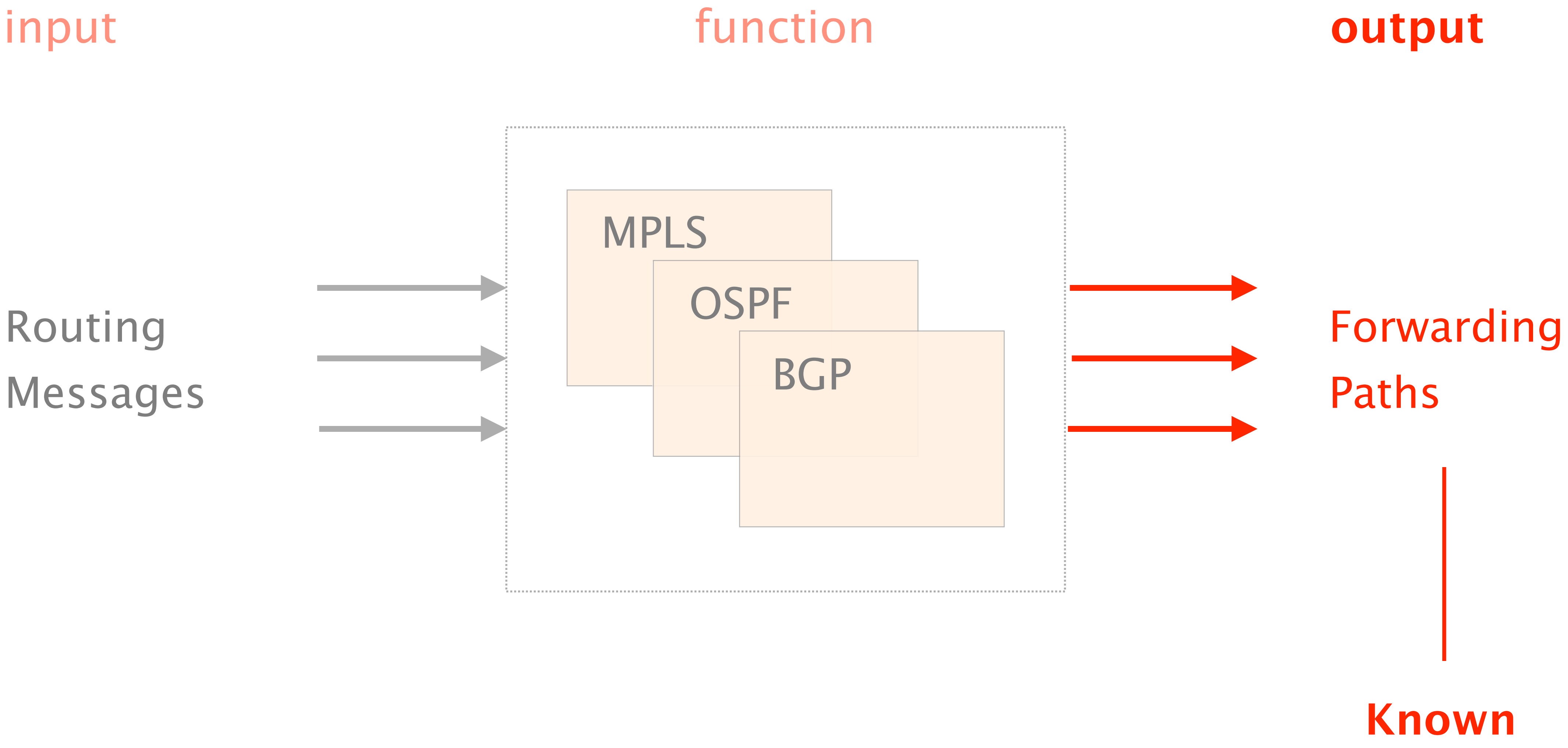
BGP

IP router

Forwarding

Paths

The forwarding paths are known, provided by the operators or by the controller



input

function

output

Routing Messages

MPLS

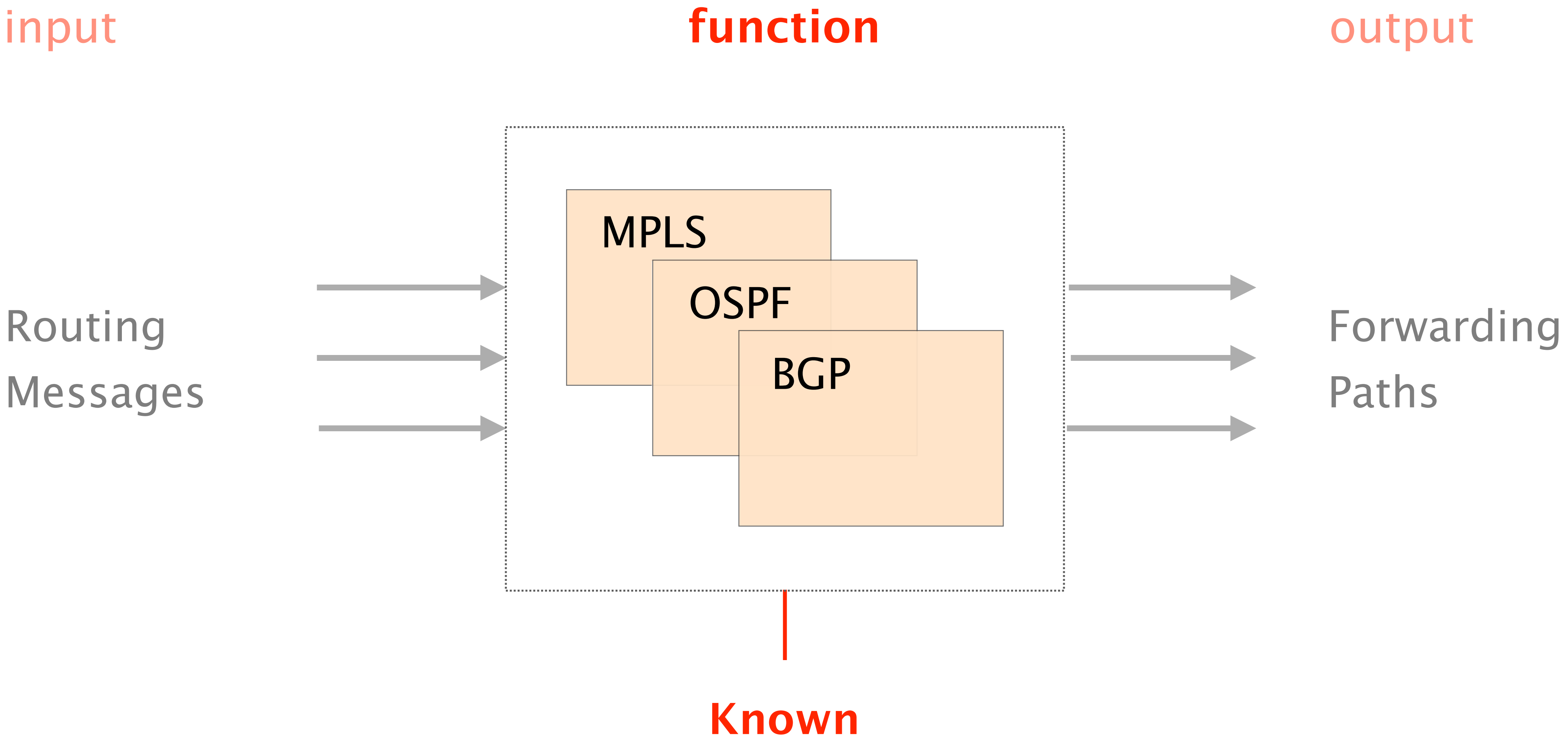
OSPF

BGP

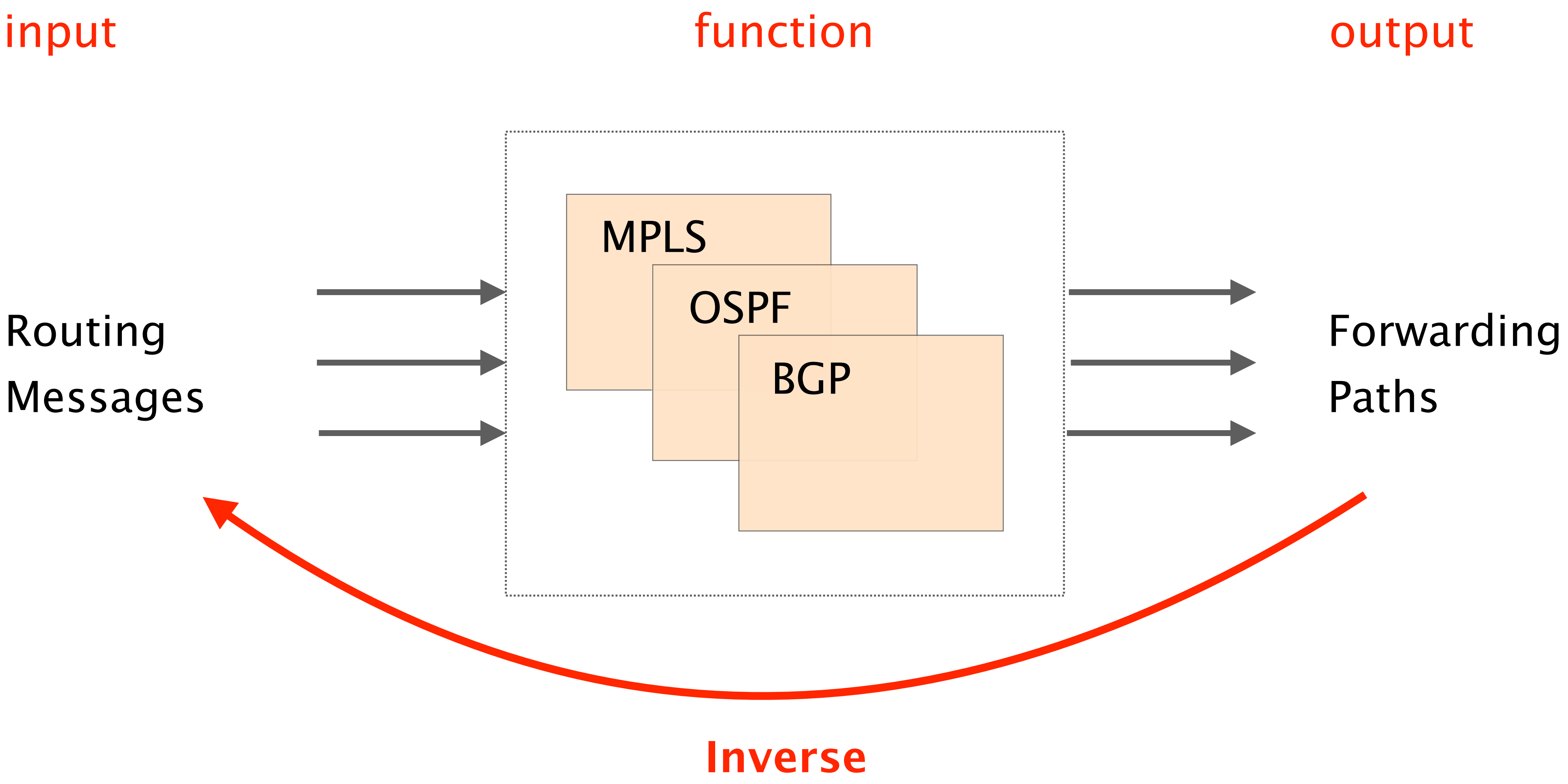
Forwarding Paths

Known

The function is known, from the protocols' specification & the configuration



Given a path and a function, our framework computes corresponding routing messages by inverting the function



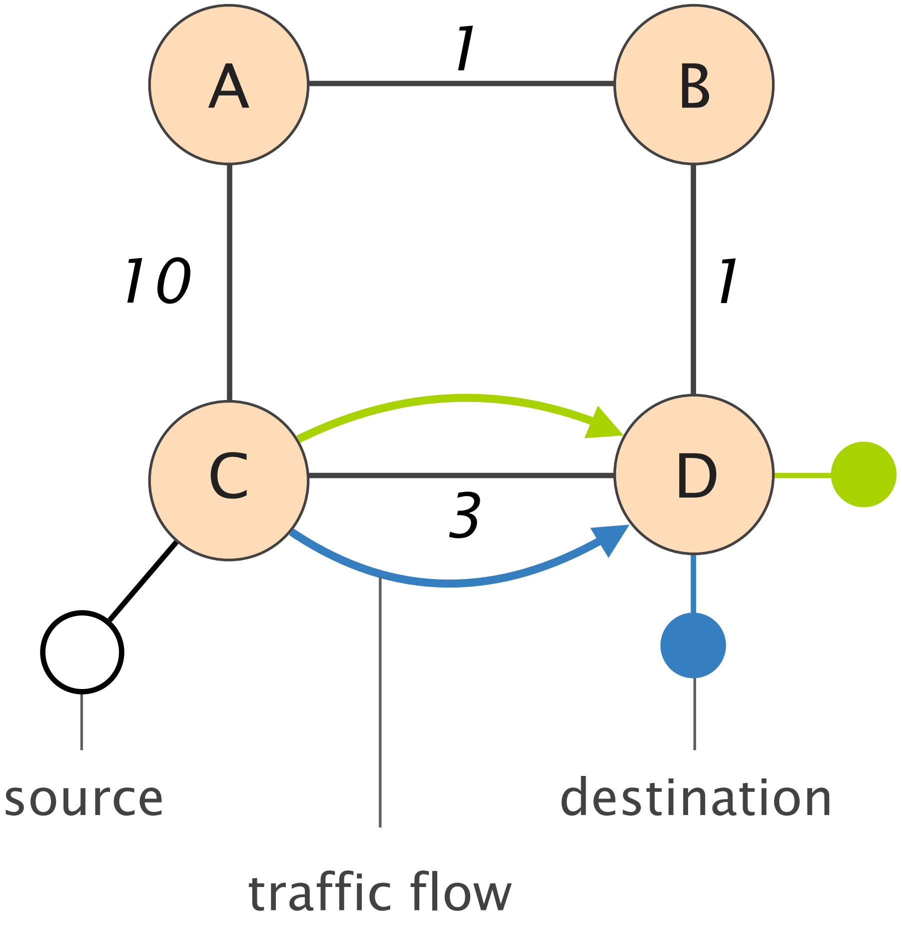
The type of input to be computed depends on the routing protocol

Protocol	Family	Algorithm/ Function	Router Input
IGP	Link-State	Dijkstra	Network graph
BGP	Path-Vector	Decision process	Routing paths

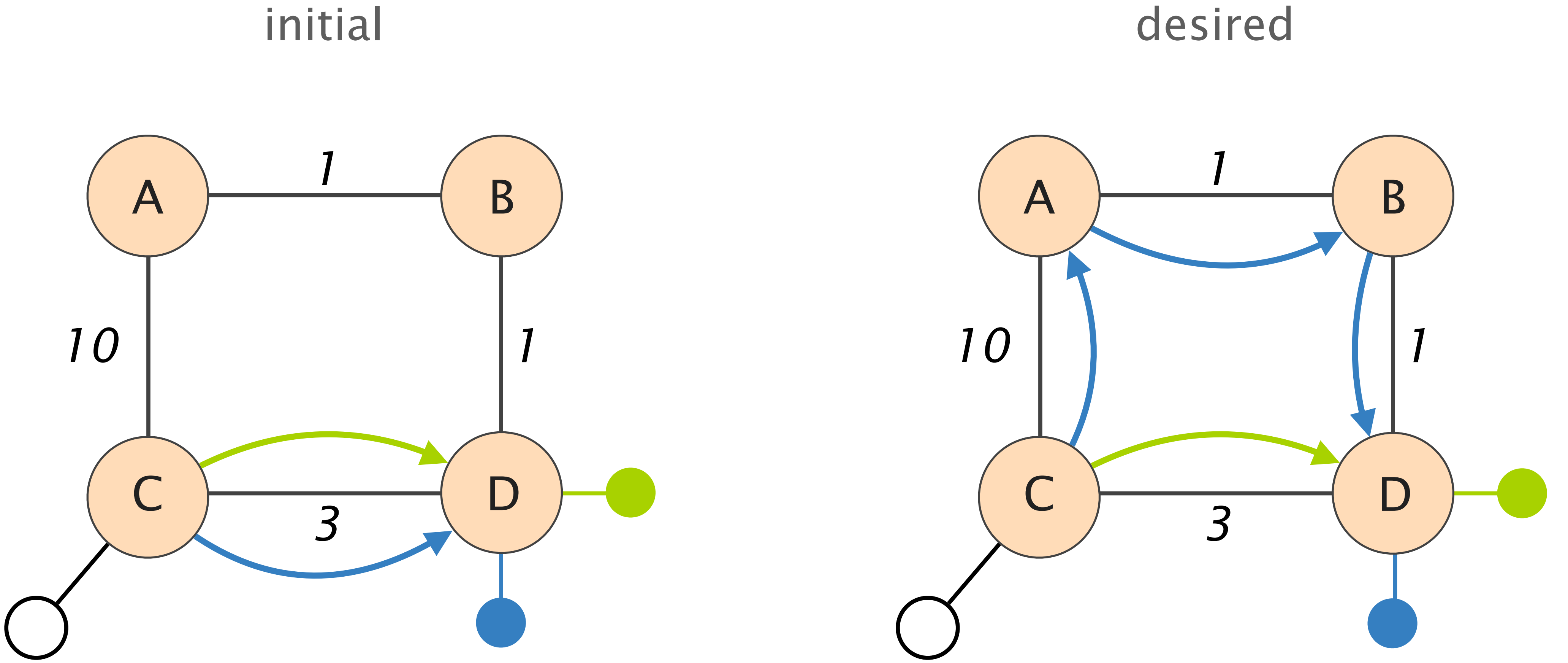
We focus on routers running link-state protocols that take the network graph as input and run Dijkstra

Protocol	Family	Algorithm/ Function	Router Input
IGP	Link-State	Dijkstra	Network graph
BGP	Path-Vector	Decision process	Routing paths

Consider this network where a source sends traffic to 2 destinations

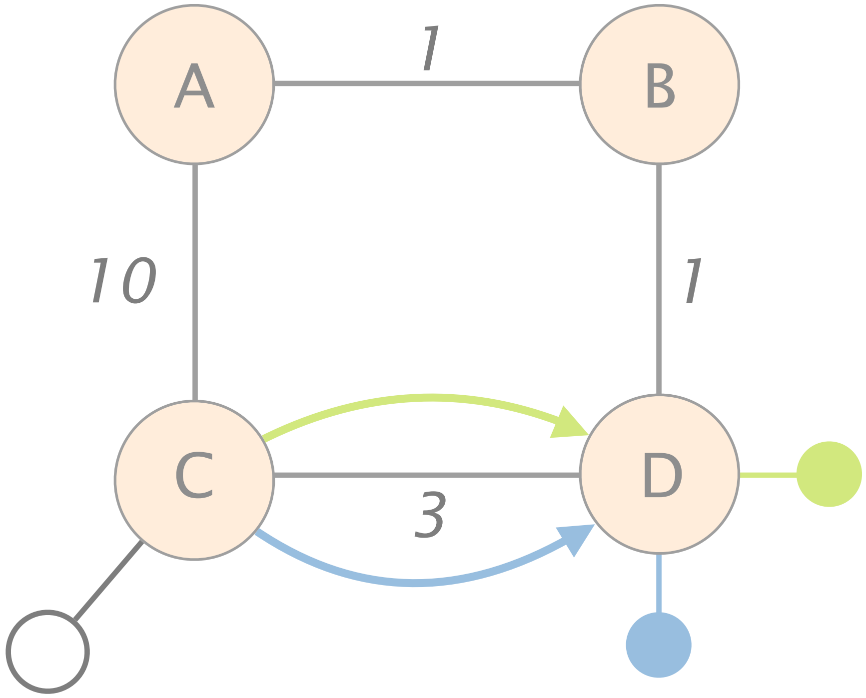


As congestion appears, the operator wants to shift away one flow from (C,D)

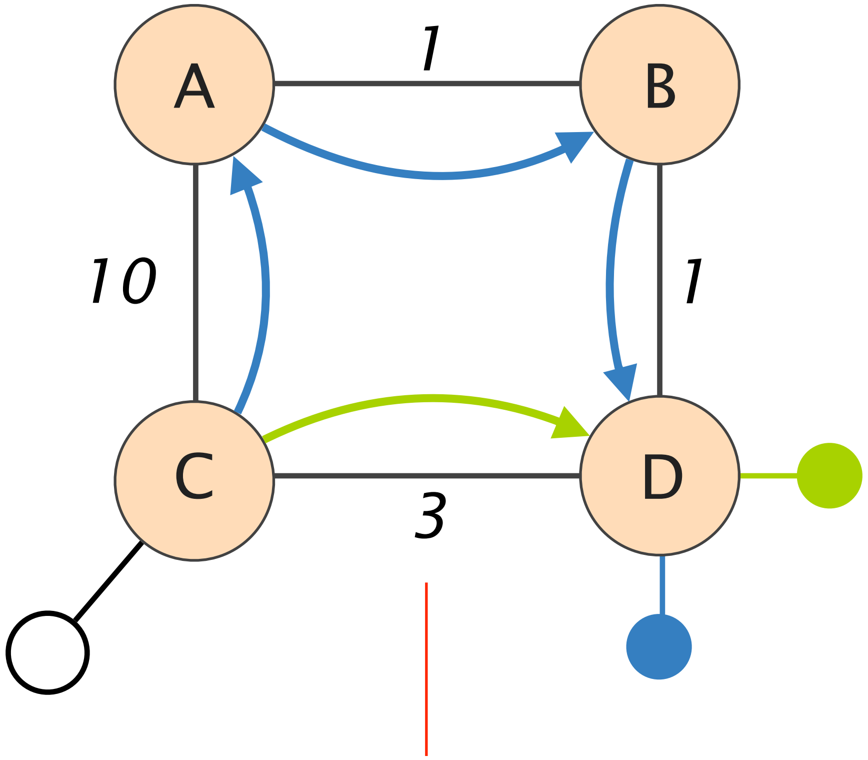


Moving only one flow is impossible though as both destinations are connected to D

initial

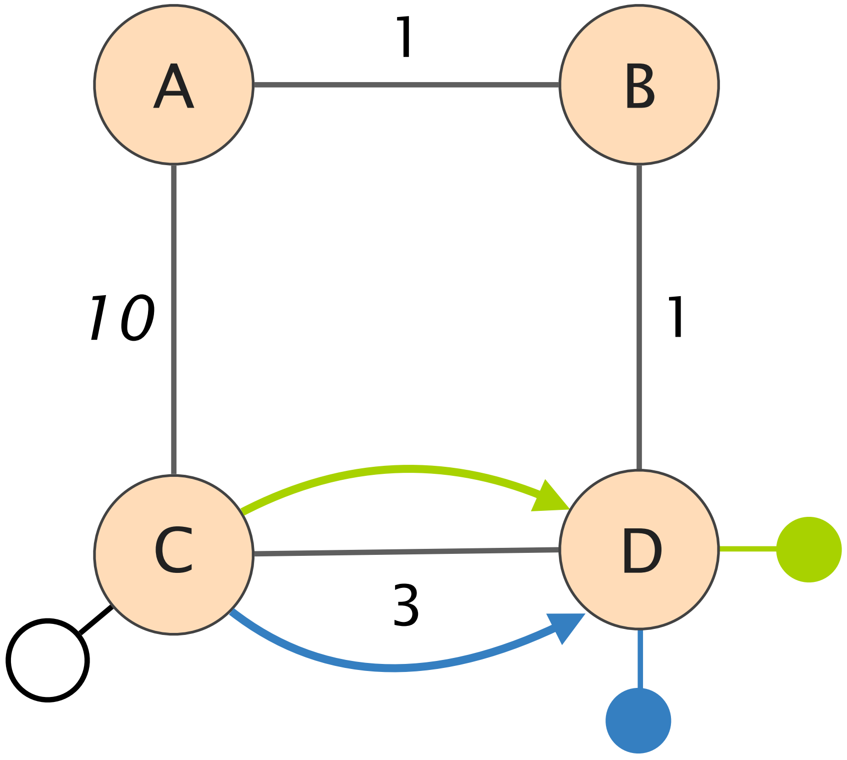


desired

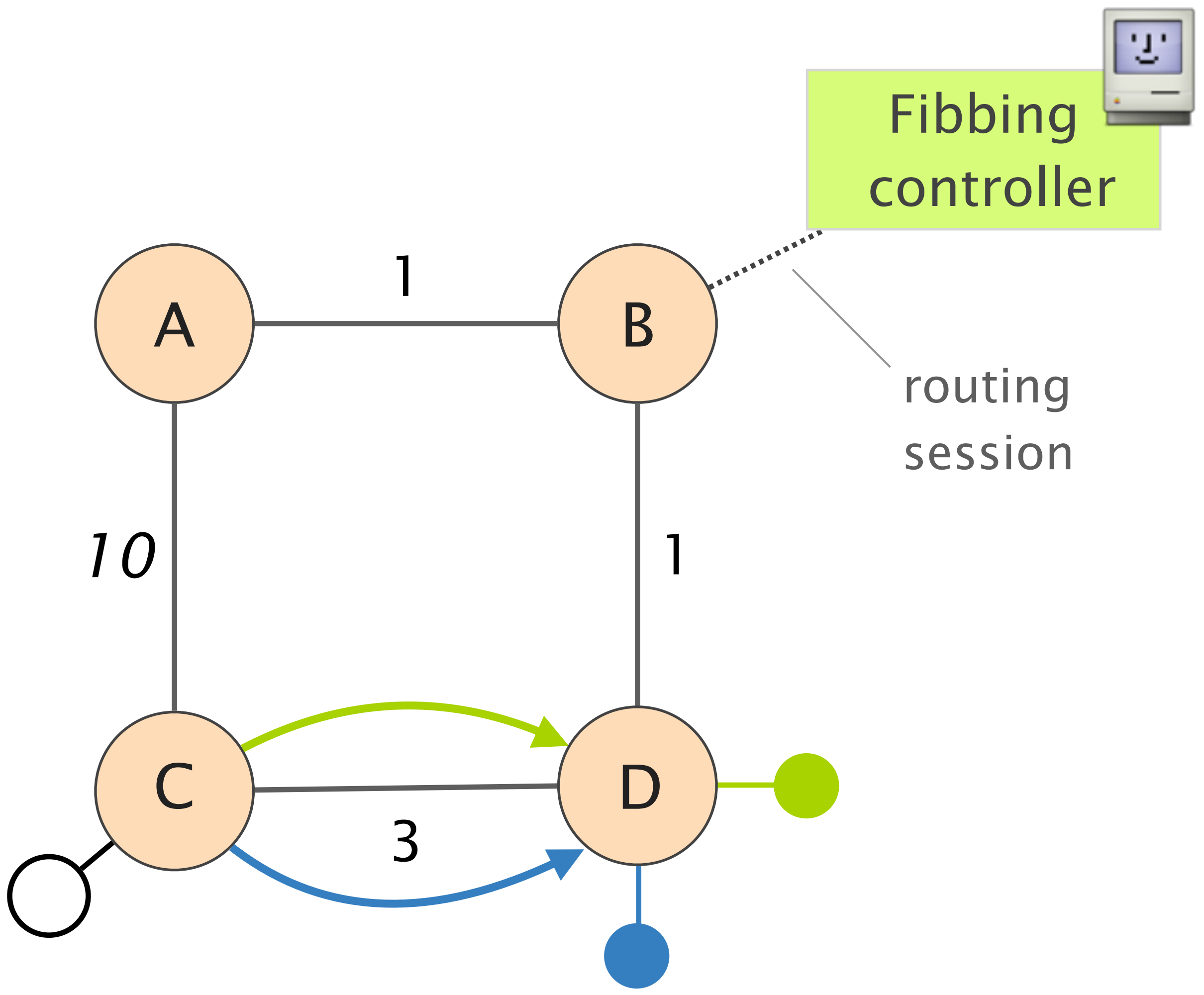


impossible to achieve by reweighing the links

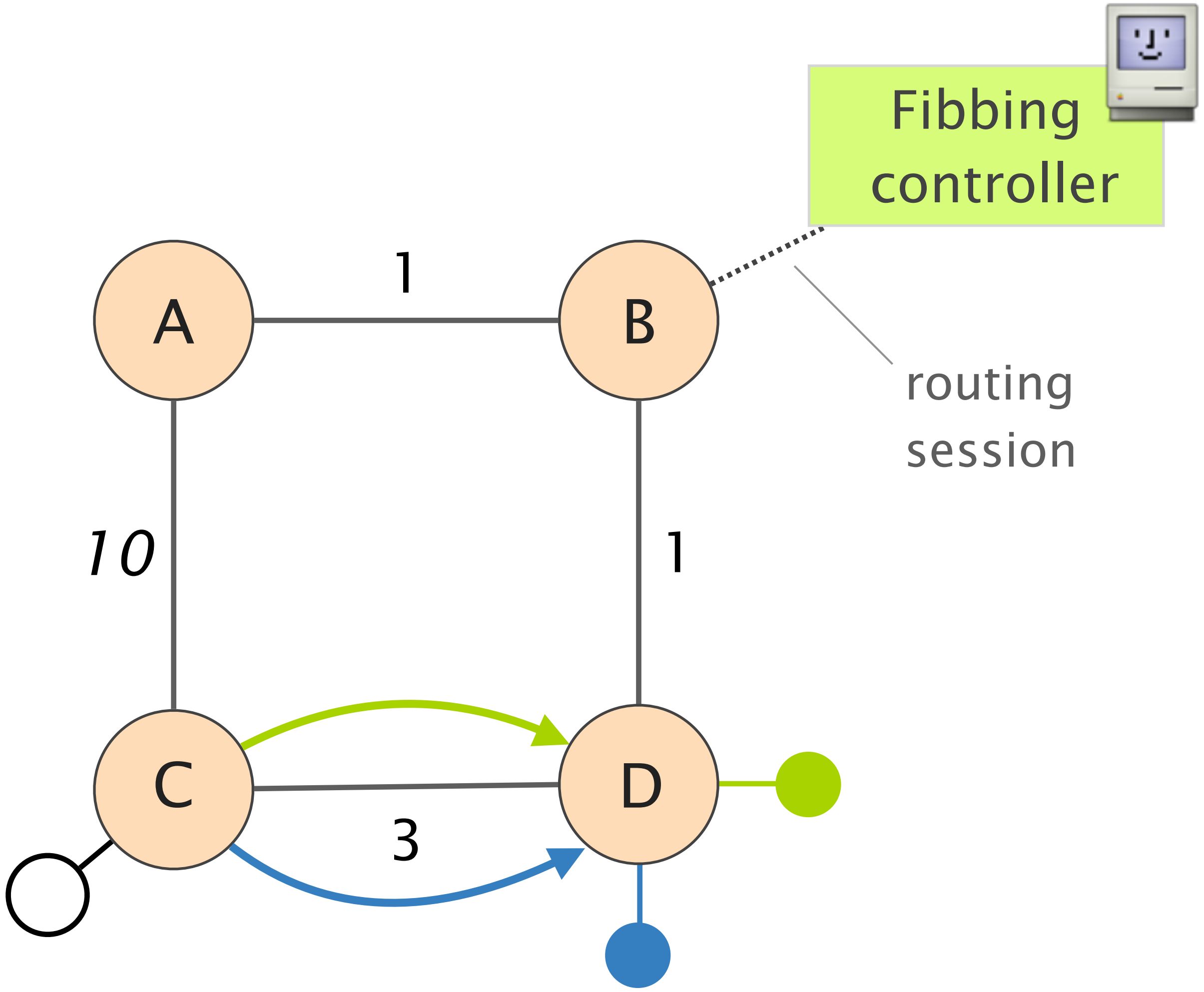
Let’s lie to the router



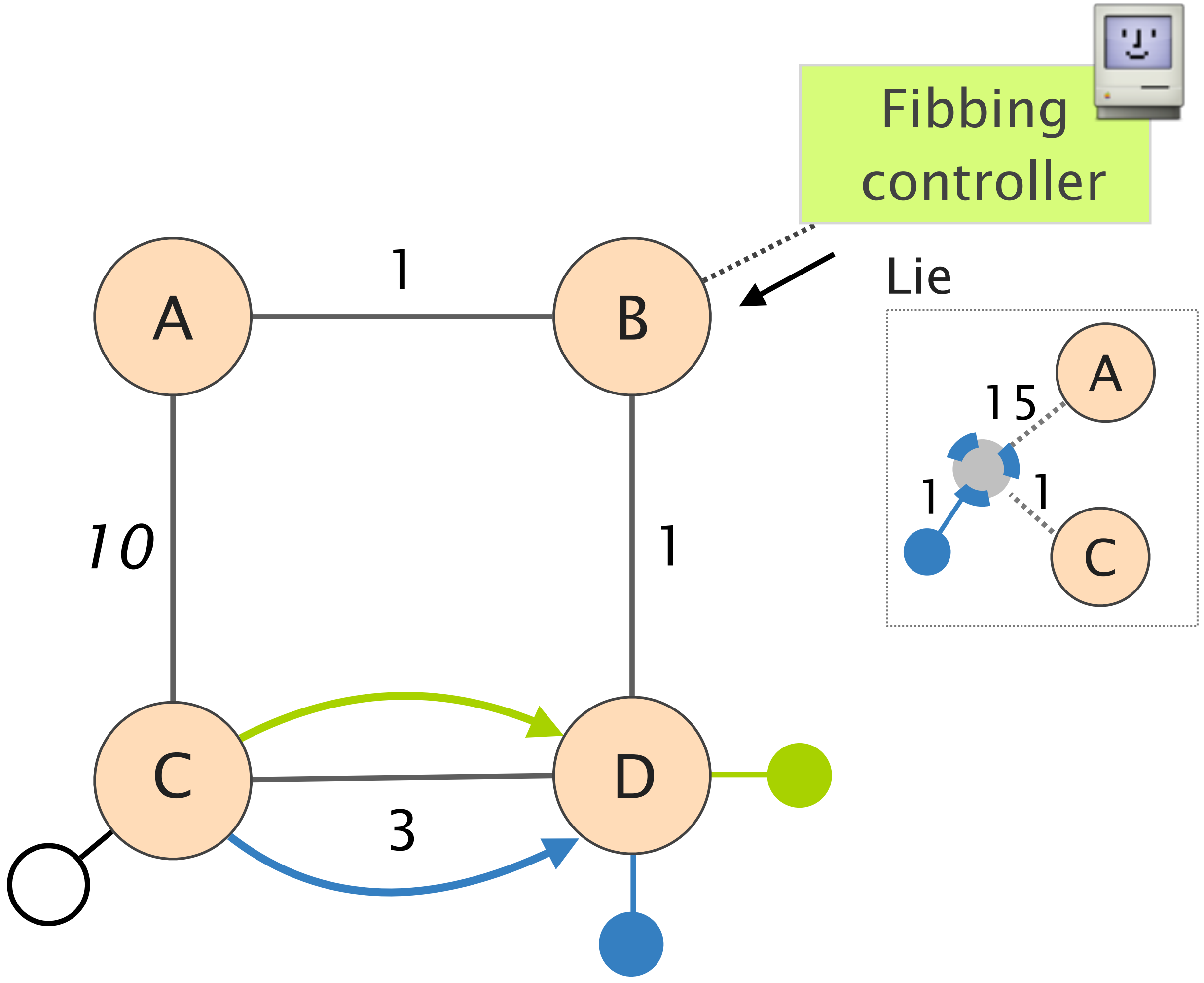
Let’s lie to the router



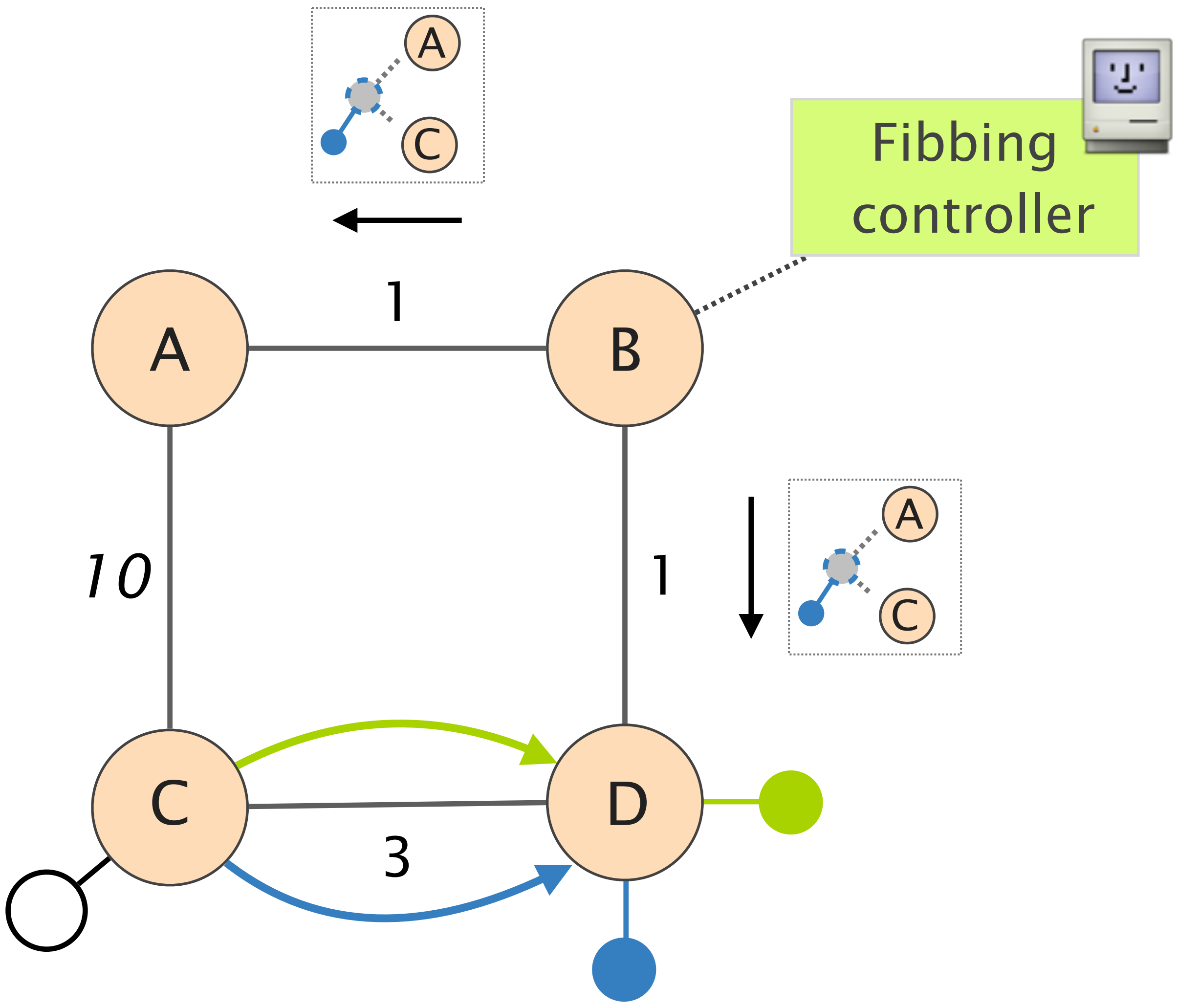
Let's **lie to the router,** by injecting fake nodes, links and destinations



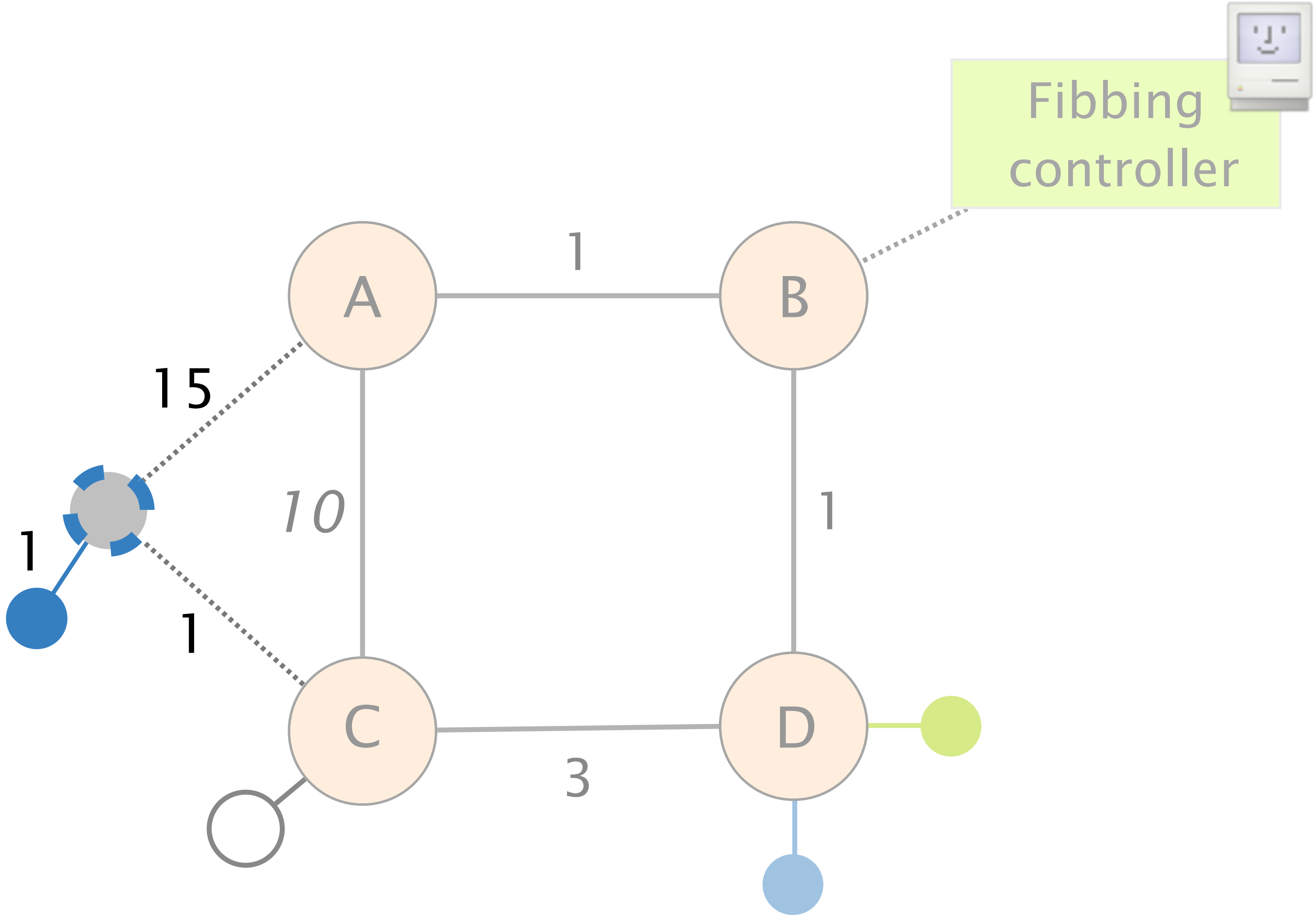
Let's **lie to the router,** by injecting fake nodes, links and destinations



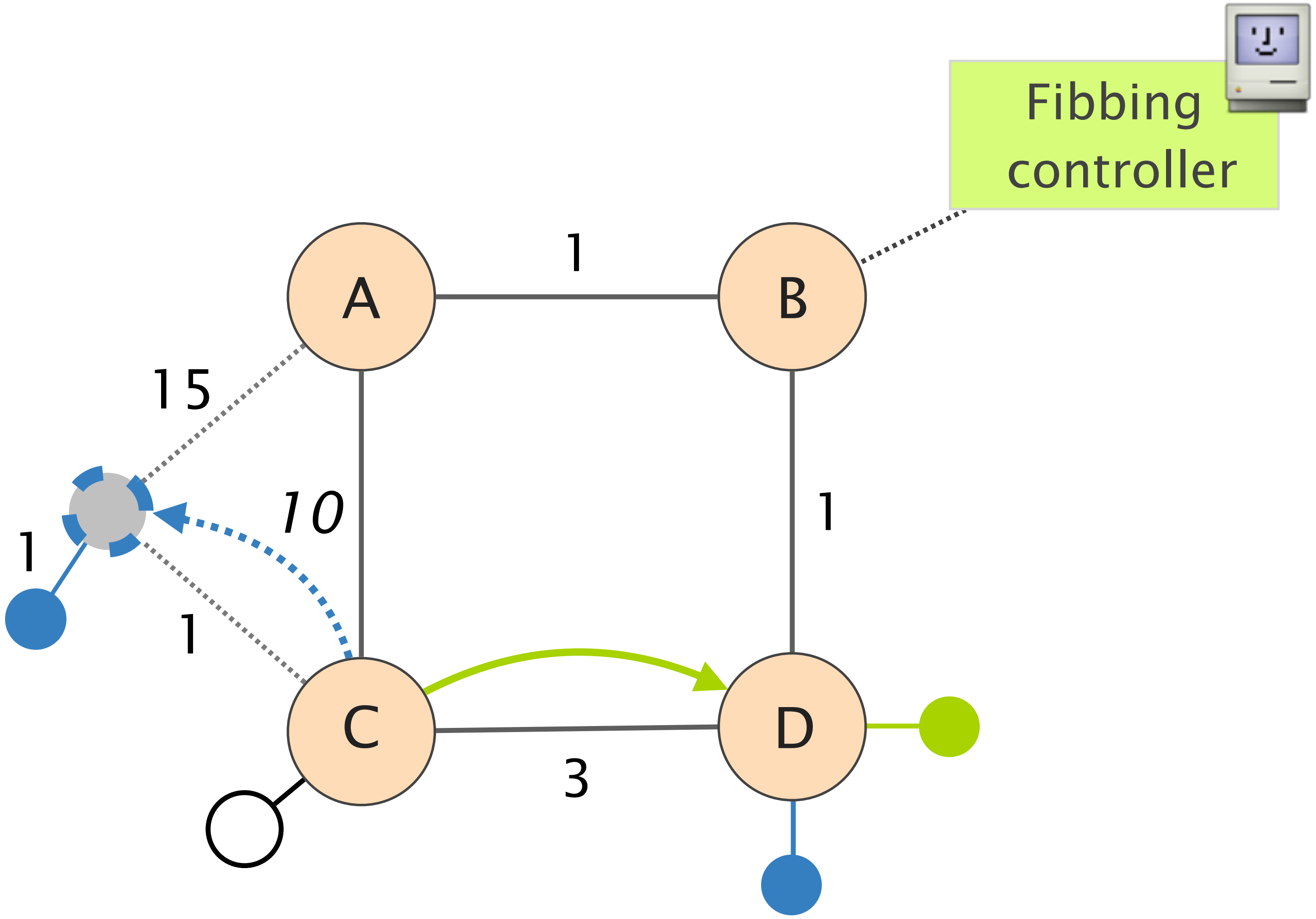
Lies are propagated network-wide by the protocol



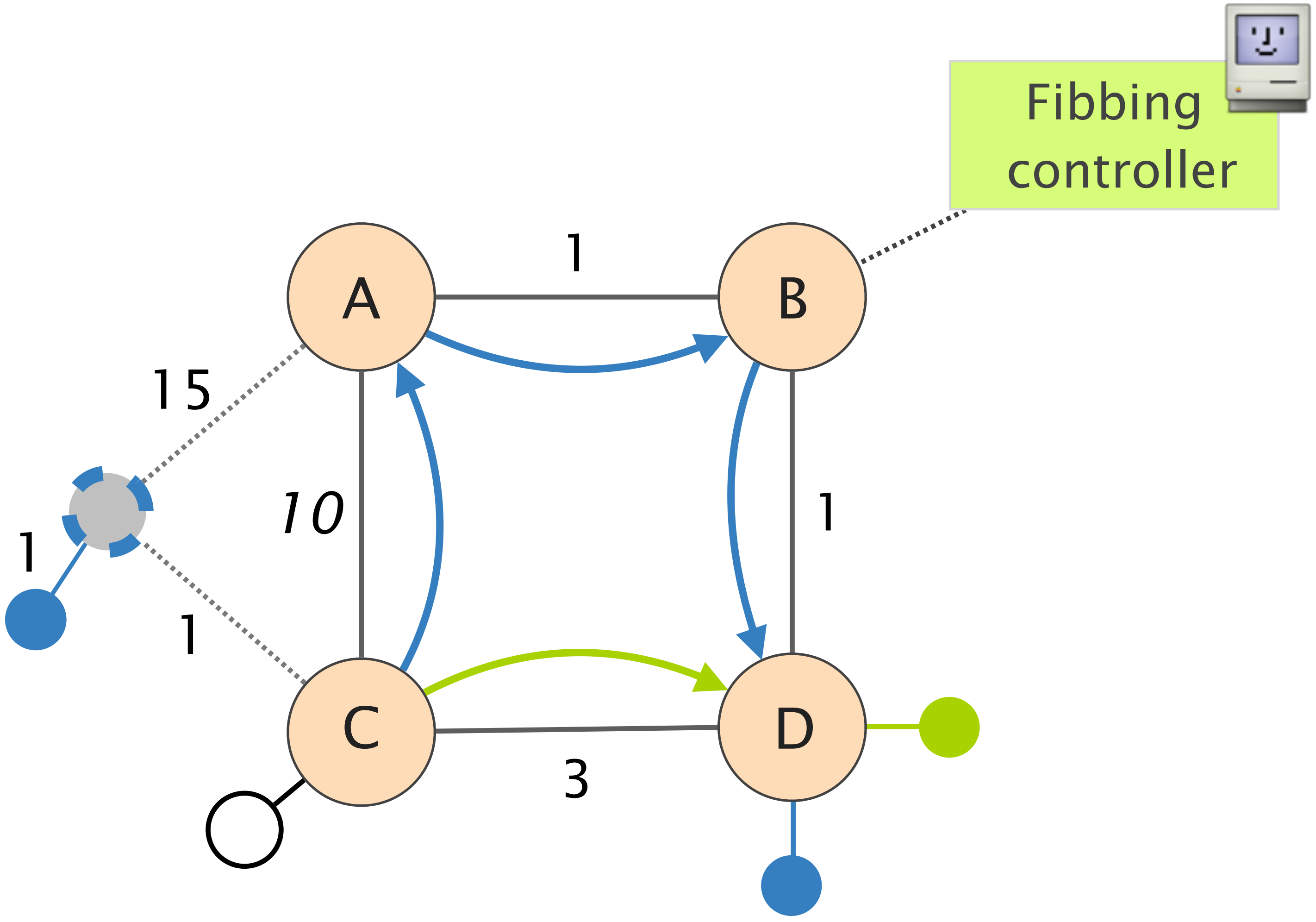
After the injection, this is the topology seen by all routers, on which they compute Dijkstra



Now, C prefers the virtual node (cost 2) to reach the blue destination…



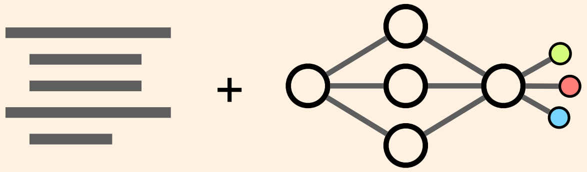
As the virtual node does not really exist,
actual traffic is *physically* sent to A



Fibbing

workflow

Fibbing starts from the operators requirements and a up-to-date representation of the network

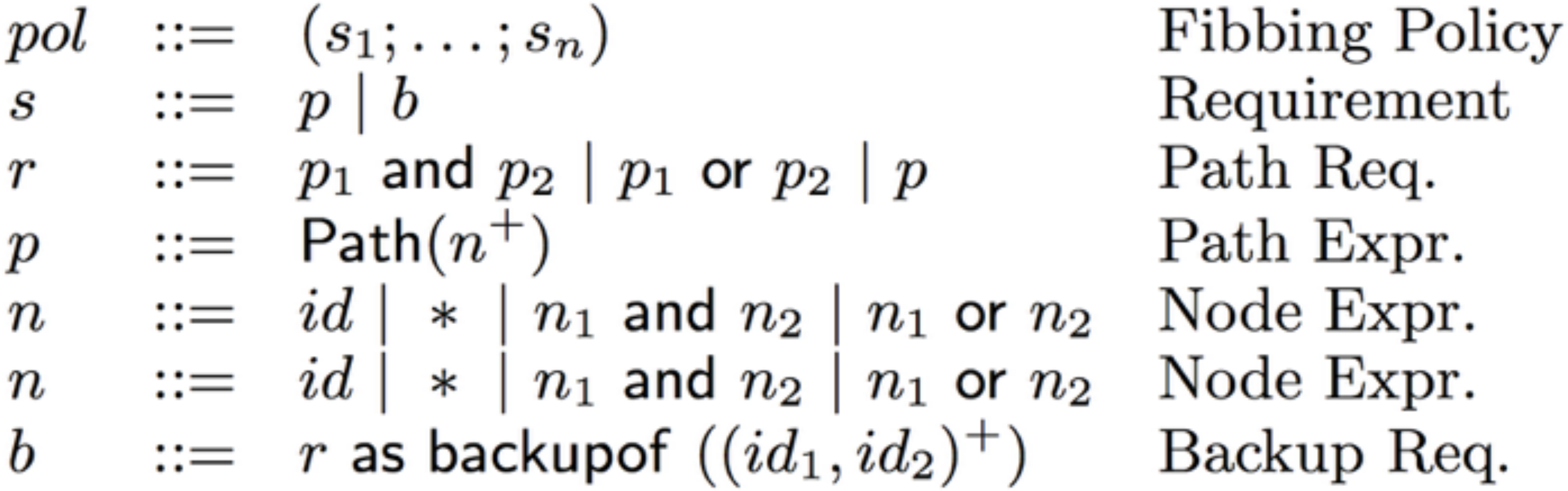


path
reqs.

network
graph

Operators requirements are expressed in a high-level language

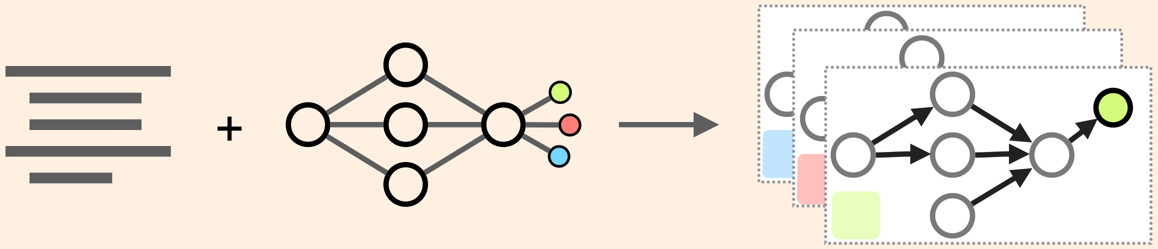
Syntax of Fibbing's path requirements language



pol	$::=$	$(s_1; \dots ; s_n)$	Fibbing Policy
s	$::=$	$p \mid b$	Requirement
r	$::=$	p_1 and $p_2 \mid p_1$ or $p_2 \mid p$	Path Req.
p	$::=$	Path(n^+)	Path Expr.
n	$::=$	$id \mid * \mid n_1$ and $n_2 \mid n_1$ or n_2	Node Expr.
n	$::=$	$id \mid * \mid n_1$ and $n_2 \mid n_1$ or n_2	Node Expr.
b	$::=$	r as backupof $((id_1, id_2)^+)$	Backup Req.

Out of these,
the **compilation** stage produces DAGs

Compilation

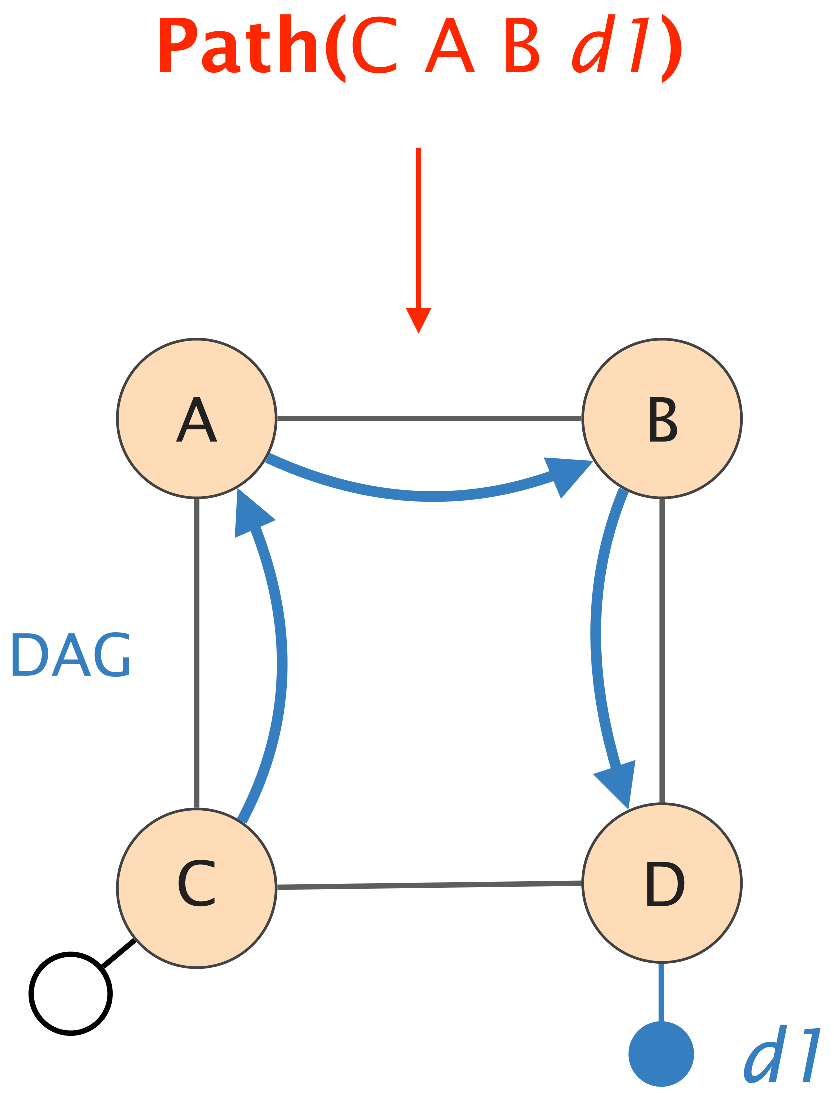


path
reqs.

network
graph

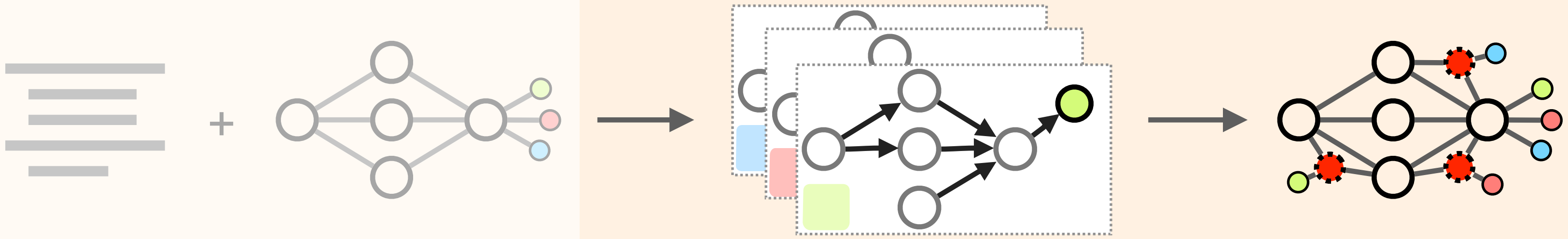
forwarding
DAGs

Forwarding graphs (DAGs) are compiled from high-level requirements



The **augmentation** stage augments the network graph with lies to implement each DAG

Augmentation



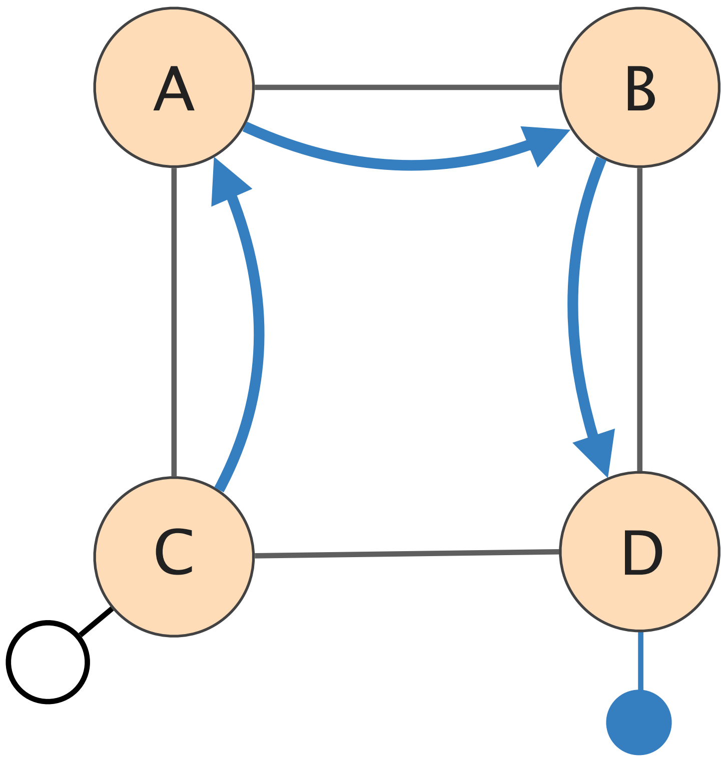
path reqs.

network graph

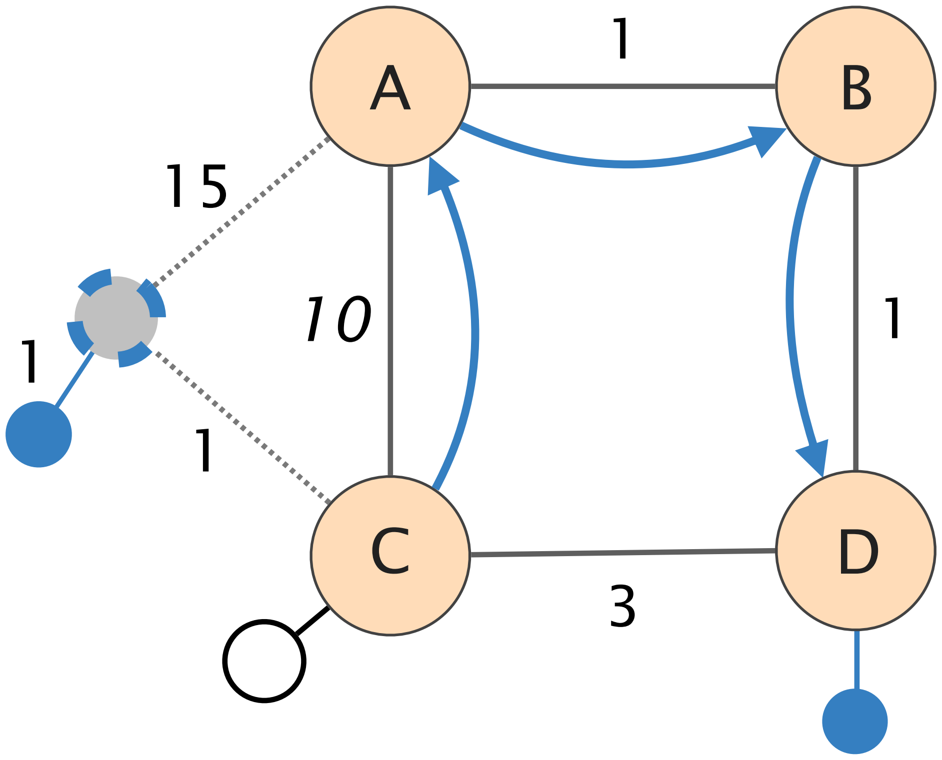
forwarding DAGs

augmented graph

The **augmentation** stage augments the network graph with lies to implement each DAG



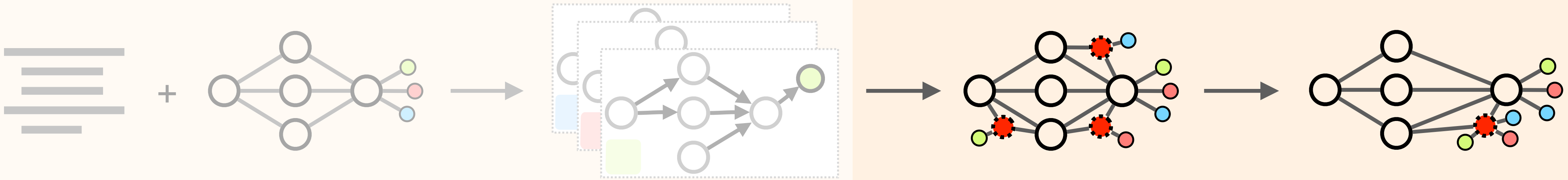
Compilation output



Augmentation output

The **optimization** stage reduces the amount of lies necessary

Optimization



path reqs.

network graph

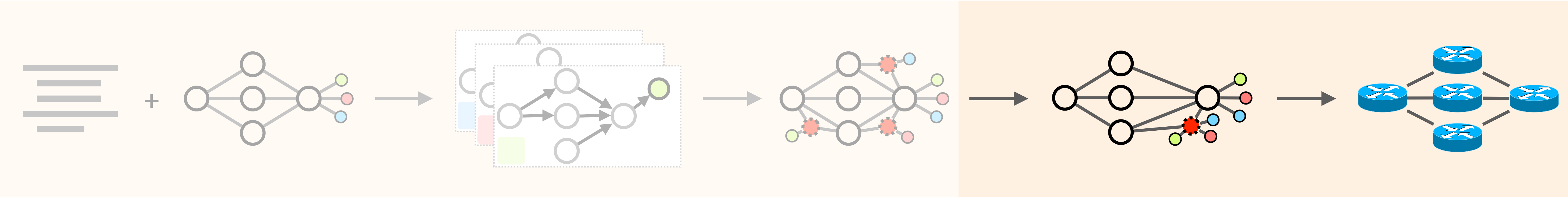
forwarding DAGs

augmented graph

reduced graph

The **injection** stage injects the lies in the production network

Injection



path reqs. + network graph → forwarding DAGs → augmented graph → reduced graph → running network

Central Control Over Distributed Routing

Fibbing

lying made useful

2 Expressivity

any path, anywhere

Scalability

1 lie is better than 2

Fibbing is powerful

Fibbing is powerful

Theorem

Fibbing can program

any set of non-contradictory paths

Fibbing is powerful

Theorem

Fibbing can program
any set of non-contradictory paths

Fibbing is powerful

Theorem

Fibbing can program
any set of non-contradictory paths

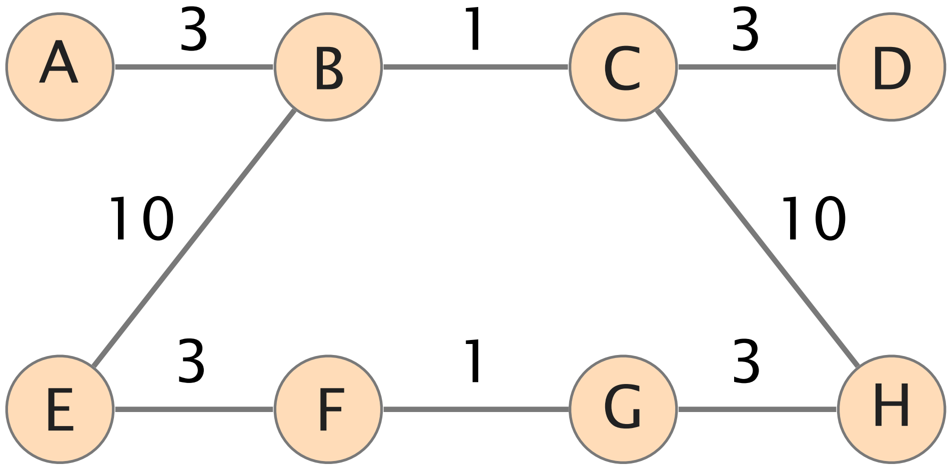
- any path is loop-free

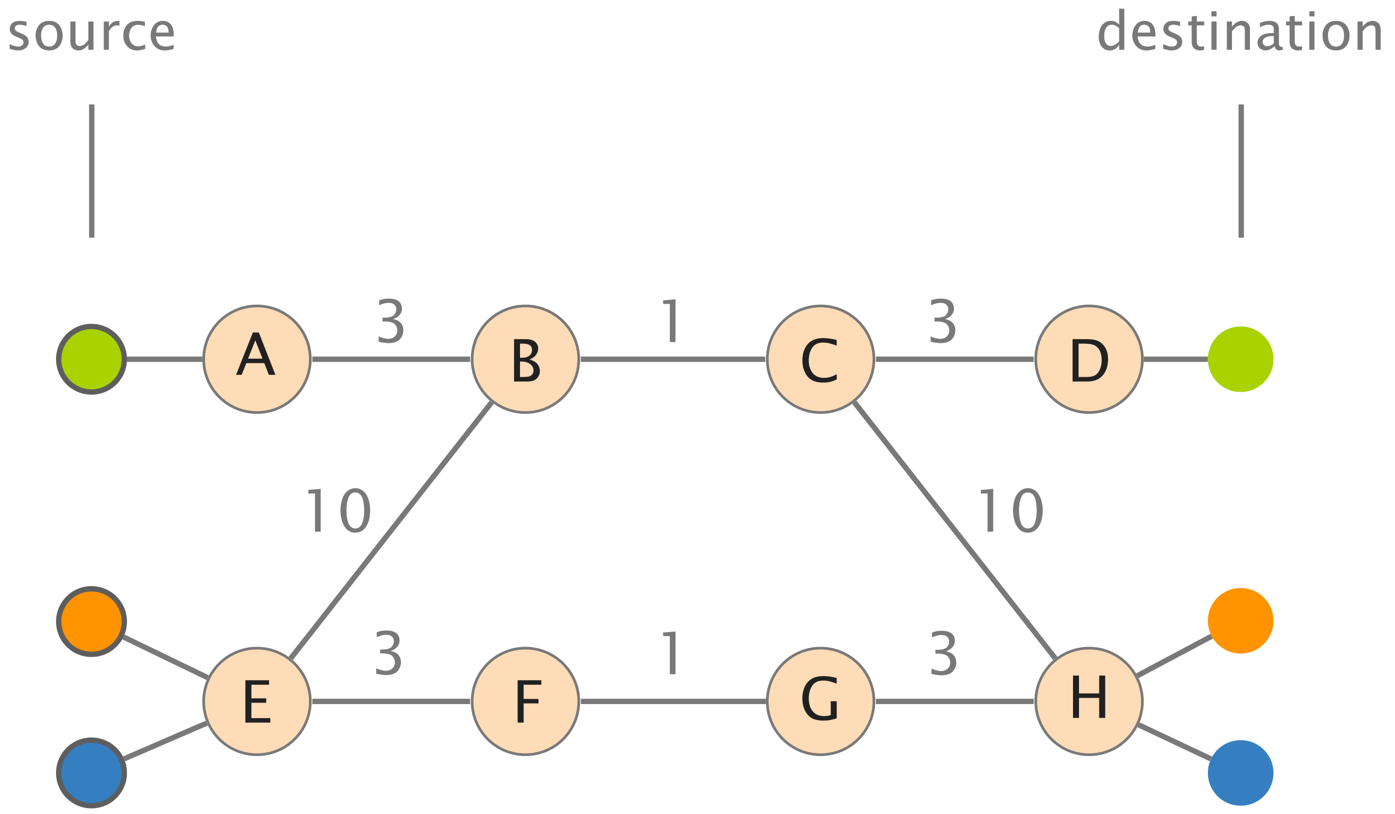
  (*e.g.*, [s1, a, b, a, d] is not possible)

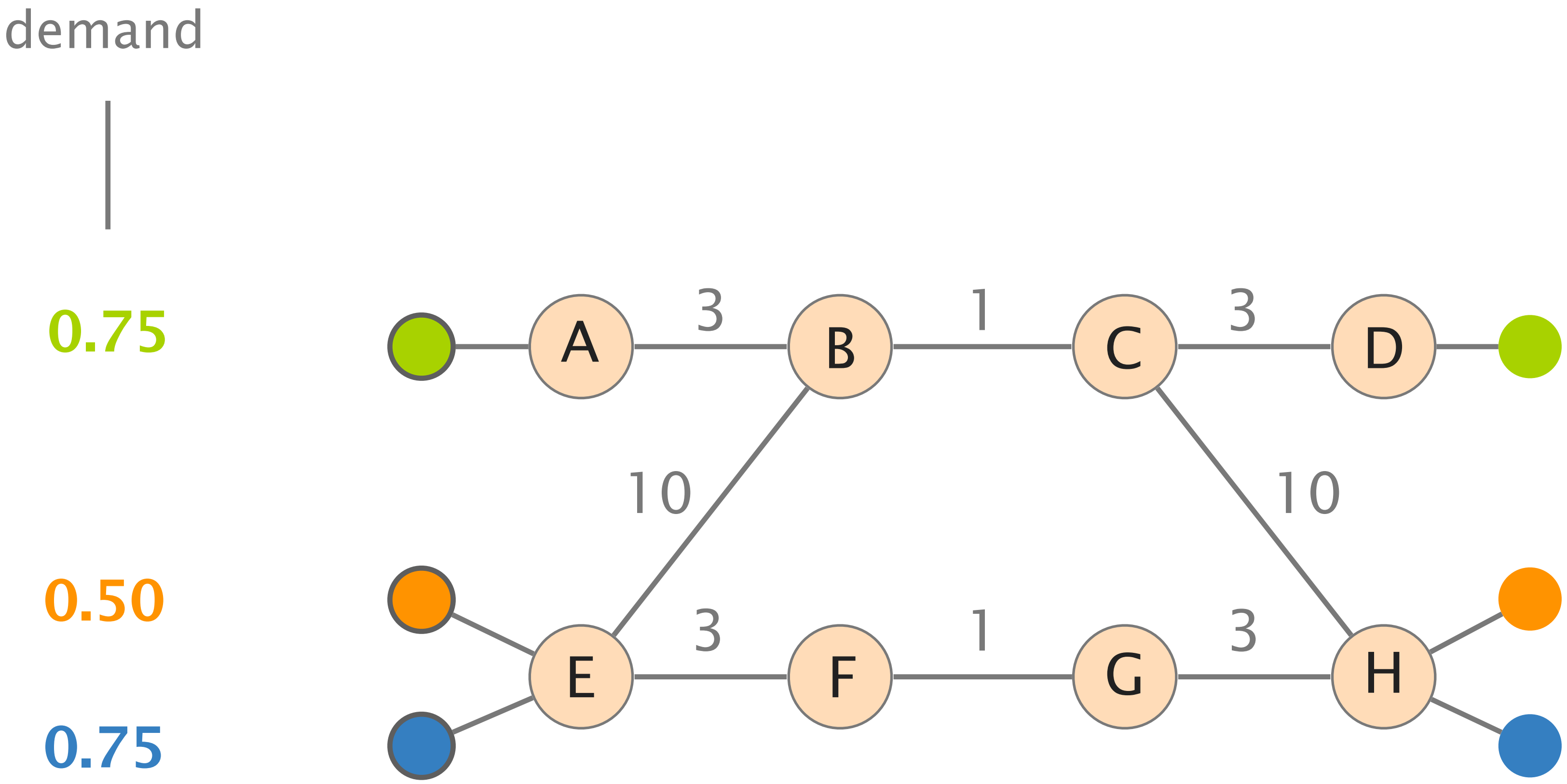
- paths are consistent

  (*e.g.* [s1, a, b, d] and
  [s2, b, a, d] are inconsistent)

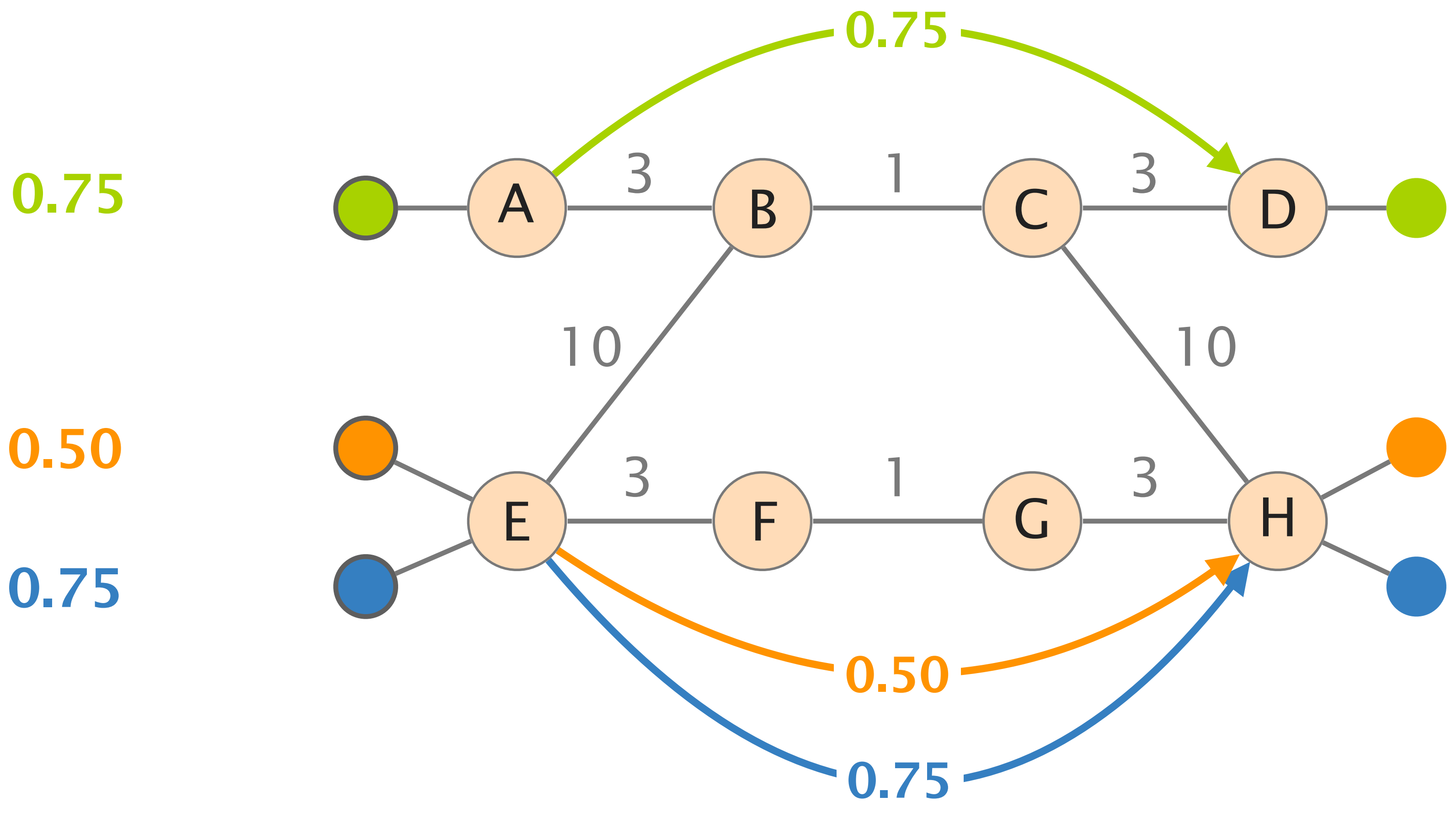
Fibbing can load-balance traffic on multiple paths




source
destination
A
B
C
D
E
F
G
H
3
1
3
10
10
3
1
3

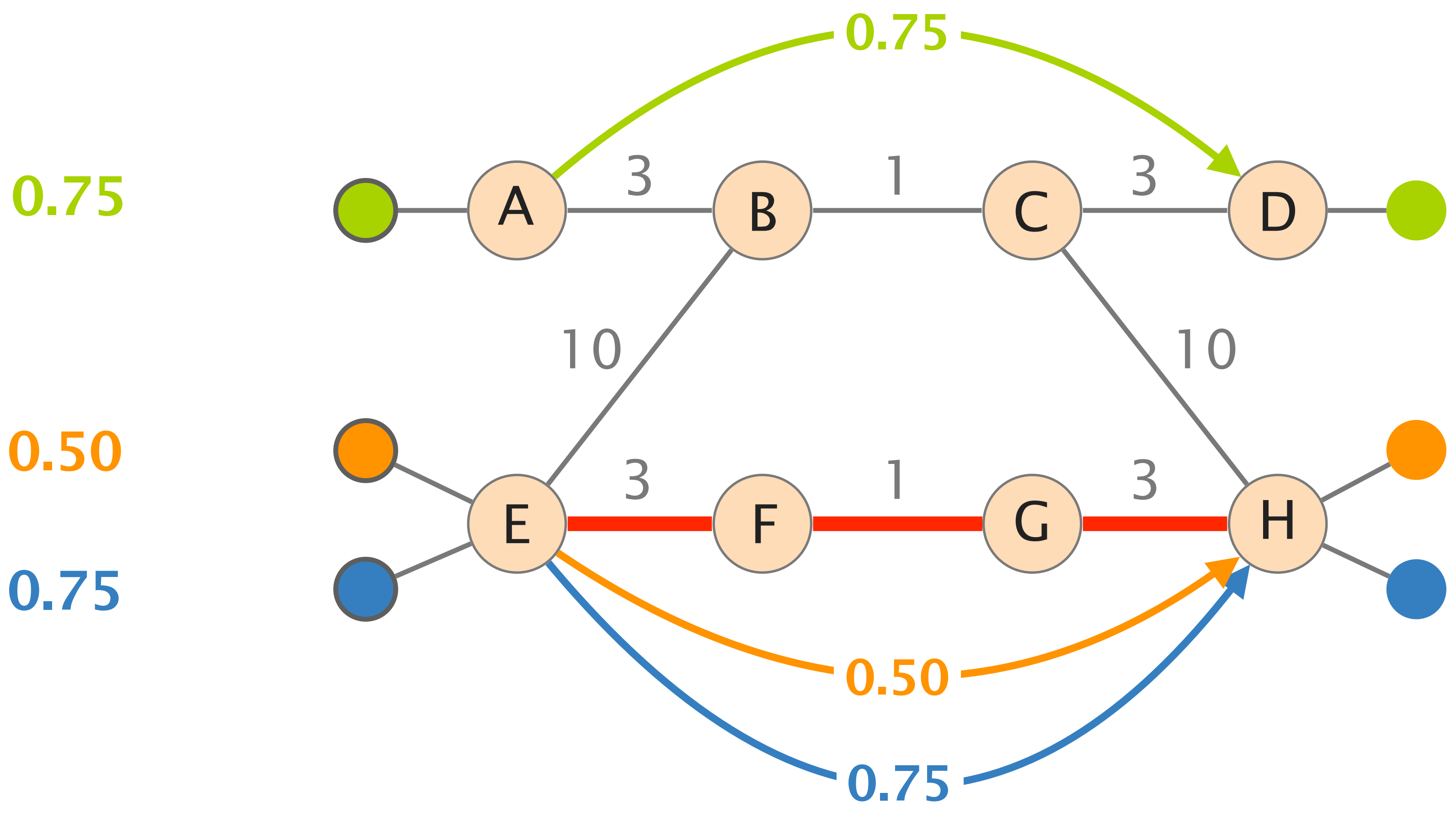


Links have a capacity of 1

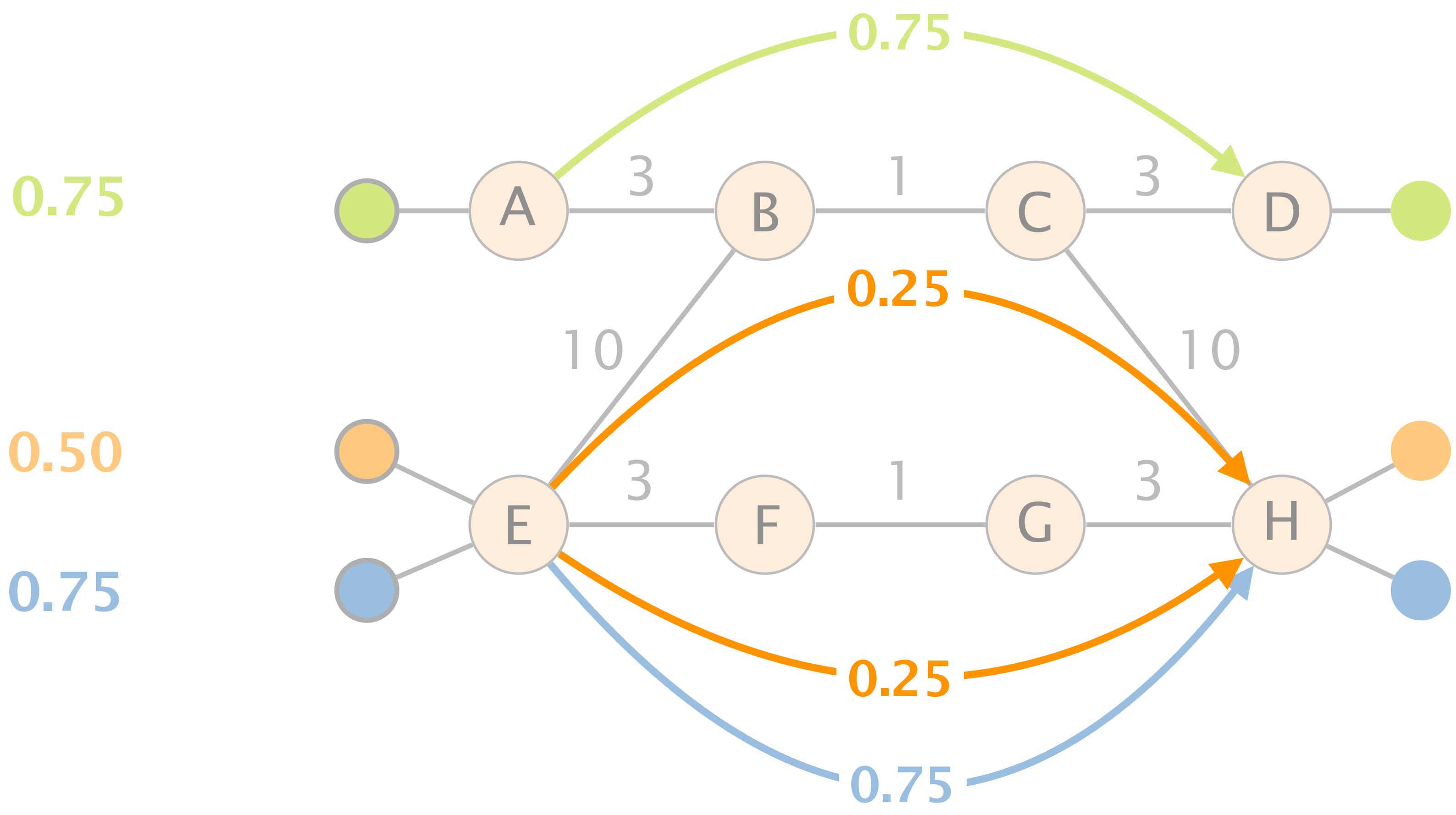


Links have a capacity of 1

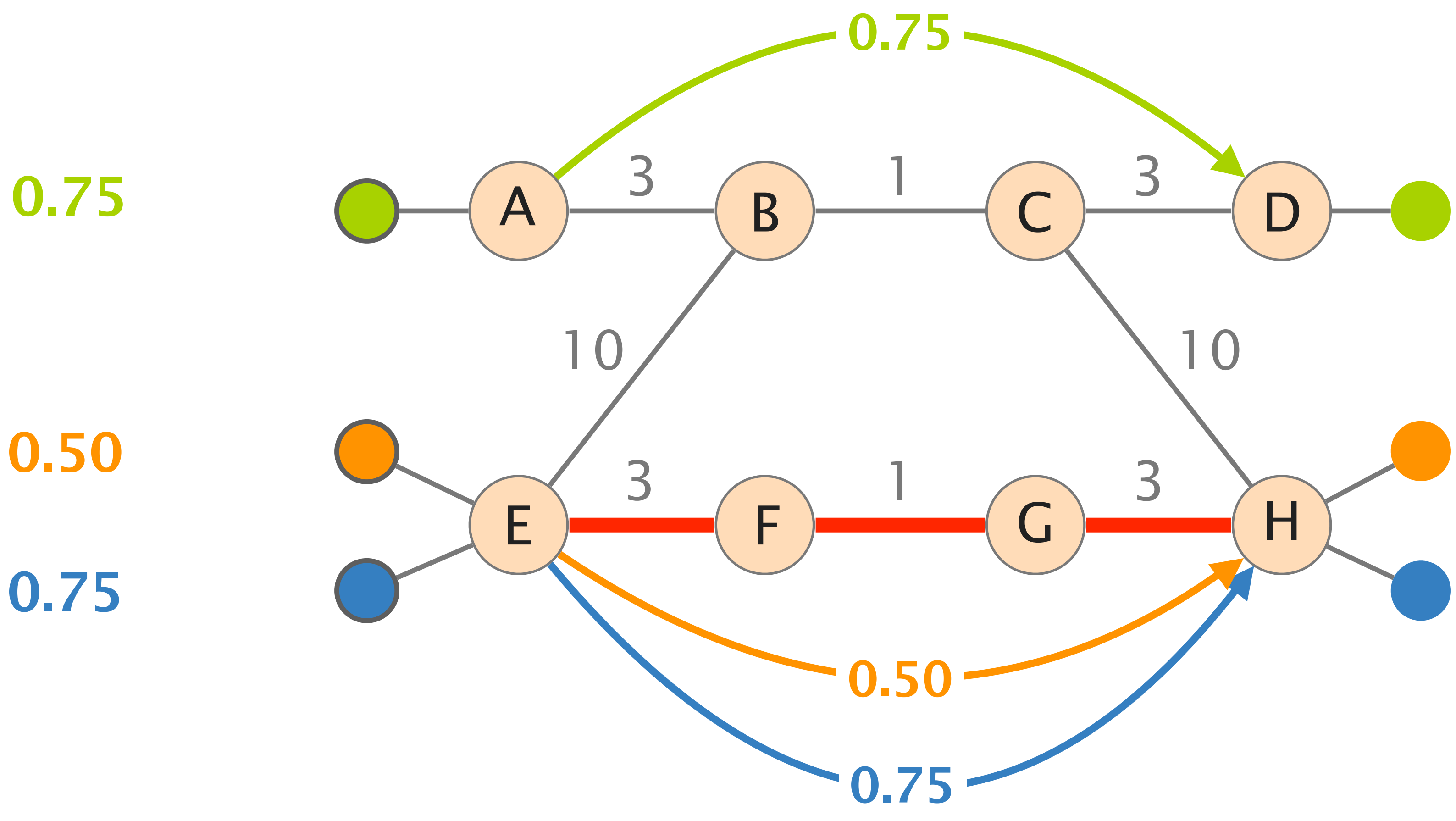
With such demands and forwarding,
the lower path is congested (1.25)



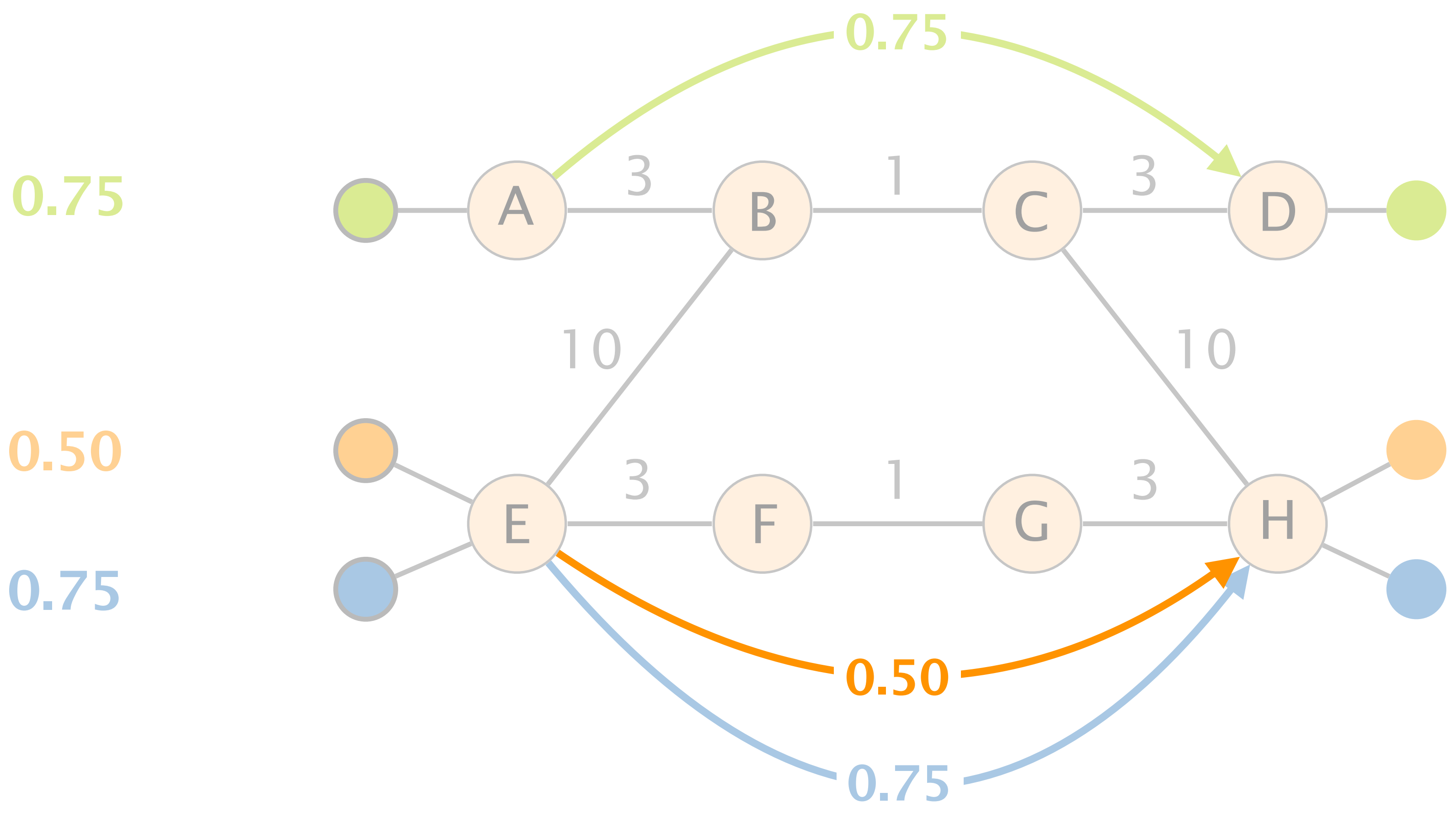
Congestion can be alleviated by splitting the orange flow into two equal parts (.25)



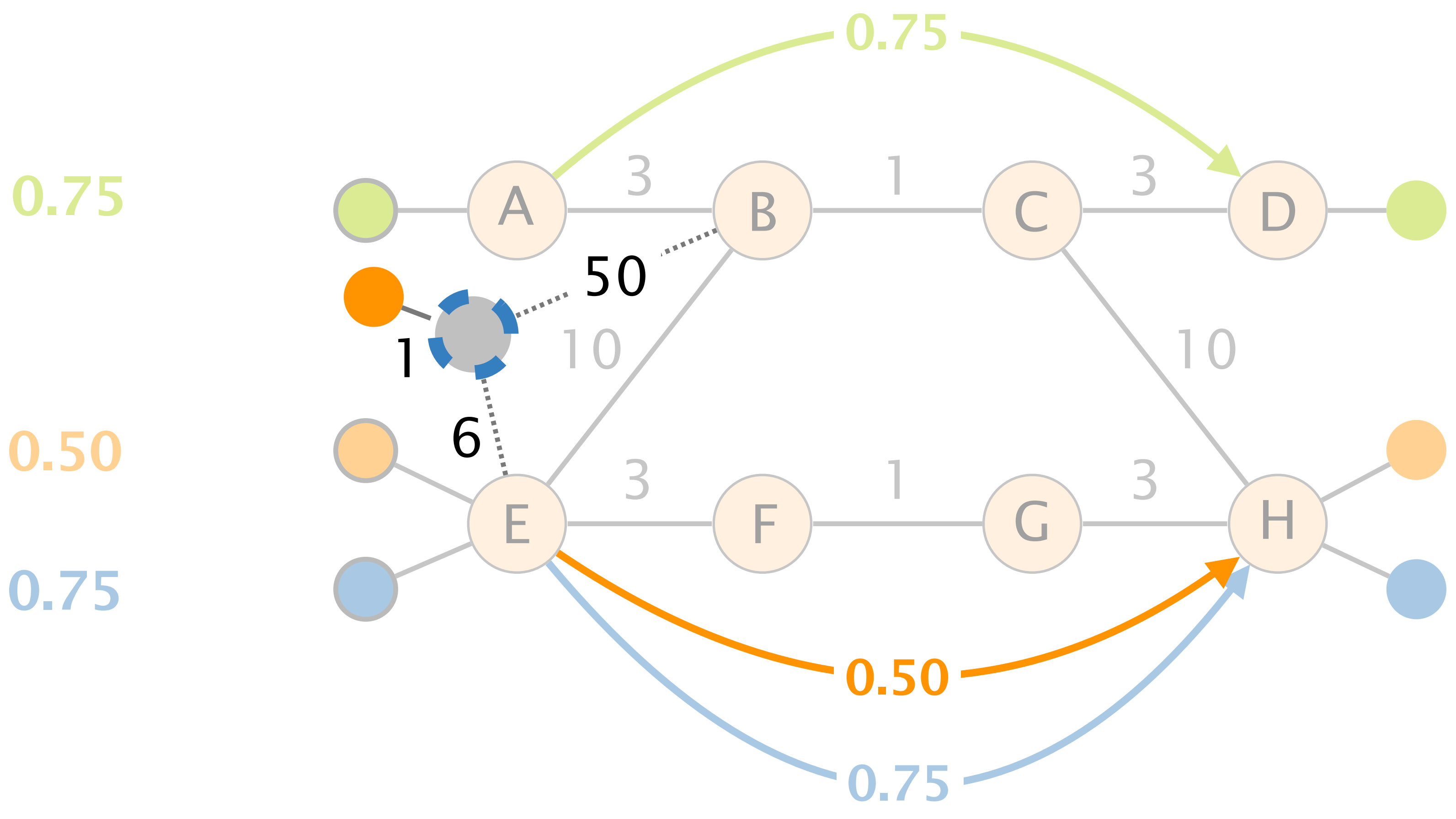
This is impossible to achieve using a link-state protocol



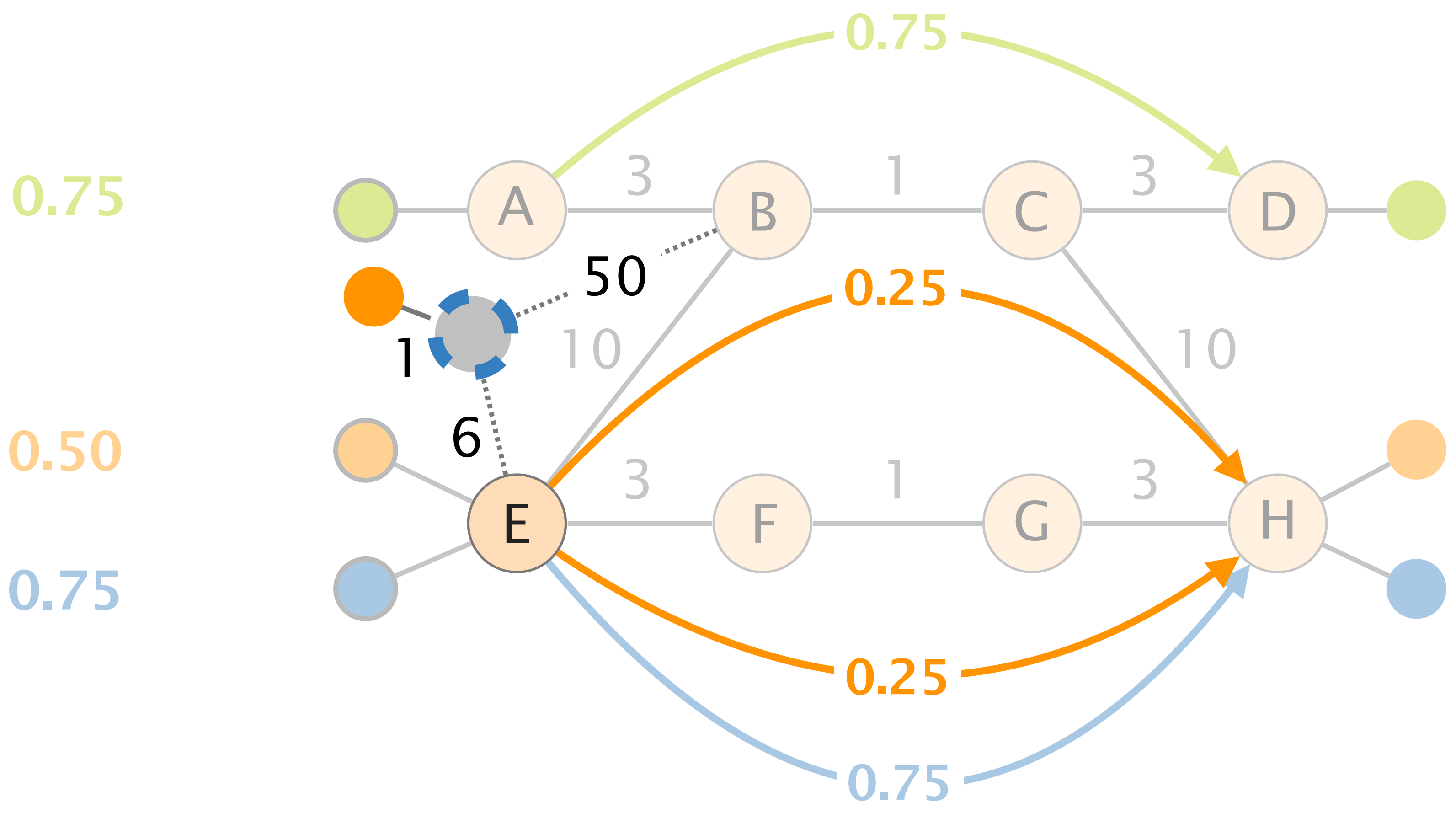
This is easily achievable with Fibbing



One lie is introduced, announcing the orange destination



Now E has two equal cost paths (7) to reach only the orange destination and use them both



Central Control Over Distributed Routing

Fibbing

lying made useful

Expressivity

any path, anywhere

3 **Scalability**

1 lie is better than 2

Scalability

Scalability

time

to compute lies

space

# of lies

Scalability

time

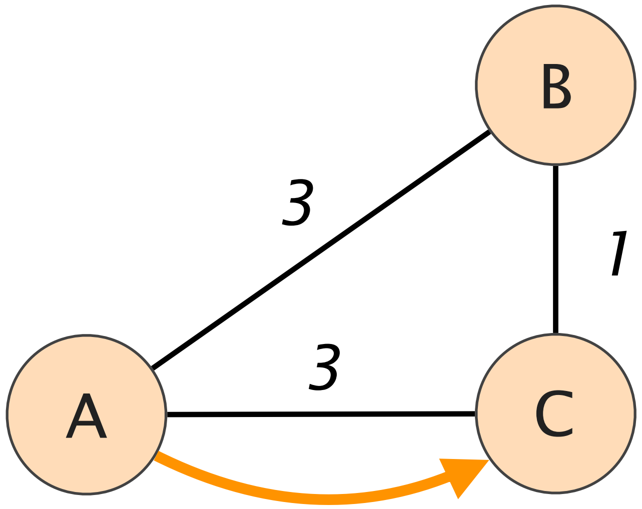
to compute lies

space

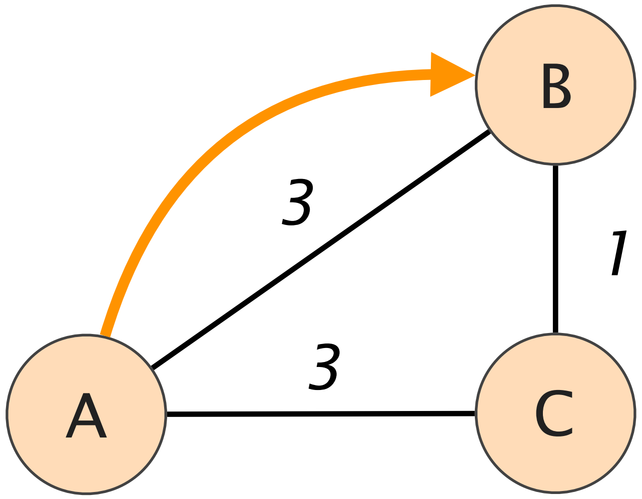
# of lies

Computing virtual topologies is easy:
polynomial in the number of requirements

Computing virtual topologies is easy: polynomial in the number of requirements

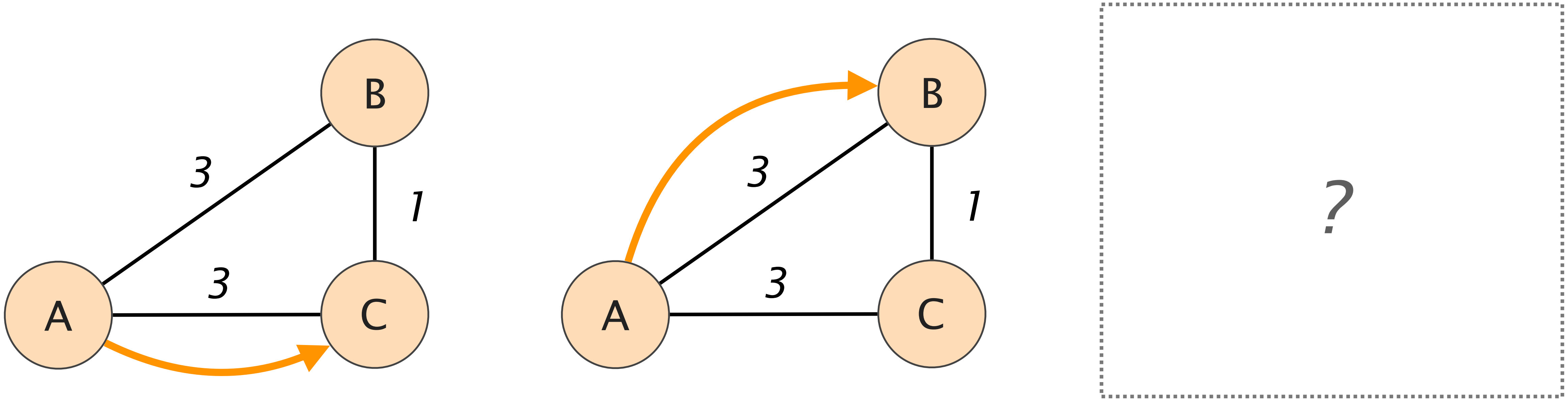


initial



desired

Computing virtual topologies is easy: polynomial in the number of requirements



initial desired virtual

For each router *r* whose next-hop

for a destination *d* changes to *j:*

For each router *r* whose next-hop
for a destination *d* changes to *j:*

- Let *w* be the current path weight between *r* and *d*
- Create one virtual node *v* advertising *d*
  with a weight $x < w$
- Connects it to *r* and *j*

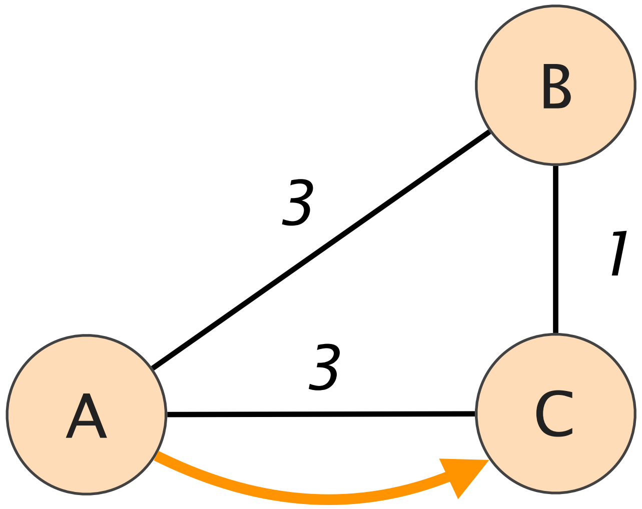
- Create one virtual node v advertising d with a weight $x < w$

always possible

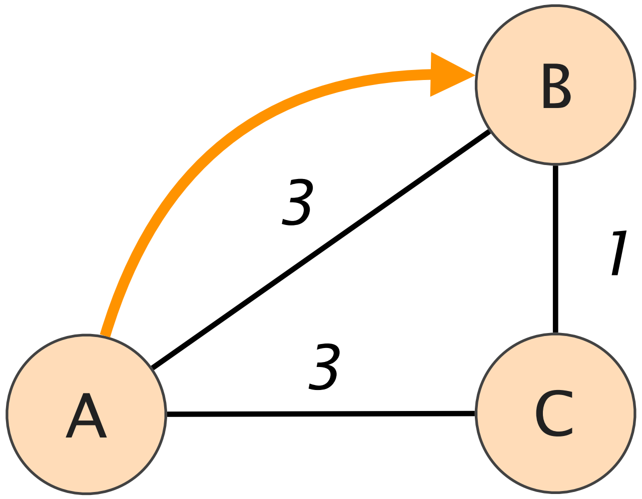
by reweighting the initial graph

- Create one virtual node *v* advertising *d* with a weight $x < w$

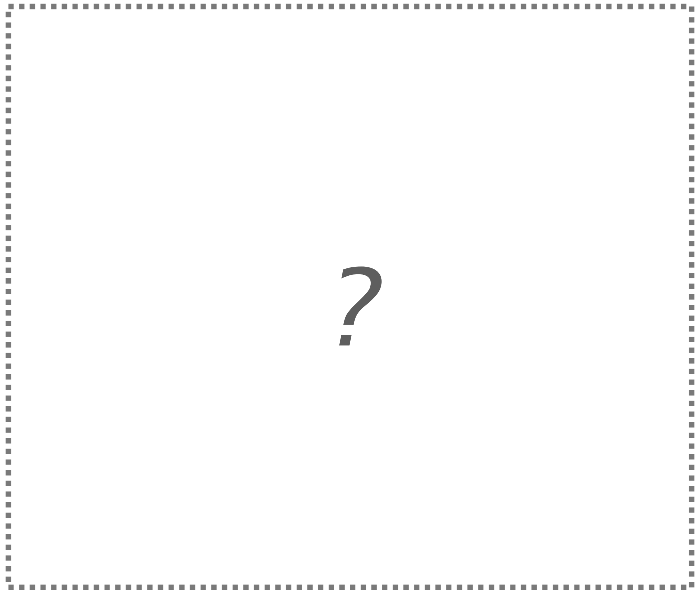
Computing virtual topologies is easy: polynomial in the number of requirements



initial

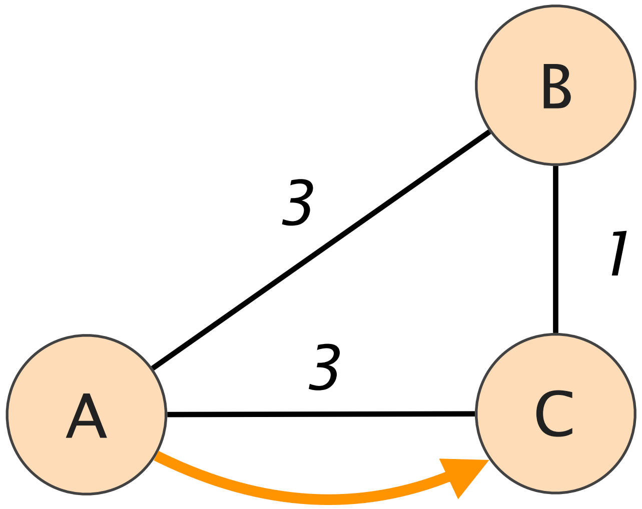


desired

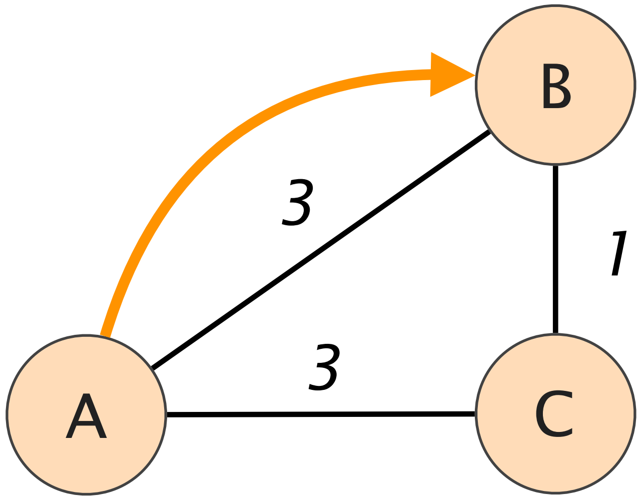


virtual

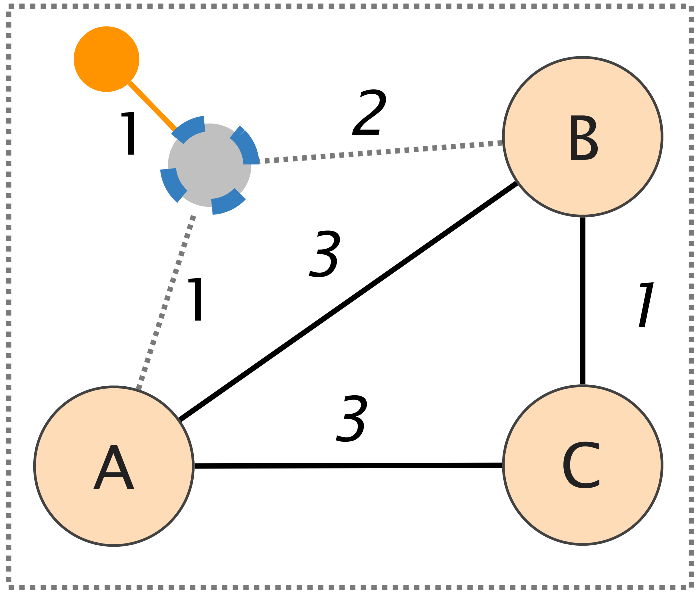
Computing virtual topologies is easy: polynomial in the number of requirements



initial



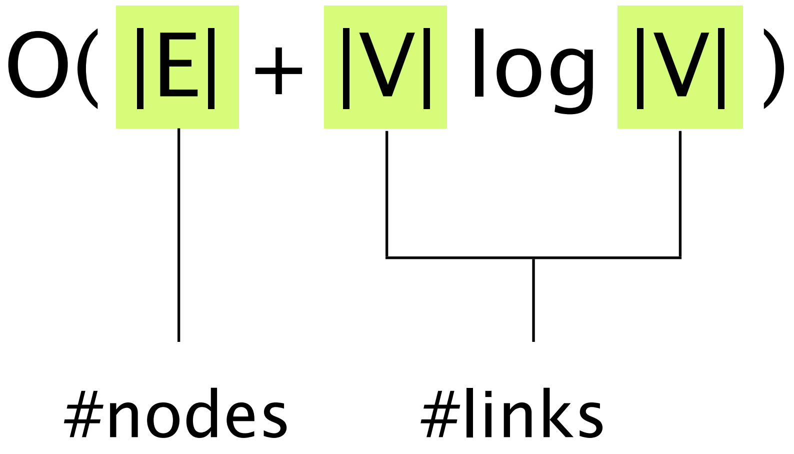
desired



virtual

The resulting topology can be huge and each router needs to run Dijkstra on it

Dijkstra's algorithm complexity



$$O(\,|E| + |V| \log |V|\,)$$

#nodes

#links

Scalability

time

to compute lies

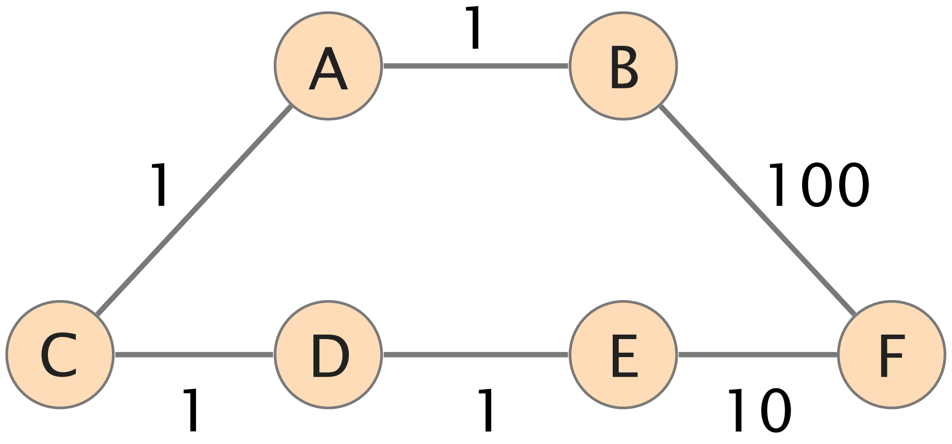
space

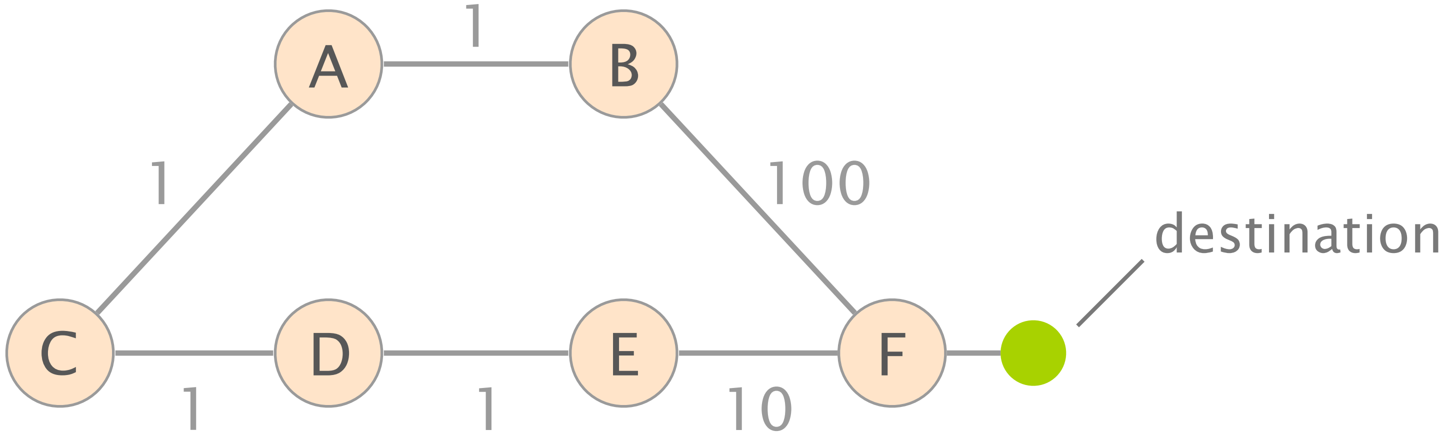
# of lies

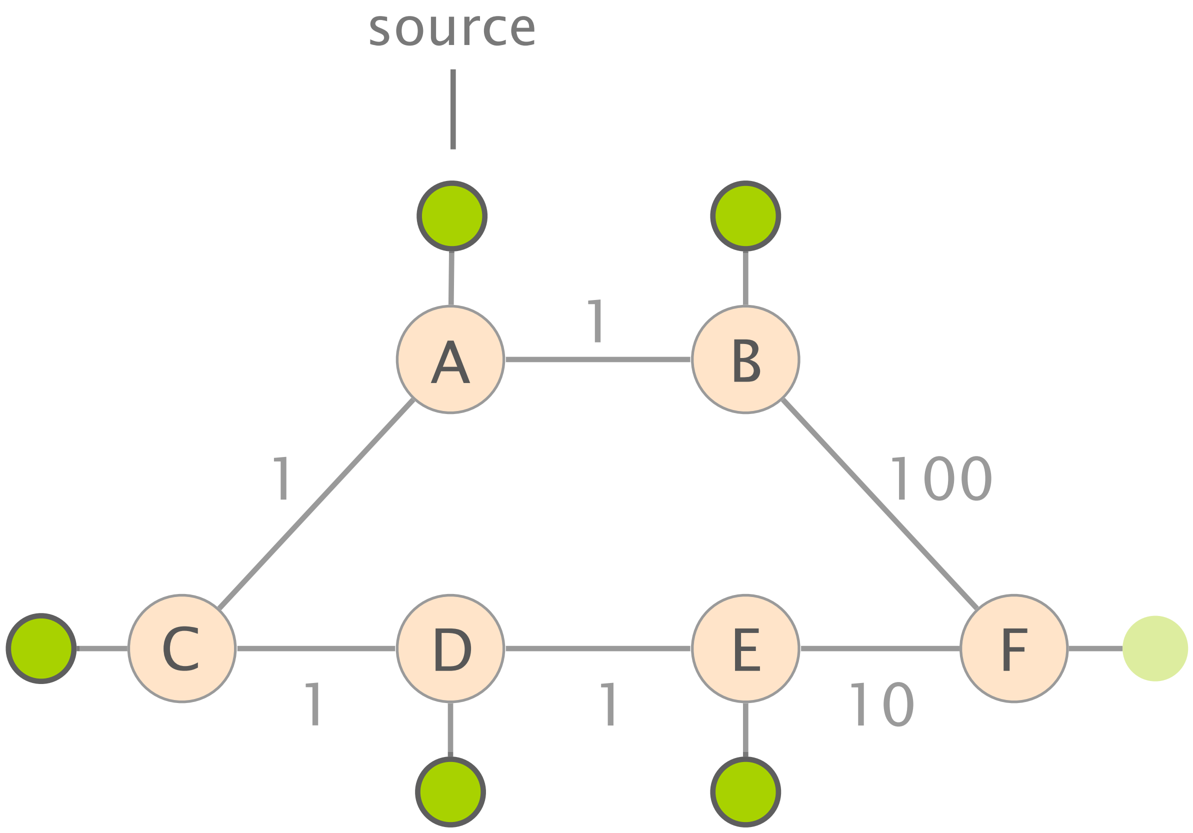
Good news

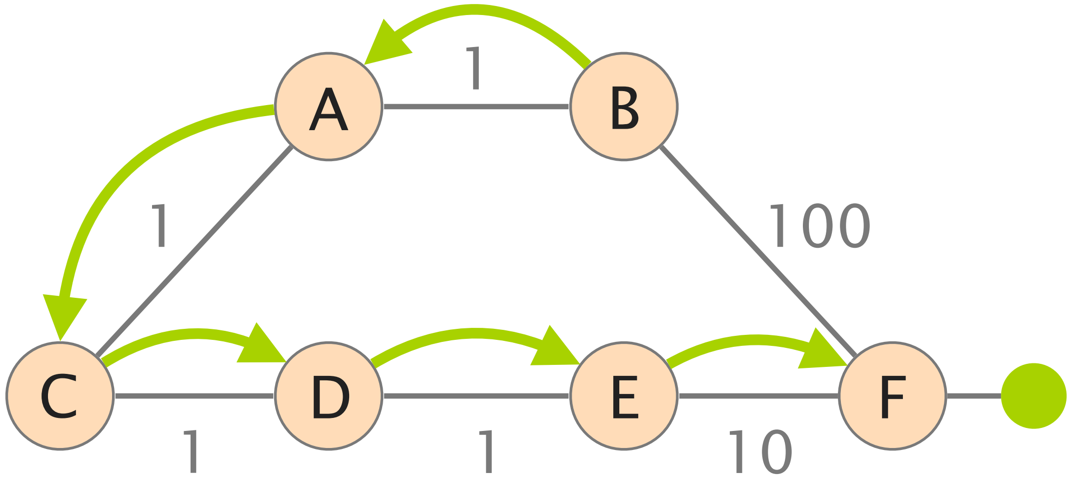
Lots of lies are not required,
some of them are redundant

Let’s us consider
a simple example


A
B
C
D
E
F
1
1
100
1
1
10

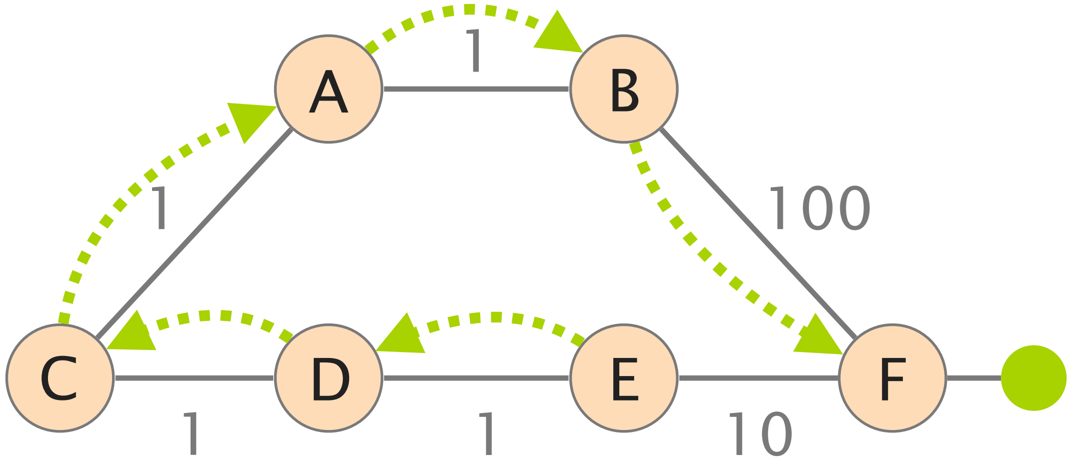

A
B
C
D
E
F
1
1
100
1
1
10
destination


source
A
B
C
D
E
F
1
1
100
1
1
10



original shortest-path

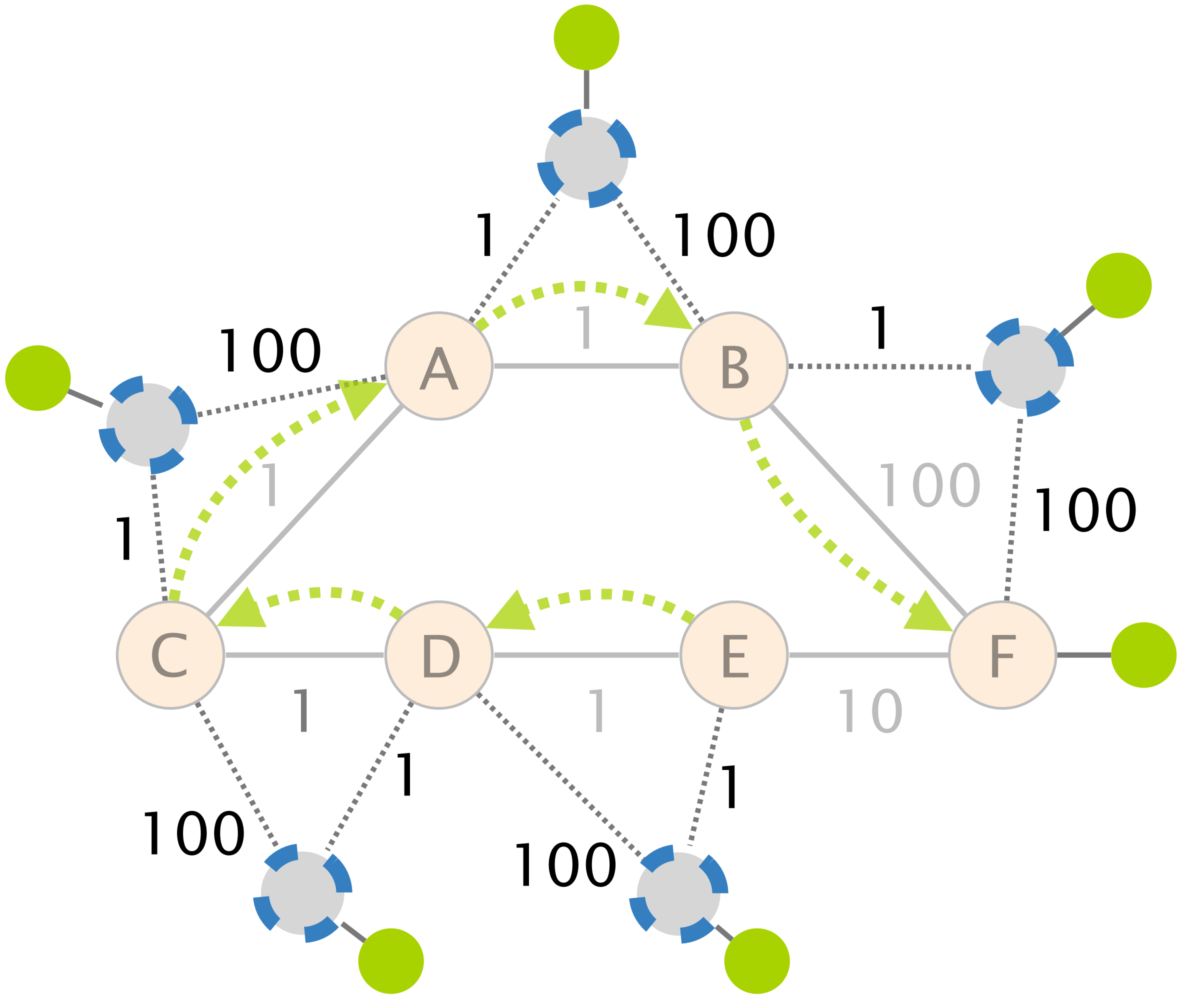
“down and to the right”



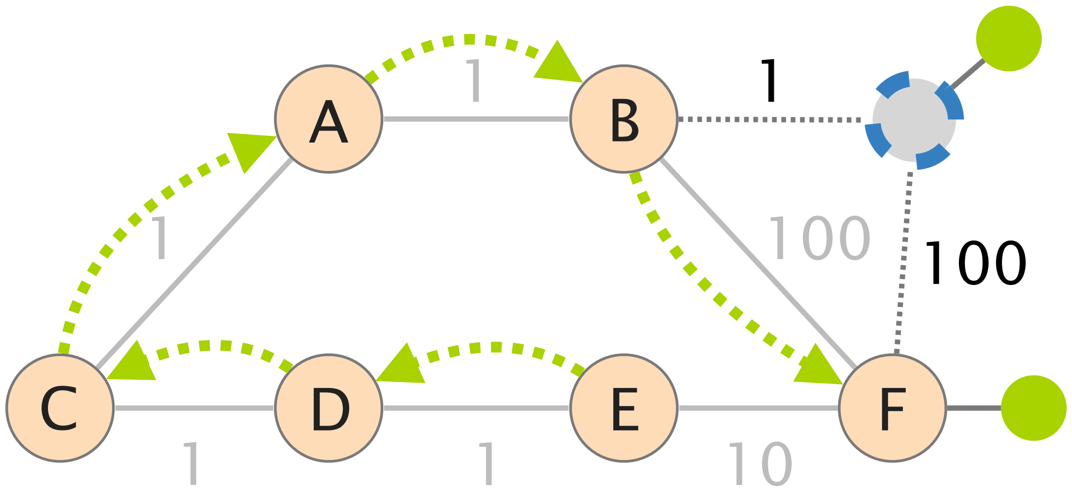
desired shortest-path

“up and to the right”

Our naive algorithm would create 5 lies—one per router



A single lie is sufficient (and necessary)

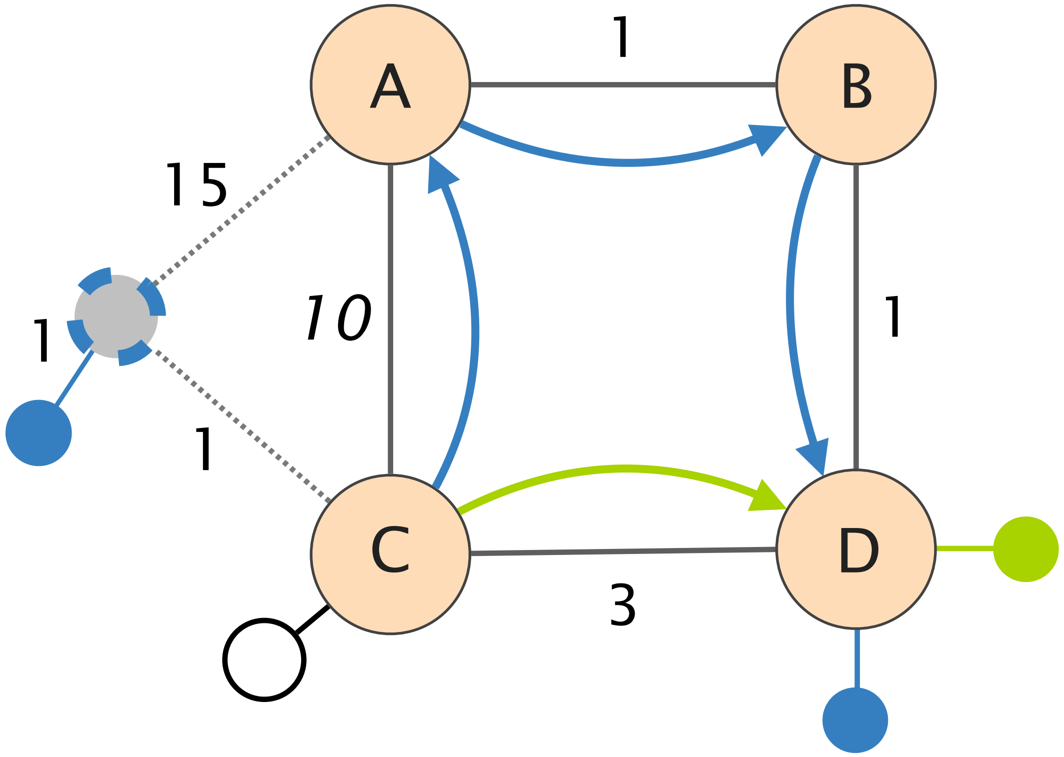


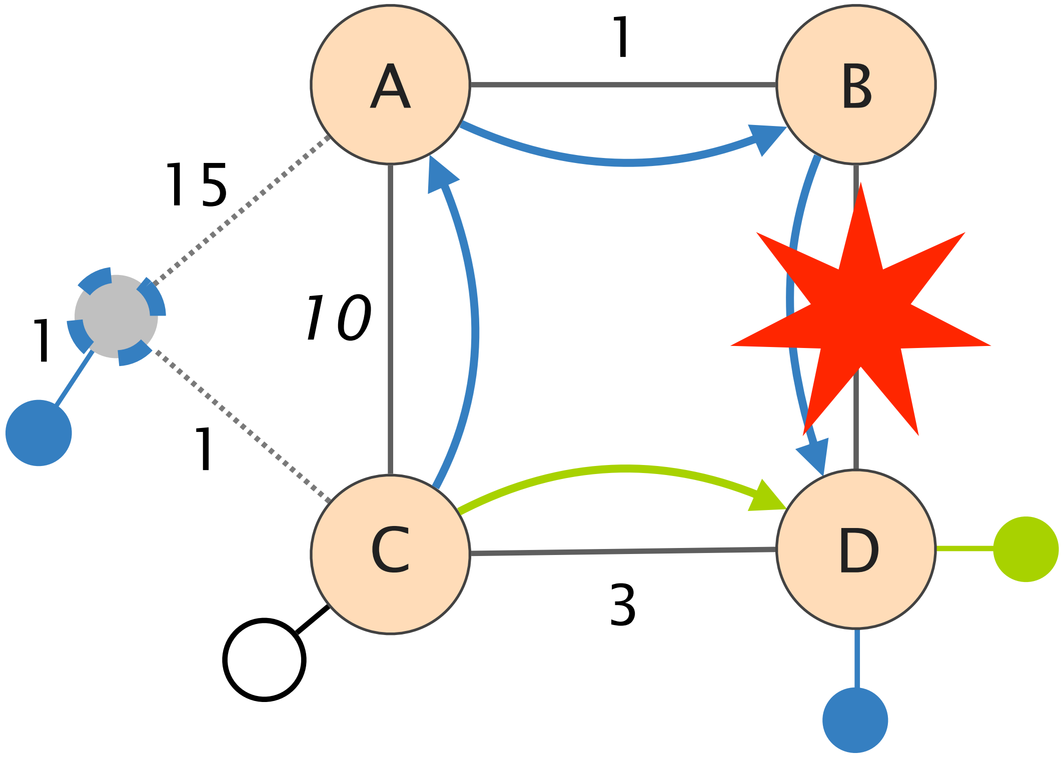
We can minimize the topology size
using an Integer Linear Program

While efficient,
an ILP is inherently slow

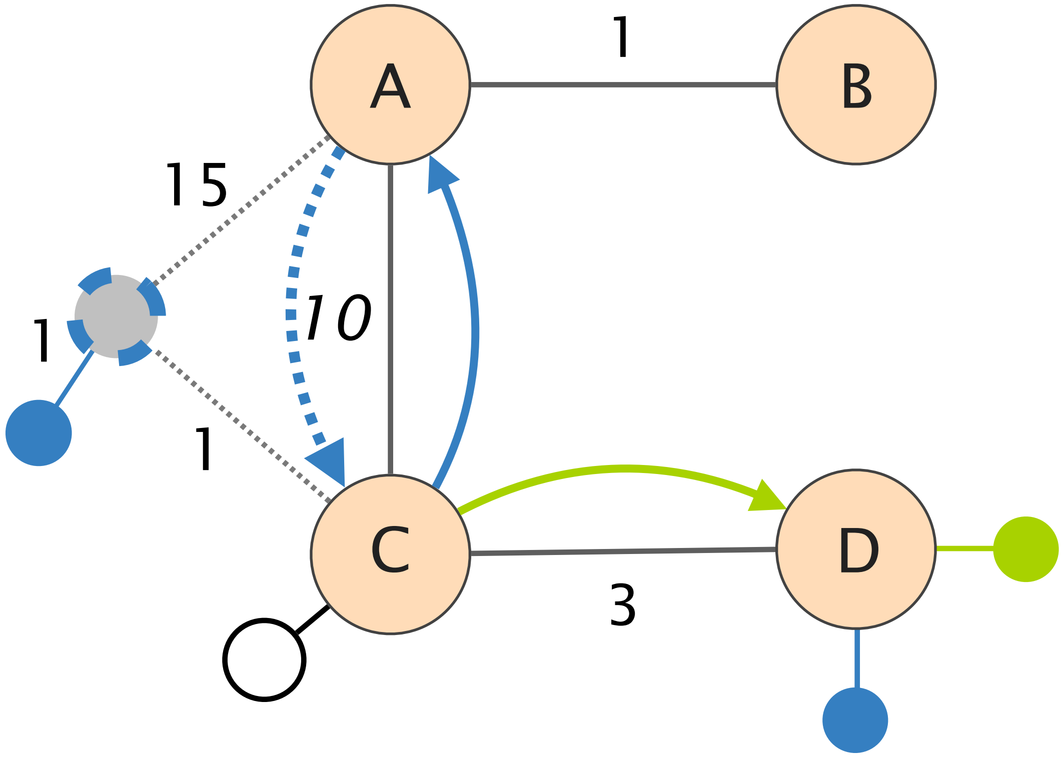
	Naive	Integer Linear Program
time	optimal	slow
space (topology size)	large	optimal

Computation time matters in case of network failures

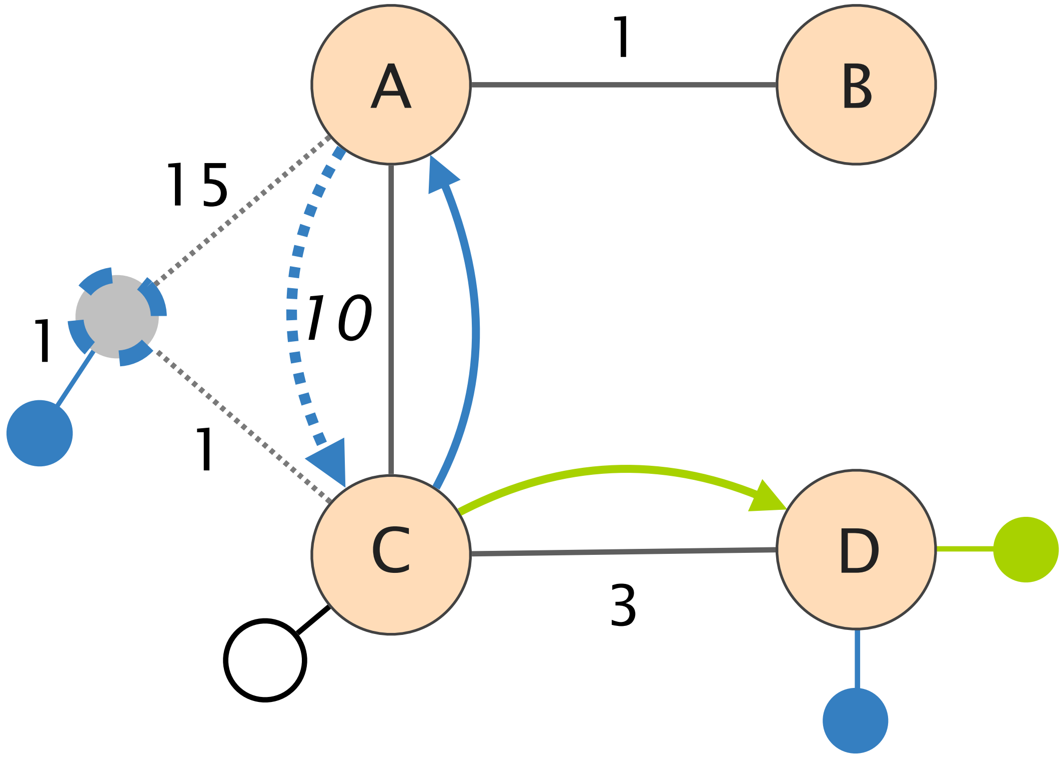

A
B
C
D
1
15
10
1
1
1
3


1
A
B
15
1
10
1
C
D
3

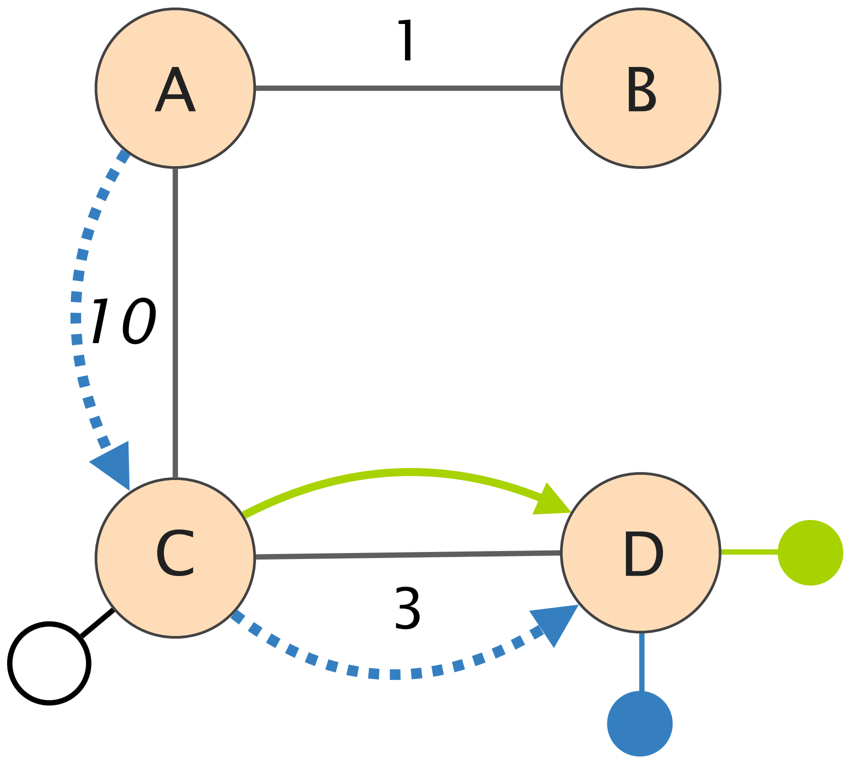
A loop is created as C starts to use A which still forwards according to the lie



The solution is to remove the lie



The solution is to remove the lie

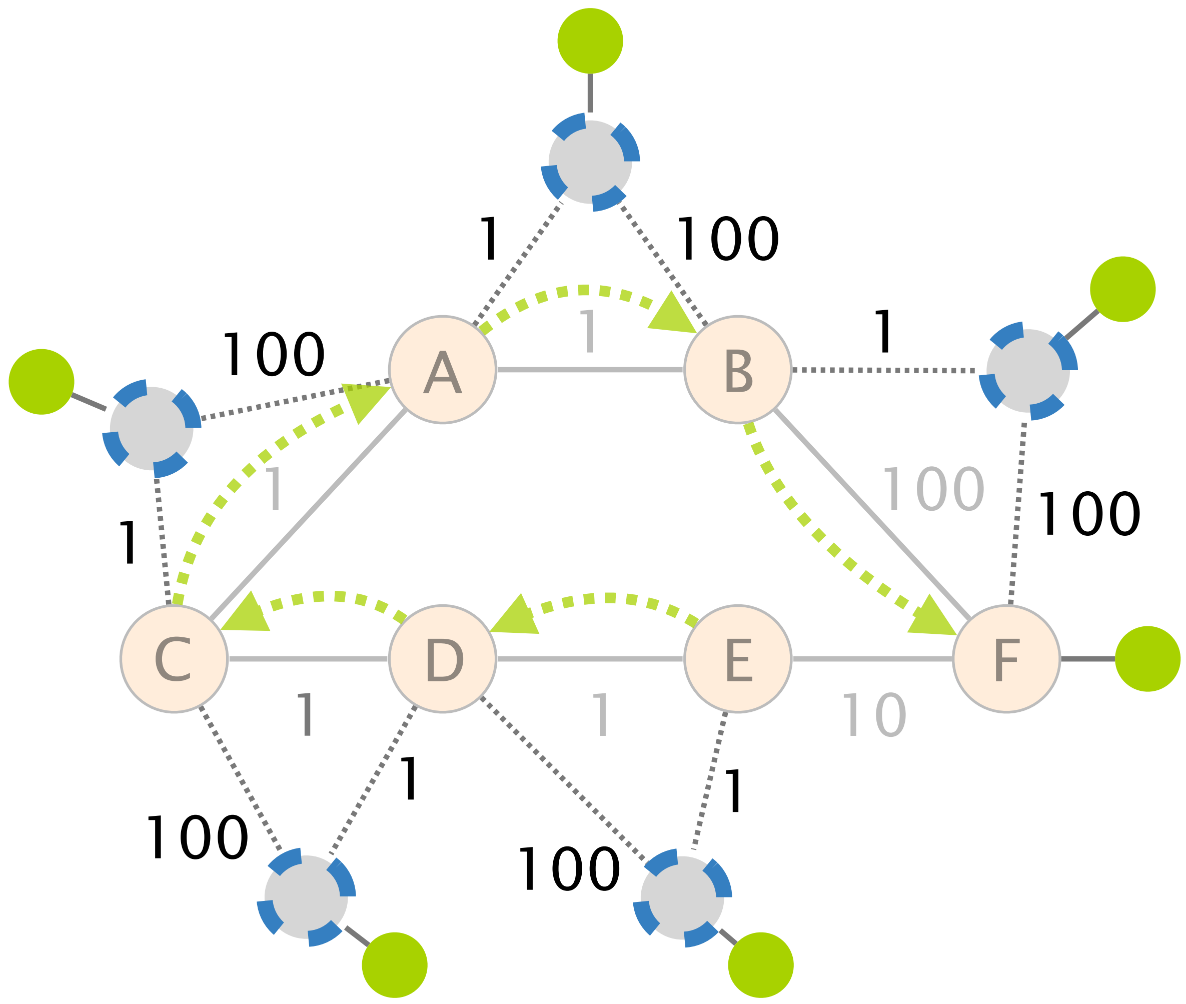


Upon failures, the network topology
has to be recomputed**, fast**

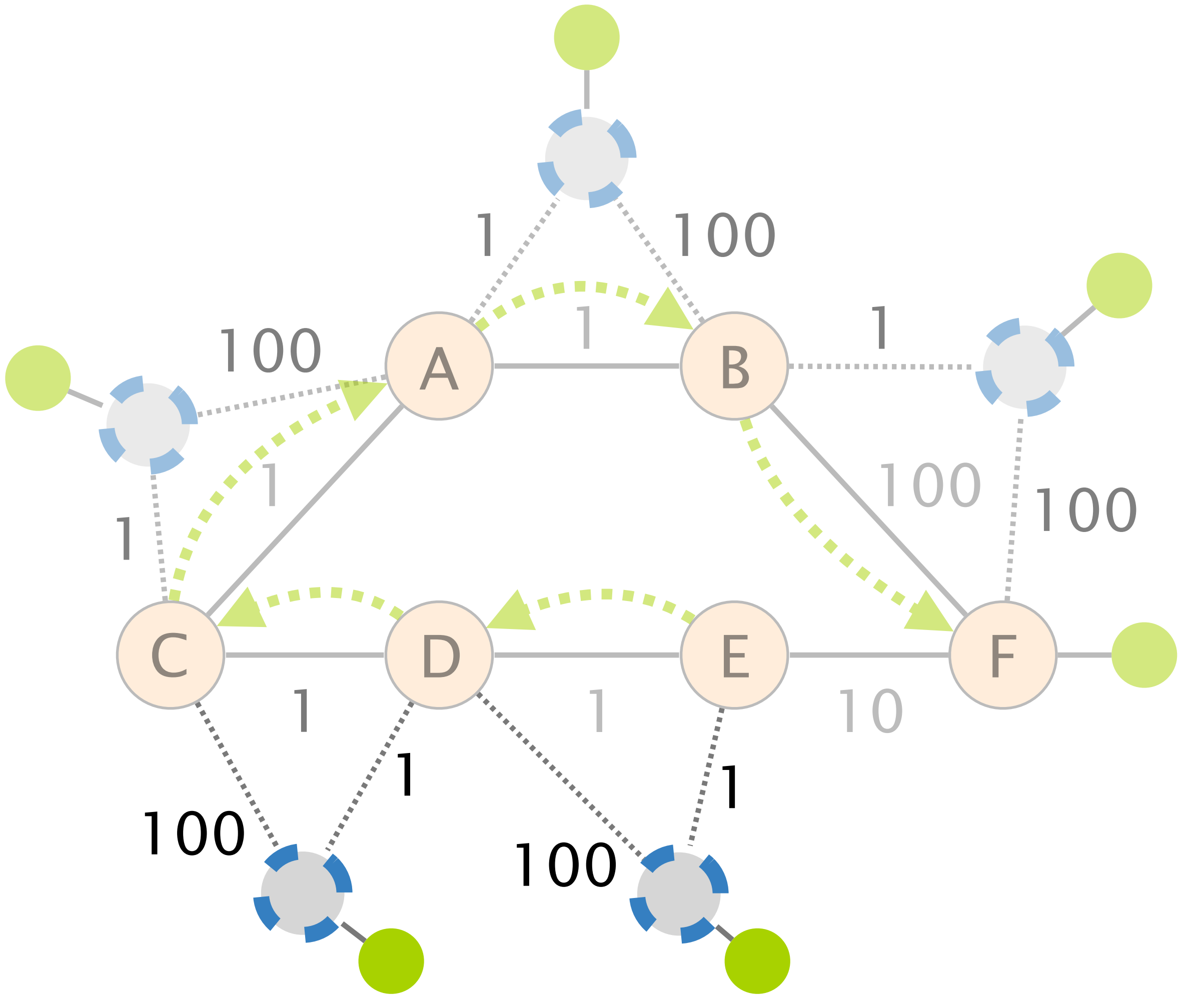
	Naive	Integer Linear Program
time	optimal	slow
space (topology size)	large	optimal

	Naive	Merger	Integer Linear Program
time	optimal	fast	slow
space (topology size)	large	small	optimal

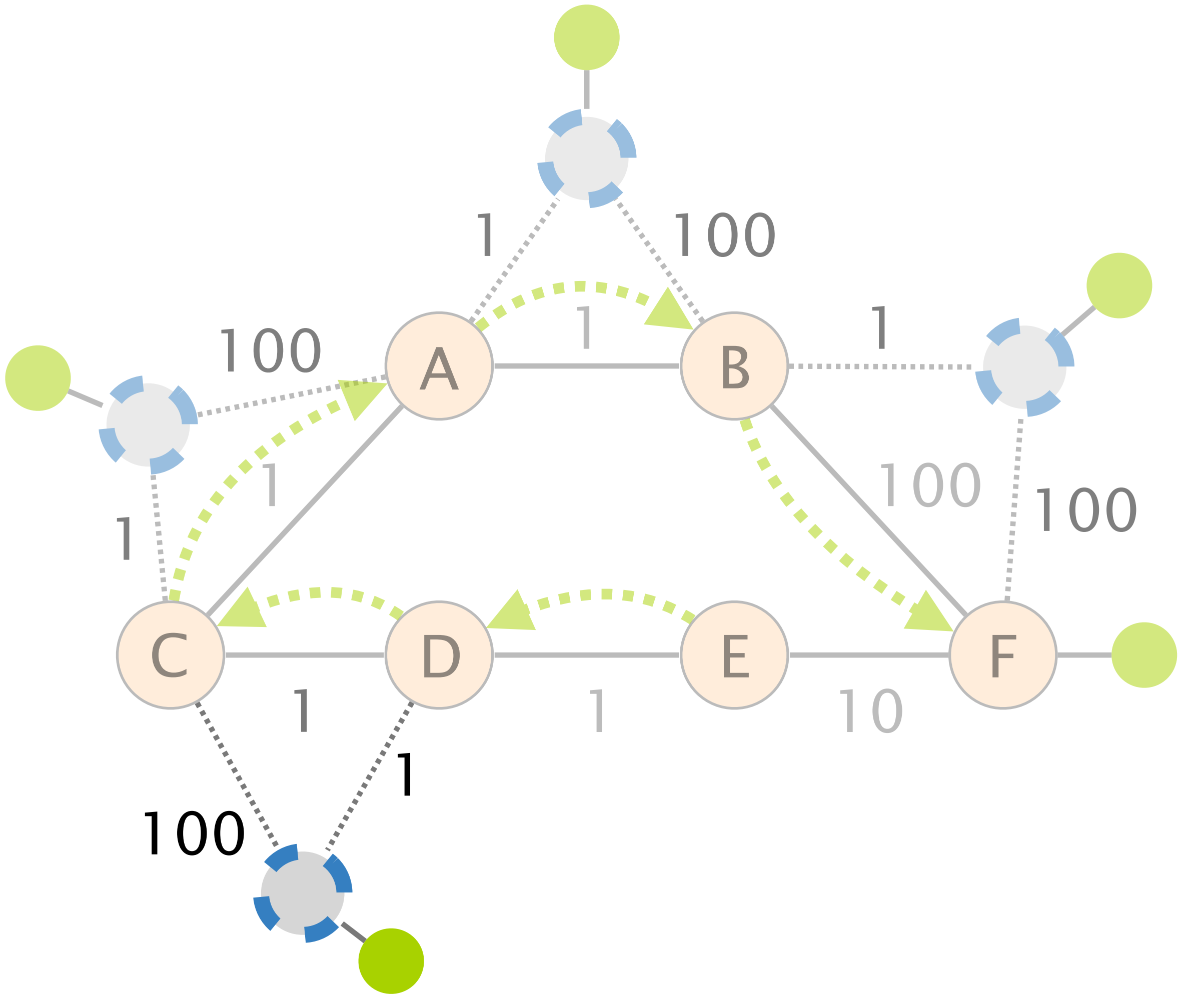
Merger iteratively tries to merge lies produced by the Naive algorithm



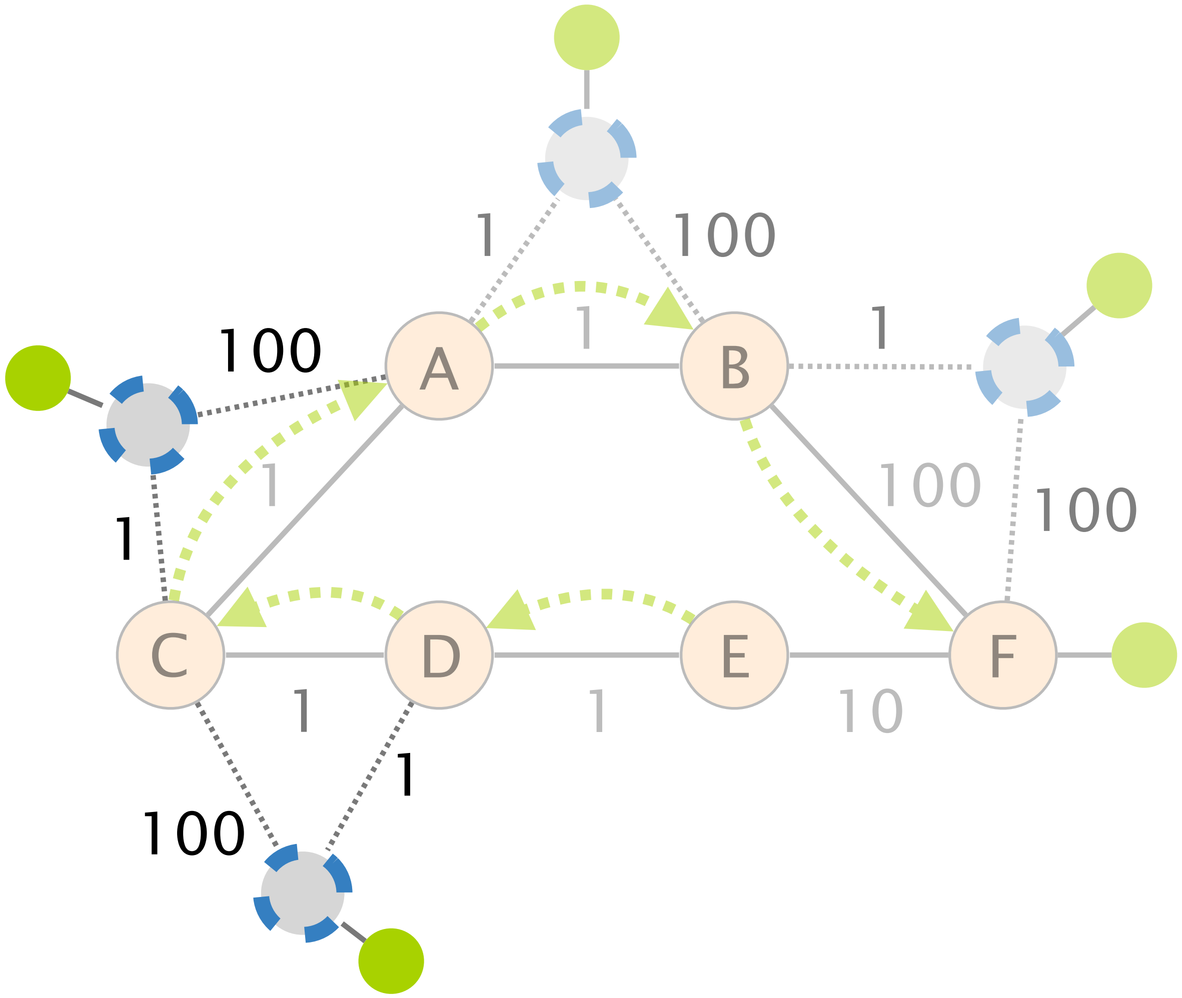
Merger iteratively tries to merge lies produced by the Naive algorithm



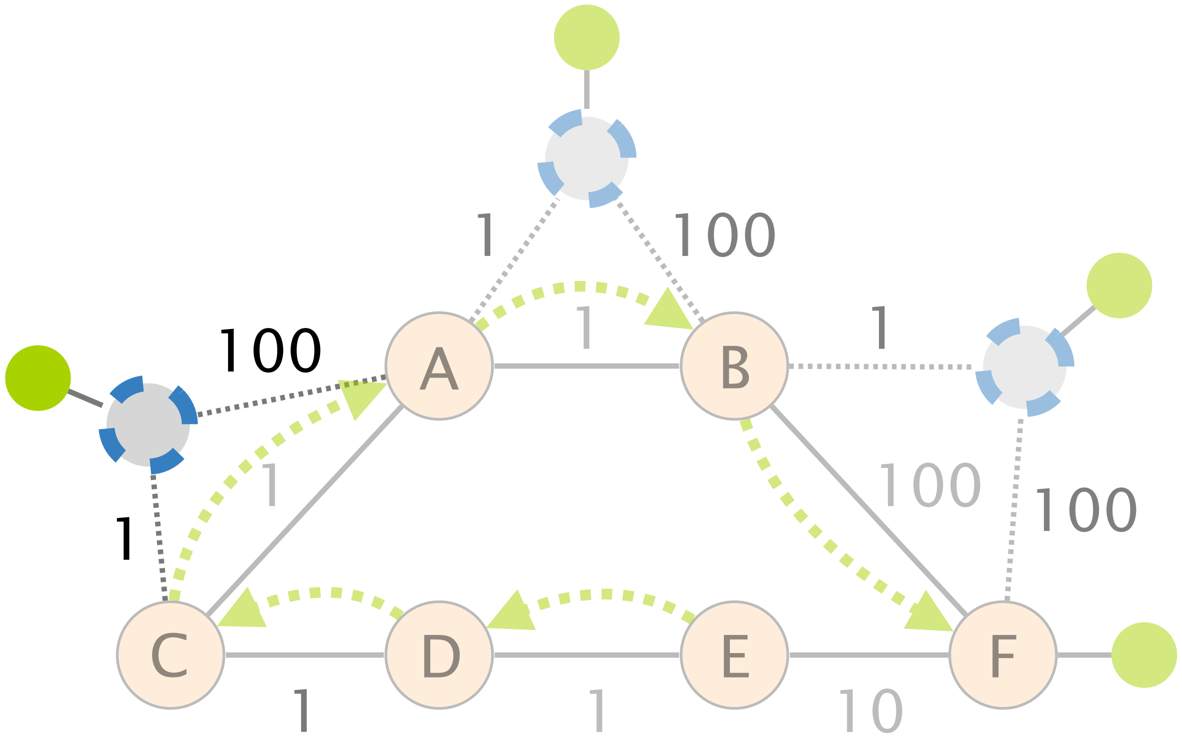
Merger iteratively tries to merge lies produced by the Naive algorithm



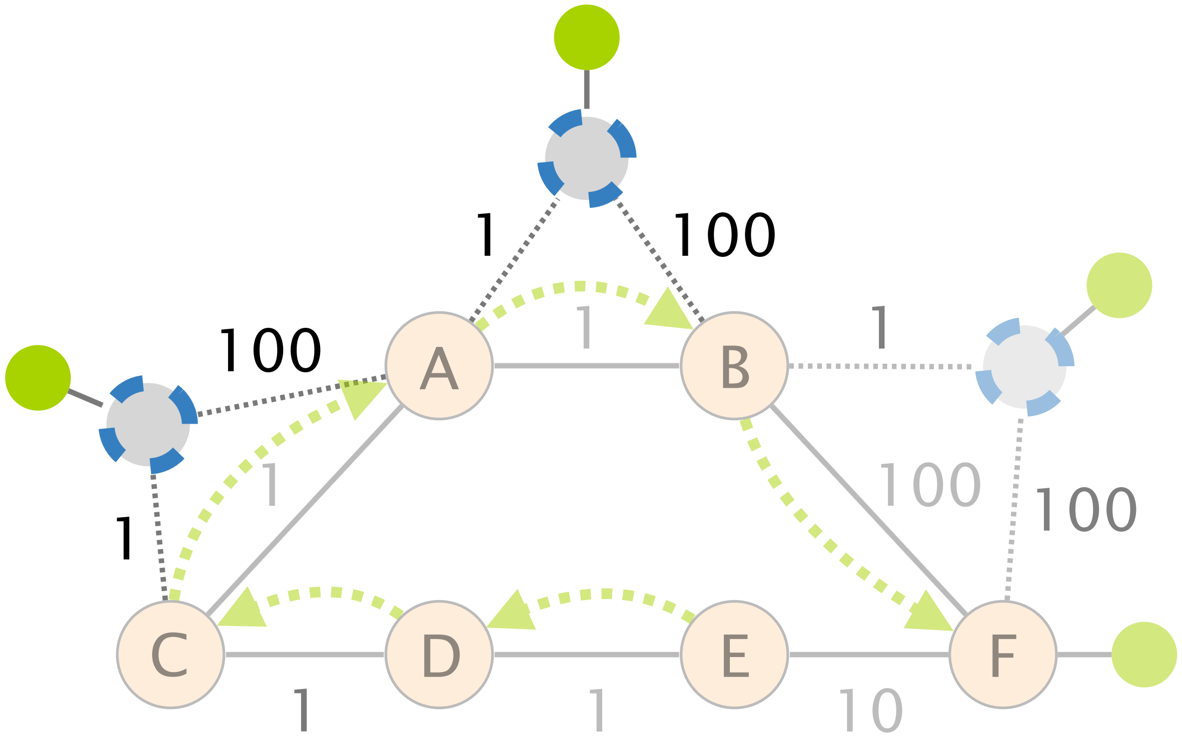
Merger iteratively tries to merge lies produced by the Naive algorithm



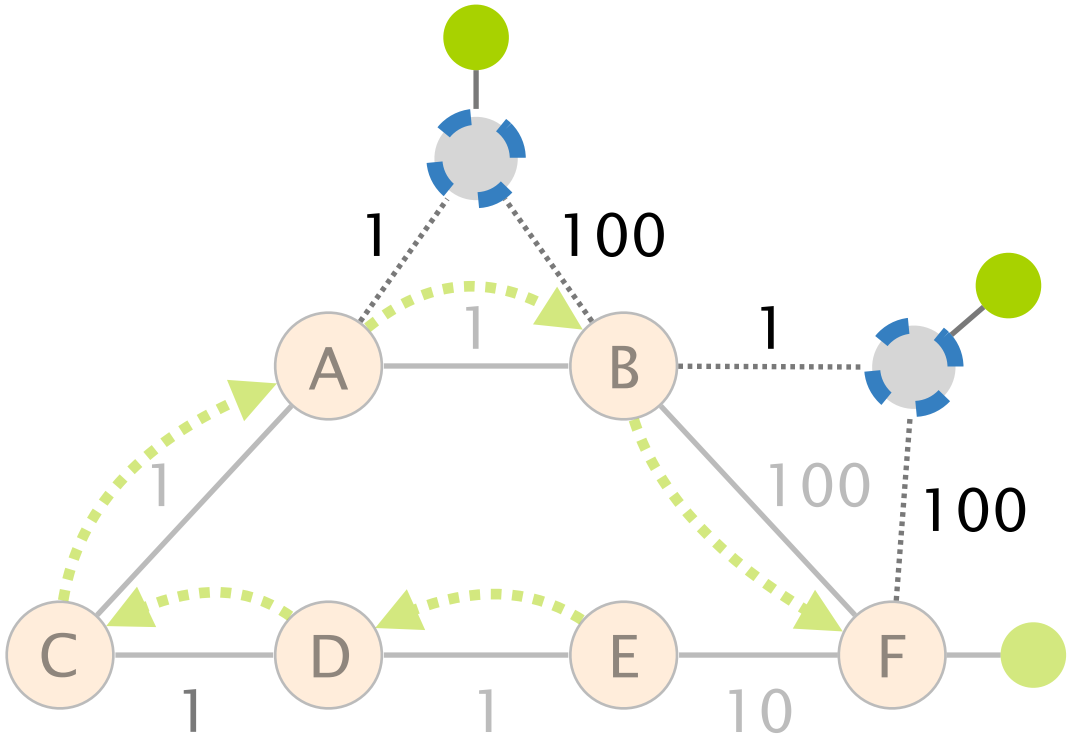
Merger iteratively tries to merge lies produced by the Naive algorithm



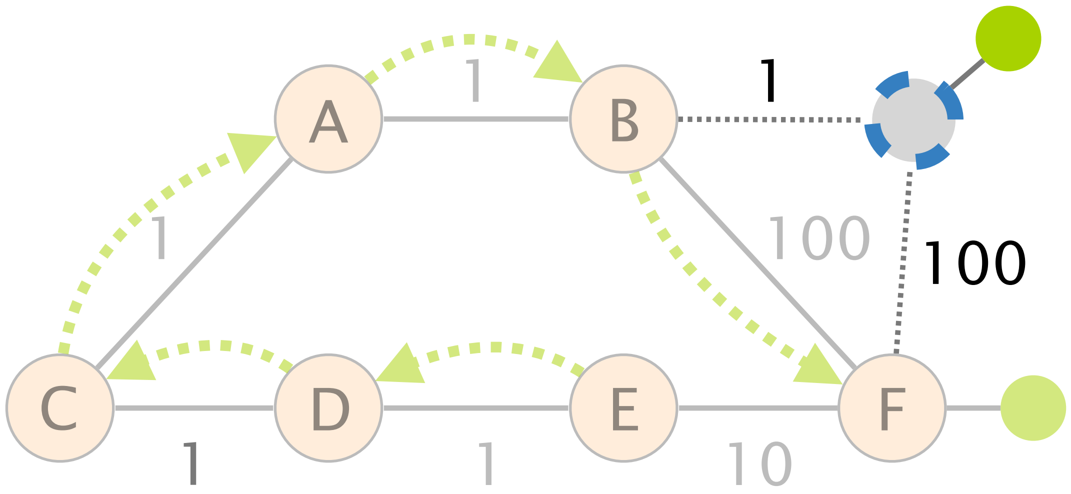
Merger iteratively tries to merge lies produced by the Naive algorithm



Merger iteratively tries to merge lies produced by the Naive algorithm



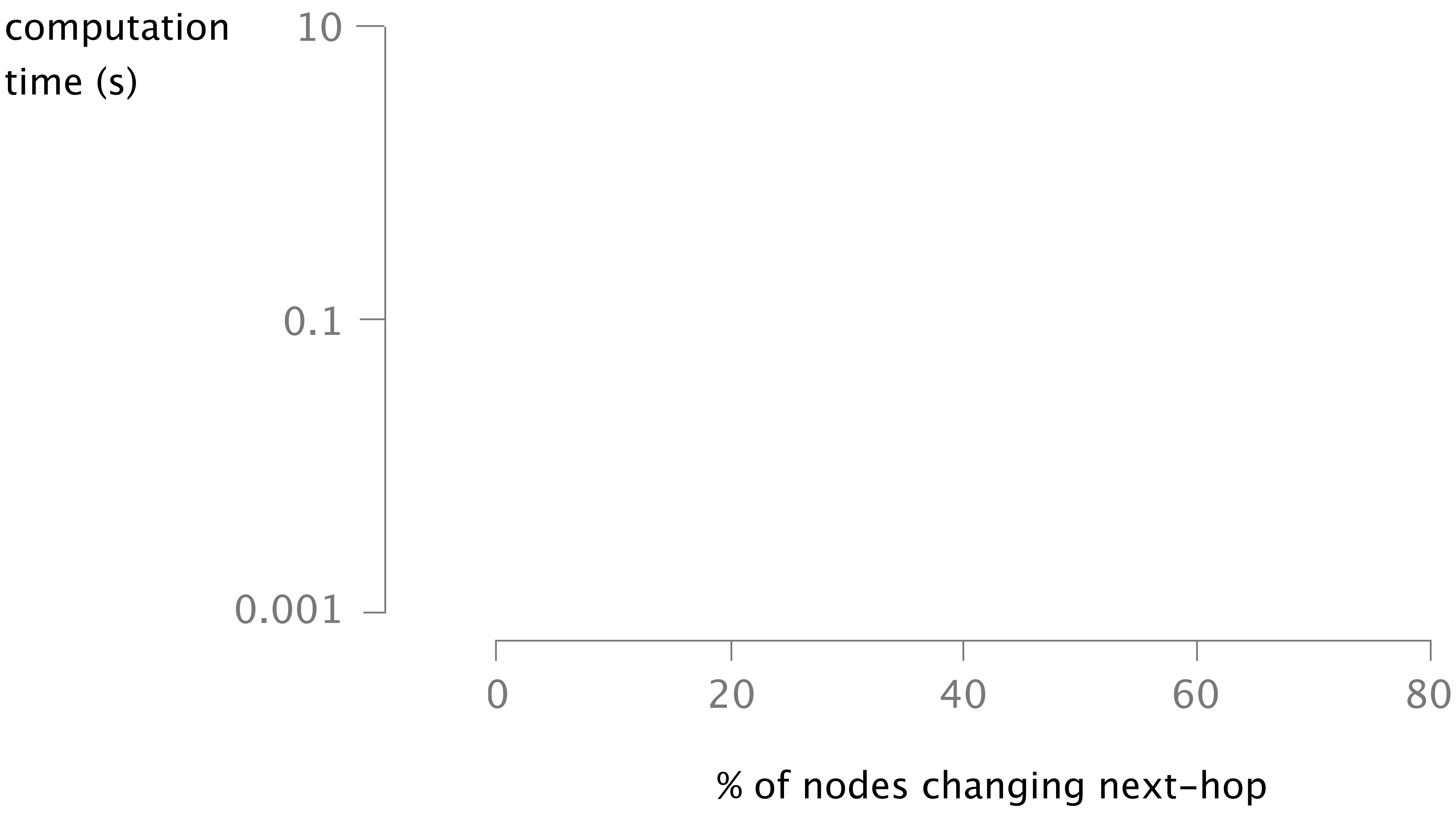
Merger iteratively tries to merge lies produced by the Naive algorithm



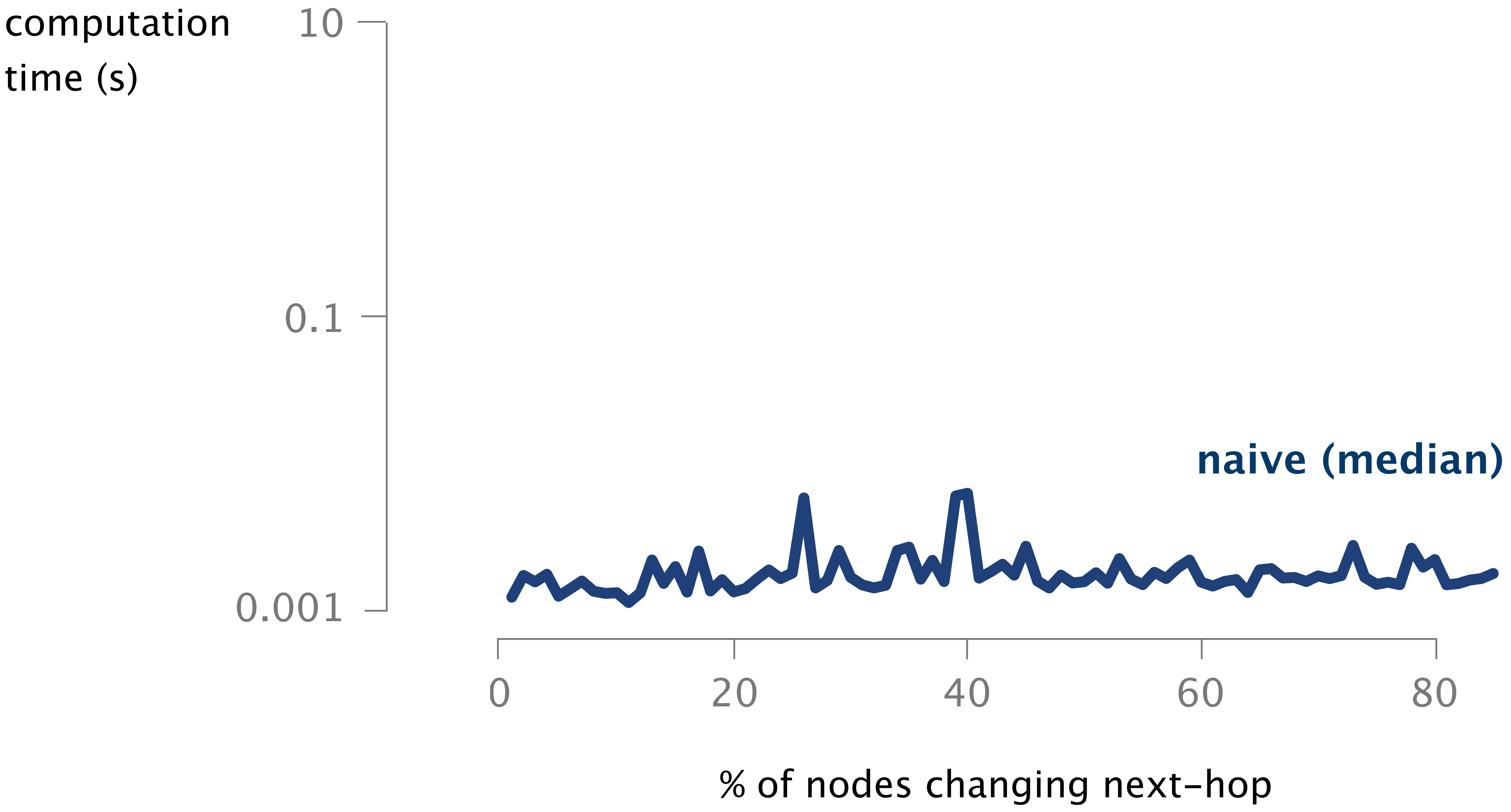
	Naive	Merger	Integer Linear Program
time	optimal	fast	slow
space (topology size)	large	small	optimal

Let’s compare the performance of Naive and Merger

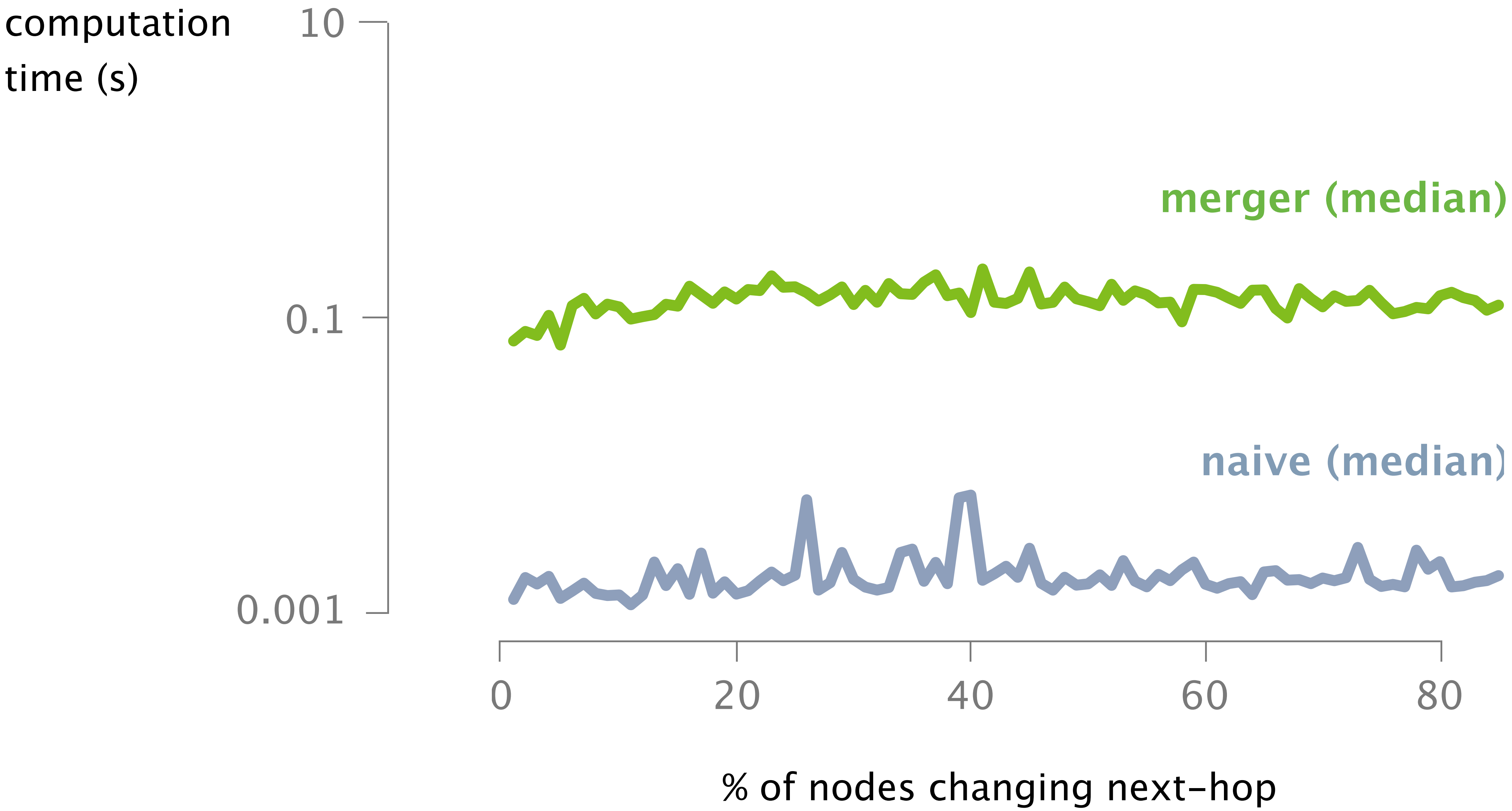
	Naive	Merger	Integer Linear Program
time	optimal	fast	slow
space (topology size)	large	small	optimal

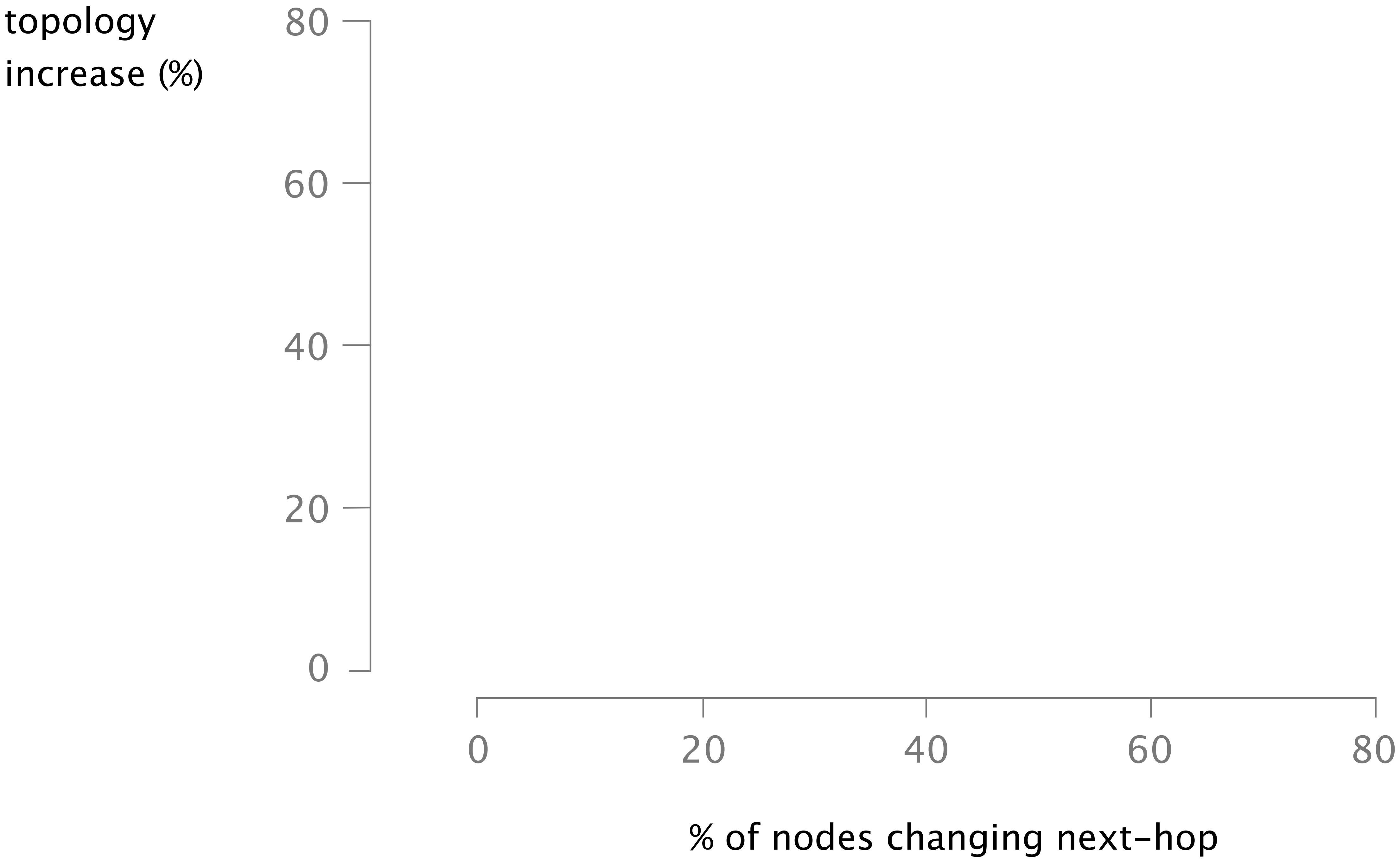

computation time (s)
10
0.1
0.001
0
20
40
60
80
% of nodes changing next-hop

Naive computes entire virtual topologies in ms

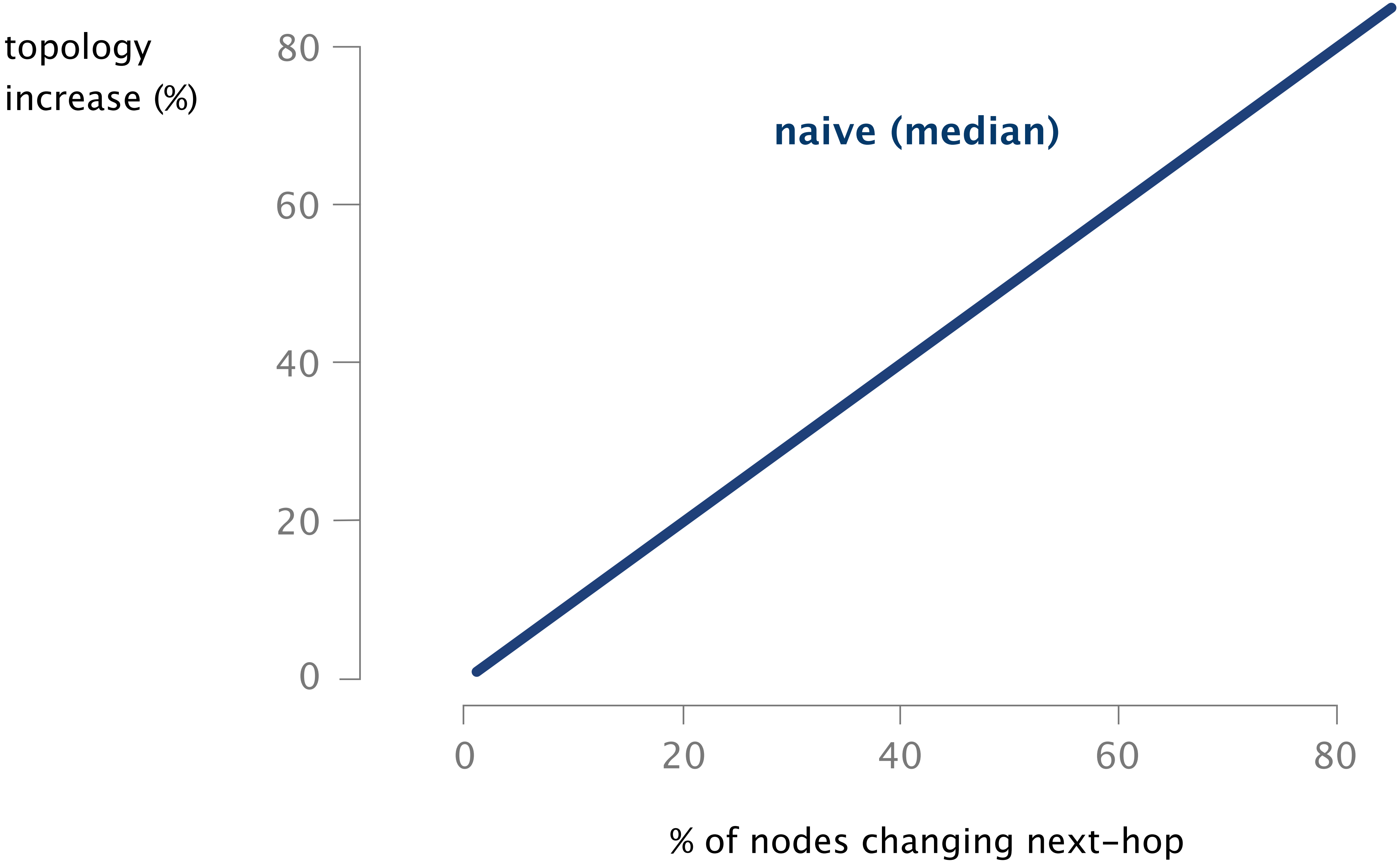


Merger is relatively slower, but still, sub-second

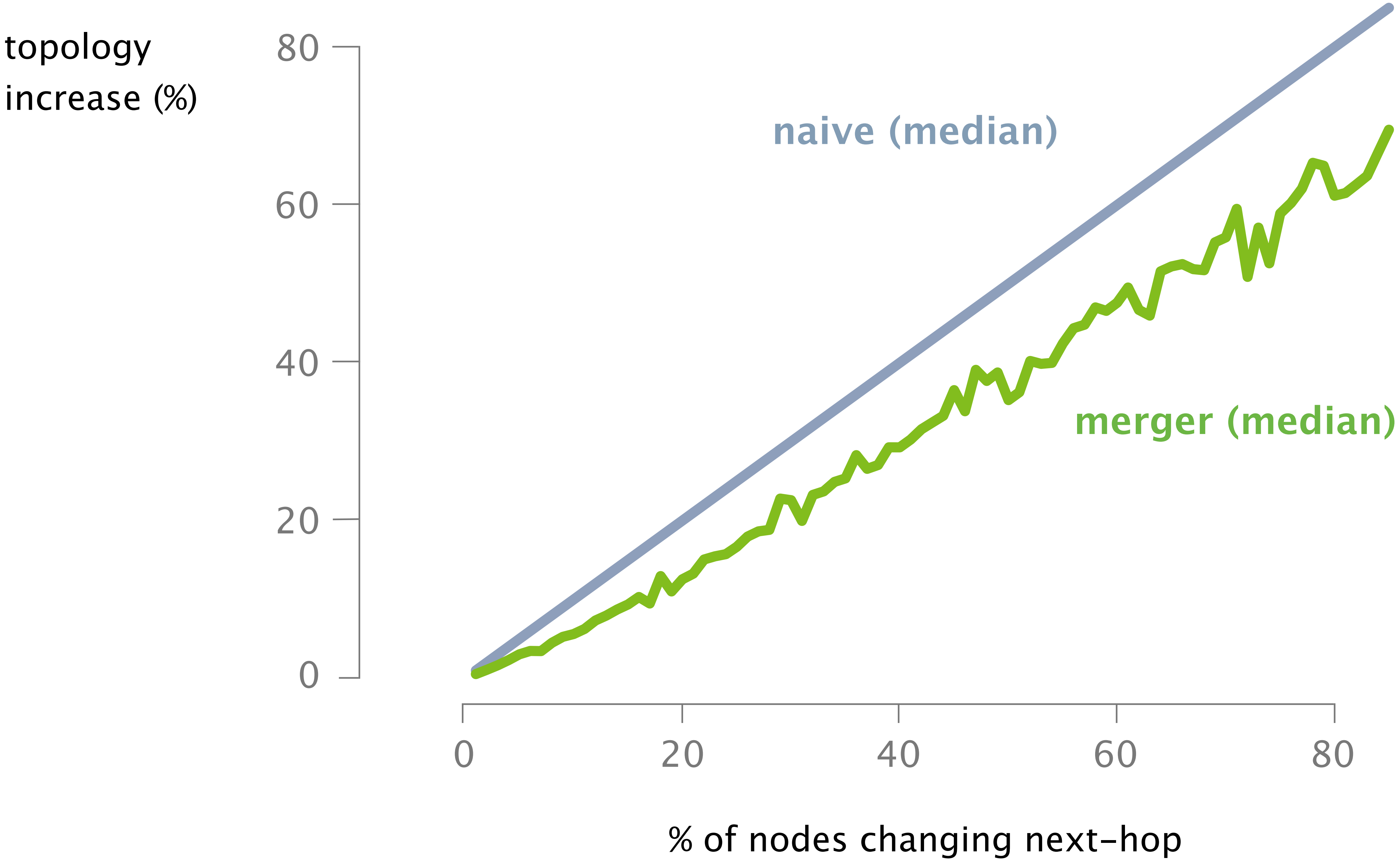



topology
increase (%)
80
60
40
20
0
0
20
40
60
80
% of nodes changing next-hop

Naive introduces one lie per changing next-hop



Merger reduces the size of the topology by 25% on average (50% in the best case)



We implemented a fully-fledged Fibbing prototype and tested it against real routers

We implemented a fully-fledged Fibbing prototype and tested it against real routers

2 measurements	How many lies can a router sustain?
	How long does it take to process a lie?

Existing routers can easily sustain Fibbing-induced load, even with huge topologies

# fake nodes	router memory (MB)	
1000	0.7	
5 000	6.8	
10 000	14.5	
50 000	76.0	
100 000	153	DRAM is cheap

Because it is entirely distributed, programming forwarding entries is fast

# fake nodes	installation time (s)	
1000	0.9	
5 000	4.5	
10 000	8.9	
50 000	44.7	
100 000	89.50	894.50 µs/entry

Central Control Over Distributed Routing

Fibbing

lying made useful

Expressivity

any path, anywhere

Scalability

1 lie is better than 2

Fibbing realizes some of the SDN promises today, on an existing network

Facilitate SDN deployment

SDN controller can program routers and SDN switches

Simplify controller implementation

most of the heavy work is still done by the routers

Maintain operators' mental model

good old protocols running, easier troubleshooting

Fibbing

improved flexibility

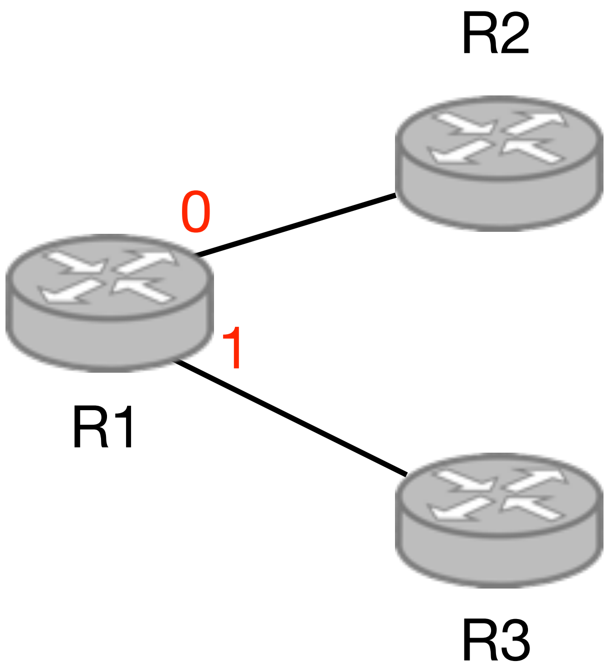
Supercharged

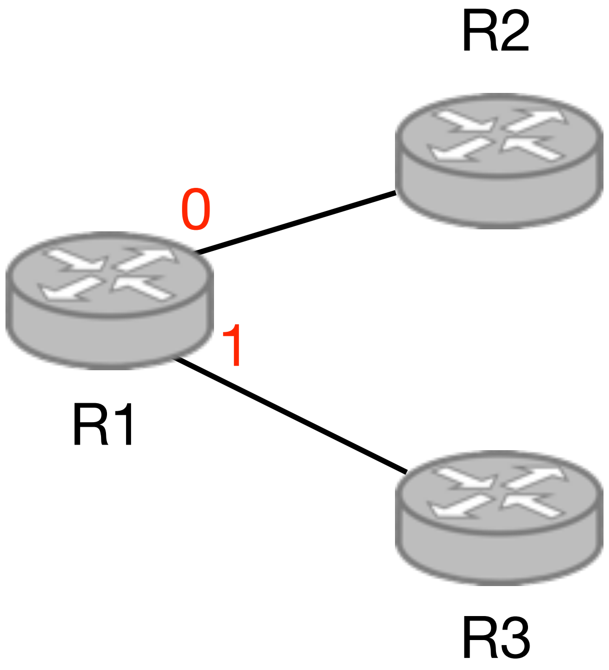
performance boost

reduce convergence time
by **1000x**

IP routers are pretty slow to converge upon link and node failures

R1


R2
0
1
R1
R3



Provider #1 ($)

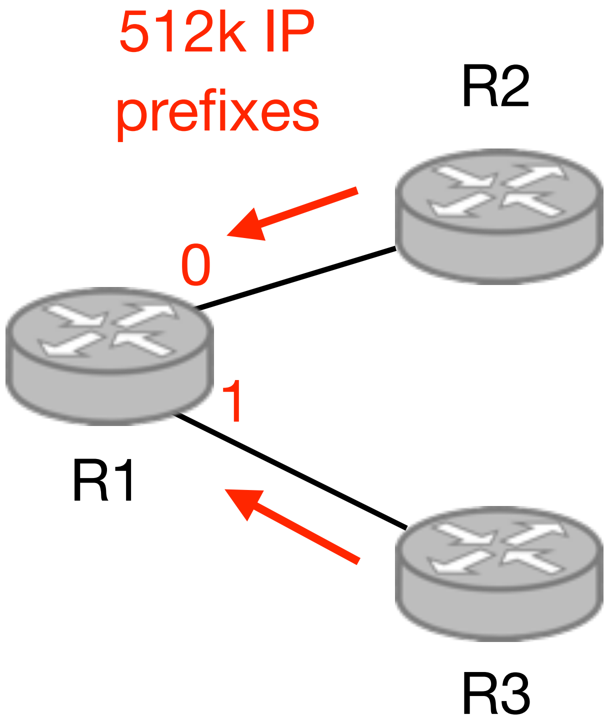
IP: 203.0.113.1

MAC: 01:aa

Provider #2 ($$)

IP: 198.51.100.2

MAC: 02:bb

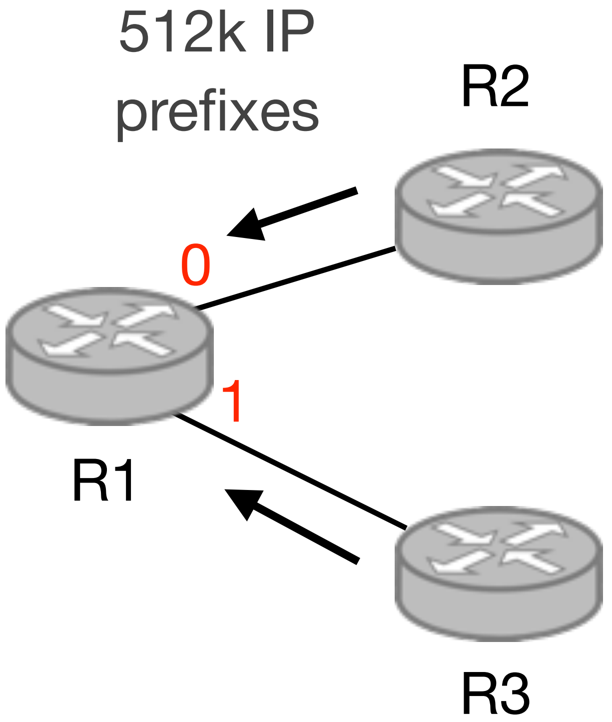


Provider #1 ($)
IP: 203.0.113.1
MAC: 01:aa

Provider #2 ($$)
IP: 198.51.100.2
MAC: 02:bb

R1's Forwarding Table

prefix	Next-Hop



Provider #1 ($)
IP: 203.0.113.1
MAC: 01:aa

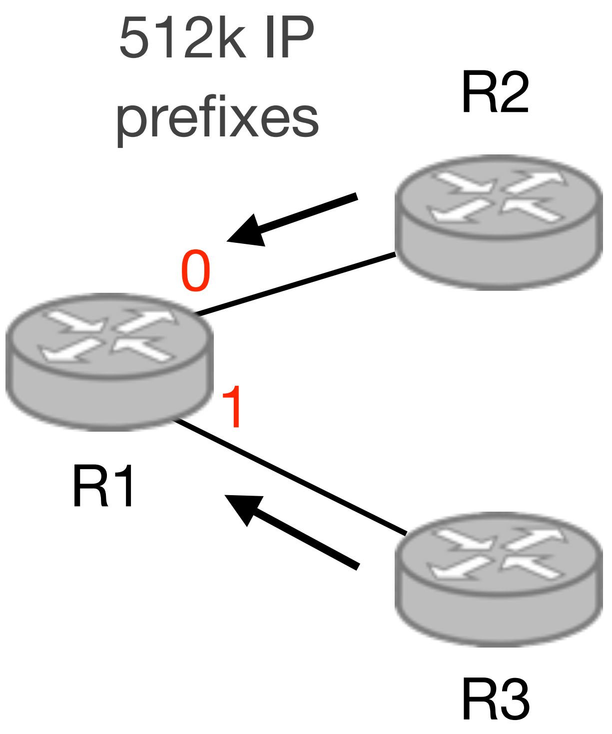
Provider #2 ($$)
IP: 198.51.100.2
MAC: 02:bb

All 512k entries point to R2

because it is cheaper

R1's Forwarding Table

	prefix	Next-Hop
1	1.0.0.0/24	(01:aa, 0)
2	1.0.1.0/16	(01:aa, 0)
…	…	…
256k	100.0.0.0/8	(01:aa, 0)
…	…	…
512k	200.99.0.0/24	(01:aa, 0)



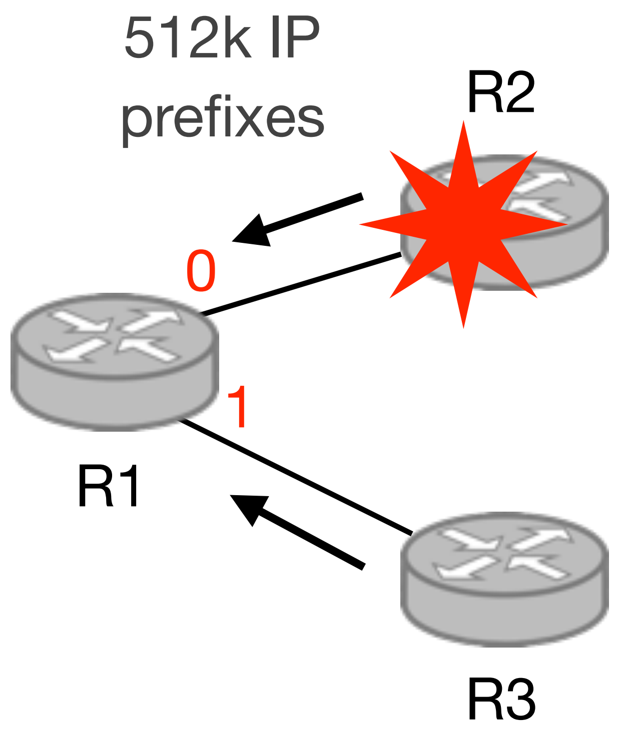
Provider #1 ($)
IP: 203.0.113.1
MAC: 01:aa

Provider #2 ($$)
IP: 198.51.100.2
MAC: 02:bb

Upon failure of R2,
all 512k entries have to be updated

R1's Forwarding Table

	prefix	Next-Hop
1	1.0.0.0/24	(01:aa, 0)
2	1.0.1.0/16	(01:aa, 0)
…	…	…
256k	100.0.0.0/8	(01:aa, 0)
…	…	…
512k	200.99.0.0/24	(01:aa, 0)



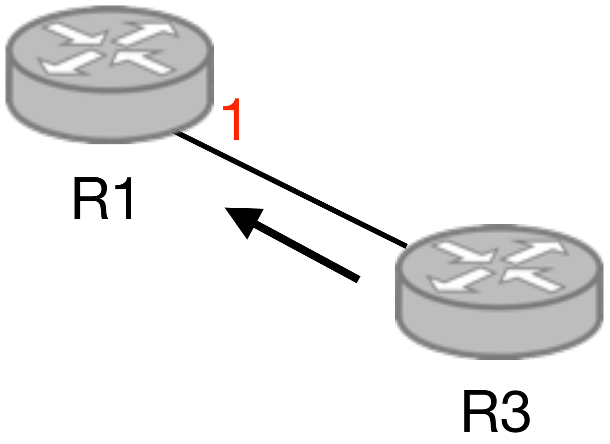
Provider #1 ($)
IP: 203.0.113.1
MAC: 01:aa

Provider #2 ($$)
IP: 198.51.100.2
MAC: 02:bb

Upon failure of R2,
all 512k entries have to be updated

R1's Forwarding Table

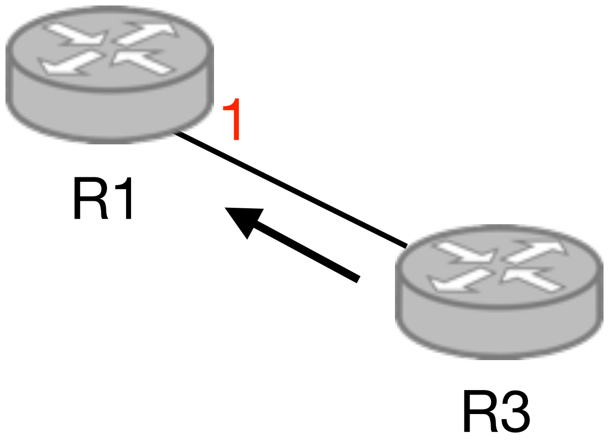
	prefix	Next-Hop
1	1.0.0.0/24	(01:aa, 0)
2	1.0.1.0/16	(01:aa, 0)
…	…	…
256k	100.0.0.0/8	(01:aa, 0)
…	…	…
512k	200.99.0.0/24	(01:aa, 0)



Provider #2 ($$)
IP: 198.51.100.2
MAC: 02:bb

R1’s Forwarding Table

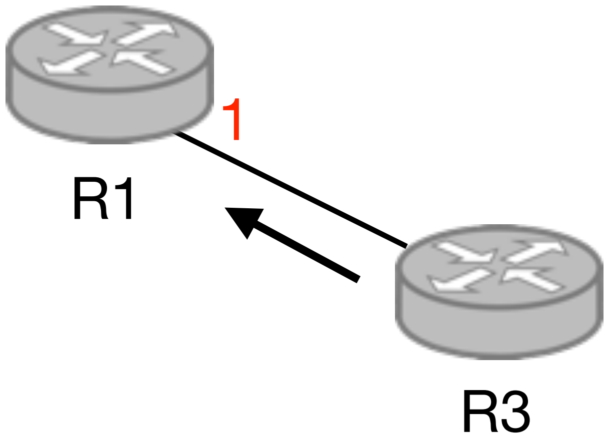
	prefix	Next-Hop
1	1.0.0.0/24	(02:bb, 1)
2	1.0.1.0/16	(01:aa, 0)
…	…	…
256k	100.0.0.0/8	(01:aa, 0)
…	…	…
512k	200.99.0.0/24	(01:aa, 0)



Provider #2 ($$)
IP: 198.51.100.2
MAC: 02:bb

R1's Forwarding Table

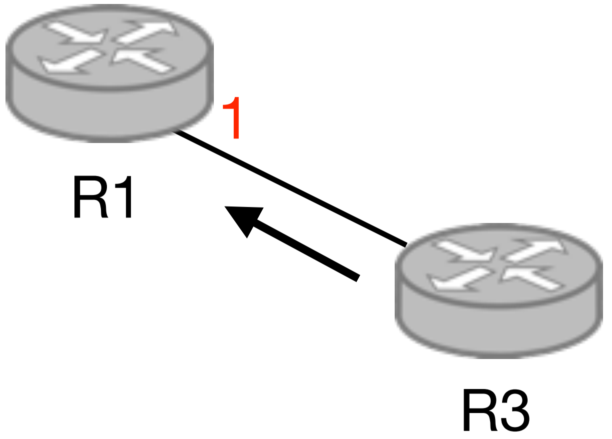
	prefix	Next-Hop
1	1.0.0.0/24	(02:bb, 1)
2	1.0.1.0/16	(02:bb, 1)
…	…	…
256k	100.0.0.0/8	(01:aa, 0)
…	…	…
512k	200.99.0.0/24	(01:aa, 0)



Provider #2 ($$)
IP: 198.51.100.2
MAC: 02:bb

R1's Forwarding Table

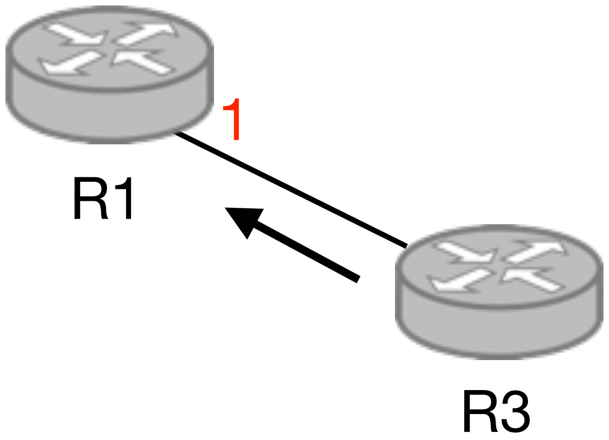
	prefix	Next-Hop
1	1.0.0.0/24	(02:bb, 1)
2	1.0.1.0/16	(02:bb, 1)
…	…	…
256k	100.0.0.0/8	(02:bb, 1)
…	…	…
512k	200.99.0.0/24	(01:aa, 0)



Provider #2 ($$)
IP: 198.51.100.2
MAC: 02:bb

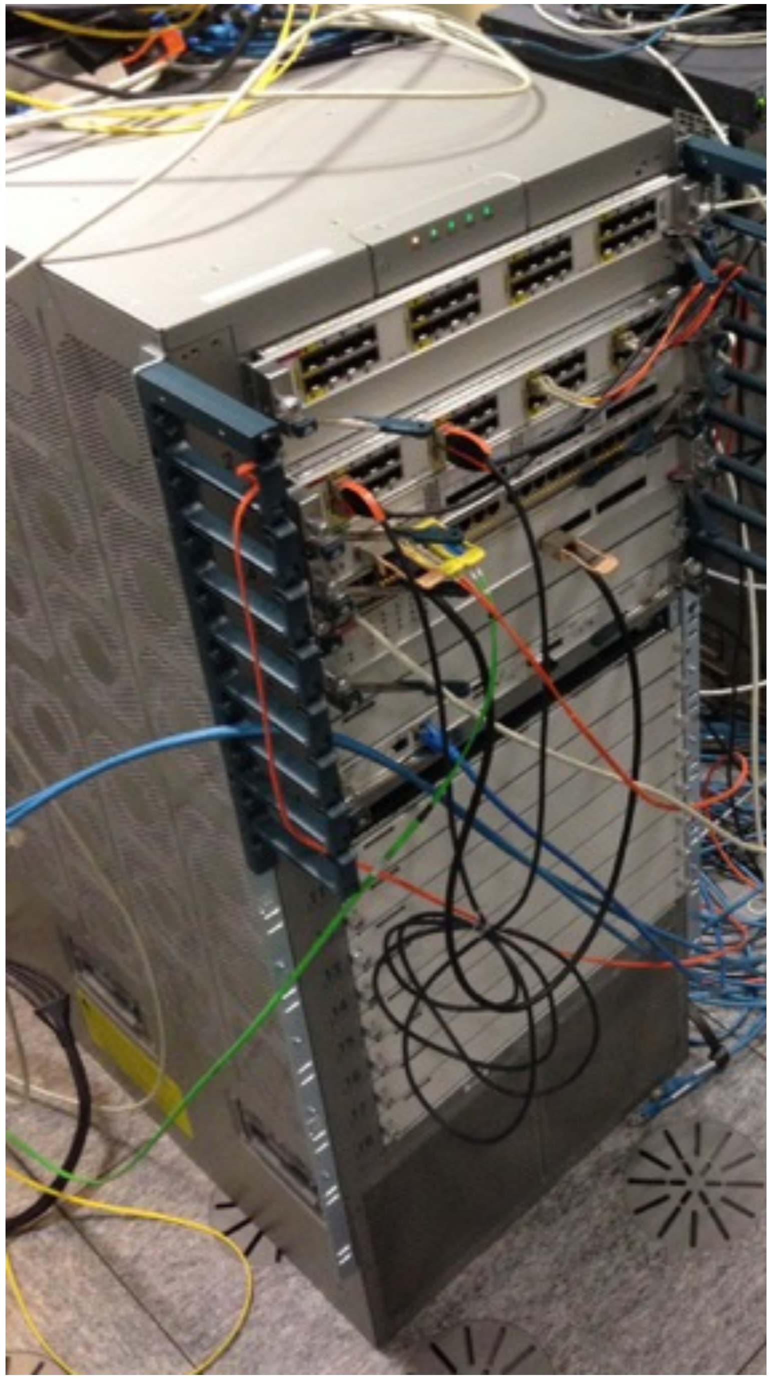
R1's Forwarding Table

	prefix	Next-Hop
1	1.0.0.0/24	(02:bb, 1)
2	1.0.1.0/16	(02:bb, 1)
…	…	…
256k	100.0.0.0/8	(02:bb, 1)
…	…	…
512k	200.99.0.0/24	(02:bb, 1)



Provider #2 ($$)
IP: 198.51.100.2
MAC: 02:bb

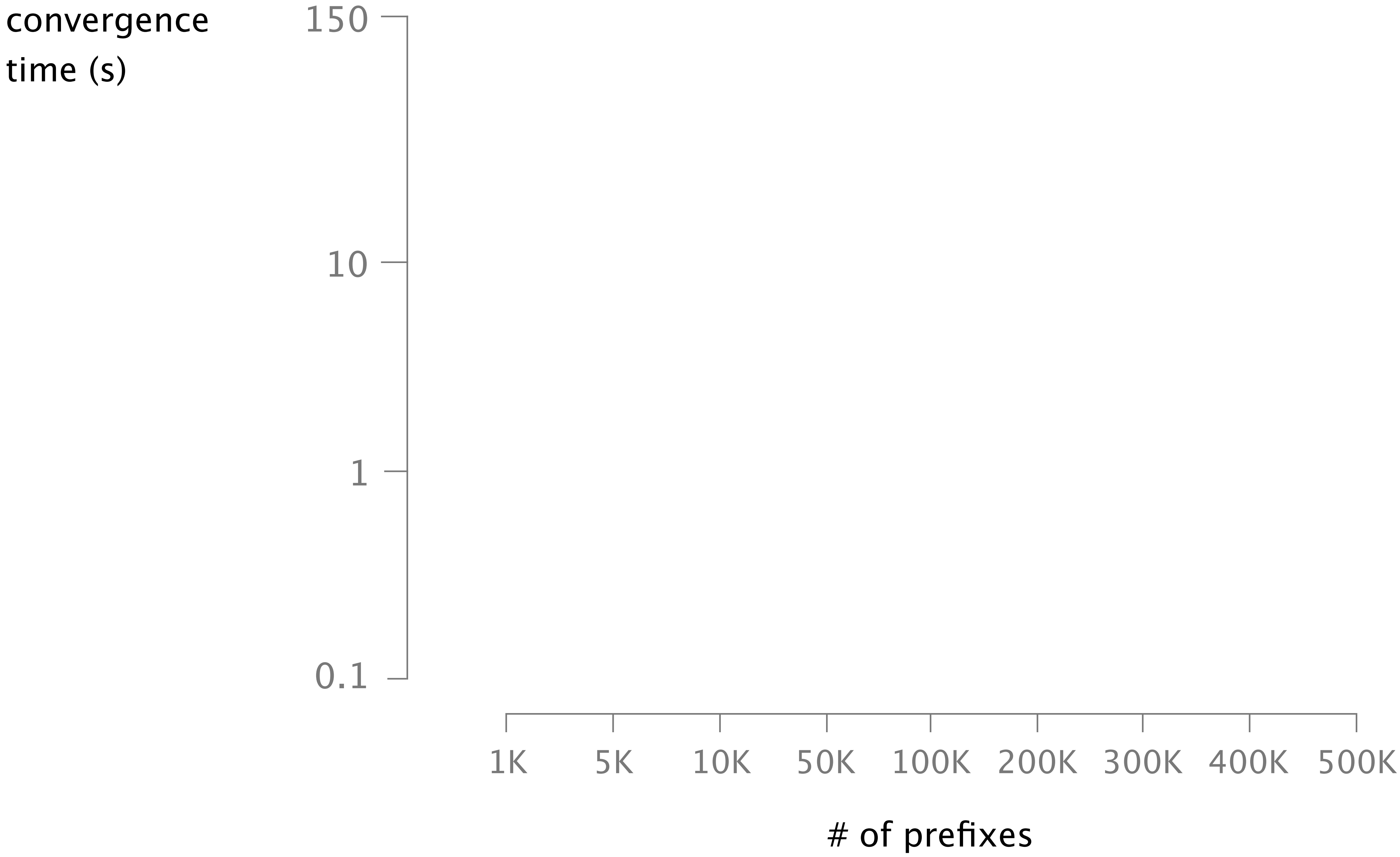
We measured how long it takes in our home network

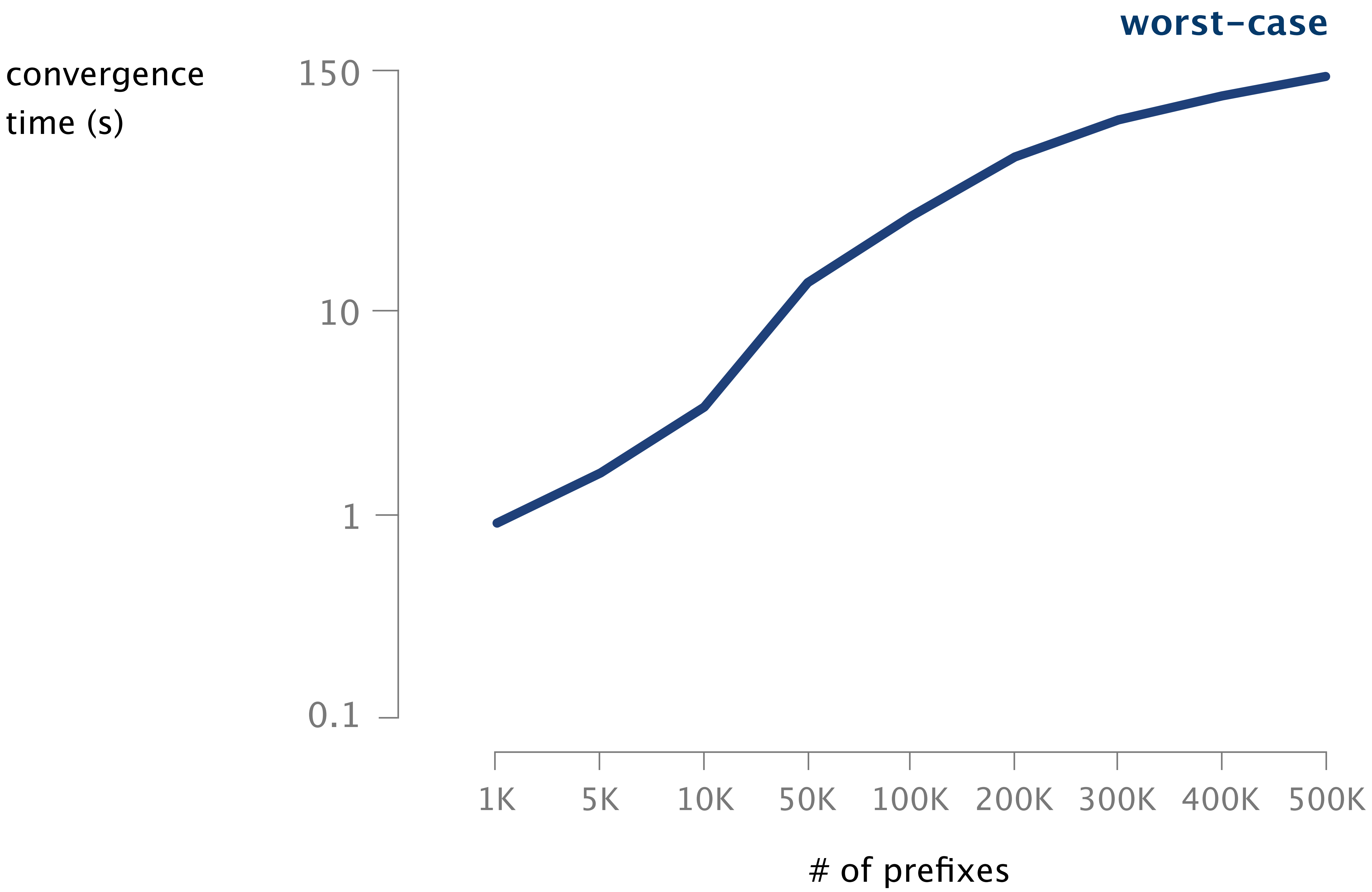


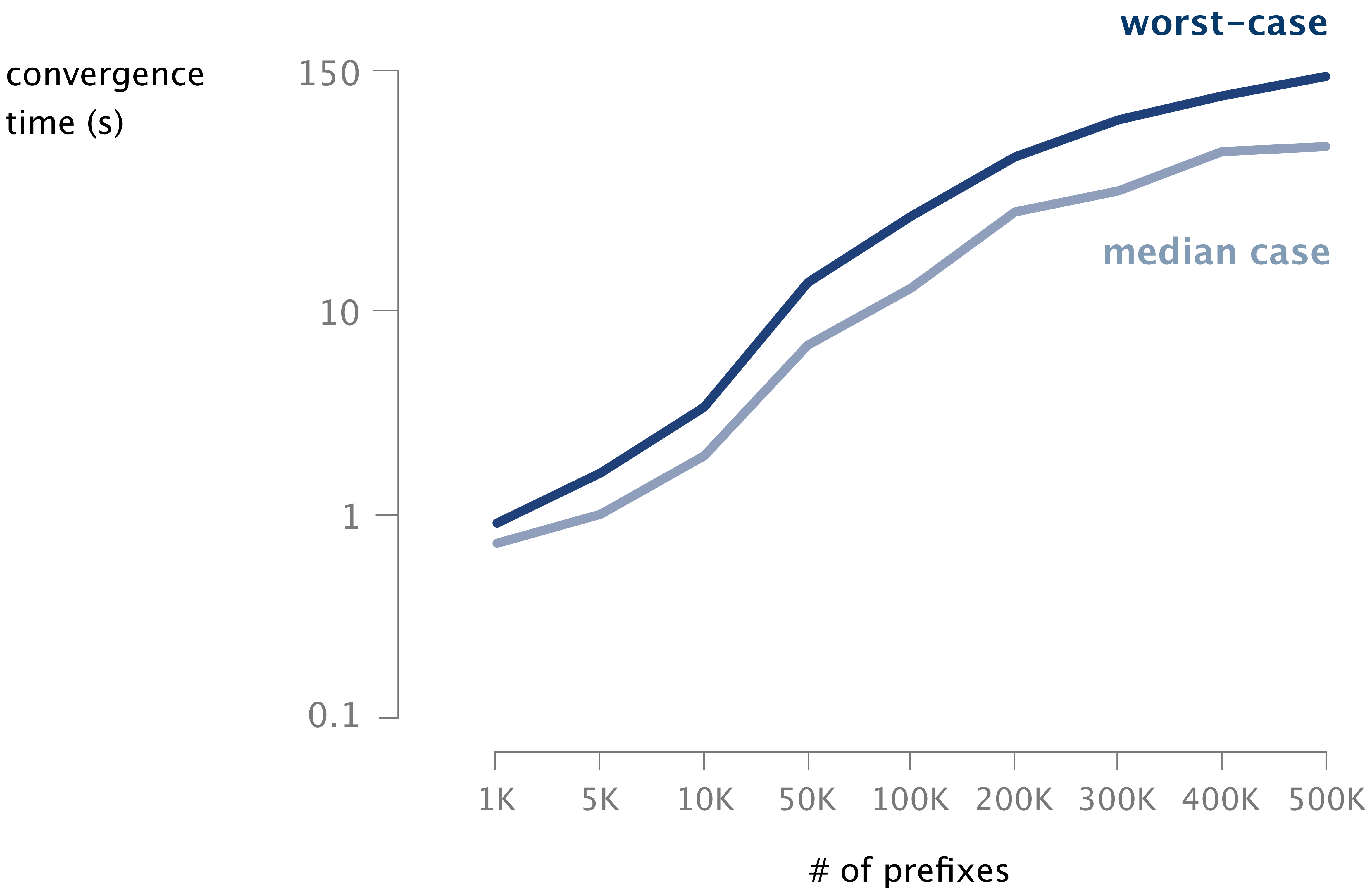
Cisco Nexus 9k

ETH recent routers

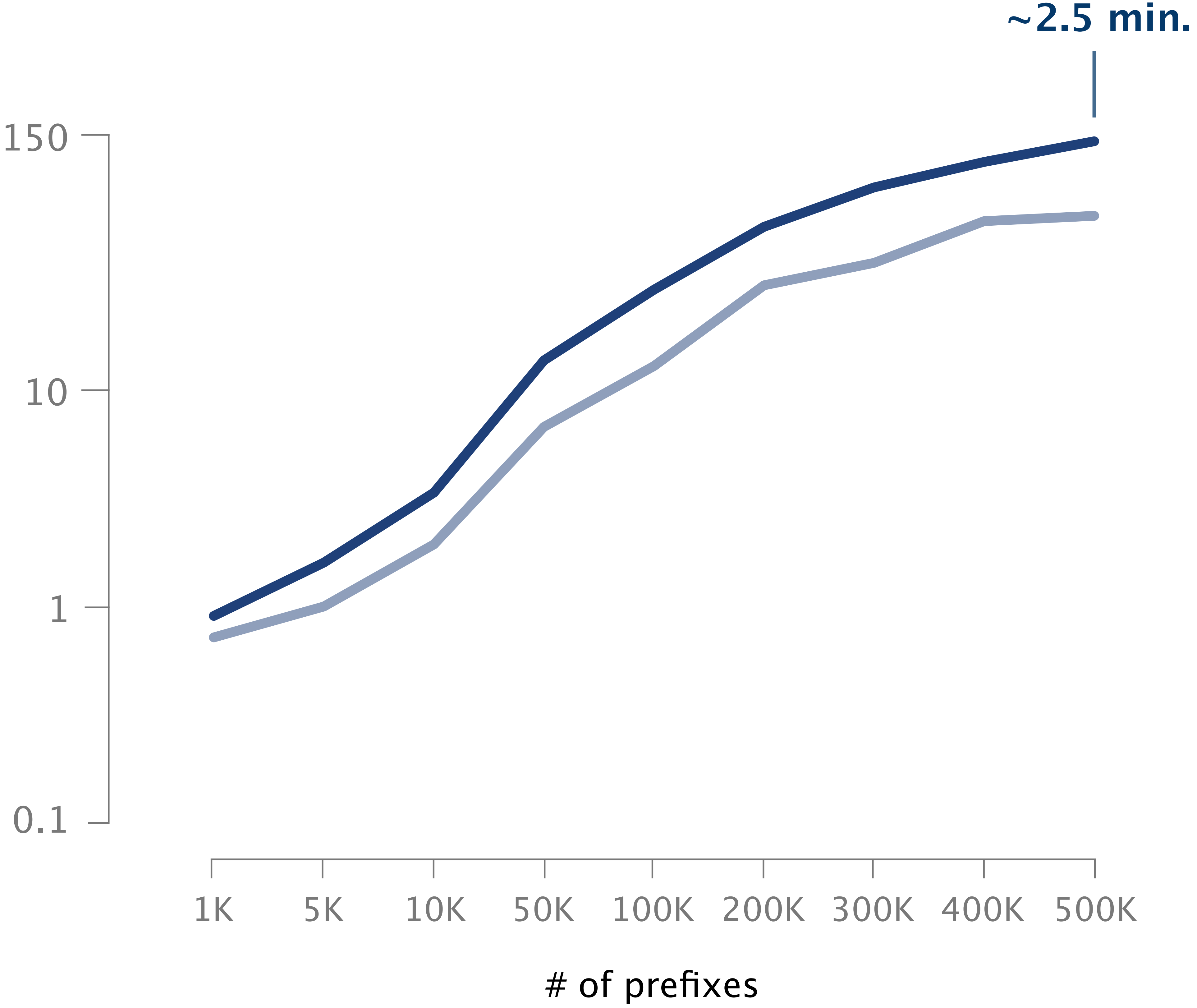
25	deployed
1M$	cost


convergence time (s)
150
10
1
0.1
1K
5K
10K
50K
100K
200K
300K
400K
500K
# of prefixes


worst-case
convergence
time (s)
150
10
1
0.1
1K
5K
10K
50K
100K
200K
300K
400K
500K
# of prefixes


worst-case
convergence time (s)
150
10
1
0.1
median case
1K
5K
10K
50K
100K
200K
300K
400K
500K
# of prefixes

Traffic can be lost for several minutes



The problem is that forwarding tables are flat

Entries do not share any information

even if they are identical

Upon failure, all of them have to be updated

inefficient, but also unnecessary

The problem is that forwarding tables are flat

Entries do not share any information

even if they are identical

Upon failure, all of them have to be updated

inefficient, but also unnecessary

Solution: introduce a hierarchy

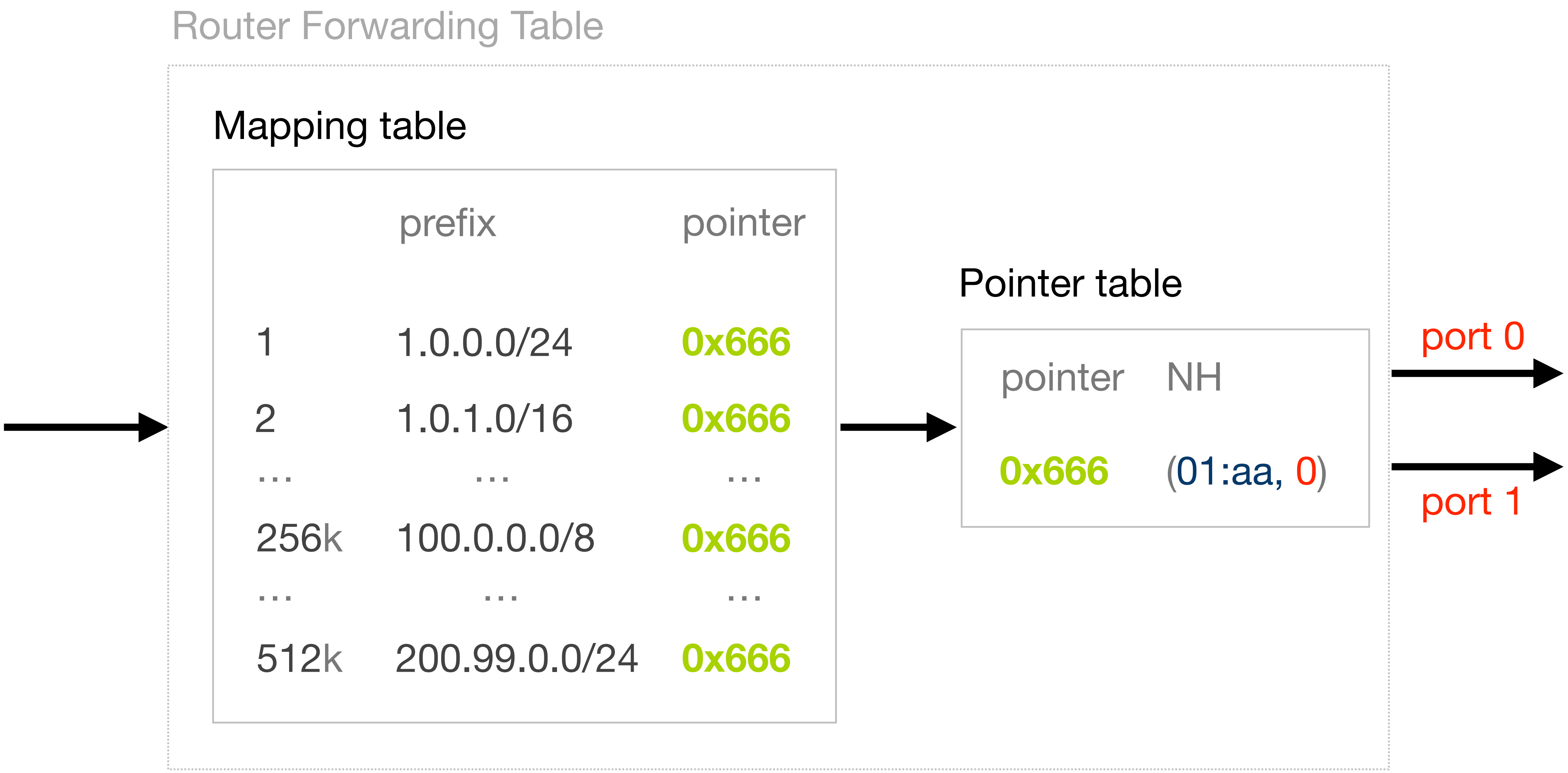
as with any problem in CS…

replace this…

Router Forwarding Table

	prefix	Next-Hop
1	1.0.0.0/24	(01:aa, 0)
2	1.0.1.0/16	(01:aa, 0)
…	…	…
256k	100.0.0.0/8	(01:aa, 0)
…	…	…
512k	200.99.0.0/24	(01:aa, 0)

… with that



Router Forwarding Table

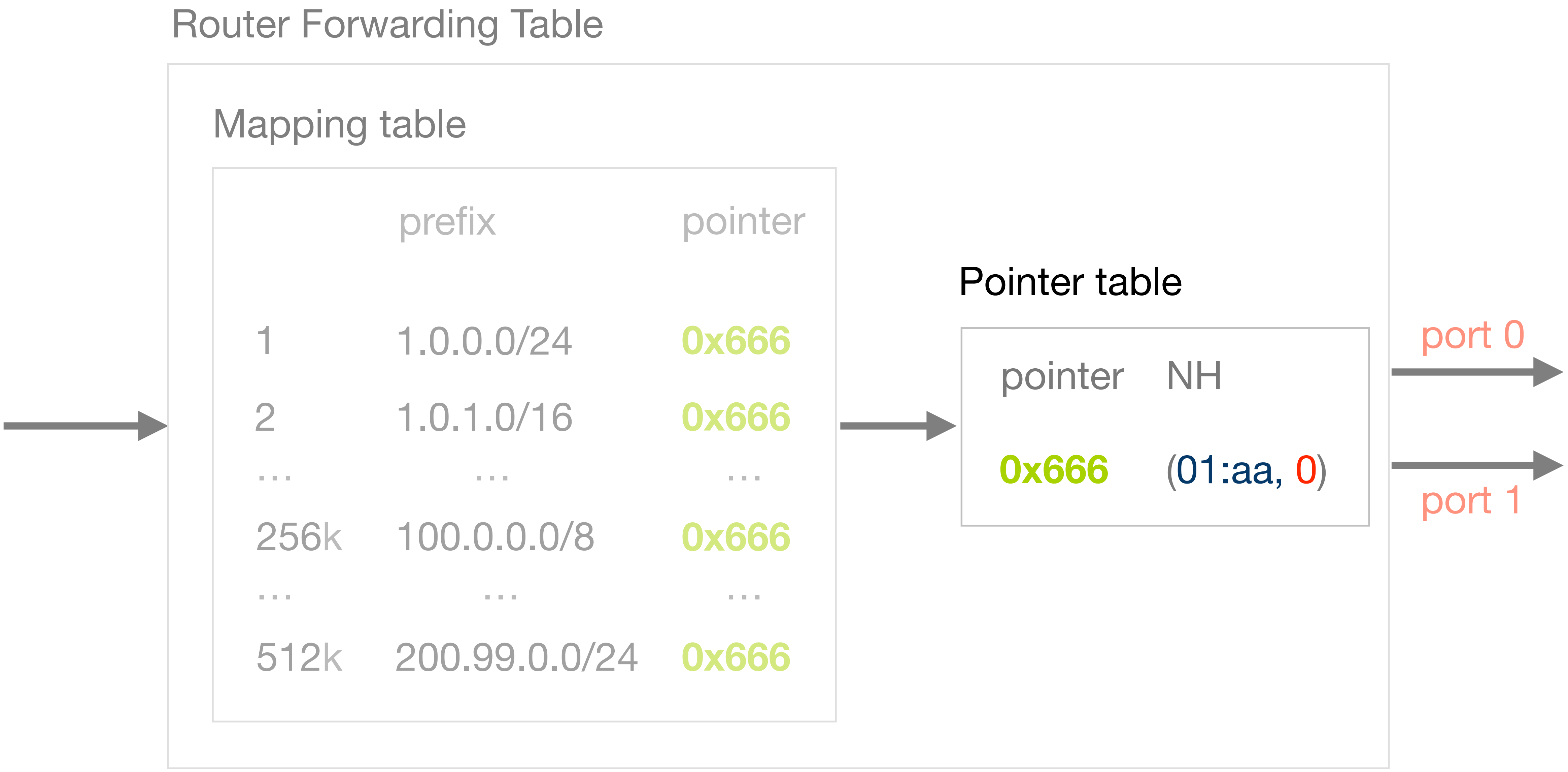
Mapping table

	prefix	pointer
1	1.0.0.0/24	0x666
2	1.0.1.0/16	0x666
…	…	…
256k	100.0.0.0/8	0x666
…	…	…
512k	200.99.0.0/24	0x666

Pointer table

pointer	NH
0x666	(01:aa, 0)

Upon failures, we update the pointer table



Router Forwarding Table

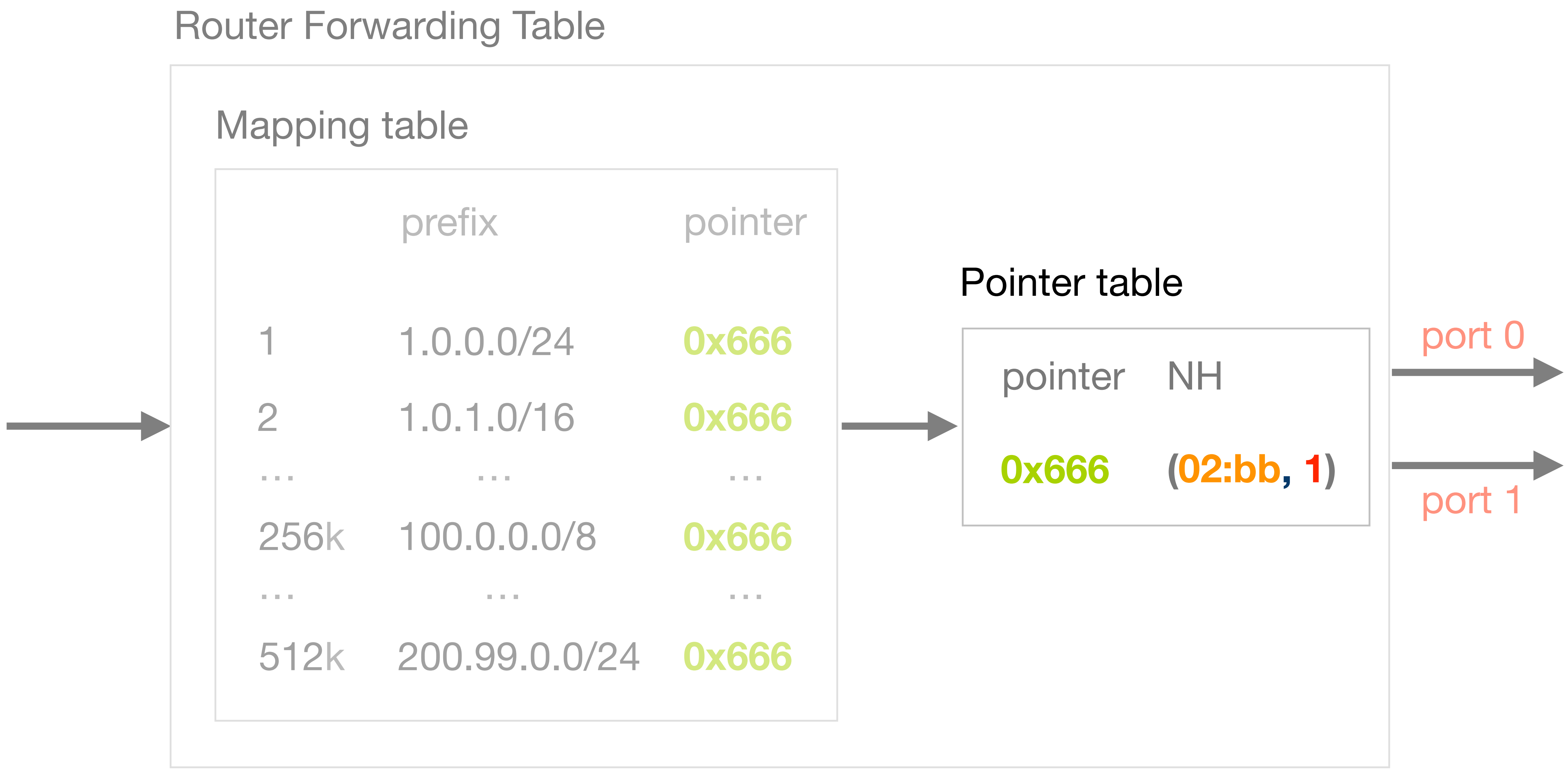
Mapping table

	prefix	pointer
1	1.0.0.0/24	0x666
2	1.0.1.0/16	0x666
…	…	…
256k	100.0.0.0/8	0x666
…	…	…
512k	200.99.0.0/24	0x666

Pointer table

pointer	NH
0x666	(01:aa, 0)

Here, we only need to do one update



Router Forwarding Table

Mapping table

	prefix	pointer
1	1.0.0.0/24	0x666
2	1.0.1.0/16	0x666
…	…	…
256k	100.0.0.0/8	0x666
…	…	…
512k	200.99.0.0/24	0x666

Pointer table

pointer	NH
0x666	(02:bb, 1)

Nowadays, only high-end routers have hierarchical forwarding table

Expensive

by orders of magnitude

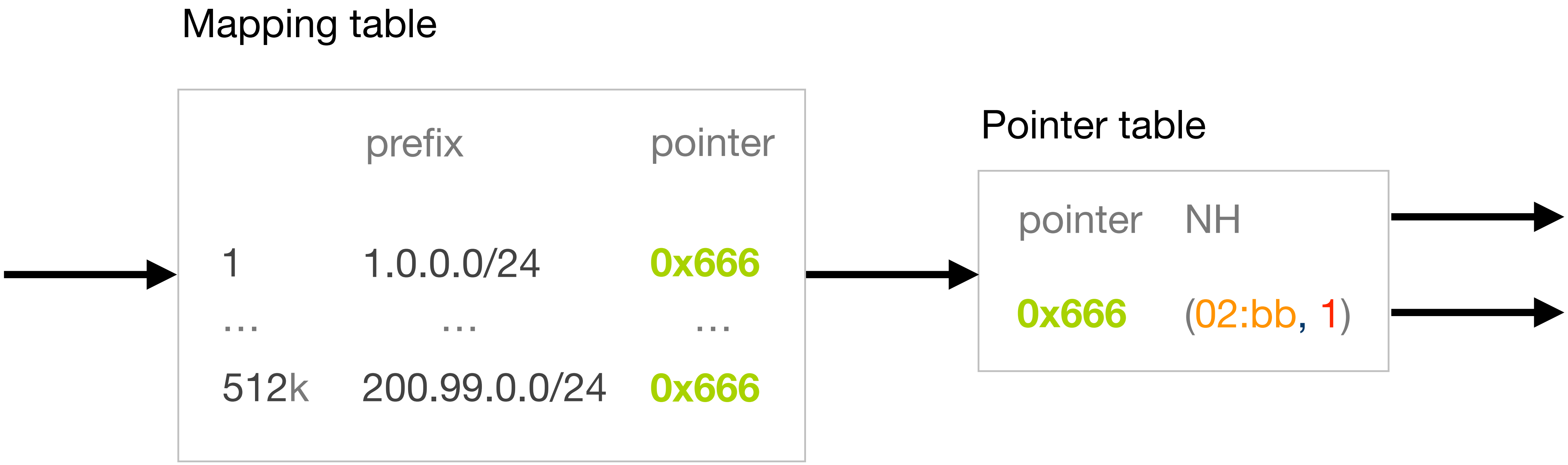
Limited availability

only a few vendors, on few models

Limited benefits

of fast convergence, if not used network-wide

We can build a hierarchical table



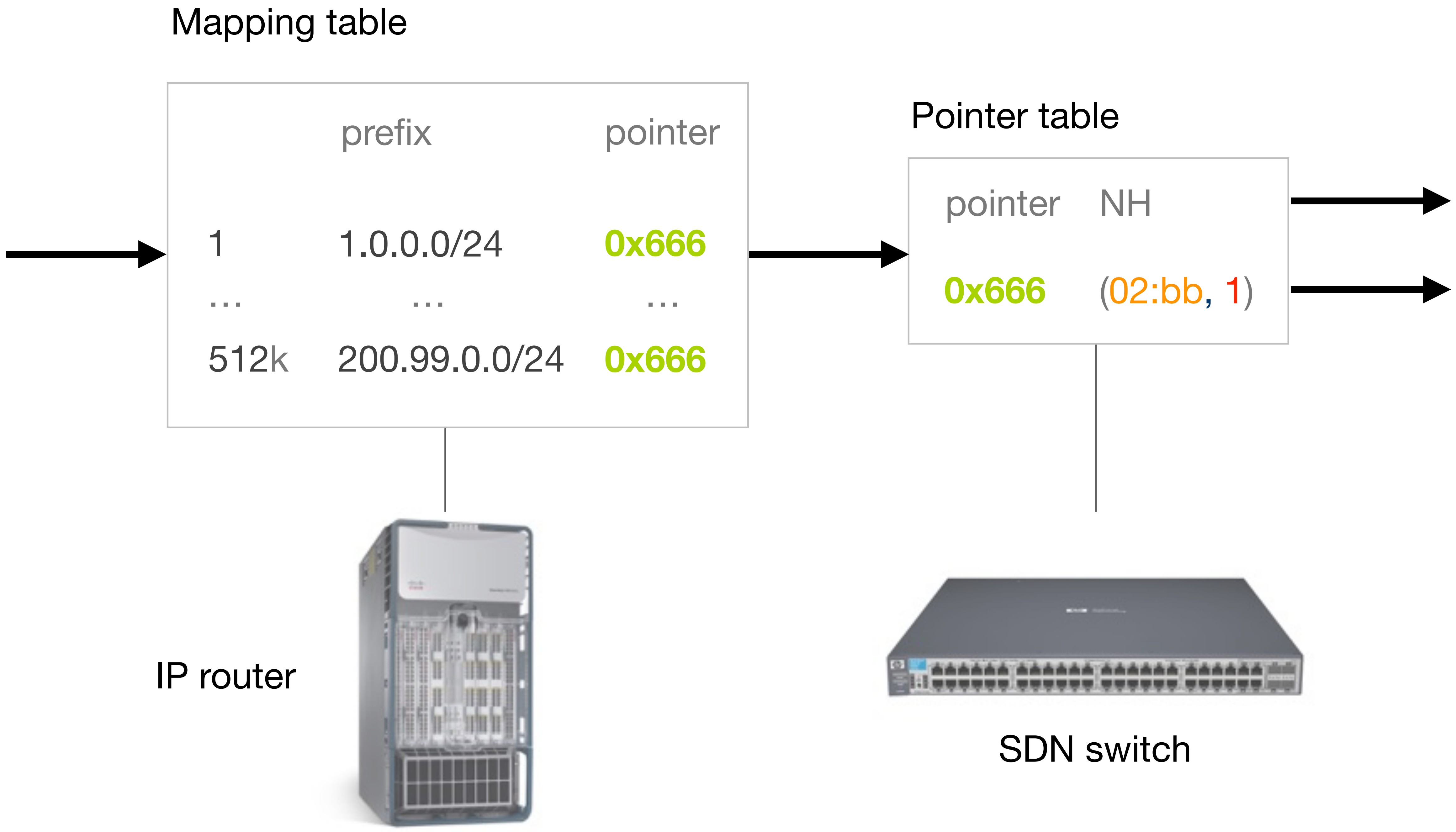
Mapping table

	prefix	pointer
1	1.0.0.0/24	**0x666**
…	…	…
512k	200.99.0.0/24	**0x666**

Pointer table

pointer	NH
0x666	(02:bb, 1)

We can build a hierarchical table using two adjacent devices



Mapping table

	prefix	pointer
1	1.0.0.0/24	0x666
…	…	…
512k	200.99.0.0/24	0x666

Pointer table

pointer	NH
0x666	(02:bb, 1)

IP router

SDN switch

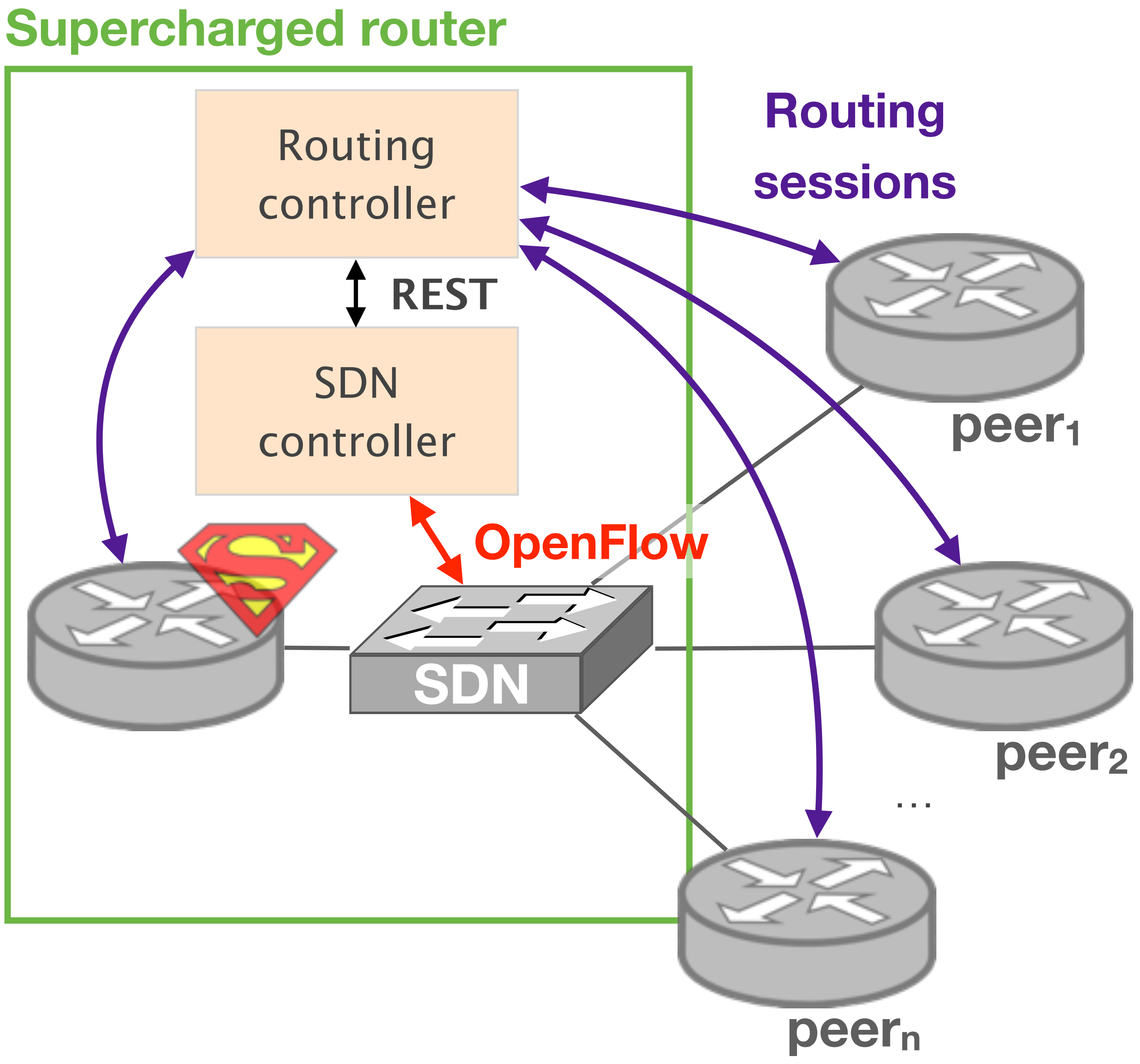
Supercharged

Supercharged

boost routers performance

by **combining** them with **SDN** devices

We have implemented a fully-functional "router supercharger"



Supercharged router

Routing controller

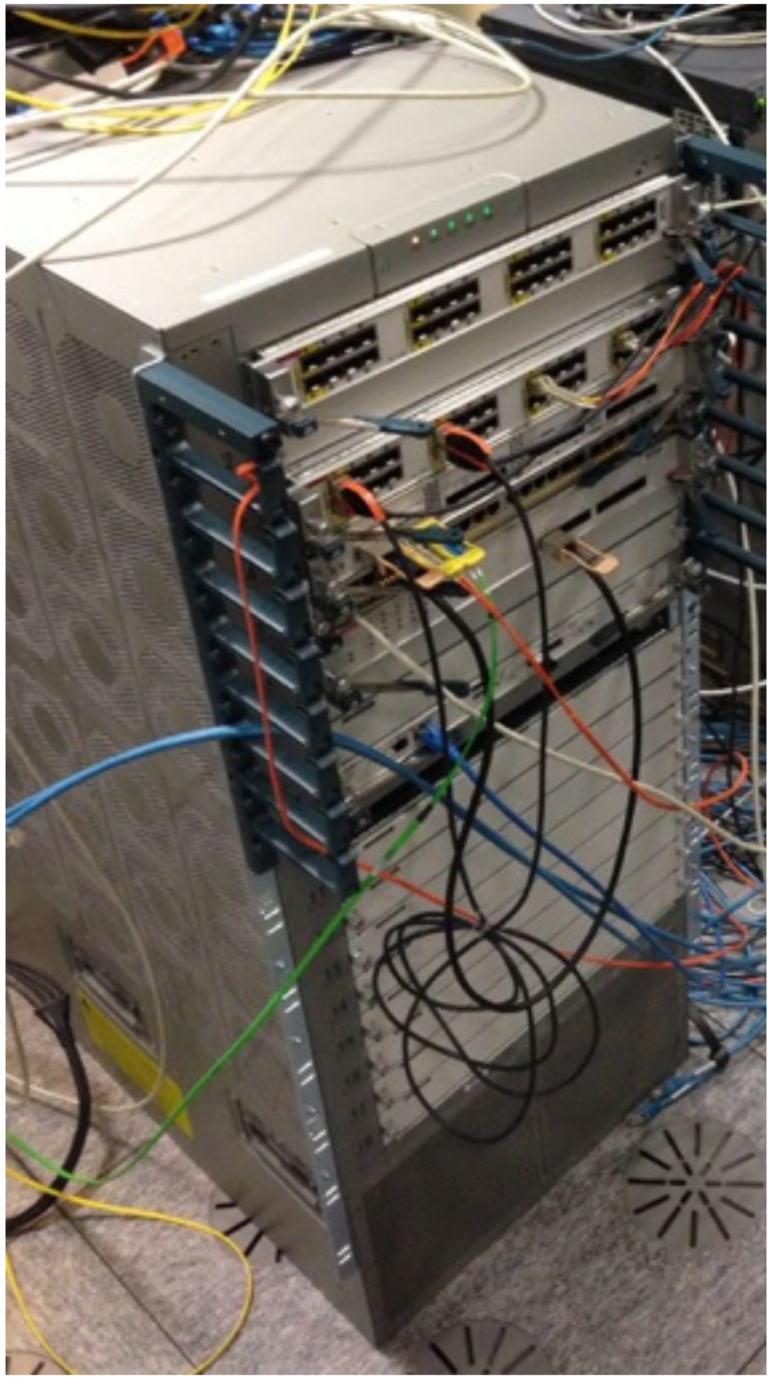
REST

SDN controller

OpenFlow

Routing sessions

We used it to supercharge the same router as before



Cisco Nexus 9k

ETH recent routers

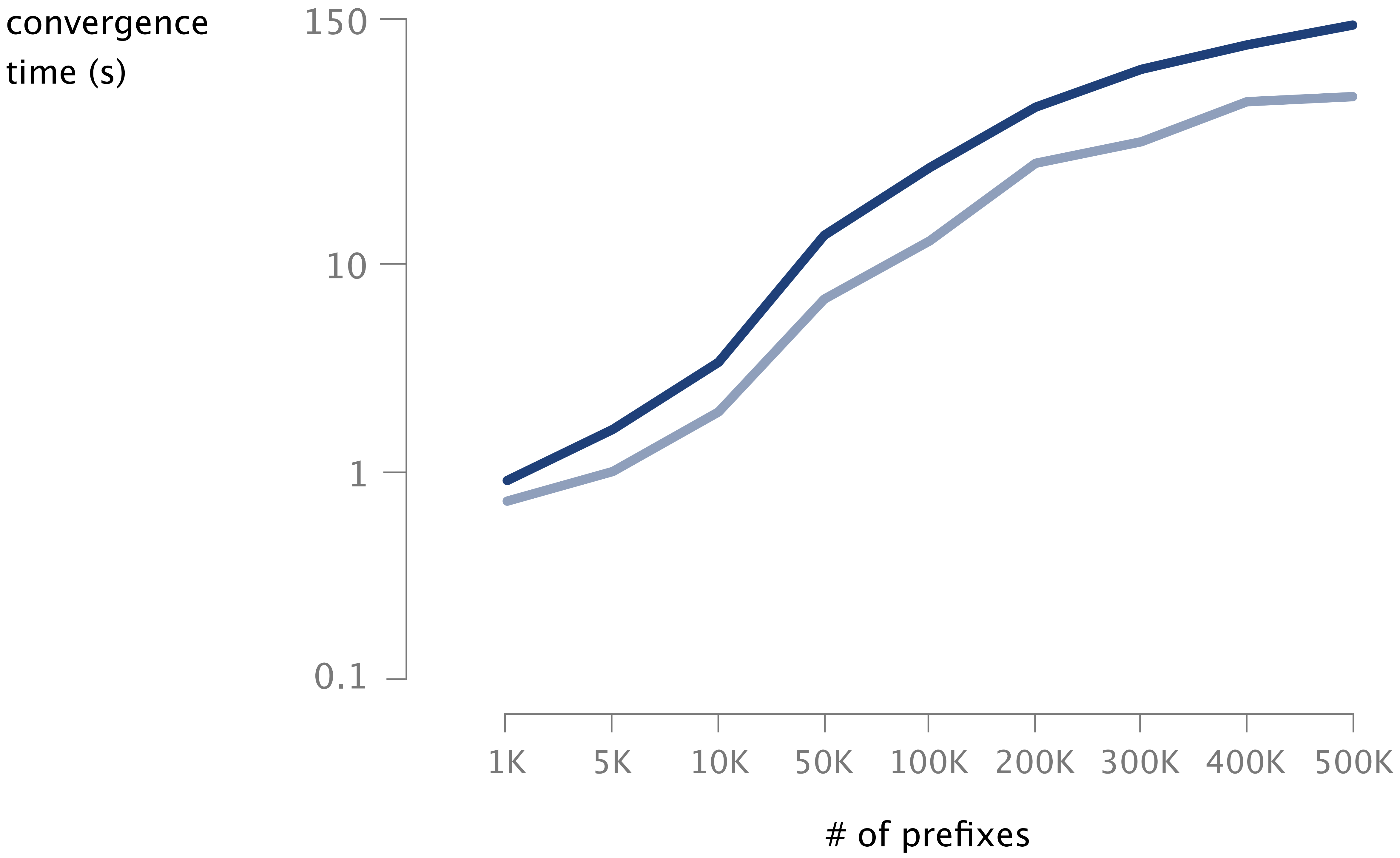
25 deployed

1M$ cost

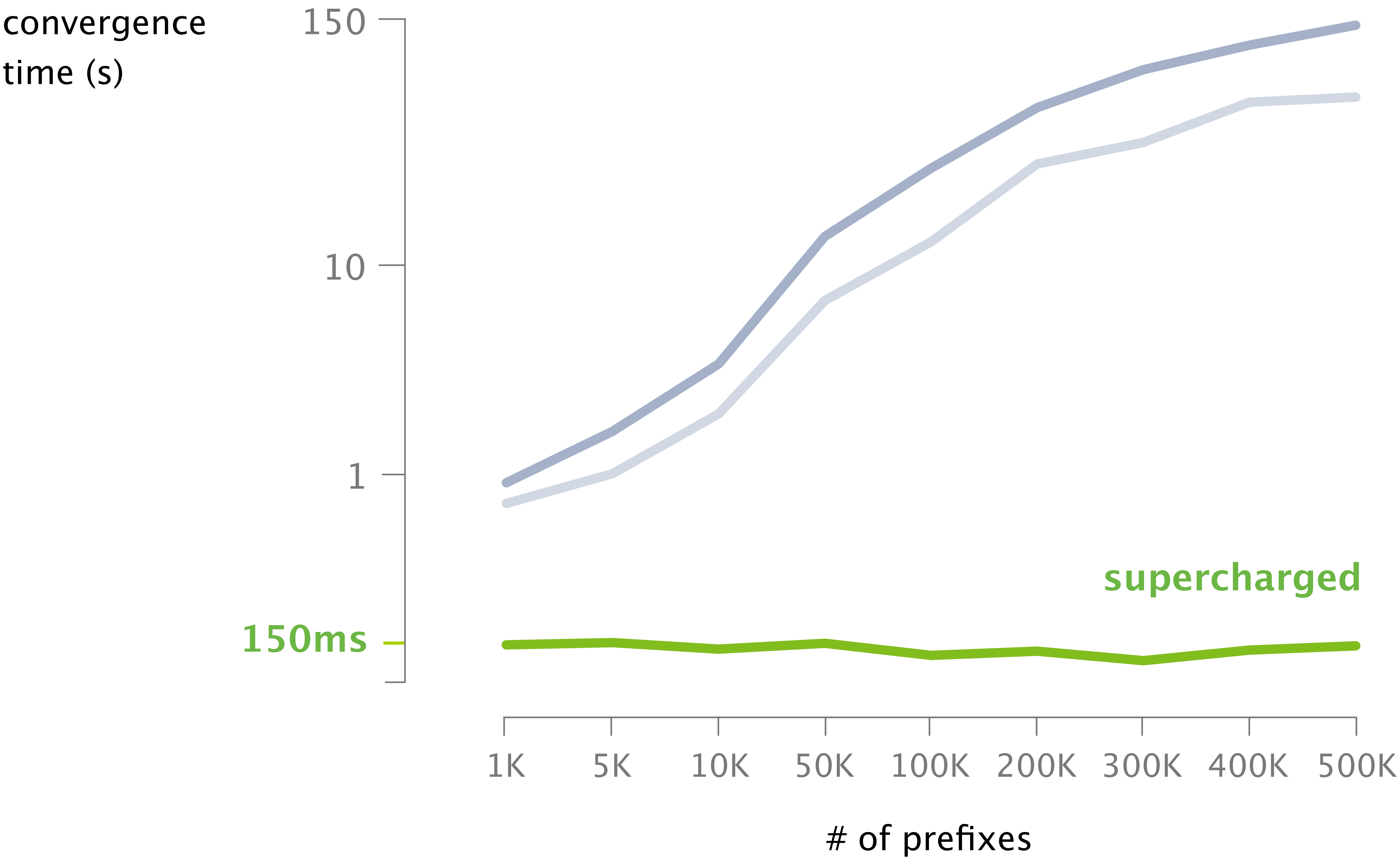
\+ (old) SDN HP switch

~2k$ cost

While the router took more than 2 min to converge in the worst-case



The supercharged router systematically converged within 150ms



Other aspects of a router performance can be supercharged

- convergence time

  systematic sub-second convergence

- memory size

  offload to SDN if no local forwarding entry

- bandwidth management

  overwrite poor routers decisions

This talk was about two SDN-based technologies that improve today's networks

Fibbing

improved flexibility

central control over distributed system

Supercharged

performance boost

reduce convergence time by 1000x

Boosting existing networks with SDN

A bird in the hand is worth two in the bush

Laurent Vanbever

www.vanbever.eu

Hebrew U. net. seminar

June, 9 2015