Network Control Planes

What? How? Where?

Laurent Vanbever

nsg.ee.ethz.ch

Dagstuhl Seminar

Wed Apr 3 2019

Network Control Planes

What? How? Where?

Network Control Planes

What? How? Where?

Main tasks

#1 provisioning policies
specifying user intents

#2 computing forwarding state
deriving compliant paths

#3 maintaining topology information
failure detection, topology discovery, etc.

#4 collecting statistics
flow-level, router-level, network-level

Main tasks

#1 **provisioning** policies

specifying user intents

#2 **computing** forwarding state

deriving compliant paths

#3 maintaining **topology** information

failure detection, topology discovery, etc.

#4 collecting **statistics**

flow-level, router-level, network-level

Network Control Planes

What? How? Where?

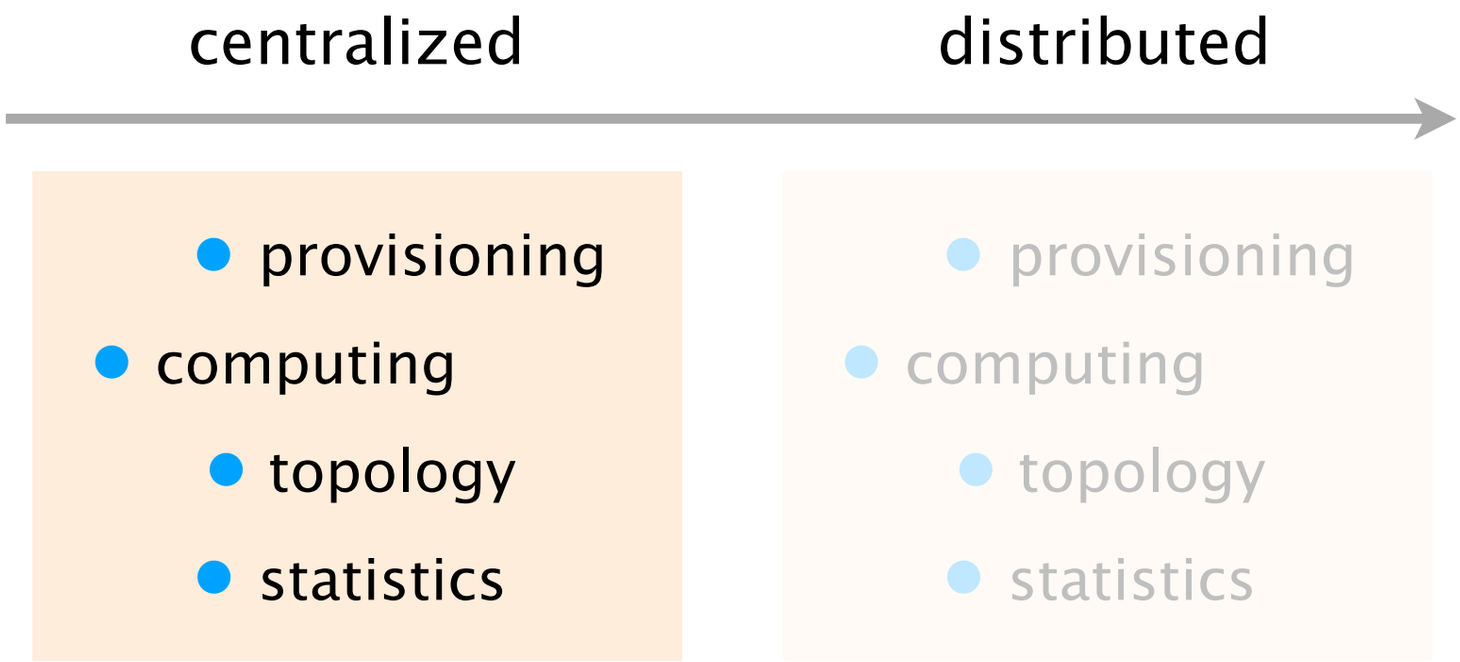

centralized
distributed

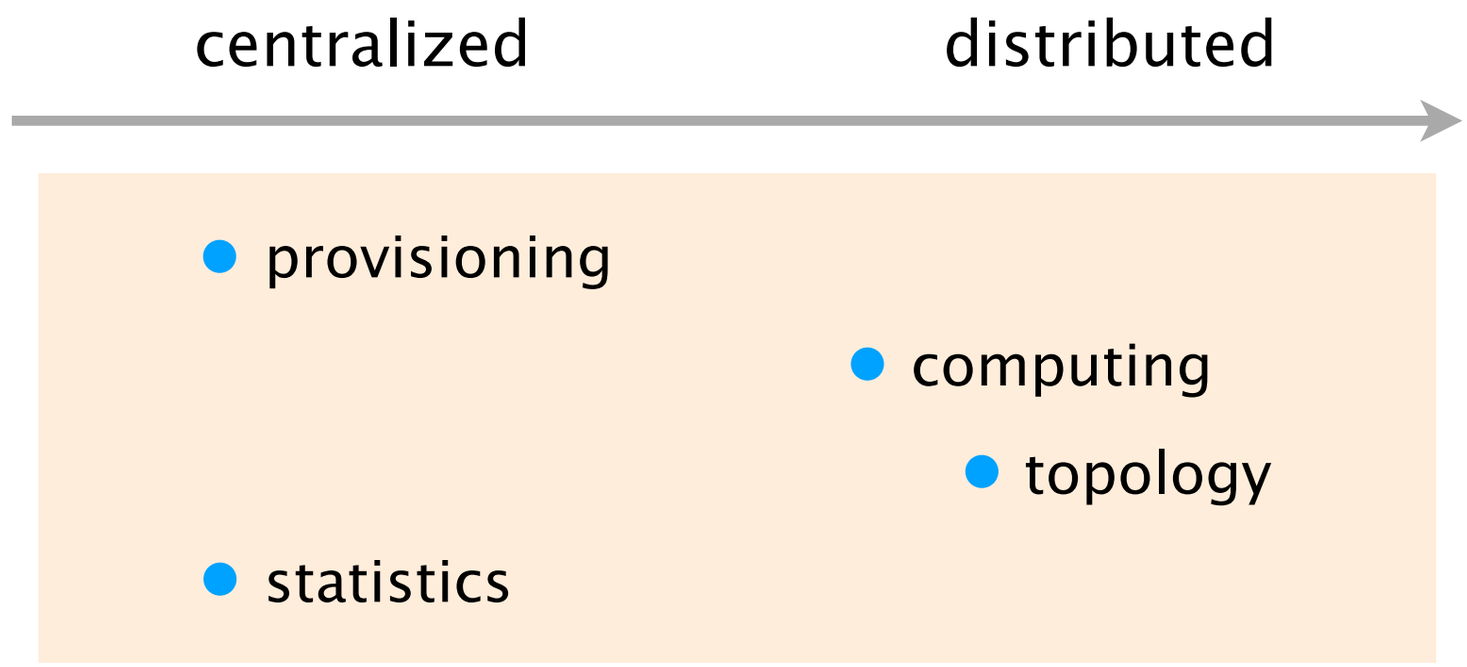
centralized

distributed

- provisioning
- computing
  - topology
  - statistics

"Traditional" CPs


centralized
distributed
provisioning
computing
topology
statistics
provisioning
computing
topology
statistics
Openflow CPs
"Traditional" CPs


centralized
distributed
provisioning
computing
topology
statistics

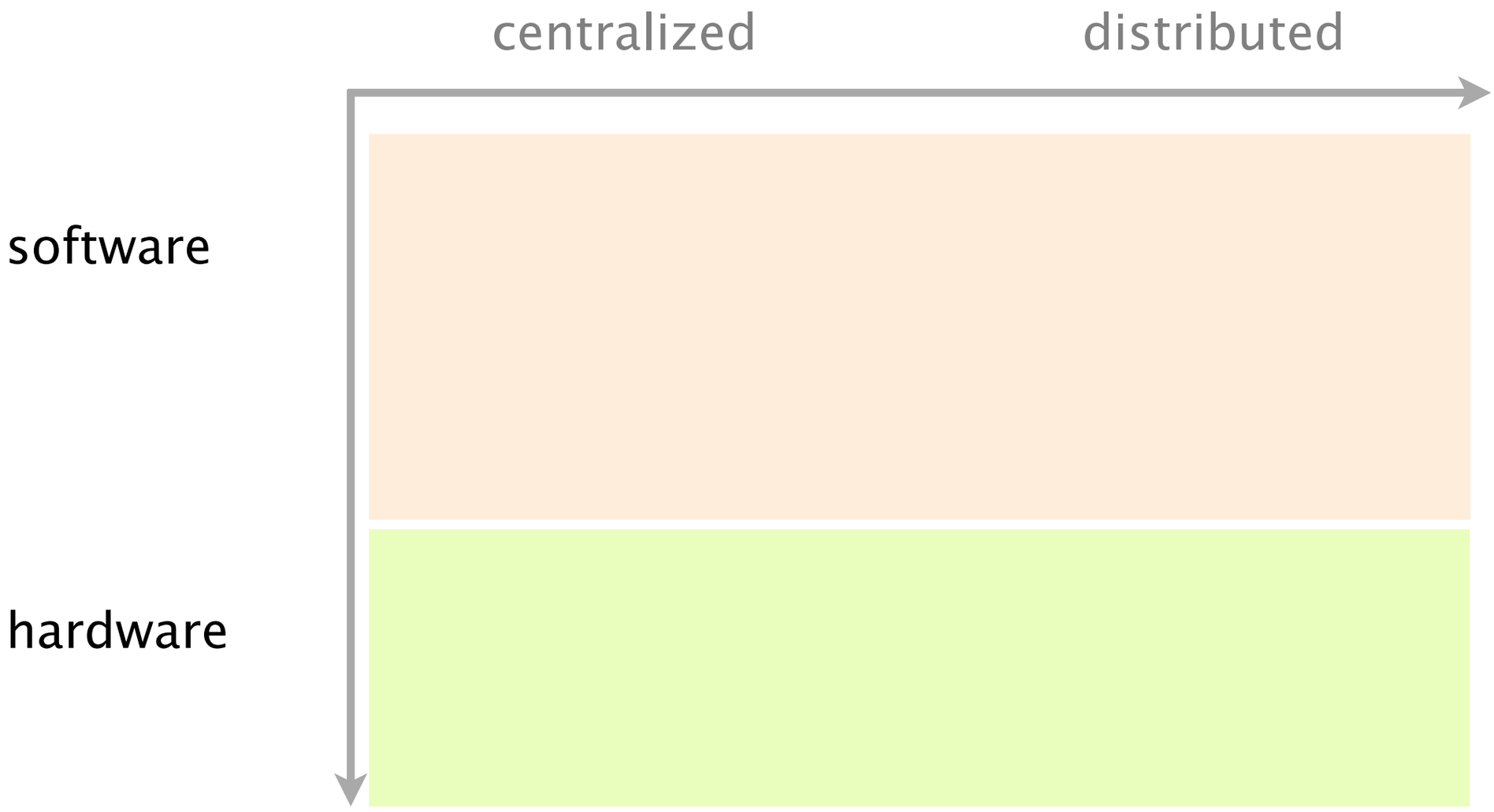
"Hybrid" CPs

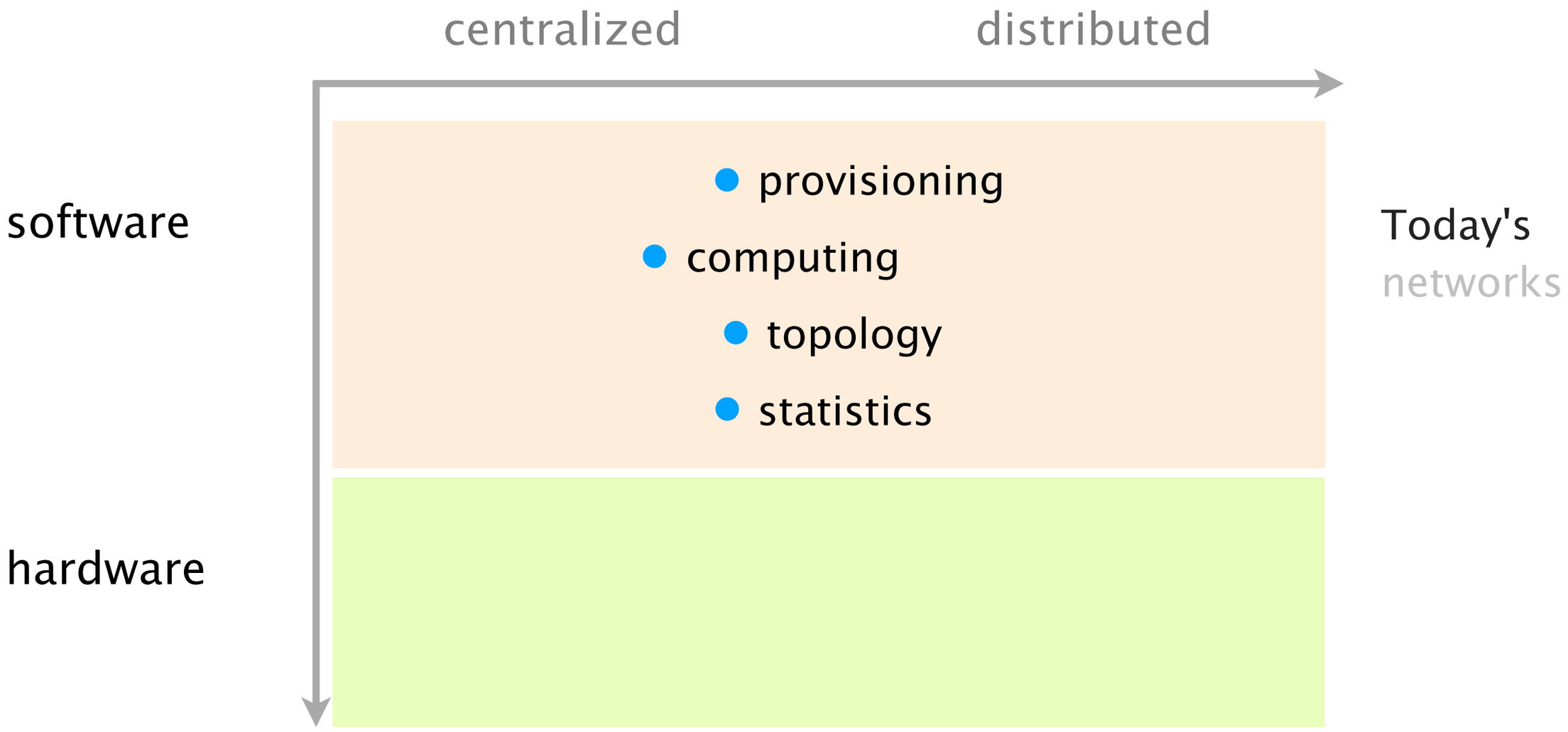
Network Control Planes

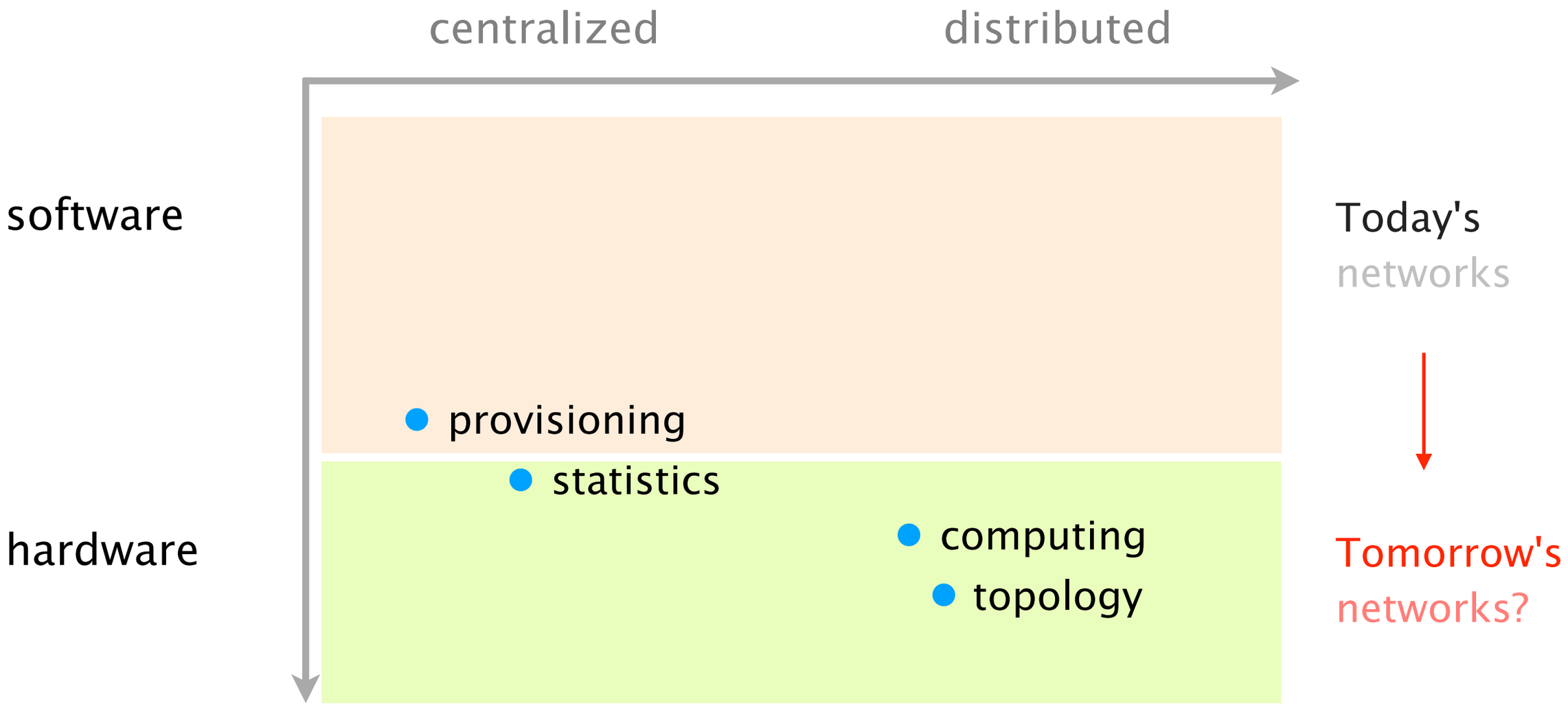
What? How? Where?


centralized
distributed


centralized
distributed
software
hardware


centralized
distributed
software
hardware


centralized
distributed
software
hardware
provisioning
computing
topology
statistics
Today's
networks

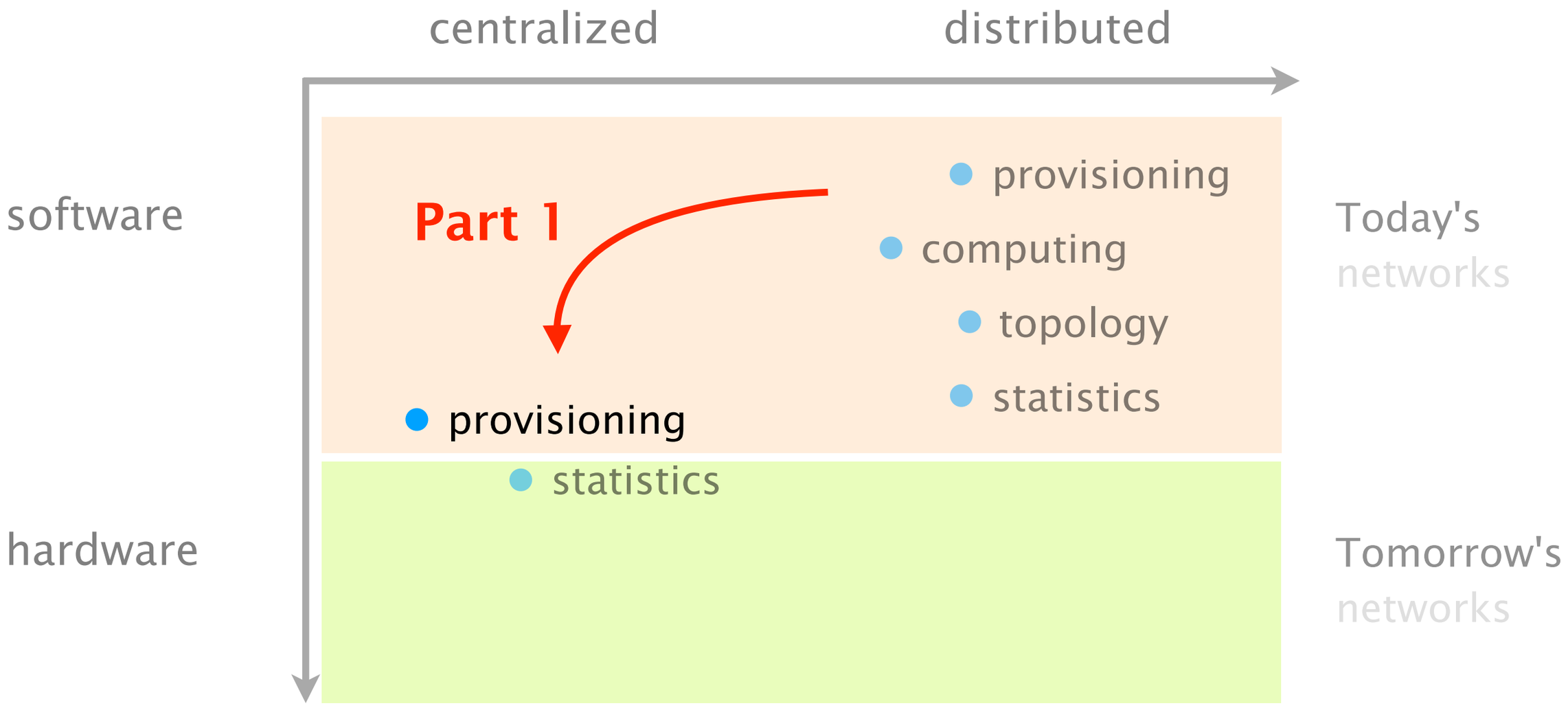

centralized
distributed
software
hardware
provisioning
statistics
computing
topology
Today's
networks
Tomorrow's
networks?

Network Control Planes

What? How? Where?

Network Control Planes

What? How? Where?



How can we centrally provision the forwarding state produced by distributed protocols?

How can we centrally provision the forwarding state produced by distributed protocols?

Fibbing [SIGCOMM'15]

vanbever_fibbing_sigcomm_2015 (1).pdf (page 1 of 14)

Central Control Over Distributed Routing

http://fibbing.net

Stefano Vissicchio*, Olivier Tilmans*, Laurent Vanbever†, Jennifer Rexford‡
* Université catholique de Louvain, † ETH Zurich, ‡ Princeton University
* name.surname@uclouvain.be, †lvanbever@ethz.ch, ‡jrex@cs.princeton.edu

ABSTRACT

Centralizing routing decisions offers tremendous flexibility, but sacrifices the robustness of distributed protocols. In this paper, we present *Fibbing*, an architecture that achieves both flexibility and robustness through central control over distributed routing. Fibbing introduces fake nodes and links into an underlying link-state routing protocol, so that routers compute their own forwarding tables based on the augmented topology. Fibbing is expressive, and readily supports flexible load balancing, traffic engineering, and backup routes. Based on high-level forwarding requirements, the Fibbing controller computes a compact augmented topology and injects the fake components through standard routing-protocol messages. Fibbing works with any unmodified commercial routers speaking OSPF. Our experiments also show that it can scale to large networks with many forwarding requirements, introduces minimal overhead, and quickly reacts to network and controller failures.

CCS Concepts

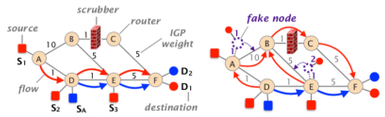
•**Networks** → **Routing protocols;** *Network architectures; Programmable networks; Network management;*

Keywords

Fibbing; SDN; link-state routing

1. INTRODUCTION

Consider a large IP network with hundreds of devices, including the components shown in Fig. 1a. A set of IP addresses (D_1) see a sudden surge of traffic, from multiple entry points (A, D, and E), that congests a part of the network. As a network operator, you suspect a denial-of-service attack (DoS), but cannot know for sure without inspecting the traffic as it could also be a flash crowd. Your goal is therefore to: (i) isolate the flows destined to these IP addresses, (ii) direct them to a scrubber connected between B and C, in order to "clean" them if needed, and (iii) reduce congestion by load-balancing the traffic on unused links, like (B, E).



(a) Initial topology (b) Augmented topology

Figure 1: Fibbing can steer the initial forwarding paths (see (a)) for D_1 through a scrubber by adding fake nodes and links (see (b)).

Performing this routine task is very difficult in traditional networks. First, since the middlebox and the destinations are not adjacent to each other, the configuration of multiple devices needs to change. Also, since intra-domain routing is typically based on shortest path algorithms, modifying the routing configuration is likely to impact many other flows not involved in the attack. In Fig. 1a, any attempt to reroute flows to D_1 would also reroute flows to D_2 since they home to the same router. Advertising D_1 from the middlebox would attract the right traffic, but would not necessarily alleviate the congestion, because *all* D_1 traffic would traverse (and congest) path (A, D, E, B), leaving (A, B) unused. Well-known Traffic-Engineering (TE) protocols (*e.g.*, MPLS RSVP-TE [1]) could help. Unfortunately, since D_1 traffic enters the network from multiple points, many tunnels (three, on A, D, and E, in our tiny example) would need to be configured and signaled. This increases both control-plane and data-plane overhead.

Software Defined Networking (SDN) could easily solve the problem as it enables centralized and direct control of the forwarding behavior. However, moving away from distributed routing protocols comes at a cost. In-

*S. Vissicchio is a postdoctoral researcher of F.R.S.-FNRS.

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom
 ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787497

NetComplete [NSDI'18]

vanbever_netcomplete_nsdi_2018 (1).pdf (page 1 of 16)

***NetComplete*: Practical Network-Wide Configuration Synthesis with Autocompletion**

Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, Martin Vechev
ETH Zürich

netcomplete.ethz.ch

Abstract

Network operators often need to adapt the configuration of a network in order to comply with changing routing policies. Evolving existing configurations, however, is a complex task as local changes can have unforeseen global effects. Not surprisingly, this often leads to mistakes that result in network downtimes.

We present NetComplete, a system that assists operators in modifying existing network-wide configurations to comply with new routing policies. NetComplete takes as input configurations with "holes" that identify the parameters to be completed and "autocompletes" these with concrete values. The use of a partial configuration addresses two important challenges inherent to existing synthesis solutions: *(i)* it allows the operators to precisely control how configurations should be changed; and *(ii)* it allows the synthesizer to leverage the existing configurations to gain performance. To scale, NetComplete relies on powerful techniques such as counter-example guided inductive synthesis (for link-state protocols) and partial evaluation (for path-vector protocols).

We implemented NetComplete and showed that it can autocomplete configurations using static routes, OSPF, and BGP. Our implementation also scales to realistic networks and complex routing policies. Among others, it is able to synthesize configurations for networks with up to 200 routers within few minutes.

1 Introduction

In a world where more and more critical services converge on IP, even slight network downtimes can cause large financial or reputational losses. This strategic importance contrasts with the fact that managing a network is surprisingly hard and brittle. Out of high-level requirements, network operators have to come up (often manually) with low-level configurations specifying the behavior of hundreds of devices running complex distributed protocols. A single misconfiguration can bring down the network infrastructure, or worse, a piece of the Internet in case of BGP-related misconfigurations. Every few months downtimes involving major players such as NYSE [1], Google [2], Facebook [3], or United Airlines [4] make the news. Actually, studies show that human-induced misconfigurations, *not* physical failures, explain the majority of downtimes [5].

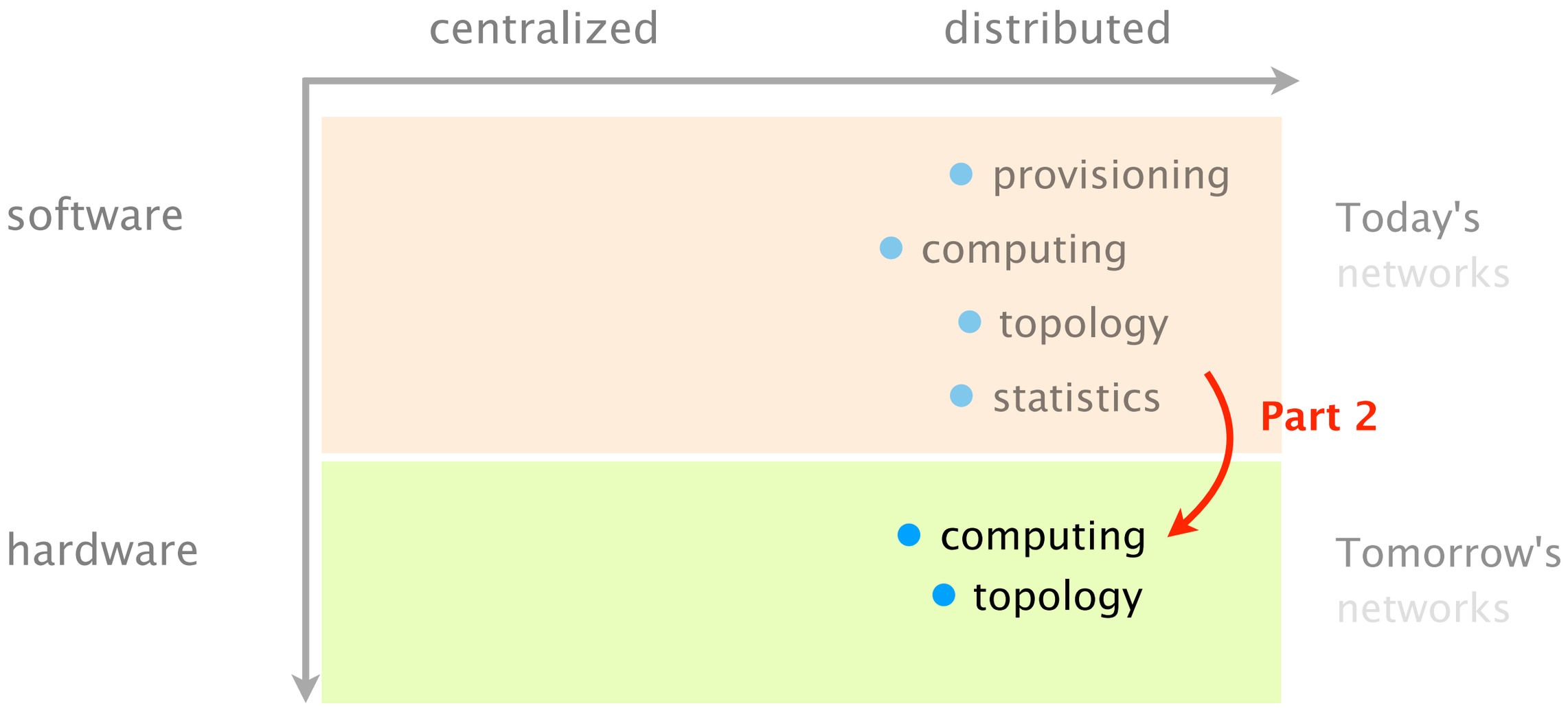
To address these challenges, recently there has been an increased interest in configuration verification [6, 7, 8, 9, 10, 11, 12, 13] and synthesis [14, 15, 16, 17, 18, 19, 20]. Configuration synthesis in particular promises to alleviate most of the operator's burdens by deriving *correct* configurations out of high-level objectives.

Challenges in network synthesis While promising, network operators can still be reluctant to use existing synthesis systems for at least three reasons: *(i) interpretability*: the synthesizer can produce configurations that differ wildly from manually provided ones, making it hard to understand what the resulting configuration does. Moreover, small policy changes can cause the synthesized configuration (or configuration templates in the case of PropaneAT [16]) to change radically; *(ii) protocol coverage*: existing systems [15, 16] are restricted to producing BGP-only configurations, while most networks rely on multiple routing protocols (e.g., to leverage OSPF's fast-convergence capabilities); and *(iii) scalability*: recent synthesizers such as SyNET [20] handle multiple protocols but do not scale to realistic networks.

NetComplete We present a system, NetComplete, which addresses the above challenges with partial synthesis. Rather than synthesizing a new configuration from scratch, NetComplete allows network operators to express their intent by sketching parts of the existing configuration that should remain intact (capturing a high-level insight) and "holes" represented with symbolic values which the synthesizer should instantiate (e.g., OSPF weights, BGP import/export policies)

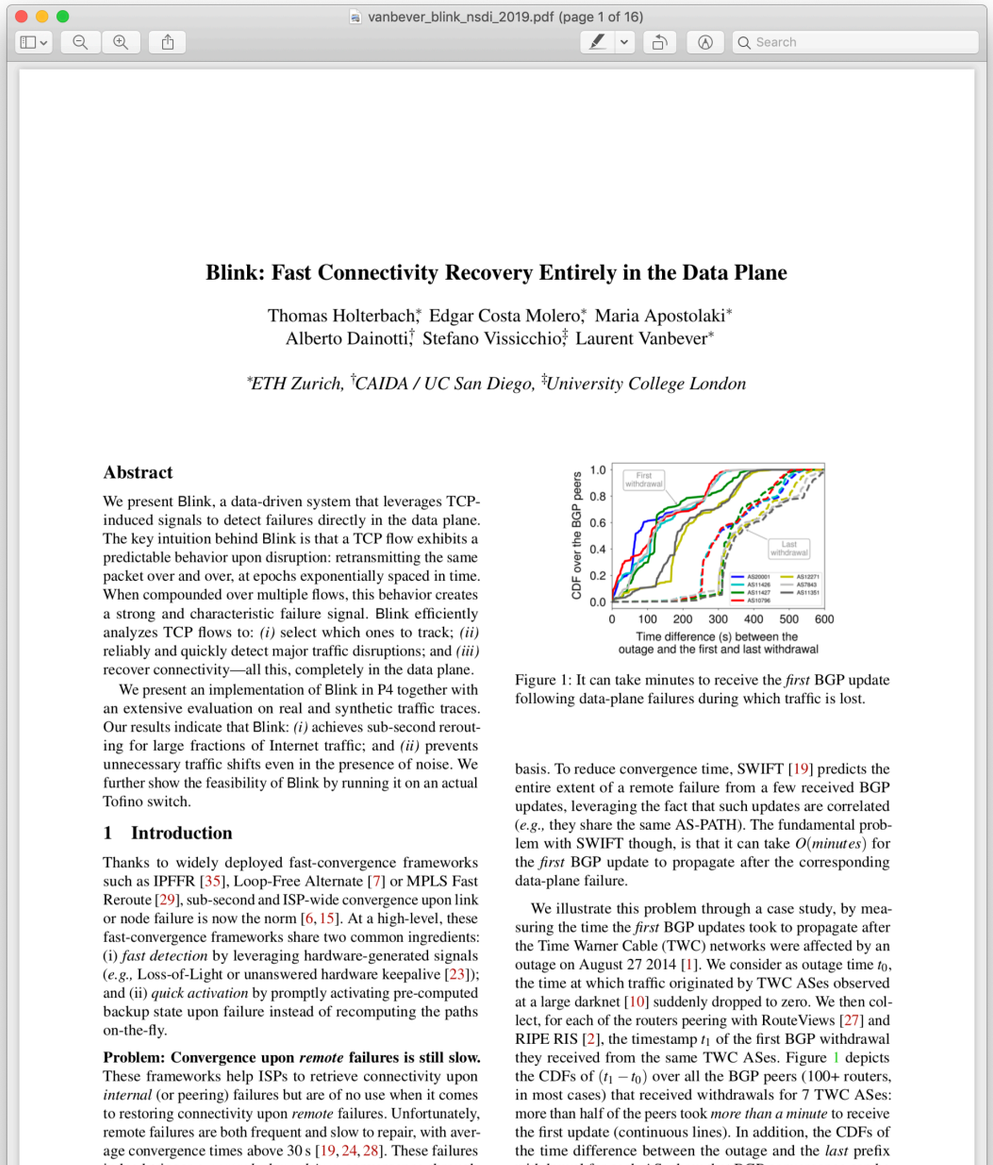
Network Control Planes

What? How? Where?



What parts of the CP should we offload (if any) and how?

Blink [NSDI'19]



vanbever_blink_nsdi_2019.pdf (page 1 of 16)

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach,* Edgar Costa Molero,* Maria Apostolaki*
Alberto Dainotti,† Stefano Vissicchio,‡ Laurent Vanbever*

*ETH Zurich, †CAIDA / UC San Diego, ‡University College London

Abstract

We present Blink, a data-driven system that leverages TCP-induced signals to detect failures directly in the data plane. The key intuition behind Blink is that a TCP flow exhibits a predictable behavior upon disruption: retransmitting the same packet over and over, at epochs exponentially spaced in time. When compounded over multiple flows, this behavior creates a strong and characteristic failure signal. Blink efficiently analyzes TCP flows to: *(i)* select which ones to track; *(ii)* reliably and quickly detect major traffic disruptions; and *(iii)* recover connectivity—all this, completely in the data plane.

We present an implementation of Blink in P4 together with an extensive evaluation on real and synthetic traffic traces. Our results indicate that Blink: *(i)* achieves sub-second rerouting for large fractions of Internet traffic; and *(ii)* prevents unnecessary traffic shifts even in the presence of noise. We further show the feasibility of Blink by running it on an actual Tofino switch.

1 Introduction

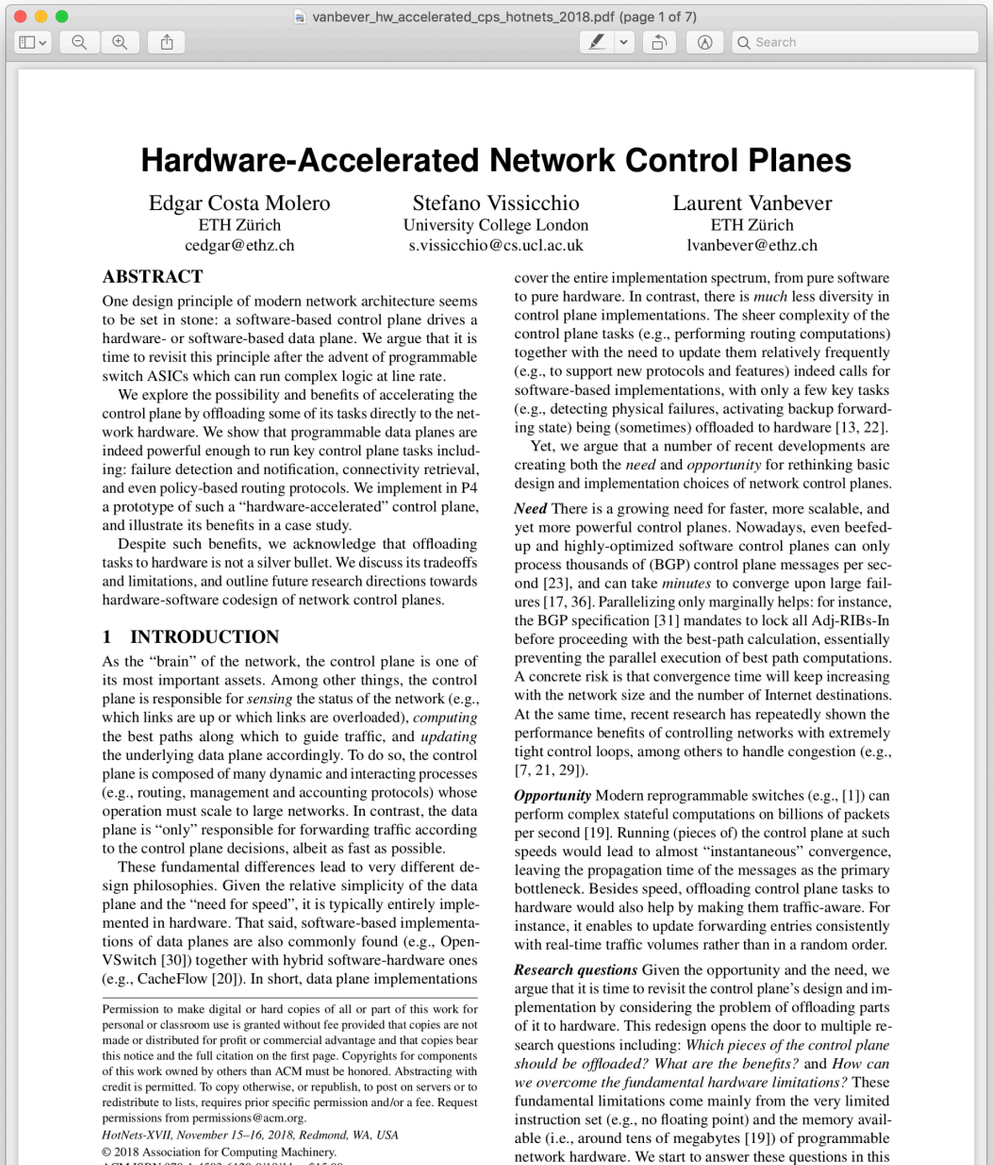
Thanks to widely deployed fast-convergence frameworks such as IPFFR [35], Loop-Free Alternate [7] or MPLS Fast Reroute [29], sub-second and ISP-wide convergence upon link or node failure is now the norm [6, 15]. At a high-level, these fast-convergence frameworks share two common ingredients: (i) *fast detection* by leveraging hardware-generated signals (*e.g.,* Loss-of-Light or unanswered hardware keepalive [23]); and (ii) *quick activation* by promptly activating pre-computed backup state upon failure instead of recomputing the paths on-the-fly.

Problem: Convergence upon *remote* failures is still slow. These frameworks help ISPs to retrieve connectivity upon *internal* (or peering) failures but are of no use when it comes to restoring connectivity upon *remote* failures. Unfortunately, remote failures are both frequent and slow to repair, with average convergence times above 30 s [19, 24, 28]. These failures indeed trigger a *control-plane-driven* convergence through the propagation of BGP updates on a per-router and per-prefix basis. To reduce convergence time, SWIFT [19] predicts the entire extent of a remote failure from a few received BGP updates, leveraging the fact that such updates are correlated (*e.g.,* they share the same AS-PATH). The fundamental problem with SWIFT though, is that it can take $O(minutes)$ for the *first* BGP update to propagate after the corresponding data-plane failure.

Figure 1: It can take minutes to receive the *first* BGP update following data-plane failures during which traffic is lost.

We illustrate this problem through a case study, by measuring the time the *first* BGP updates took to propagate after the Time Warner Cable (TWC) networks were affected by an outage on August 27 2014 [1]. We consider as outage time t_0, the time at which traffic originated by TWC ASes observed at a large darknet [10] suddenly dropped to zero. We then collect, for each of the routers peering with RouteViews [27] and RIPE RIS [2], the timestamp t_1 of the first BGP withdrawal they received from the same TWC ASes. Figure 1 depicts the CDFs of $(t_1 - t_0)$ over all the BGP peers (100+ routers, in most cases) that received withdrawals for 7 TWC ASes: more than half of the peers took *more than a minute* to receive the first update (continuous lines). In addition, the CDFs of the time difference between the outage and the *last* prefix withdrawal for each AS, show that BGP convergence can be as slow as several minutes (dashed lines).

HW-accelerated CPs [HotNets'18]



vanbever_hw_accelerated_cps_hotnets_2018.pdf (page 1 of 7)

Hardware-Accelerated Network Control Planes

Edgar Costa Molero
ETH Zürich
cedgar@ethz.ch

Stefano Vissicchio
University College London
s.vissicchio@cs.ucl.ac.uk

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

ABSTRACT

One design principle of modern network architecture seems to be set in stone: a software-based control plane drives a hardware- or software-based data plane. We argue that it is time to revisit this principle after the advent of programmable switch ASICs which can run complex logic at line rate.

We explore the possibility and benefits of accelerating the control plane by offloading some of its tasks directly to the network hardware. We show that programmable data planes are indeed powerful enough to run key control plane tasks including: failure detection and notification, connectivity retrieval, and even policy-based routing protocols. We implement in P4 a prototype of such a "hardware-accelerated" control plane, and illustrate its benefits in a case study.

Despite such benefits, we acknowledge that offloading tasks to hardware is not a silver bullet. We discuss its tradeoffs and limitations, and outline future research directions towards hardware-software codesign of network control planes.

1 INTRODUCTION

As the "brain" of the network, the control plane is one of its most important assets. Among other things, the control plane is responsible for *sensing* the status of the network (e.g., which links are up or which links are overloaded), *computing* the best paths along which to guide traffic, and *updating* the underlying data plane accordingly. To do so, the control plane is composed of many dynamic and interacting processes (e.g., routing, management and accounting protocols) whose operation must scale to large networks. In contrast, the data plane is "only" responsible for forwarding traffic according to the control plane decisions, albeit as fast as possible.

These fundamental differences lead to very different design philosophies. Given the relative simplicity of the data plane and the "need for speed", it is typically entirely implemented in hardware. That said, software-based implementations of data planes are also commonly found (e.g., OpenVSwitch [30]) together with hybrid software-hardware ones (e.g., CacheFlow [20]). In short, data plane implementations cover the entire implementation spectrum, from pure software to pure hardware. In contrast, there is *much* less diversity in control plane implementations. The sheer complexity of the control plane tasks (e.g., performing routing computations) together with the need to update them relatively frequently (e.g., to support new protocols and features) indeed calls for software-based implementations, with only a few key tasks (e.g., detecting physical failures, activating backup forwarding state) being (sometimes) offloaded to hardware [13, 22].

Yet, we argue that a number of recent developments are creating both the *need* and *opportunity* for rethinking basic design and implementation choices of network control planes.

Need There is a growing need for faster, more scalable, and yet more powerful control planes. Nowadays, even beefed-up and highly-optimized software control planes can only process thousands of (BGP) control plane messages per second [23], and can take *minutes* to converge upon large failures [17, 36]. Parallelizing only marginally helps: for instance, the BGP specification [31] mandates to lock all Adj-RIBs-In before proceeding with the best-path calculation, essentially preventing the parallel execution of best path computations. A concrete risk is that convergence time will keep increasing with the network size and the number of Internet destinations. At the same time, recent research has repeatedly shown the performance benefits of controlling networks with extremely tight control loops, among others to handle congestion (e.g., [7, 21, 29]).

Opportunity Modern reprogrammable switches (e.g., [1]) can perform complex stateful computations on billions of packets per second [19]. Running (pieces of) the control plane at such speeds would lead to almost "instantaneous" convergence, leaving the propagation time of the messages as the primary bottleneck. Besides speed, offloading control plane tasks to hardware would also help by making them traffic-aware. For instance, it enables to update forwarding entries consistently with real-time traffic volumes rather than in a random order.

Research questions Given the opportunity and the need, we argue that it is time to revisit the control plane's design and implementation by considering the problem of offloading parts of it to hardware. This redesign opens the door to multiple research questions including: *Which pieces of the control plane should be offloaded? What are the benefits?* and *How can we overcome the fundamental hardware limitations?* These fundamental limitations come mainly from the very limited instruction set (e.g., no floating point) and the memory available (i.e., around tens of megabytes [19]) of programmable network hardware. We start to answer these questions in this paper and make two contributions.

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA

ACM ISBN 978-1-4503-6120-0/18/11…$15.00
https://doi.org/10.1145/3286062.3286080

How can we centrally provision the forwarding state produced by distributed protocols?

Fibbing [SIGCOMM'15]

vanbever_fibbing_sigcomm_2015 (1).pdf (page 1 of 14)

Central Control Over Distributed Routing

http://fibbing.net

Stefano Vissicchio*, Olivier Tilmans*, Laurent Vanbever†, Jennifer Rexford‡
* Université catholique de Louvain, † ETH Zurich, ‡ Princeton University
* name.surname@uclouvain.be, †lvanbever@ethz.ch, ‡jrex@cs.princeton.edu

ABSTRACT

Centralizing routing decisions offers tremendous flexibility, but sacrifices the robustness of distributed protocols. In this paper, we present *Fibbing*, an architecture that achieves both flexibility and robustness through central control over distributed routing. Fibbing introduces fake nodes and links into an underlying link-state routing protocol, so that routers compute their own forwarding tables based on the augmented topology. Fibbing is expressive, and readily supports flexible load balancing, traffic engineering, and backup routes. Based on high-level forwarding requirements, the Fibbing controller computes a compact augmented topology and injects the fake components through standard routing-protocol messages. Fibbing works with any unmodified commercial routers speaking OSPF. Our experiments also show that it can scale to large networks with many forwarding requirements, introduces minimal overhead, and quickly reacts to network and controller failures.

CCS Concepts

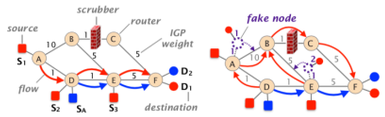
•**Networks** → **Routing protocols;** *Network architectures; Programmable networks; Network management;*

Keywords

Fibbing; SDN; link-state routing

1. INTRODUCTION

Consider a large IP network with hundreds of devices, including the components shown in Fig. 1a. A set of IP addresses (D_1) see a sudden surge of traffic, from multiple entry points (A, D, and E), that congests a part of the network. As a network operator, you suspect a denial-of-service attack (DoS), but cannot know for sure without inspecting the traffic as it could also be a flash crowd. Your goal is therefore to: (i) isolate the flows destined to these IP addresses, (ii) direct them to a scrubber connected between B and C, in order to "clean" them if needed, and (iii) reduce congestion by load-balancing the traffic on unused links, like (B, E).



(a) Initial topology (b) Augmented topology

Figure 1: Fibbing can steer the initial forwarding paths (see (a)) for D_1 through a scrubber by adding fake nodes and links (see (b)).

Performing this routine task is very difficult in traditional networks. First, since the middlebox and the destinations are not adjacent to each other, the configuration of multiple devices needs to change. Also, since intra-domain routing is typically based on shortest path algorithms, modifying the routing configuration is likely to impact many other flows not involved in the attack. In Fig. 1a, any attempt to reroute flows to D_1 would also reroute flows to D_2 since they home to the same router. Advertising D_1 from the middlebox would attract the right traffic, but would not necessarily alleviate the congestion, because *all* D_1 traffic would traverse (and congest) path (A, D, E, B), leaving (A, B) unused. Well-known Traffic-Engineering (TE) protocols (*e.g.*, MPLS RSVP-TE [1]) could help. Unfortunately, since D_1 traffic enters the network from multiple points, many tunnels (three, on A, D, and E, in our tiny example) would need to be configured and signaled. This increases both control-plane and data-plane overhead.

Software Defined Networking (SDN) could easily solve the problem as it enables centralized and direct control of the forwarding behavior. However, moving away from distributed routing protocols comes at a cost. In-

*S. Vissicchio is a postdoctoral researcher of F.R.S.-FNRS.

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom
 ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787497

NetComplete [NSDI'18]

vanbever_netcomplete_nsdi_2018 (1).pdf (page 1 of 16)

***NetComplete*: Practical Network-Wide Configuration Synthesis with Autocompletion**

Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, Martin Vechev
ETH Zürich

netcomplete.ethz.ch

Abstract

Network operators often need to adapt the configuration of a network in order to comply with changing routing policies. Evolving existing configurations, however, is a complex task as local changes can have unforeseen global effects. Not surprisingly, this often leads to mistakes that result in network downtimes.

We present NetComplete, a system that assists operators in modifying existing network-wide configurations to comply with new routing policies. NetComplete takes as input configurations with "holes" that identify the parameters to be completed and "autocompletes" these with concrete values. The use of a partial configuration addresses two important challenges inherent to existing synthesis solutions: *(i)* it allows the operators to precisely control how configurations should be changed; and *(ii)* it allows the synthesizer to leverage the existing configurations to gain performance. To scale, NetComplete relies on powerful techniques such as counter-example guided inductive synthesis (for link-state protocols) and partial evaluation (for path-vector protocols).

We implemented NetComplete and showed that it can autocomplete configurations using static routes, OSPF, and BGP. Our implementation also scales to realistic networks and complex routing policies. Among others, it is able to synthesize configurations for networks with up to 200 routers within few minutes.

1 Introduction

In a world where more and more critical services converge on IP, even slight network downtimes can cause large financial or reputational losses. This strategic importance contrasts with the fact that managing a network is surprisingly hard and brittle. Out of high-level requirements, network operators have to come up (often manually) with low-level configurations specifying the behavior of hundreds of devices running complex distributed protocols. A single misconfiguration can bring down the network infrastructure, or worse, a piece of the Internet in case of BGP-related misconfigurations. Every few months downtimes involving major players such as NYSE [1], Google [2], Facebook [3], or United Airlines [4] make the news. Actually, studies show that human-induced misconfigurations, *not* physical failures, explain the majority of downtimes [5].

To address these challenges, recently there has been an increased interest in configuration verification [6, 7, 8, 9, 10, 11, 12, 13] and synthesis [14, 15, 16, 17, 18, 19, 20]. Configuration synthesis in particular promises to alleviate most of the operator's burdens by deriving *correct* configurations out of high-level objectives.

Challenges in network synthesis While promising, network operators can still be reluctant to use existing synthesis systems for at least three reasons: *(i) interpretability*: the synthesizer can produce configurations that differ wildly from manually provided ones, making it hard to understand what the resulting configuration does. Moreover, small policy changes can cause the synthesized configuration (or configuration templates in the case of PropaneAT [16]) to change radically; *(ii) protocol coverage*: existing systems [15, 16] are restricted to producing BGP-only configurations, while most networks rely on multiple routing protocols (e.g., to leverage OSPF's fast-convergence capabilities); and *(iii) scalability*: recent synthesizers such as SyNET [20] handle multiple protocols but do not scale to realistic networks.

NetComplete We present a system, NetComplete, which addresses the above challenges with partial synthesis. Rather than synthesizing a new configuration from scratch, NetComplete allows network operators to express their intent by sketching parts of the existing configuration that should remain intact (capturing a high-level insight) and "holes" represented with symbolic values which the synthesizer should instantiate (e.g., OSPF weights, BGP import/export policies)

Goal

Centrally control distributed routing protocols

where the computation of the forwarding state is distributed

Goal

Centrally control distributed routing protocols

where the computation of the forwarding state is distributed

Why?

Designing central, scalable *and* robust control is hard

must ensure always-on connectivity to the controller

Distributed protocols are *still* ruling over networks

the vast majority of the networks rely on OSPF, BGP, MPLS, …

How can we control the network-wide forwarding state produced by distributed protocols?

How can we control the network-wide forwarding state produced by distributed protocols?

What are our knobs?

The network-wide forwarding state depends on three parameters

Network-wide
Forwarding state

Topology
(fixed)

+

Network-wide
Configuration

+

Network
Environment

→

Network-wide
Forwarding state

Topology
(fixed)

+

Network-wide
Configuration

+

Network
Environment

→

Network-wide
Forwarding state

links & nodes status (given)
routing announcements

Out of these three parameters,

two can be controlled

~~Topology~~
(fixed)

+

Network-wide
Configuration

+

Network
Environment

→

Network-wide
Forwarding state

~~links & nodes status~~

routing announcements

Given a forwarding state we want to program,

we therefore have two ways to provision it

Given a network-wide forwarding state
to provision, one can synthesize

way 1 the routing messages shown to the routers

way 2 the configurations run by the routers

Given a network-wide forwarding state

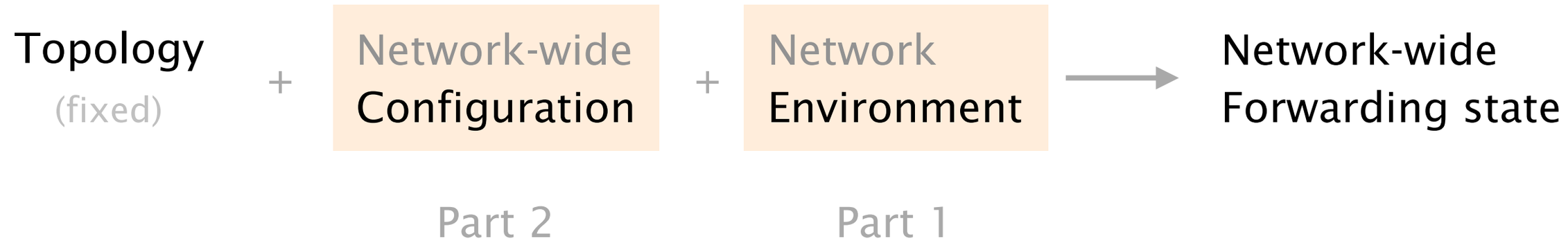
output to provision, one can synthesize

inputs the routing messages shown to the routers

functions the configurations run by the routers

Controlling distributed computation

through synthesis

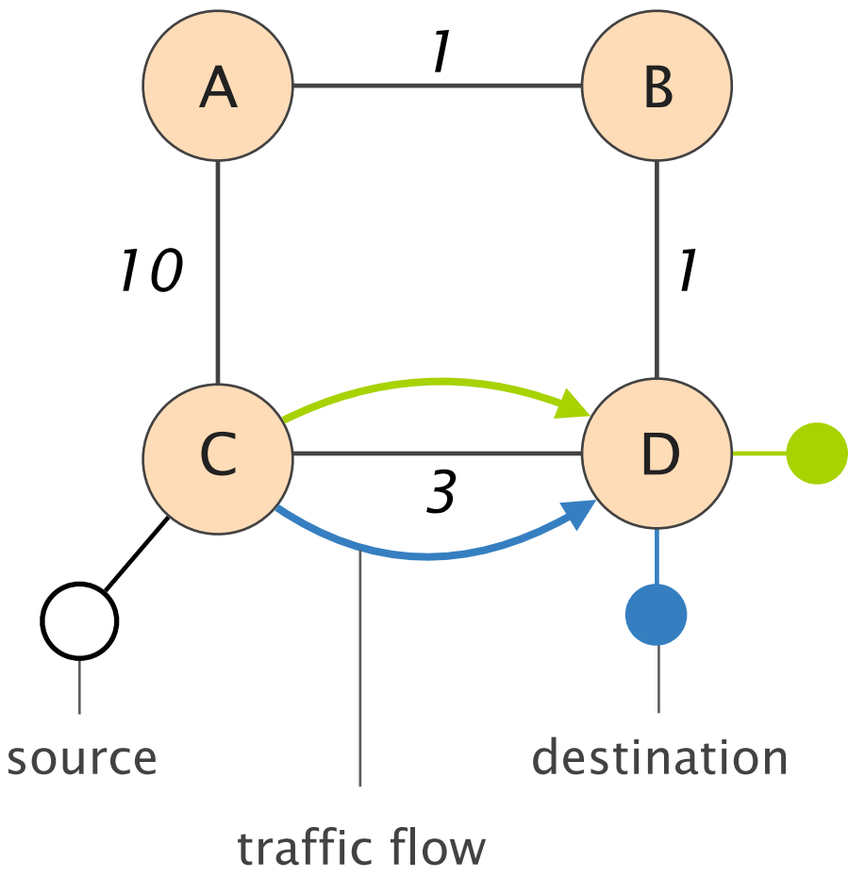


Controlling distributed computation

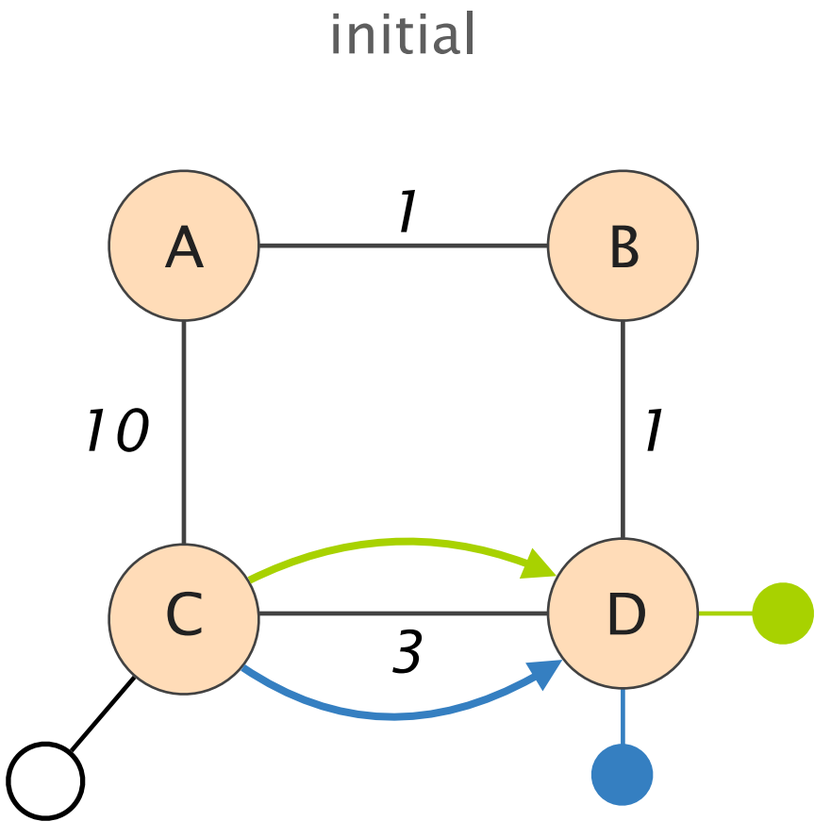
through synthesis

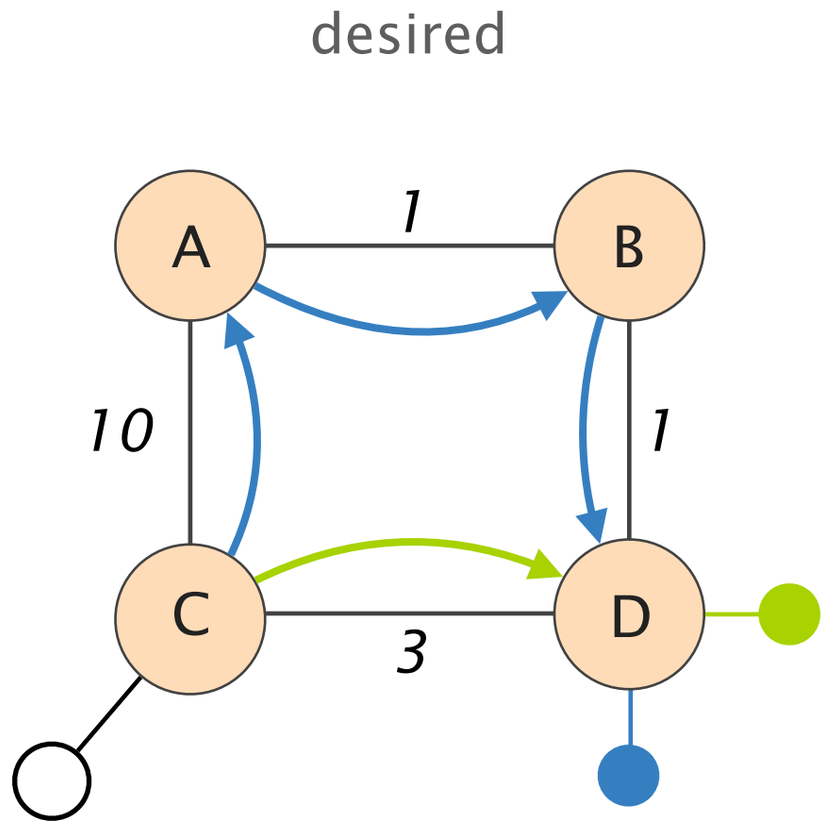
Part 1

Consider this network where a source sends traffic to 2 destinations



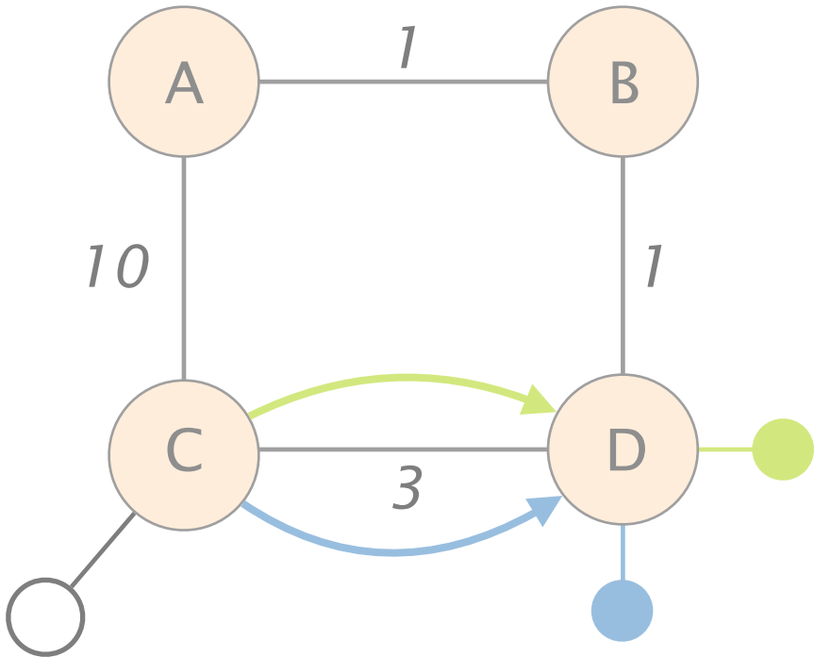
As congestion appears, the operator wants to shift away one flow from (C,D)



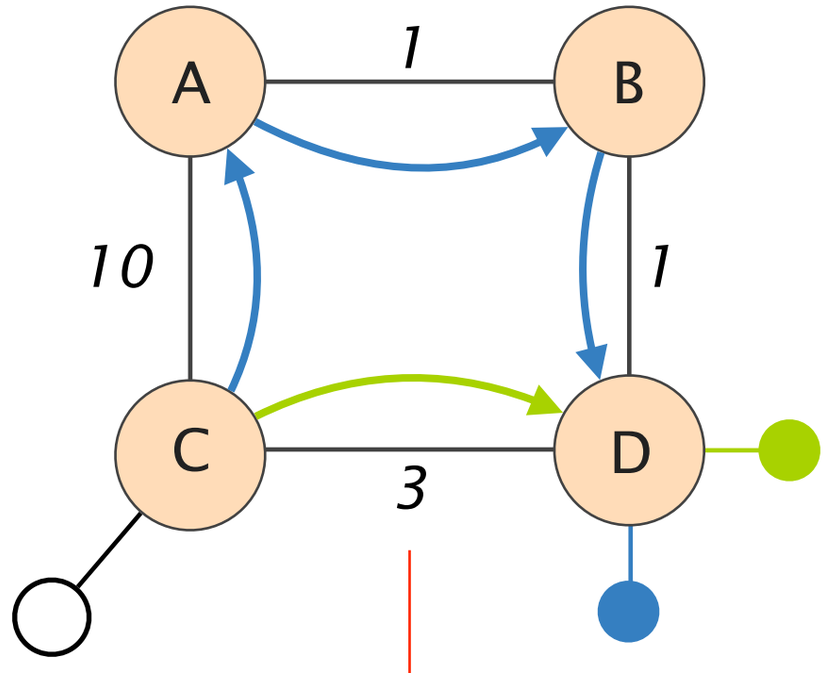


Moving only one flow is impossible though
as both destinations are connected to D

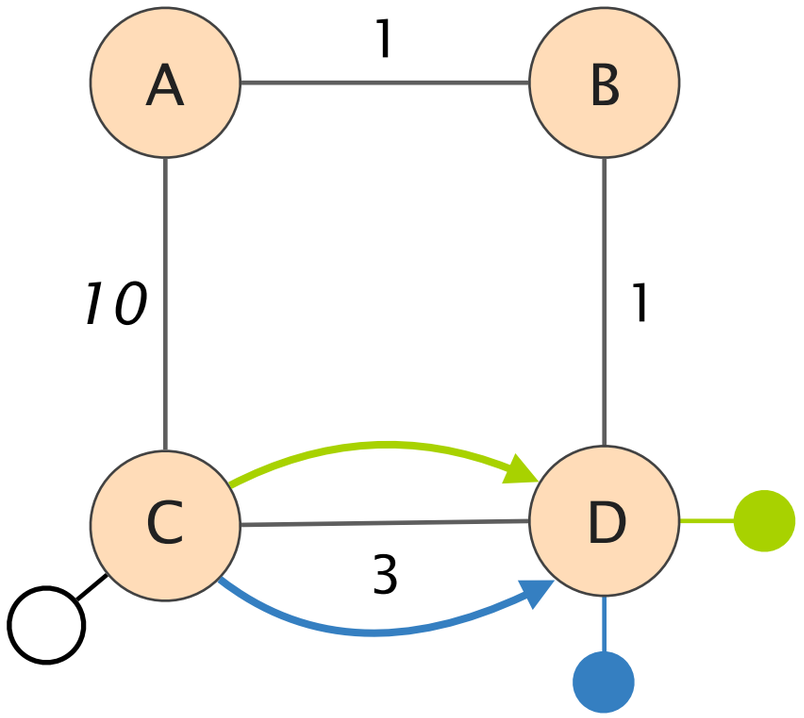
initial



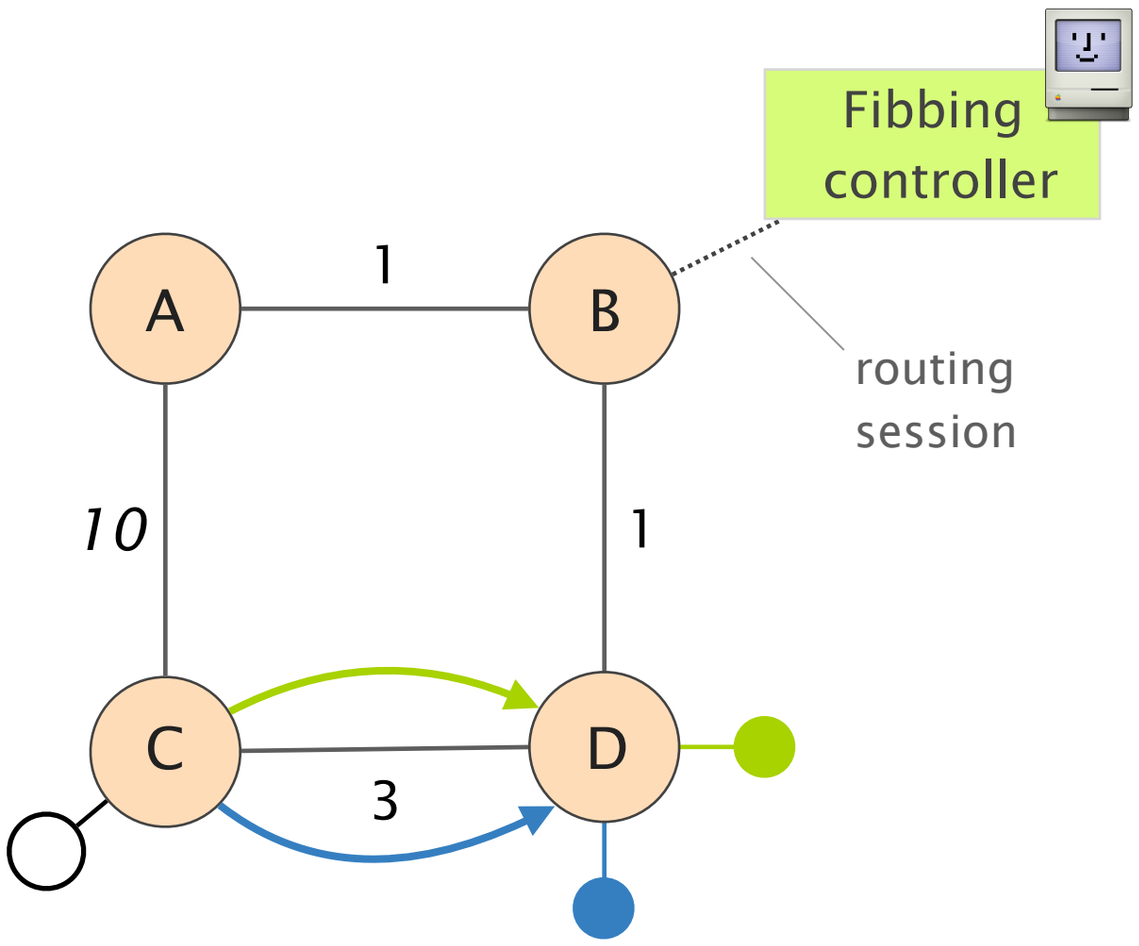
desired



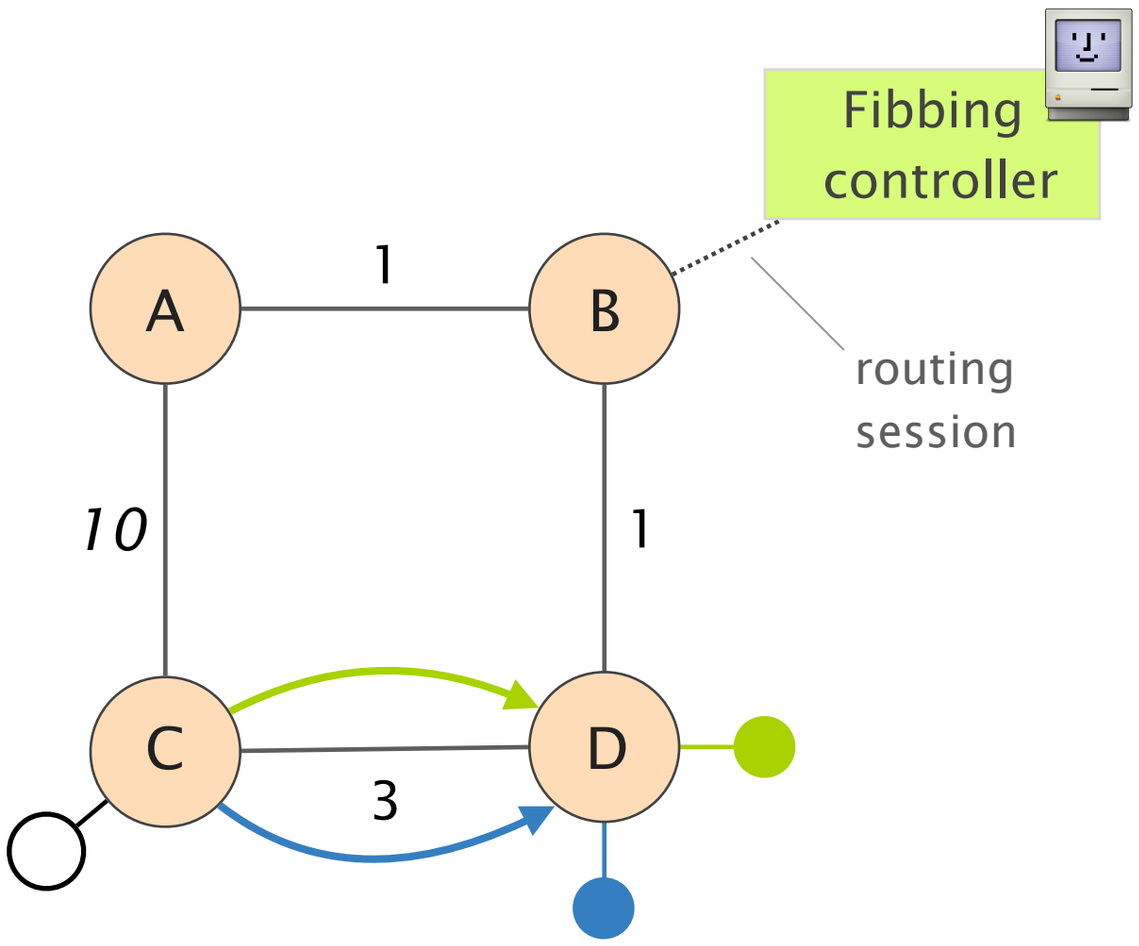
impossible to achieve by
reweighing the links

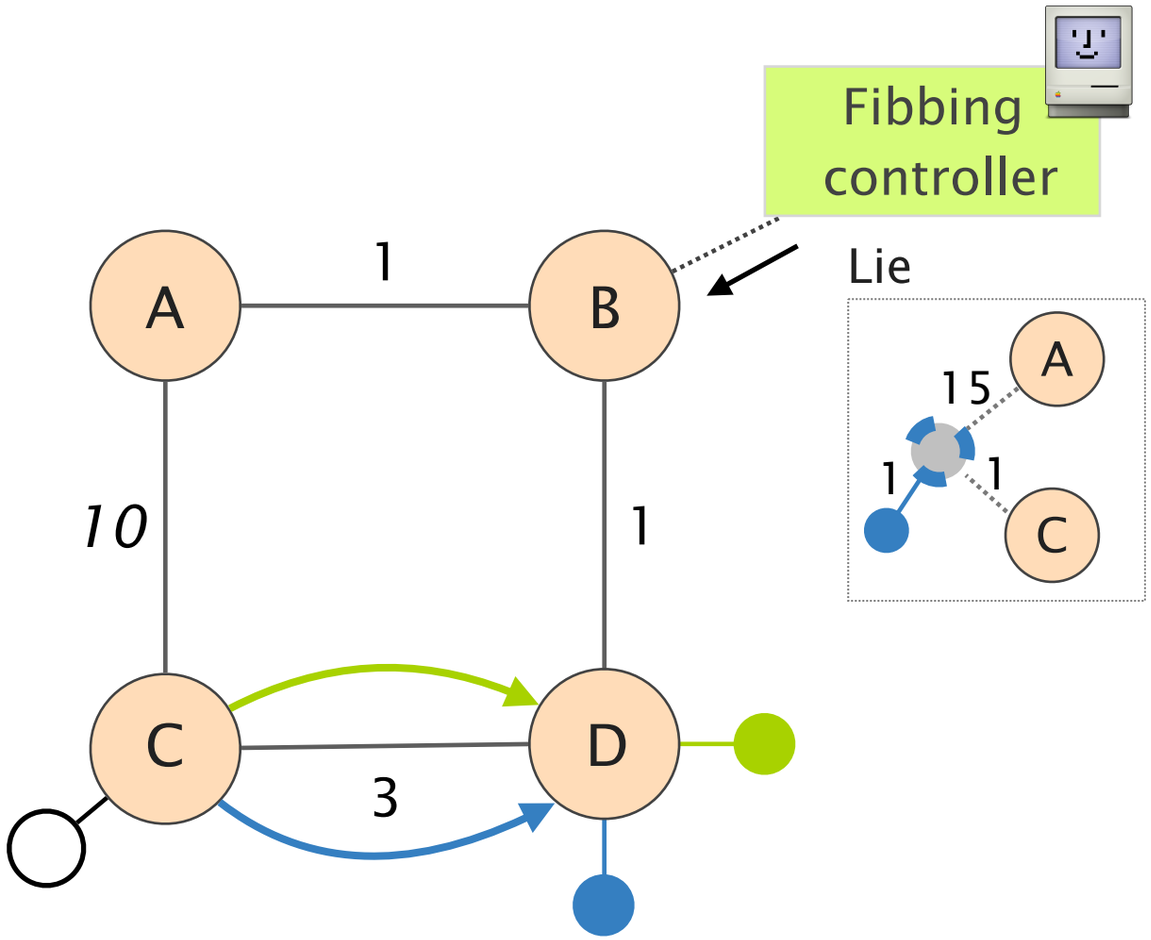

A
1
B
10
1
C
D
3

Let’s lie to the routers



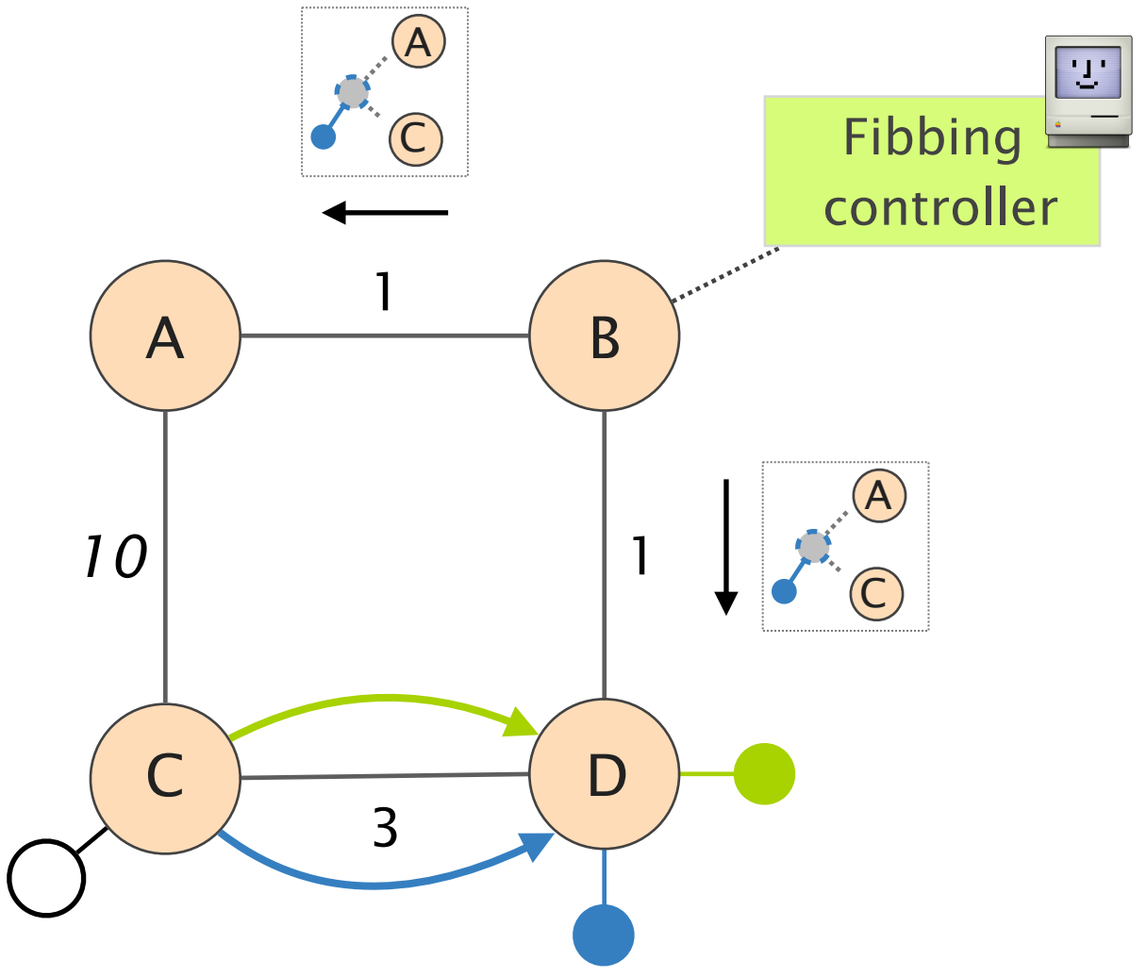
Let's lie to the routers, by injecting fake nodes, links and destinations



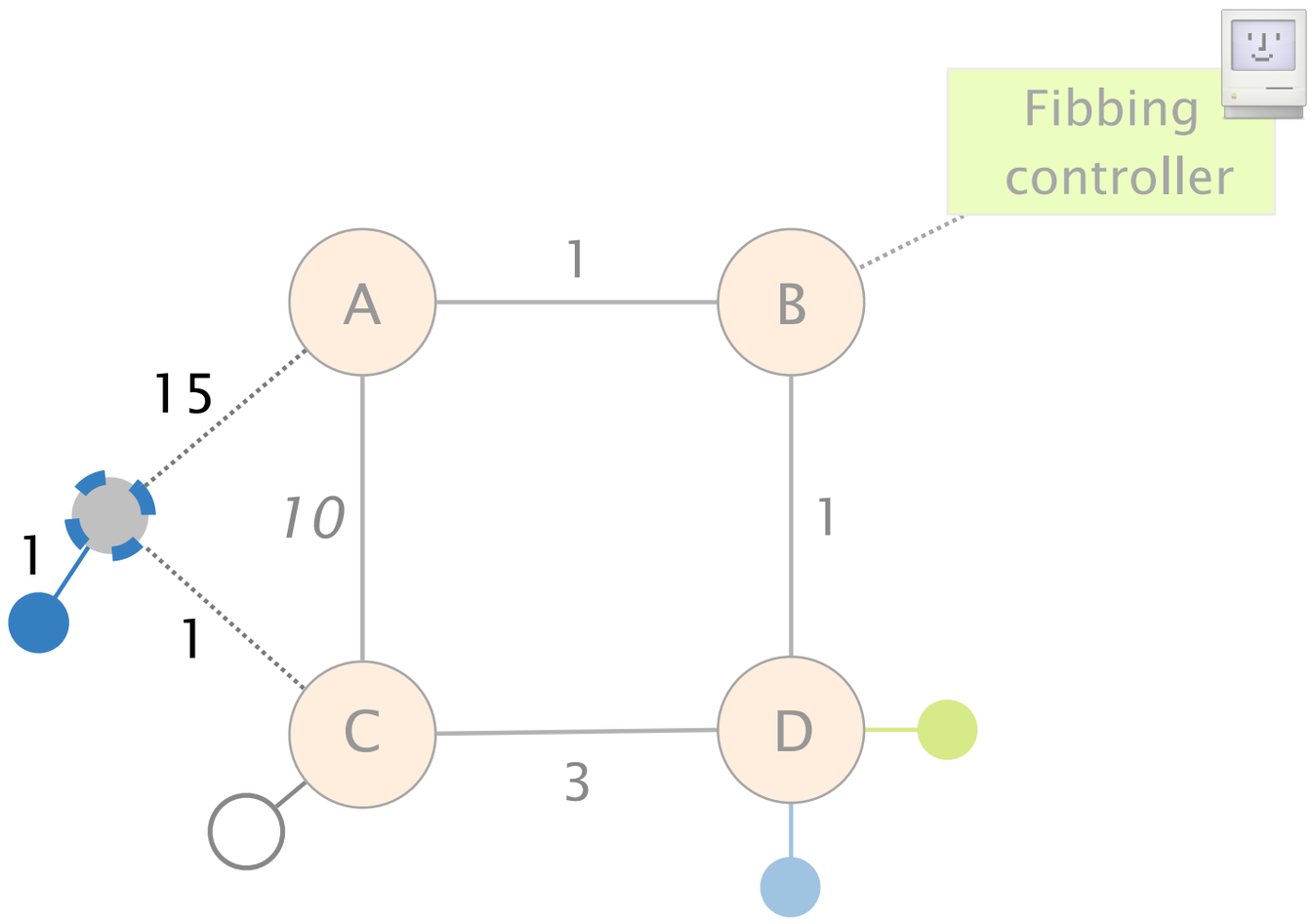

Fibbing controller
Lie
A
B
C
D
1
10
1
3
A
15
1
1
C

Lies are propagated network-wide

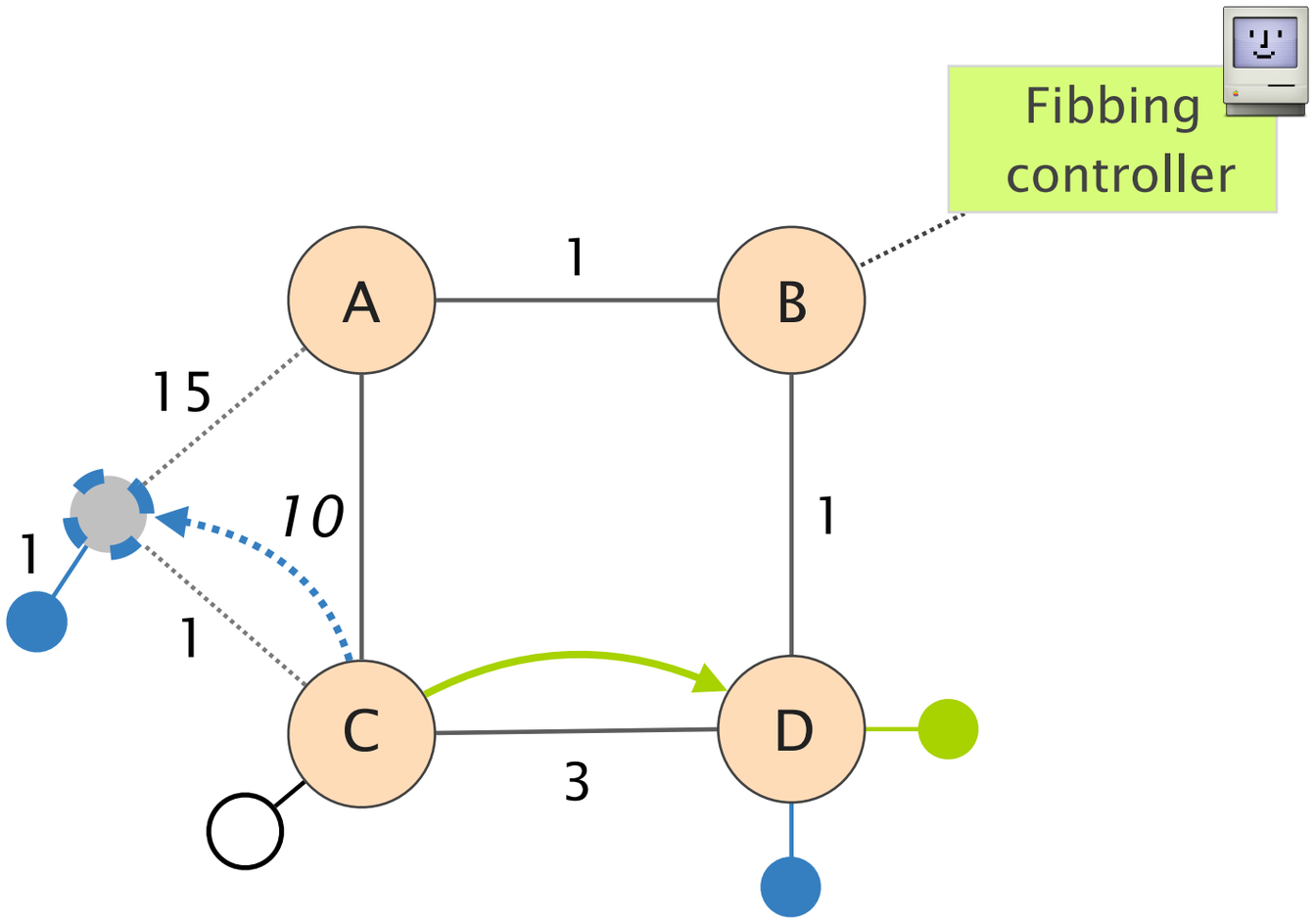
by the routing protocol



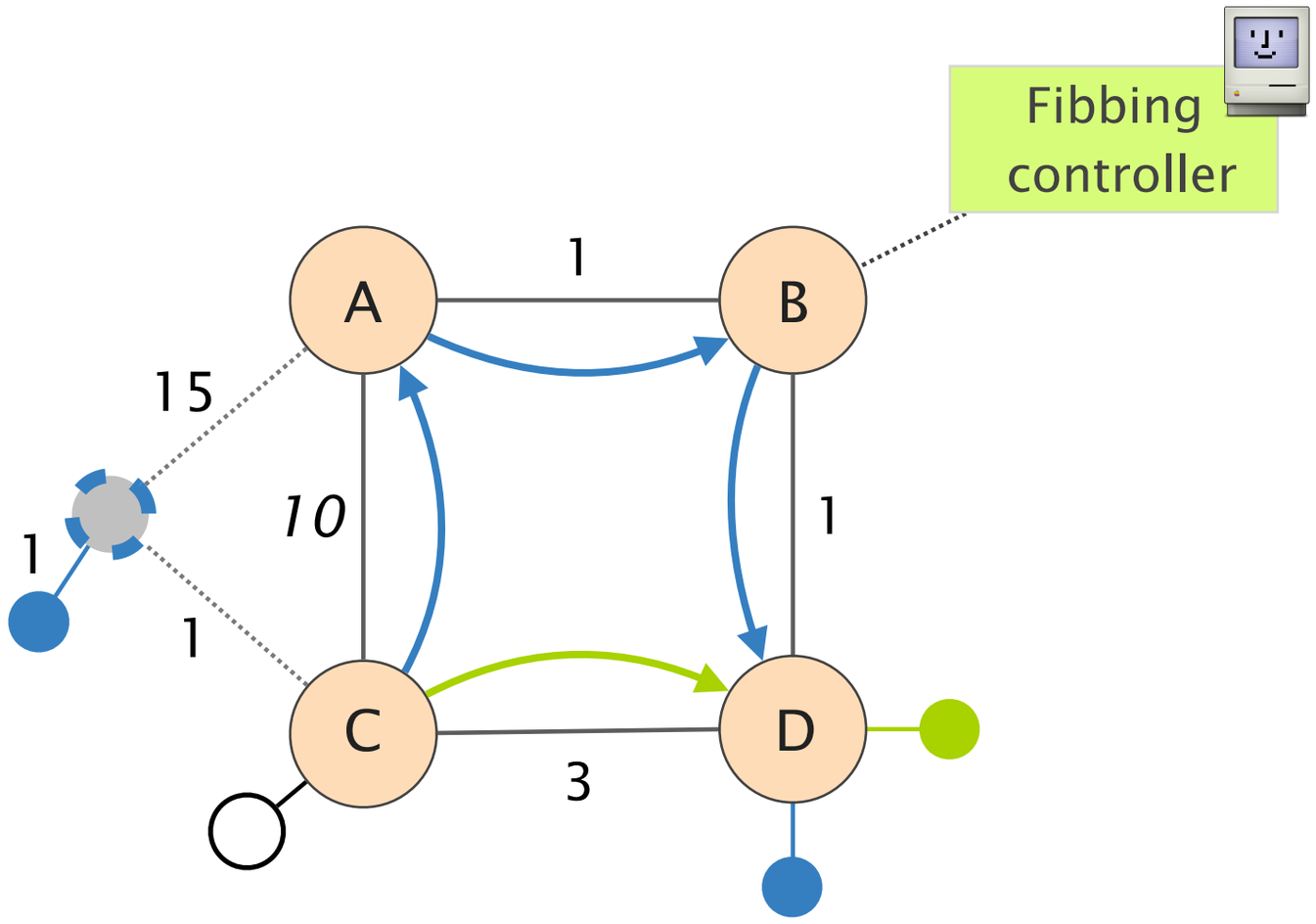
All routers compute their shortest-paths
on the augmented topology



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



Synthesizing routing messages is powerful

Theorem

Fibbing can program

any set of non-contradictory paths

Theorem

Fibbing can program

any set of non-contradictory paths

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)

Synthesizing routing messages is fast and works in practice

We developed efficient algorithms

polynomial in the # of requirements

Compute and minimize topologies in ms

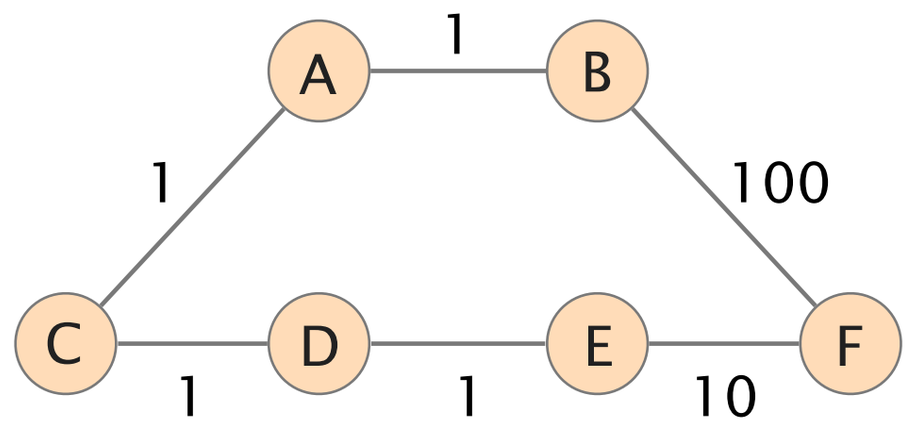
independently of the size of the network

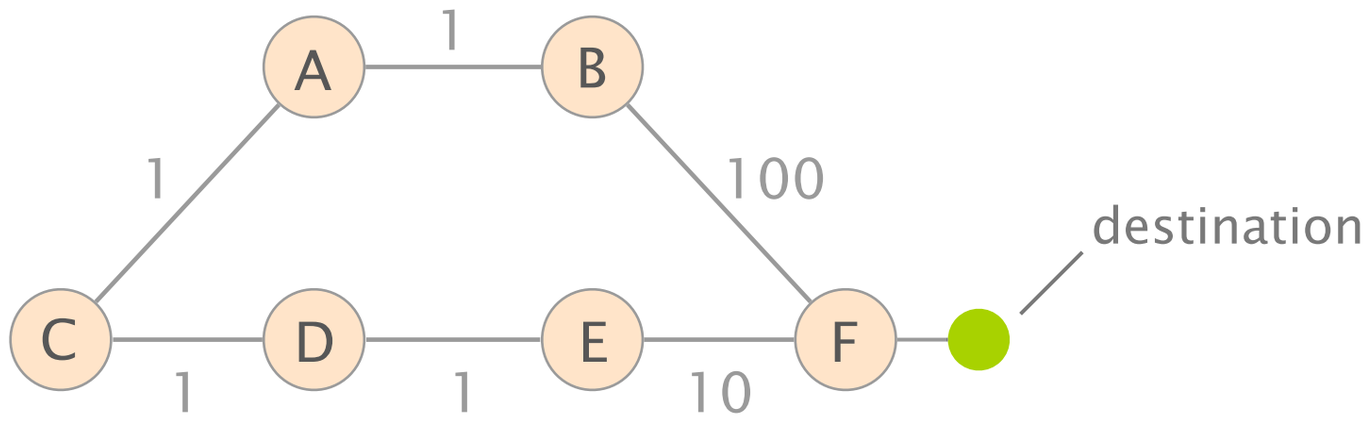
We tested them against real routers

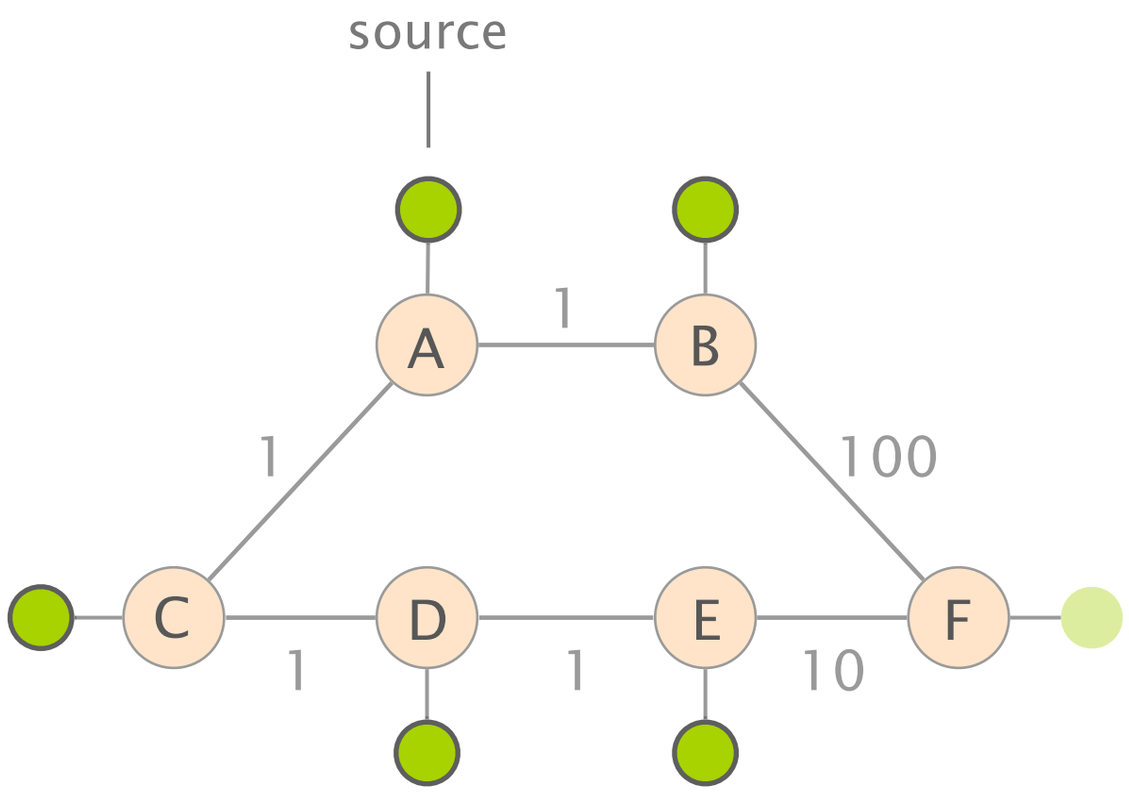
works on both Cisco and Juniper

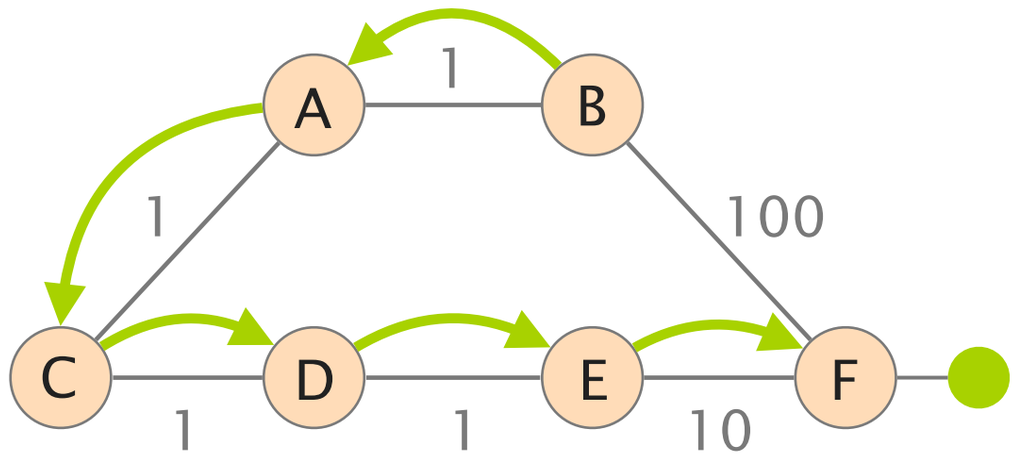
Good news

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


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

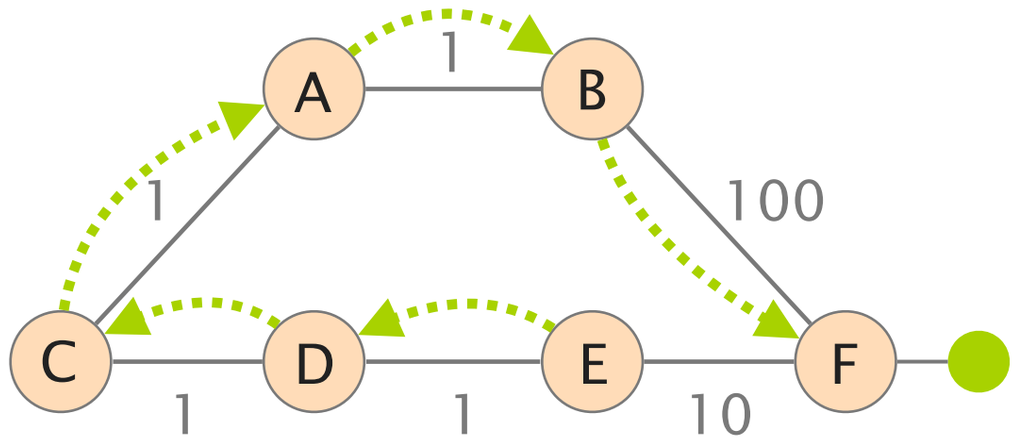

A
B
C
D
E
F
destination
1
1
100
1
1
10


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



original shortest-path

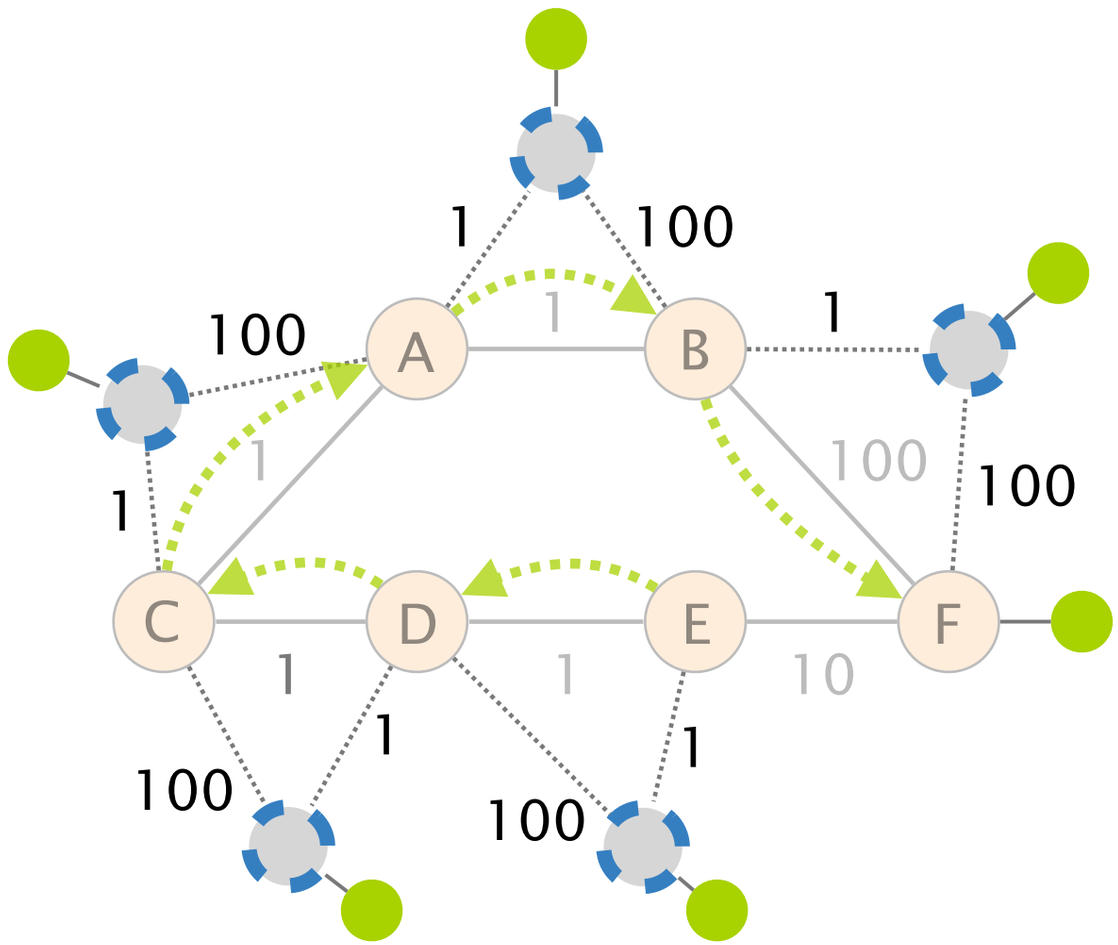
"down and to the right"



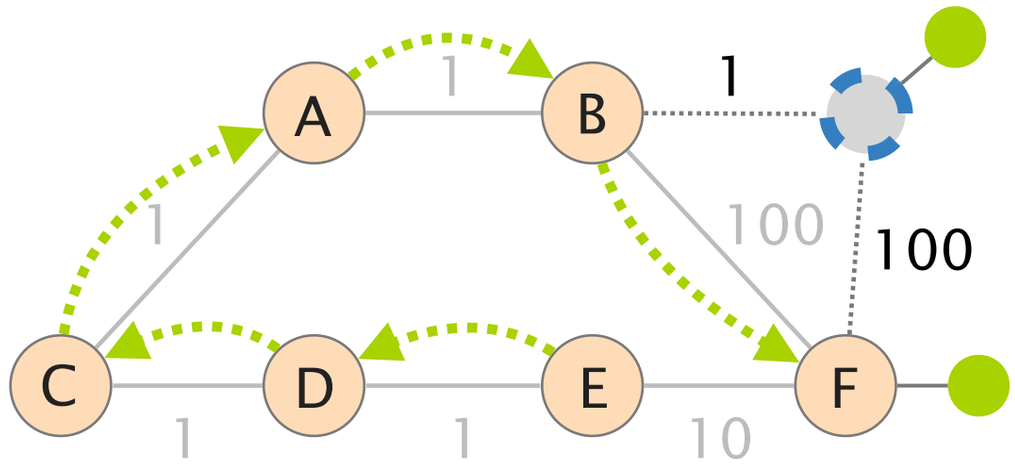
desired shortest-path

“up and to the right”

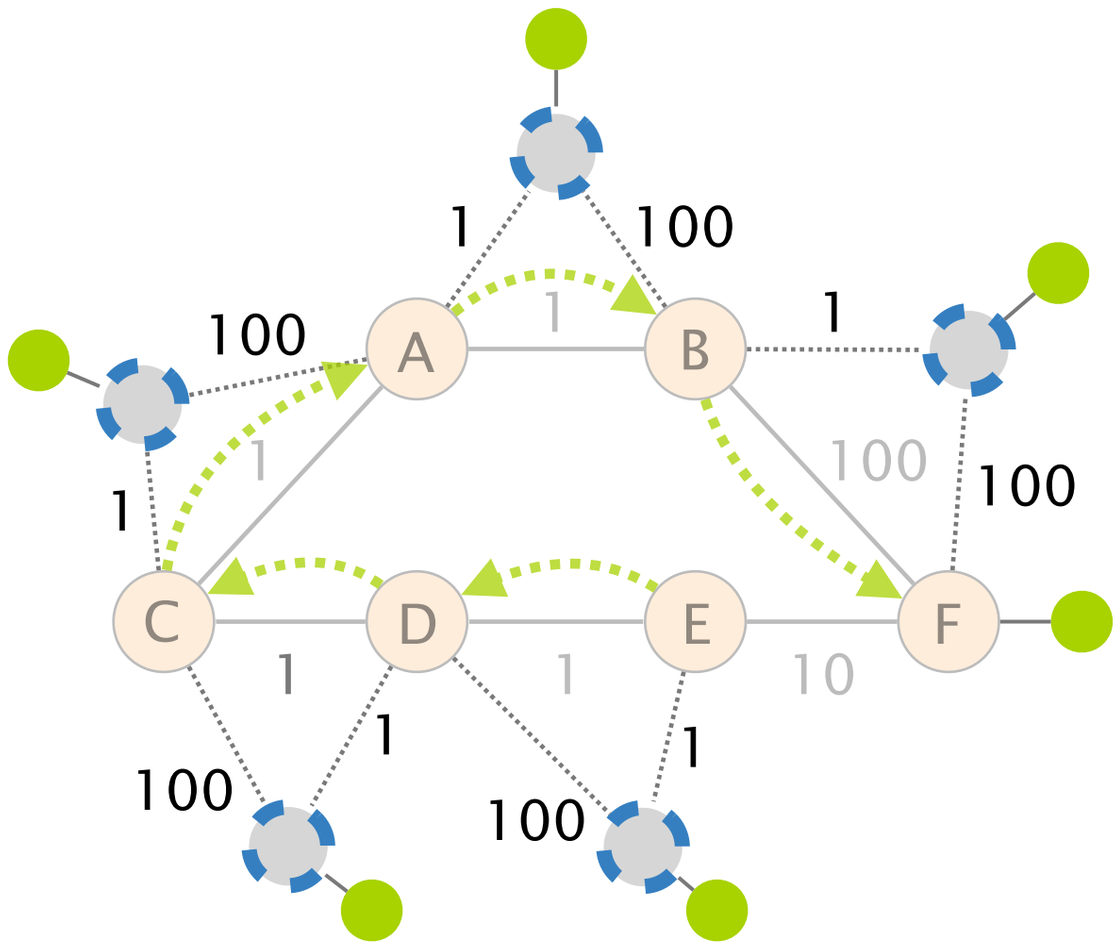
Our naive algorithm would create 5 lies—one per router



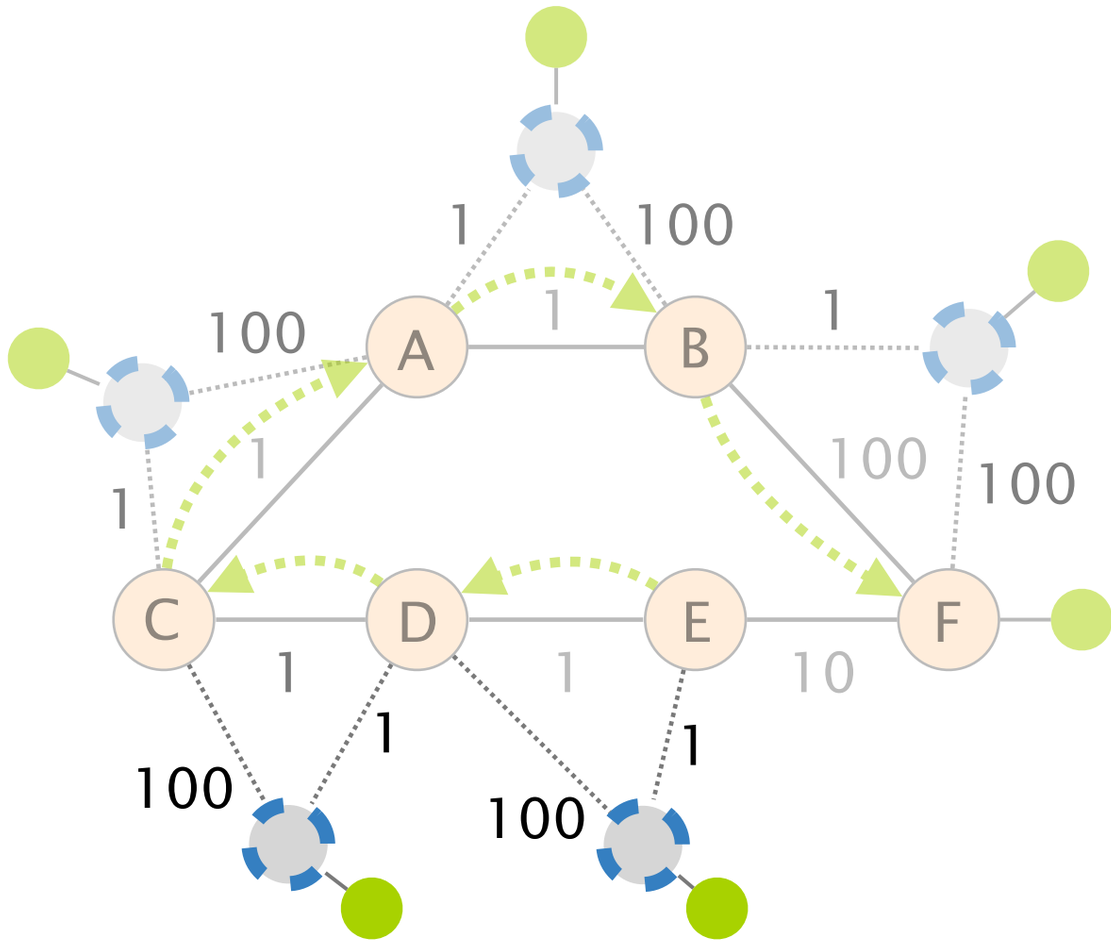
A single lie is sufficient (and necessary)



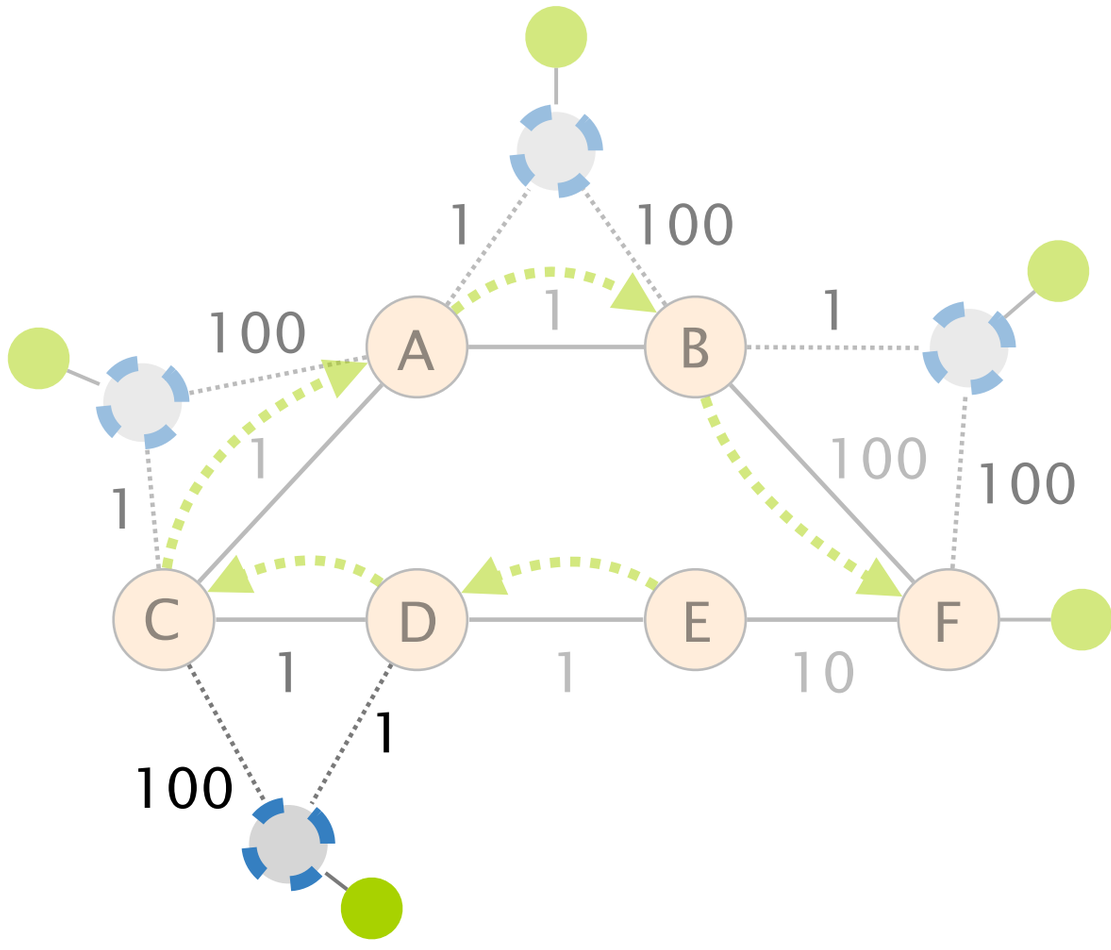
Merger iteratively tries to merge lies produced by the Naive algorithm



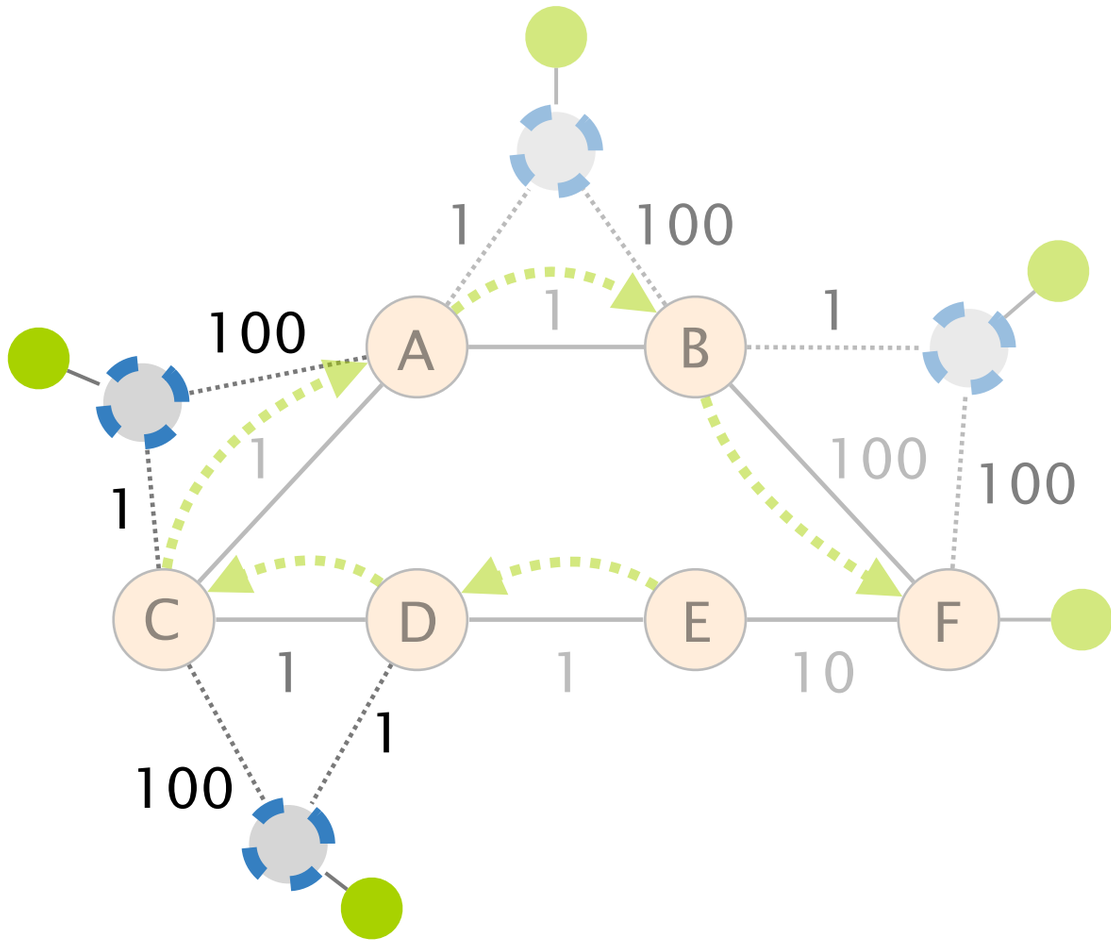
Merger iteratively tries to merge lies produced by the Naive algorithm



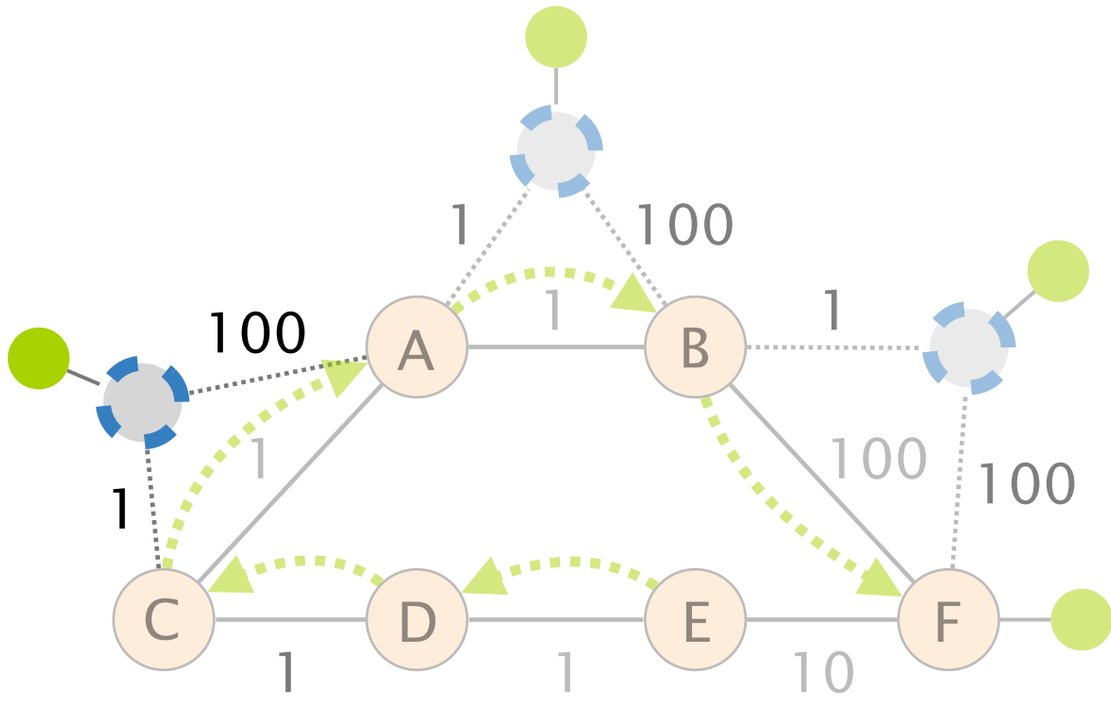
Merger iteratively tries to merge lies produced by the Naive algorithm



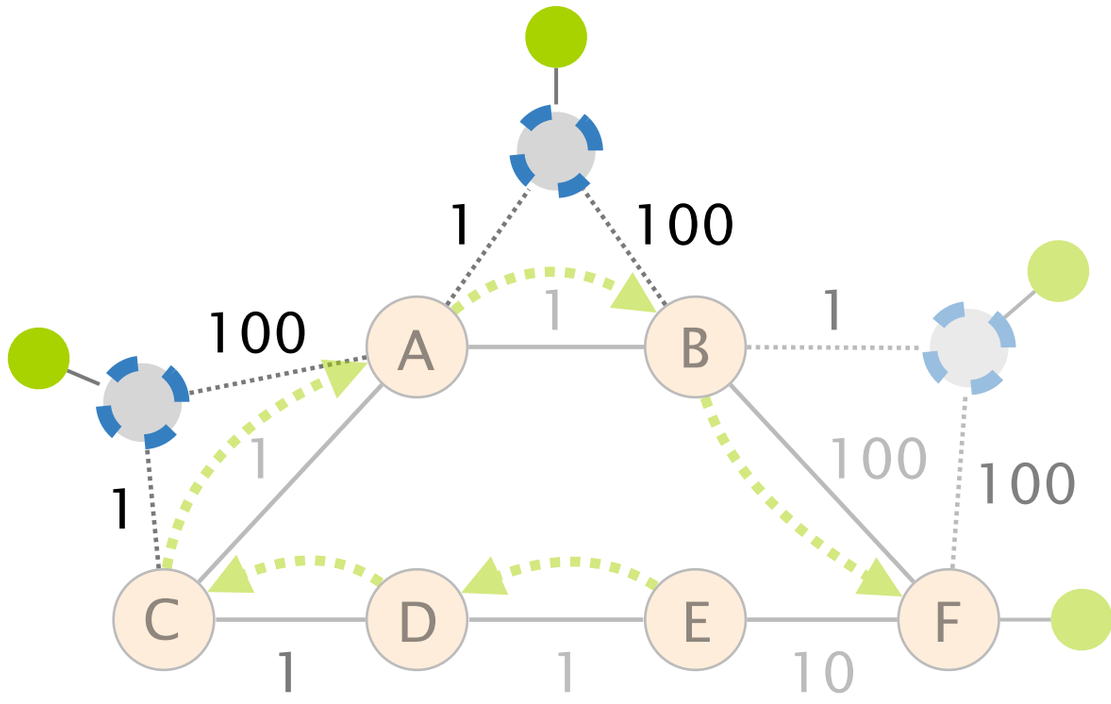
Merger iteratively tries to merge lies produced by the Naive algorithm



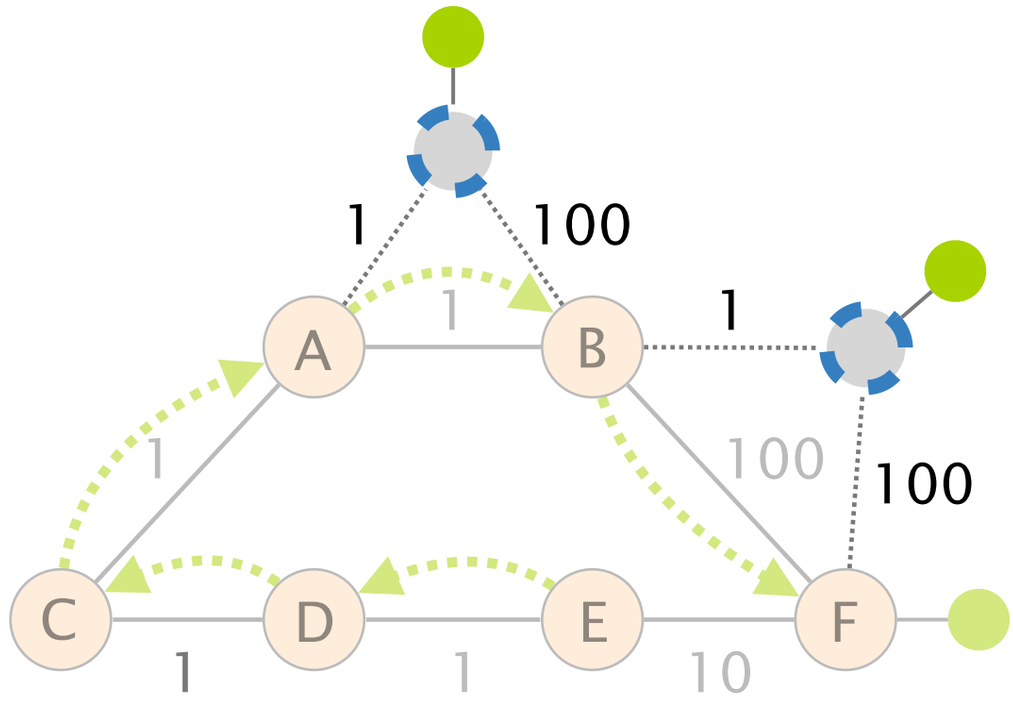
Merger iteratively tries to merge lies produced by the Naive algorithm



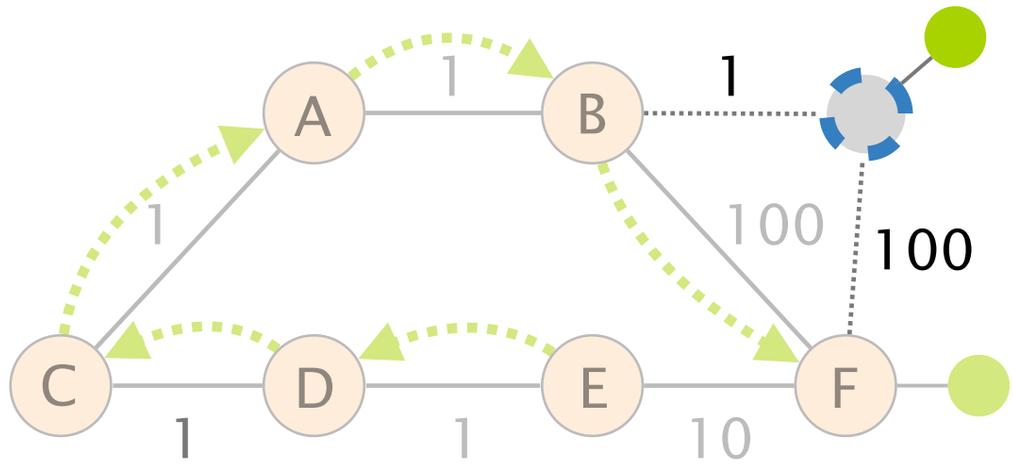
Merger iteratively tries to merge lies produced by the Naive algorithm

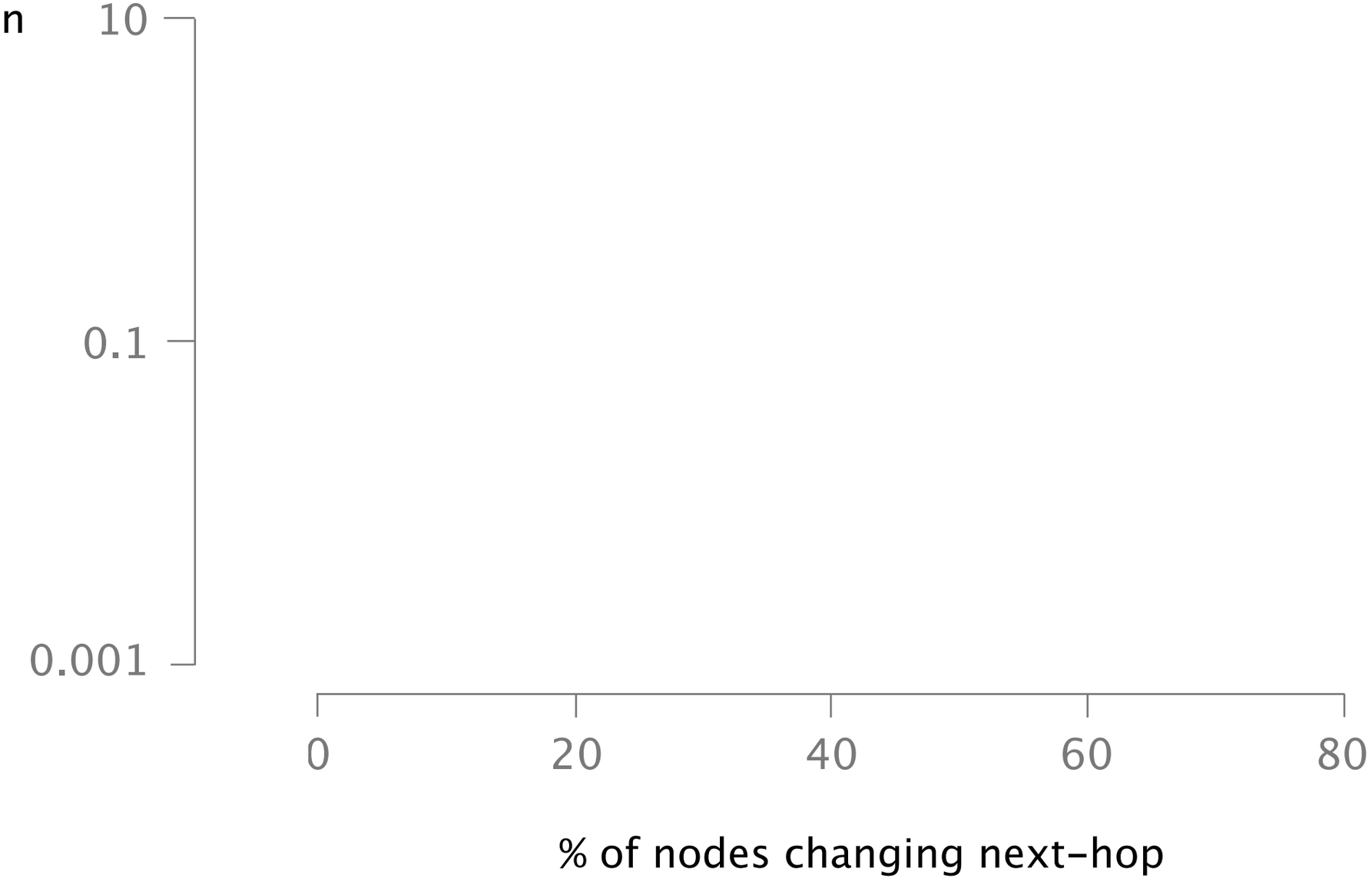


Merger iteratively tries to merge lies produced by the Naive algorithm

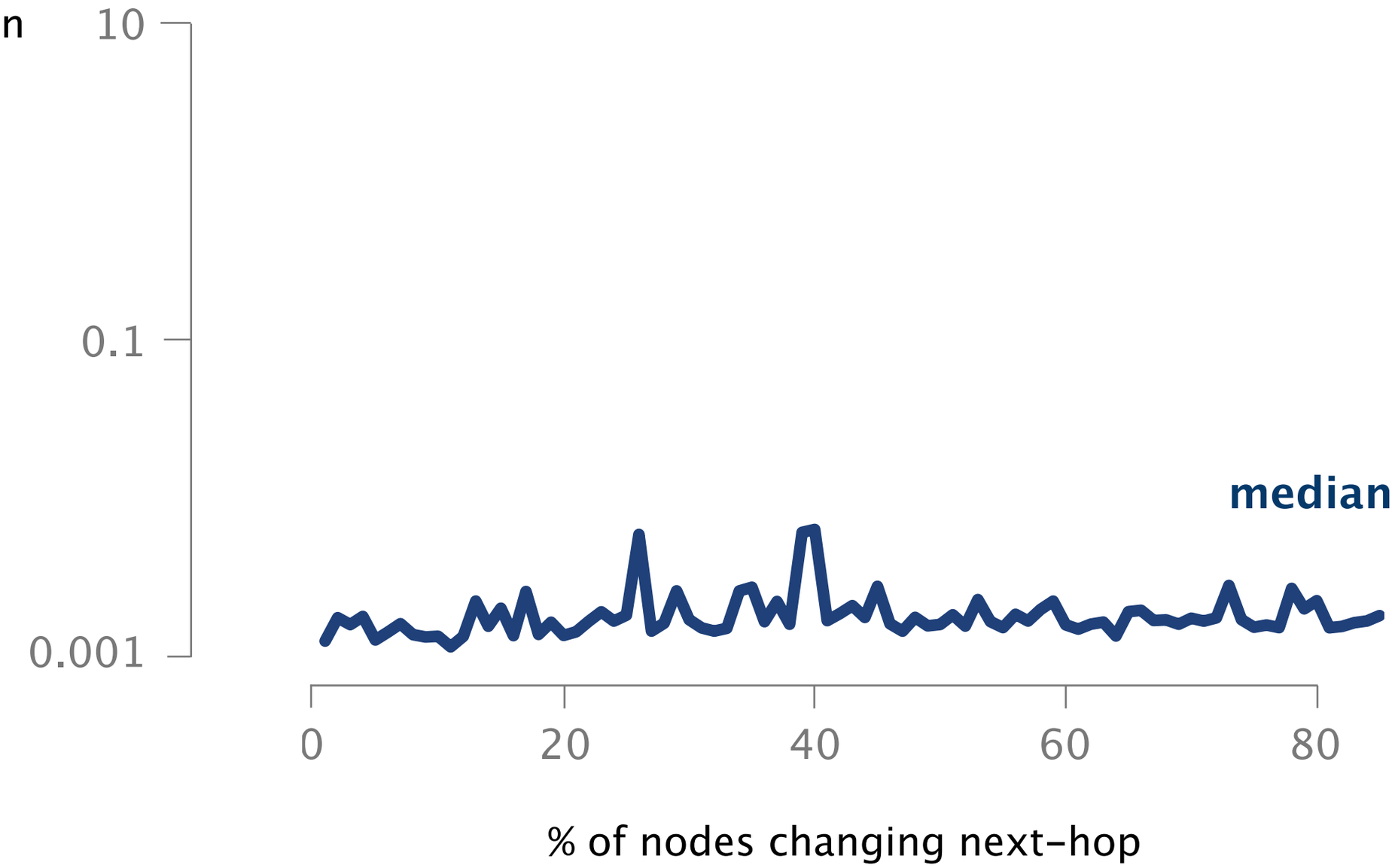


Merger iteratively tries to merge lies produced by the Naive algorithm

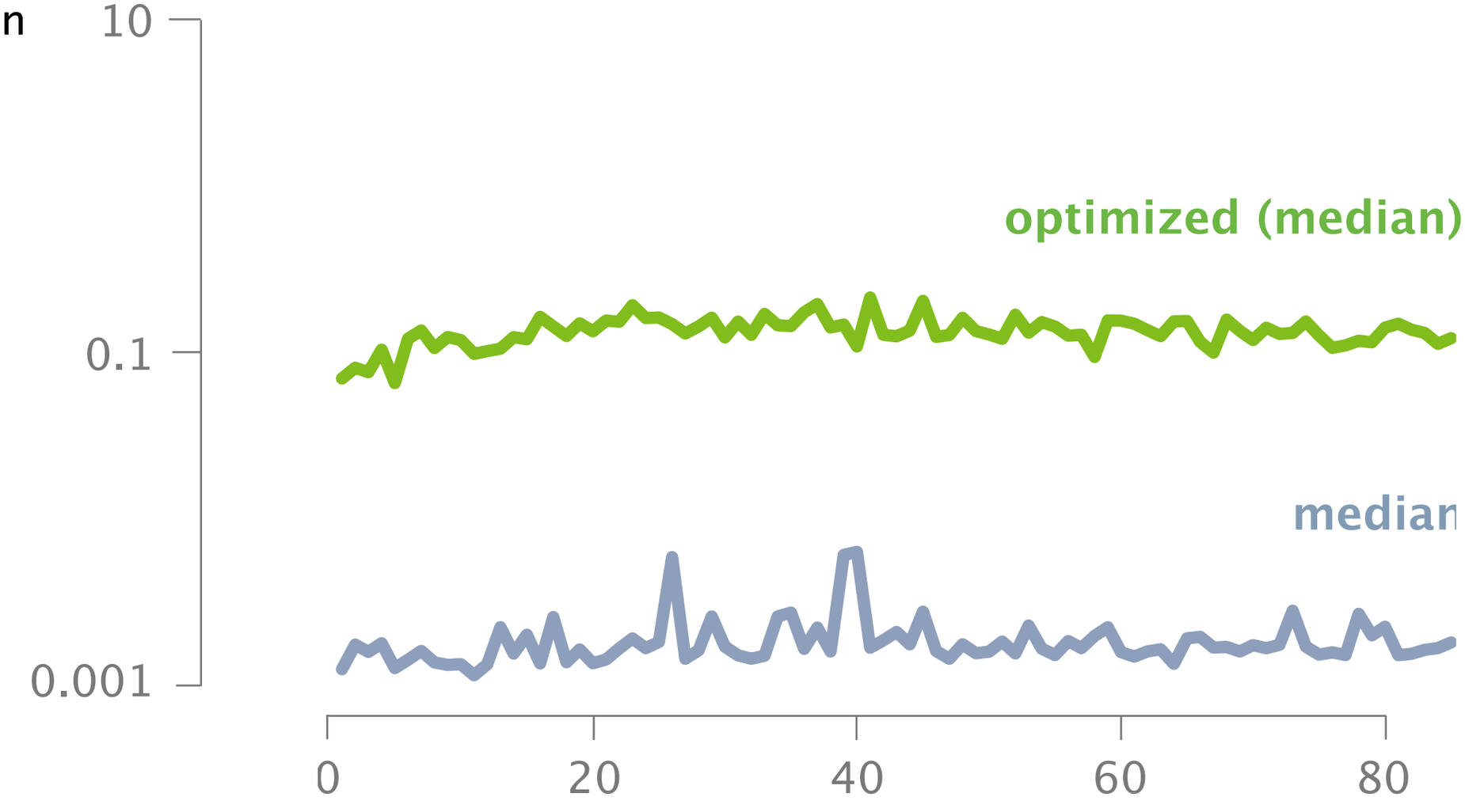



computation time (s)
10
0.1
0.001
0
20
40
60
80
% of nodes changing next-hop

Fibbing computes routing messages to inject in ~1ms



Fibbing minimizes the # of routing messages
to inject in ~100ms



Fibbing is fully implemented
and works with real routers

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

Fibbing is limited though, among others
by the configurations running on the routers

Works with a single protocol family

Dijkstra-based shortest-path routing

Can lead to loads of messages

if the configuration is not adapted

Suffers from reliability issues

need to remove the lies upon failures

Controlling distributed computation

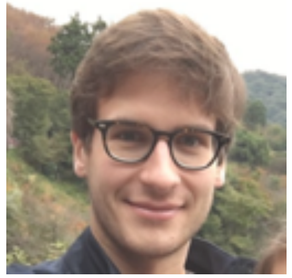
through synthesis

Part 2

NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion

Ahmed El-Hassany

Petar Tsankov



Laurent Vanbever

Martin Vechev

SRL
SOFTWARE RELIABILITY LAB

Networked Systems
ETH Zürich — seit 2015

ETH zürich

Steve Uhlig
9 March at 14:30 ·

Curious if the Internet is also better during IETF/NANOG/RIPE...

Fewer heart attack patients die when top cardiologists are away at conferences, study finds

Heart attack patients are more likely to survive when top cardiologists are not in the hospital, a new study suggests.Researchers at Harvard Medical School...

FLIP.IT

Like Comment Share

Steve Uhlig
9 March at 14:30 ·

Curious if the Internet is also better during IETF/NANOG/RIPE...

Fewer heart attack patients die when top cardiologists are away at conferences, study finds

Heart attack patients are more likely to survive when top cardiologists are not in the hospital, a new study suggests.Researchers at Harvard Medical School...

FLIP.IT

Like Comment Share

Steve Uhlig
9 March at 14:30 ·

Curious if the Internet is also better during IETF/NANOG/RIPE...

Fewer heart attack patients die when top cardiologists are away at conferences, study finds

Heart attack patients are more likely to survive when top cardiologists are not in the hospital, a new study suggests.Researchers at Harvard Medical School...

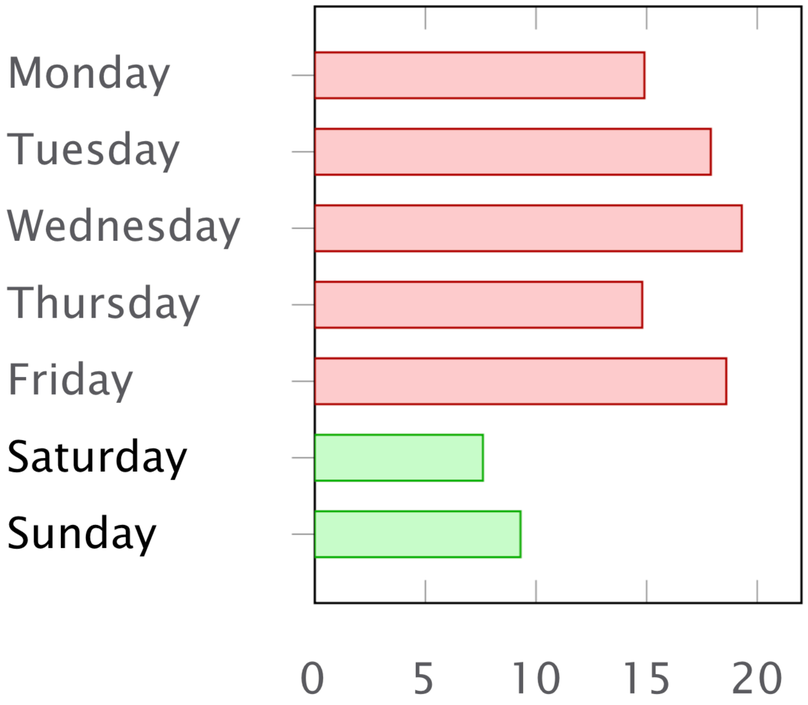
FLIP.IT

Like Comment Share

Yes.

Yes.

The Internet seems to be better off during week-ends…



source: Job Snijders (NTT)

This is a far too common story…

News

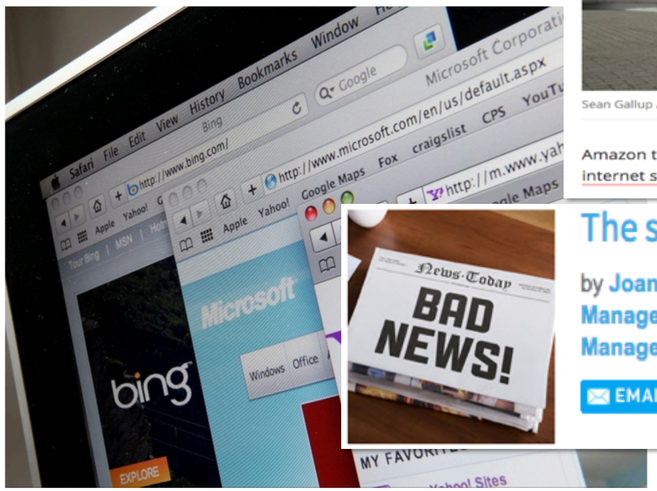
'Configuration Error' for AWS Outage

By David Ramel ■ 08/12/2015

A "configuration error" caused this week's

Widespread internet outages affected Comcast, Spectrum, Verizon and AT&T customers

BY: CNN
POSTED: 1:45 PM, Nov 6, 2017
UPDATED: 7:35 PM, Nov 6, 2017



People around the U.S. experienced some internet downtime on Monday. The outage was brief and service has been restored.

Amazon's massive AWS outage was cau

One incorrect command and the whole internet suffers.

BY JASON DEL REY | @DELREY | MAR 2, 2017, 2:20PM EST

TWEET SHARE LINKEDIN

Sean Gallup / Getty

Amazon today blamed human error for the the big AWS outage that took down a b internet sites for several hours on Tuesday afternoon.

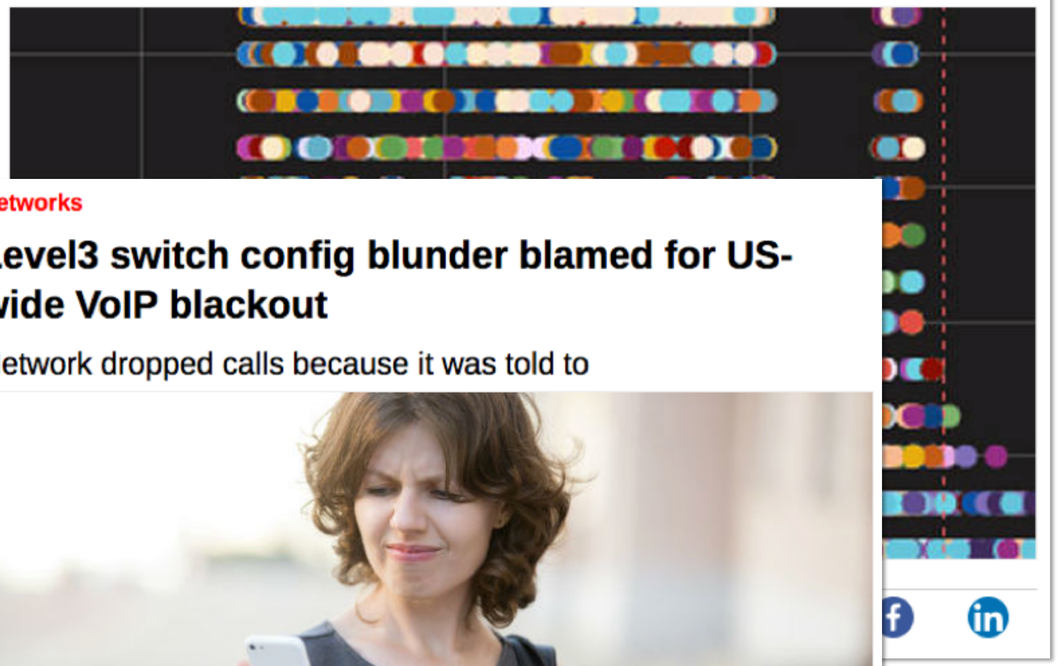
The summer of network misconfigurations

by Joanne Godfrey on August 11, 2016 in Application Connectivity Management, Firewall Change Management, Information Security Management and Vulnerabilities, Security Policy Management

EMAIL Share 32 Tweet Share 6 Lik

CloudFlare apologizes for Telia screwing you over

Unhappy about massive outage



Networks

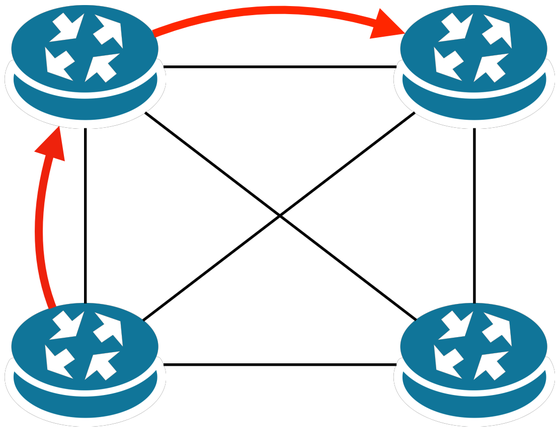
Level3 switch config blunder blamed for US-wide VoIP blackout

Network dropped calls because it was told to

5 Oct 2016 at 11:33, Shaun Nichols

Why do we have so many misconfigurations?

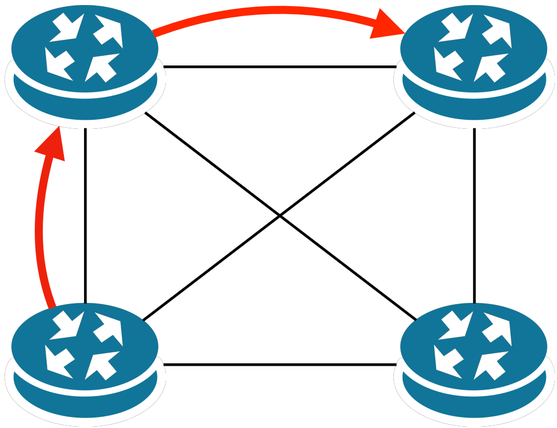
Given



an existing network behavior

induced by a low-level configuration C

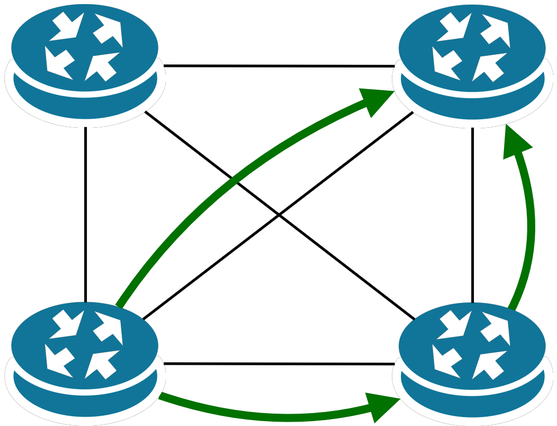
Given



an existing network behavior

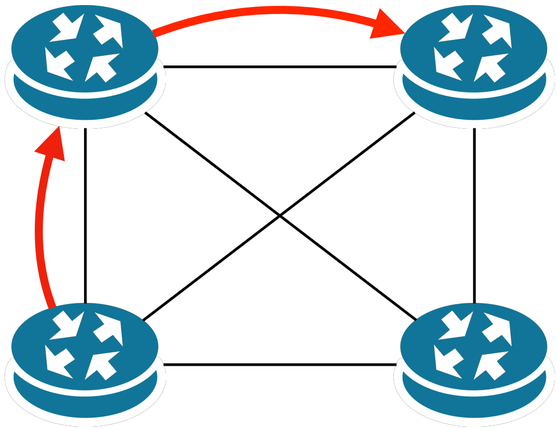
induced by a low-level configuration C

and

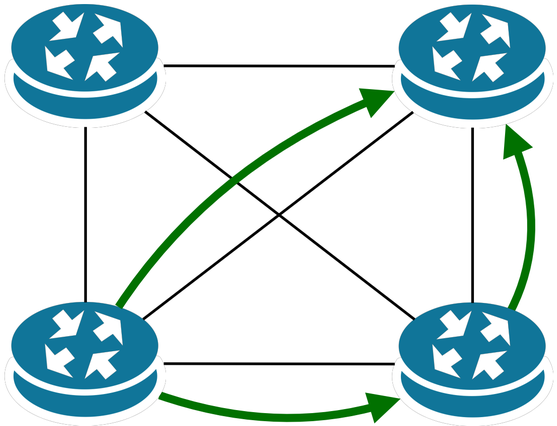


a desired network behavior

Given



and



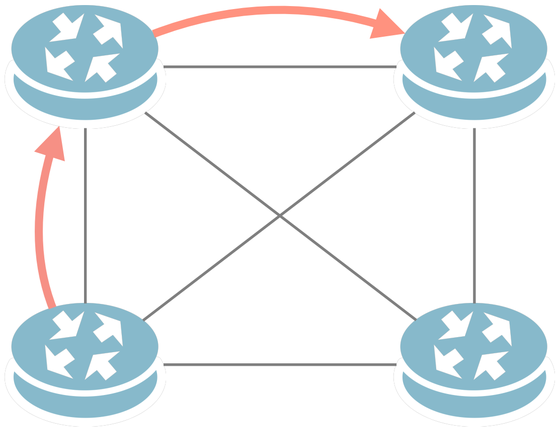
an existing network behavior

induced by a low-level configuration C

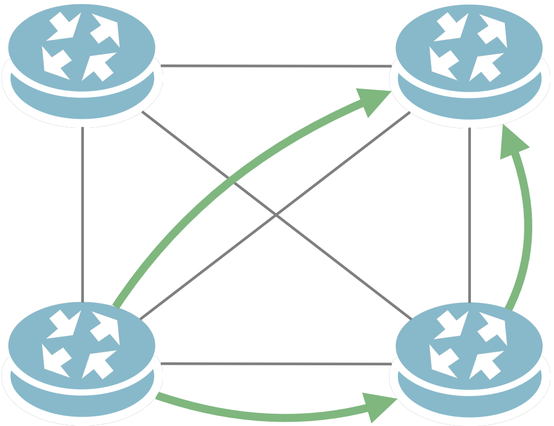
a desired network behavior

Adapt C so that the network follows the new behavior

Given



and



an existing network behavior

induced by a low-level configuration C

a desired network behavior

Adapt C so that the network follows the new behavior

Nowadays these adaptations are still mostly done manually, which is error-prone and time-consuming

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
  neighbor 125.1.17.1 activate
 !
```

Nowadays these adaptations are still mostly done manually, which is error-prone and time-consuming

Cisco IOS

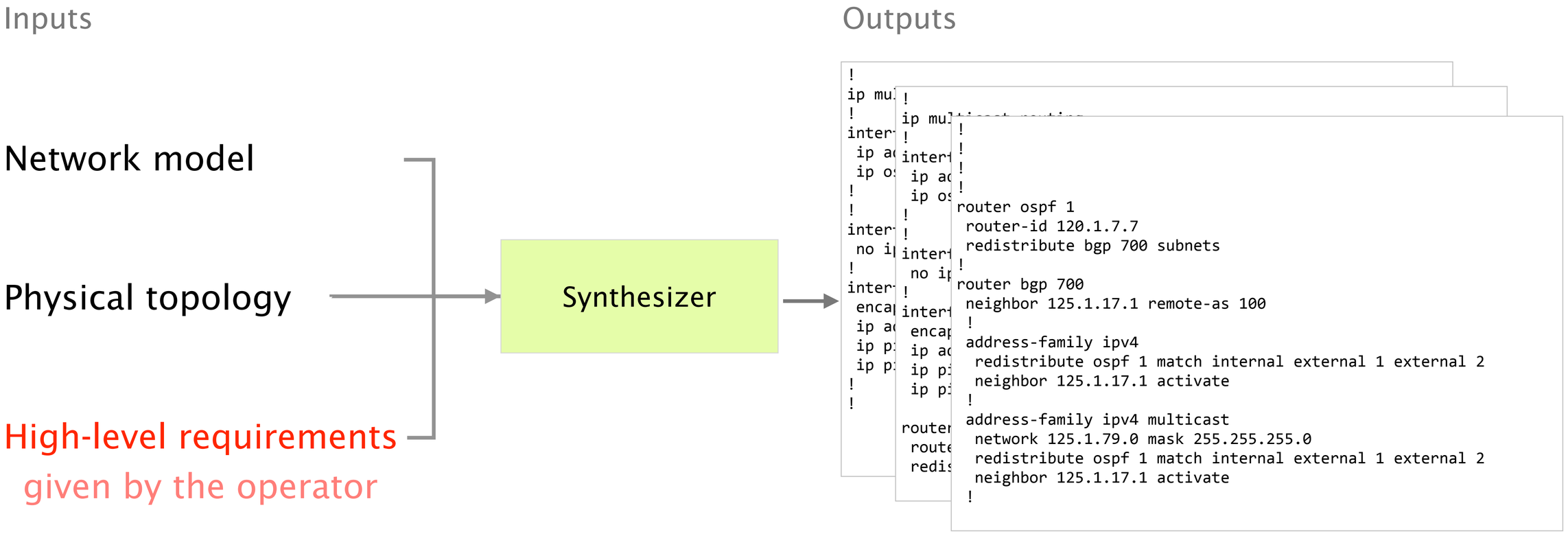
```
!
ip multicast-routing
!
interface Loopback0
 ip address 120.1.7.7 255.255.255.255
 ip ospf 1 area 0
!
!
interface Ethernet0/0
 no ip address
!
interface Ethernet0/0.17
 encapsulation dot1Q 17
 ip address 125.1.17.7 255.255.255.0
 ip pim bsr-border
 ip pim sparse-mode
!
!
router ospf 1
 router-id 120.1.7.7
 redistribute bgp 700 subnets
!
…
```

Anything else than 700 creates blackholes

```
…
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
  neighbor 125.1.17.1 activate
 !
```

Configuration synthesis addresses this problem by deriving low-level configurations from high-level requirements

Configuration synthesis addresses this problem by deriving low-level configurations from high-level requirements



Configuration synthesis:

a booming research area!

	Out of high-level requirements, automatically derive…
Genesis [POPL'17]	forwarding rules
Propane [SIGCOMM'16] PropaneAT [PLDI'17]	BGP configurations
SyNET [CAV'17] Zeppelin [SIGMETRICS'18]	OSPF + BGP configurations

Synthesizing configuration is great, but comes with challenges preventing a wide adoption

Existing synthesizers…

Existing synthesizers…

Problem #1

interpretability

can produce configurations that
widely differ from humanly-generated ones

Existing synthesizers…

Problem #1
interpretability

can produce configurations that
widely differ from humanly-generated ones

Problem #2
continuity

can produce widely different configurations
given slightly different requirements

Existing synthesizers…

Problem #1
interpretability

can produce configurations that
widely differ from humanly-generated ones

Problem #2
continuity

can produce widely different configurations
given slightly different requirements

Problem #3
deployability

cannot flexibly adapt to operational requirements,
requiring configuration heterogeneity

A key issue is that synthesizers do not provide operators
with a fine-grained control over the synthesized configurations

Introducing...

NetComplete

NetComplete allows network operators to flexibly express their intents through configuration sketches

A configuration with “holes”

```
interface TenGigabitEthernet1/1/1
  ip address ? ?
  ip ospf cost 10 < ? < 100

router ospf 100
  ?
  ...

router bgp 6500
  ...
  neighbor AS200 import route-map imp-p1
  neighbor AS200 export route-map exp-p1
  ...
ip community-list C1 permit ?
ip community-list C2 permit ?
```

```
route-map imp-p1 permit 10
  ?

route-map exp-p1 ? 10
  match community C2
route-map exp-p2 ? 20
  match community C1
...
```

```
interface TenGigabitEthernet1/1/1
  ip address ? ?
  ip ospf cost 10 < ? < 100

router ospf 100
  ?
  ...

router bgp 6500
  ...
  neighbor AS200 import route-map imp-p1
  neighbor AS200 export route-map exp-p1
  ...
ip community-list C1 permit ?
ip community-list C2 permit ?
```

Holes can identify
specific attributes such as:

- IP addresses
- link costs
- BGP local preferences

```
interface TenGigabitEthernet1/1/1
  ip address ? ?
  ip ospf cost 10 < ? < 100

router ospf 100
  ?
  ...

router bgp 6500
  ...
  neighbor AS200 import route-map imp-p1
  neighbor AS200 export route-map exp-p1
  ...
ip community-list C1 permit ?
ip community-list C2 permit ?
```

```
route-map imp-p1 permit 10
  ?

route-map exp-p1 ? 10
  match community C2
route-map exp-p2 ? 20
  match community C1
...
```

Holes can also identify

entire pieces of the configuration

NetComplete “autocompletes” the holes such that
the output configuration complies with the requirements

```
interface TenGigabitEthernet1/1/1
  ip address ? ?
  ip ospf cost 10 < ? < 100

router ospf 100
  ?
  ...

router bgp 6500
  ...
  neighbor AS200 import route-map imp-p1
  neighbor AS200 export route-map exp-p1
  ...
ip community-list C1 permit ?
ip community-list C2 permit ?
```

```
route-map imp-p1 permit 10
  ?

route-map exp-p1 ? 10
  match community C2
route-map exp-p2 ? 20
  match community C1
...
```

```
interface TenGigabitEthernet1/1/1
  ip address 10.0.0.1 255.255.255.254
  ip ospf cost 15

router ospf 100
  network 10.0.0.1 0.0.0.1 area 0.0.0.0


router bgp 6500
  ...
  neighbor AS200 import route-map imp-p1
  neighbor AS200 export route-map exp-p1
  ...
ip community-list C1 permit 6500:1
ip community-list C2 permit 6500:2
```

```
route-map imp-p1 permit 10
  set community 6500:1
  set local-pref 50
route-map exp-p1 permit 10
  match community C2
route-map exp-p2 deny 20
  match community C1
...
```

NetComplete reduces the autocompletion problem
to a constraint satisfaction problem

First Encode the
- protocol semantics
- high-level requirements as a logical formula (in SMT)
- partial configurations

First Encode the
- protocol semantics
- high-level requirements as a logical formula (in SMT)
- partial configurations

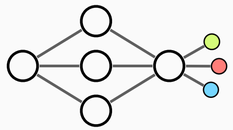
Then Use a solver (Z3) to find an assignment for the undefined configuration variables s.t. the formula evaluates to True

Inputs

NetComplete

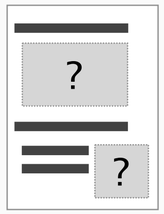
Outputs

Topology



Reqs

Sketch



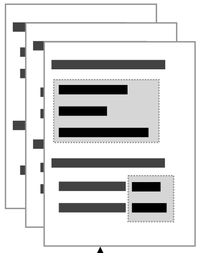
Links/adjacencies/
static routes synthesis

BGP synthesis

OSPF synthesis

Z3 solver

network-wide
configurations

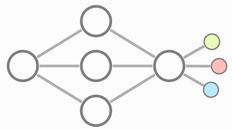


Inputs

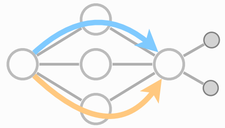
NetComplete

Outputs

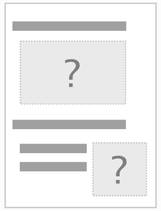
Topology



Reqs



Sketch



Links/adjacencies/
static routes synthesis

φ_{STATIC}

BGP synthesis

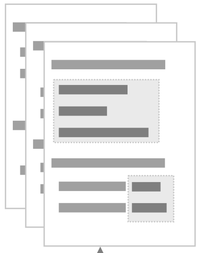
$\varphi_{BGP} + \varphi_{STATIC}$

OSPF synthesis

$\varphi_{OSPF} + \varphi_{BGP} + \varphi_{STATIC}$

Z3 solver

network-wide
configurations



Main challenge:

Scalability

Insight #1

network-specific heuristics

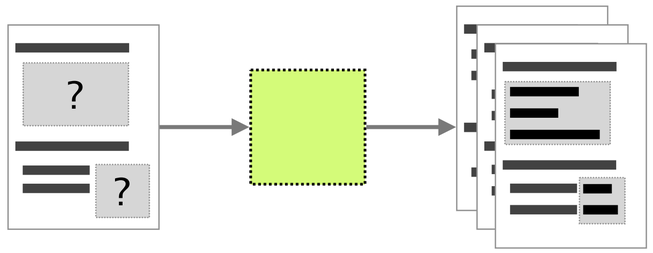
search space navigation

Insight #2

partial evaluation

search space reduction

NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion



1. **BGP synthesis**
   optimized encoding
2. **OSPF synthesis**
   counter-examples-based
3. **Evaluation**
   flexible, *yet* scalable

But first…

"How to configure routing protocols" 101

inter-domain routing	intra-domain routing
BGP	OSPF

But first…

"How to configure routing protocols" 101

inter-domain routing

BGP

intra-domain routing

Internet

Internet

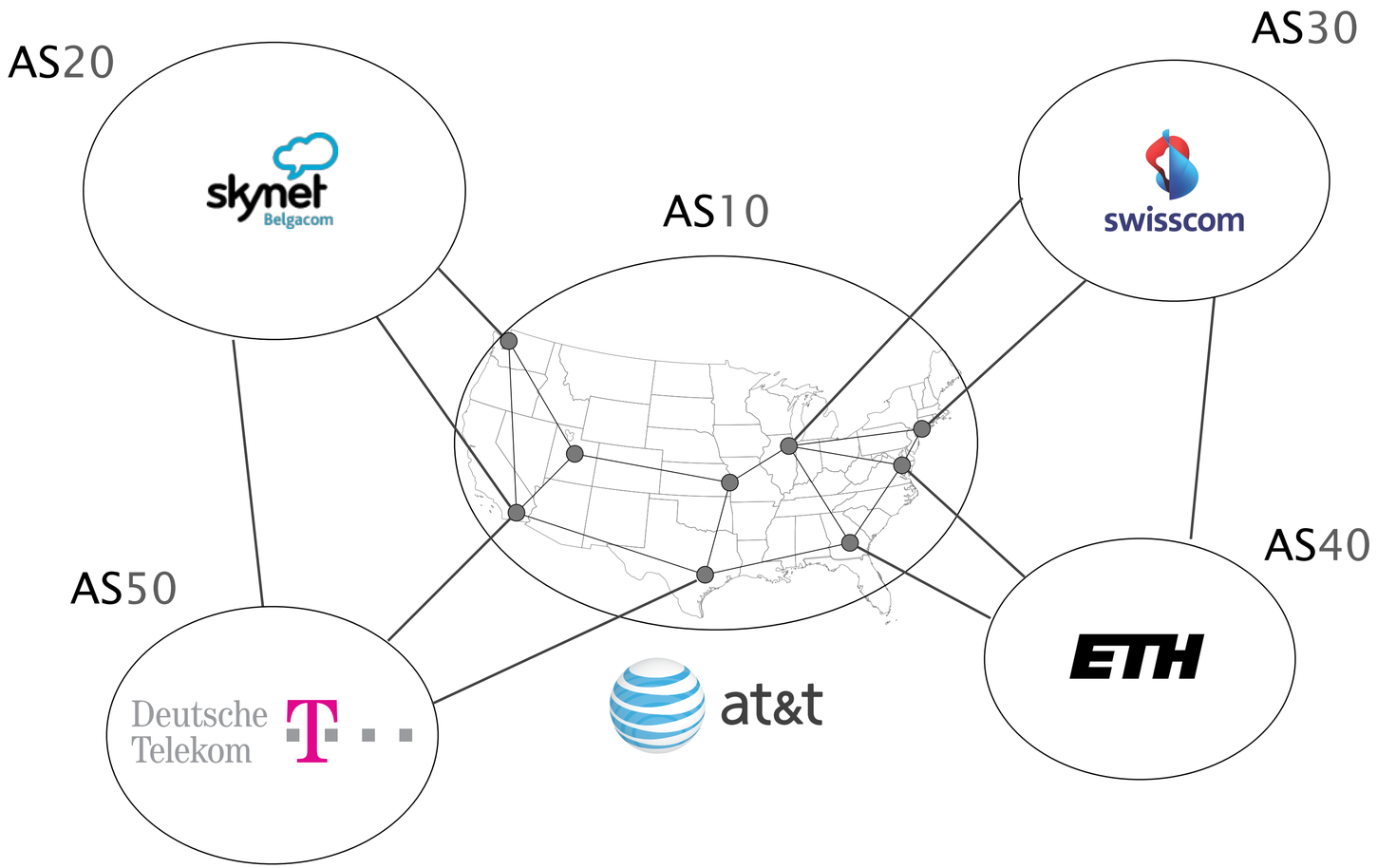
Internet

A network of *networks*

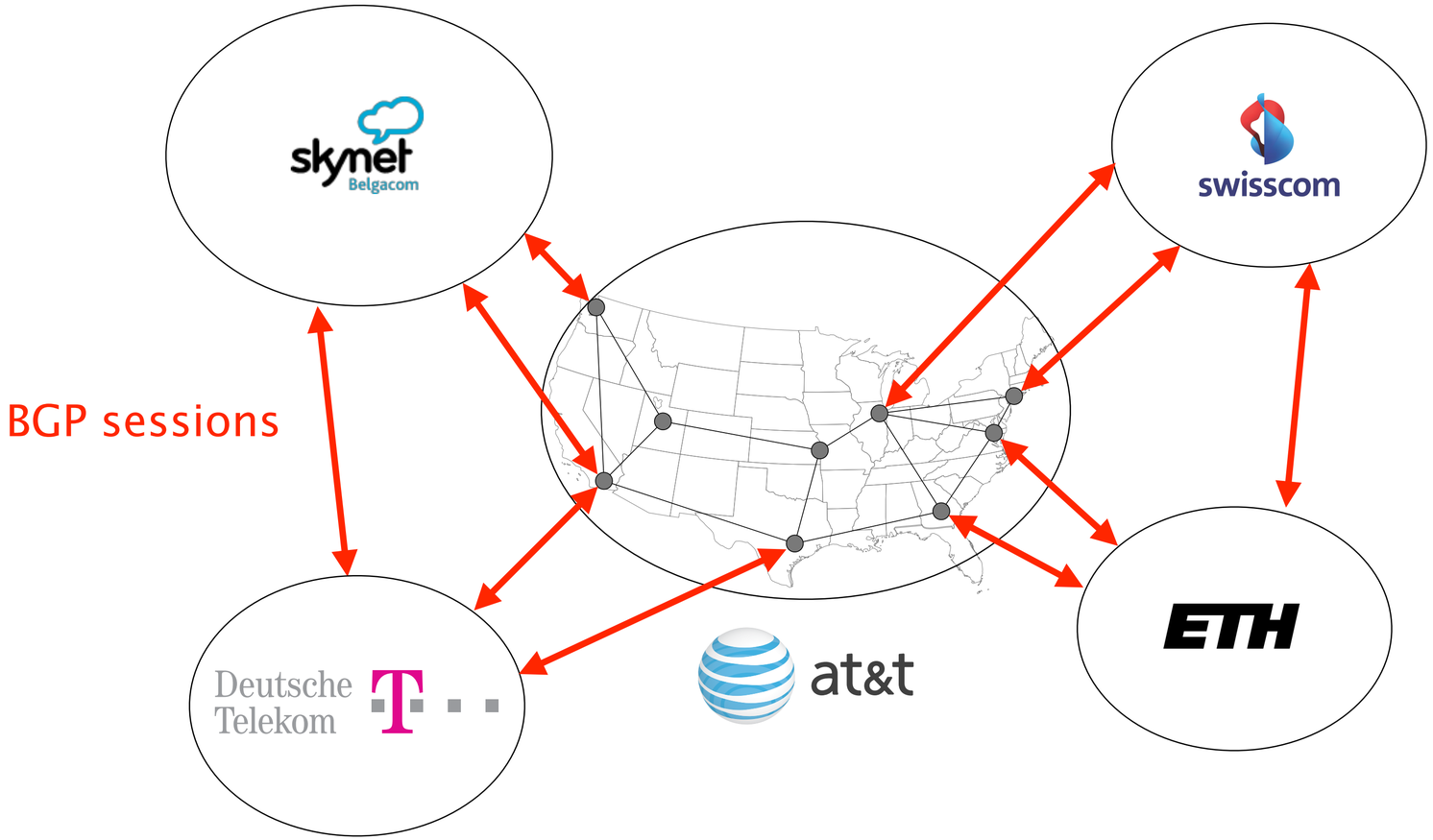
Internet

Border Gateway Protocol (BGP)

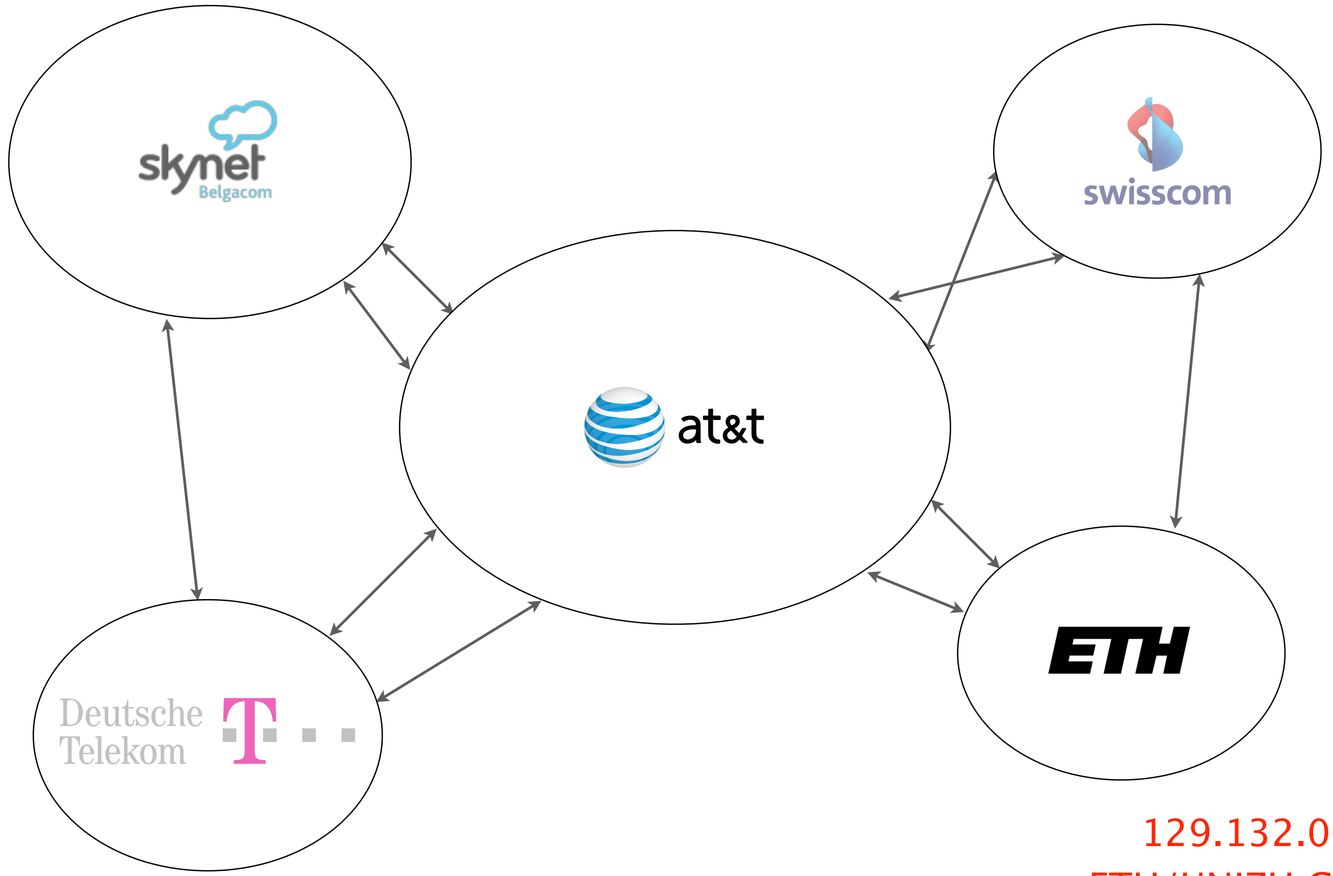
The Internet is a network of networks,
referred to as Autonomous Systems (AS)



BGP is the routing protocol
"glueing" the Internet together

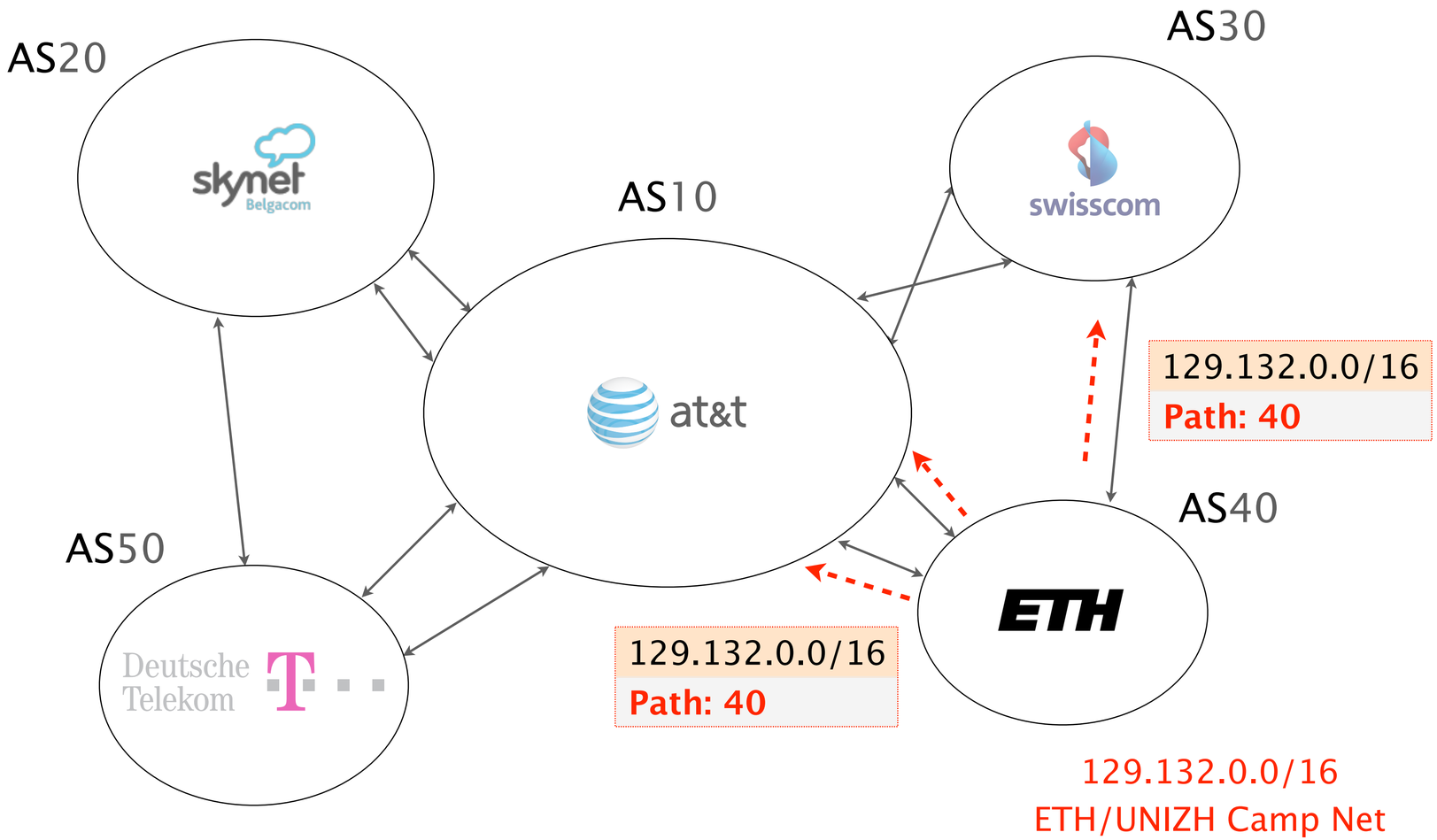


Using BGP, ASes exchange information about
the IP prefixes they can reach, directly or indirectly

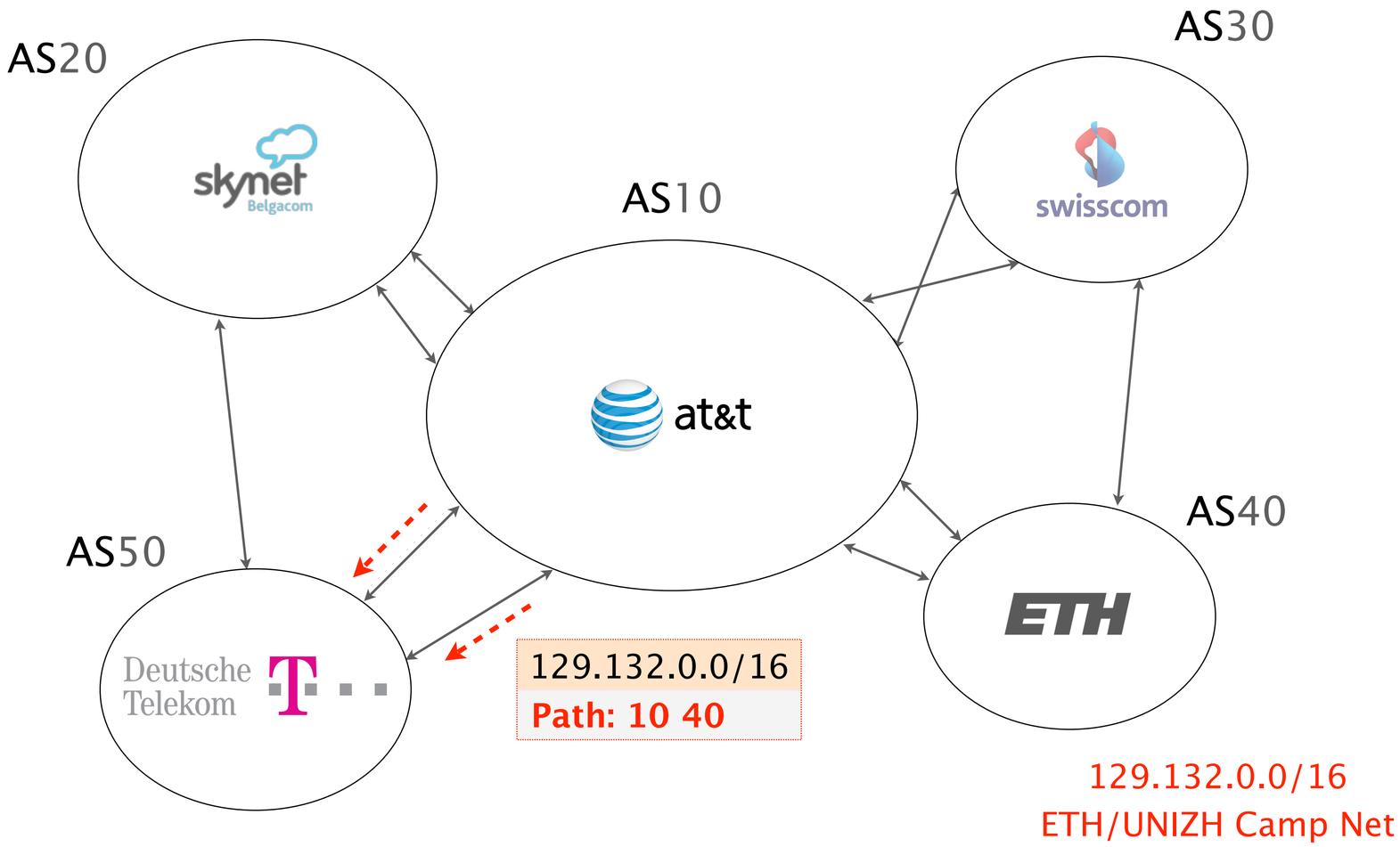


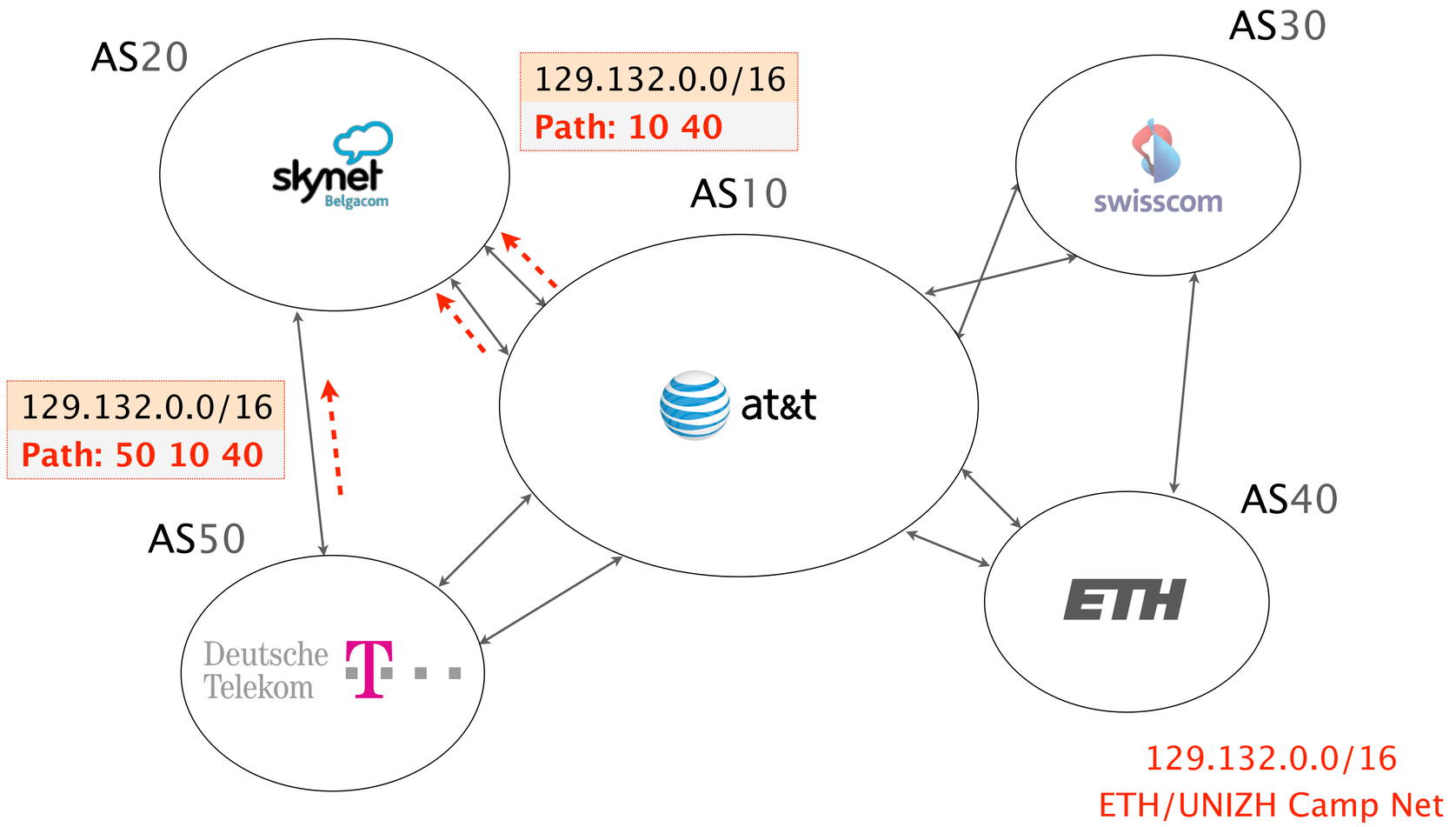
129.132.0.0/16
ETH/UNIZH Camp Net

BGP routes carry complete path information instead of distance



Each AS appends itself to the path when it propagates announcements




AS20
skynet
Belgacom
129.132.0.0/16
Path: 10 40
AS10
AS30
swisscom
at&t
129.132.0.0/16
Path: 50 10 40
AS50
AS40
Deutsche
Telekom
ETH
129.132.0.0/16
ETH/UNIZH Camp Net

Network operators need to configure each router
to adapt how it selects and exports BGP advertisements

Network operators need to configure each router
to adapt how it selects and exports BGP advertisements

Selection

out of all paths a router receives:

along which one should it direct traffic?

Network operators need to configure each router
to adapt how it selects and exports BGP advertisements

Selection

out of all paths a router receives:

along which one should it direct traffic?

control where traffic is going

Network operators need to configure each router
to adapt how it selects and exports BGP advertisements

Selection

out of all paths a router receives:

along which one should it direct traffic?

control where traffic is going

Export

for each selected path:

to which neighbors propagate it?

Network operators need to configure each router
to adapt how it selects and exports BGP advertisements

Selection

out of all paths a router receives:

along which one should it direct traffic?

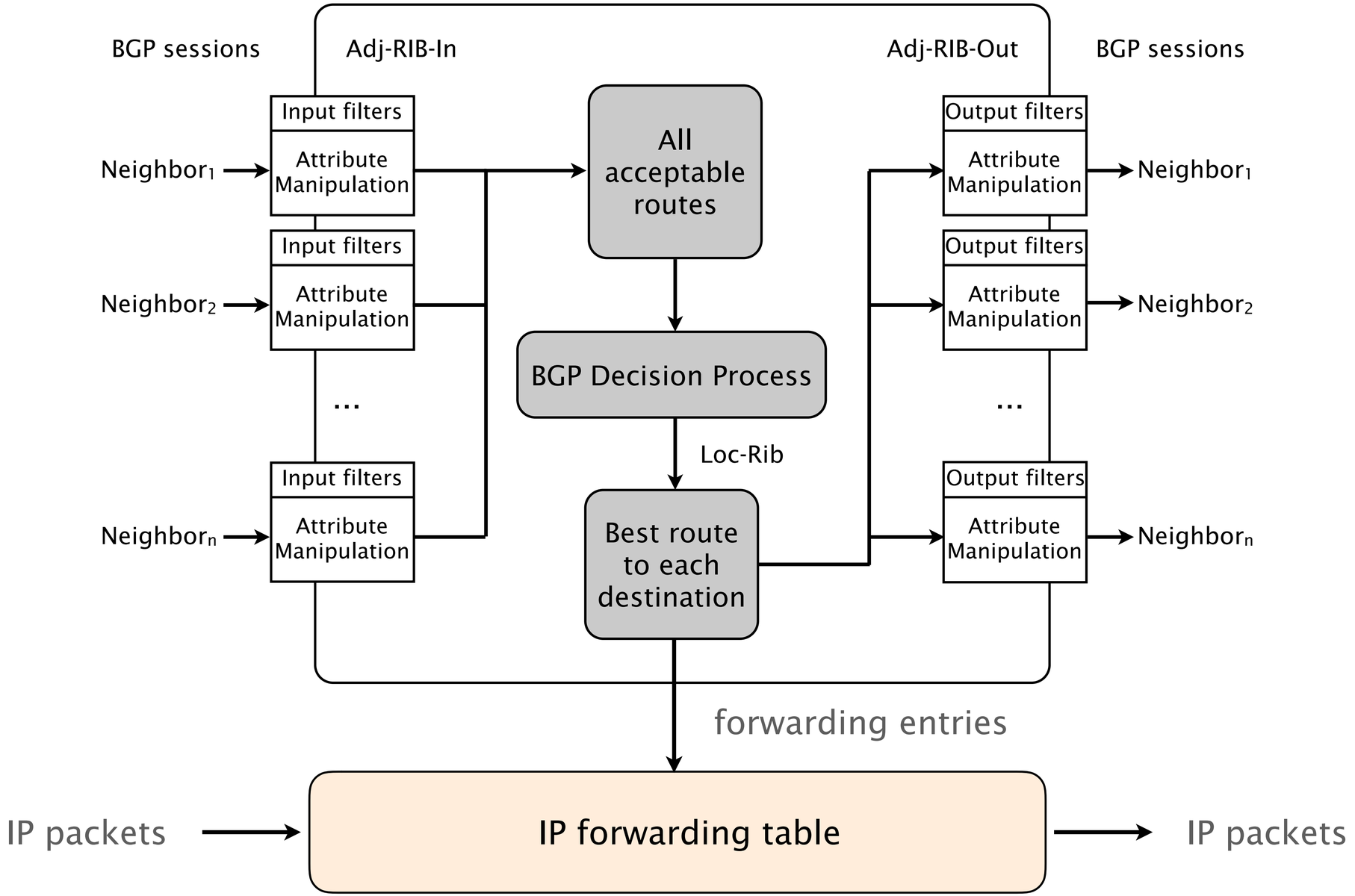
control where traffic is going

Export

for each selected path:

to which neighbors propagate it?

control where traffic is coming from


BGP sessions
Adj-RIB-In
Adj-RIB-Out
BGP sessions
Input filters
Attribute Manipulation
Neighbor1
Input filters
Attribute Manipulation
Neighbor2
...
Input filters
Attribute Manipulation
Neighborn
All acceptable routes
BGP Decision Process
Loc-Rib
Best route to each destination
Output filters
Attribute Manipulation
Neighbor1
Output filters
Attribute Manipulation
Neighbor2
...
Output filters
Attribute Manipulation
Neighborn
forwarding entries
IP packets
IP forwarding table
IP packets


BGP sessions
Adj-RIB-In
Adj-RIB-Out
BGP sessions
Input filters
Attribute Manipulation
Neighbor1
Input filters
Attribute Manipulation
Neighbor2
...
Input filters
Attribute Manipulation
Neighborn
All acceptable routes
fixed, well-known
BGP Decision Process
Loc-Rib
Best route to each destination
Output filters
Attribute Manipulation
Neighbor1
Output filters
Attribute Manipulation
Neighbor2
...
Output filters
Attribute Manipulation
Neighborn
forwarding entries
IP packets
IP forwarding table
IP packets

Prefer routes…

with higher preference

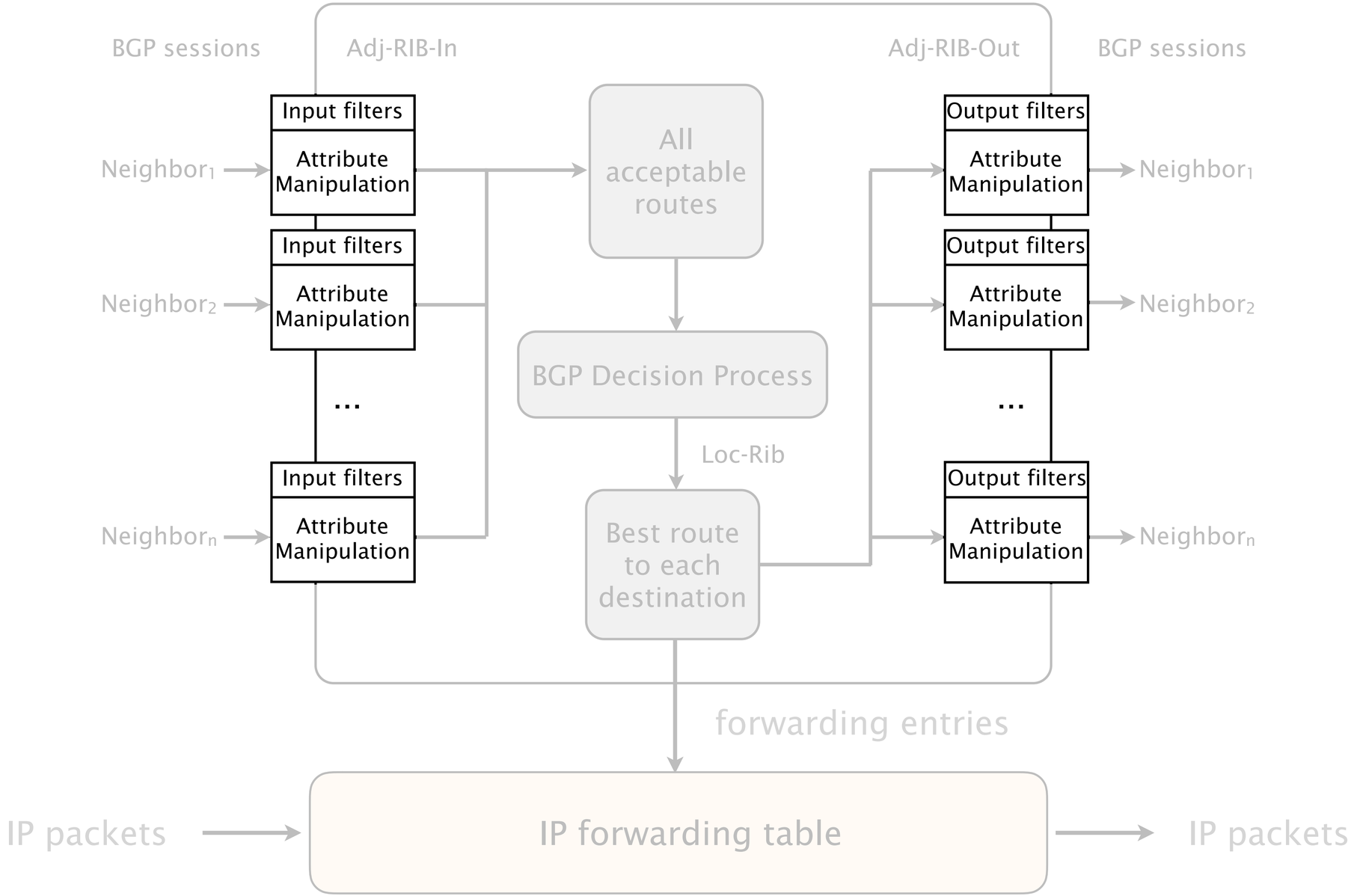
with shorter path length

…

learned externally rather than internally

whose egress point is the closest

with smaller egress IP address (tie-break)


BGP sessions
Adj-RIB-In
Adj-RIB-Out
BGP sessions
Input filters
Attribute Manipulation
Neighbor1
Input filters
Attribute Manipulation
Neighbor2
...
Input filters
Attribute Manipulation
Neighborn
All acceptable routes
BGP Decision Process
Loc-Rib
Best route to each destination
Output filters
Attribute Manipulation
Neighbor1
Output filters
Attribute Manipulation
Neighbor2
...
Output filters
Attribute Manipulation
Neighborn
forwarding entries
IP packets
IP forwarding table
IP packets

Network operators adapt how a router selects and exports BGP advertisements by configuring inbound/outbound filters

commonly known as BGP policies

Network operators adapt how a router selects and exports BGP advertisements by configuring inbound/outbound filters

BGP filter

$$f : Adv \rightarrow (Adv \cup \perp)$$

Network operators adapt how a router selects and exports BGP advertisements by configuring inbound/outbound filters

BGP filter

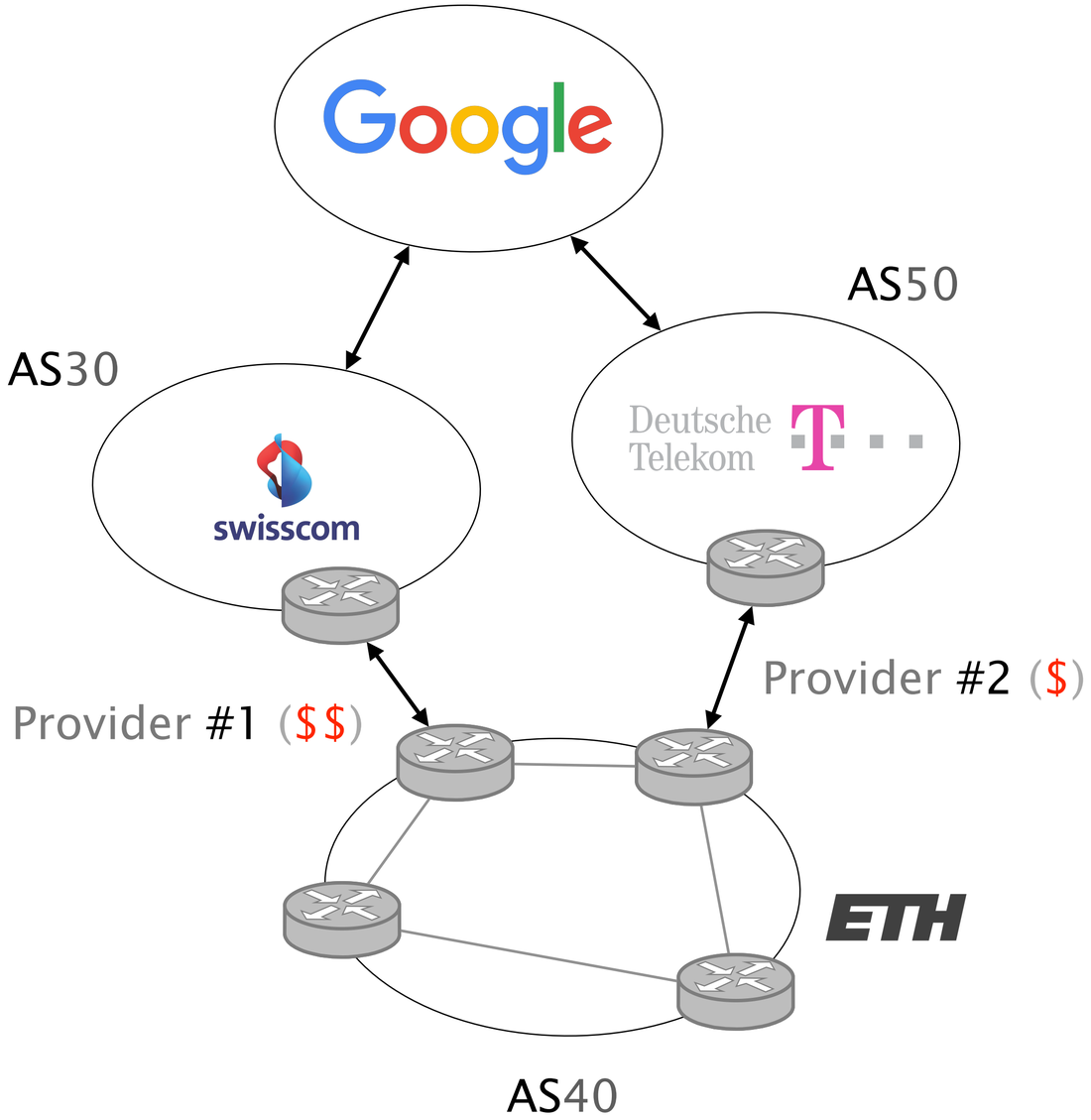
$f : Adv \rightarrow (Adv \cup \perp)$

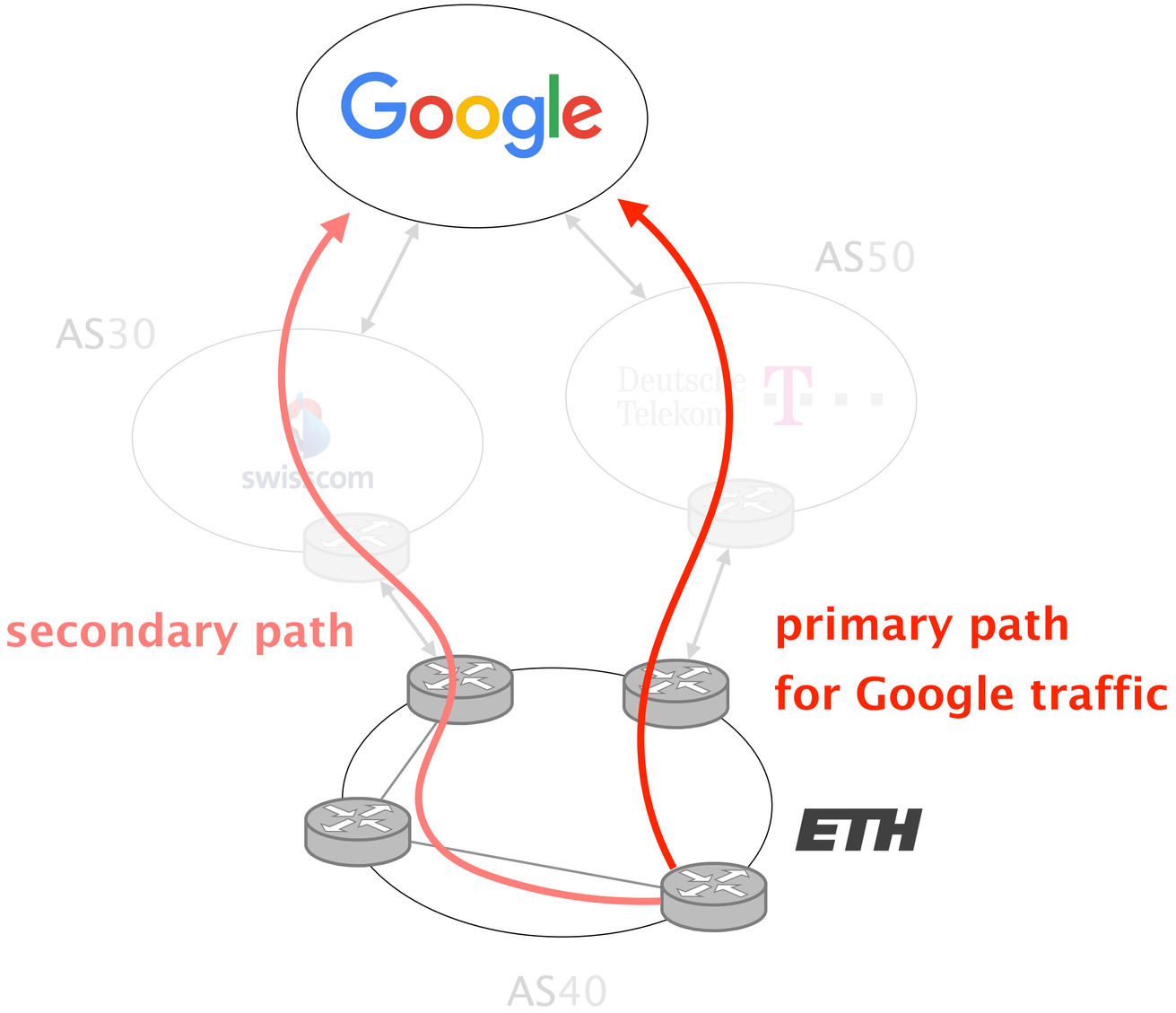
predicate

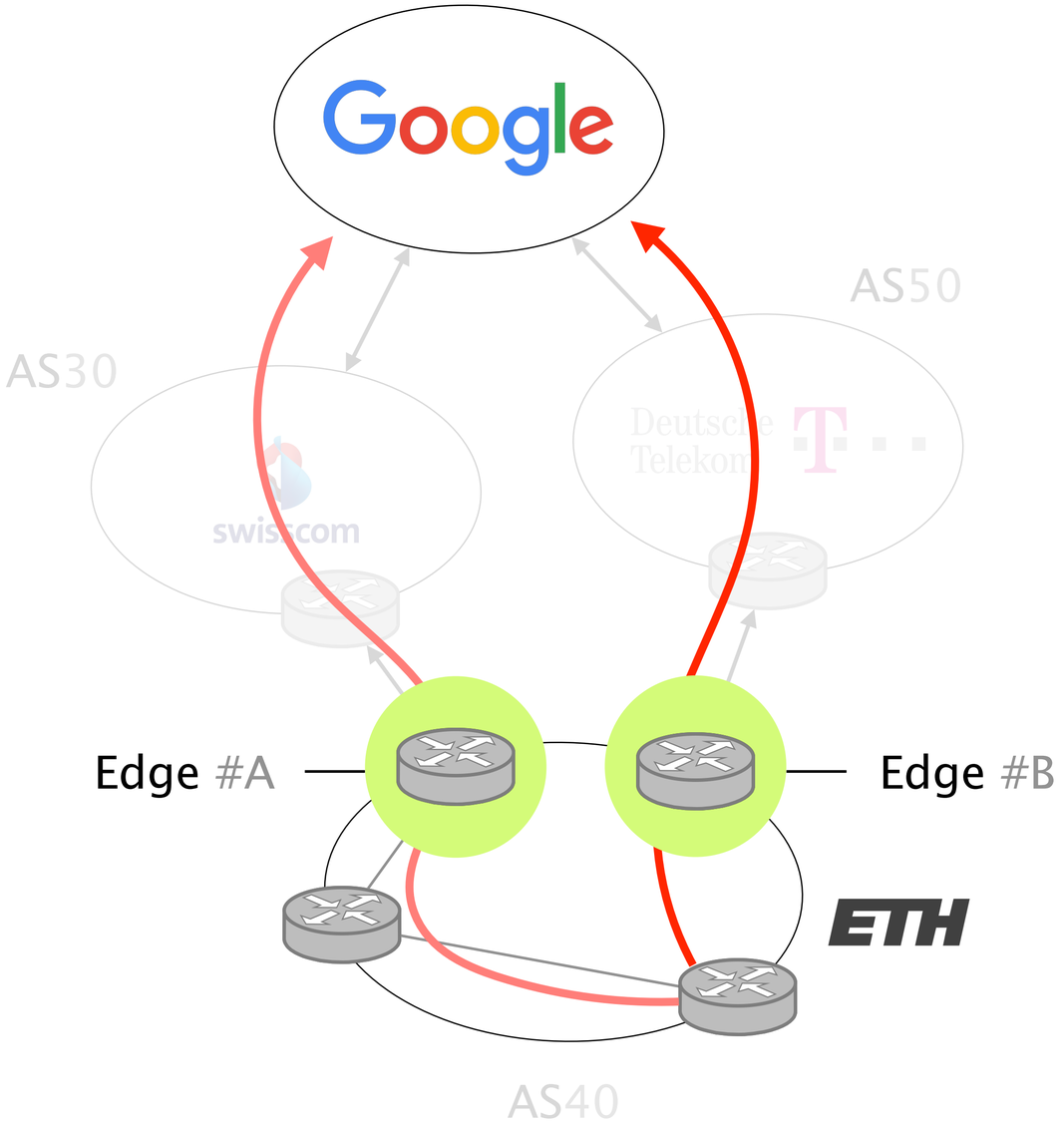
- prefix from Google
- path matches a regular expression
- label contains X
- path received from AS X
- …

action

- set preference X
- attach/strip label Y
- drop
- …

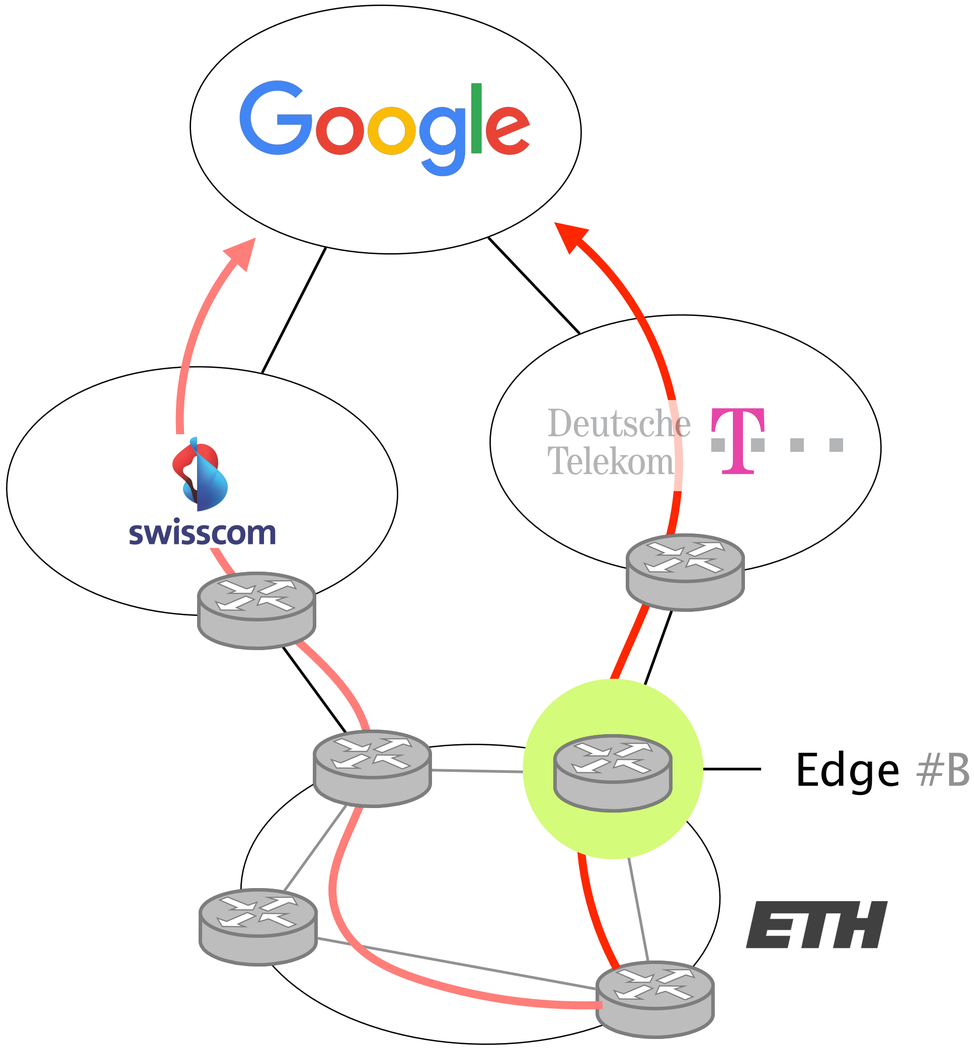

Google
AS30
AS50
swisscom
Deutsche Telekom
Provider #1 ($$)
Provider #2 ($)
ETH
AS40


Google
AS30
AS50
swisscom
Deutsche Telekom
secondary path
primary path
for Google traffic
AS40


Google
AS30
AS50
swisscom
Deutsche Telekom
Edge #A
Edge #B
ETH
AS40

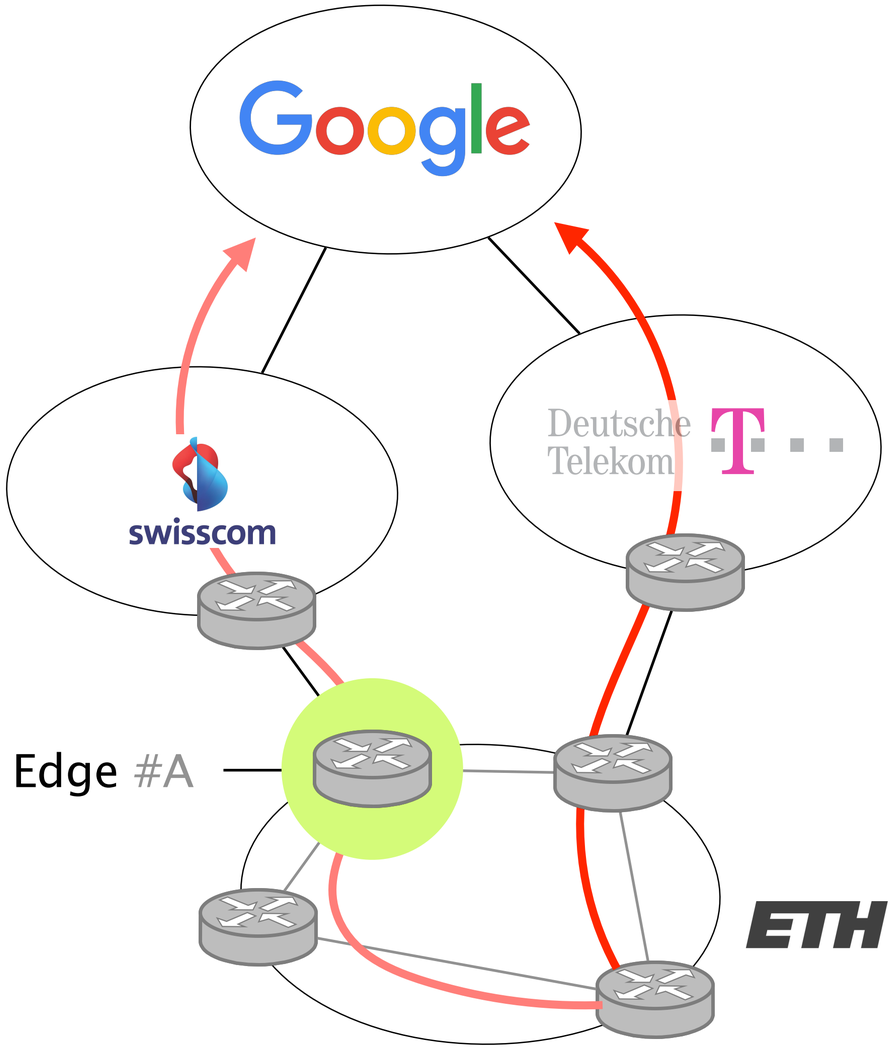
Edge #B configuration

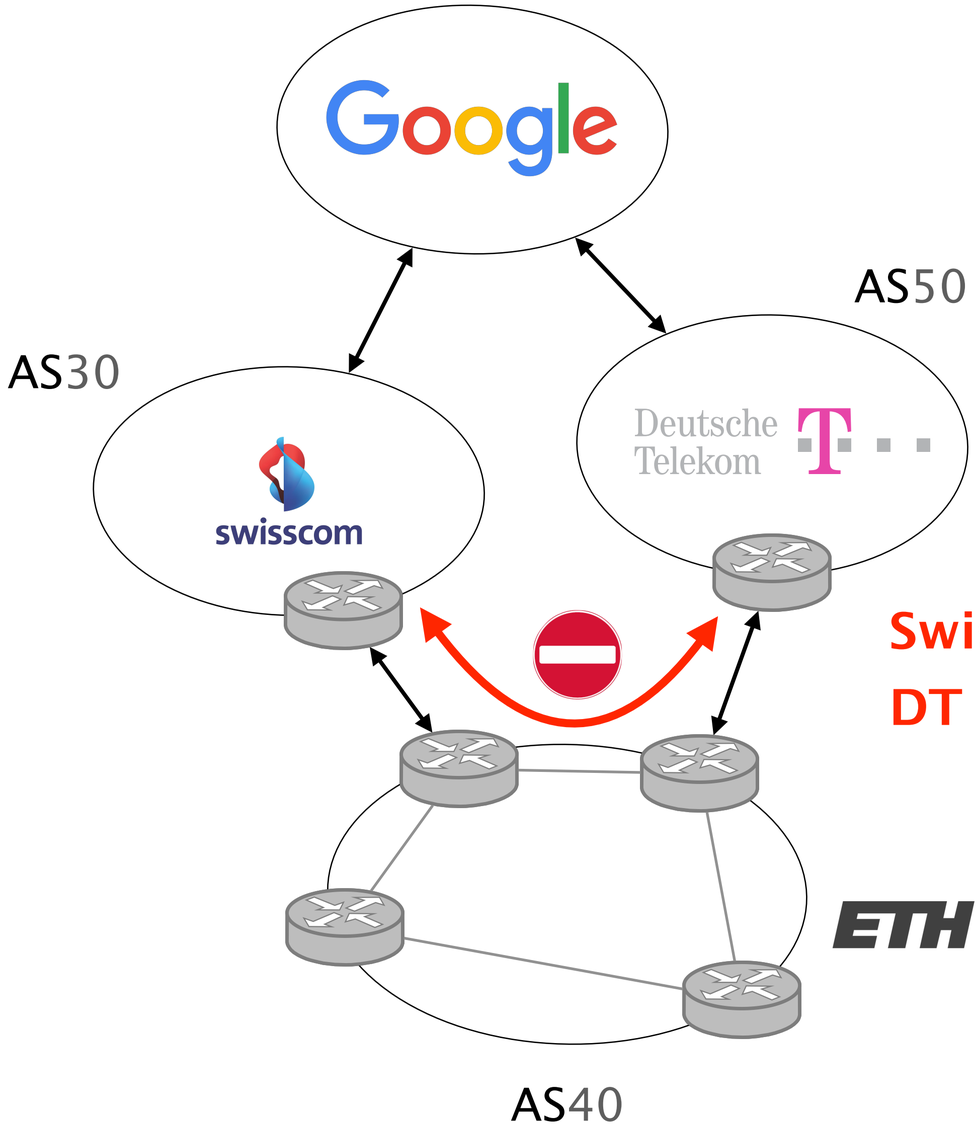
```
router bgp 10
  ...
  neighbor AS50 in_filter in_dt
  neighbor AS50 out_filter out_dt
  ...
route-map in_dt
  set preference 100
```



Edge #A configuration

```
router bgp 10
  ...
  neighbor AS30 in_filter in_swiss
  neighbor AS30 out_filter out_swiss
  ...
route-map in_swiss
  set preference 50
```



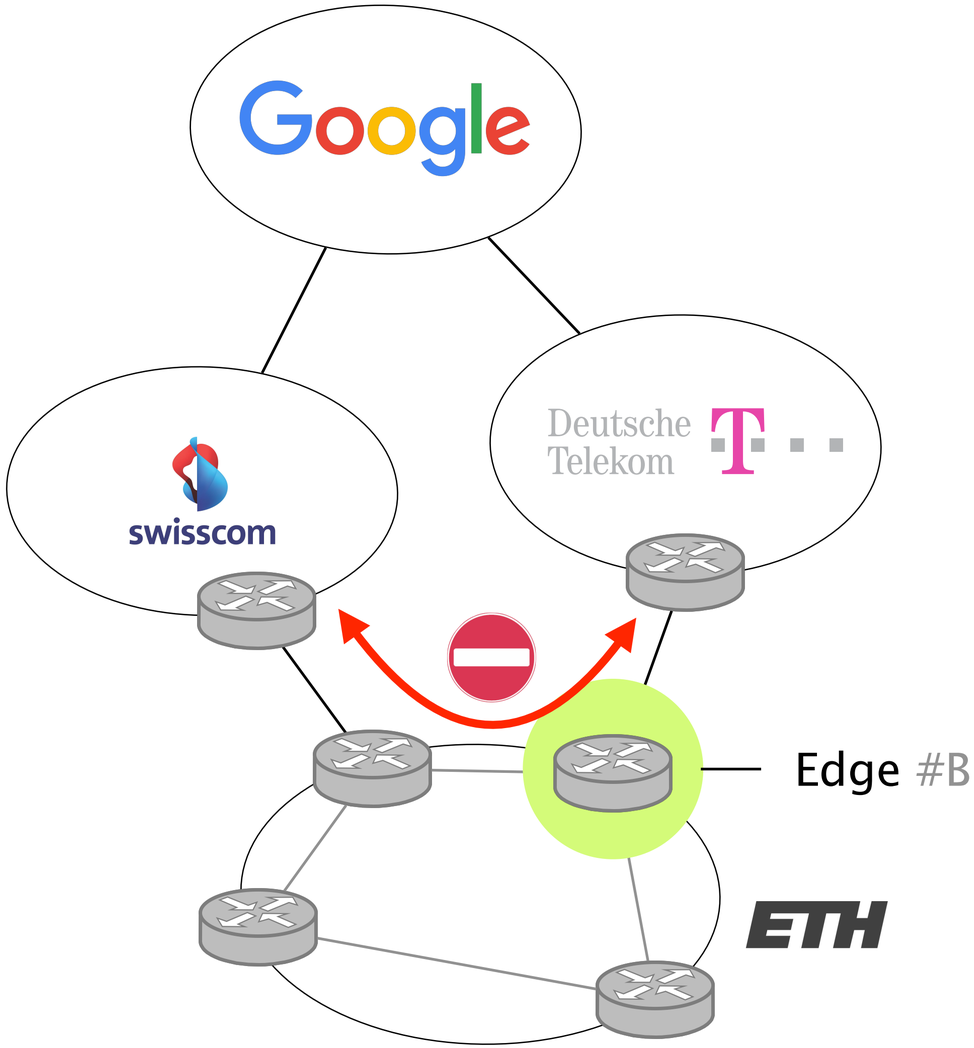


Swisscom shouldn't reach

DT *via* ETH (and vice-versa)

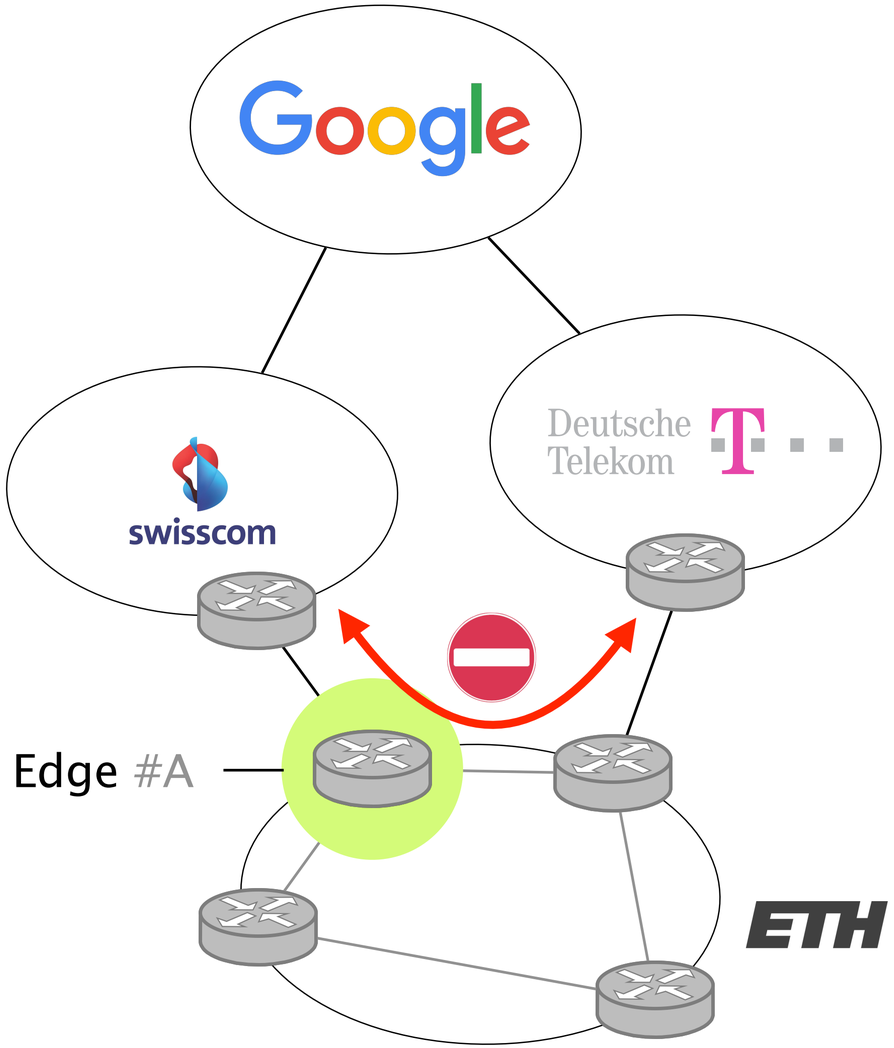
Edge #B configuration

```
router bgp 10
  ...
  neighbor AS50 in_filter in_dt
  neighbor AS50 out_filter out_dt
  ...
route-map in_dt
  set preference 100
  set label PROVIDER
  ...
route-map out_dt
  if(label PROVIDER): drop;
  else allow;
```



Edge #A configuration

```
router bgp 10
  ...
  neighbor AS30 in_filter in_swiss
  neighbor AS30 out_filter out_swiss
  ...
route-map in_swiss
  set preference 50
  set label PROVIDER
  ...
route-map out_swiss
  if(label PROVIDER): drop;
  else allow;
```



But first…

"How to configure routing protocols" 101

inter-domain routing

intra-domain routing

OSPF

In OSPF, routers build a precise map of the network by flooding its local view to everyone

Each router keeps track of its incident links and cost

as well as whether they are up or down

Each router broadcasts its own link state

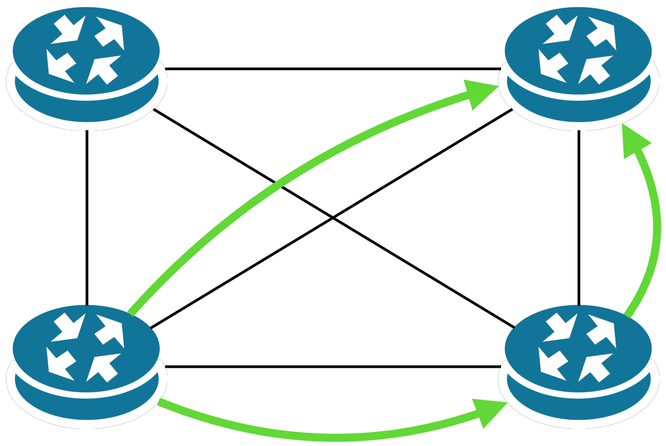
to give every router a complete view of the graph

Routers run Dijkstra on the corresponding graph

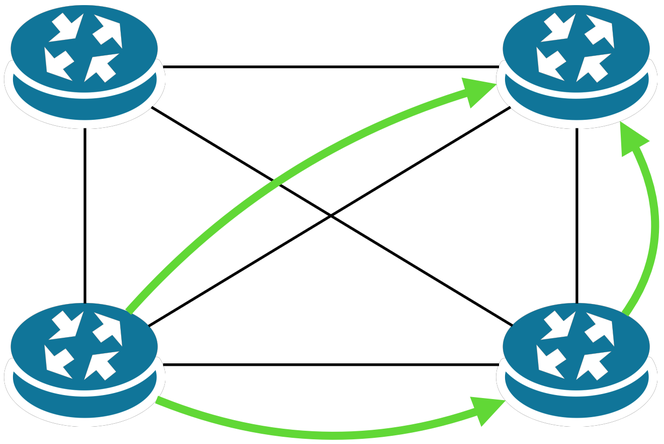
to compute their shortest-paths and forwarding tables

OSPF configuration mainly consists in figuring out link weights inducing an intended network-wide forwarding state

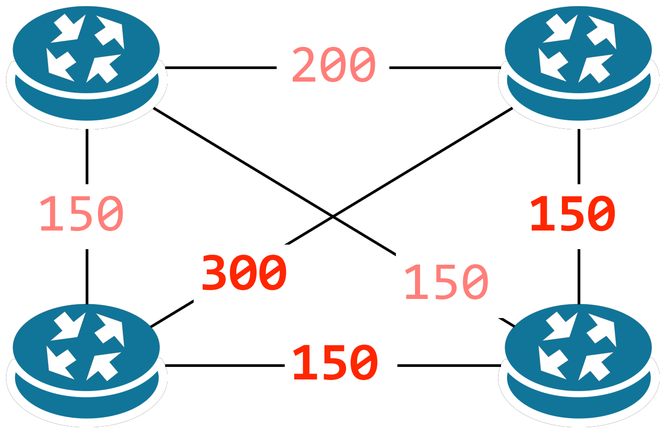
intended forwarding state



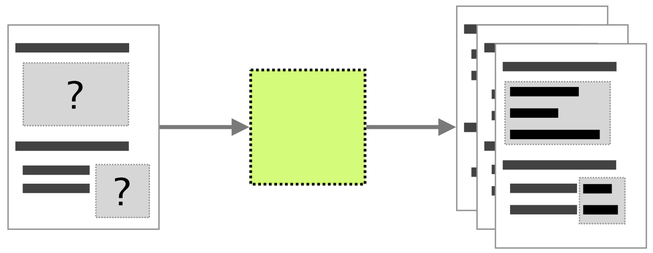
intended forwarding state



OSPF configuration

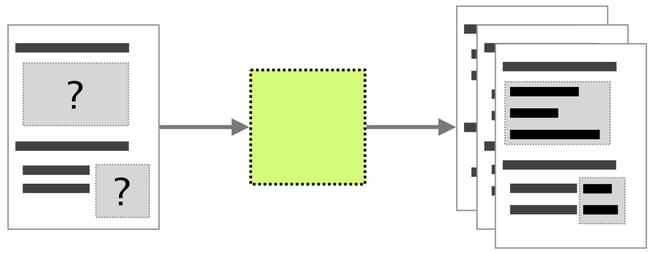


NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion



1. **BGP synthesis**
   optimized encoding
2. **OSPF synthesis**
   counter-examples-based
3. **Evaluation**
   flexible, *yet* scalable

NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion



1 **BGP synthesis**

optimized encoding

OSPF synthesis

counter-examples-based

Evaluation

flexible, *yet* scalable

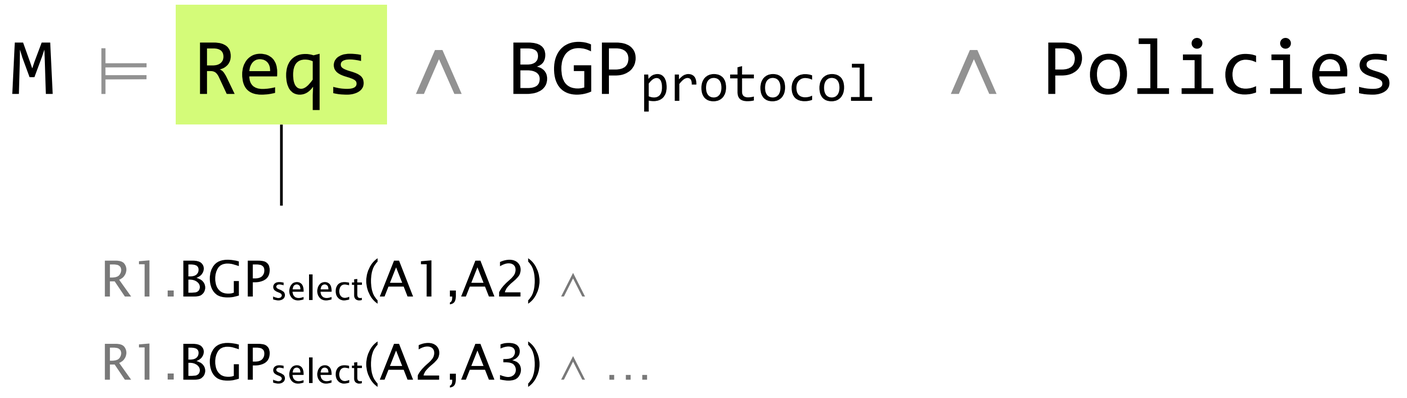
NetComplete autocompletes router-level BGP policies by encoding the desired BGP behavior as a logical formula

$$M \models Reqs \land BGP_{protocol} \land Policies$$

$$M \models \texttt{Reqs} \land \texttt{BGP}_{\texttt{protocol}} \land \texttt{Policies}$$

how should the network forward traffic

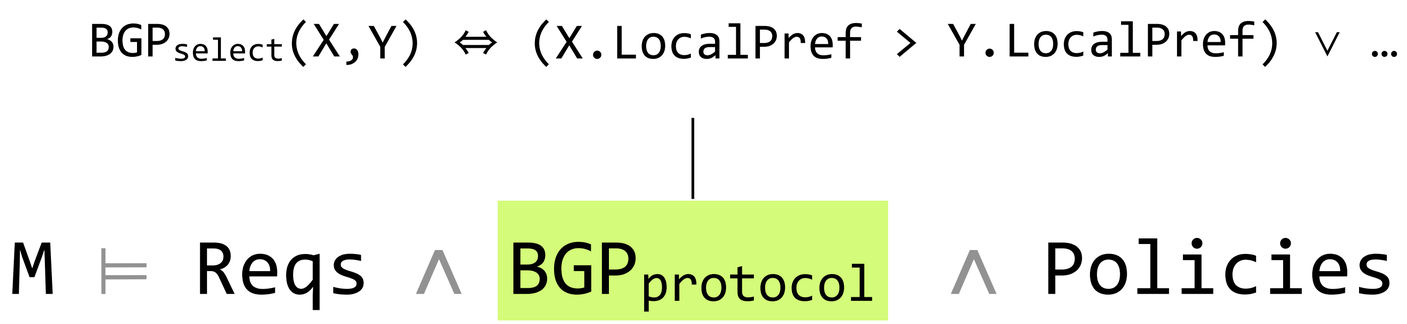
concrete, part of the input


M ⊨ Reqs ∧ BGP_protocol ∧ Policies
R1.BGP_select(A1,A2) ∧
R1.BGP_select(A2,A3) ∧ ...

concrete, protocol semantic

how do BGP routers select routes

$$M \models Reqs \wedge BGP_{protocol} \wedge Policies$$


$BGP_{select}(X,Y) \Leftrightarrow (X.LocalPref > Y.LocalPref) \vee \ldots$
$M \models Reqs \wedge BGP_{protocol} \wedge Policies$

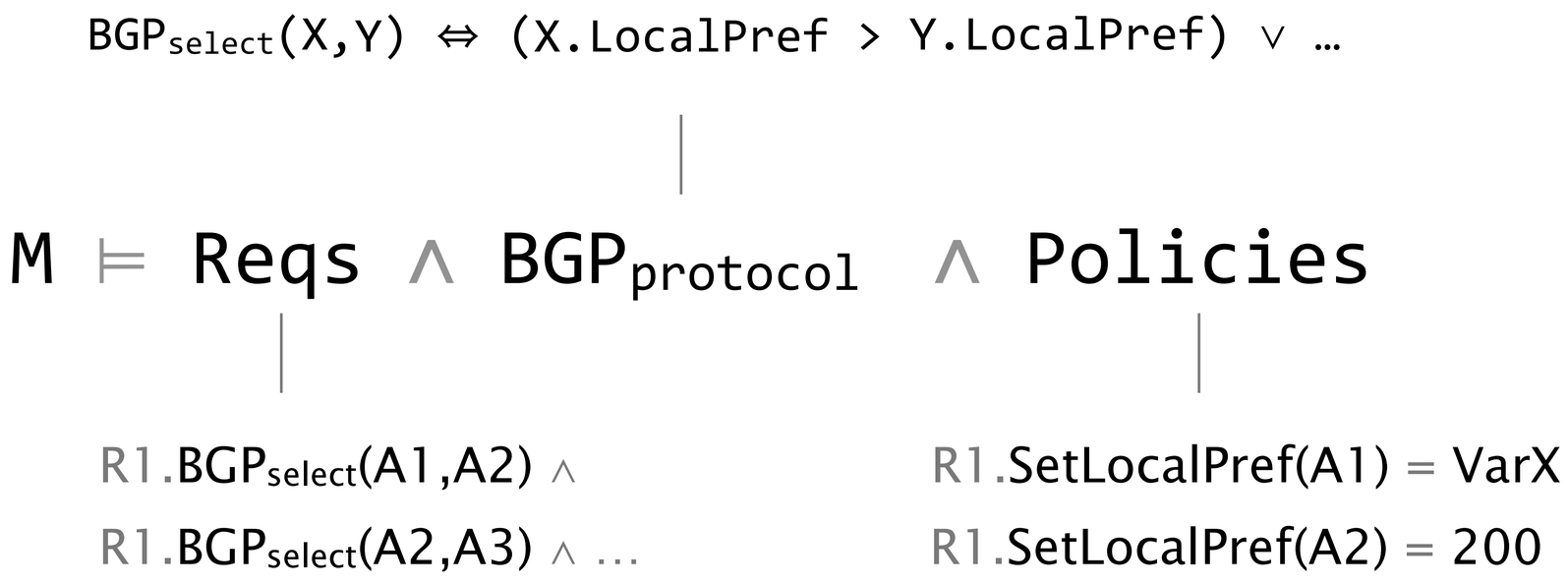
$$M \models Reqs \wedge BGP_{protocol} \wedge Policies$$

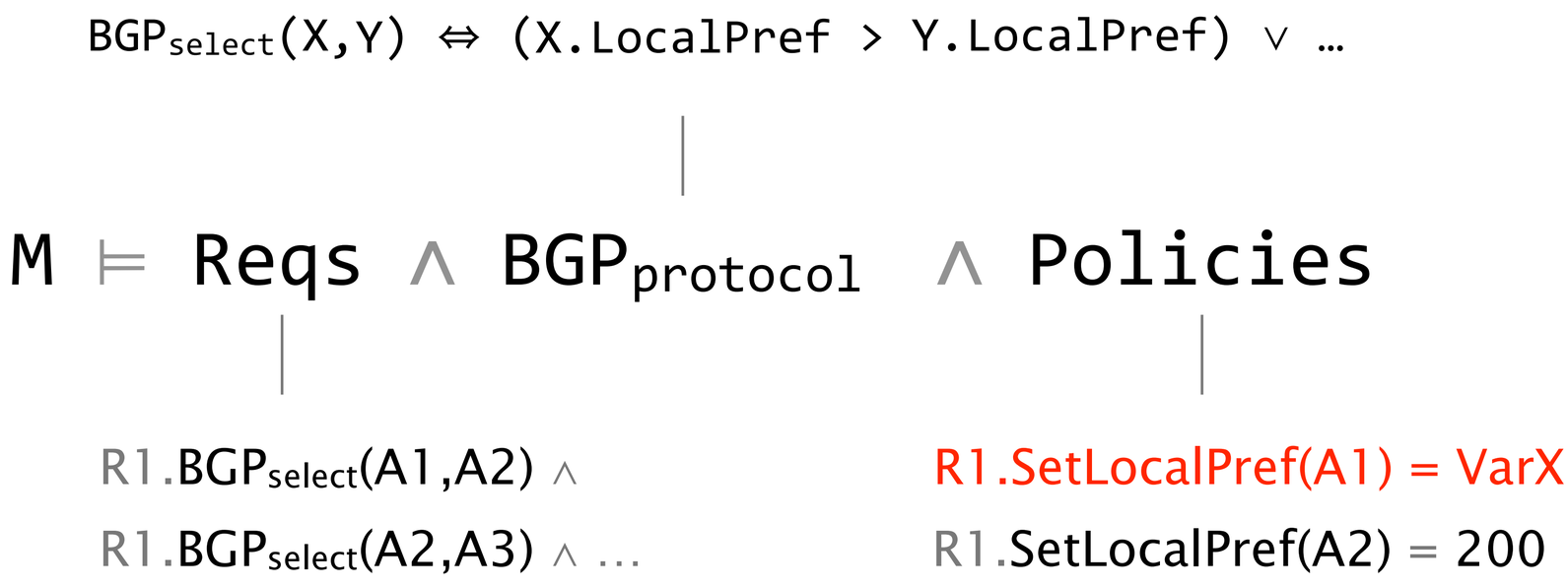
how routes should be modified

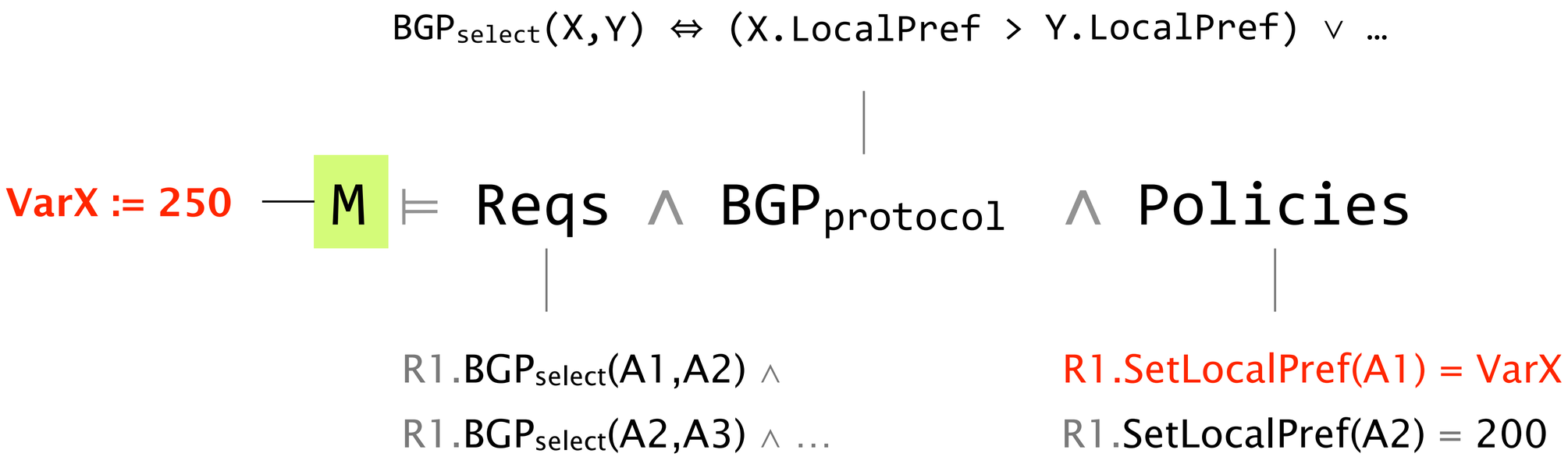
symbolic, to be found


M ⊨ Reqs ∧ BGP protocol ∧ Policies
R1.SetLocalPref(A1) = VarX
R1.SetLocalPref(A2) = 200

Solving this logical formula consists in assigning each symbolic variable with a concrete value

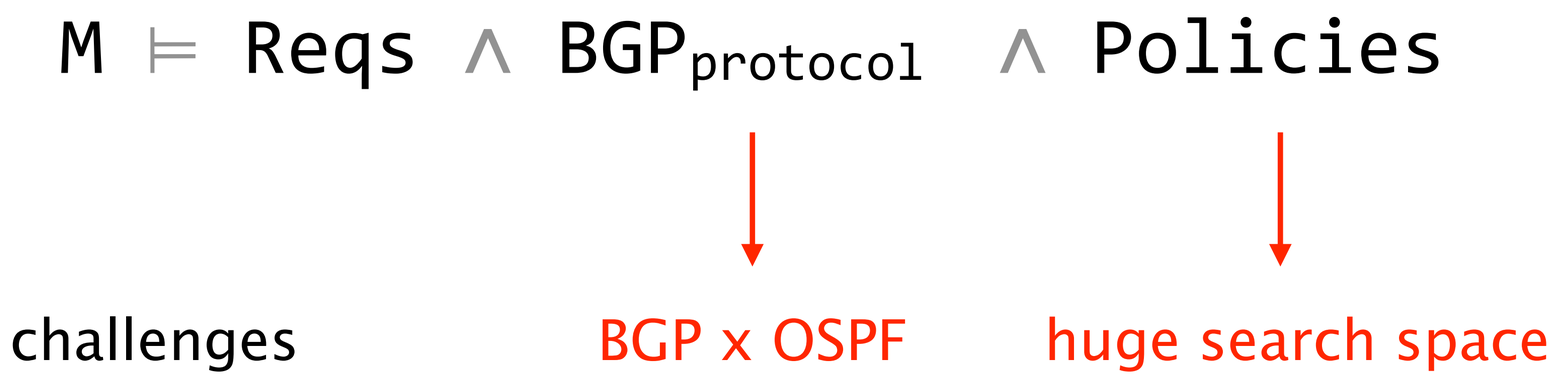



BGP_select(X,Y) ⇔ (X.LocalPref > Y.LocalPref) ∨ …
M ⊨ Reqs ∧ BGP_protocol ∧ Policies
R1.BGP_select(A1,A2) ∧
R1.BGP_select(A2,A3) ∧ …
R1.SetLocalPref(A1) = VarX
R1.SetLocalPref(A2) = 200


$BGP_{select}(X,Y) \Leftrightarrow (X.LocalPref > Y.LocalPref) \vee \ldots$
VarX := 250
M
$\models$
Reqs
$\wedge$
$BGP_{protocol}$
$\wedge$
Policies
$R1.BGP_{select}(A1,A2) \wedge$
$R1.BGP_{select}(A2,A3) \wedge \ldots$
R1.SetLocalPref(A1) = VarX
R1.SetLocalPref(A2) = 200

Naive encodings lead to complex constraints
that cannot be solved in a reasonable time

Naive encodings lead to complex constraints
that cannot be solved in a reasonable time



Naive encodings lead to complex constraints
that cannot be solved in a reasonable time

$$M \models Reqs \wedge BGP_{protocol} \wedge Policies$$

challenges	BGP x OSPF	huge search space
solutions	iterative synthesis	partial evaluation

Naive encodings lead to complex constraints
that cannot be solved in a reasonable time

$$M \models Reqs \wedge BGP_{protocol} \wedge Policies$$

challenges	BGP x OSPF	huge search space
solutions	iterative synthesis	partial evaluation

NetComplete encodes reduced policies by relying on the requirements and the sketches

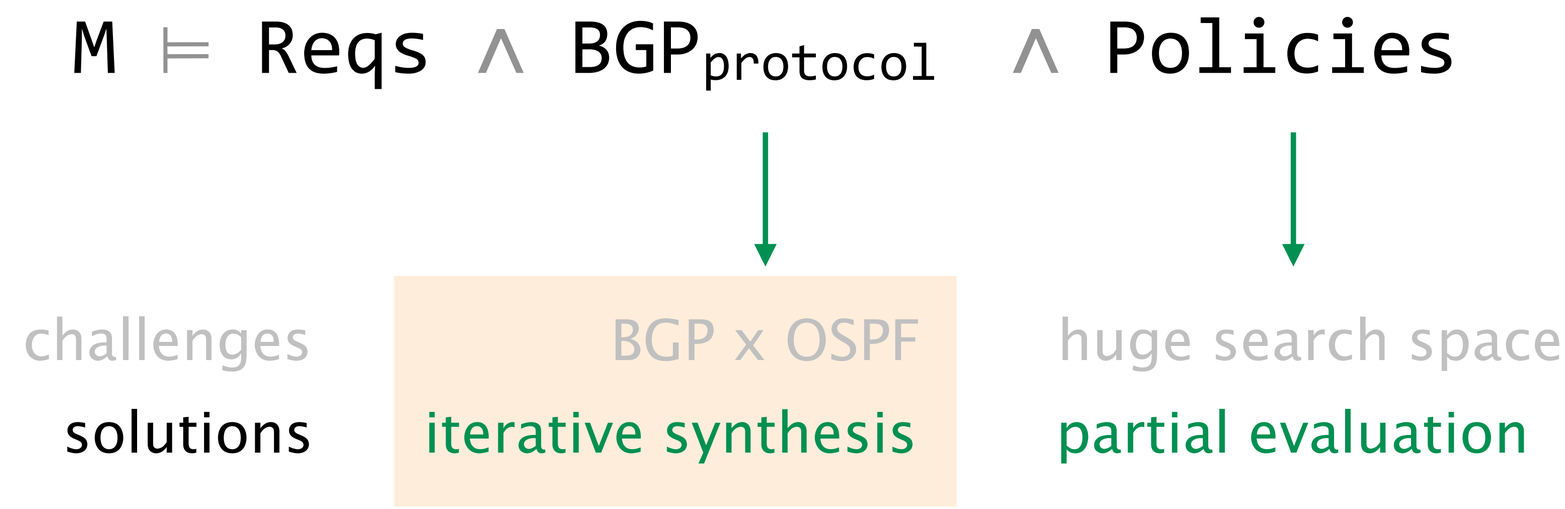
NetComplete encodes reduced policies by relying on the requirements and the sketches

Step 1	Capture how announcements should propagate using the requirements
Output	BGP propagation graph

NetComplete encodes reduced policies by relying on the requirements and the sketches

Step 1	Capture how announcements should propagate using the requirements
Output	BGP propagation graph
Step 2	Combine the graph with constraints imposed by sketches via symbolic execution
Output	partially evaluated formulas

Naive encodings lead to complex constraints
that cannot be solved in a reasonable time



$$M \models Reqs \wedge BGP_{protocol} \wedge Policies$$

BGP Decision Process

1. Higher local preference
2. Shorter AS Path
3. Lowest Origin
4. Lowest MED
5. eBGP over iBGP
6. Lower OSPF weight

BGP Decision Process

1. Higher local preference
2. Shorter AS Path
3. Lowest Origin
4. Lowest MED
5. eBGP over iBGP
6. Lower OSPF weight

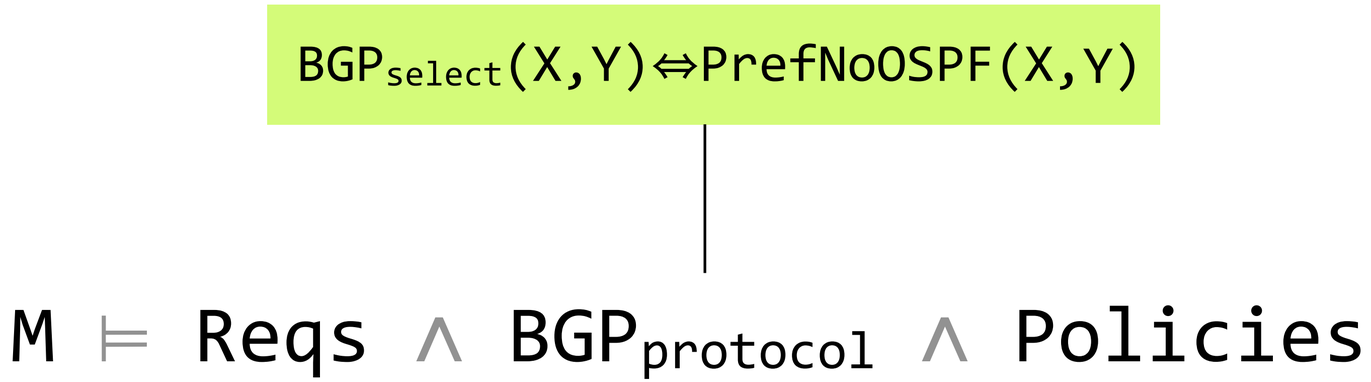
If we hit this step,
it means that the BGP decision depends on OSPF

NetComplete first tries to find a BGP-only assignment,
one in which the BGP behavior does not depend on OSPF

NetComplete first searches for a solution using solely Step 1 to 5

Decision Process		Constraints
1	Higher local preference	
2	Shorter AS Path	
3	Lowest Origin	$PrefNoOSPF(X,Y)$
4	Lowest MED	
5	eBGP over iBGP	
6	Lower OSPF weight	$PrefOSPF(X,Y) \Leftrightarrow \neg PrefNoOSPF(X,Y)$

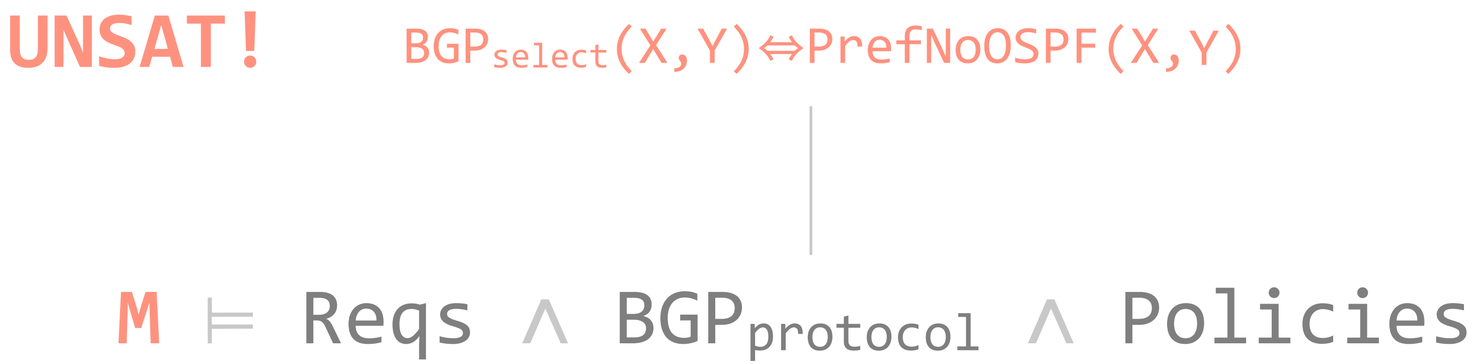
NetComplete first searches for a solution using solely Step 1 to 5

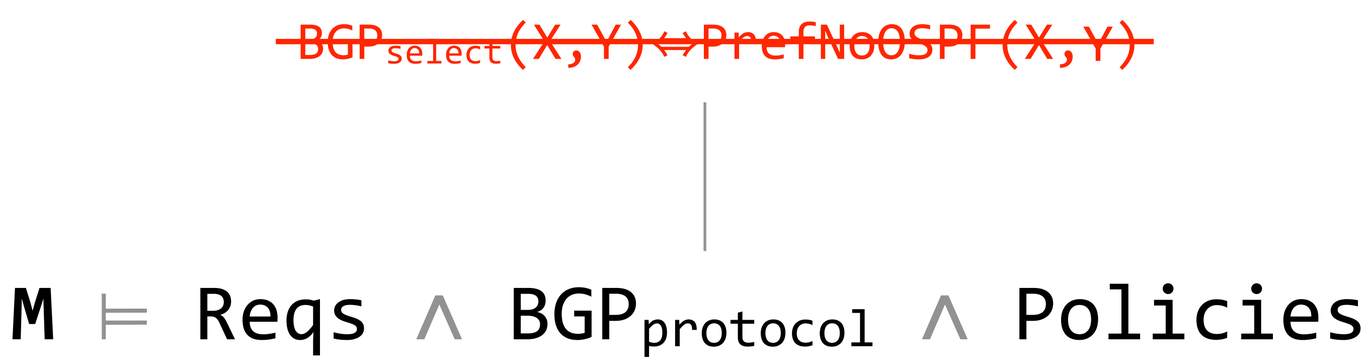


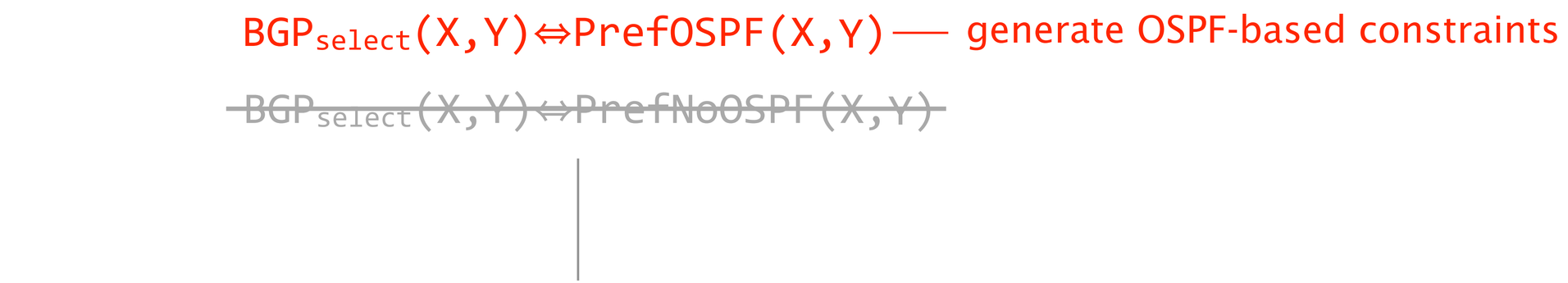
UNSAT! $BGP_{select}(X,Y) \Leftrightarrow PrefNoOSPF(X,Y)$

$$M \models Reqs \wedge BGP_{protocol} \wedge Policies$$

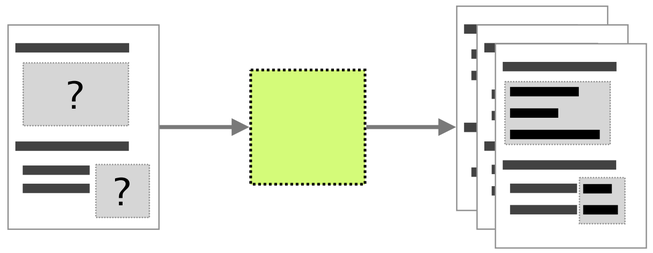
If NetComplete cannot find an assignment,
it then allows the BGP decisions to depend on OSPF




~~$BGP_{select}(X,Y) \Leftrightarrow PrefNoOSPF(X,Y)$~~
$\mathbf{M} \models Reqs \wedge BGP_{protocol} \wedge Policies$


$BGP_{select}(X,Y) \Leftrightarrow PrefOSPF(X,Y)$
generate OSPF-based constraints
$BGP_{select}(X,Y) \Leftrightarrow PrefNoOSPF(X,Y)$
$M \models Reqs \wedge BGP_{protocol} \wedge Policies$

NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion



BGP synthesis

optimized encoding

2 OSPF synthesis

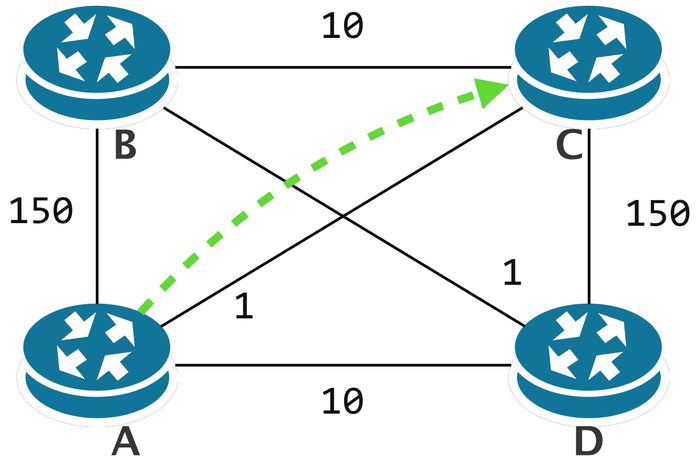
counter-examples-based

Evaluation

flexible, *yet* scalable

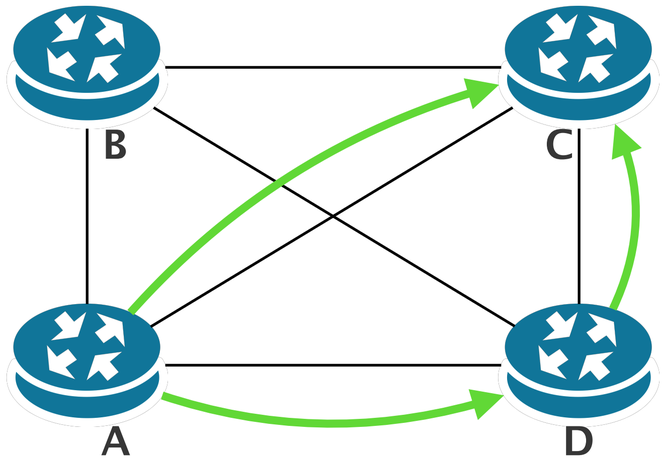
As for BGP, Netcomplete phrases the problem of finding weights as a constraint satisfaction problem

Consider this initial configuration in which (A,C) traffic is forwarded along the direct link

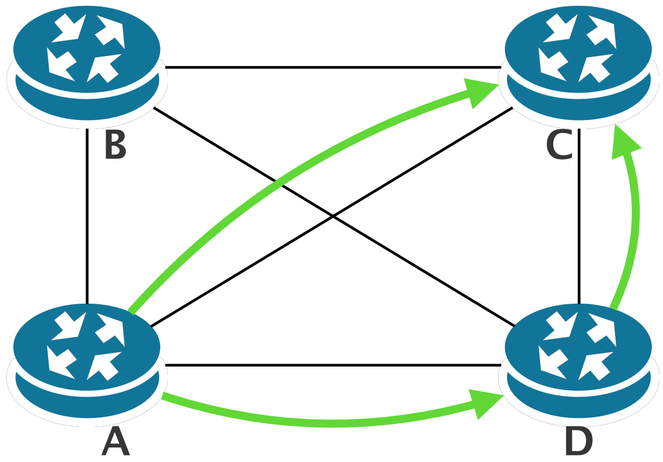


For performance reasons,

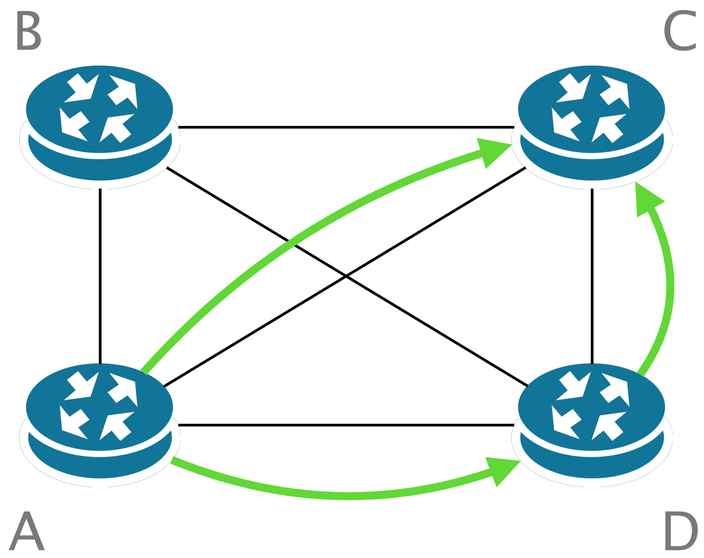
the operators want to enable load-balancing



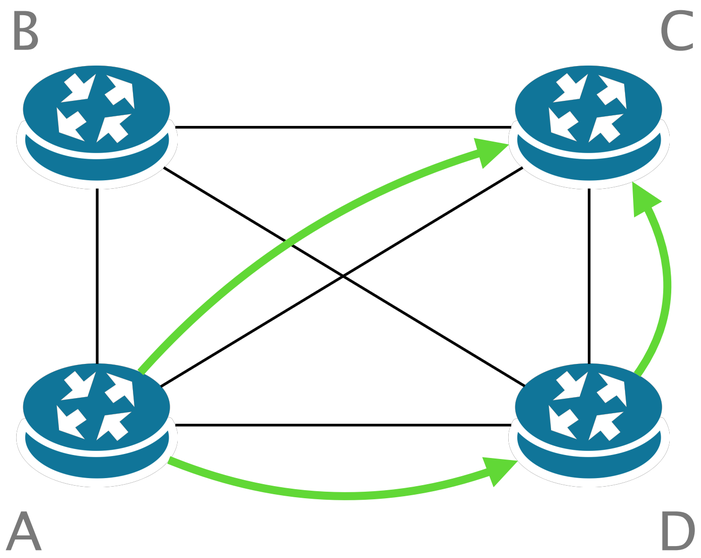
What should be the weights for this to happen?



input requirements

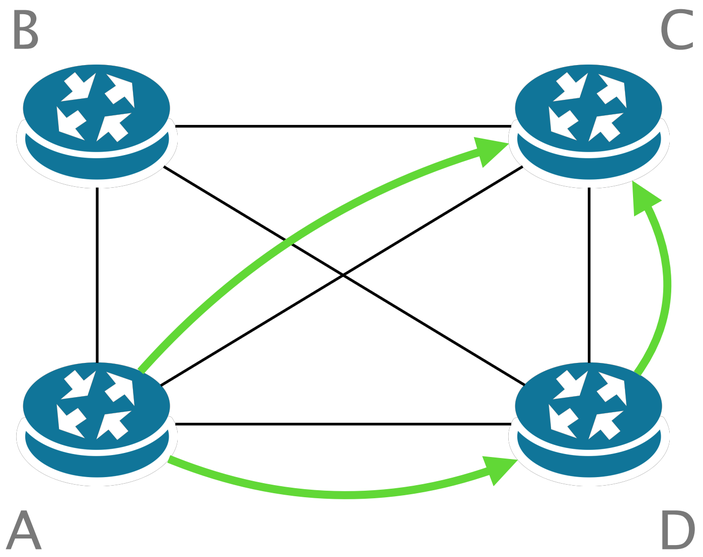


input requirements



synthesis procedure

input requirements



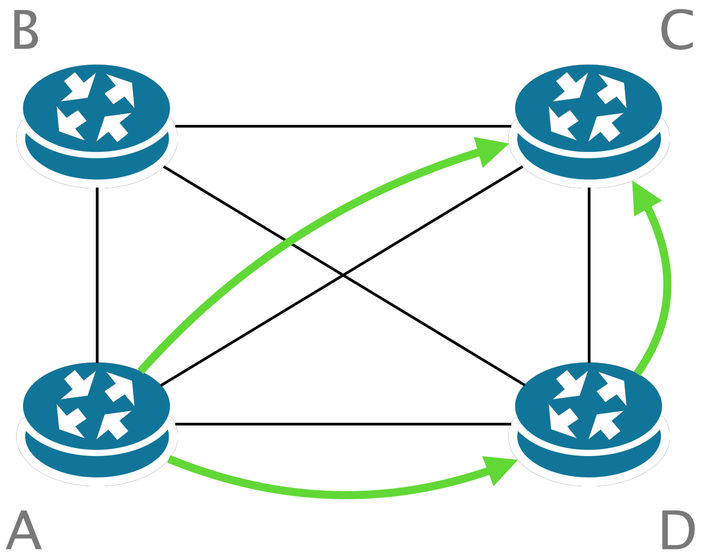
synthesis procedure

$$\forall X \in Paths(A,C) \setminus Reqs$$

$$\downarrow$$

$$Cost(A \rightarrow C) = Cost(A \rightarrow D \rightarrow C) < Cost(X)$$

input requirements



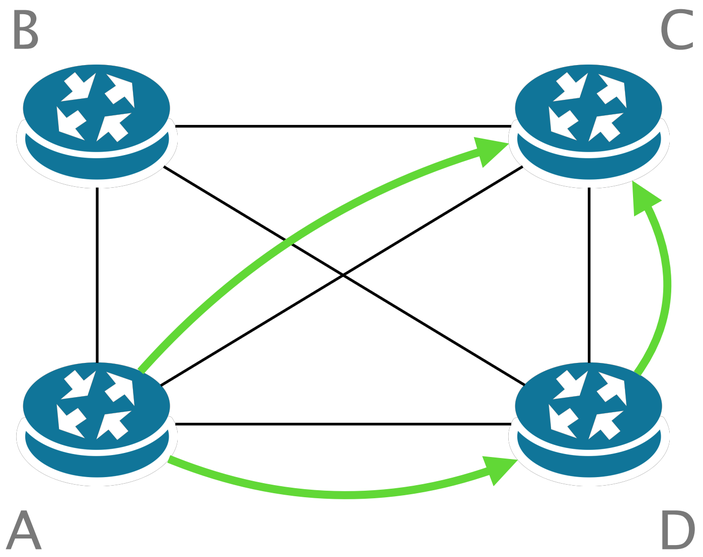
synthesis procedure

$\forall X \in Paths(A,C) \backslash Reqs$

$Cost(A \rightarrow C) = Cost(A \rightarrow D \rightarrow C) < Cost(X)$

Solve

input requirements



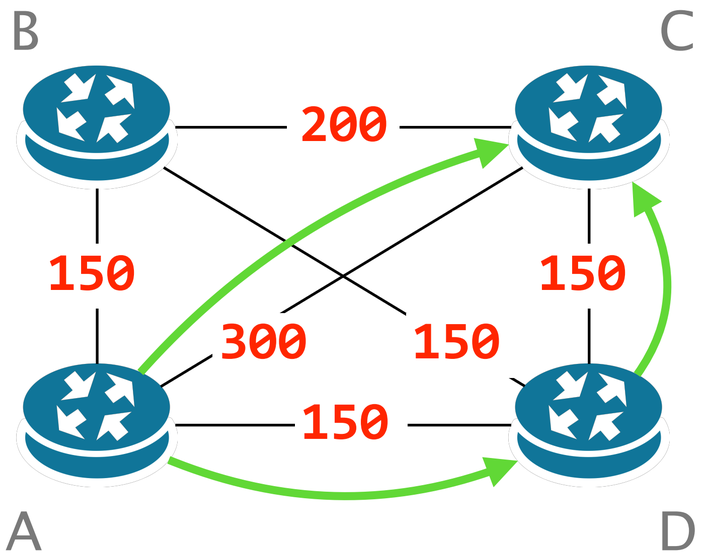
synthesis procedure

$\forall X \in Paths(A,C) \setminus Reqs$

$Cost(A \rightarrow C) = Cost(A \rightarrow D \rightarrow C) < Cost(X)$

Solve

input requirements



Synthesized weights

synthesis procedure

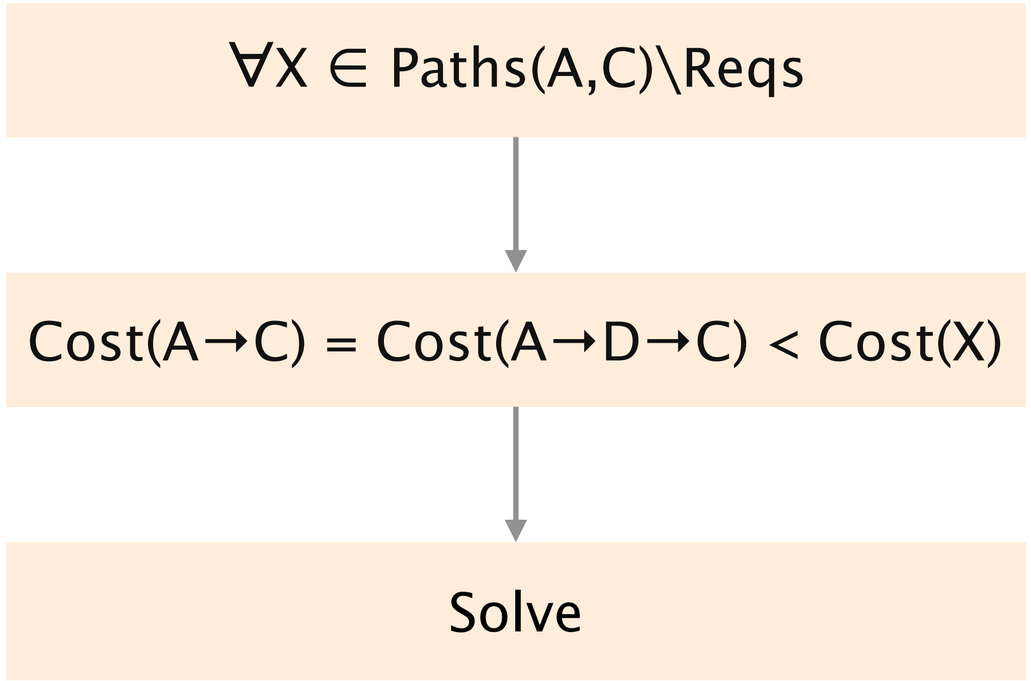
$\forall X \in Paths(A,C) \backslash Reqs$

$Cost(A \rightarrow C) = Cost(A \rightarrow D \rightarrow C) < Cost(X)$

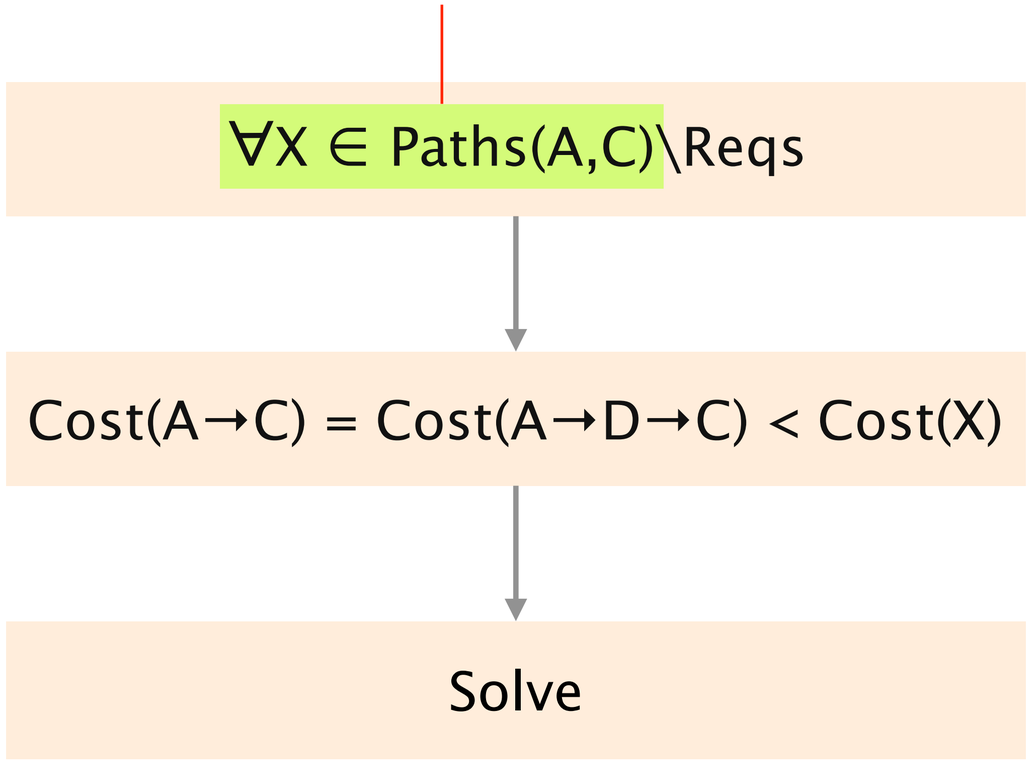
Solve

This was easy, but…

it does **not** scale



There can be an exponential number of paths between A and C…



To scale, NetComplete leverages

Counter-Example Guided Inductive Synthesis (CEGIS)

An contemporary approach to synthesis where

a solution is iteratively learned from counter-examples

While enumerating all paths is hard,

computing shortest paths given weights is easy!

Instead of considering all paths between X and Y

Instead of considering all paths between X and Y

CEGIS Part 1

Consider a random subset S of them and synthesize the weights considering S only

Instead of considering all paths between X and Y

CEGIS Part 1

Consider a random subset S of them and synthesize the weights considering S only

intuition

Fast as S is small compared to all paths

Instead of considering all paths between X and Y

CEGIS Part 1

Consider a random subset S of them and synthesize the weights considering S only

intuition

Fast as S is small compared to all paths

but can be wrong

Instead of considering all paths between X and Y

CEGIS Part 1

Consider a random subset S of them and synthesize the weights considering S only

CEGIS Part 2

Check whether the weights found comply with the requirements over all paths

If so return

Else take a counter-example (a path) that violates the Req and add it to S

Repeat.

Instead of considering all paths between X and Y

CEGIS Part 1

Consider a random subset S of them and synthesize the weights considering S only

CEGIS Part 2

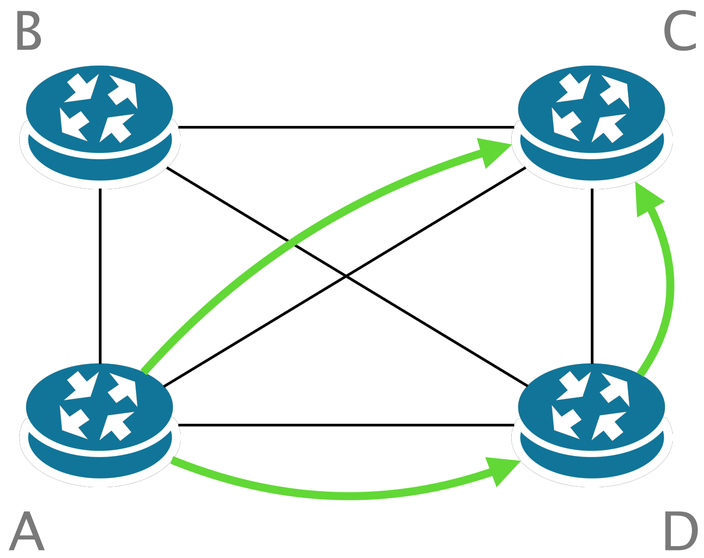
Check whether the weights found comply with the requirements **over all paths**

intuition

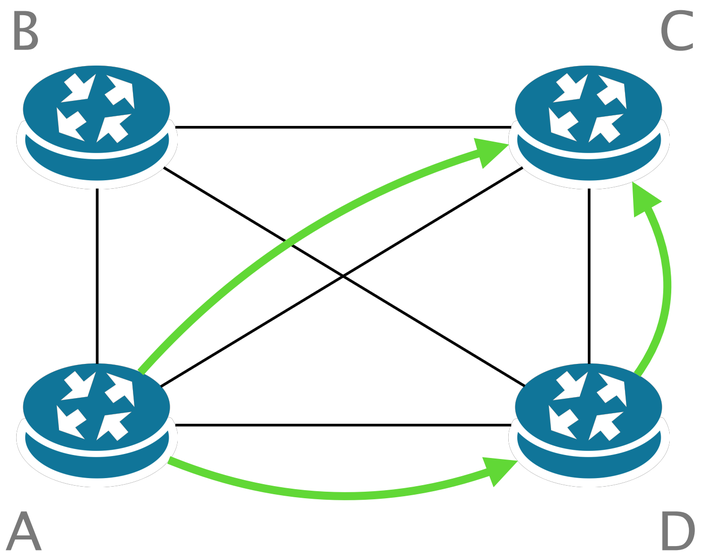
Fast too

simple shortest-path computation

input requirements

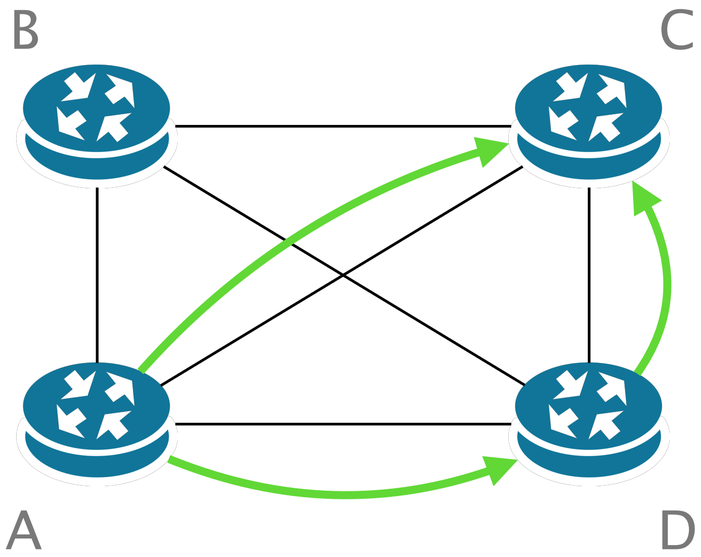


input requirements



synthesis procedure

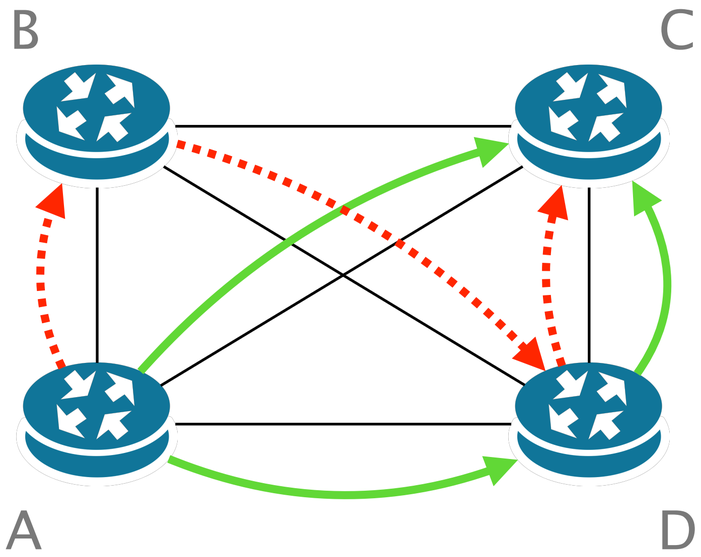
input requirements



synthesis procedure

$\forall X \in \text{SamplePaths}(A,C) \setminus \text{Reqs}$

input requirements



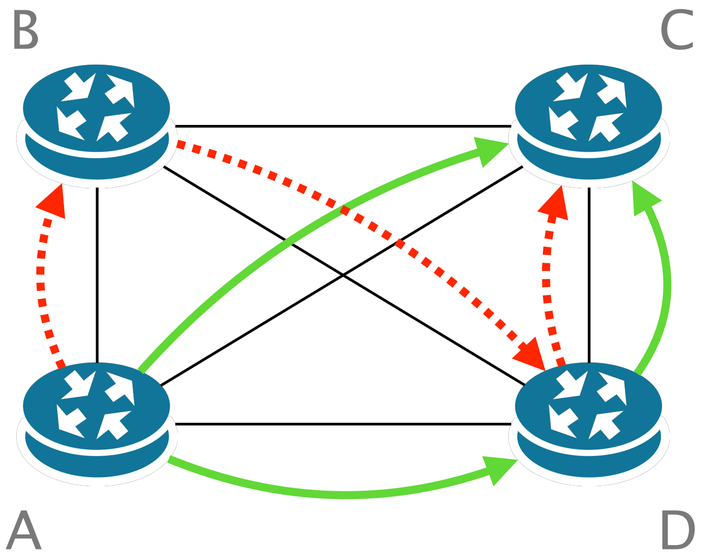
synthesis procedure

$\forall X \in \text{SamplePaths}(A,C) \backslash \text{Reqs}$

↓

Sample: { [A,B,D,C] }

input requirements



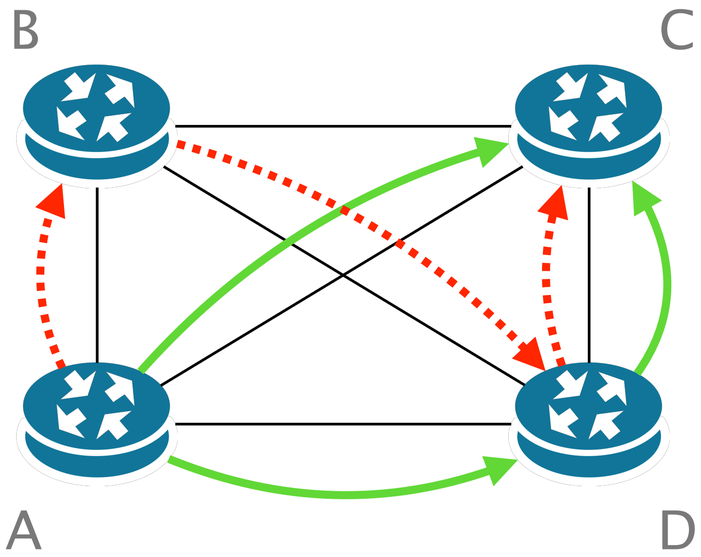
synthesis procedure

$$\forall X \in \text{SamplePaths}(A,C) \setminus \text{Reqs}$$

$$\downarrow$$

$$\text{Cost}(A \rightarrow C) = \text{Cost}(A \rightarrow D \rightarrow C) < \text{Cost}(X)$$

input requirements



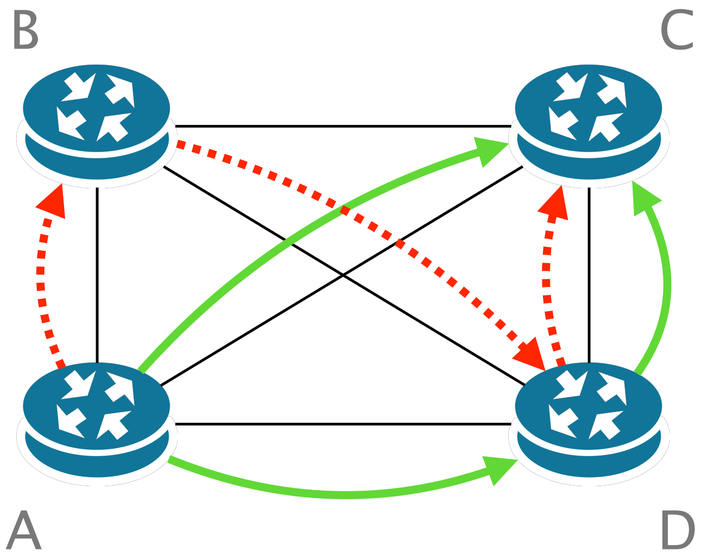
synthesis procedure

$\forall X \in \text{SamplePaths}(A,C) \setminus \text{Reqs}$

$\text{Cost}(A \rightarrow C) = \text{Cost}(A \rightarrow D \rightarrow C) < \text{Cost}(X)$

Solve

input requirements



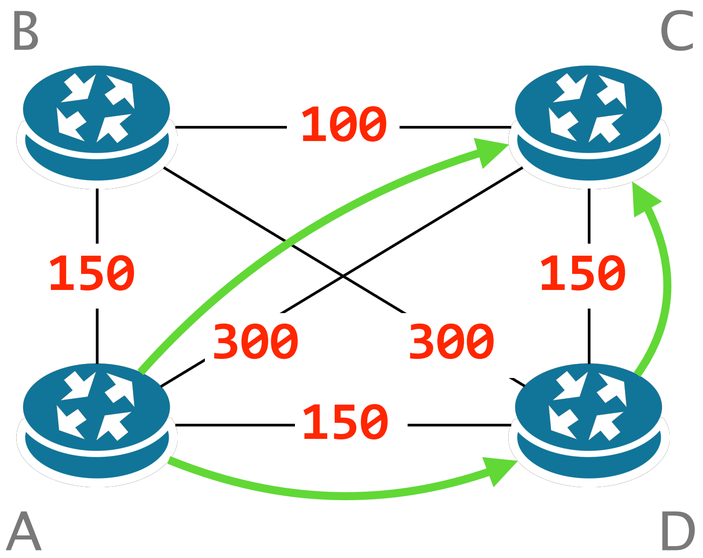
synthesis procedure

$\forall X \in \text{SamplePaths}(A,C) \setminus \text{Reqs}$

$\text{Cost}(A \rightarrow C) = \text{Cost}(A \rightarrow D \rightarrow C) < \text{Cost}(X)$

Solve

input requirements



Synthesized weights

synthesis procedure

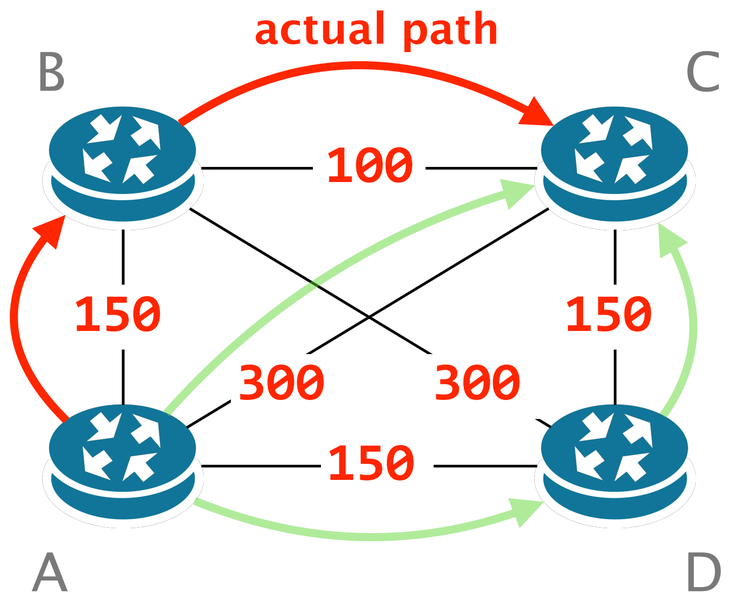
$\forall X \in \text{SamplePaths}(A,C) \setminus \text{Reqs}$

$\text{Cost}(A \rightarrow C) = \text{Cost}(A \rightarrow D \rightarrow C) < \text{Cost}(X)$

Solve

The synthesized weights are incorrect:

cost(A → B → C]) = 250 < cost(A → C) = 300



∀X ∈ SamplePaths(A,C)\Reqs

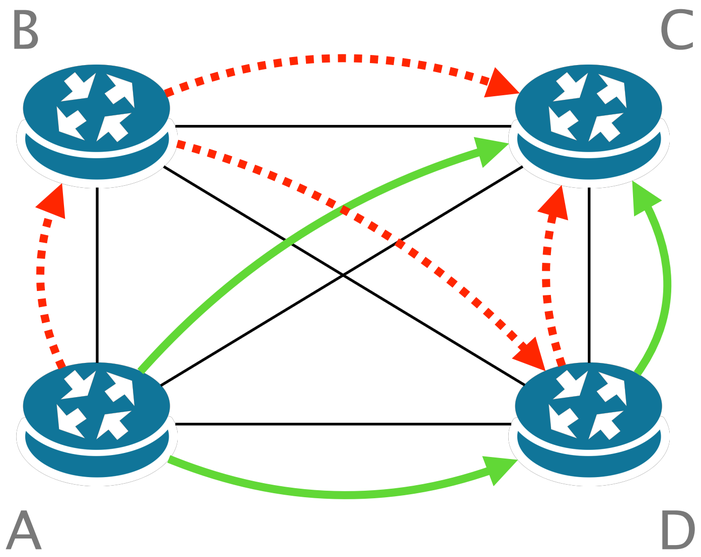
↓

Cost(A→C) = Cost(A→D→C) < Cost(X)

↓

Solve

We simply add the counter example to SamplePaths and repeat the procedure



$\forall X \in$ SamplePaths(A,C)\Reqs

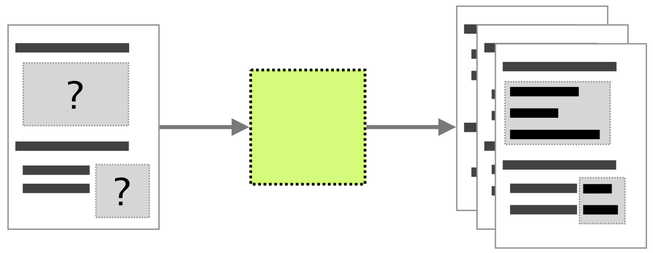
↓

Sample: { [A,B,D,C] } U { [A,B,C] }

The entire procedure usually converges in few iterations

making it very fast in practice

NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion



BGP synthesis

optimized encoding

OSPF synthesis

counter-examples-based

3 Evaluation

flexible, *yet* scalable

Question #1

Can NetComplete synthesize large-scale configurations?

Question #2

How does the concreteness of the sketch influence the running time?

We fully implemented NetComplete and showed its practicality

Code	~10K lines of Python SMT-LIB v2 and Z3
Input	OSPF, BGP, static routes as partial and concrete configs
Output	Cisco-compatible configurations validated with actual Cisco routers

Methodology

Topology	15 topologies from Topology Zoo small, medium, and large
Requirement	Simple, Any, ECMP, and ordered (random) using OSPF/BGP
Sketch	Built from a fully concrete configuration from which we made a % of the variables symbolic

NetComplete synthesizes configurations
for large networks in few minutes

NetComplete synthesizes configurations for large networks in few minutes

	Network size	Reqs. type	Synthesis time
OSPF synthesis time (sec)	Large ~150 nodes	Simple	14s
		ECMP	13s
		Ordered	249s

settings

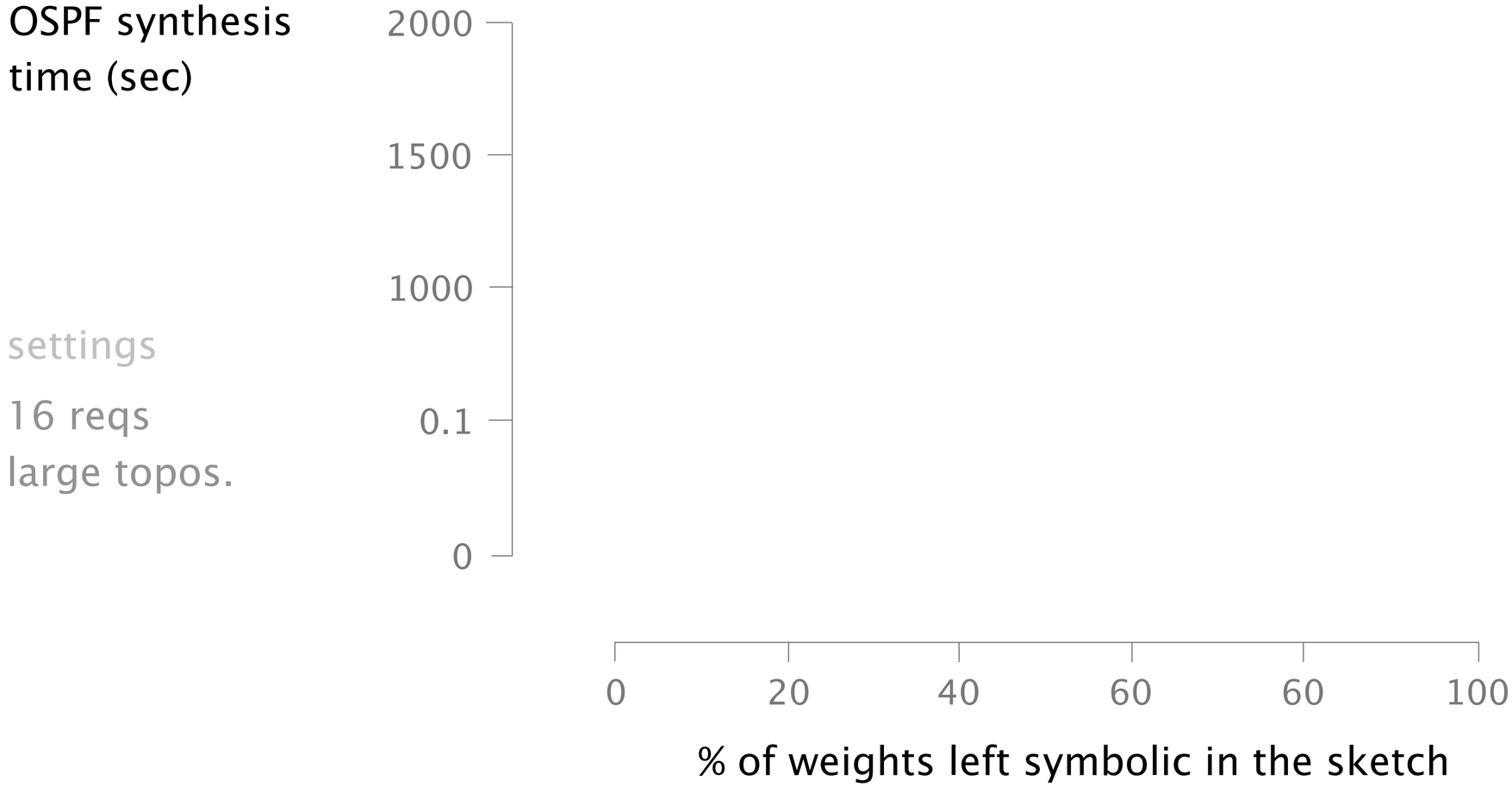
16 reqs, 50% symbolic, 5 repet.

CEGIS enabled

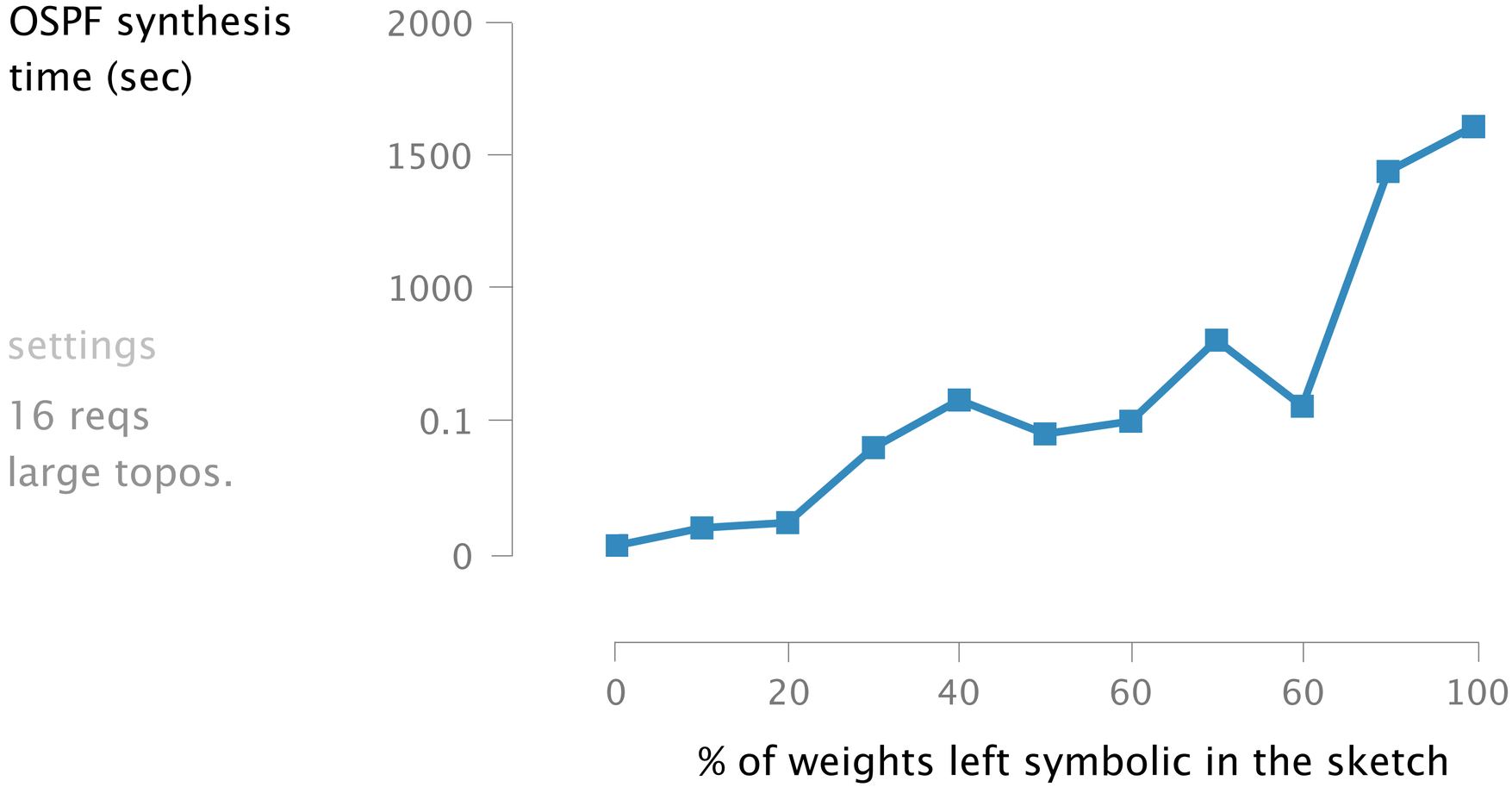
Without CEGIS, OSPF synthesis is

>100x slower and often timeouts

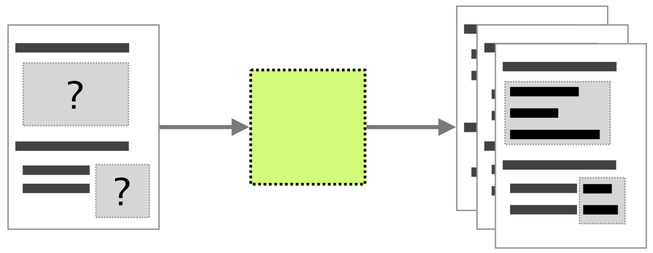
NetComplete synthesis time increases as the sketch becomes more symbolic



NetComplete synthesis time increases as the sketch becomes more symbolic



NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion



BGP synthesis

optimized encoding

OSPF synthesis

counter-examples-based

Evaluation

flexible, *yet* scalable

NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion

Autocompletes configurations with “holes”

leaving the concrete parts intact

Phrases the problem as constraints satisfaction

scales using network-specific heuristics & partial evaluation

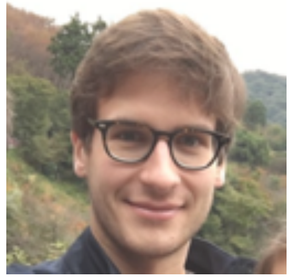
Scales to realistic network size

synthesizes configurations for large network in minutes

NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion

Ahmed El-Hassany

Petar Tsankov



Laurent Vanbever

Martin Vechev

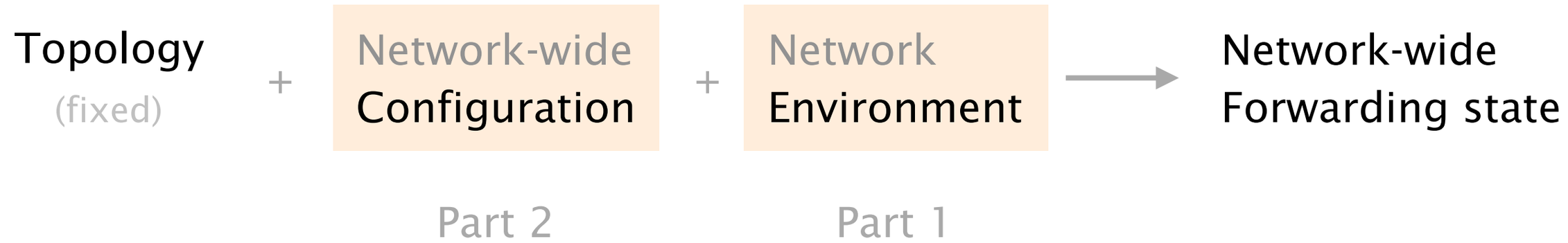
SRL
SOFTWARE RELIABILITY LAB

Networked Systems
ETH Zürich — seit 2015

ETH zürich

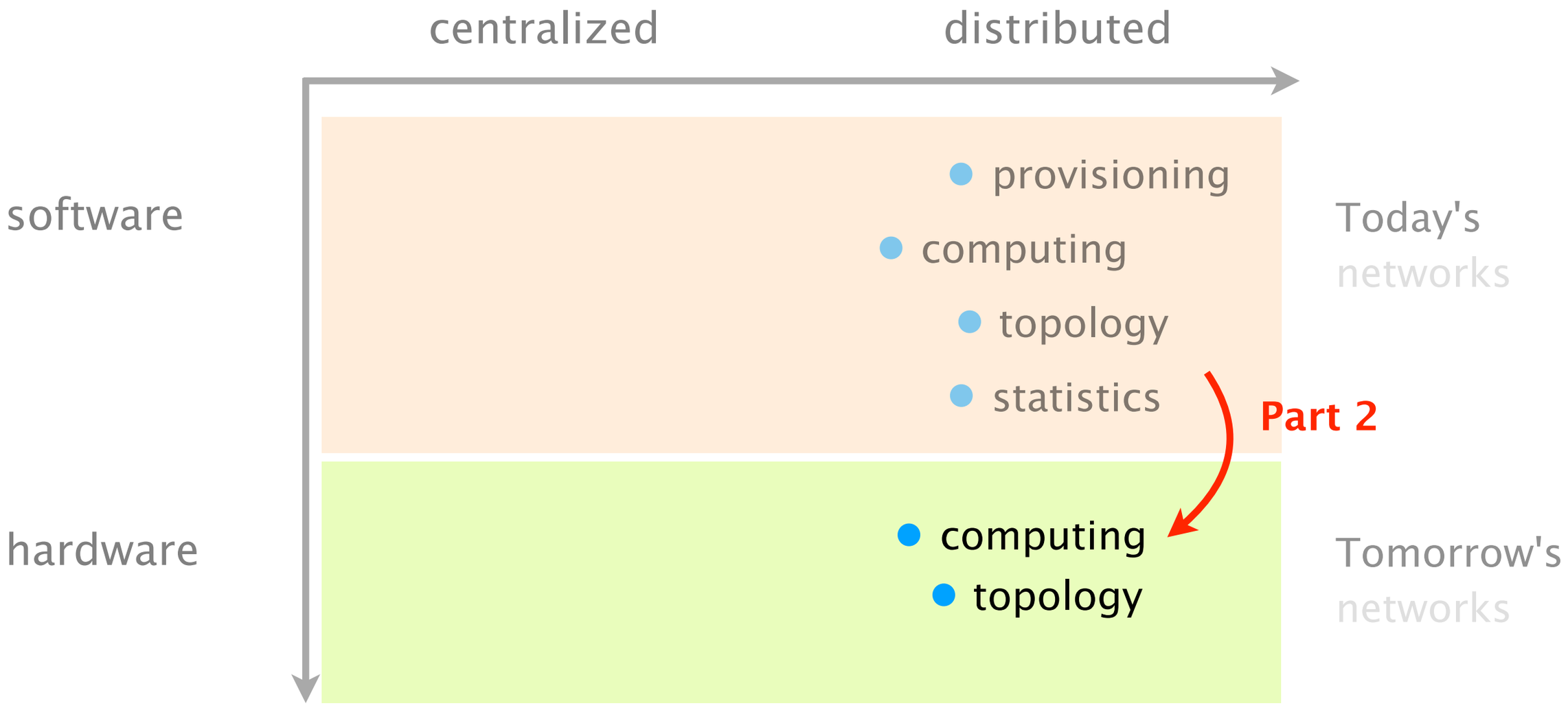
Controlling distributed computation

through synthesis



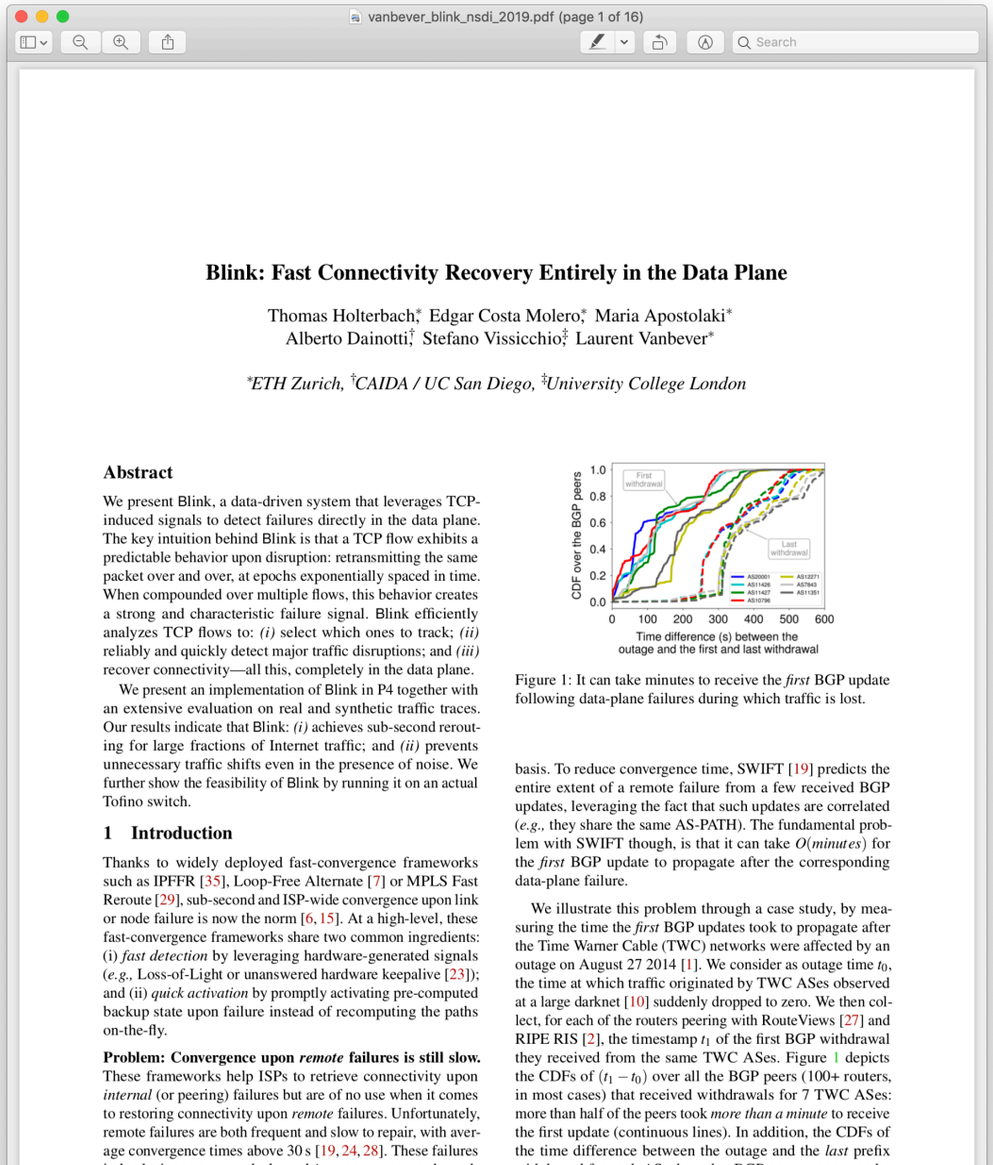
Network Control Planes

What? How? Where?



What parts of the CP should we offload (if any) and how?

Blink [NSDI'19]



vanbever_blink_nsdi_2019.pdf (page 1 of 16)

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach,* Edgar Costa Molero,* Maria Apostolaki*
Alberto Dainotti,† Stefano Vissicchio,‡ Laurent Vanbever*

*ETH Zurich, †CAIDA / UC San Diego, ‡University College London

Abstract

We present Blink, a data-driven system that leverages TCP-induced signals to detect failures directly in the data plane. The key intuition behind Blink is that a TCP flow exhibits a predictable behavior upon disruption: retransmitting the same packet over and over, at epochs exponentially spaced in time. When compounded over multiple flows, this behavior creates a strong and characteristic failure signal. Blink efficiently analyzes TCP flows to: *(i)* select which ones to track; *(ii)* reliably and quickly detect major traffic disruptions; and *(iii)* recover connectivity—all this, completely in the data plane.

We present an implementation of Blink in P4 together with an extensive evaluation on real and synthetic traffic traces. Our results indicate that Blink: *(i)* achieves sub-second rerouting for large fractions of Internet traffic; and *(ii)* prevents unnecessary traffic shifts even in the presence of noise. We further show the feasibility of Blink by running it on an actual Tofino switch.

1 Introduction

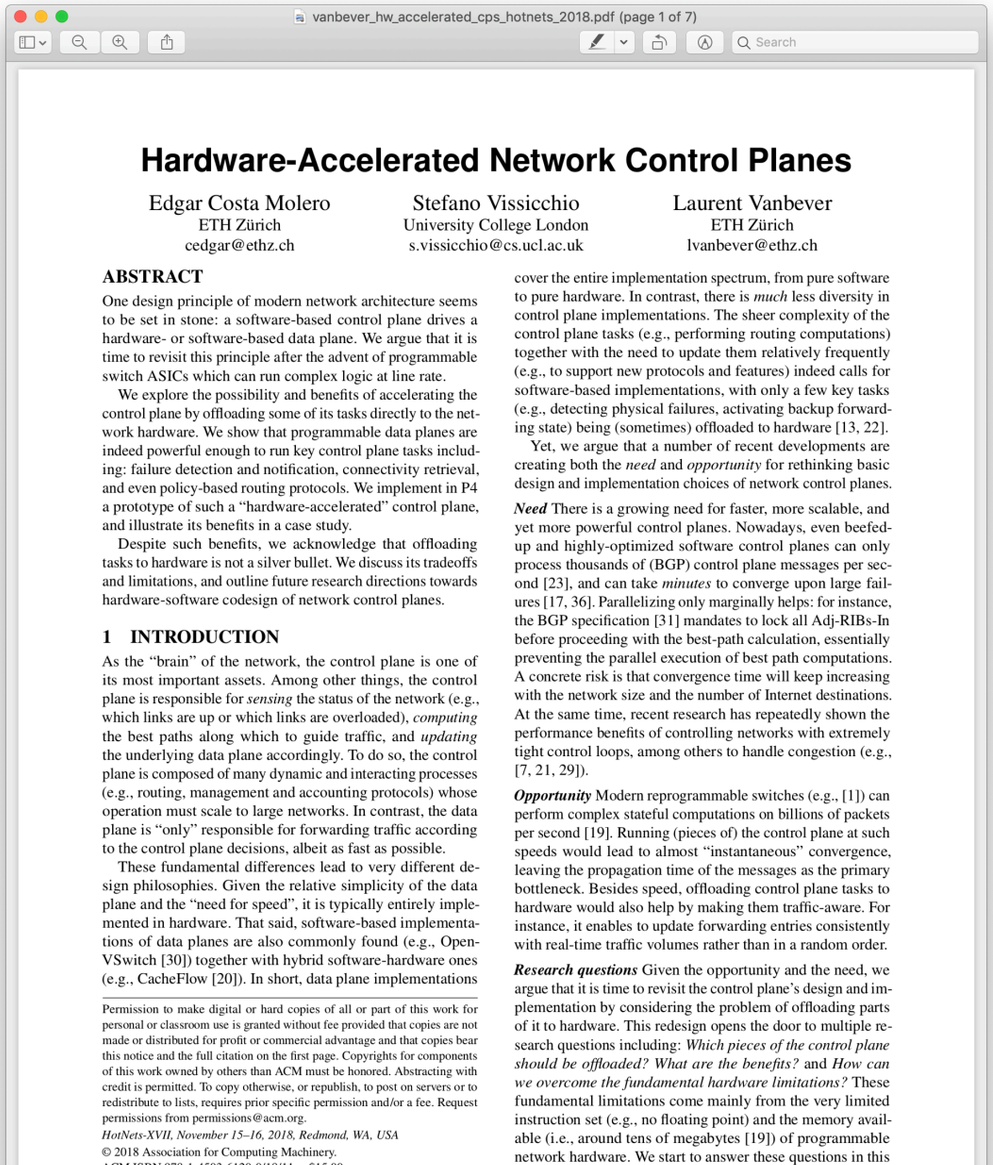
Thanks to widely deployed fast-convergence frameworks such as IPFFR [35], Loop-Free Alternate [7] or MPLS Fast Reroute [29], sub-second and ISP-wide convergence upon link or node failure is now the norm [6, 15]. At a high-level, these fast-convergence frameworks share two common ingredients: (i) *fast detection* by leveraging hardware-generated signals (*e.g.,* Loss-of-Light or unanswered hardware keepalive [23]); and (ii) *quick activation* by promptly activating pre-computed backup state upon failure instead of recomputing the paths on-the-fly.

Problem: Convergence upon *remote* failures is still slow. These frameworks help ISPs to retrieve connectivity upon *internal* (or peering) failures but are of no use when it comes to restoring connectivity upon *remote* failures. Unfortunately, remote failures are both frequent and slow to repair, with average convergence times above 30 s [19, 24, 28]. These failures indeed trigger a *control-plane-driven* convergence through the propagation of BGP updates on a per-router and per-prefix basis. To reduce convergence time, SWIFT [19] predicts the entire extent of a remote failure from a few received BGP updates, leveraging the fact that such updates are correlated (*e.g.,* they share the same AS-PATH). The fundamental problem with SWIFT though, is that it can take $O(minutes)$ for the *first* BGP update to propagate after the corresponding data-plane failure.

Figure 1: It can take minutes to receive the *first* BGP update following data-plane failures during which traffic is lost.

We illustrate this problem through a case study, by measuring the time the *first* BGP updates took to propagate after the Time Warner Cable (TWC) networks were affected by an outage on August 27 2014 [1]. We consider as outage time t_0, the time at which traffic originated by TWC ASes observed at a large darknet [10] suddenly dropped to zero. We then collect, for each of the routers peering with RouteViews [27] and RIPE RIS [2], the timestamp t_1 of the first BGP withdrawal they received from the same TWC ASes. Figure 1 depicts the CDFs of $(t_1 - t_0)$ over all the BGP peers (100+ routers, in most cases) that received withdrawals for 7 TWC ASes: more than half of the peers took *more than a minute* to receive the first update (continuous lines). In addition, the CDFs of the time difference between the outage and the *last* prefix withdrawal for each AS, show that BGP convergence can be as slow as several minutes (dashed lines).

HW-accelerated CPs [HotNets'18]



vanbever_hw_accelerated_cps_hotnets_2018.pdf (page 1 of 7)

Hardware-Accelerated Network Control Planes

Edgar Costa Molero
ETH Zürich
cedgar@ethz.ch

Stefano Vissicchio
University College London
s.vissicchio@cs.ucl.ac.uk

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

ABSTRACT

One design principle of modern network architecture seems to be set in stone: a software-based control plane drives a hardware- or software-based data plane. We argue that it is time to revisit this principle after the advent of programmable switch ASICs which can run complex logic at line rate.

We explore the possibility and benefits of accelerating the control plane by offloading some of its tasks directly to the network hardware. We show that programmable data planes are indeed powerful enough to run key control plane tasks including: failure detection and notification, connectivity retrieval, and even policy-based routing protocols. We implement in P4 a prototype of such a "hardware-accelerated" control plane, and illustrate its benefits in a case study.

Despite such benefits, we acknowledge that offloading tasks to hardware is not a silver bullet. We discuss its tradeoffs and limitations, and outline future research directions towards hardware-software codesign of network control planes.

1 INTRODUCTION

As the "brain" of the network, the control plane is one of its most important assets. Among other things, the control plane is responsible for *sensing* the status of the network (e.g., which links are up or which links are overloaded), *computing* the best paths along which to guide traffic, and *updating* the underlying data plane accordingly. To do so, the control plane is composed of many dynamic and interacting processes (e.g., routing, management and accounting protocols) whose operation must scale to large networks. In contrast, the data plane is "only" responsible for forwarding traffic according to the control plane decisions, albeit as fast as possible.

These fundamental differences lead to very different design philosophies. Given the relative simplicity of the data plane and the "need for speed", it is typically entirely implemented in hardware. That said, software-based implementations of data planes are also commonly found (e.g., OpenVSwitch [30]) together with hybrid software-hardware ones (e.g., CacheFlow [20]). In short, data plane implementations cover the entire implementation spectrum, from pure software to pure hardware. In contrast, there is *much* less diversity in control plane implementations. The sheer complexity of the control plane tasks (e.g., performing routing computations) together with the need to update them relatively frequently (e.g., to support new protocols and features) indeed calls for software-based implementations, with only a few key tasks (e.g., detecting physical failures, activating backup forwarding state) being (sometimes) offloaded to hardware [13, 22].

Yet, we argue that a number of recent developments are creating both the *need* and *opportunity* for rethinking basic design and implementation choices of network control planes.

Need There is a growing need for faster, more scalable, and yet more powerful control planes. Nowadays, even beefed-up and highly-optimized software control planes can only process thousands of (BGP) control plane messages per second [23], and can take *minutes* to converge upon large failures [17, 36]. Parallelizing only marginally helps: for instance, the BGP specification [31] mandates to lock all Adj-RIBs-In before proceeding with the best-path calculation, essentially preventing the parallel execution of best path computations. A concrete risk is that convergence time will keep increasing with the network size and the number of Internet destinations. At the same time, recent research has repeatedly shown the performance benefits of controlling networks with extremely tight control loops, among others to handle congestion (e.g., [7, 21, 29]).

Opportunity Modern reprogrammable switches (e.g., [1]) can perform complex stateful computations on billions of packets per second [19]. Running (pieces of) the control plane at such speeds would lead to almost "instantaneous" convergence, leaving the propagation time of the messages as the primary bottleneck. Besides speed, offloading control plane tasks to hardware would also help by making them traffic-aware. For instance, it enables to update forwarding entries consistently with real-time traffic volumes rather than in a random order.

Research questions Given the opportunity and the need, we argue that it is time to revisit the control plane's design and implementation by considering the problem of offloading parts of it to hardware. This redesign opens the door to multiple research questions including: *Which pieces of the control plane should be offloaded? What are the benefits?* and *How can we overcome the fundamental hardware limitations?* These fundamental limitations come mainly from the very limited instruction set (e.g., no floating point) and the memory available (i.e., around tens of megabytes [19]) of programmable network hardware. We start to answer these questions in this paper and make two contributions.

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA

ACM ISBN 978-1-4503-6120-0/18/11…$15.00
https://doi.org/10.1145/3286062.3286080

What parts of the CP should we offload (if any) and how?

Blink [NSDI'19]

vanbever_blink_nsdi_2019.pdf (page 1 of 16)

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach*, Edgar Costa Molero*, Maria Apostolaki*
Alberto Dainotti†, Stefano Vissicchio‡, Laurent Vanbever*

*ETH Zurich, †CAIDA / UC San Diego, ‡University College London

Abstract

We present Blink, a data-driven system that leverages TCP-induced signals to detect failures directly in the data plane. The key intuition behind Blink is that a TCP flow exhibits a predictable behavior upon disruption: retransmitting the same packet over and over, at epochs exponentially spaced in time. When compounded over multiple flows, this behavior creates a strong and characteristic failure signal. Blink efficiently analyzes TCP flows to: *(i)* select which ones to track; *(ii)* reliably and quickly detect major traffic disruptions; and *(iii)* recover connectivity—all this, completely in the data plane.

We present an implementation of Blink in P4 together with an extensive evaluation on real and synthetic traffic traces. Our results indicate that Blink: *(i)* achieves sub-second rerouting for large fractions of Internet traffic; and *(ii)* prevents unnecessary traffic shifts even in the presence of noise. We further show the feasibility of Blink by running it on an actual Tofino switch.

1 Introduction

Thanks to widely deployed fast-convergence frameworks such as IPFFR [35], Loop-Free Alternate [7] or MPLS Fast Reroute [29], sub-second and ISP-wide convergence upon link or node failure is now the norm [6, 15]. At a high-level, these fast-convergence frameworks share two common ingredients: (i) *fast detection* by leveraging hardware-generated signals (*e.g.,* Loss-of-Light or unanswered hardware keepalive [23]); and (ii) *quick activation* by promptly activating pre-computed backup state upon failure instead of recomputing the paths on-the-fly.

Problem: Convergence upon *remote* failures is still slow. These frameworks help ISPs to retrieve connectivity upon *internal* (or peering) failures but are of no use when it comes to restoring connectivity upon *remote* failures. Unfortunately, remote failures are both frequent and slow to repair, with average convergence times above 30 s [19, 24, 28]. These failures indeed trigger a *control-plane-driven* convergence through the propagation of BGP updates on a per-router and per-prefix basis. To reduce convergence time, SWIFT [19] predicts the entire extent of a remote failure from a few received BGP updates, leveraging the fact that such updates are correlated (*e.g.,* they share the same AS-PATH). The fundamental problem with SWIFT though, is that it can take $O(minutes)$ for the *first* BGP update to propagate after the corresponding data-plane failure.

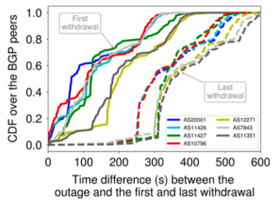


Figure 1: It can take minutes to receive the *first* BGP update following data-plane failures during which traffic is lost.

We illustrate this problem through a case study, by measuring the time the *first* BGP updates took to propagate after the Time Warner Cable (TWC) networks were affected by an outage on August 27 2014 [1]. We consider as outage time t_0, the time at which traffic originated by TWC ASes observed at a large darknet [10] suddenly dropped to zero. We then collect, for each of the routers peering with RouteViews [27] and RIPE RIS [2], the timestamp t_1 of the first BGP withdrawal they received from the same TWC ASes. Figure 1 depicts the CDFs of $(t_1 - t_0)$ over all the BGP peers (100+ routers, in most cases) that received withdrawals for 7 TWC ASes: more than half of the peers took *more than a minute* to receive the first update (continuous lines). In addition, the CDFs of the time difference between the outage and the *last* prefix withdrawal for each AS, show that BGP convergence can be as slow as several minutes (dashed lines).

HW-accelerated CPs [HotNets'18]

vanbever_hw_accelerated_cps_hotnets_2018.pdf (page 1 of 7)

Hardware-Accelerated Network Control Planes

Edgar Costa Molero
ETH Zürich
cedgar@ethz.ch

Stefano Vissicchio
University College London
s.vissicchio@cs.ucl.ac.uk

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

ABSTRACT

One design principle of modern network architecture seems to be set in stone: a software-based control plane drives a hardware- or software-based data plane. We argue that it is time to revisit this principle after the advent of programmable switch ASICs which can run complex logic at line rate.

We explore the possibility and benefits of accelerating the control plane by offloading some of its tasks directly to the network hardware. We show that programmable data planes are indeed powerful enough to run key control plane tasks including: failure detection and notification, connectivity retrieval, and even policy-based routing protocols. We implement in P4 a prototype of such a "hardware-accelerated" control plane, and illustrate its benefits in a case study.

Despite such benefits, we acknowledge that offloading tasks to hardware is not a silver bullet. We discuss its tradeoffs and limitations, and outline future research directions towards hardware-software codesign of network control planes.

1 INTRODUCTION

As the "brain" of the network, the control plane is one of its most important assets. Among other things, the control plane is responsible for *sensing* the status of the network (e.g., which links are up or which links are overloaded), *computing* the best paths along which to guide traffic, and *updating* the underlying data plane accordingly. To do so, the control plane is composed of many dynamic and interacting processes (e.g., routing, management and accounting protocols) whose operation must scale to large networks. In contrast, the data plane is "only" responsible for forwarding traffic according to the control plane decisions, albeit as fast as possible.

These fundamental differences lead to very different design philosophies. Given the relative simplicity of the data plane and the "need for speed", it is typically entirely implemented in hardware. That said, software-based implementations of data planes are also commonly found (e.g., OpenVSwitch [30]) together with hybrid software-hardware ones (e.g., CacheFlow [20]). In short, data plane implementations cover the entire implementation spectrum, from pure software to pure hardware. In contrast, there is *much* less diversity in control plane implementations. The sheer complexity of the control plane tasks (e.g., performing routing computations) together with the need to update them relatively frequently (e.g., to support new protocols and features) indeed calls for software-based implementations, with only a few key tasks (e.g., detecting physical failures, activating backup forwarding state) being (sometimes) offloaded to hardware [13, 22].

Yet, we argue that a number of recent developments are creating both the *need* and *opportunity* for rethinking basic design and implementation choices of network control planes.

Need There is a growing need for faster, more scalable, and yet more powerful control planes. Nowadays, even beefed-up and highly-optimized software control planes can only process thousands of (BGP) control plane messages per second [23], and can take *minutes* to converge upon large failures [17, 36]. Parallelizing only marginally helps: for instance, the BGP specification [31] mandates to lock all Adj-RIBs-In before proceeding with the best-path calculation, essentially preventing the parallel execution of best path computations. A concrete risk is that convergence time will keep increasing with the network size and the number of Internet destinations. At the same time, recent research has repeatedly shown the performance benefits of controlling networks with extremely tight control loops, among others to handle congestion (e.g., [7, 21, 29]).

Opportunity Modern reprogrammable switches (e.g., [1]) can perform complex stateful computations on billions of packets per second [19]. Running (pieces of) the control plane at such speeds would lead to almost "instantaneous" convergence, leaving the propagation time of the messages as the primary bottleneck. Besides speed, offloading control plane tasks to hardware would also help by making them traffic-aware. For instance, it enables to update forwarding entries consistently with real-time traffic volumes rather than in a random order.

Research questions Given the opportunity and the need, we argue that it is time to revisit the control plane's design and implementation by considering the problem of offloading parts of it to hardware. This redesign opens the door to multiple research questions including: *Which pieces of the control plane should be offloaded? What are the benefits?* and *How can we overcome the fundamental hardware limitations?* These fundamental limitations come mainly from the very limited instruction set (e.g., no floating point) and the memory available (i.e., around tens of megabytes [19]) of programmable network hardware. We start to answer these questions in this paper and make two contributions.

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA

ACM ISBN 978-1-4503-6120-0/18/11…$15.00
https://doi.org/10.1145/3286062.3286080

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach
ETH Zürich

NSDI
26th February 2019

https://blink.ethz.ch

Joint work with

Edgar Costa Molero ETH Zürich
Maria Apostolaki ETH Zürich
Stefano Vissicchio University College London
Alberto Dainotti CAIDA, UC San Diego
Laurent Vanbever ETH Zürich

Fire at AT&T facility causes widespread outage in North Texas

TELECOM

Nationwide internet outage affects CenturyLink customers

Time Warner Cable comes back from nationwide Internet outage

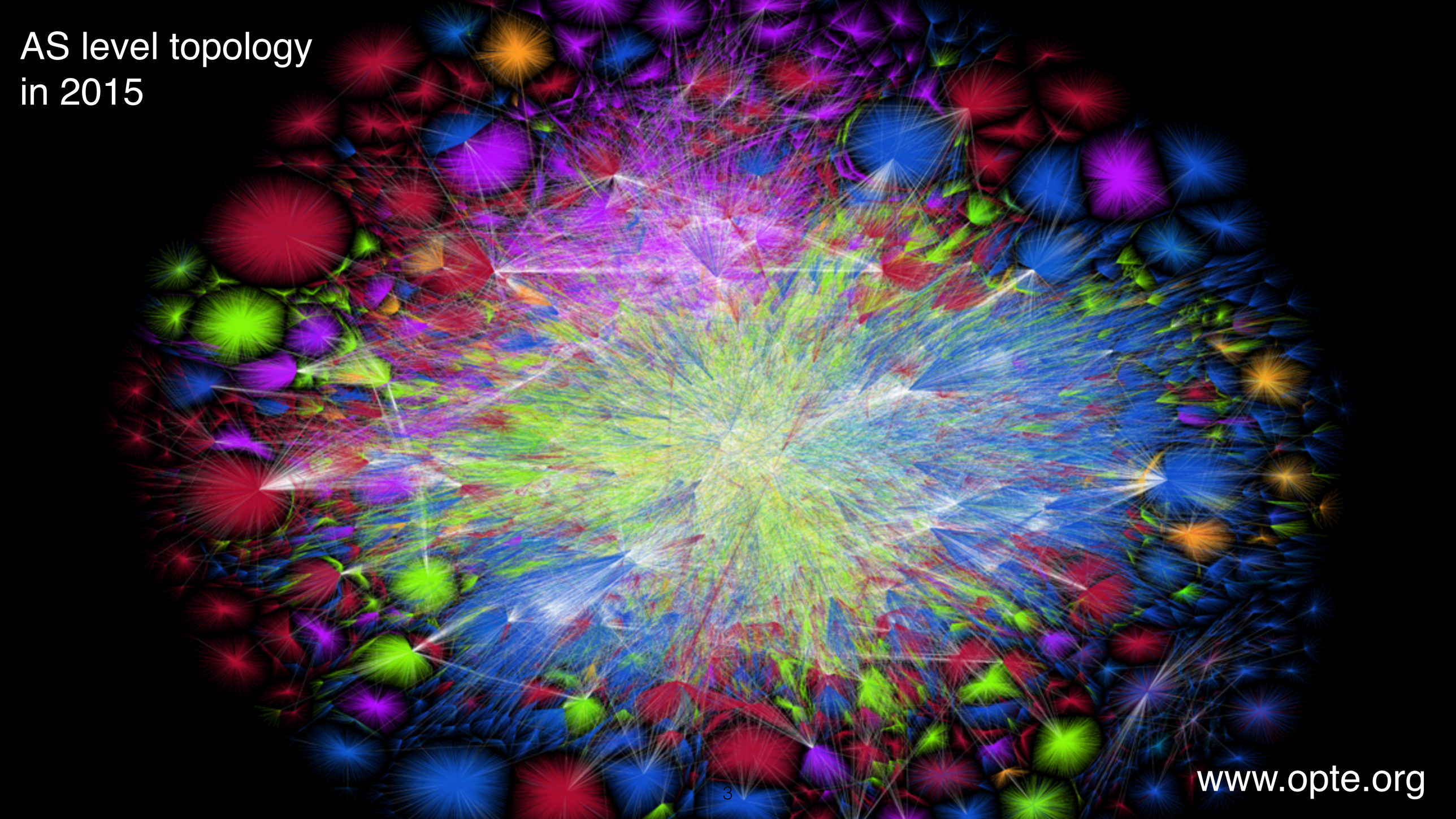
by Brian Stelter @brianstelter

August 27, 2014: 11:07 PM ET

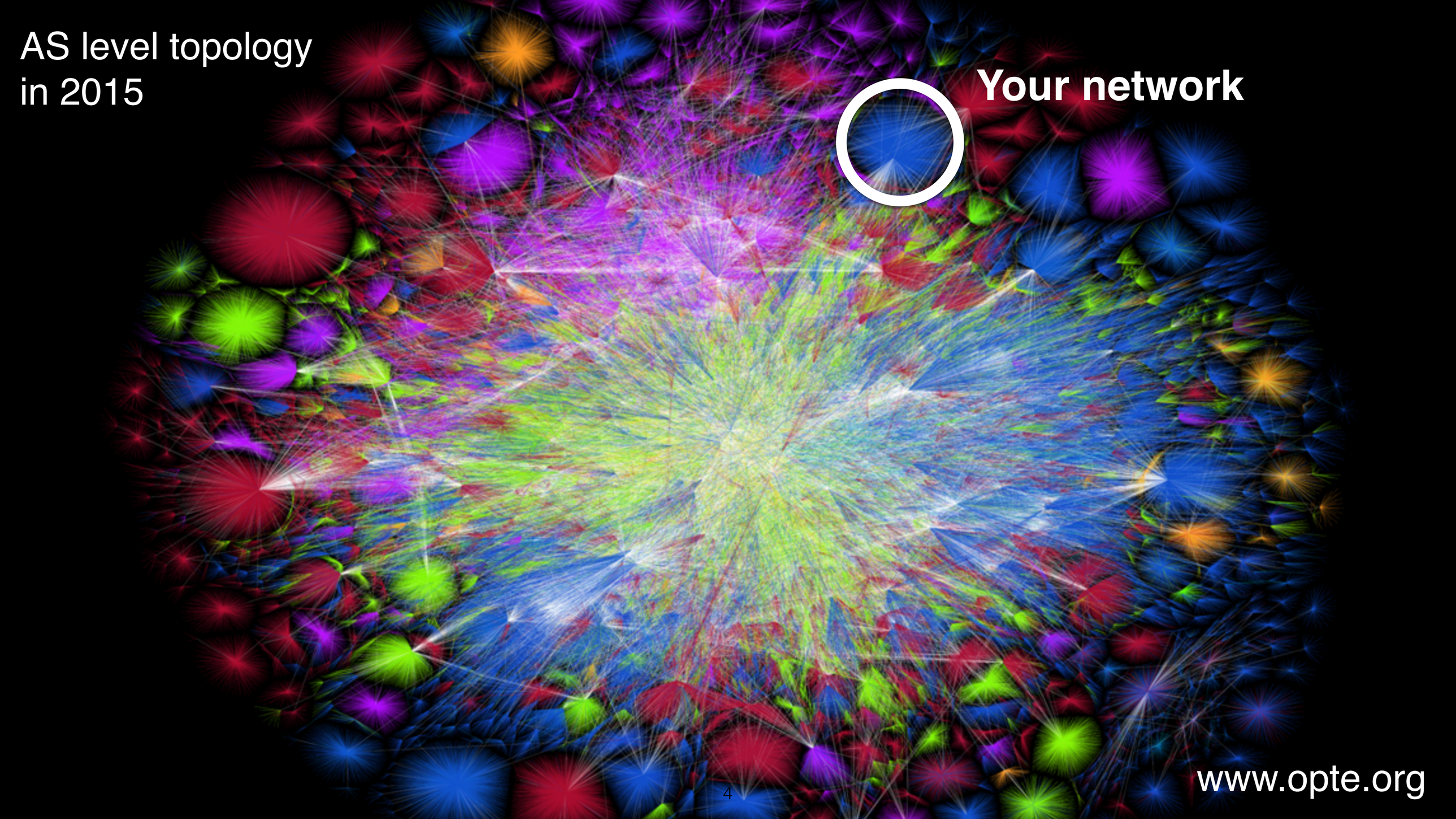
Major internet outage hits the U.S. - Affecting customers of Comcast, Verizon, and AT&T

November 6, 2017 | Emerging Threats

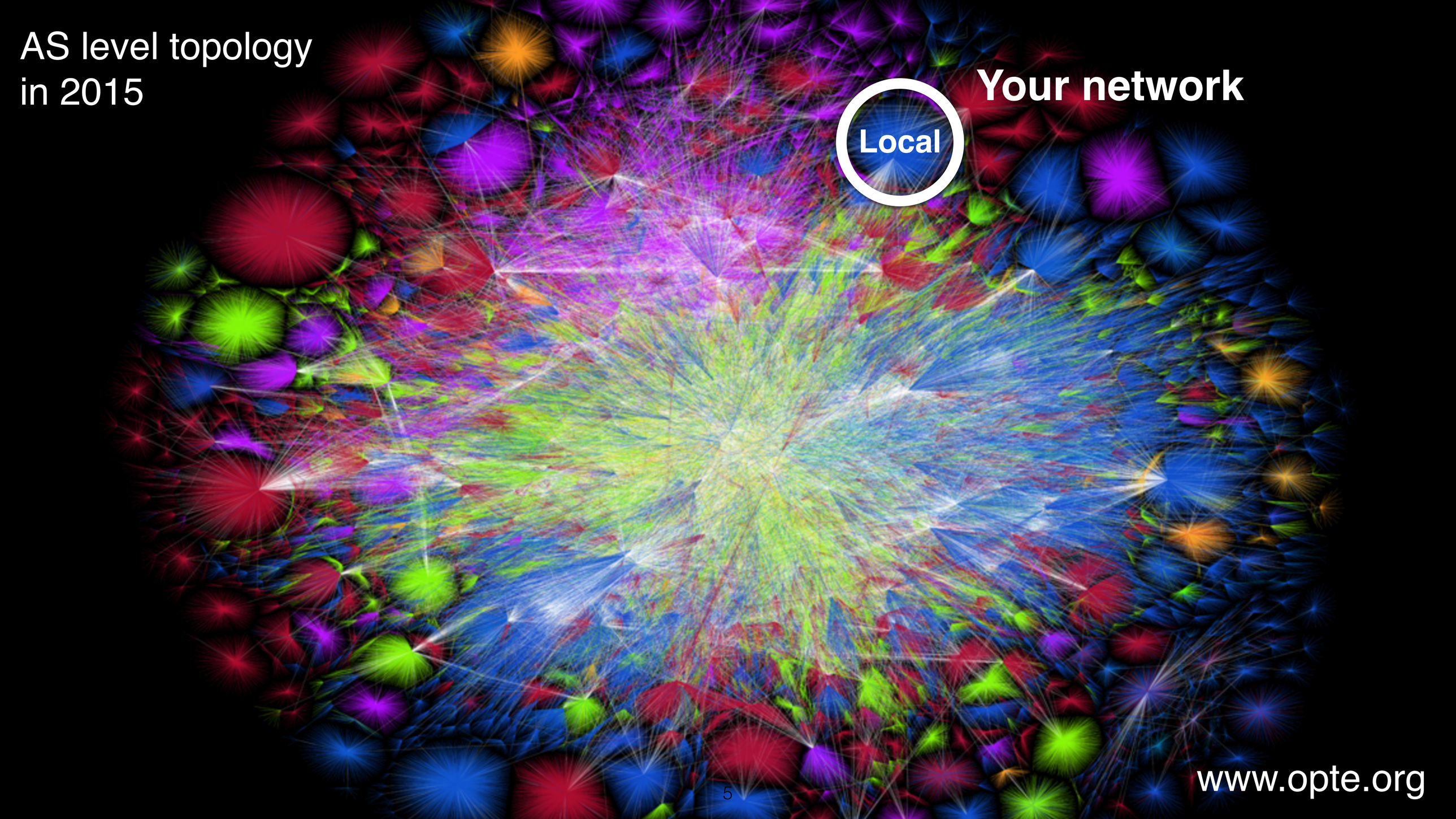
AS level topology in 2015



www.opte.org

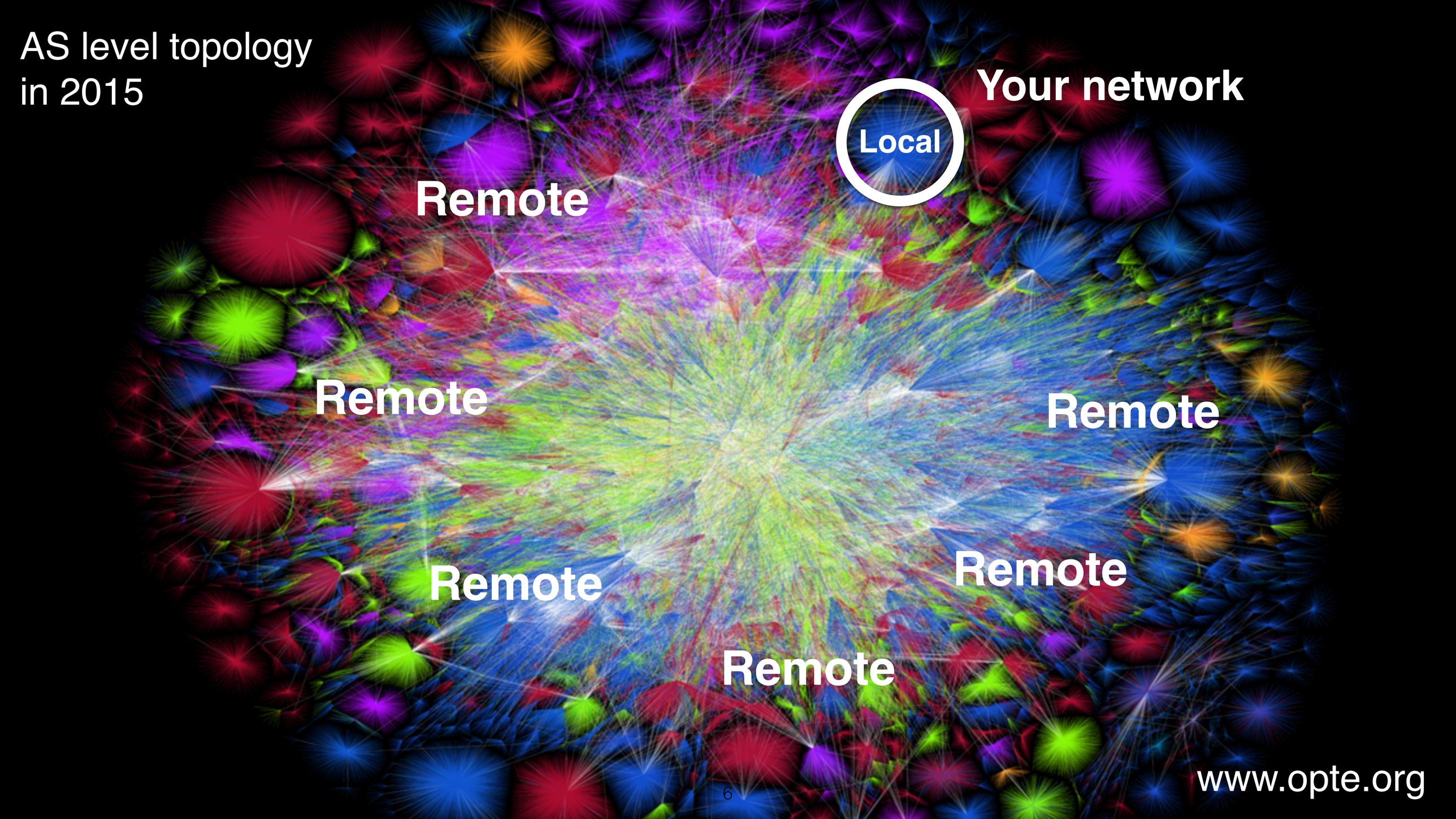

AS level topology
in 2015
Your network
www.opte.org

AS level topology
in 2015



www.opte.org

AS level topology
in 2015



www.opte.org

Upon local failures, connectivity can be quickly restored

Upon local failures, connectivity can be quickly restored

Fast failure detection
using *e.g.,* hardware-generated signals

Fast traffic rerouting
using *e.g.,* Prefix Independent Convergence
or MPLS Fast Reroute

Upon remote failures, the only way to restore connectivity is to wait for the Internet to converge

Upon remote failures, the only way to restore connectivity is to wait for the Internet to converge

… and the Internet converges *very* slowly*

*Holterbach et al. SWIFT: Predictive Fast Reroute ACM SIGCOMM, 2017

Fire at AT&T facility causes widespread outage in North Texas

TELECOM

Nationwide internet outage affects CenturyLink customers

Time Warner Cable comes back from nationwide Internet outage

by Brian Stelter @brianstelter

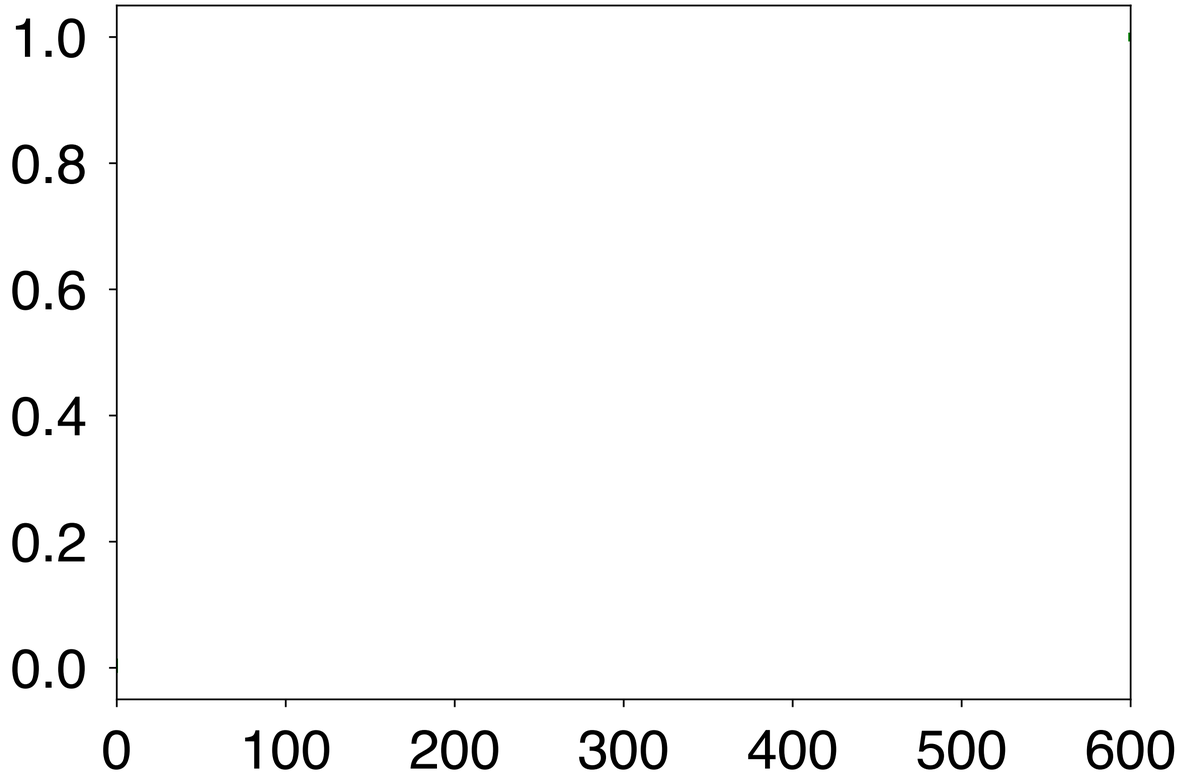
August 27, 2014: 11:07 PM ET

Major internet outage hits the U.S. - Affecting customers of Comcast, Verizon, and AT&T

November 6, 2017 | Emerging Threats

BGP took minutes to converge upon the Time Warner Cable outage in 2014

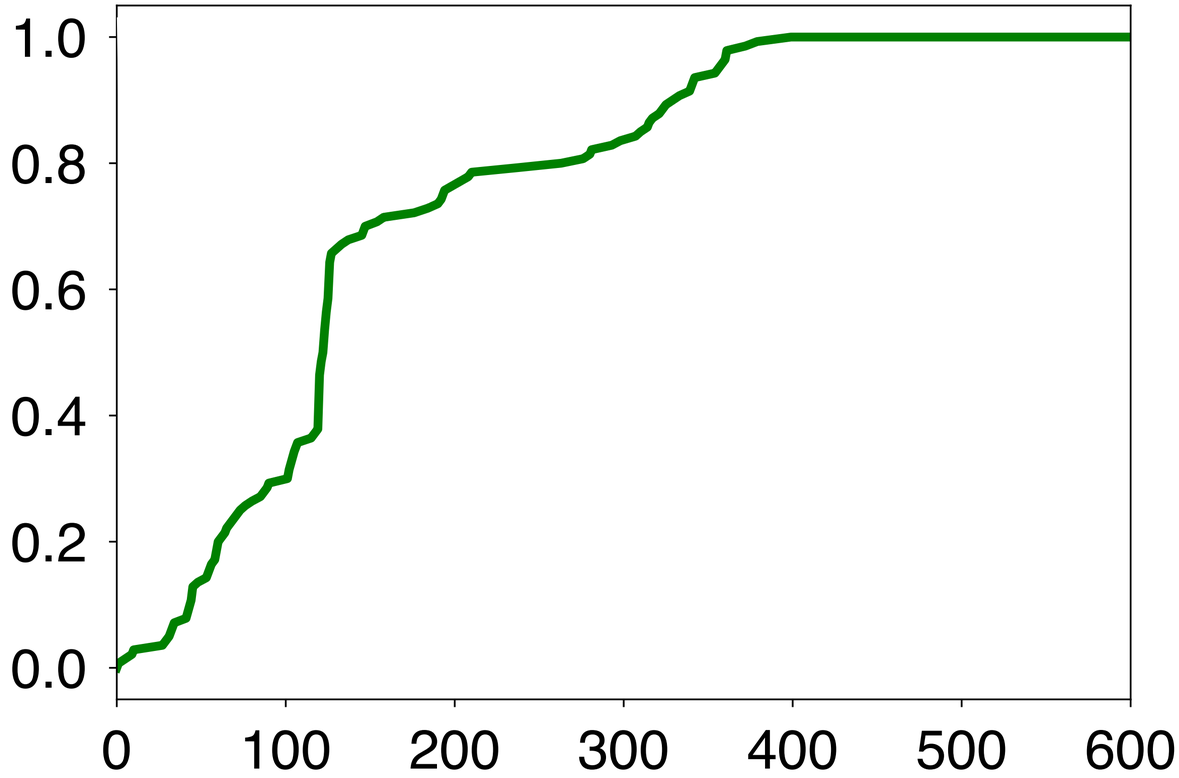
CDF over the BGP peers



Time difference between the outage
and the BGP withdrawals (s)

BGP took minutes to converge upon the Time Warner Cable outage in 2014

CDF over the BGP peers



Time difference between the outage and the BGP withdrawals (s)

Control-plane (*e.g.,* BGP) based techniques typically converge slowly upon remote outages

Control-plane (*e.g.,* BGP) based techniques typically converge slowly upon remote outages

What about using data-plane signals for fast rerouting?

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach
ETH Zürich

NSDI
26th February 2019

https://blink.ethz.ch

Outline

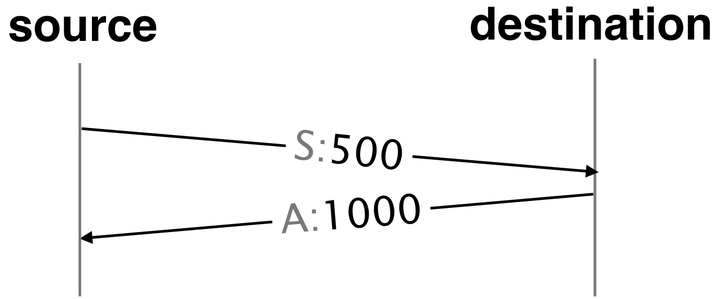
1. Why and how to use data-plane signals for fast rerouting
2. ***Blink*** infers more than 80% of the failures, often within 1s
3. ***Blink*** quickly reroutes traffic to working backup paths
4. ***Blink*** works in practice, on existing devices

Outline

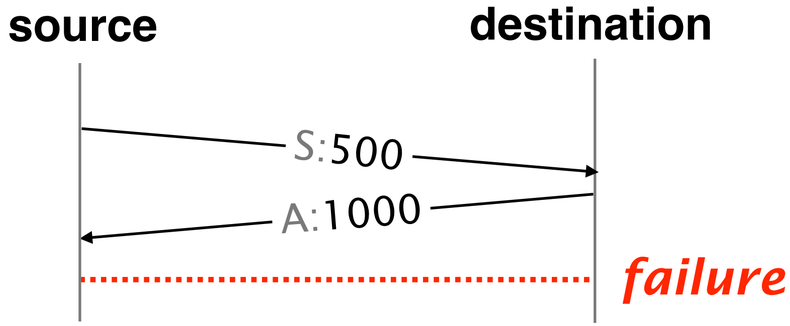
1. Why and how to use data-plane signals for fast rerouting
2. ***Blink*** infers more than 80% of the failures, often within 1s
3. ***Blink*** quickly reroutes traffic to working backup paths
4. ***Blink*** works in practice, on existing devices

TCP flows exhibit the same behavior upon failures

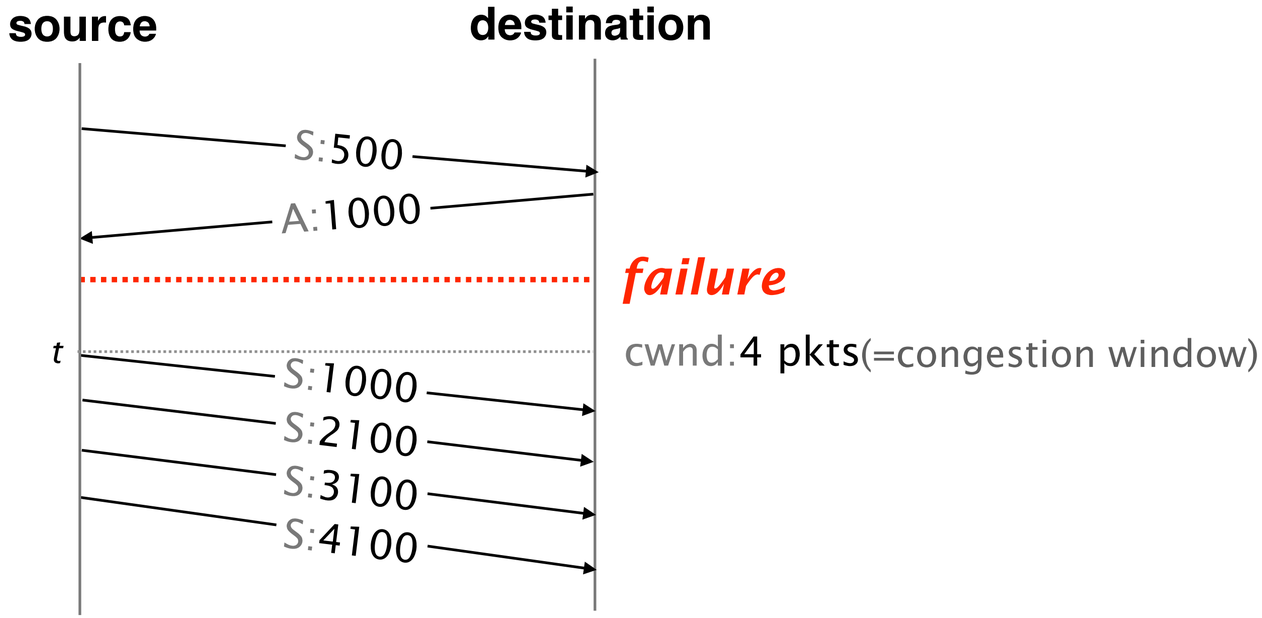
TCP flows exhibit the same behavior upon failures



TCP flows exhibit the same behavior upon failures



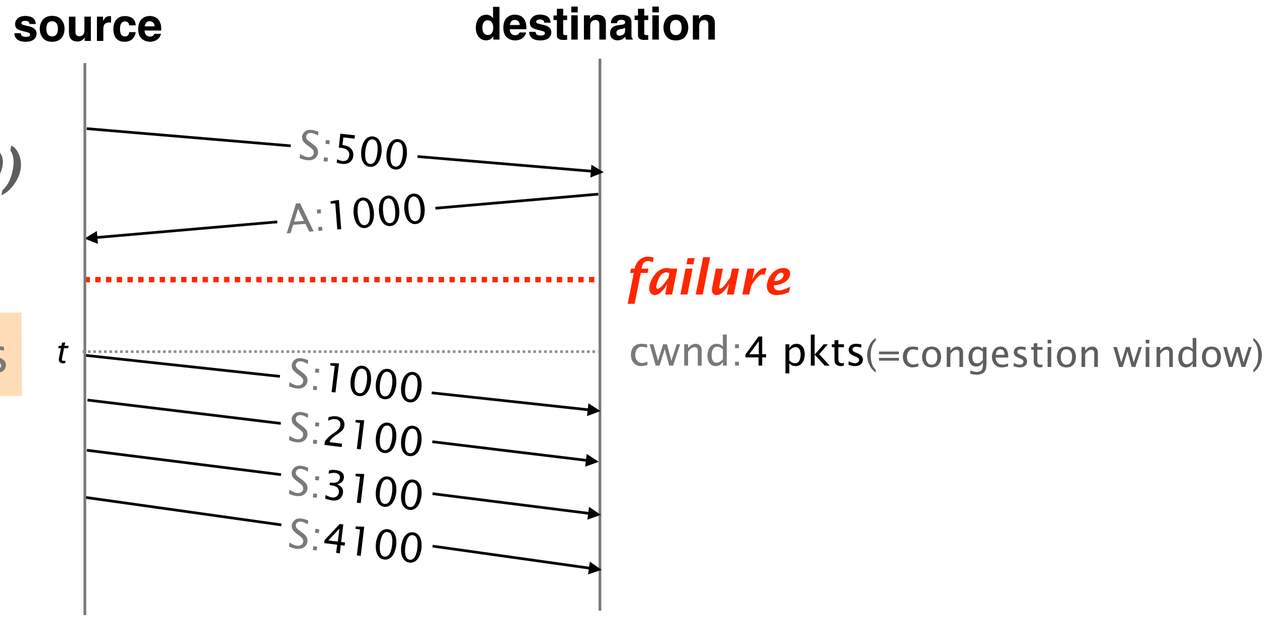
TCP flows exhibit the same behavior upon failures



TCP flows exhibit the same behavior upon failures

Retransmission timeout (RTO)
= SRTT + 4∗RTT_VAR

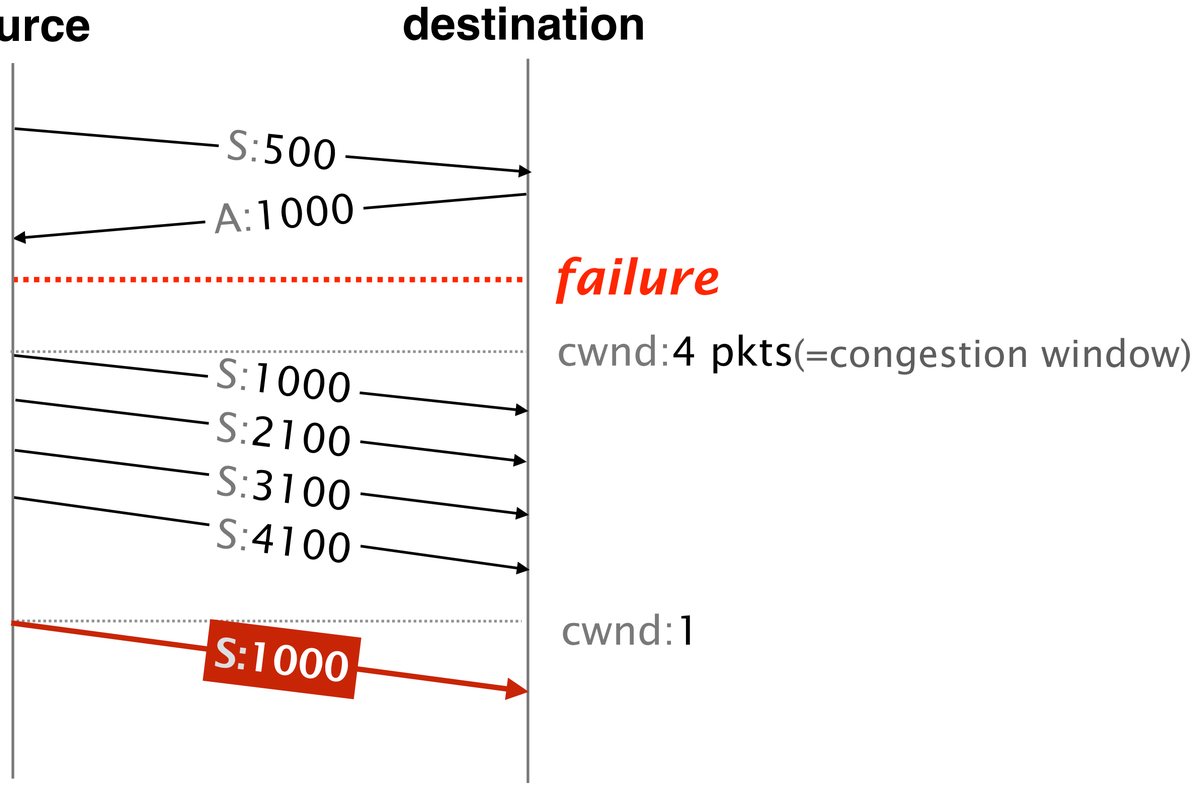
RTO: 200ms



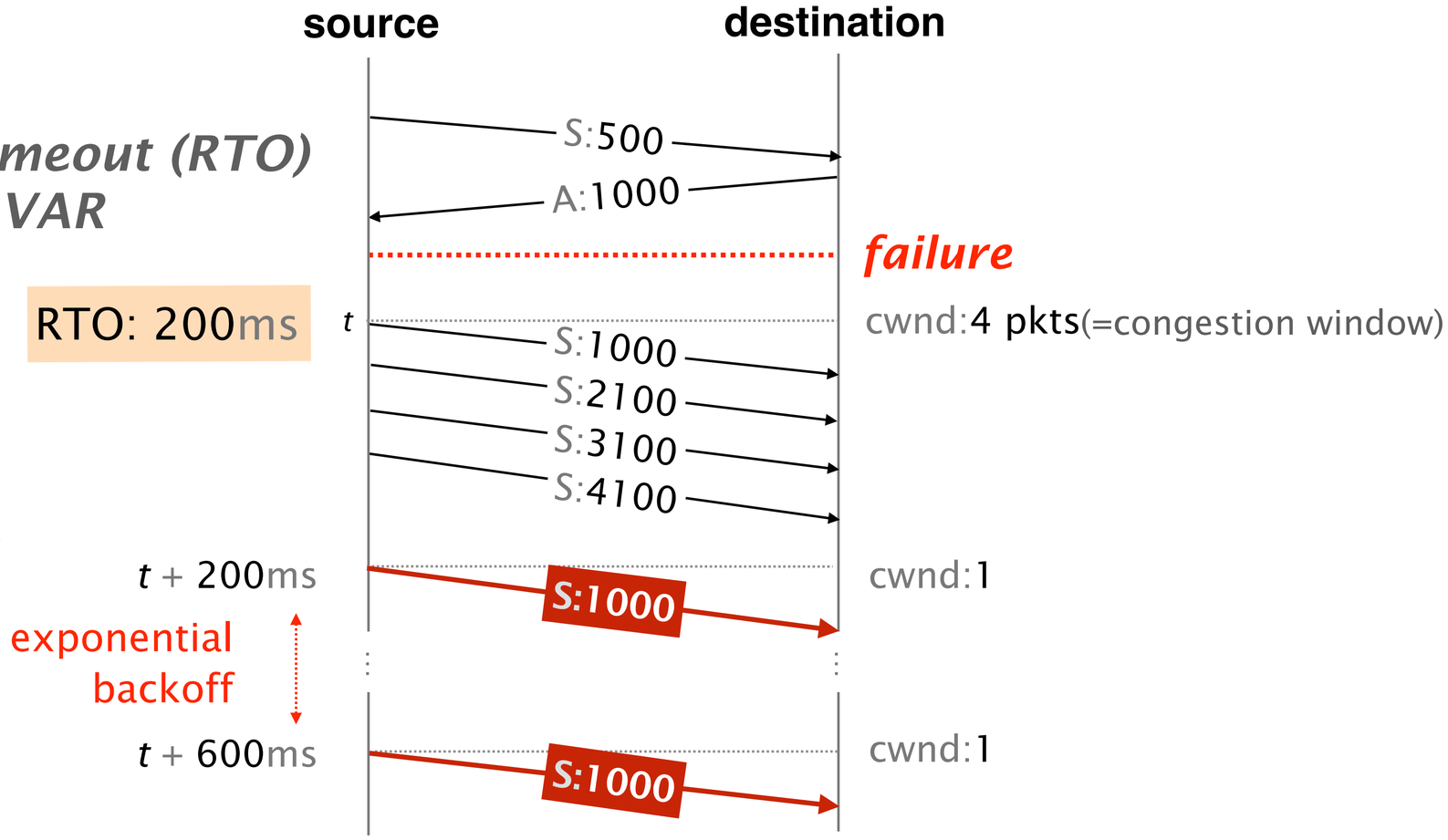
TCP flows exhibit the same behavior upon failures

Retransmission timeout (RTO)
= SRTT + 4∗RTT_VAR

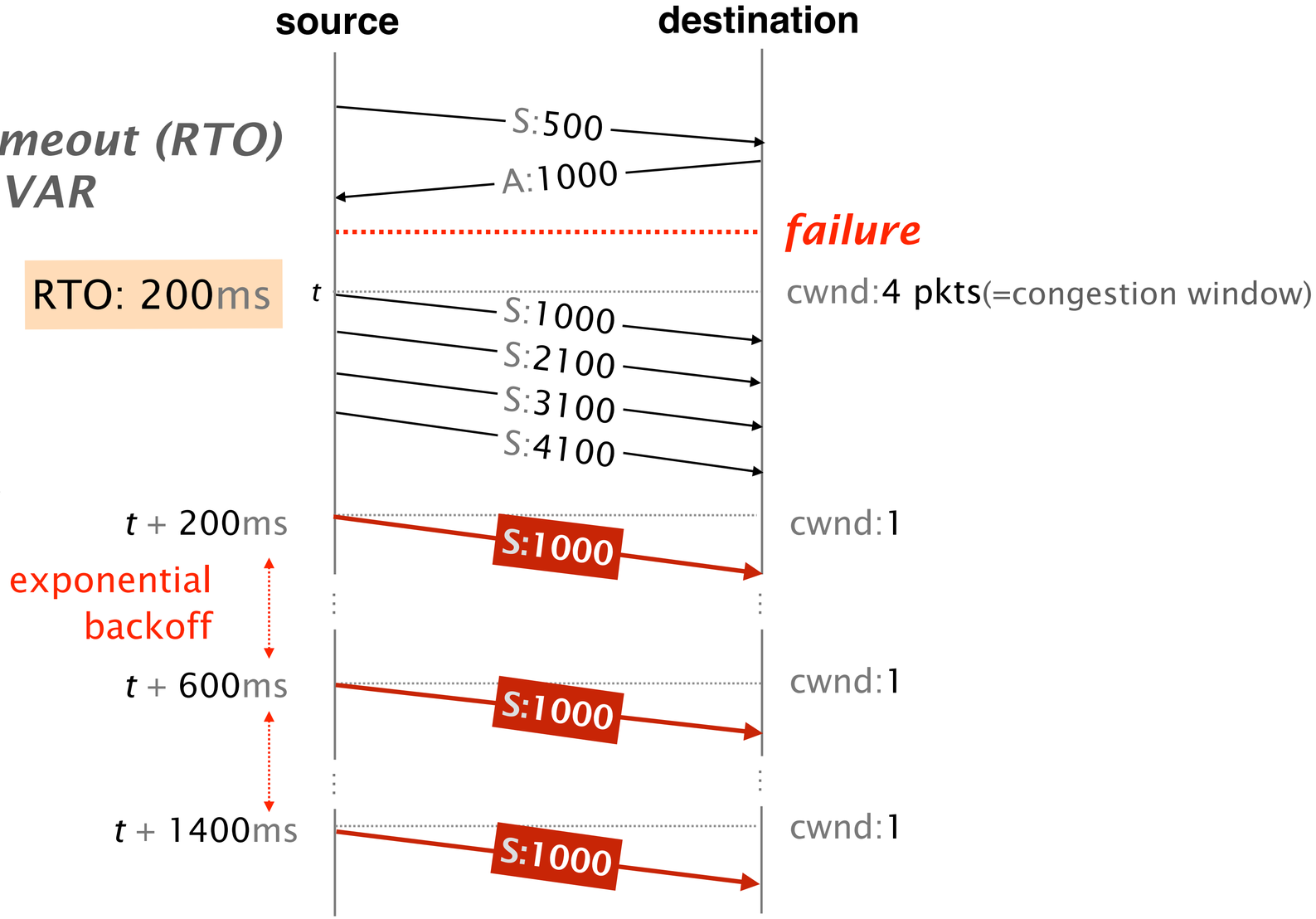
RTO: 200ms



TCP flows exhibit the same behavior upon failures



TCP flows exhibit the same behavior upon failures



When multiple flows experience the same failure

the signal is a wave of retransmissions

When multiple flows experience the same failure

the signal is a wave of retransmissions

We simulated a failure affecting
100k flows with NS3

Same RTT distribution
than in a real trace*

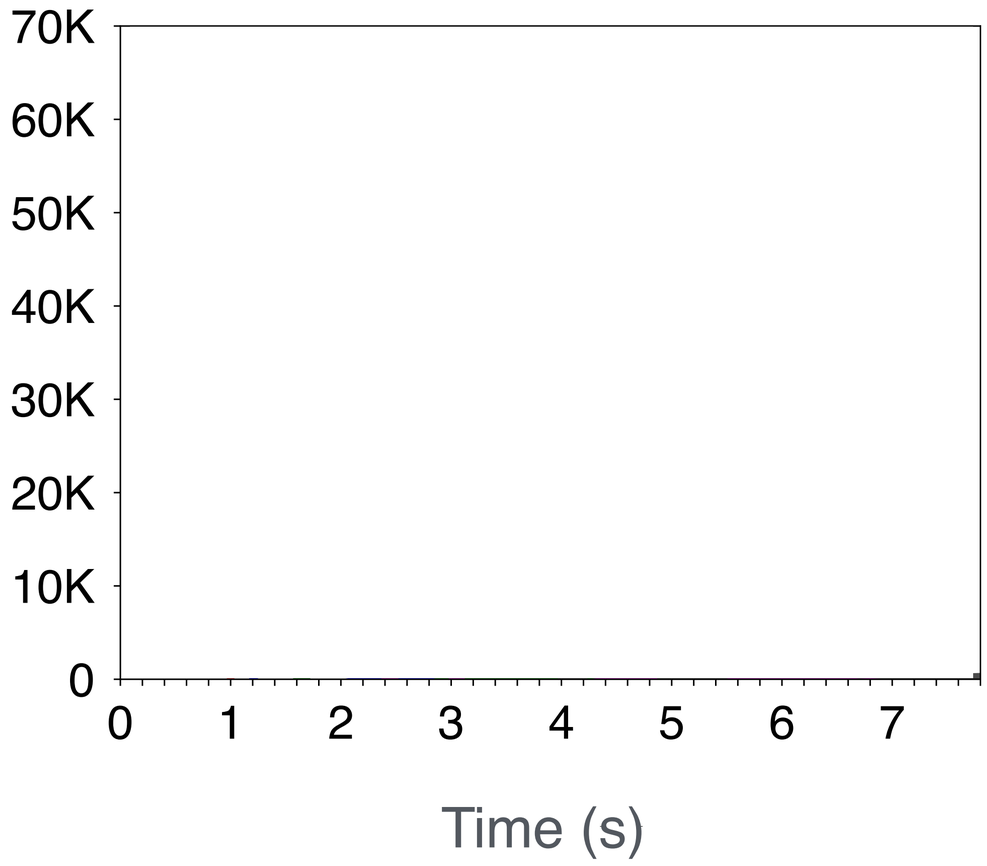
*CAIDA equinix-chicago
direction A, 2015

When multiple flows experience the same failure

the signal is a wave of retransmissions

We simulated a failure affecting 100k flows with NS3

Same RTT distribution than in a real trace*



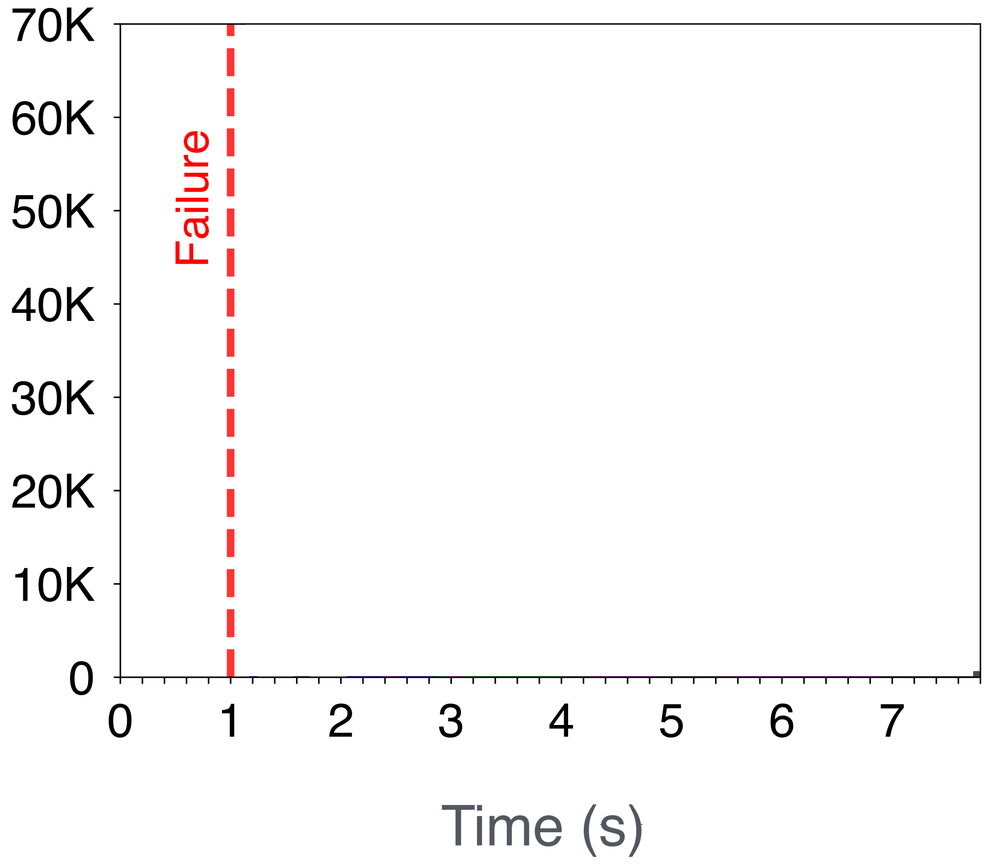
*CAIDA equinix-chicago direction A, 2015

When multiple flows experience the same failure

the signal is a wave of retransmissions

We simulated a failure affecting 100k flows with NS3

Same RTT distribution than in a real trace*



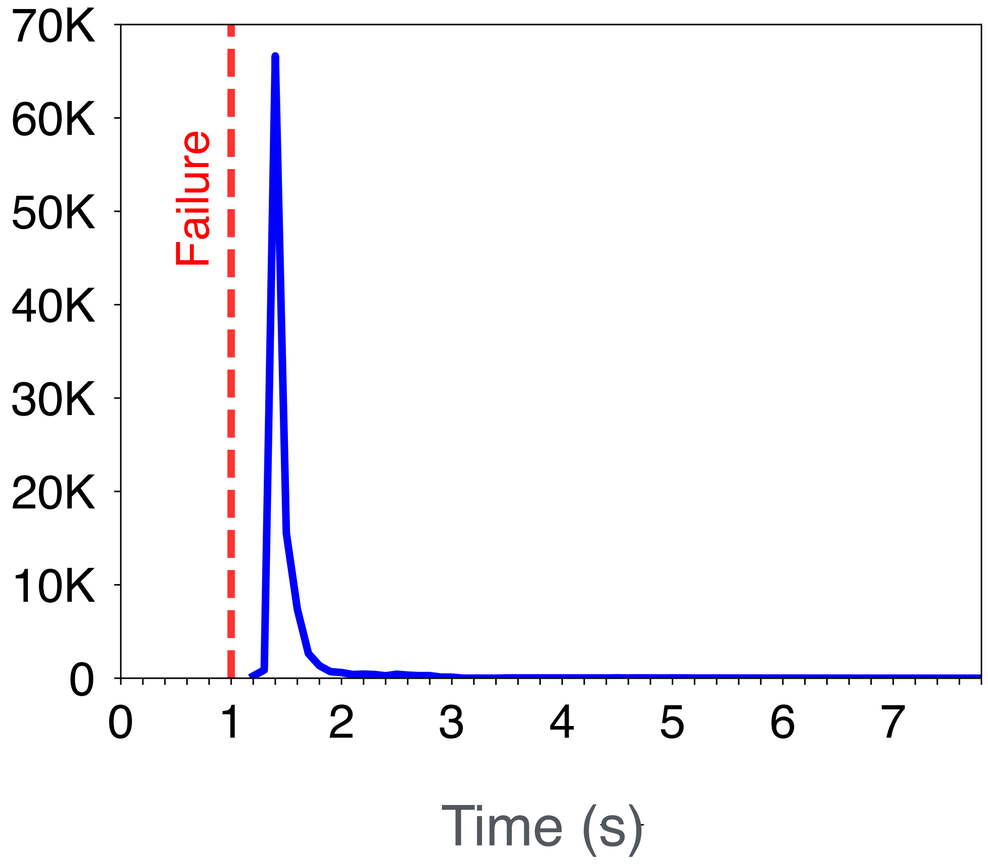
*CAIDA equinix-chicago direction A, 2015

When multiple flows experience the same failure

the signal is a wave of retransmissions

We simulated a failure affecting 100k flows with NS3

Same RTT distribution than in a real trace*



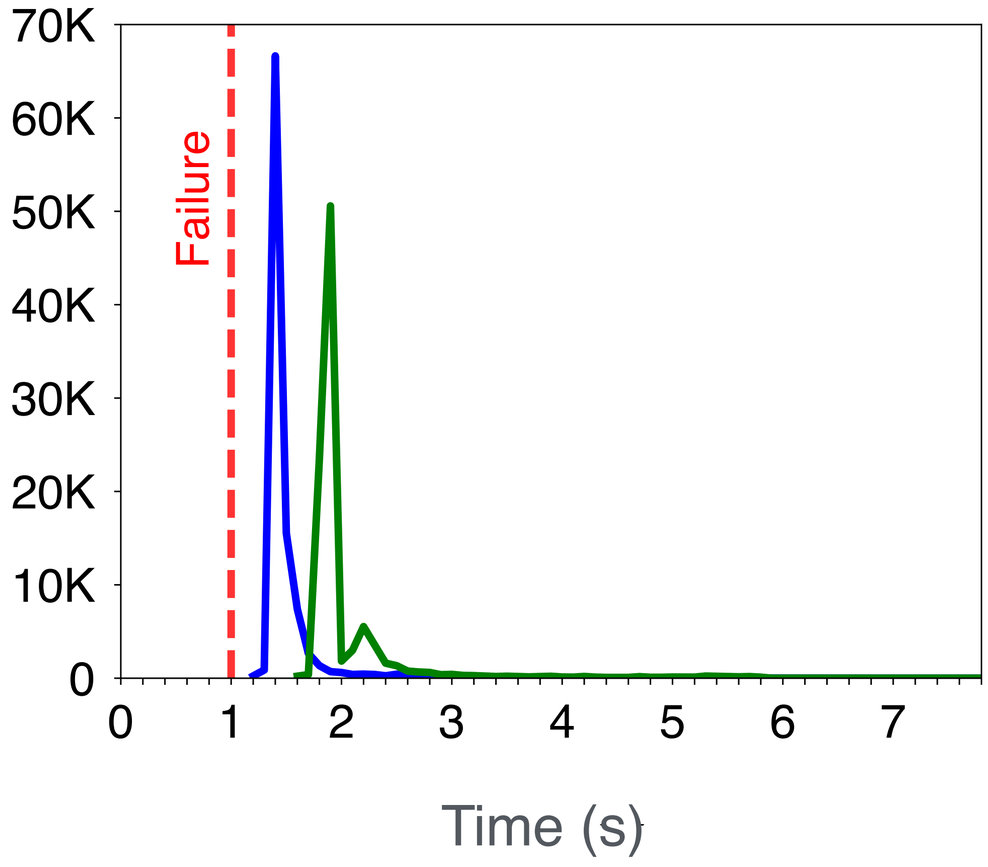
*CAIDA equinix-chicago direction A, 2015

When multiple flows experience the same failure

the signal is a wave of retransmissions

We simulated a failure affecting 100k flows with NS3

Same RTT distribution than in a real trace*



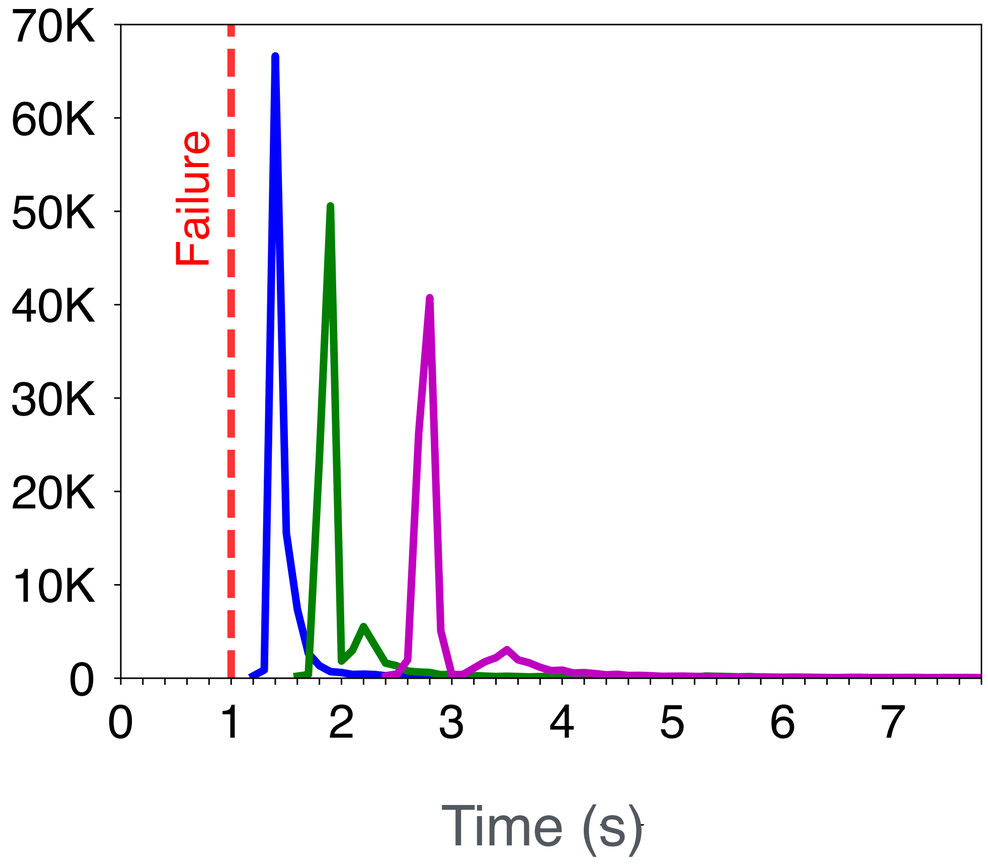
*CAIDA equinix-chicago direction A, 2015

When multiple flows experience the same failure

the signal is a wave of retransmissions

We simulated a failure affecting 100k flows with NS3

Same RTT distribution than in a real trace*



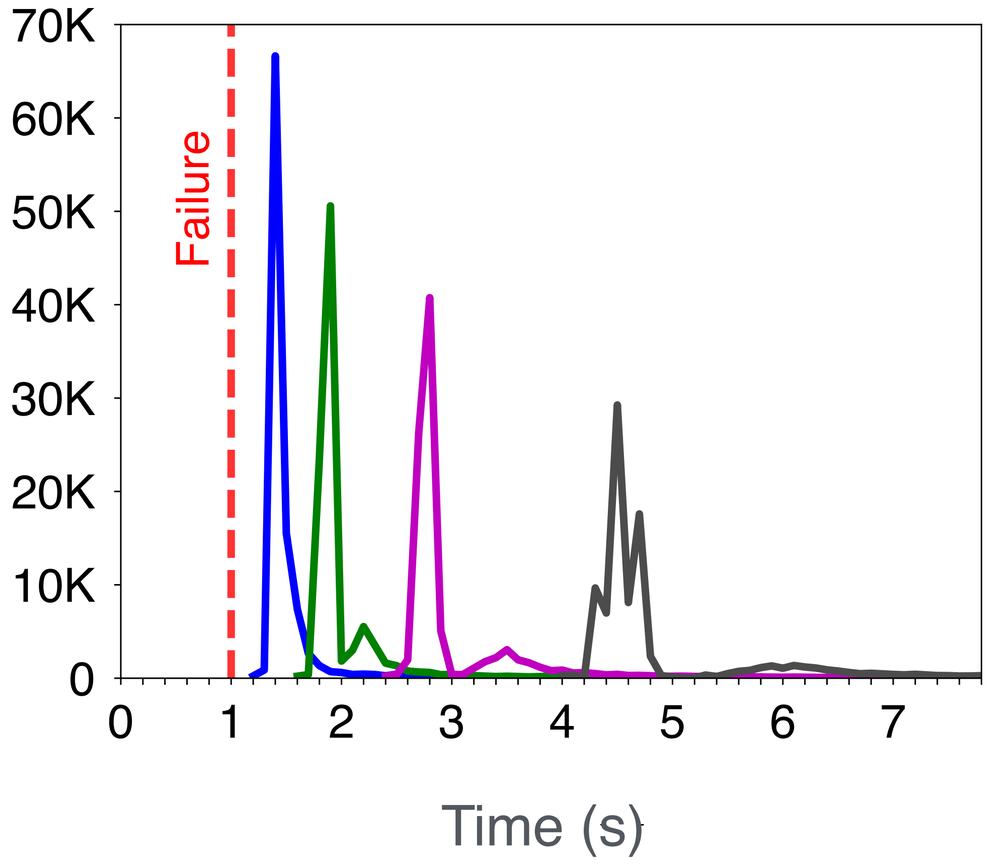
*CAIDA equinix-chicago direction A, 2015

When multiple flows experience the same failure

the signal is a wave of retransmissions

We simulated a failure affecting 100k flows with NS3

Same RTT distribution than in a real trace*



*CAIDA equinix-chicago direction A, 2015

Outline

1. Why and how to use data-plane signals for fast rerouting
2. ***Blink*** infers more than 80% of the failures, often within 1s
3. ***Blink*** quickly reroutes traffic to working backup paths
4. ***Blink*** works in practice, on existing devices

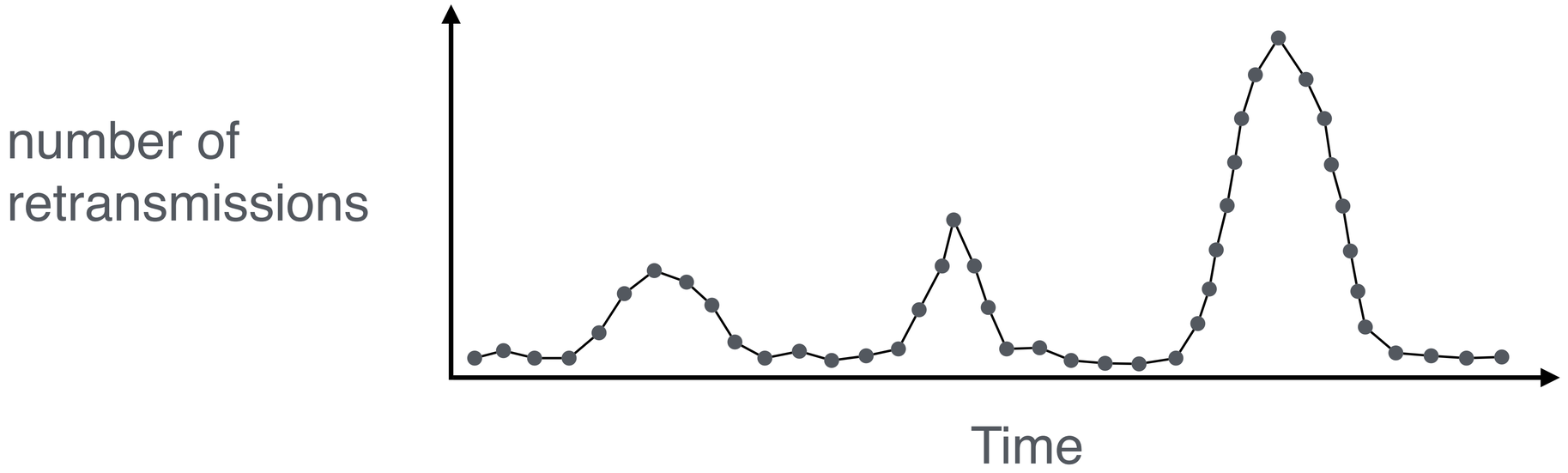
To detect failures, ***Blink*** looks at TCP retransmissions

To detect failures, ***Blink*** looks at TCP retransmissions

Problem: TCP retransmissions can be unrelated to a failure (*i.e.,* noise)

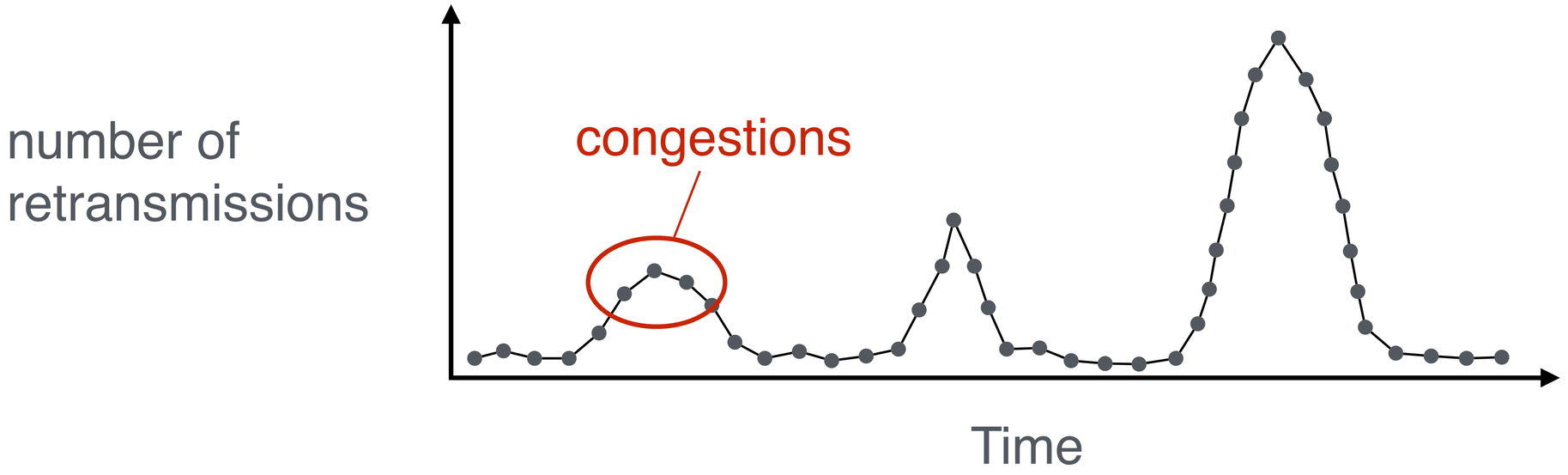
To detect failures, ***Blink*** looks at TCP retransmissions

Problem: TCP retransmissions can be unrelated to a failure (*i.e.*, noise)



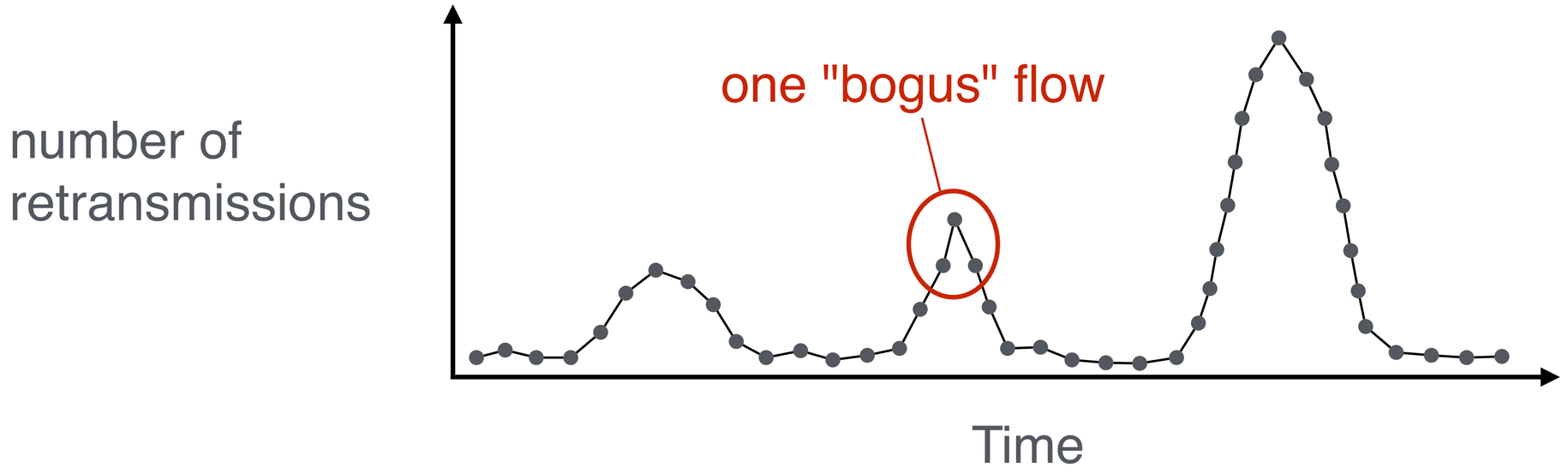
To detect failures, ***Blink*** looks at TCP retransmissions

Problem: TCP retransmissions can be unrelated to a failure (*i.e.*, noise)



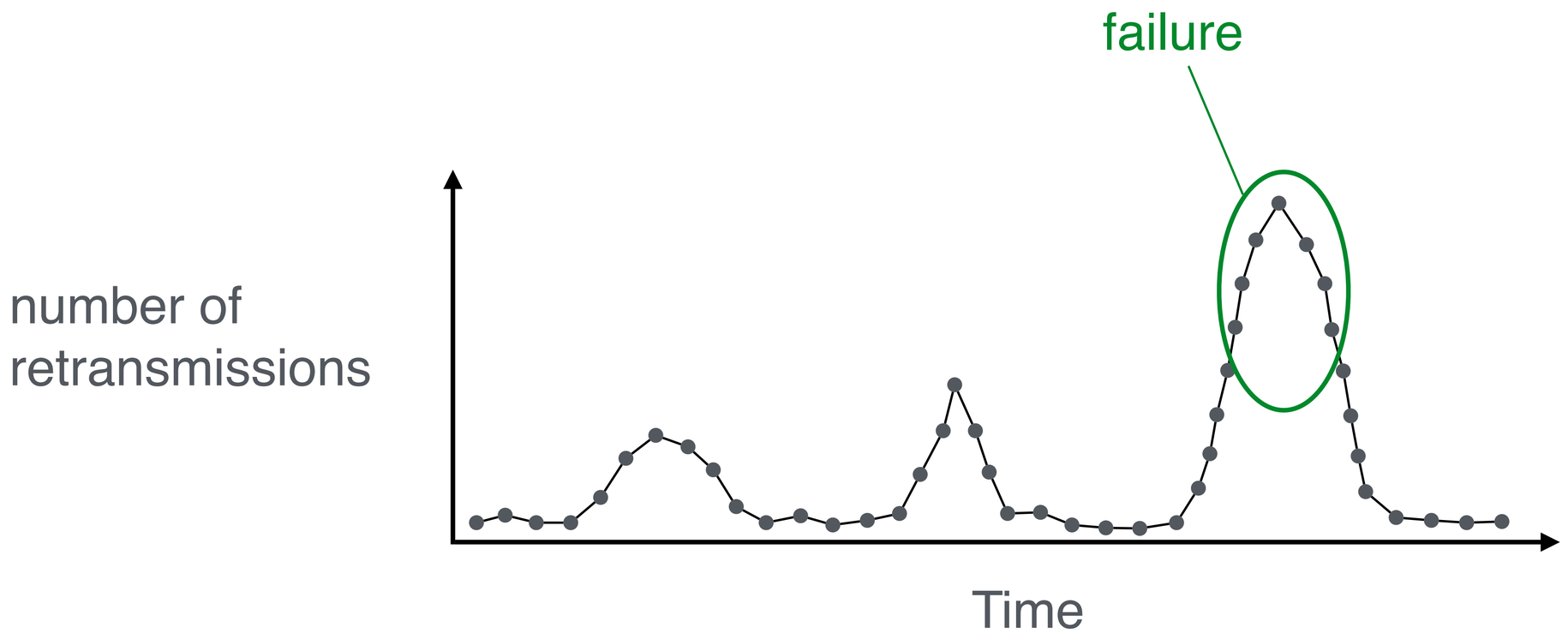
To detect failures, ***Blink*** looks at TCP retransmissions

Problem: TCP retransmissions can be unrelated to a failure (*i.e.*, noise)



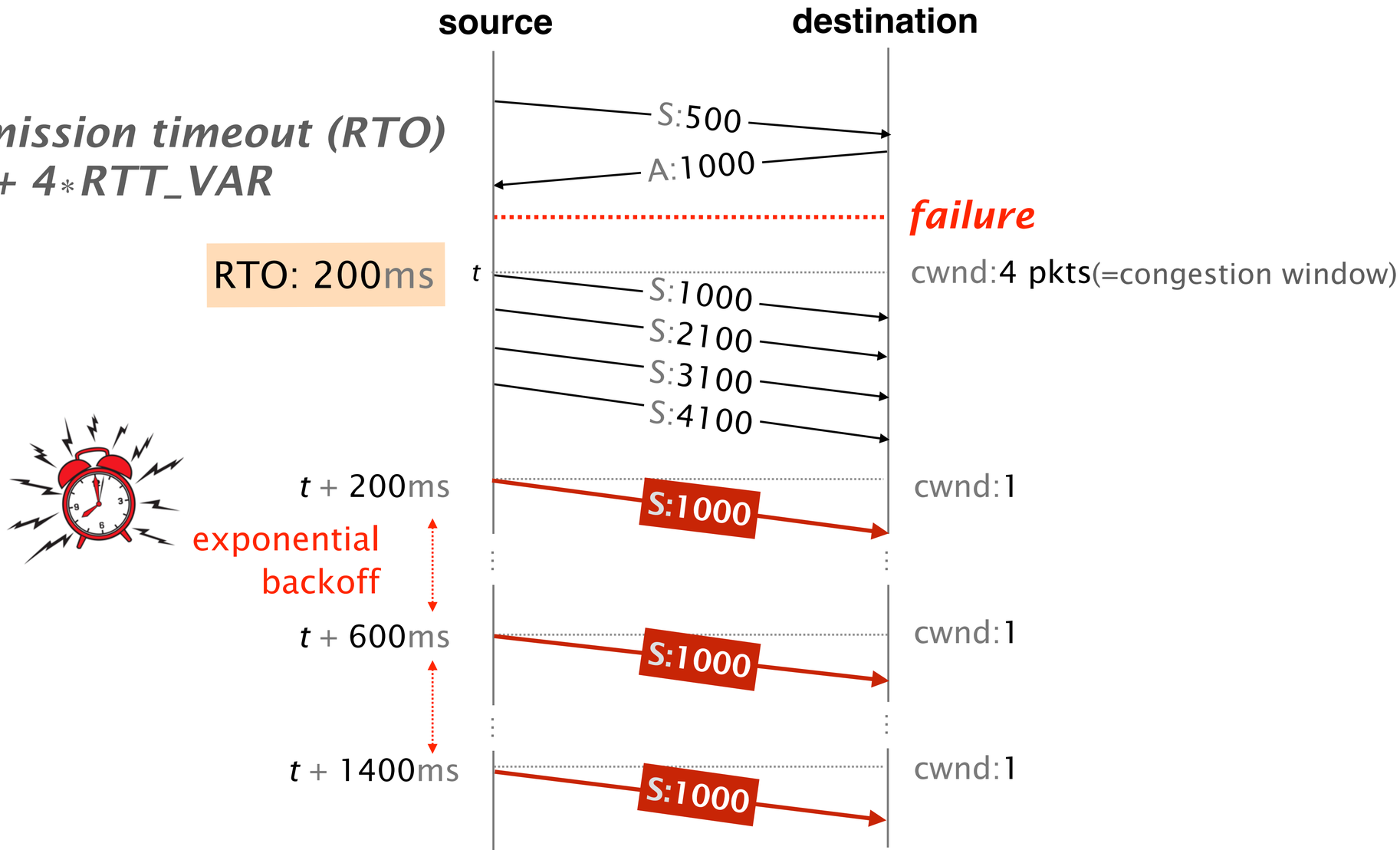
To detect failures, ***Blink*** looks at TCP retransmissions

Problem: TCP retransmissions can be unrelated to a failure (*i.e.*, noise)



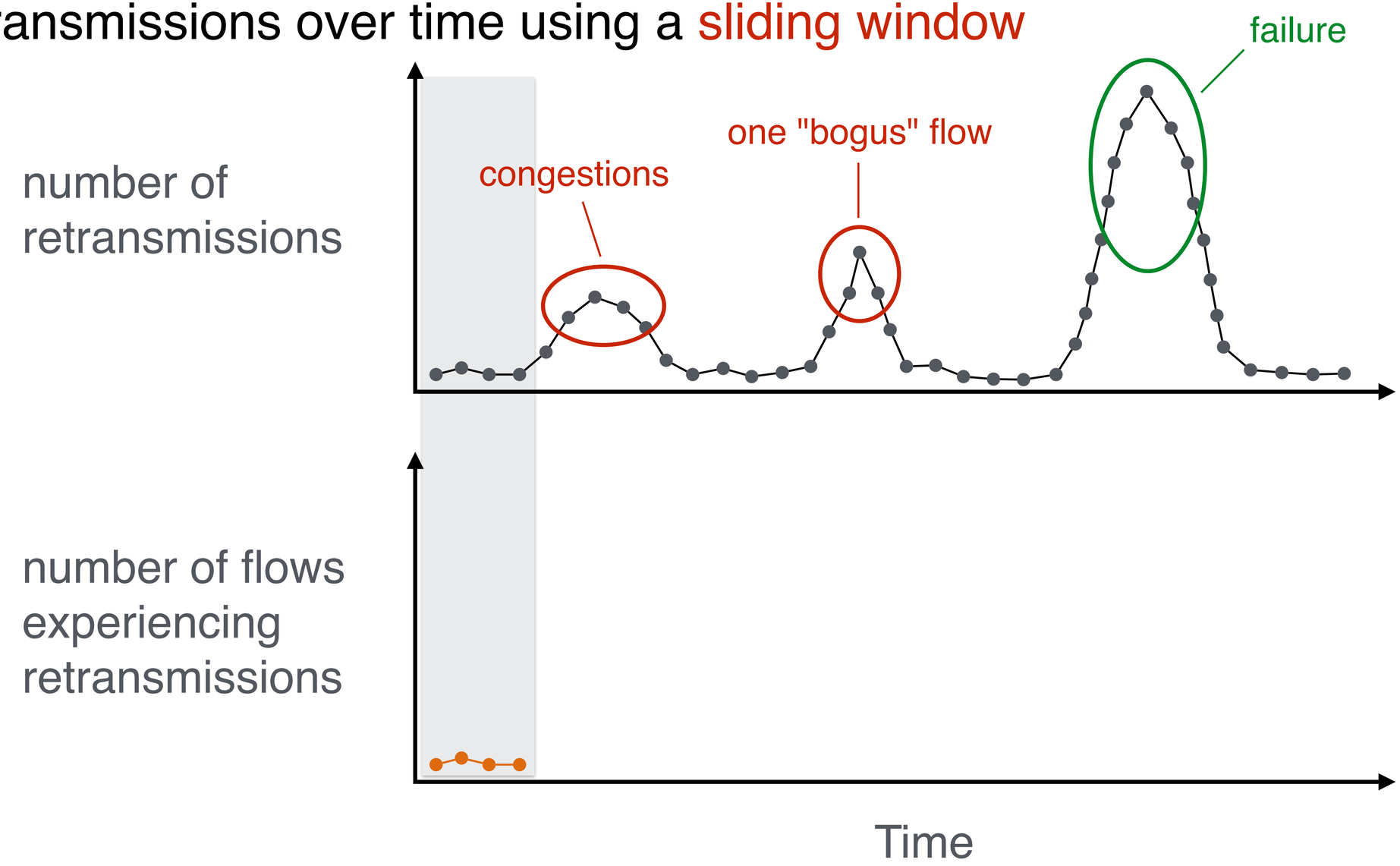
Solution #1: ***Blink*** looks at consecutive packets with the same sequence number

Solution #1: ***Blink*** looks at consecutive packets with the same sequence number

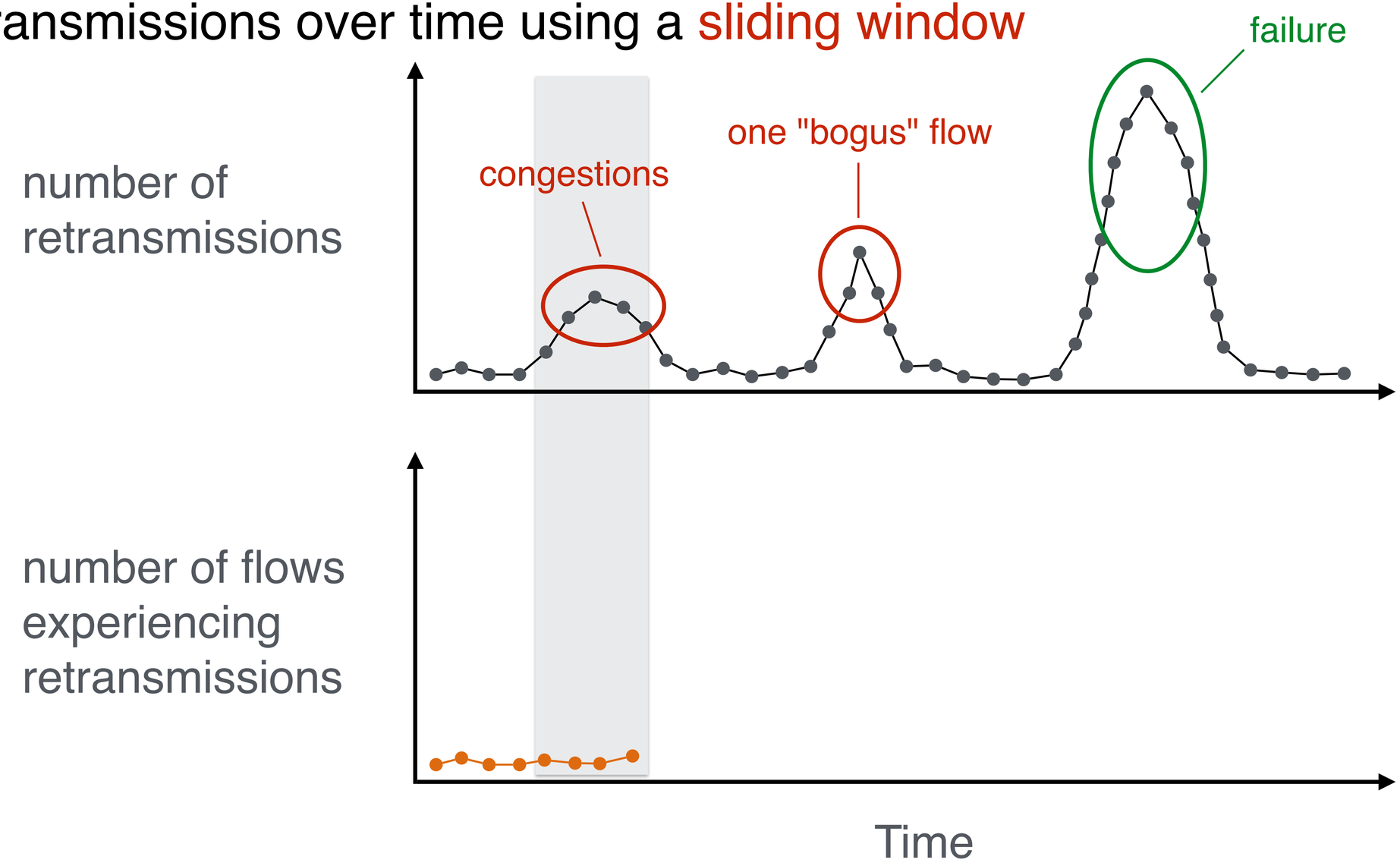


Solution #2: *Blink* monitors the number of flows experiencing retransmissions over time using a sliding window

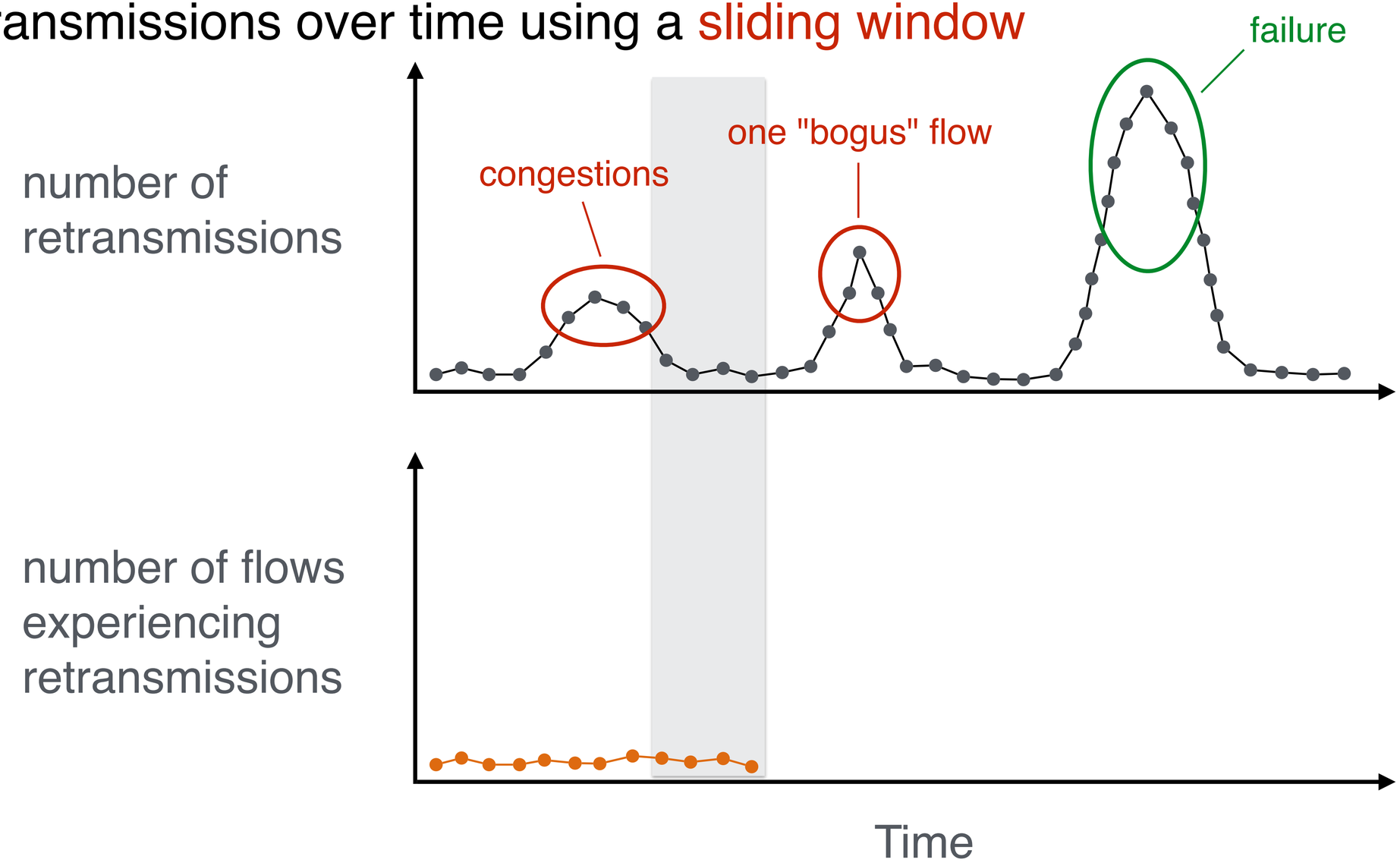
Solution #2: ***Blink*** monitors the number of flows experiencing retransmissions over time using a sliding window



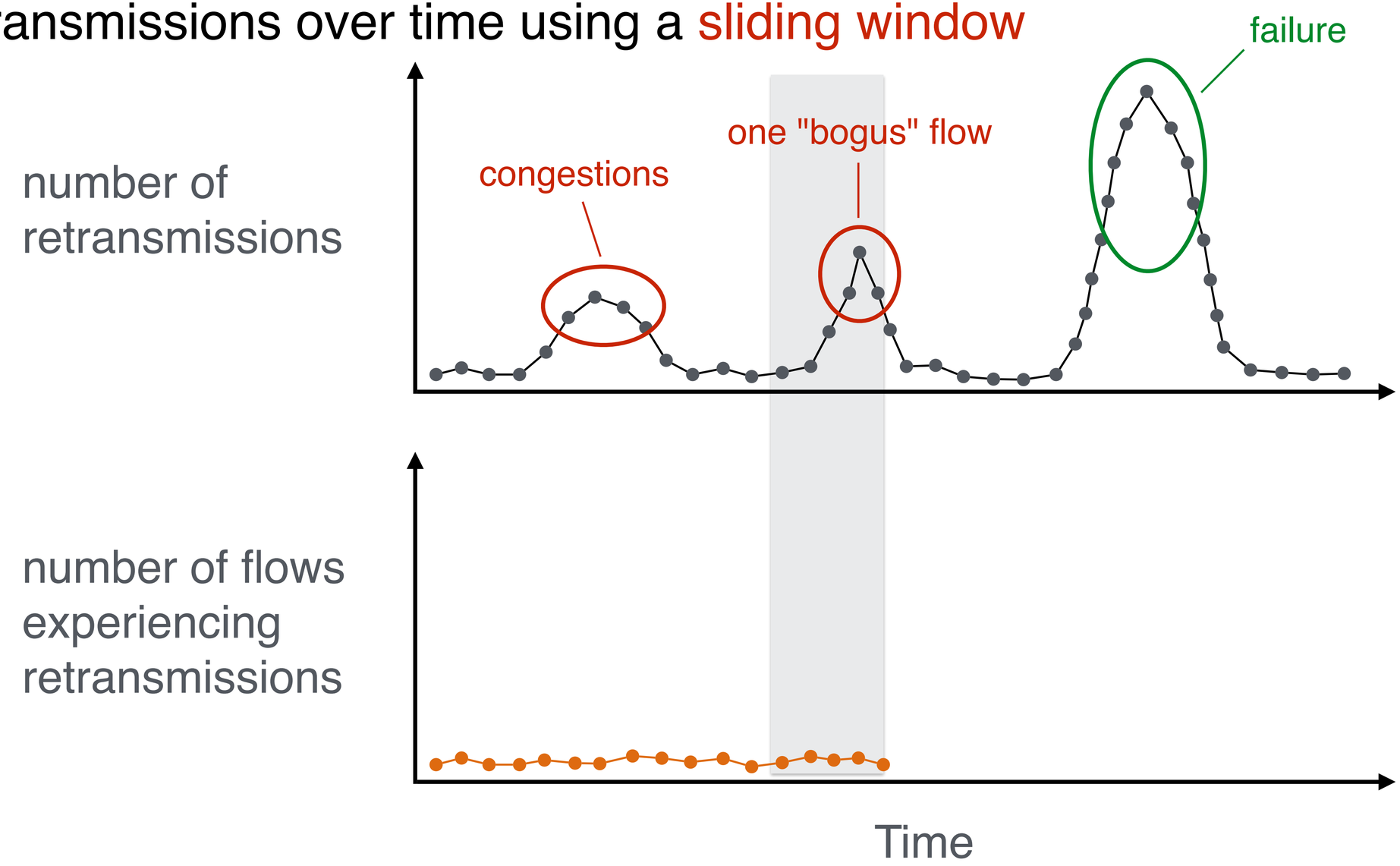
Solution #2: ***Blink*** monitors the number of flows experiencing retransmissions over time using a sliding window



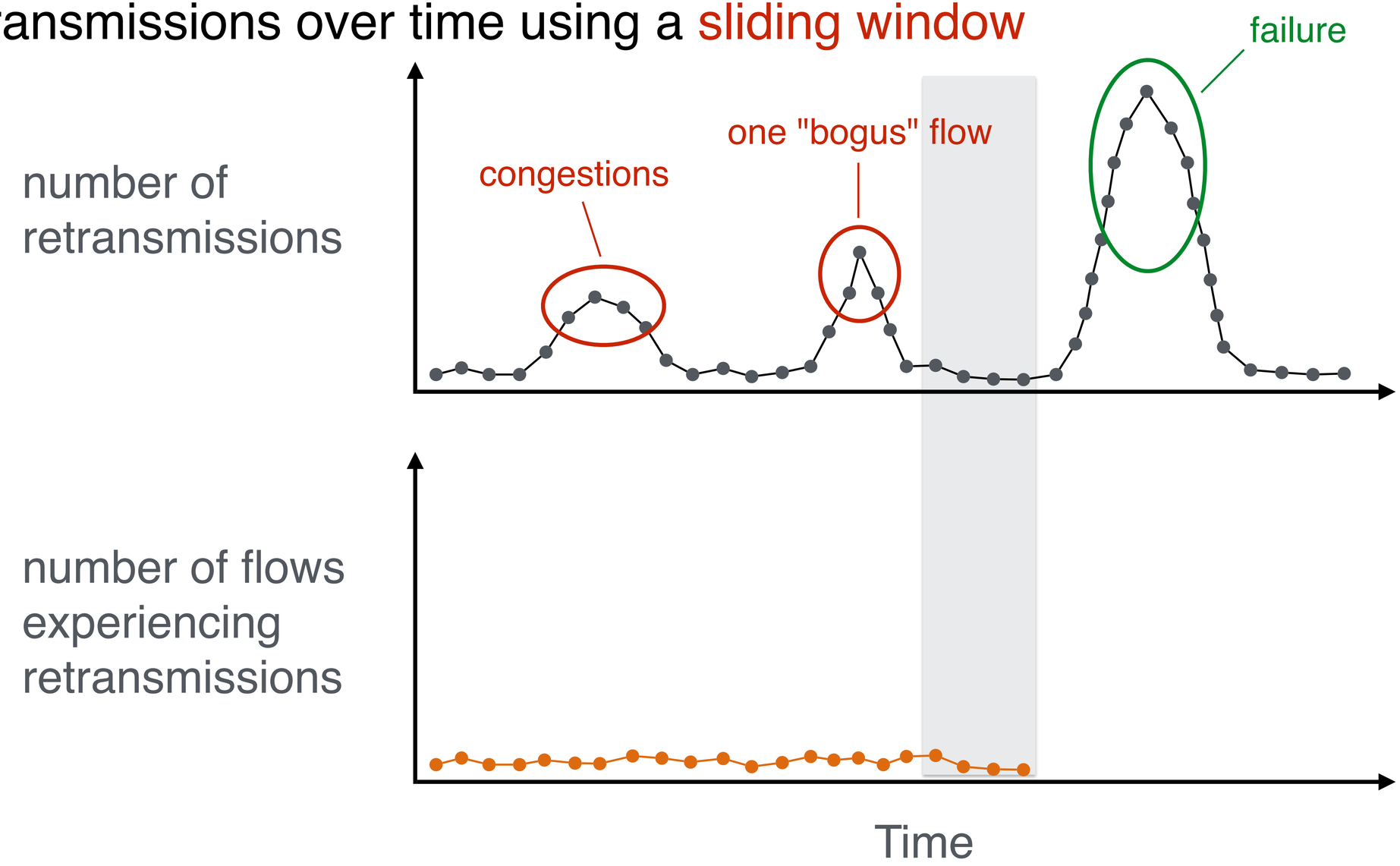
Solution #2: ***Blink*** monitors the number of flows experiencing retransmissions over time using a sliding window



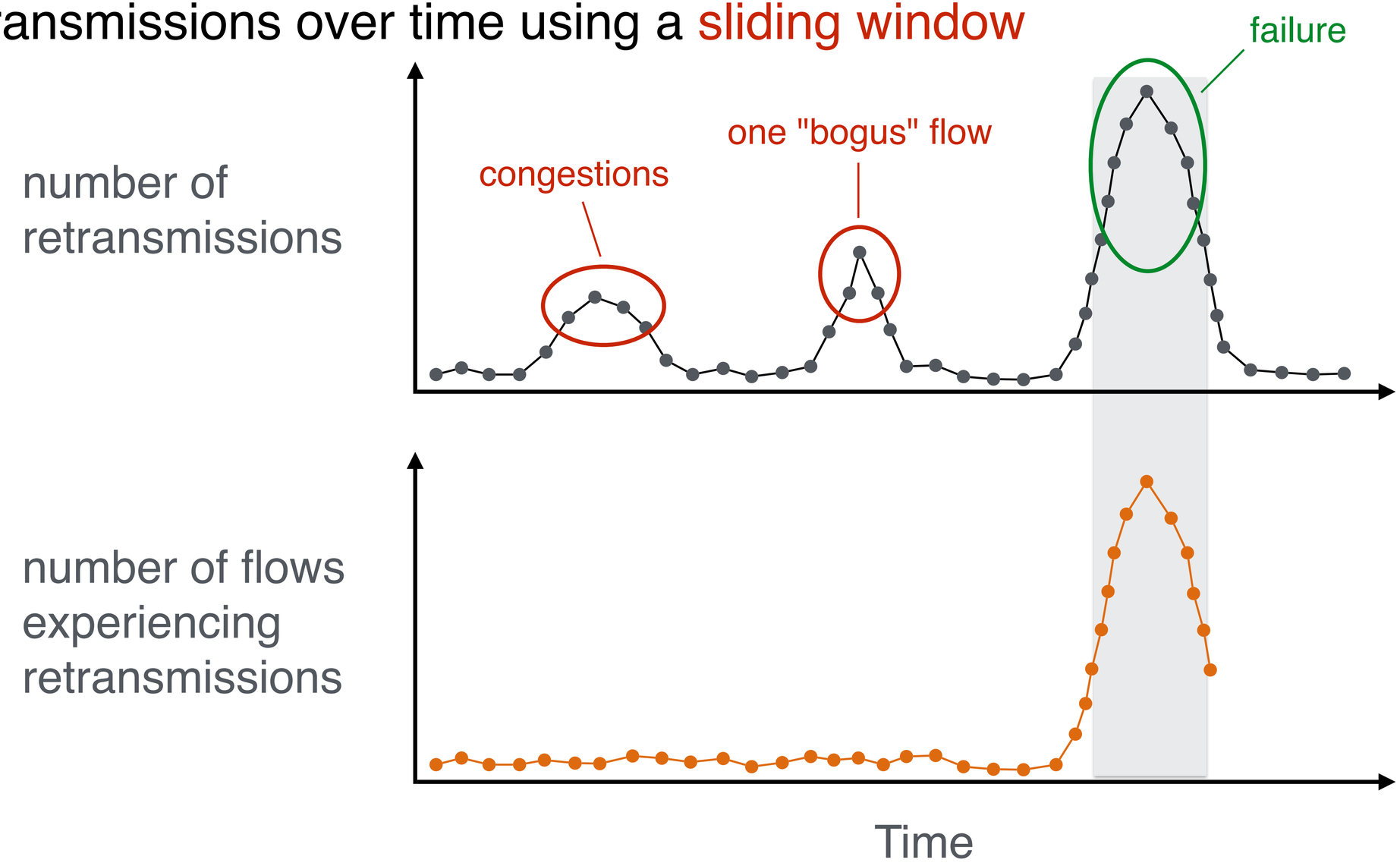
Solution #2: ***Blink*** monitors the number of flows experiencing retransmissions over time using a sliding window



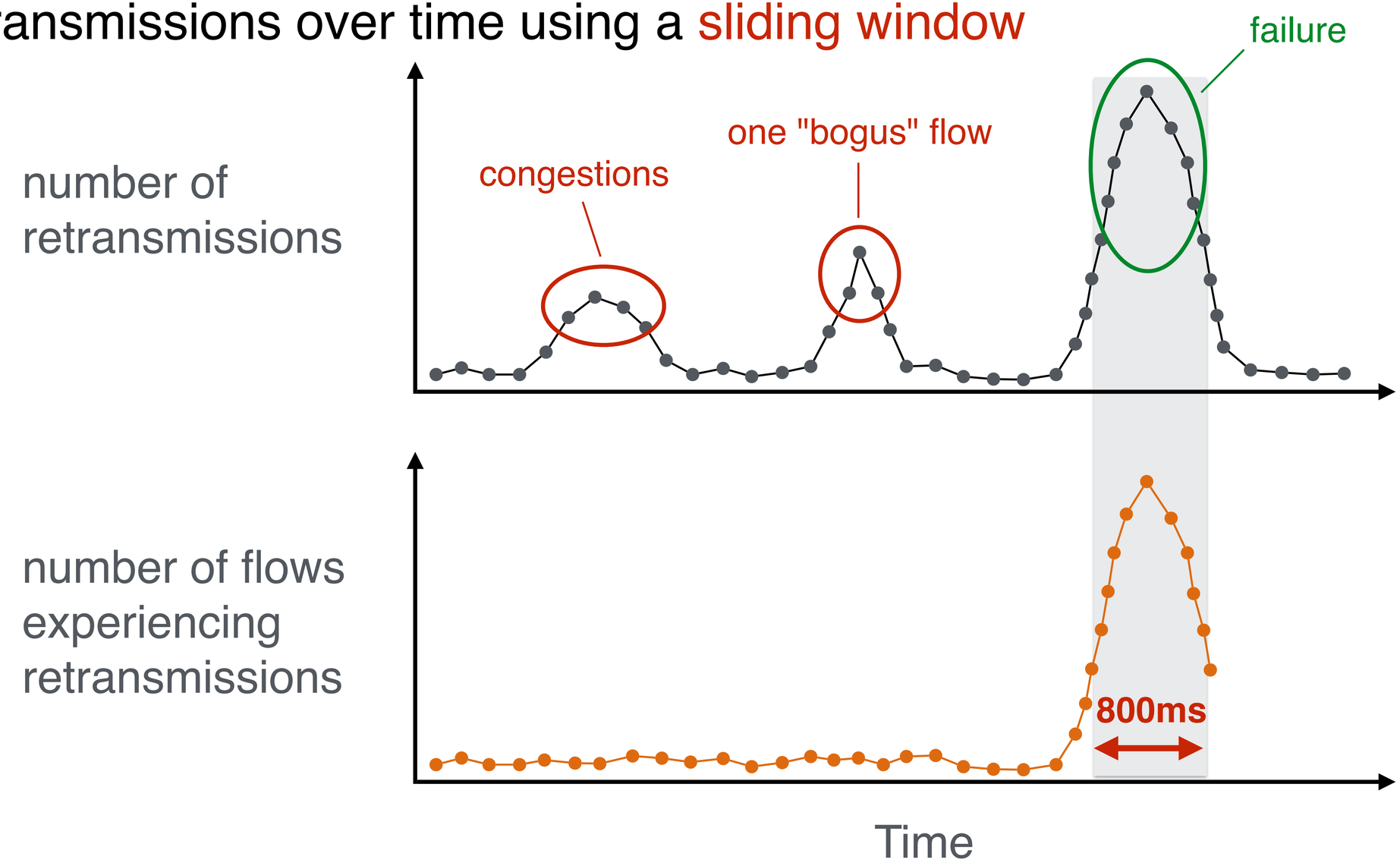
Solution #2: ***Blink*** monitors the number of flows experiencing retransmissions over time using a sliding window



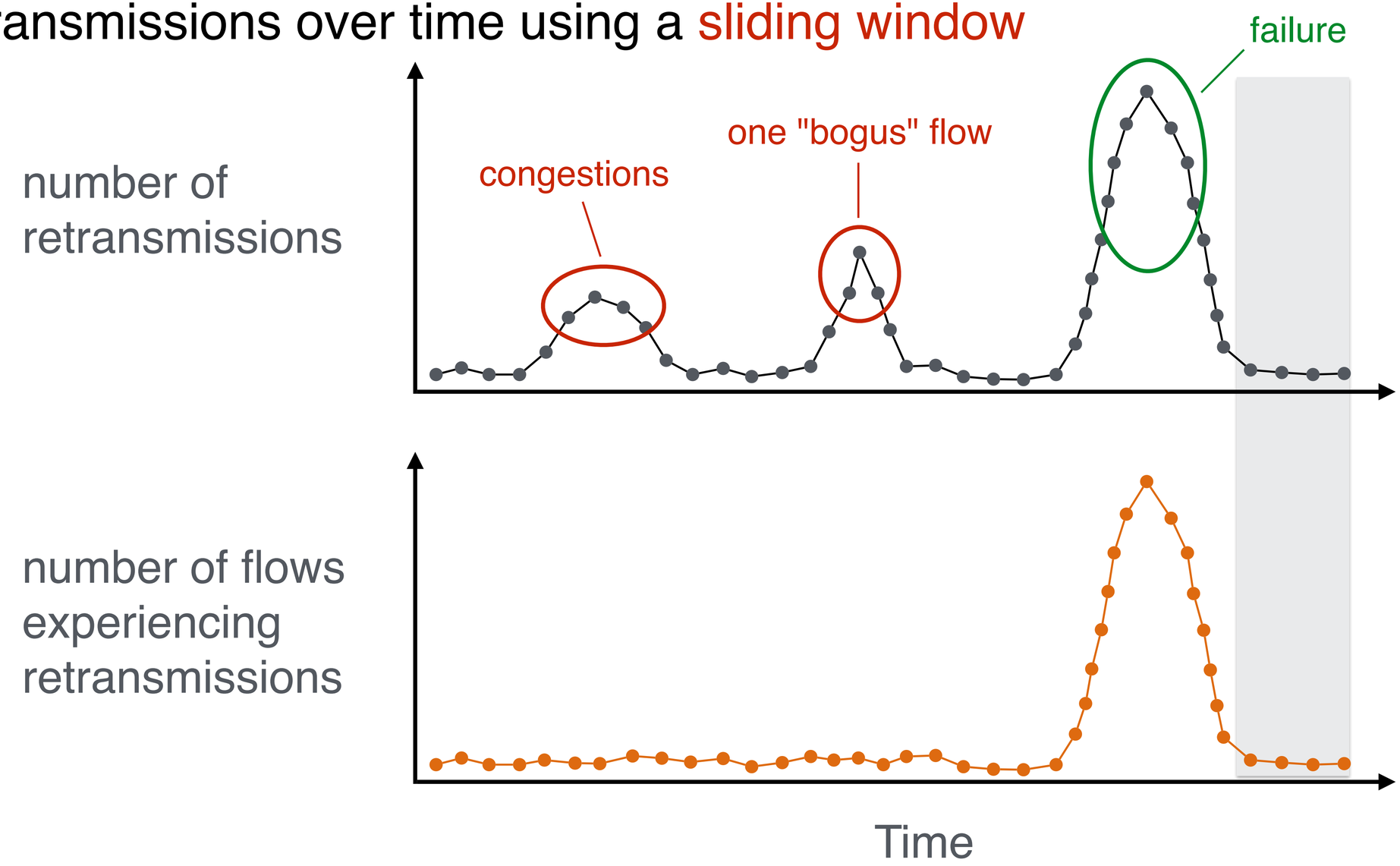
Solution #2: ***Blink*** monitors the number of flows experiencing retransmissions over time using a sliding window



Solution #2: ***Blink*** monitors the number of flows experiencing retransmissions over time using a sliding window



Solution #2: ***Blink*** monitors the number of flows experiencing retransmissions over time using a sliding window



Blink is intended to run in programmable switches

Blink is intended to run in programmable switches

Problem: those switches have very limited resources

Solution #1: ***Blink*** focuses on the popular prefixes,

i.e., the ones that attract data traffic

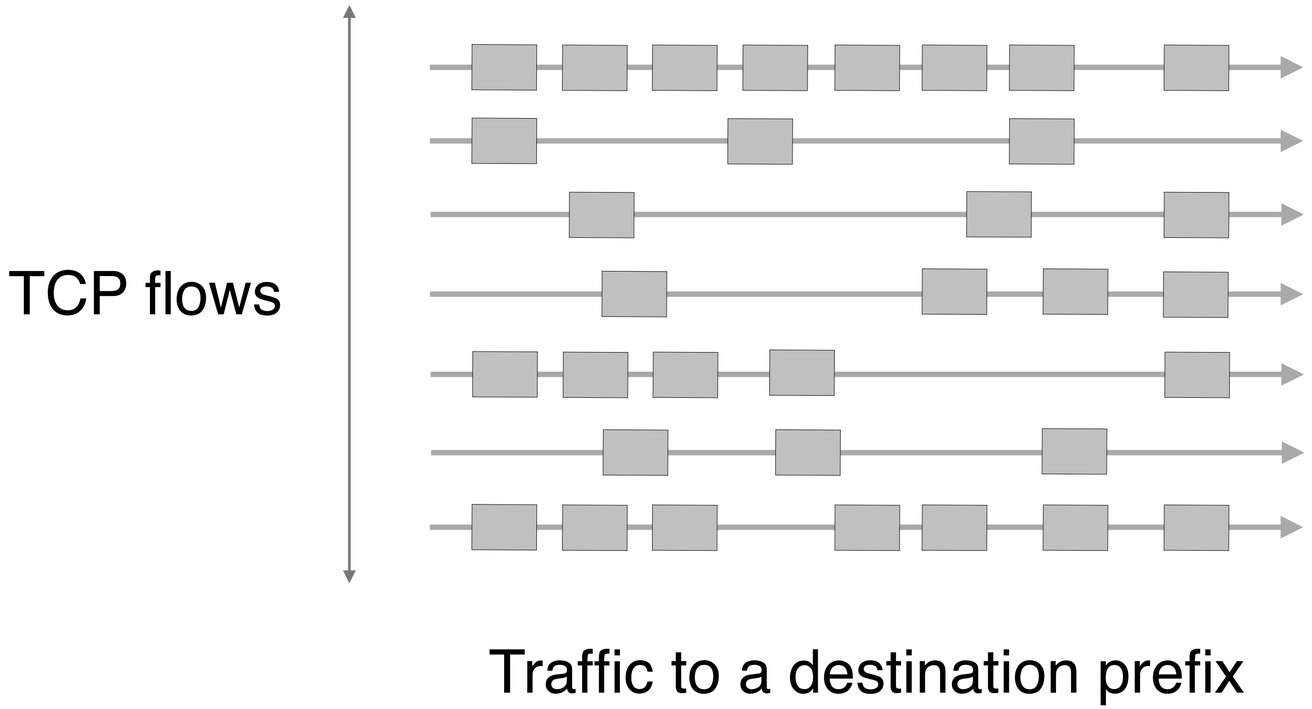
Solution #1: ***Blink*** focuses on the popular prefixes,

i.e., the ones that attract data traffic

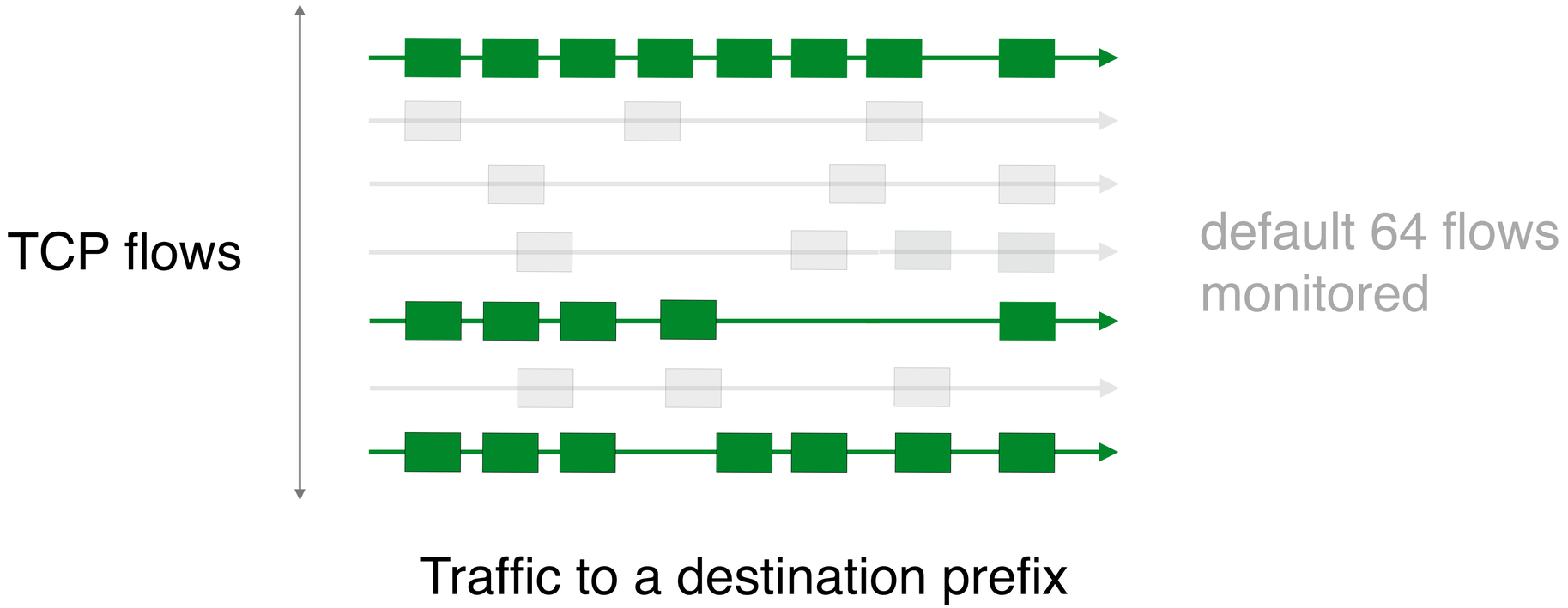
As Internet traffic follows a Zipf-like distribution* (1k pref. account for >50%),
Blink covers the vast majority of the Internet traffic

*Sarra et al. Leveraging Zipf's Law for Traffic offloading
ACM CCR, 2012

Solution #2: ***Blink*** monitors a sample of the flows for each monitored prefix



Solution #2: ***Blink*** monitors a sample of the flows for each monitored prefix



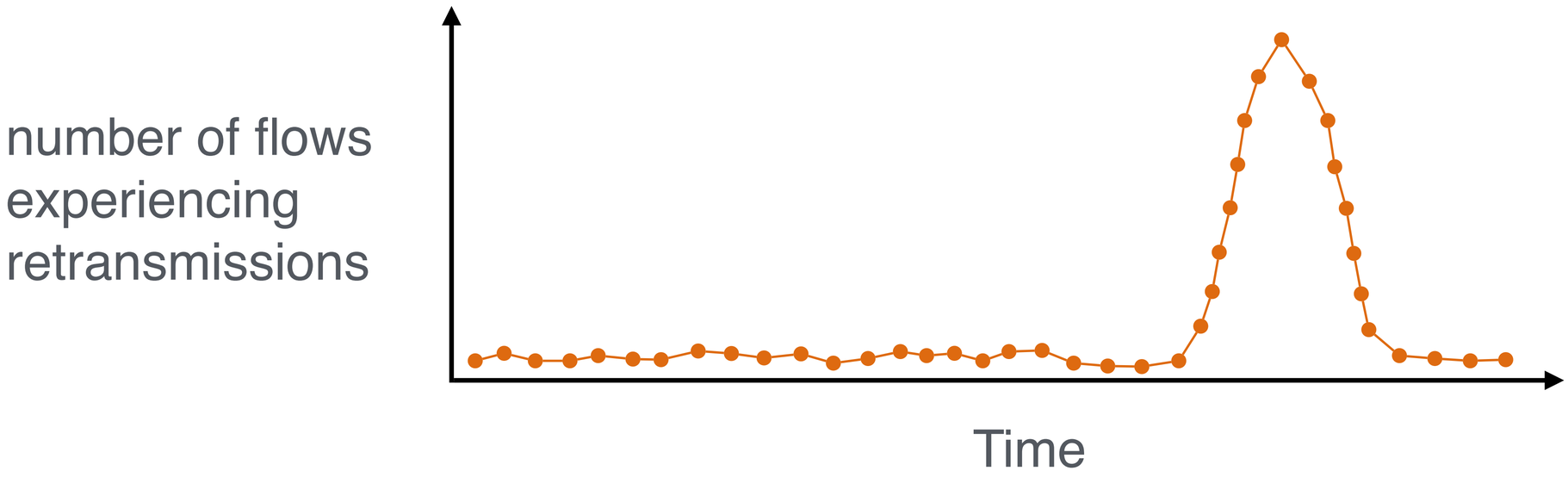
To monitor active flows, ***Blink*** evicts a flow from the sample if it does not send a packet for a given time (default 2s)

To monitor active flows, ***Blink*** evicts a flow from the sample
if it does not send a packet for a given time (default 2s)

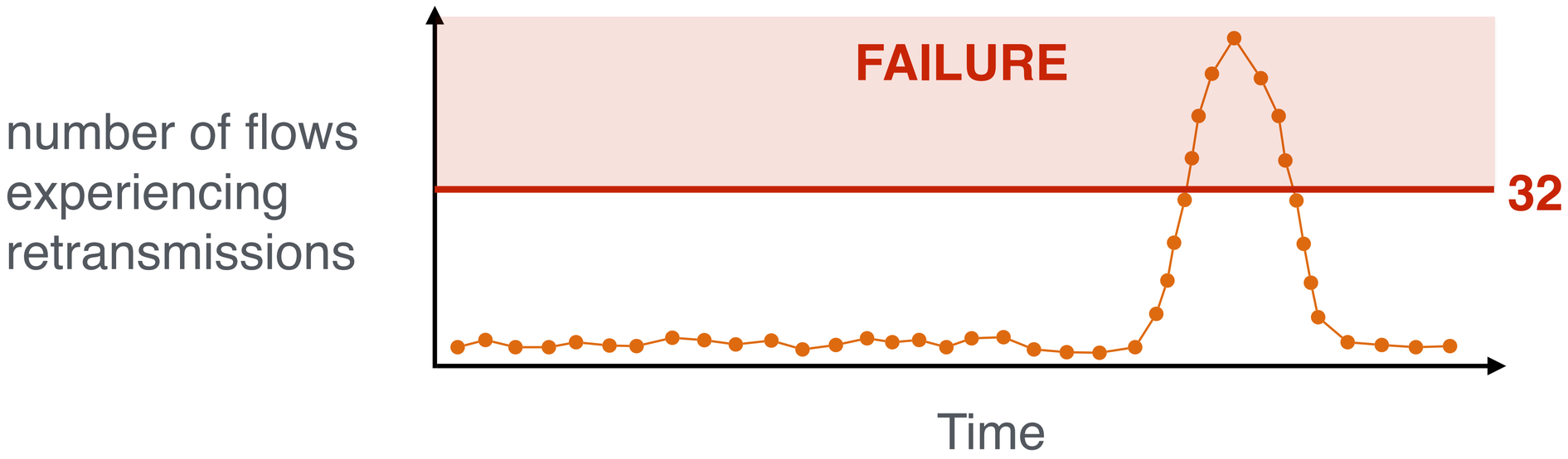
and selects a new one in a
first-seen, first-selected manner

Blink infers a failure for a prefix when the majority of the monitored flows experience retransmissions

Blink infers a failure for a prefix when the majority of the monitored flows experience retransmissions



Blink infers a failure for a prefix when the majority of the monitored flows experience retransmissions



We evaluated ***Blink*** failure inference using 15 real traces,

13 from CAIDA, 2 from MAWI, covering a total of 15.8 hours

We evaluated ***Blink*** failure inference using 15 real traces,

13 from CAIDA, 2 from MAWI, covering a total of 15.8 hours

We are interested in:

Accuracy: True Positive Rate vs False Positive Rate

Speed: How long does Blink take to infer failures

As we do not have ground truth, we generated synthetic traces following the traffic characteristics extracted from the real traces

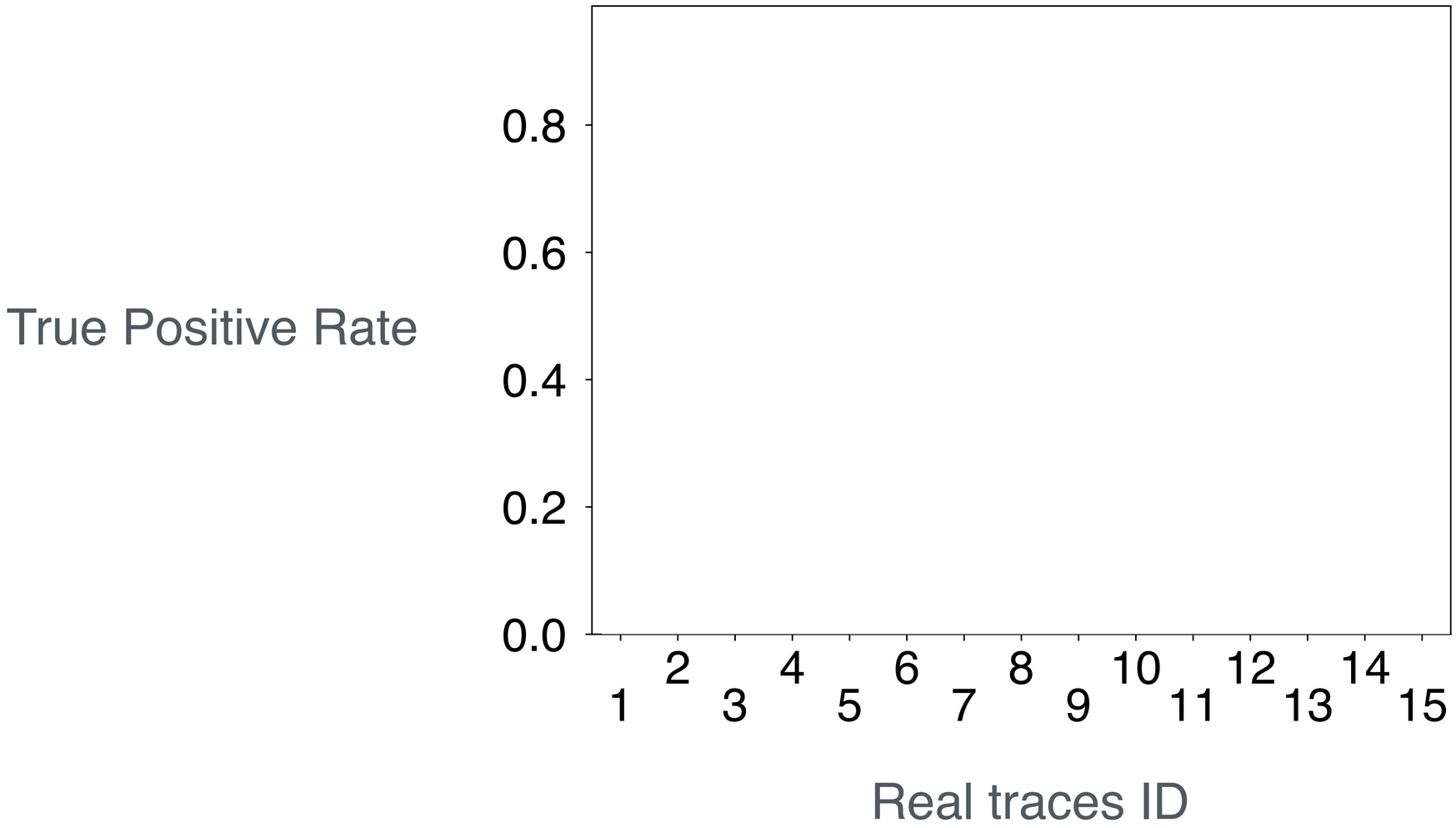
As we do not have ground truth, we generated synthetic traces following the traffic characteristics extracted from the real traces

Step #1 - We extracted the RTT, Packet rate, Flow duration from the real traces

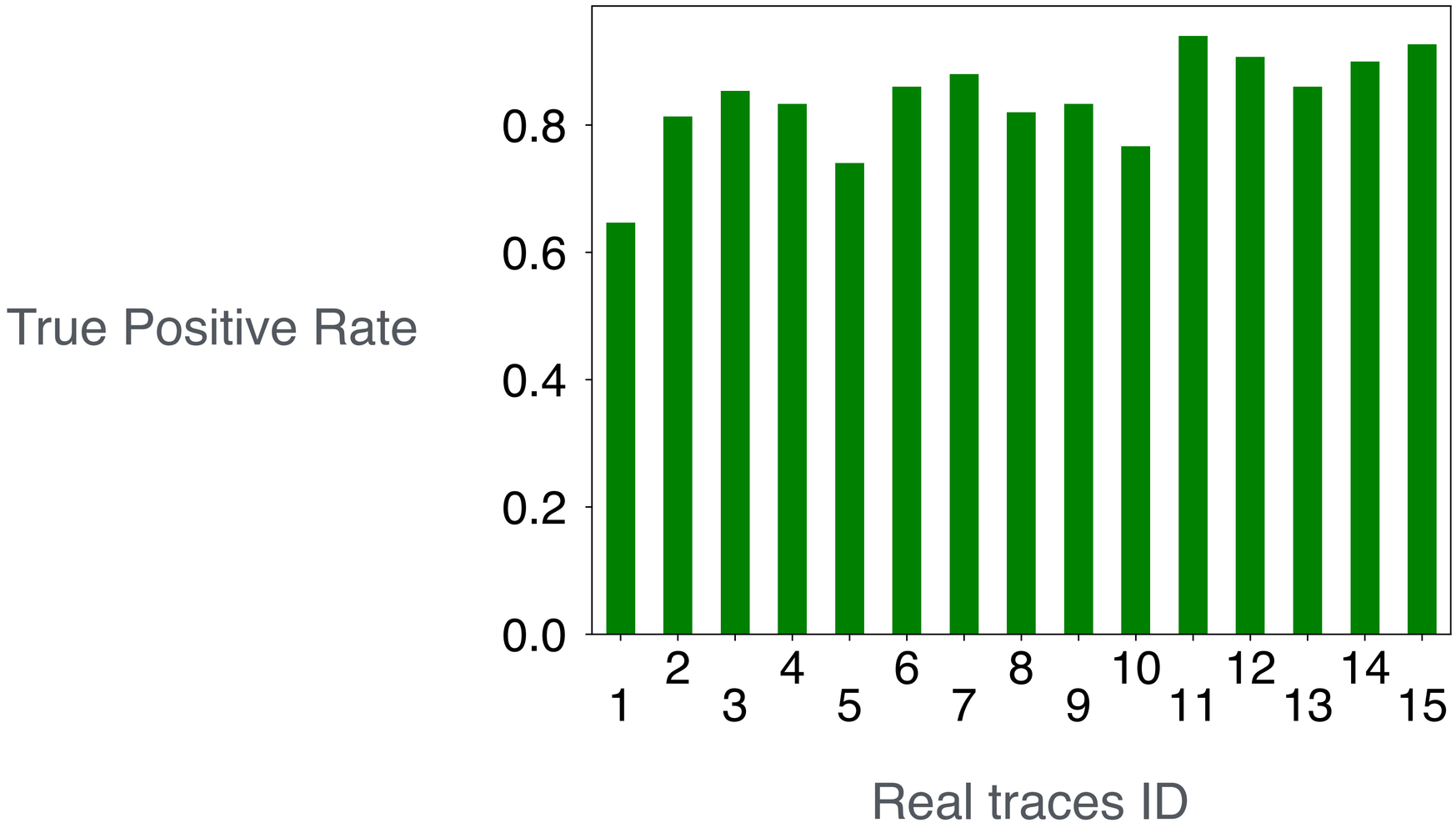
Step #2 - We used NS3 to replay these flows and simulate a failure

Step #3 - We ran a Python-based version of ***Blink*** on the resulting traces

Blink failure inference accuracy is above 80% for 13 real traces out of 15



Blink failure inference accuracy is above 80% for 13 real traces out of 15



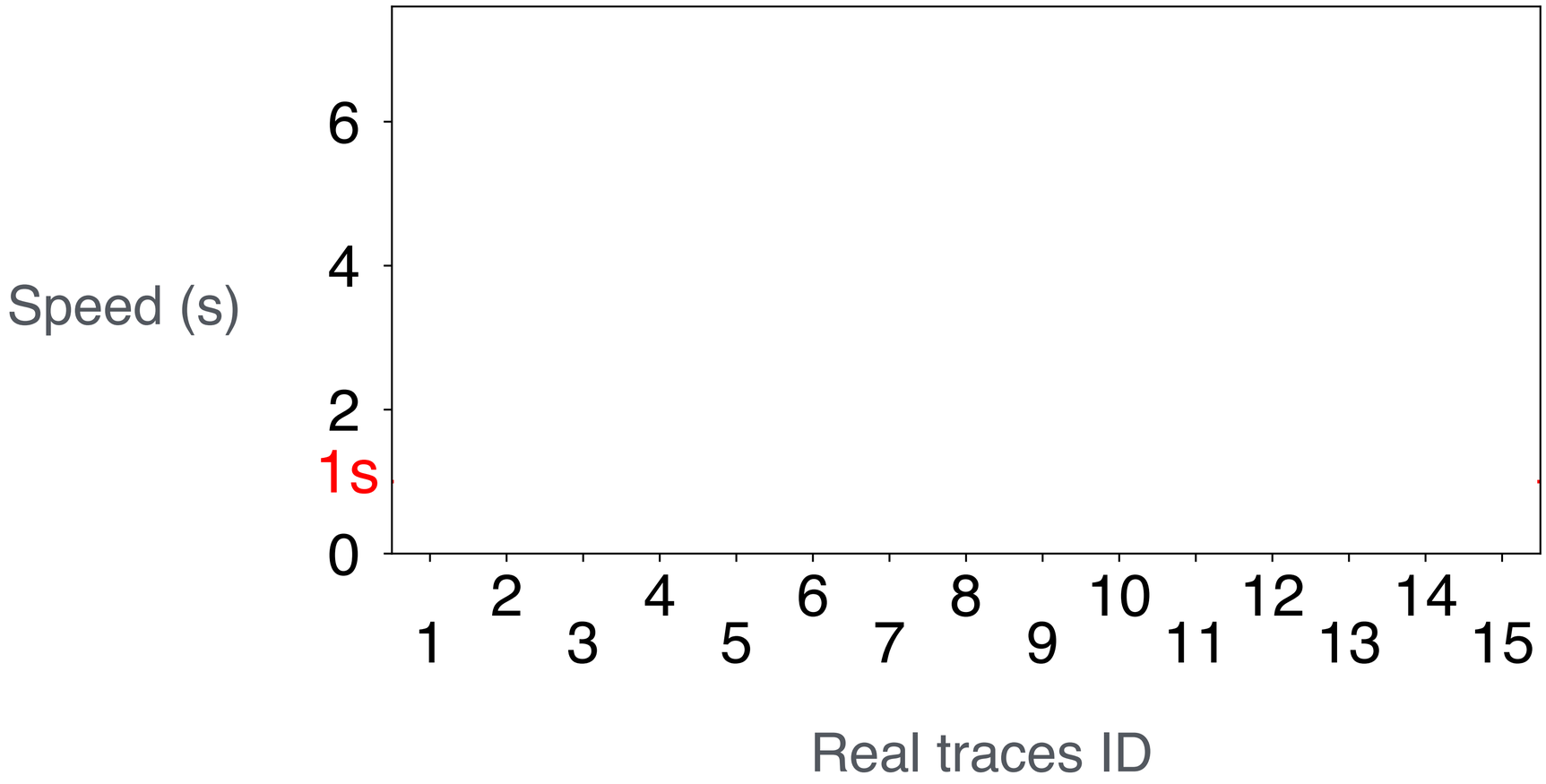
Blink avoids incorrectly inferring failures when packet loss is below 4%

packet loss %	1	2	3	4	5	…	8	9
False Positive Rate								

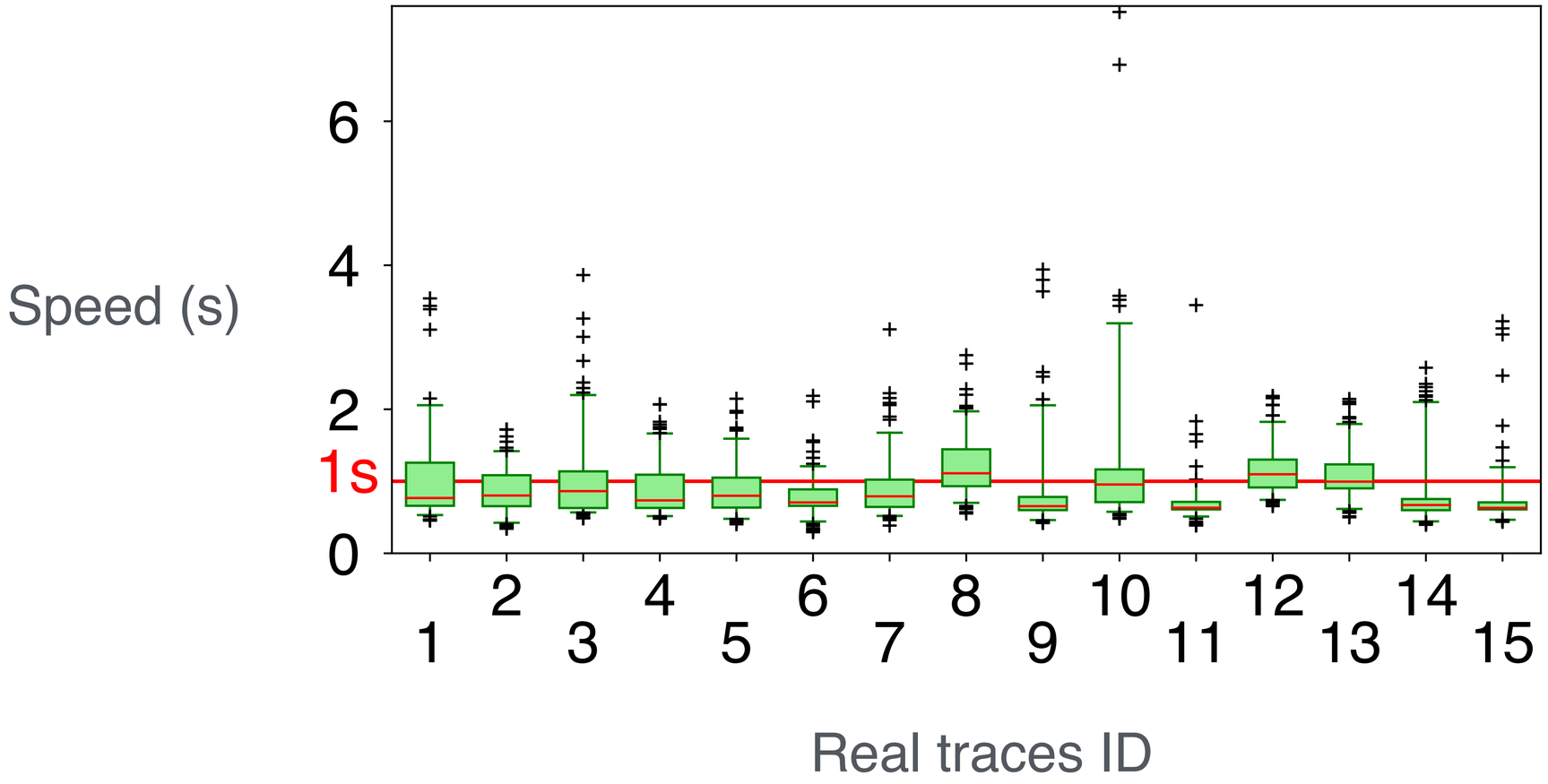
Blink avoids incorrectly inferring failures when packet loss is below 4%

packet loss %	1	2	3	4	5	…	8	9
False Positive Rate	**0**	**0**	**0**	0.67	0.67	…	1.3	2.7

Blink infers a failure within 1s for the majority of the cases



Blink infers a failure within 1s for the majority of the cases



Outline

1. Why and how to use data-plane signals for fast rerouting
2. ***Blink*** infers more than 80% of the failures, often within 1s
3. ***Blink*** quickly reroutes traffic to working backup paths
4. ***Blink*** works in practice, on existing devices

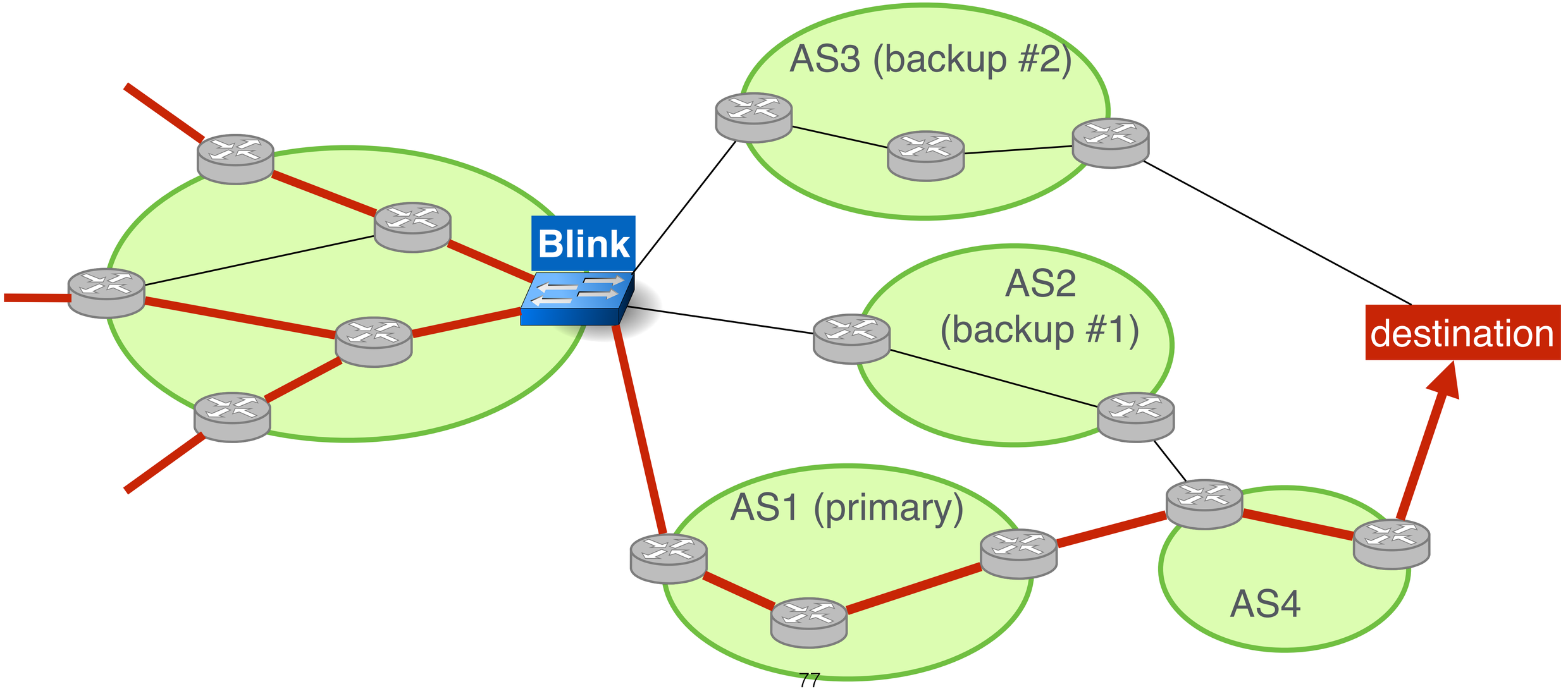
Upon detection of a failure, ***Blink*** immediately activates backup paths pre-populated by the control-plane

Problem: since the rerouting is done entirely in the data-plane,

Blink cannot prevent forwarding issues

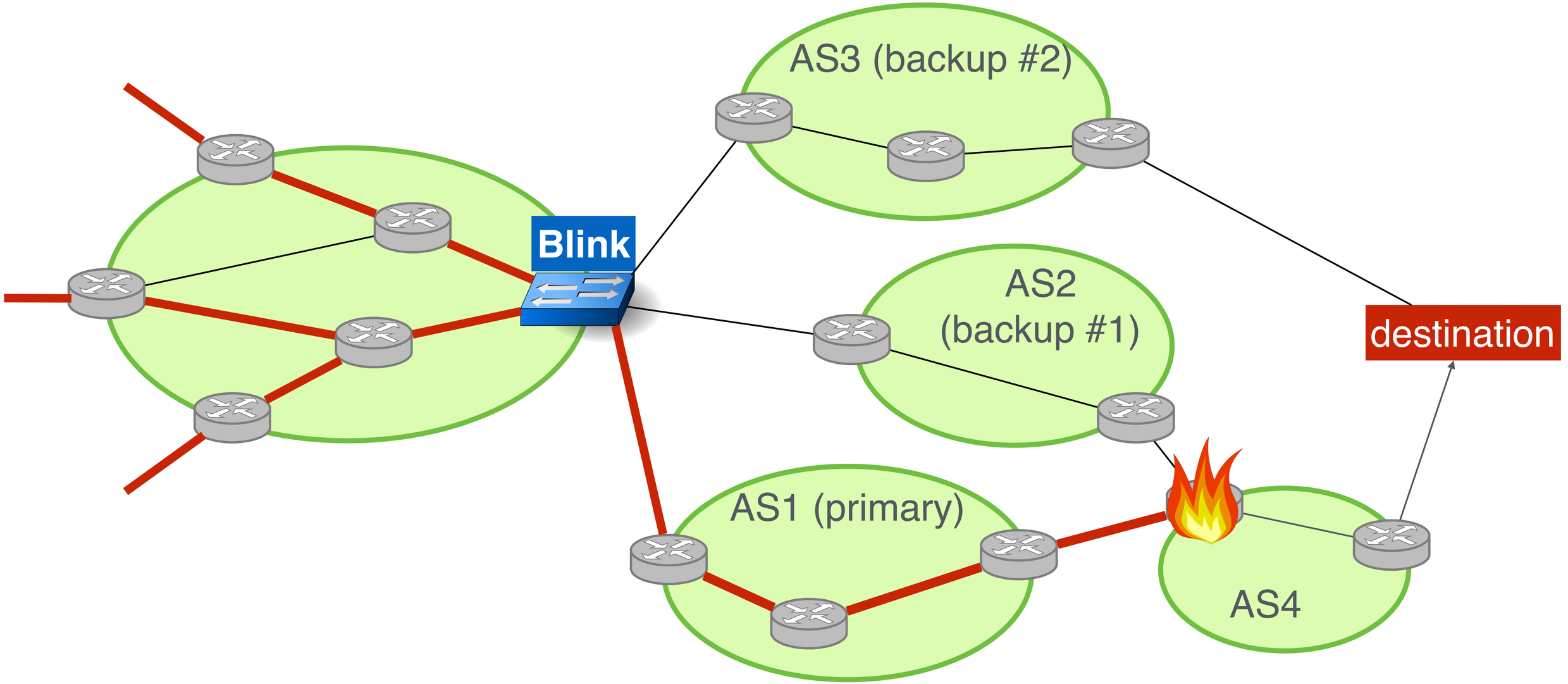
Problem: since the rerouting is done entirely in the data-plane,

Blink cannot prevent forwarding issues



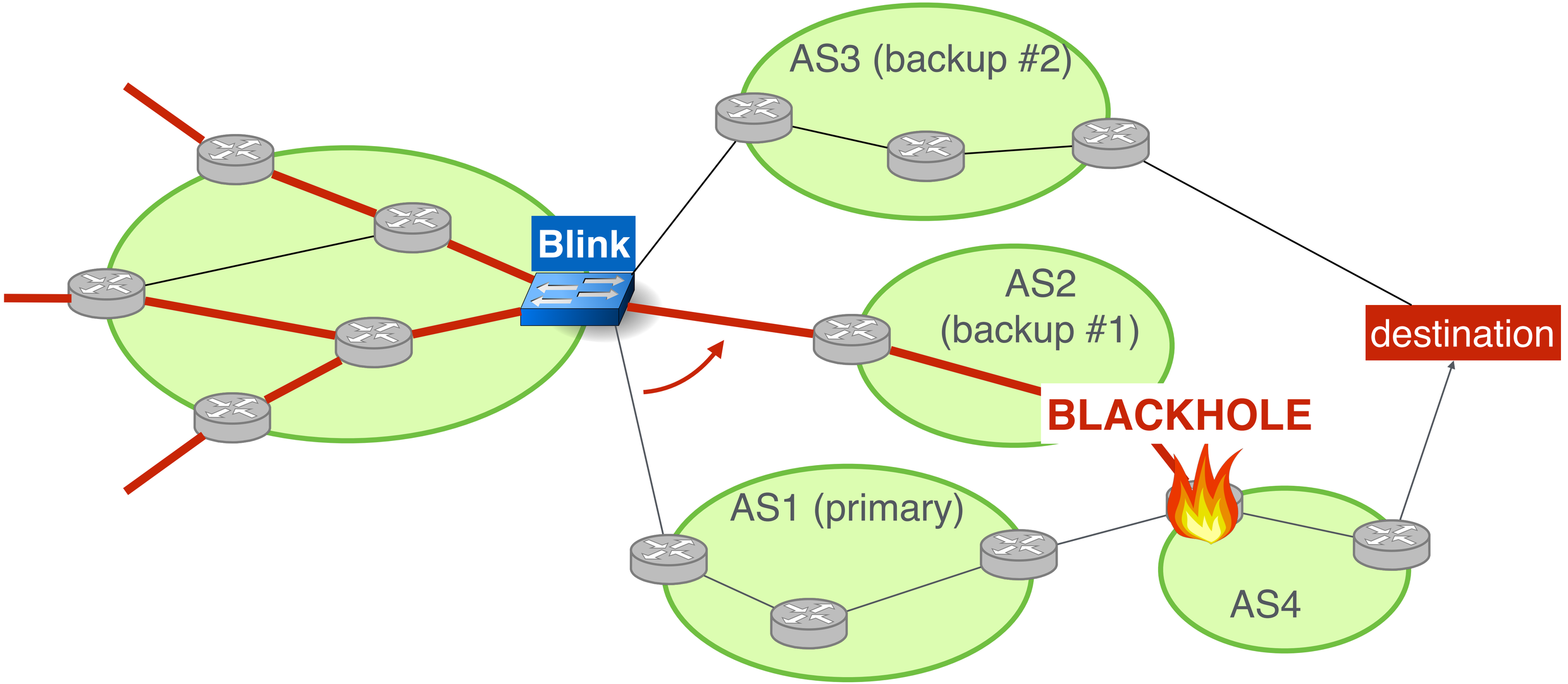
Problem: since the rerouting is done entirely in the data-plane,

Blink cannot prevent forwarding issues



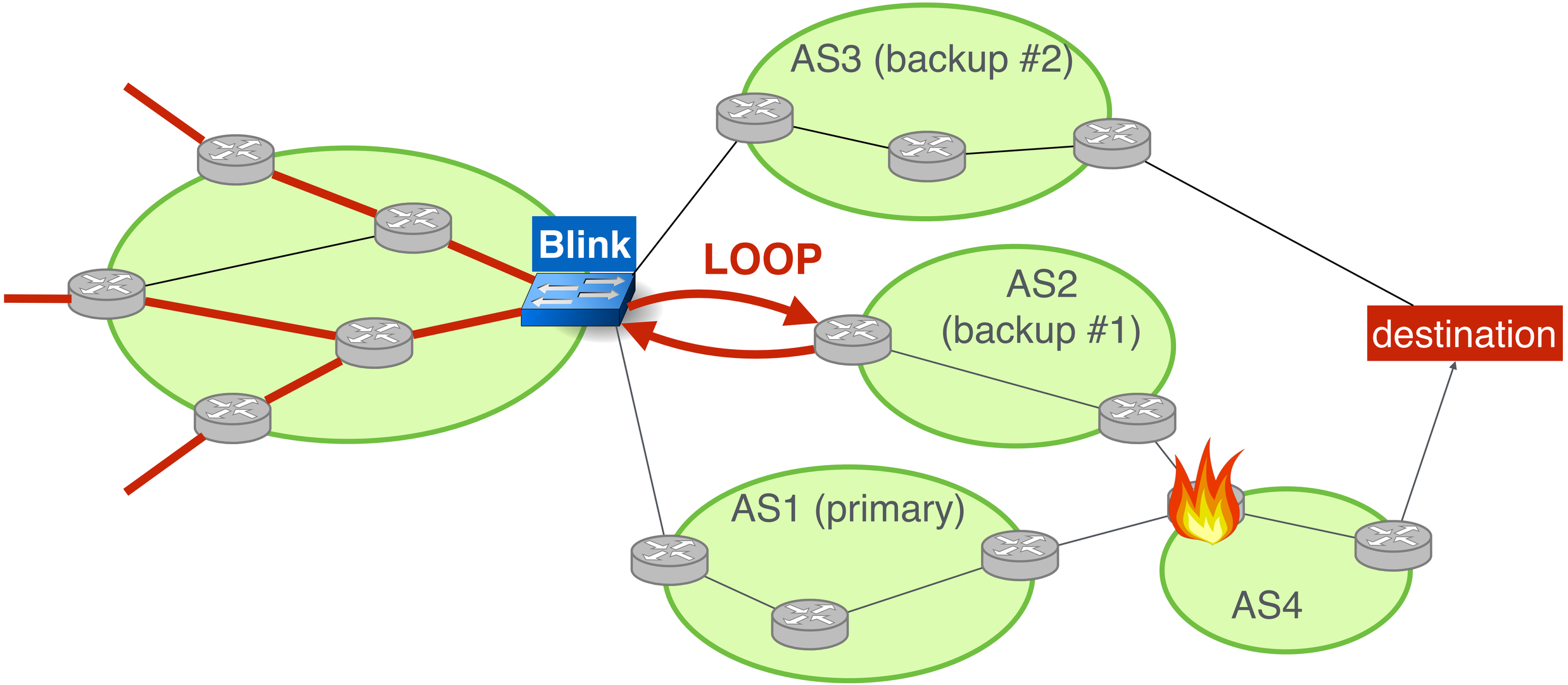
Problem: since the rerouting is done entirely in the data-plane,

Blink cannot prevent forwarding issues



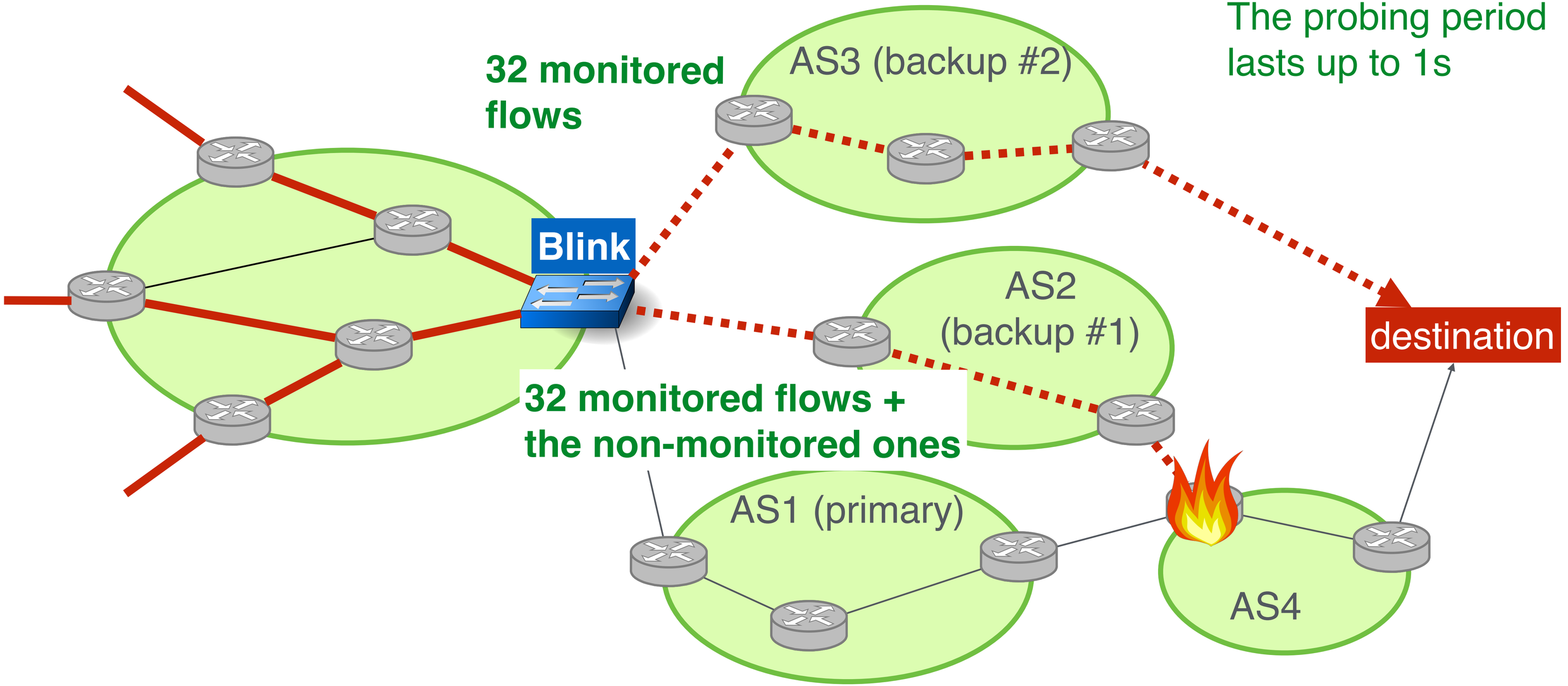
Problem: since the rerouting is done entirely in the data-plane,

Blink cannot prevent forwarding issues

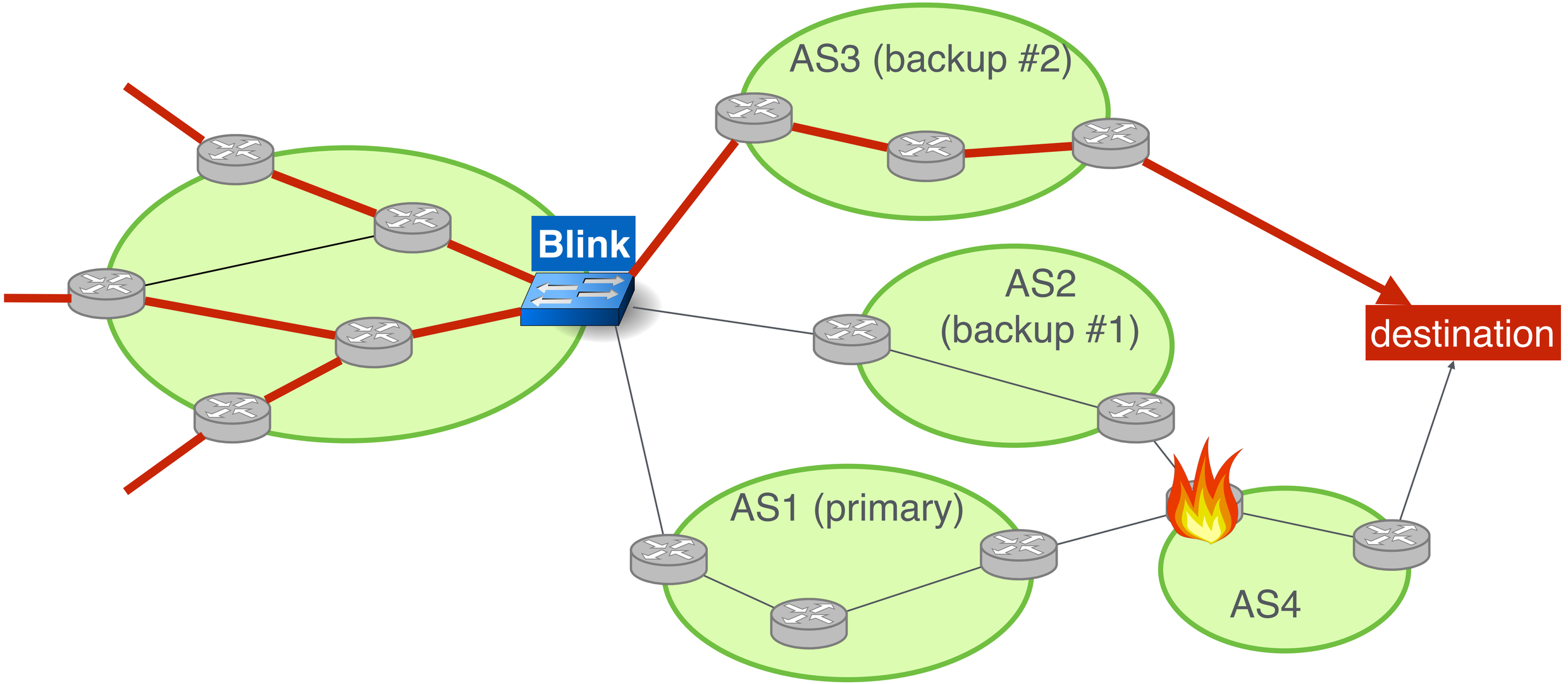


Solution: As for failures, ***Blink*** uses data-plane signals to pick a working backup path

Solution: As for failures, ***Blink*** uses data-plane signals to pick a working backup path



Solution: As for failures, ***Blink*** uses data-plane signals to pick a working backup path



As for failures, ***Blink*** compares the sequence number of consecutive packets to detect blackholes or loops*

*See the paper for an evaluation of the rerouting

Outline

1. Why and how to use data-plane signals for fast rerouting
2. ***Blink*** infers more than 80% of the failures, often within 1s
3. ***Blink*** quickly reroutes traffic to working backup paths
4. ***Blink*** works in practice, on existing devices

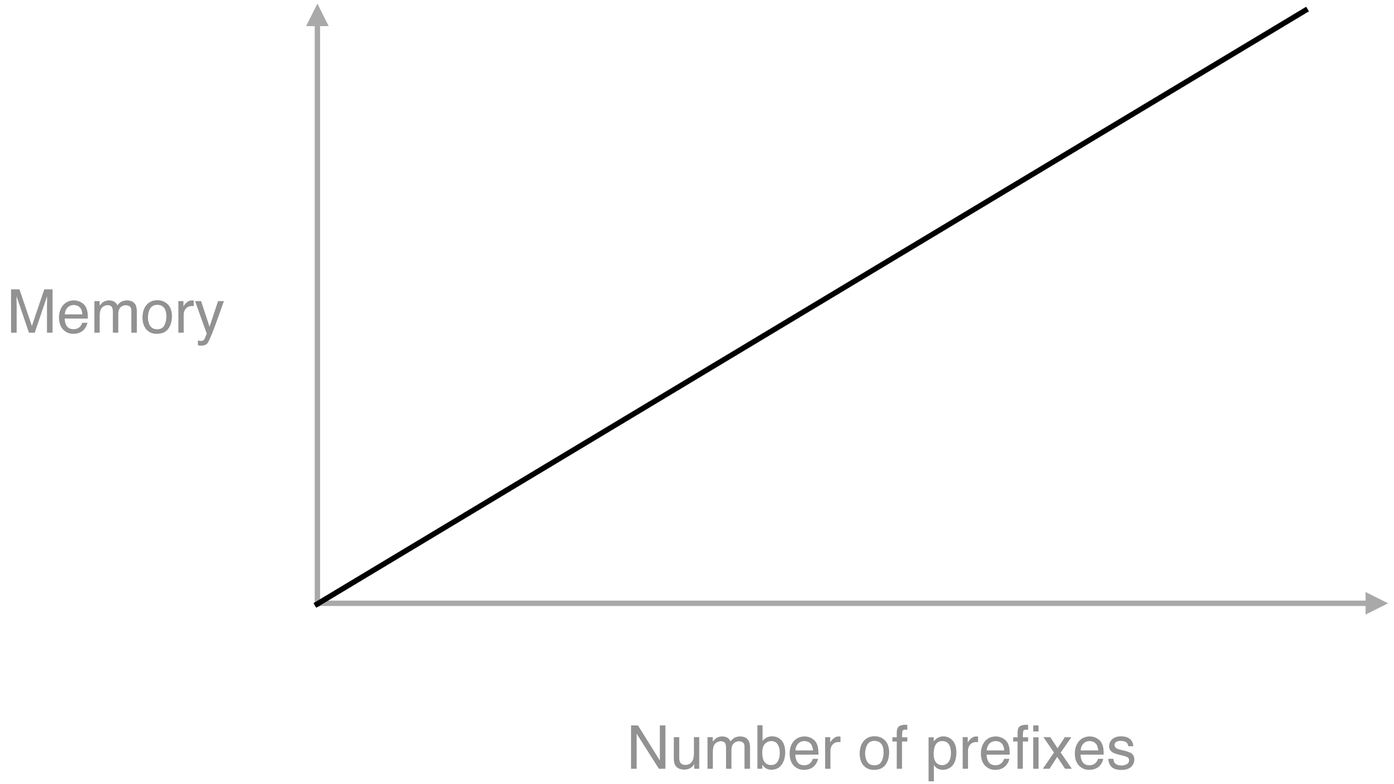
We ran ***Blink*** on the 15 real traces (15.8 hours)

We ran ***Blink*** on the 15 real traces (15.8 hours)

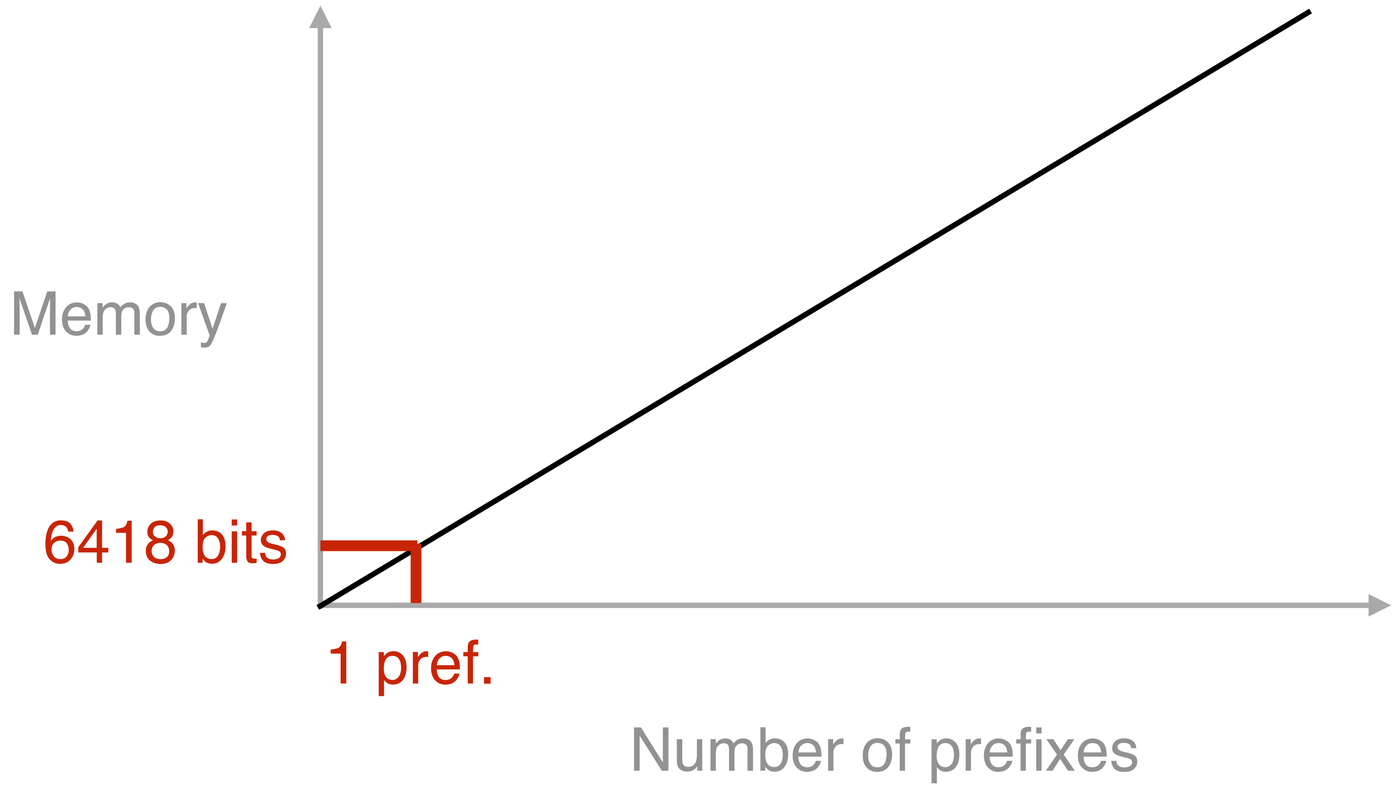
and it detected 6 outages, each affecting *at least* 42% of *all* the flows

On current programmable switches, ***Blink*** supports up to 10k prefixes

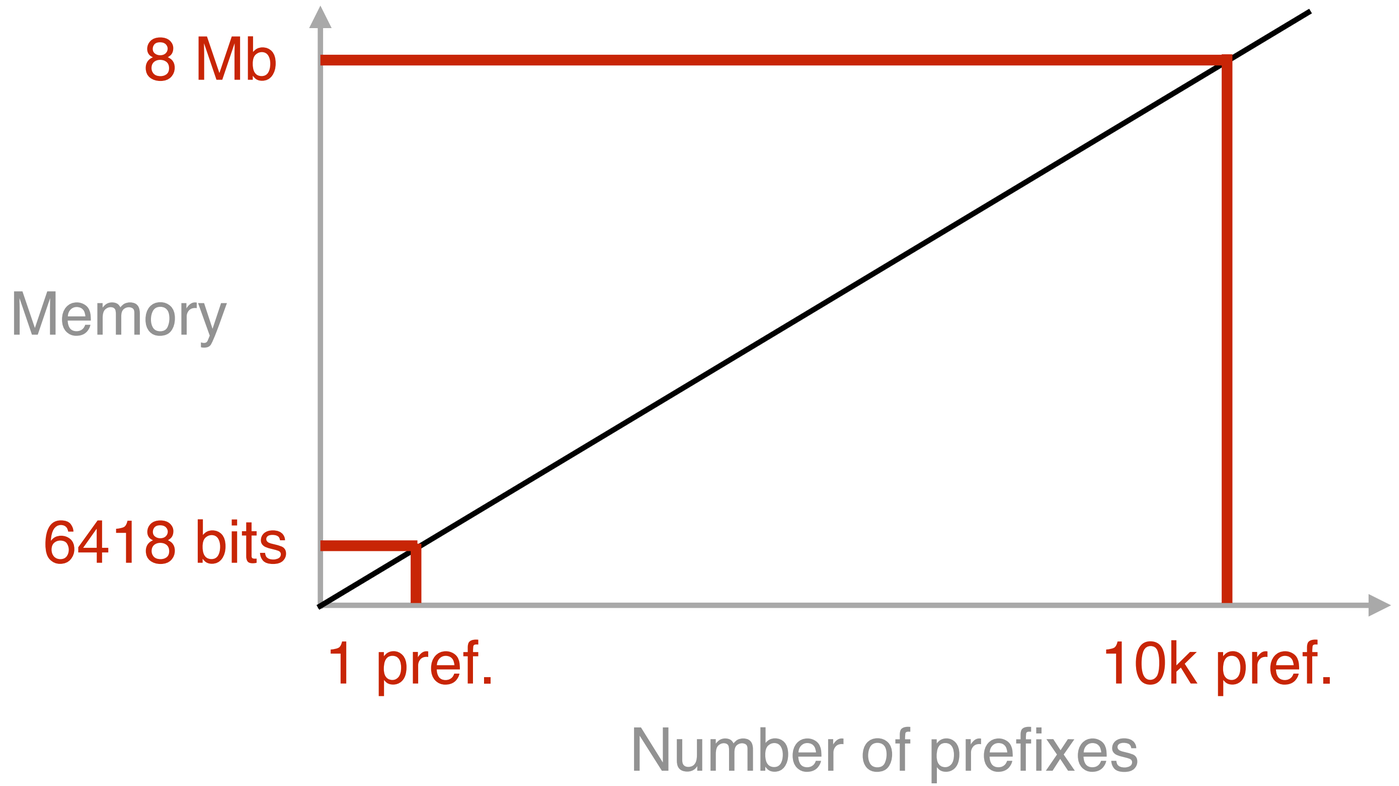
On current programmable switches, ***Blink*** supports up to 10k prefixes



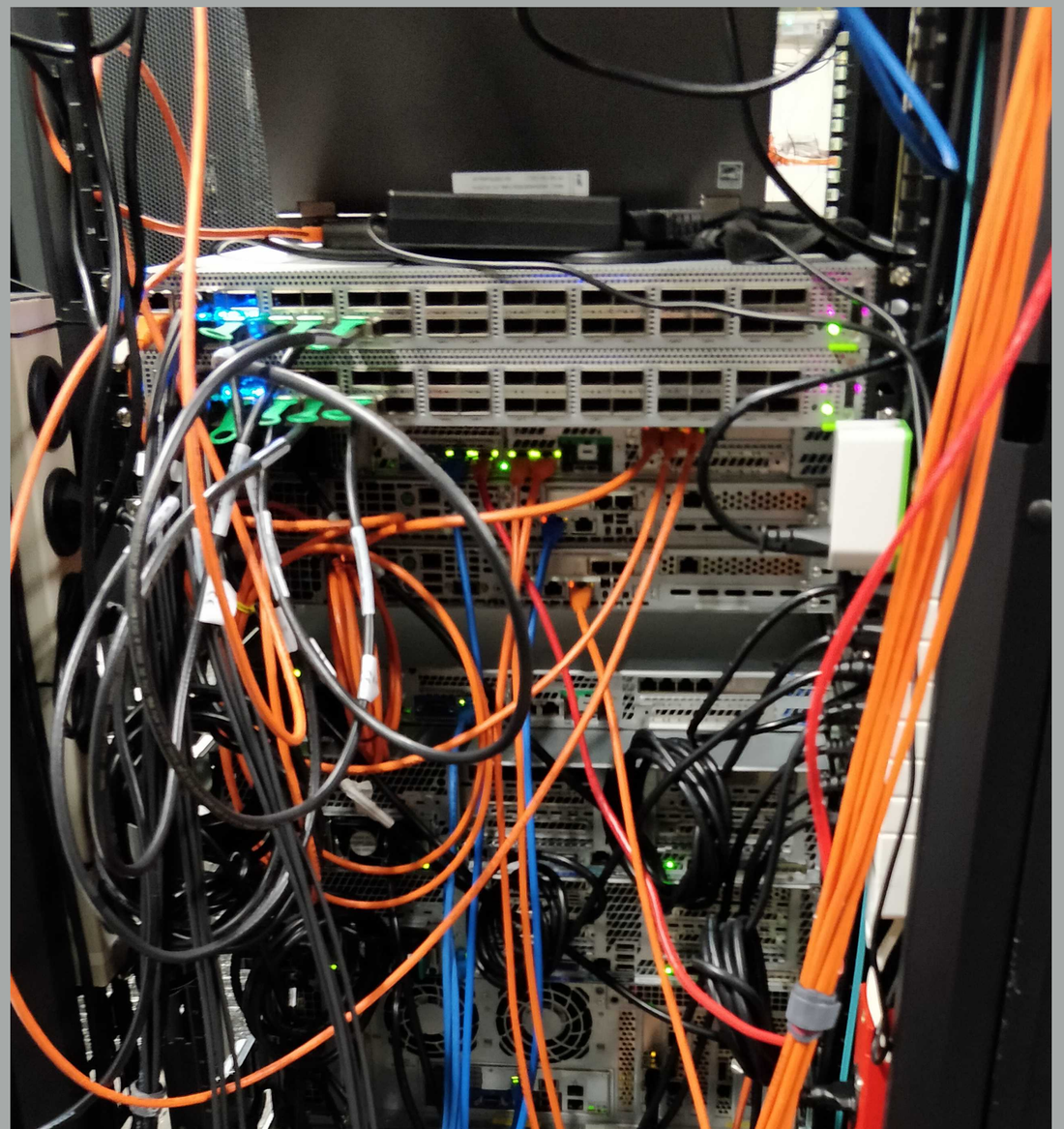
On current programmable switches, ***Blink*** supports up to 10k prefixes



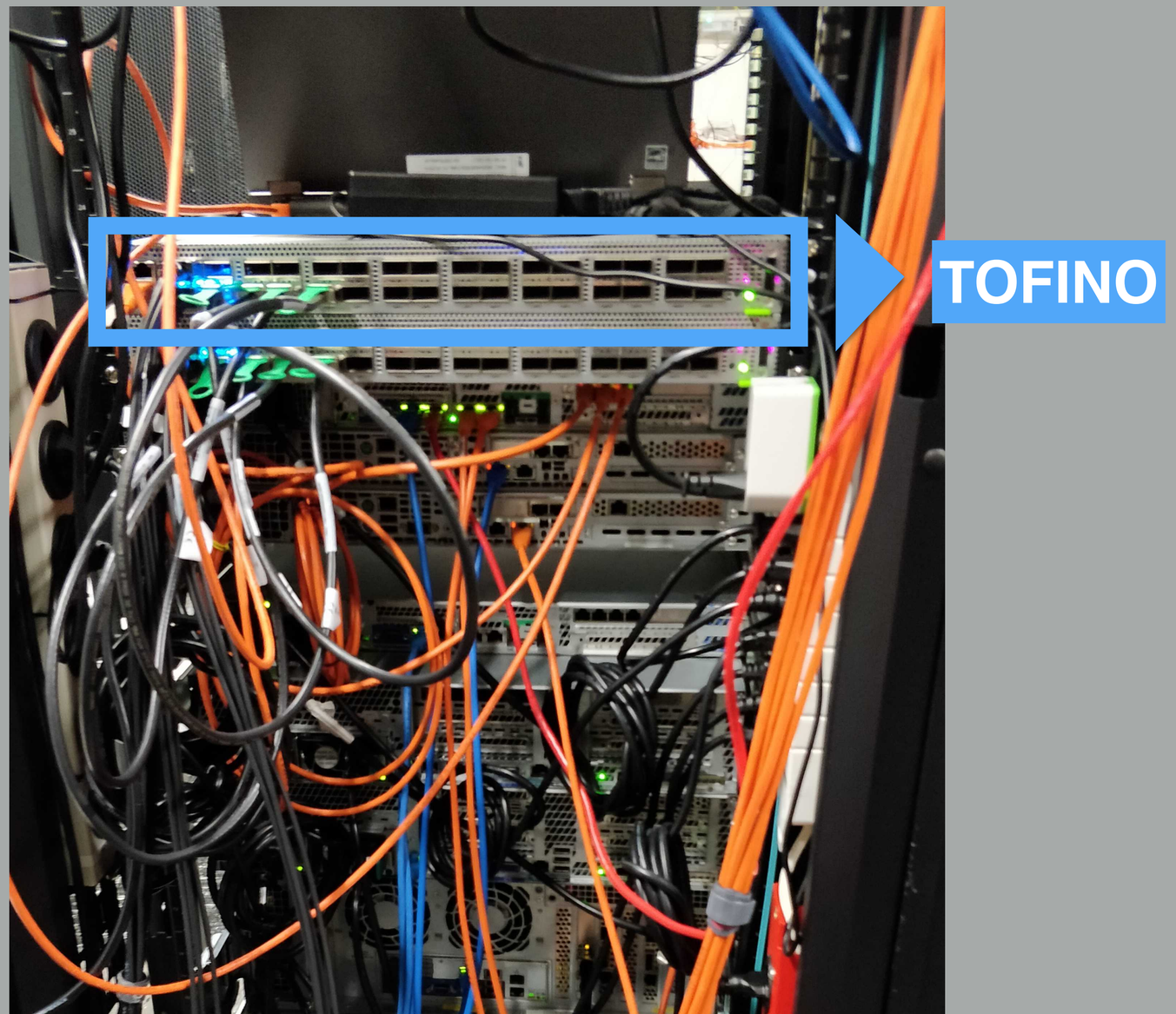
On current programmable switches, ***Blink*** supports up to 10k prefixes



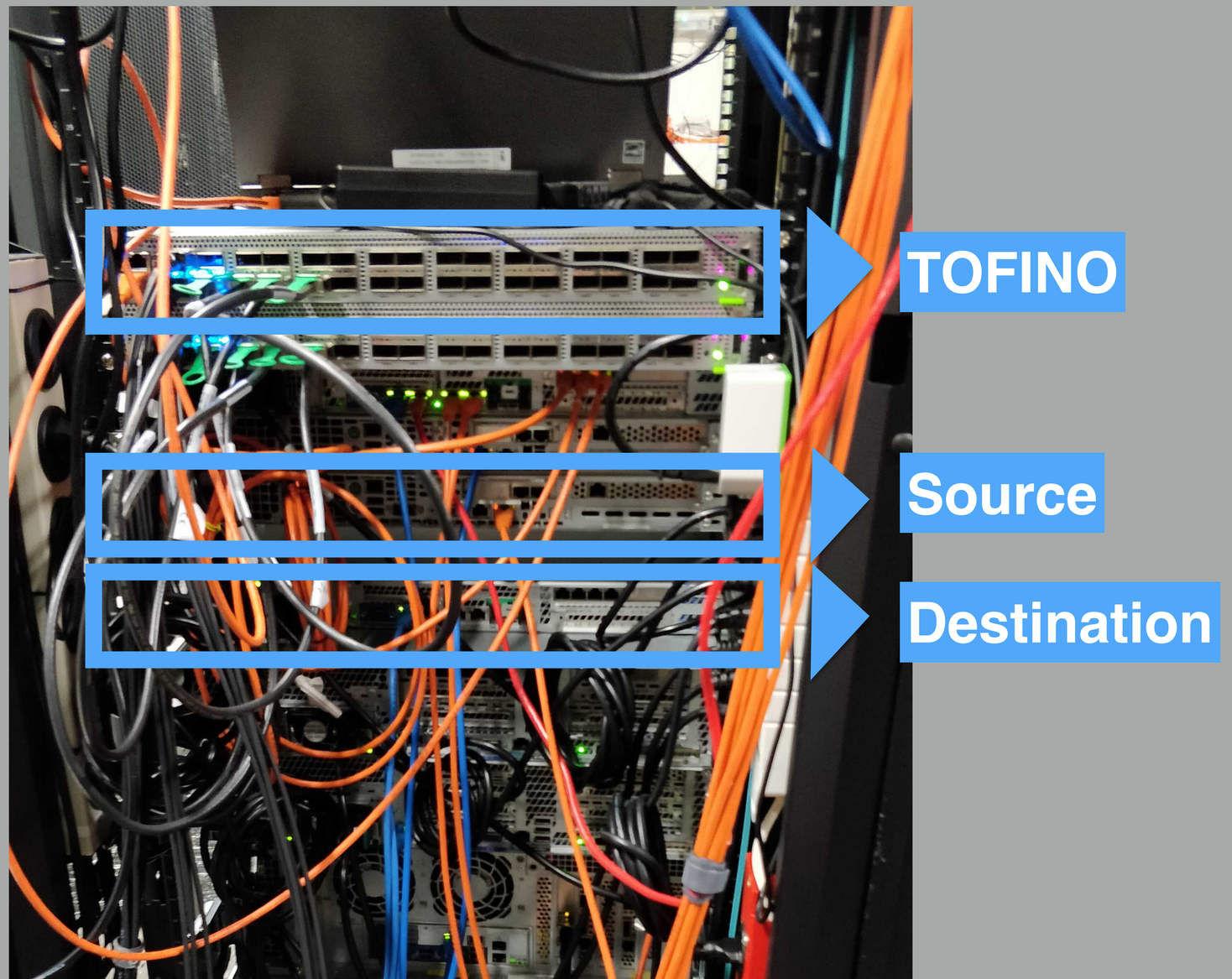
Blink works on a real Barefoot Tofino switch



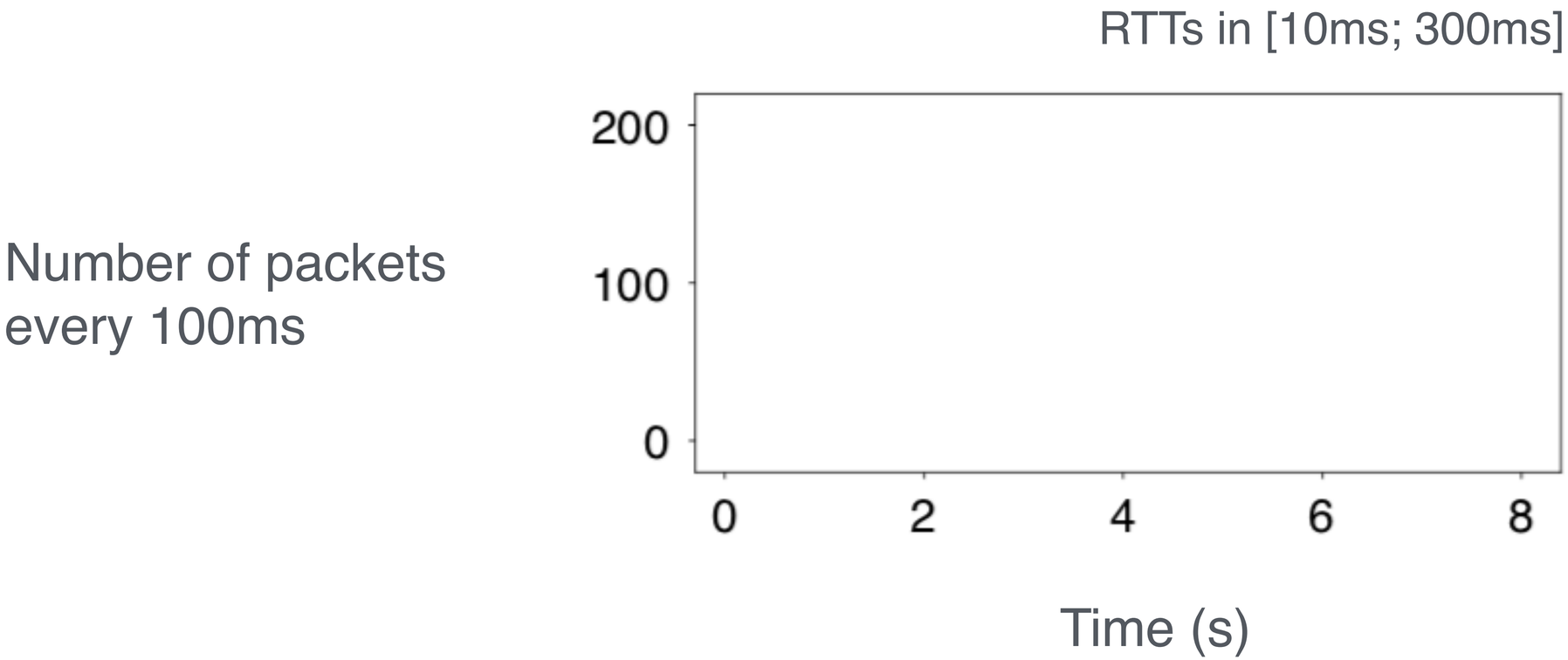
Blink works on a real Barefoot Tofino switch



Blink works on a real Barefoot Tofino switch

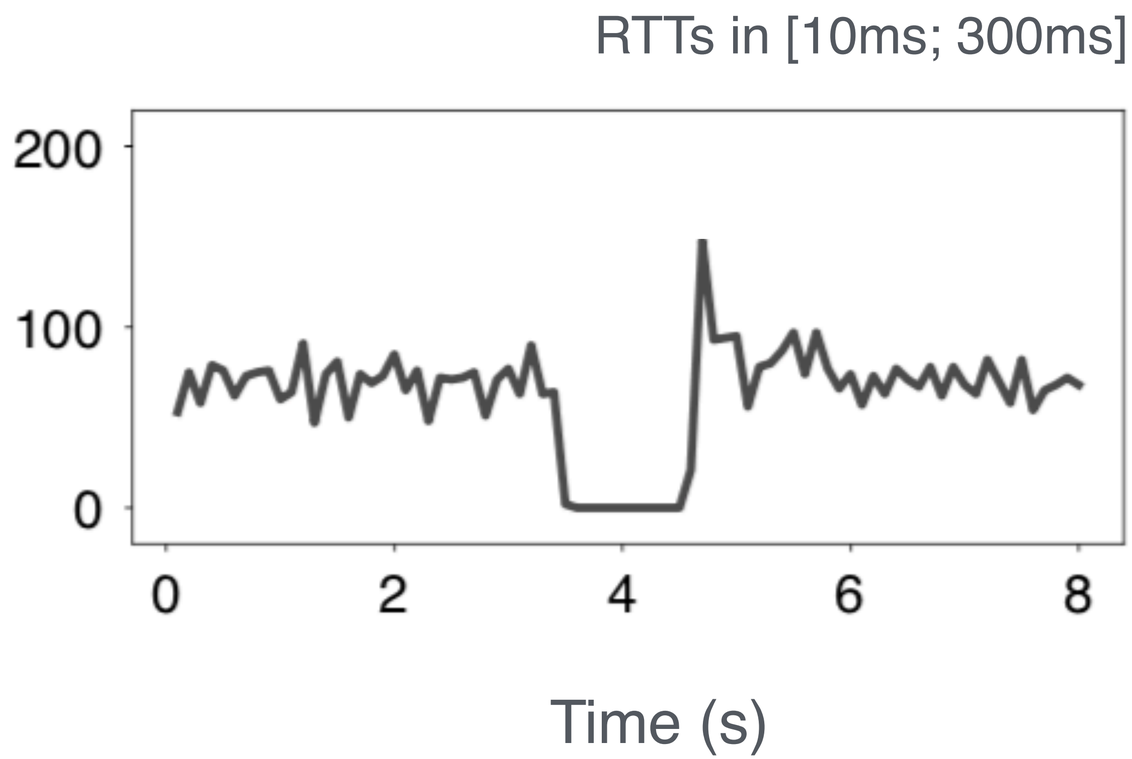


Blink works on a real Barefoot Tofino switch



Blink works on a real Barefoot Tofino switch

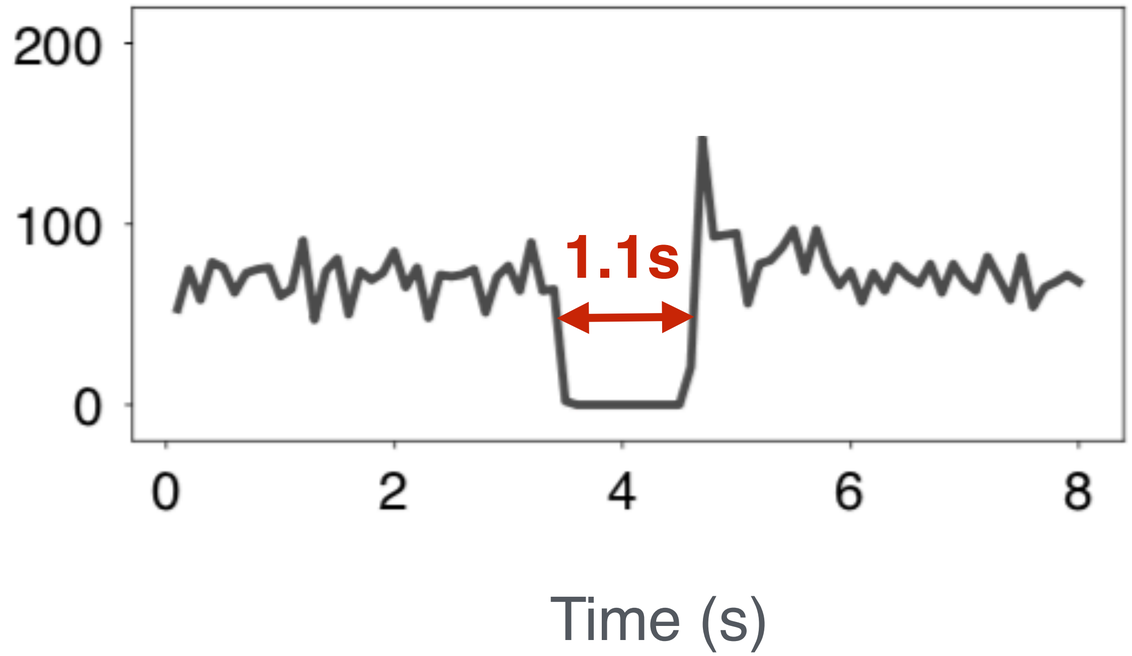
Number of packets every 100ms



Blink works on a real Barefoot Tofino switch

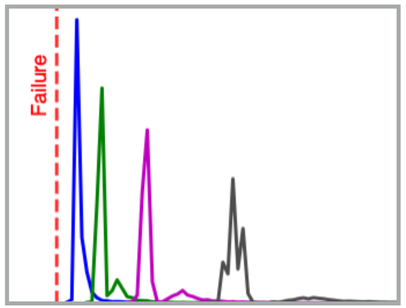
RTTs in [10ms; 300ms]

Number of packets every 100ms

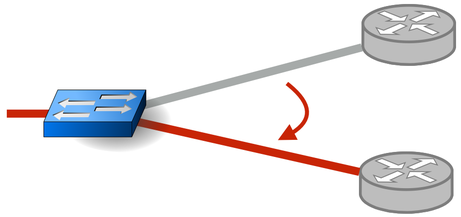


Blink: Fast Connectivity Recovery Entirely in the Data Plane

Infers failures from data-plane signals
with more than 80% accuracy, and often within 1s



Fast reroutes traffic at line rate
to working backup paths



Works on real traffic traces and on existing devices

https://blink.ethz.ch

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach
ETH Zürich

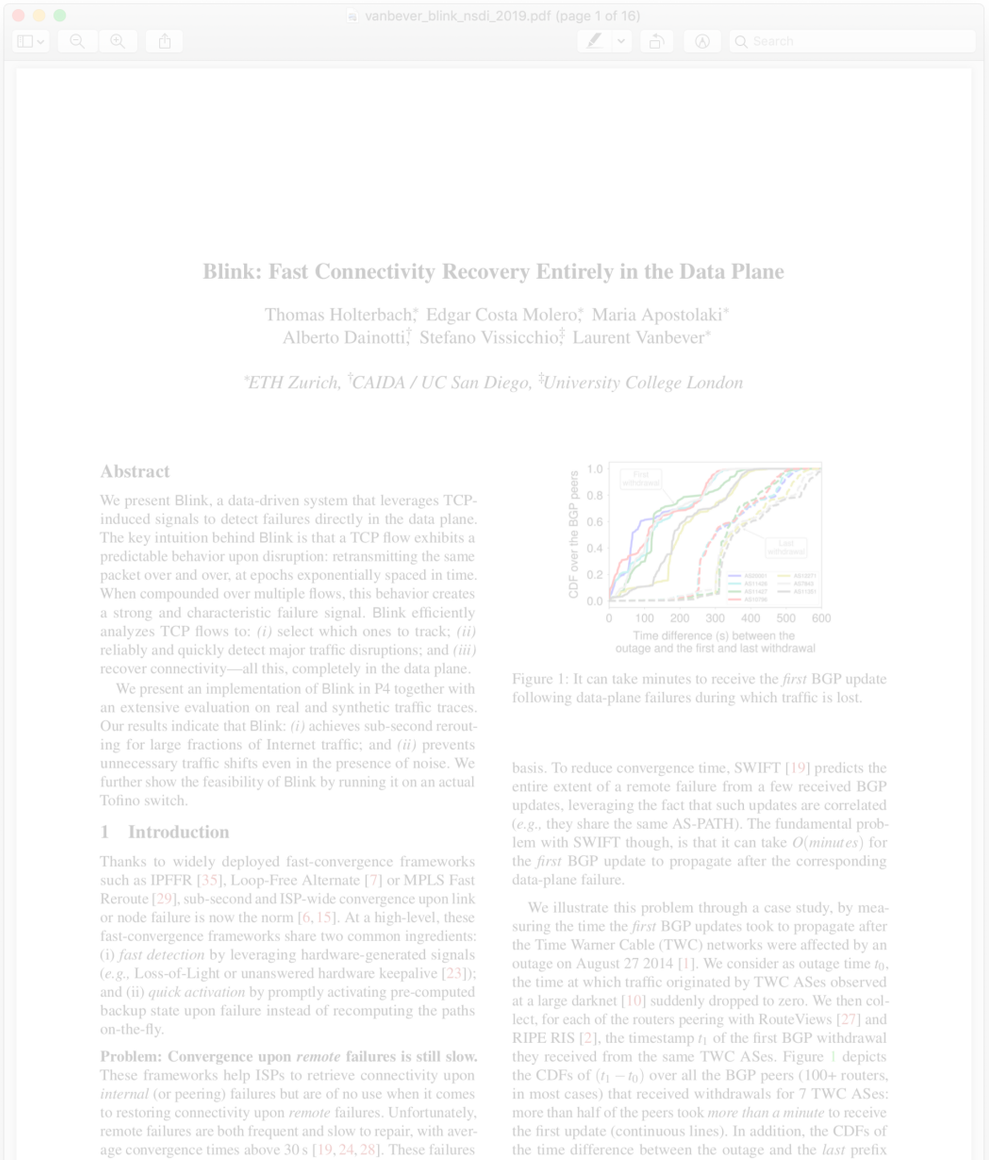
NSDI
26th February 2019

Joint work with

Edgar Costa Molero	**ETH Zürich**
Maria Apostolaki	**ETH Zürich**
Stefano Vissicchio	**University College London**
Alberto Dainotti	**CAIDA, UC San Diego**
Laurent Vanbever	**ETH Zürich**

What parts of the CP should we offload (if any) and how?

Blink [NSDI'19]



vanbever_blink_nsdi_2019.pdf (page 1 of 16)

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach,* Edgar Costa Molero,* Maria Apostolaki*
Alberto Dainotti,† Stefano Vissicchio,‡ Laurent Vanbever*

*ETH Zurich, †CAIDA / UC San Diego, ‡University College London

Abstract

We present Blink, a data-driven system that leverages TCP-induced signals to detect failures directly in the data plane. The key intuition behind Blink is that a TCP flow exhibits a predictable behavior upon disruption: retransmitting the same packet over and over, at epochs exponentially spaced in time. When compounded over multiple flows, this behavior creates a strong and characteristic failure signal. Blink efficiently analyzes TCP flows to: *(i)* select which ones to track; *(ii)* reliably and quickly detect major traffic disruptions; and *(iii)* recover connectivity—all this, completely in the data plane.

We present an implementation of Blink in P4 together with an extensive evaluation on real and synthetic traffic traces. Our results indicate that Blink: *(i)* achieves sub-second rerouting for large fractions of Internet traffic; and *(ii)* prevents unnecessary traffic shifts even in the presence of noise. We further show the feasibility of Blink by running it on an actual Tofino switch.

1 Introduction

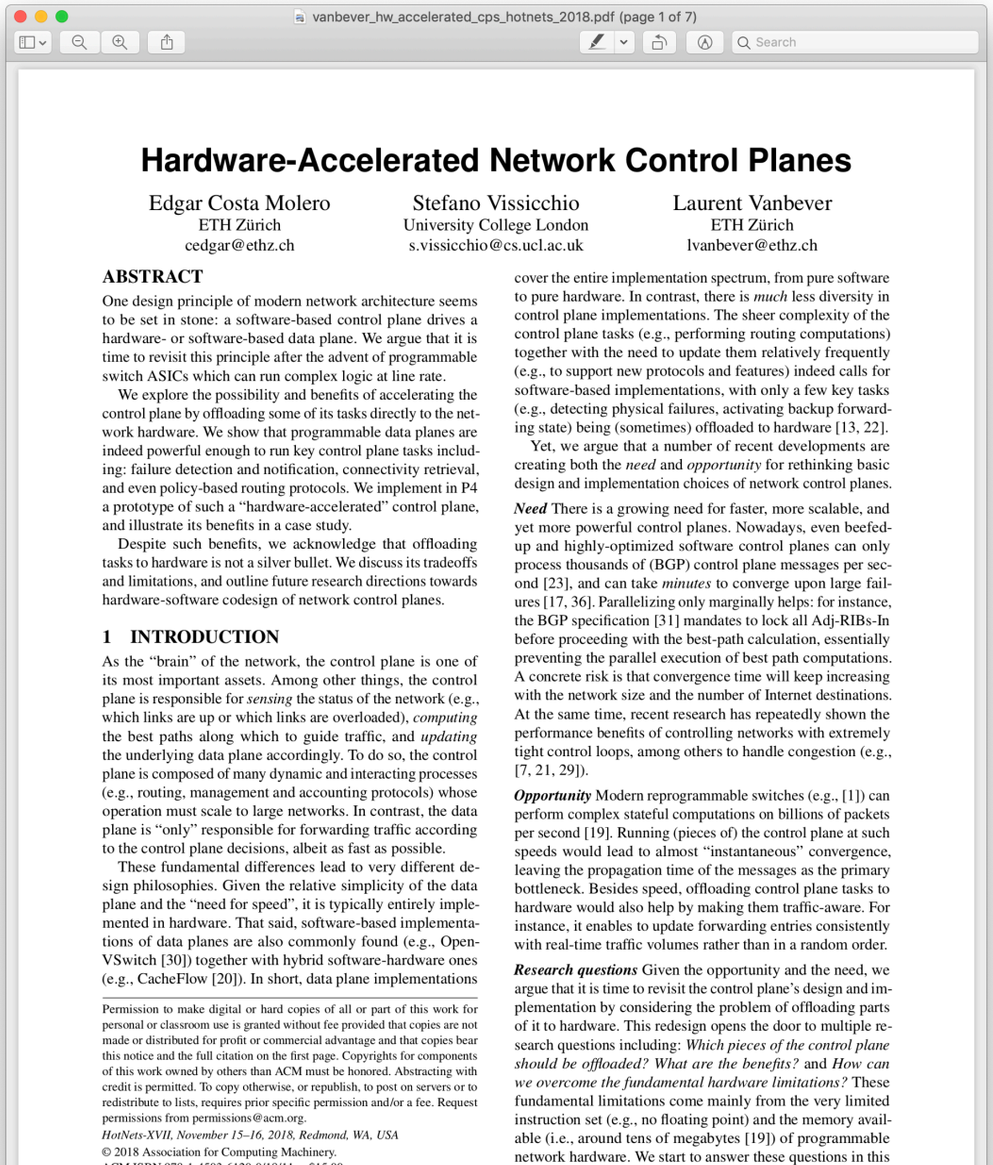
Thanks to widely deployed fast-convergence frameworks such as IPFFR [35], Loop-Free Alternate [7] or MPLS Fast Reroute [29], sub-second and ISP-wide convergence upon link or node failure is now the norm [6, 15]. At a high-level, these fast-convergence frameworks share two common ingredients: (i) *fast detection* by leveraging hardware-generated signals (*e.g.,* Loss-of-Light or unanswered hardware keepalive [23]); and (ii) *quick activation* by promptly activating pre-computed backup state upon failure instead of recomputing the paths on-the-fly.

Problem: Convergence upon *remote* failures is still slow. These frameworks help ISPs to retrieve connectivity upon *internal* (or peering) failures but are of no use when it comes to restoring connectivity upon *remote* failures. Unfortunately, remote failures are both frequent and slow to repair, with average convergence times above 30 s [19, 24, 28]. These failures indeed trigger a *control-plane-driven* convergence through the propagation of BGP updates on a per-router and per-prefix basis. To reduce convergence time, SWIFT [19] predicts the entire extent of a remote failure from a few received BGP updates, leveraging the fact that such updates are correlated (*e.g.,* they share the same AS-PATH). The fundamental problem with SWIFT though, is that it can take $O(minutes)$ for the *first* BGP update to propagate after the corresponding data-plane failure.

Figure 1: It can take minutes to receive the *first* BGP update following data-plane failures during which traffic is lost.

We illustrate this problem through a case study, by measuring the time the *first* BGP updates took to propagate after the Time Warner Cable (TWC) networks were affected by an outage on August 27 2014 [1]. We consider as outage time t_0, the time at which traffic originated by TWC ASes observed at a large darknet [10] suddenly dropped to zero. We then collect, for each of the routers peering with RouteViews [27] and RIPE RIS [2], the timestamp t_1 of the first BGP withdrawal they received from the same TWC ASes. Figure 1 depicts the CDFs of $(t_1 - t_0)$ over all the BGP peers (100+ routers, in most cases) that received withdrawals for 7 TWC ASes: more than half of the peers took *more than a minute* to receive the first update (continuous lines). In addition, the CDFs of the time difference between the outage and the *last* prefix withdrawal for each AS, show that BGP convergence can be as slow as several minutes (dashed lines).

HW-accelerated CPs [HotNets'18]



vanbever_hw_accelerated_cps_hotnets_2018.pdf (page 1 of 7)

Hardware-Accelerated Network Control Planes

Edgar Costa Molero
ETH Zürich
cedgar@ethz.ch

Stefano Vissicchio
University College London
s.vissicchio@cs.ucl.ac.uk

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

ABSTRACT

One design principle of modern network architecture seems to be set in stone: a software-based control plane drives a hardware- or software-based data plane. We argue that it is time to revisit this principle after the advent of programmable switch ASICs which can run complex logic at line rate.

We explore the possibility and benefits of accelerating the control plane by offloading some of its tasks directly to the network hardware. We show that programmable data planes are indeed powerful enough to run key control plane tasks including: failure detection and notification, connectivity retrieval, and even policy-based routing protocols. We implement in P4 a prototype of such a "hardware-accelerated" control plane, and illustrate its benefits in a case study.

Despite such benefits, we acknowledge that offloading tasks to hardware is not a silver bullet. We discuss its tradeoffs and limitations, and outline future research directions towards hardware-software codesign of network control planes.

1 INTRODUCTION

As the "brain" of the network, the control plane is one of its most important assets. Among other things, the control plane is responsible for *sensing* the status of the network (e.g., which links are up or which links are overloaded), *computing* the best paths along which to guide traffic, and *updating* the underlying data plane accordingly. To do so, the control plane is composed of many dynamic and interacting processes (e.g., routing, management and accounting protocols) whose operation must scale to large networks. In contrast, the data plane is "only" responsible for forwarding traffic according to the control plane decisions, albeit as fast as possible.

These fundamental differences lead to very different design philosophies. Given the relative simplicity of the data plane and the "need for speed", it is typically entirely implemented in hardware. That said, software-based implementations of data planes are also commonly found (e.g., OpenVSwitch [30]) together with hybrid software-hardware ones (e.g., CacheFlow [20]). In short, data plane implementations cover the entire implementation spectrum, from pure software to pure hardware. In contrast, there is *much* less diversity in control plane implementations. The sheer complexity of the control plane tasks (e.g., performing routing computations) together with the need to update them relatively frequently (e.g., to support new protocols and features) indeed calls for software-based implementations, with only a few key tasks (e.g., detecting physical failures, activating backup forwarding state) being (sometimes) offloaded to hardware [13, 22].

Yet, we argue that a number of recent developments are creating both the *need* and *opportunity* for rethinking basic design and implementation choices of network control planes.

Need There is a growing need for faster, more scalable, and yet more powerful control planes. Nowadays, even beefed-up and highly-optimized software control planes can only process thousands of (BGP) control plane messages per second [23], and can take *minutes* to converge upon large failures [17, 36]. Parallelizing only marginally helps: for instance, the BGP specification [31] mandates to lock all Adj-RIBs-In before proceeding with the best-path calculation, essentially preventing the parallel execution of best path computations. A concrete risk is that convergence time will keep increasing with the network size and the number of Internet destinations. At the same time, recent research has repeatedly shown the performance benefits of controlling networks with extremely tight control loops, among others to handle congestion (e.g., [7, 21, 29]).

Opportunity Modern reprogrammable switches (e.g., [1]) can perform complex stateful computations on billions of packets per second [19]. Running (pieces of) the control plane at such speeds would lead to almost "instantaneous" convergence, leaving the propagation time of the messages as the primary bottleneck. Besides speed, offloading control plane tasks to hardware would also help by making them traffic-aware. For instance, it enables to update forwarding entries consistently with real-time traffic volumes rather than in a random order.

Research questions Given the opportunity and the need, we argue that it is time to revisit the control plane's design and implementation by considering the problem of offloading parts of it to hardware. This redesign opens the door to multiple research questions including: *Which pieces of the control plane should be offloaded? What are the benefits?* and *How can we overcome the fundamental hardware limitations?* These fundamental limitations come mainly from the very limited instruction set (e.g., no floating point) and the memory available (i.e., around tens of megabytes [19]) of programmable network hardware. We start to answer these questions in this paper and make two contributions.

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA

ACM ISBN 978-1-4503-6120-0/18/11…$15.00
https://doi.org/10.1145/3286062.3286080

Hardware-Accelerated Network Control Planes

Edgar Costa Molero[1],

Stefano Vissicchio[2], Laurent Vanbever[1]

[1] ETH zürich [2] UCL

Software-based control planes have room for improvements

Software-based control planes have room for improvements

1	Reaction time	It can take seconds to minutes to detect failures

Software-based control planes have room for improvements

1 Reaction time

2 Compute

It can take minutes to recompute an entire forwarding table

Software-based control planes have room for improvements

1 Reaction time

2 Compute

3 Update

It takes ~100us to update a single forwarding entry

Modern programmable devices can perform computations on billions of packets per second

Modern programmable devices can perform computations on billions of packets per second

- Read & modify packet headers

  e.g. to update network state

- Perform (simple) operations

  e.g. min & max

- Add or remove custom headers

  e.g. to carry routing information

- Maintain state

  e.g. to save best paths

Could we offload control-plane tasks to the data plane?

Could we offload control-plane tasks to the data plane?

Yes… ***but…***

Could we offload control-plane tasks to the data plane?

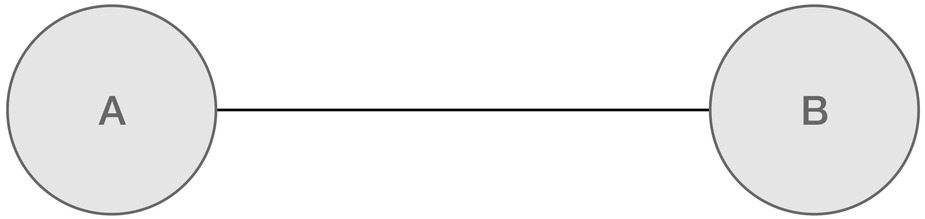
Could we offload control-plane tasks to the data plane?

sensing, notification, computation

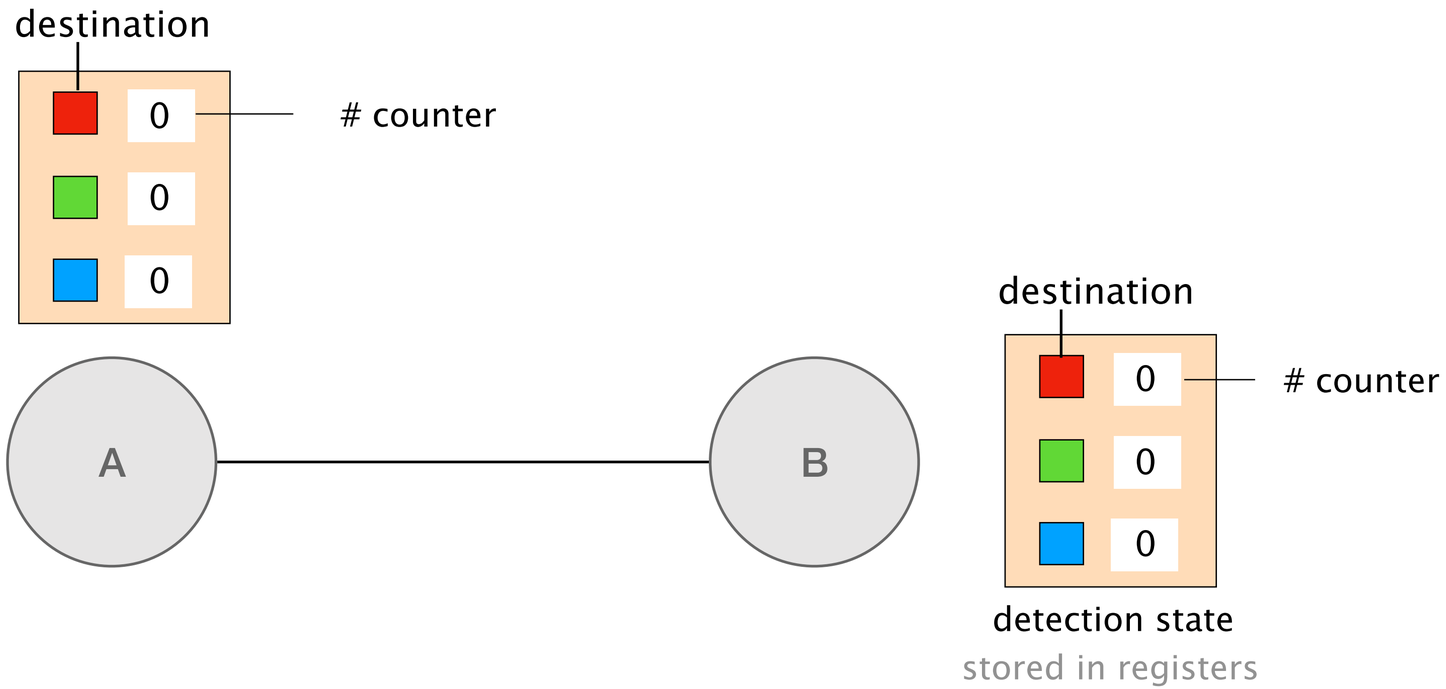
Could we offload control-plane tasks to the data plane?

sensing, notification, computation

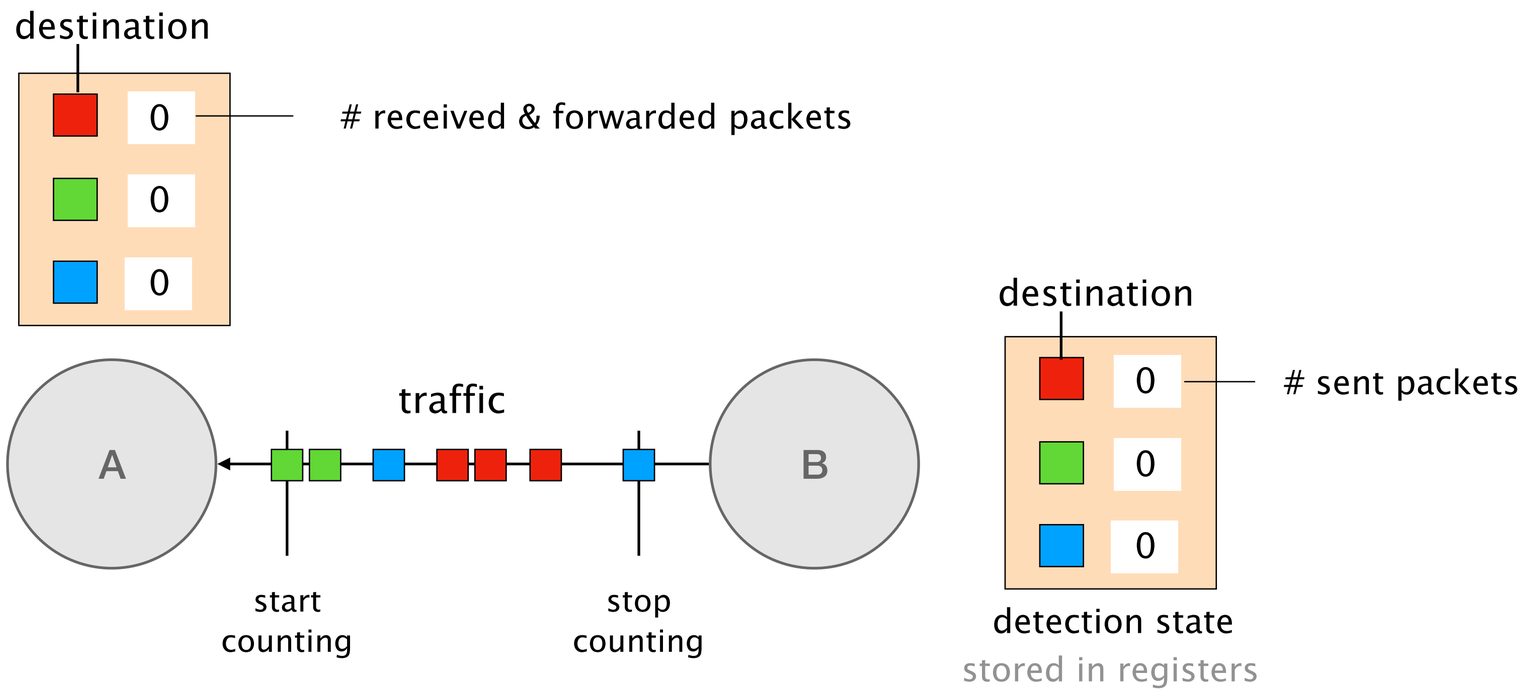
Switches can precisely "sense" the network by synchronously exchanging packet counts



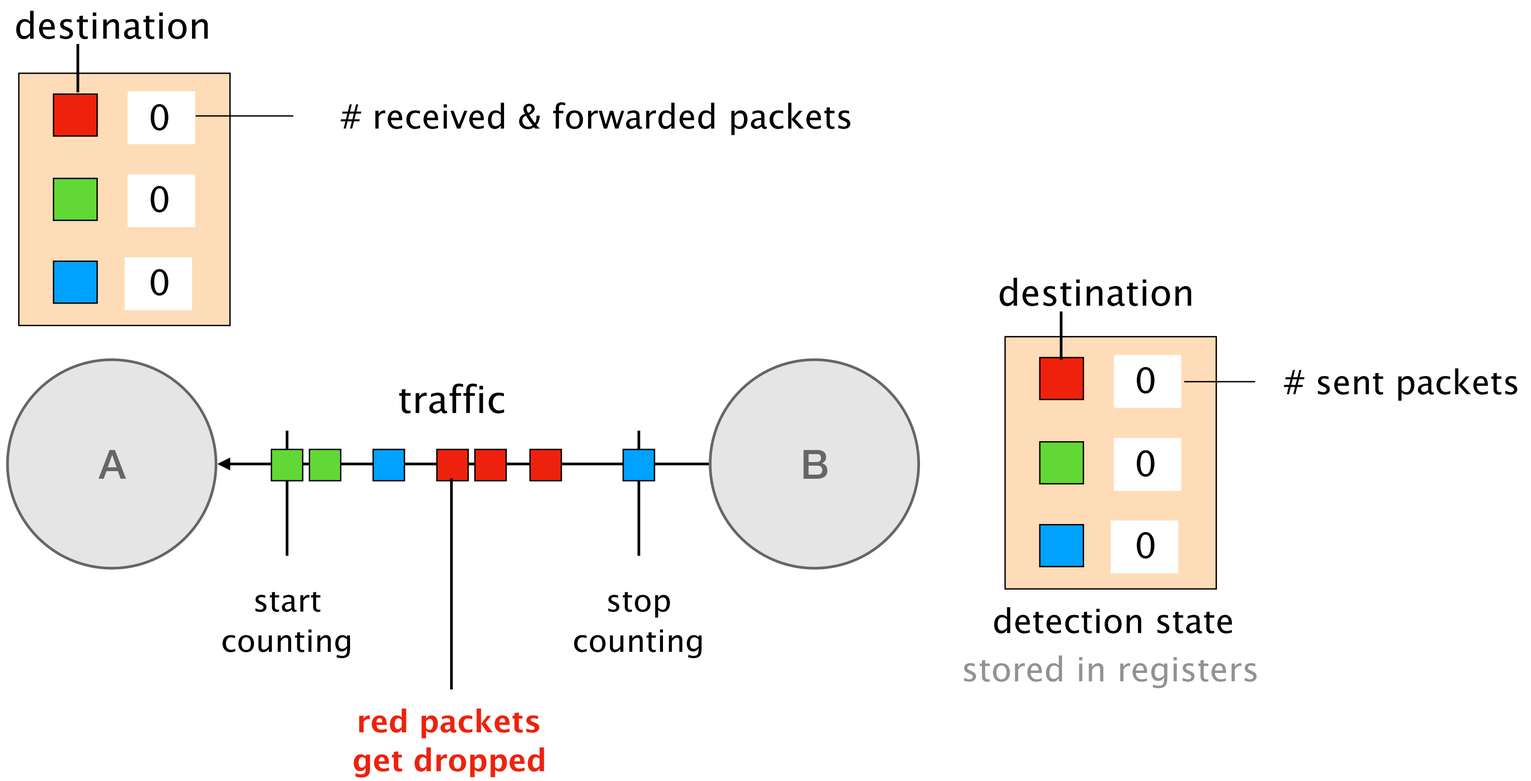
Switches can precisely "sense" the network by synchronously exchanging packet counts



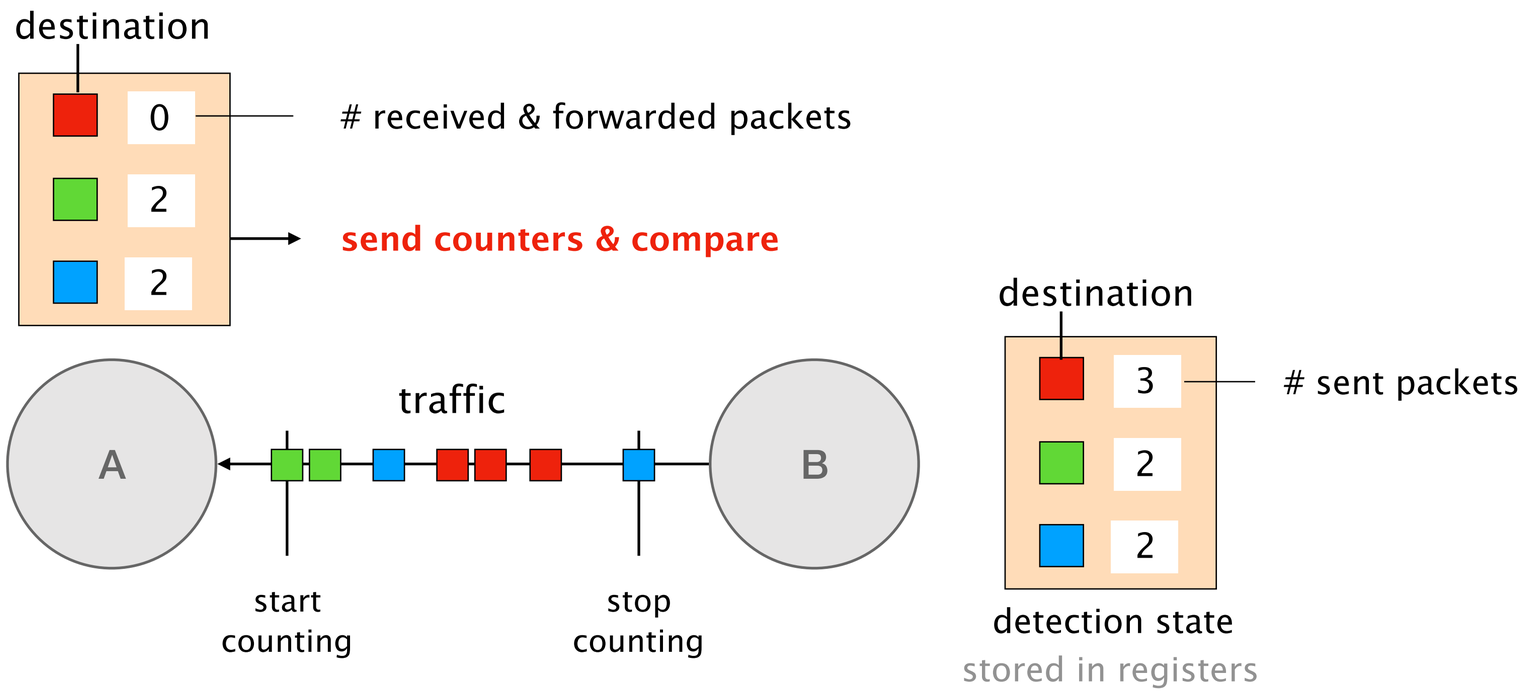
Upstream switch starts probing campaigns



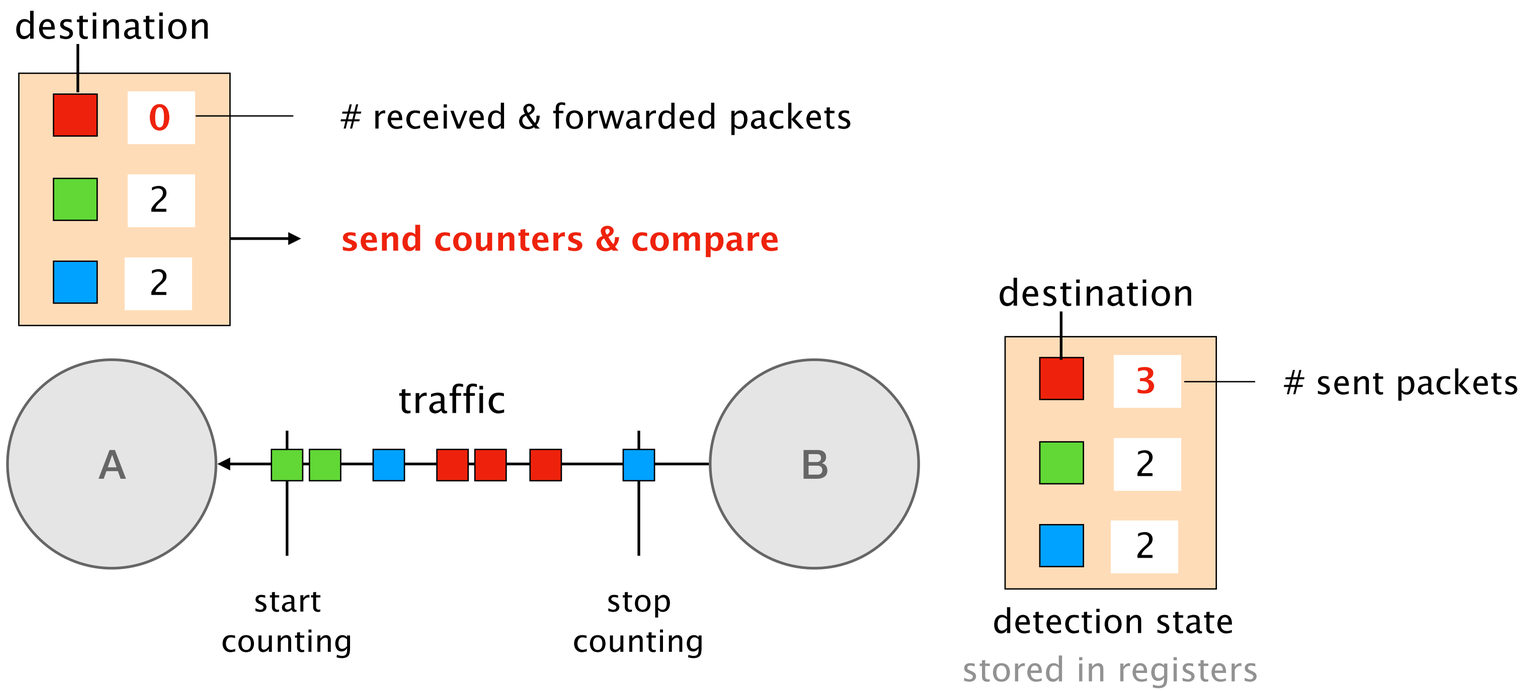
Traffic for some prefixes gets dropped



Downstream switch sends counters to upstream



Upstream switch detects the failure by comparing counters



Could we offload control-plane tasks to the data plane?

sensing, **notification**, computation

Upon detecting a failure, switches can flood notifications network-wide

Avoid broadcast storms

- Use per switch broadcast sequence numbers

Simple reliable communication

- Send notification duplicates
- Use maximum priority queues

Could we offload control-plane tasks to the data plane?

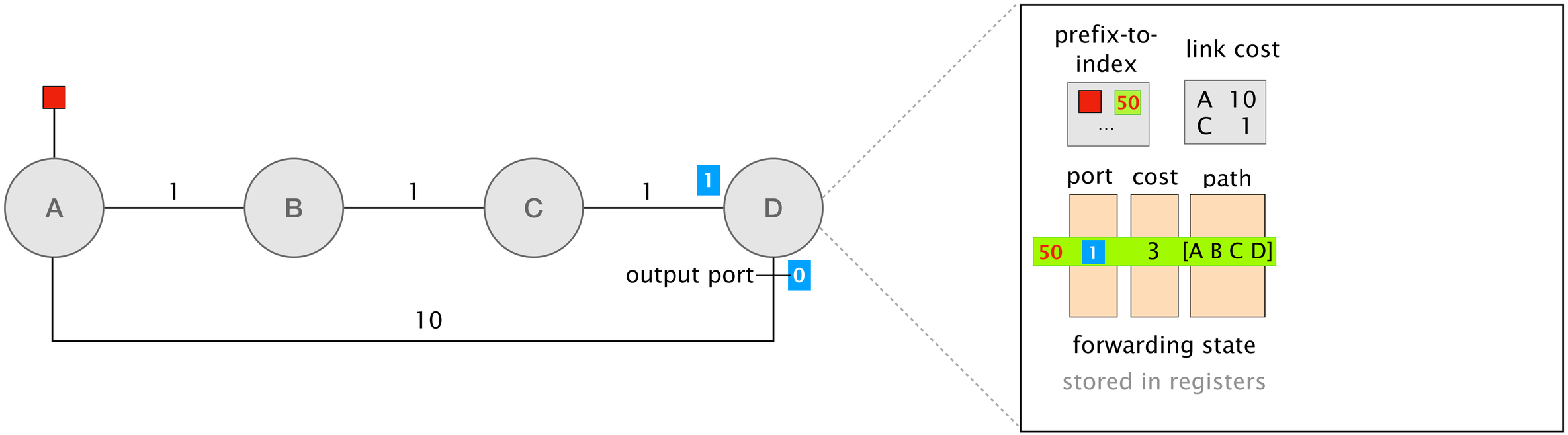
sensing, notification, **computation**

Switches can run distributed routing protocols

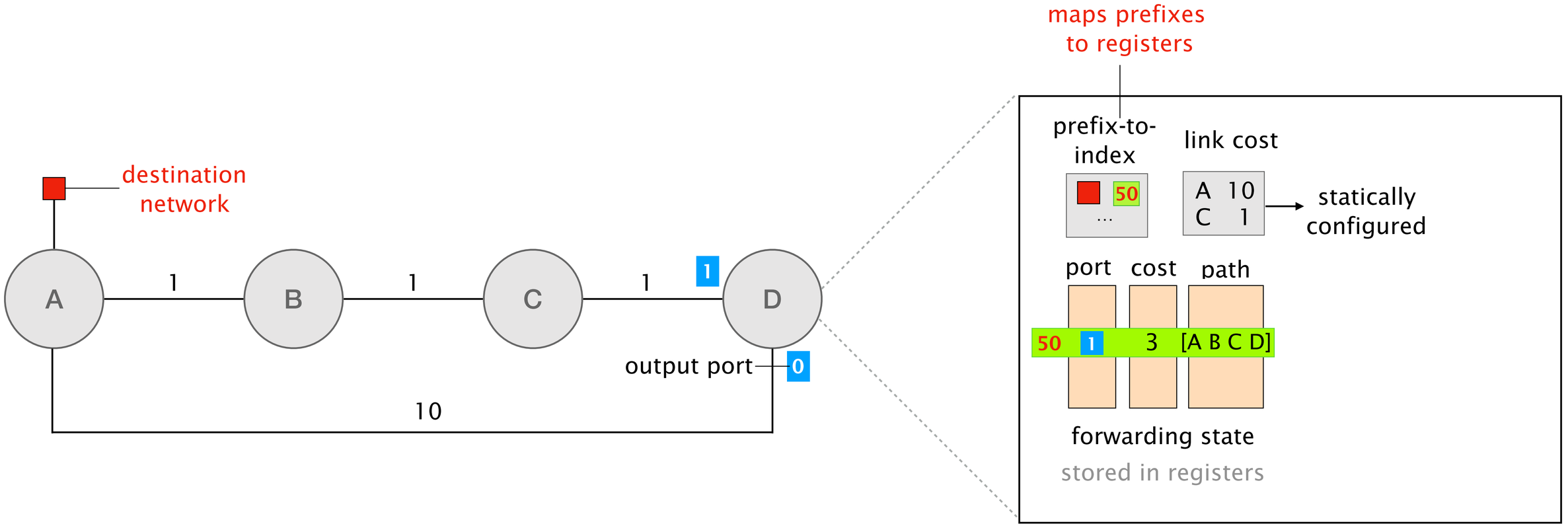
in hardware

Switches can run distributed routing protocols

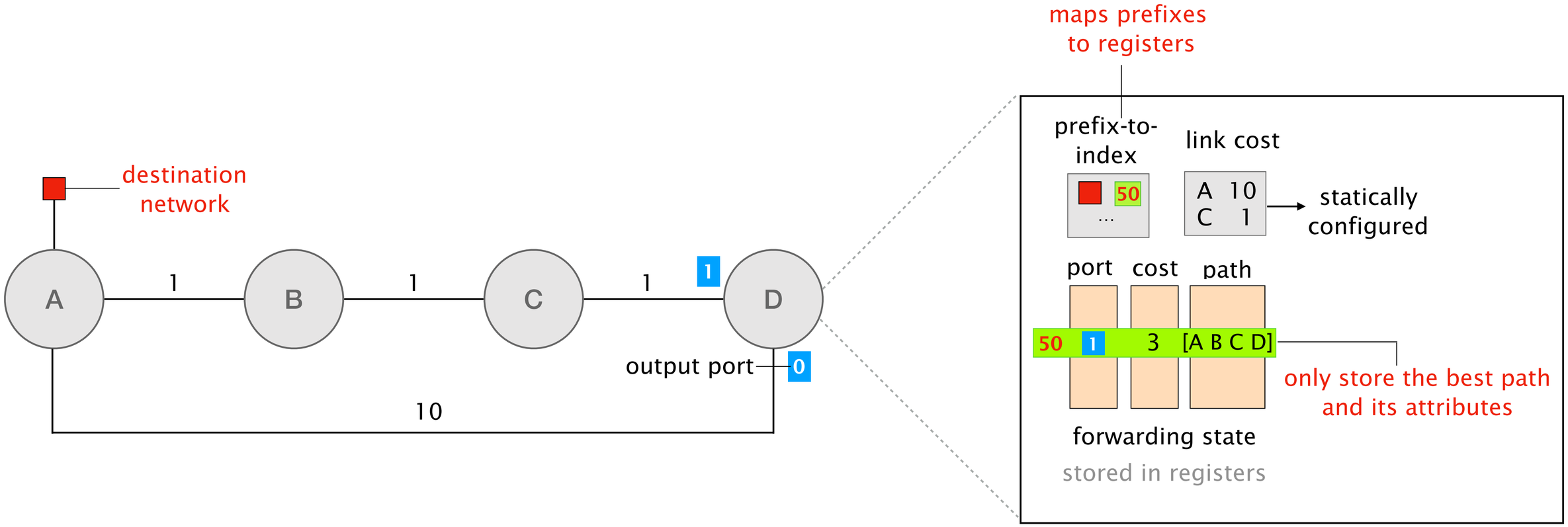
in hardware



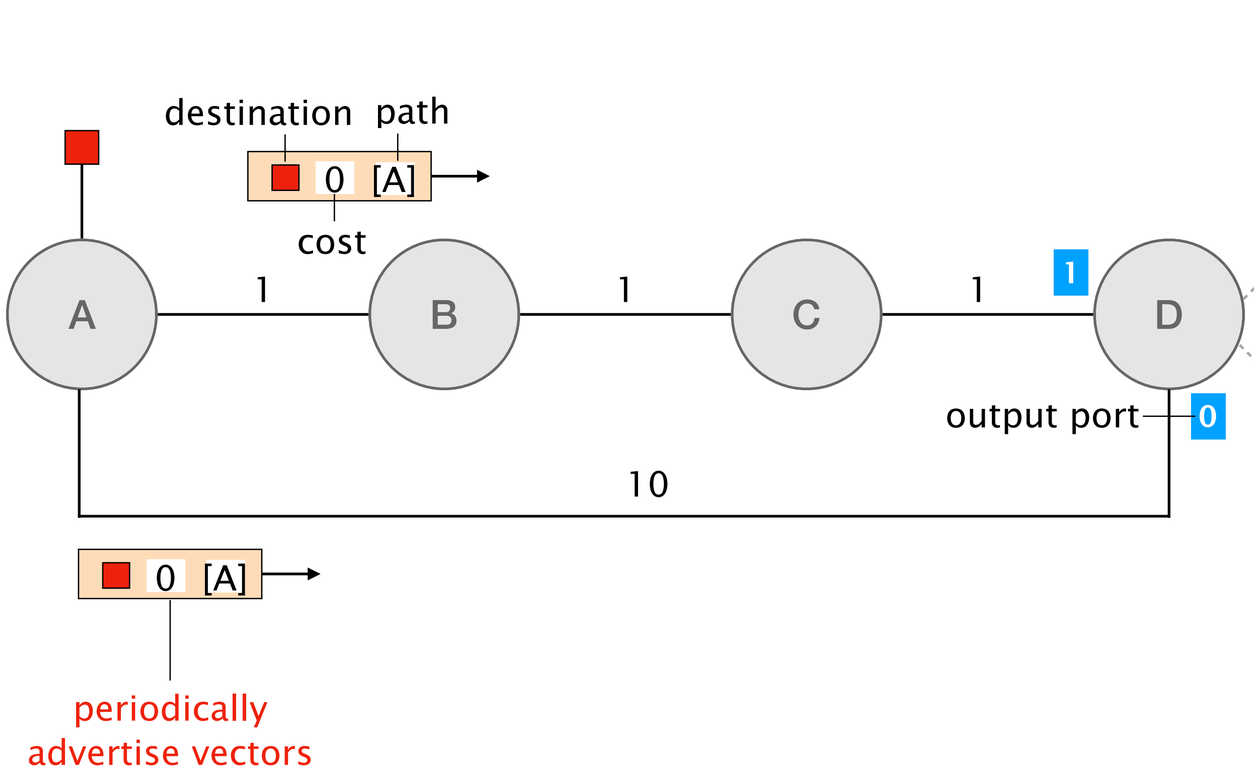
Statically configured tables map prefixes to registers in memory

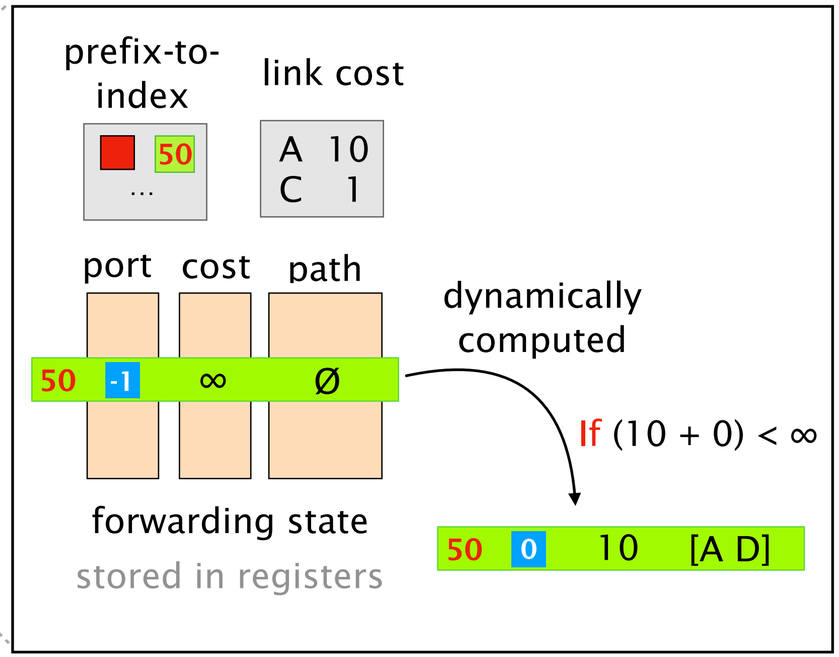


Registers store best paths and its attributes

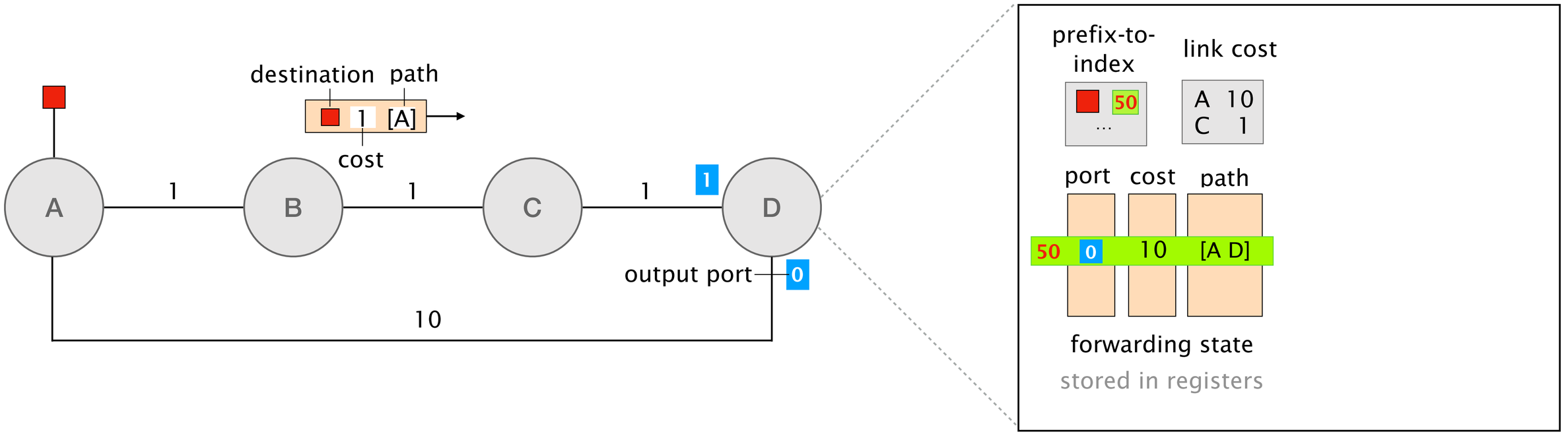


Switches periodically advertise vectors to neighbors

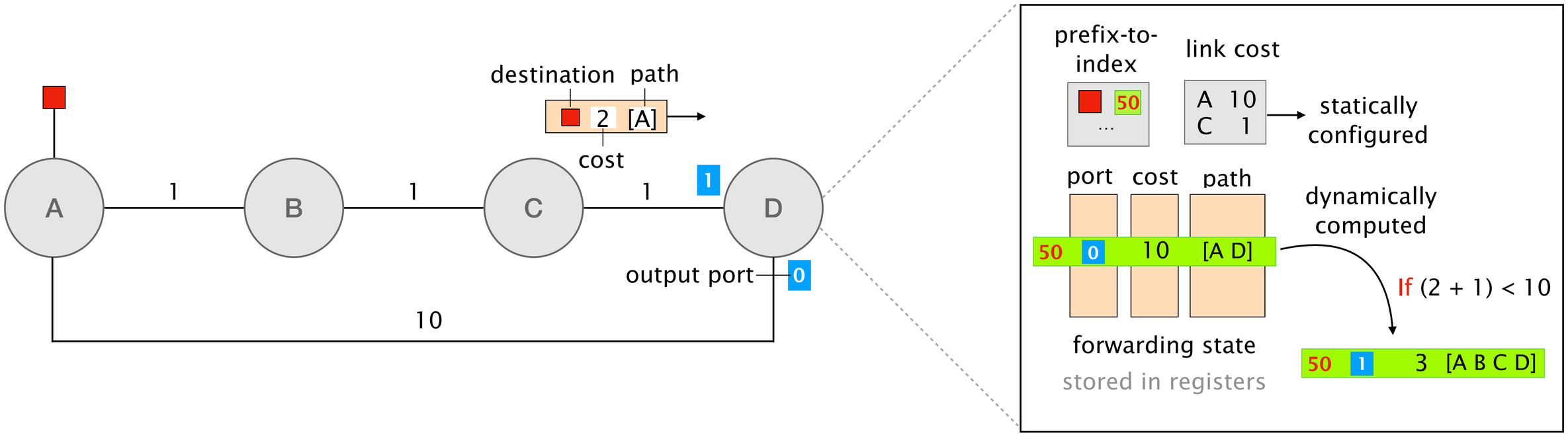




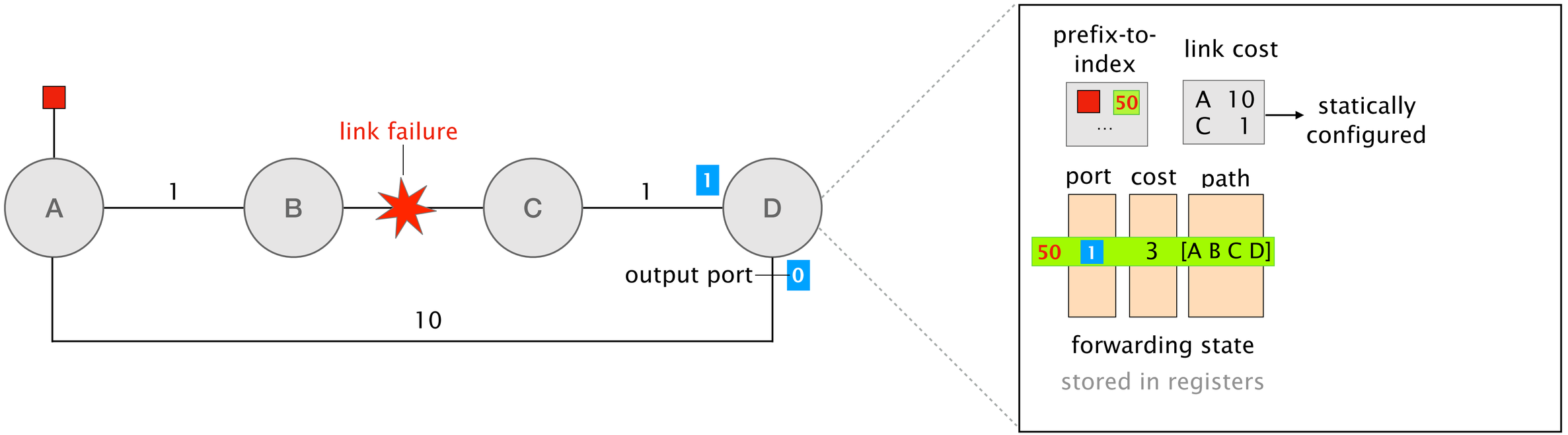
Switches periodically advertise vectors to neighbors



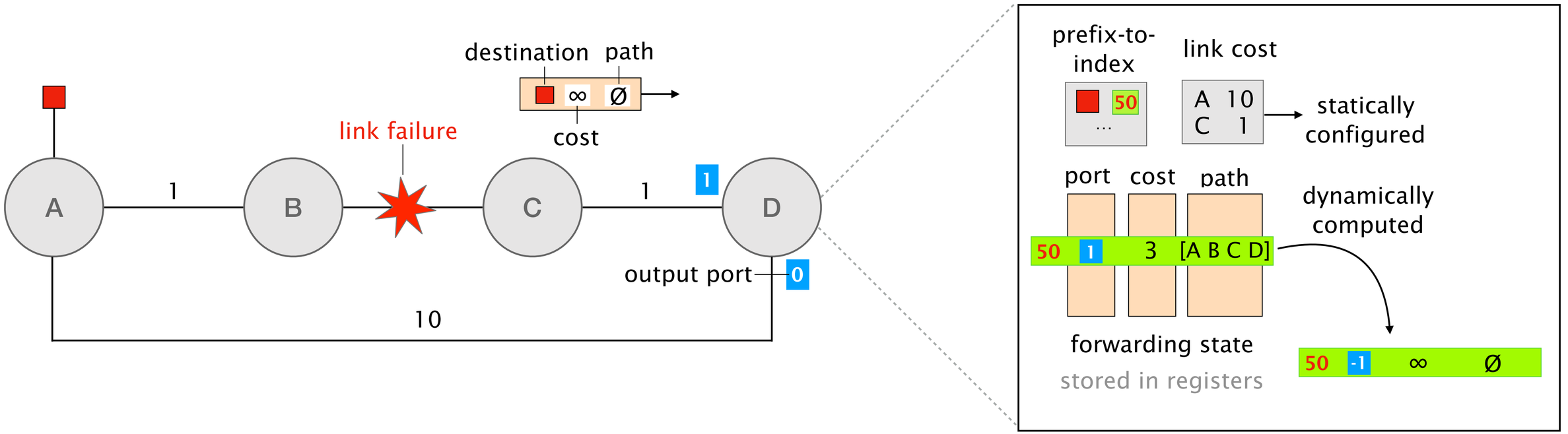
Switches periodically advertise vectors to neighbors



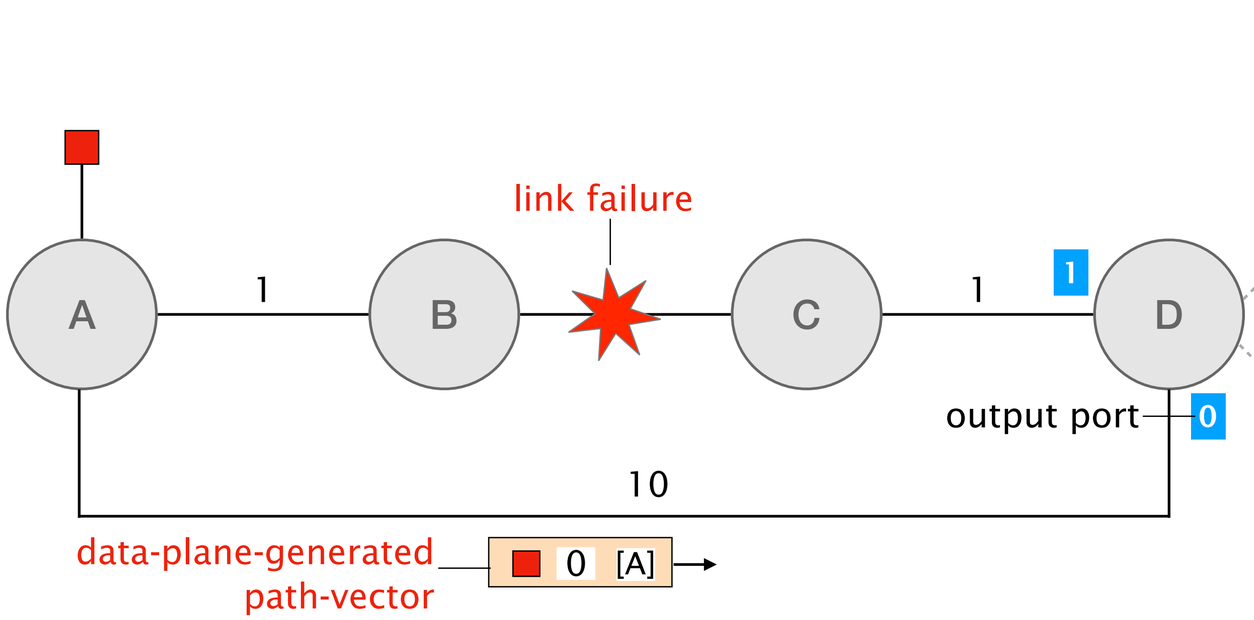
Computing new forwarding state after a a link failure

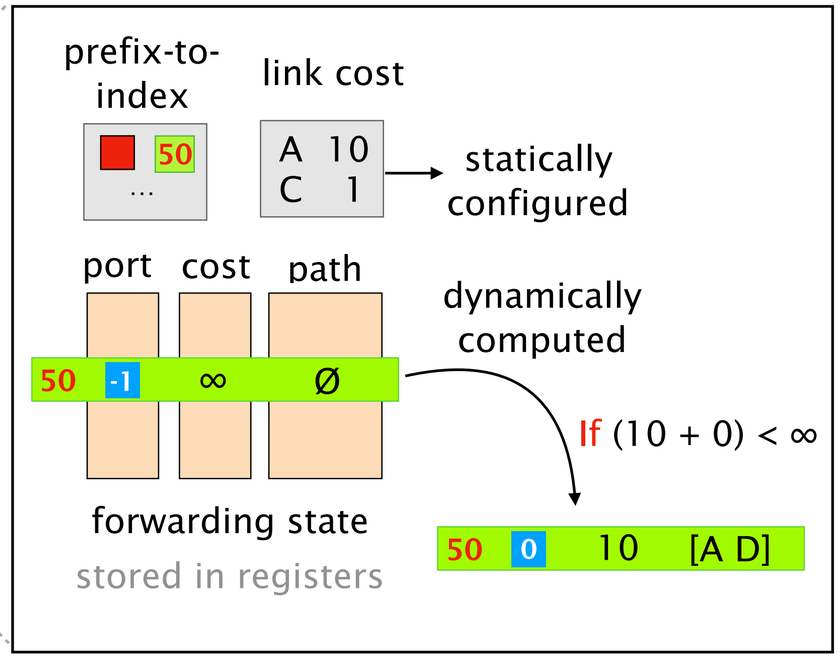


Computing new forwarding state after a a link failure



Computing new forwarding state after a a link failure





Does it actually work?

Does it actually work? ***Yes!***

We built a P4$_{16}$ prototype

(we're working on a Tofino implementation)

Implementation

Implemented in P4$_{16}$

Compiled it to bmv2

2k LoC

Capabilities

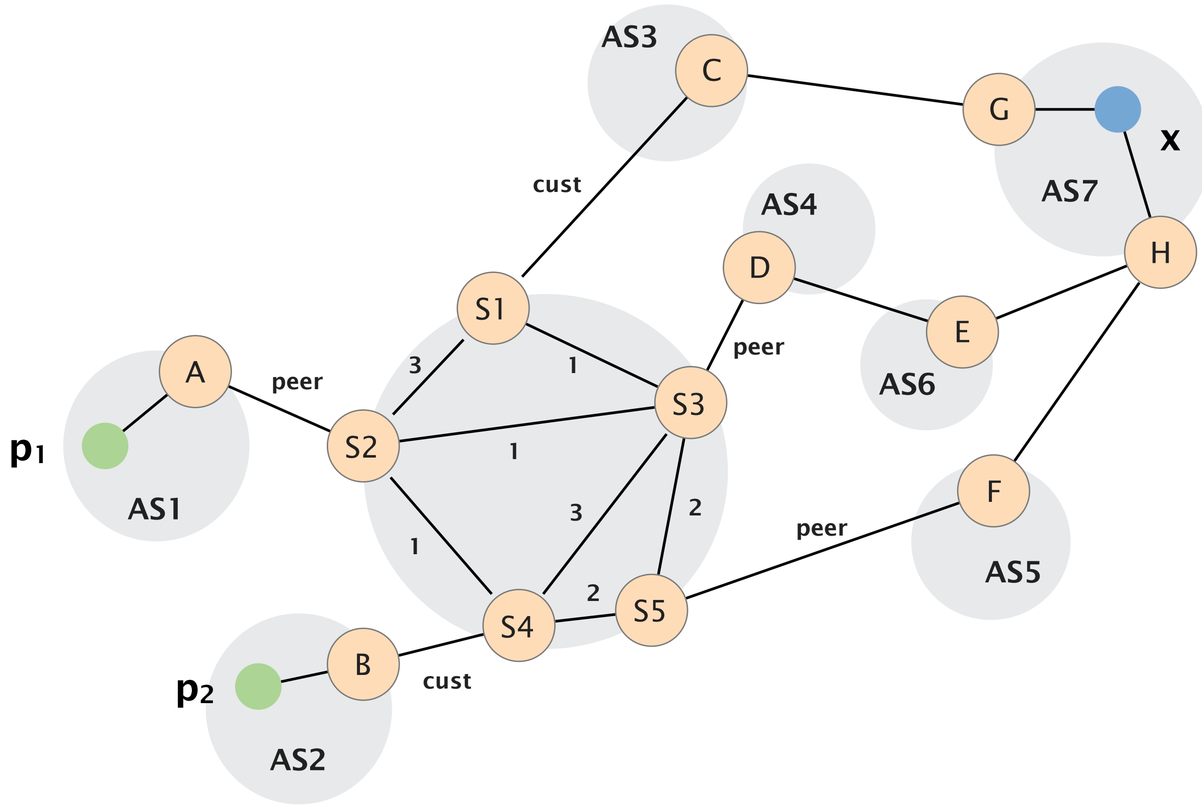
- Intra-domain destinations

  path-vector routing

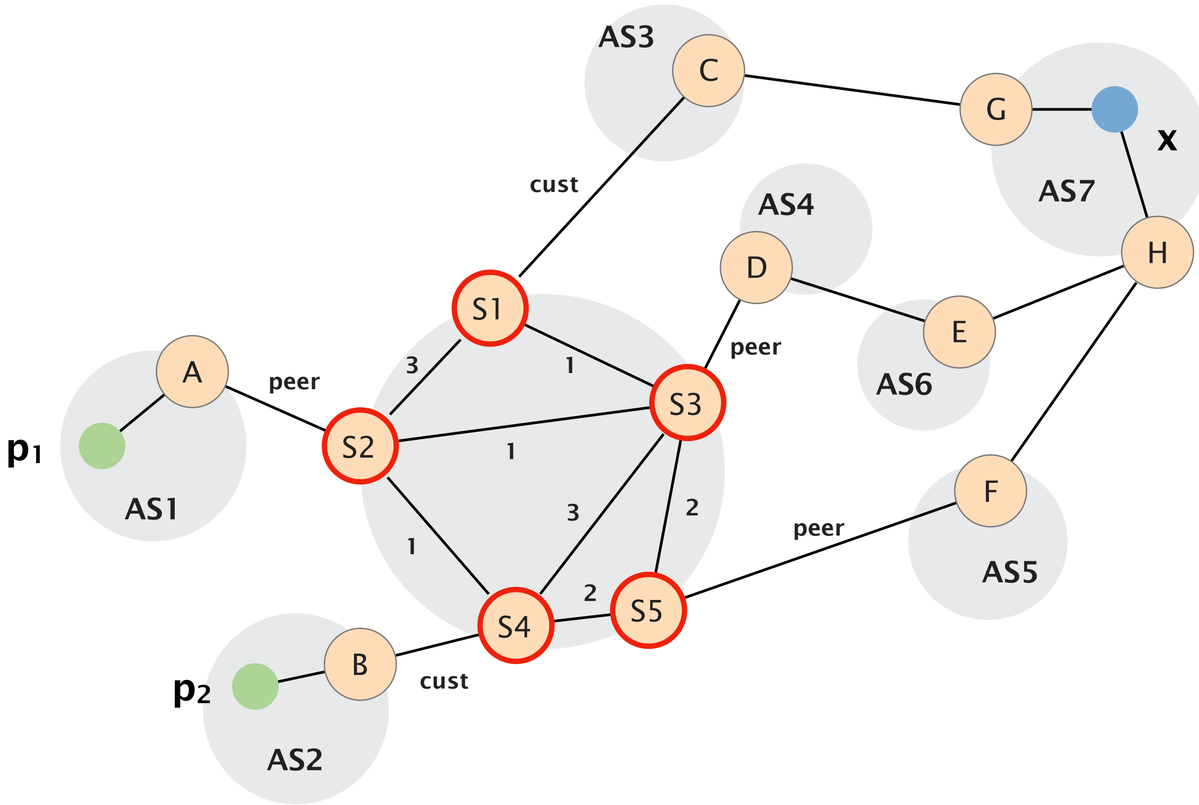
- Inter-domain destinations

  BGP-like route selection

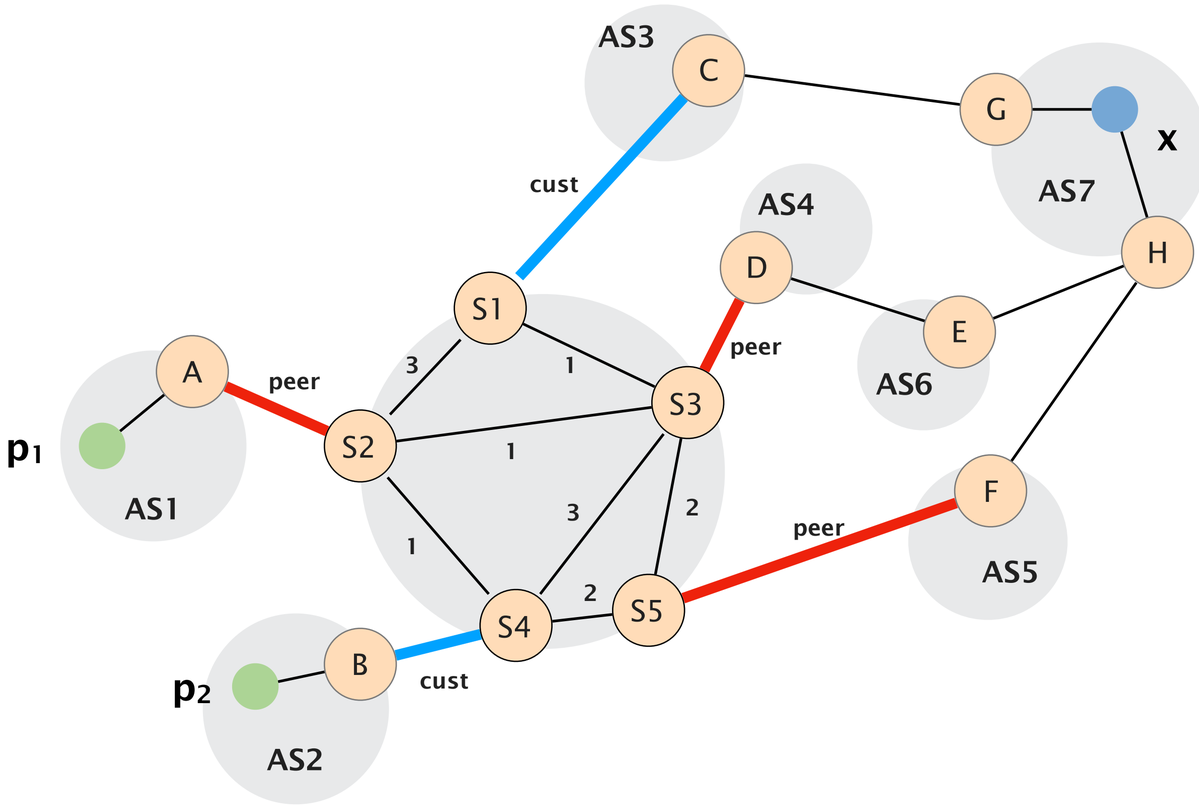
We tested our implementation in a simple case study



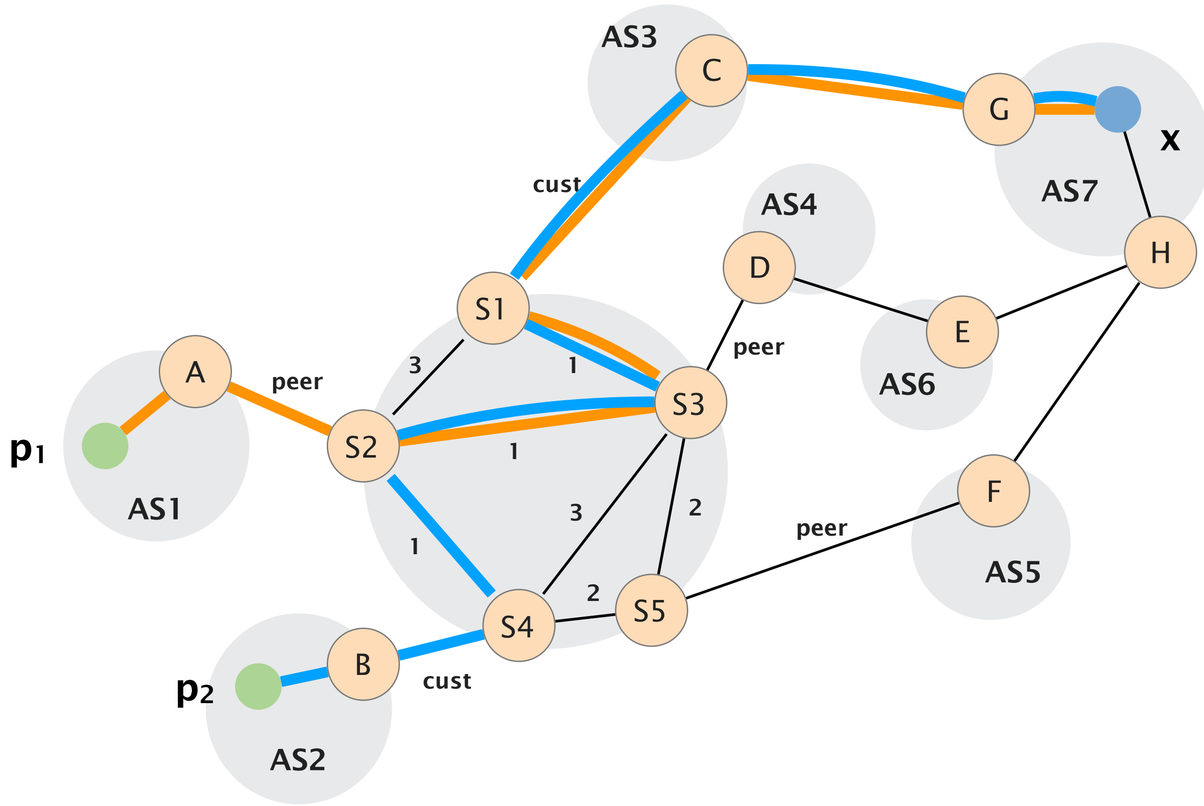
Only the internal switches run the hardware-based control plane



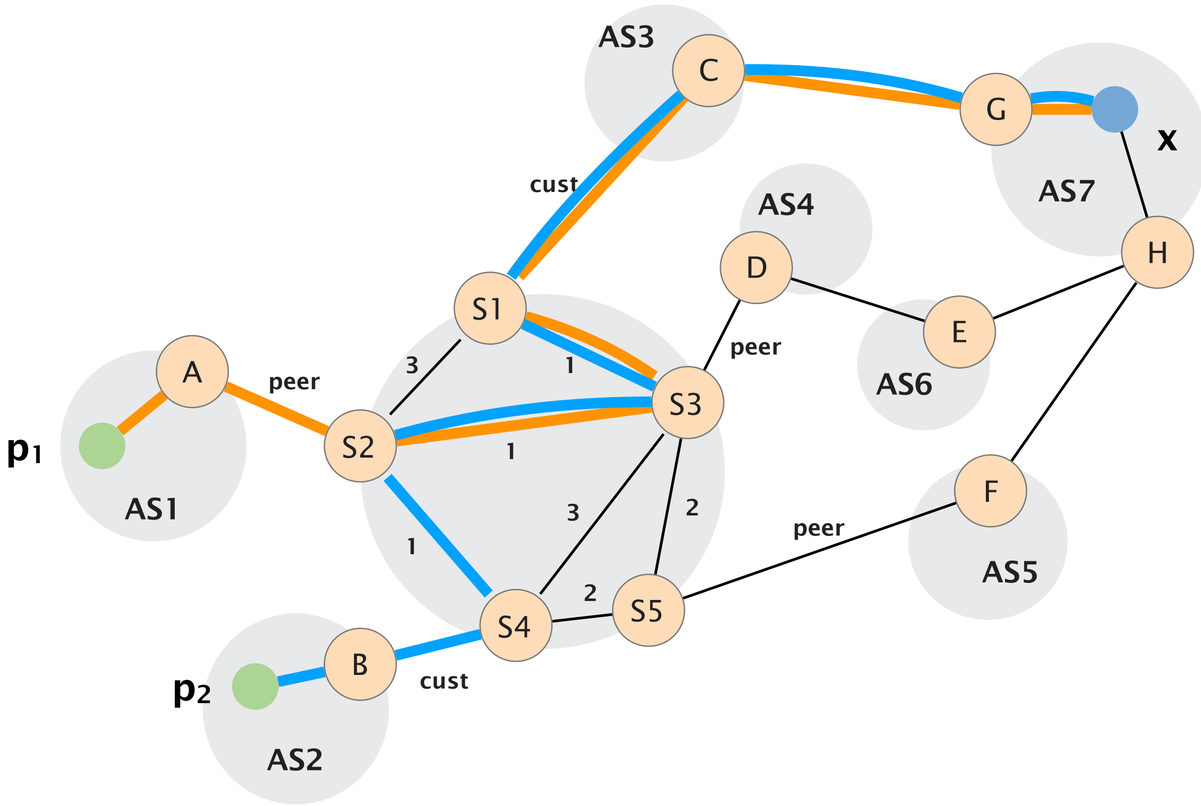
Each switch is connected to an external peer or customer



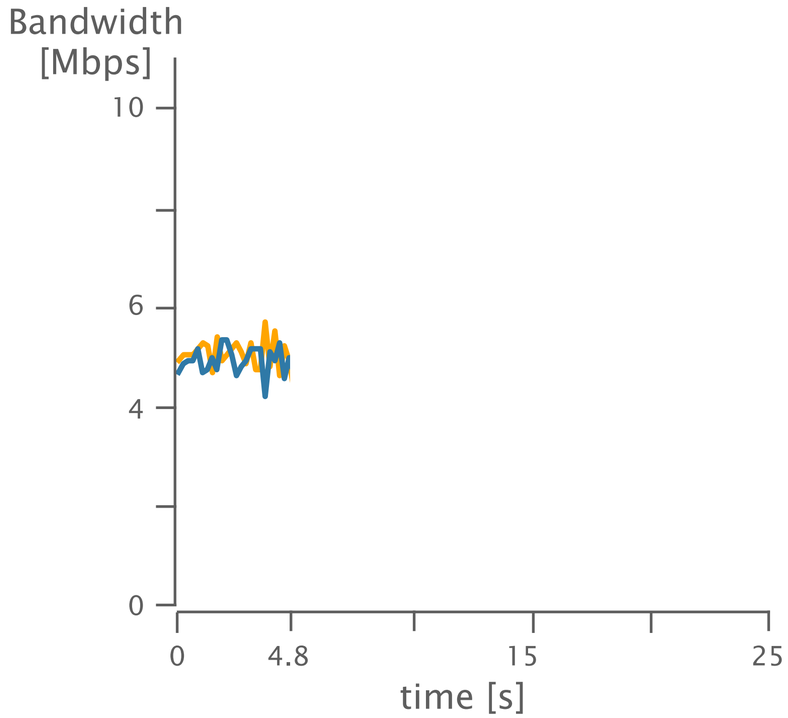
We generate two TCP flows from AS1 and AS2



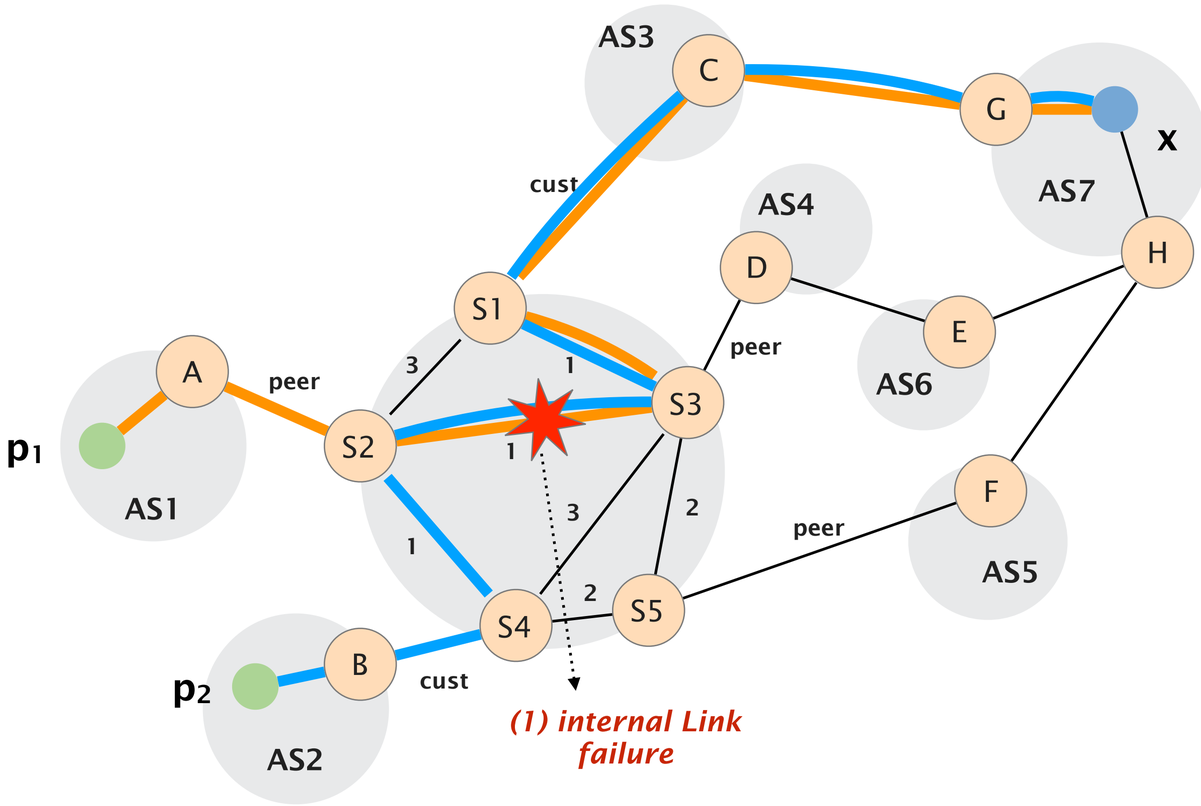
Switches monitor the traffic



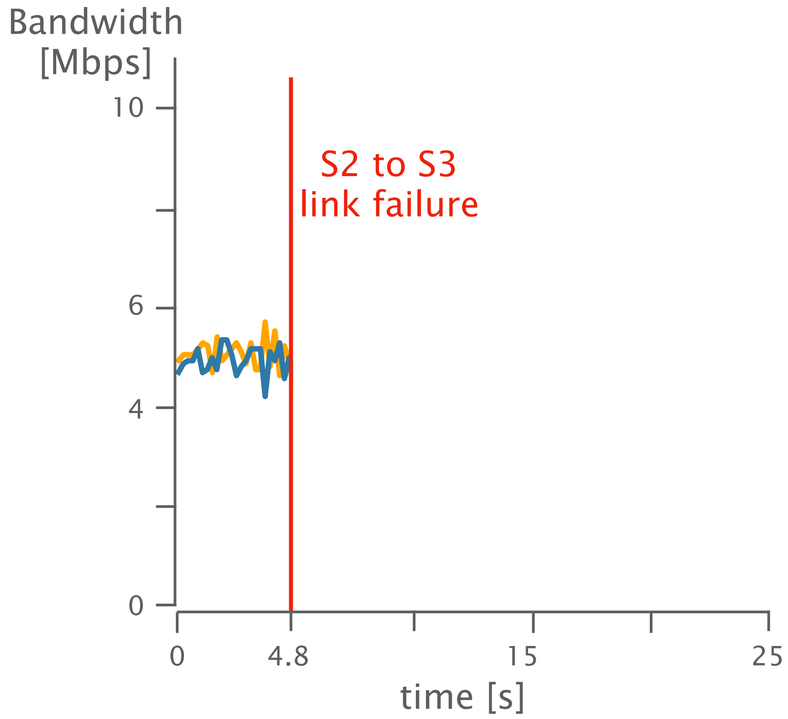
Traffic S1- AS3

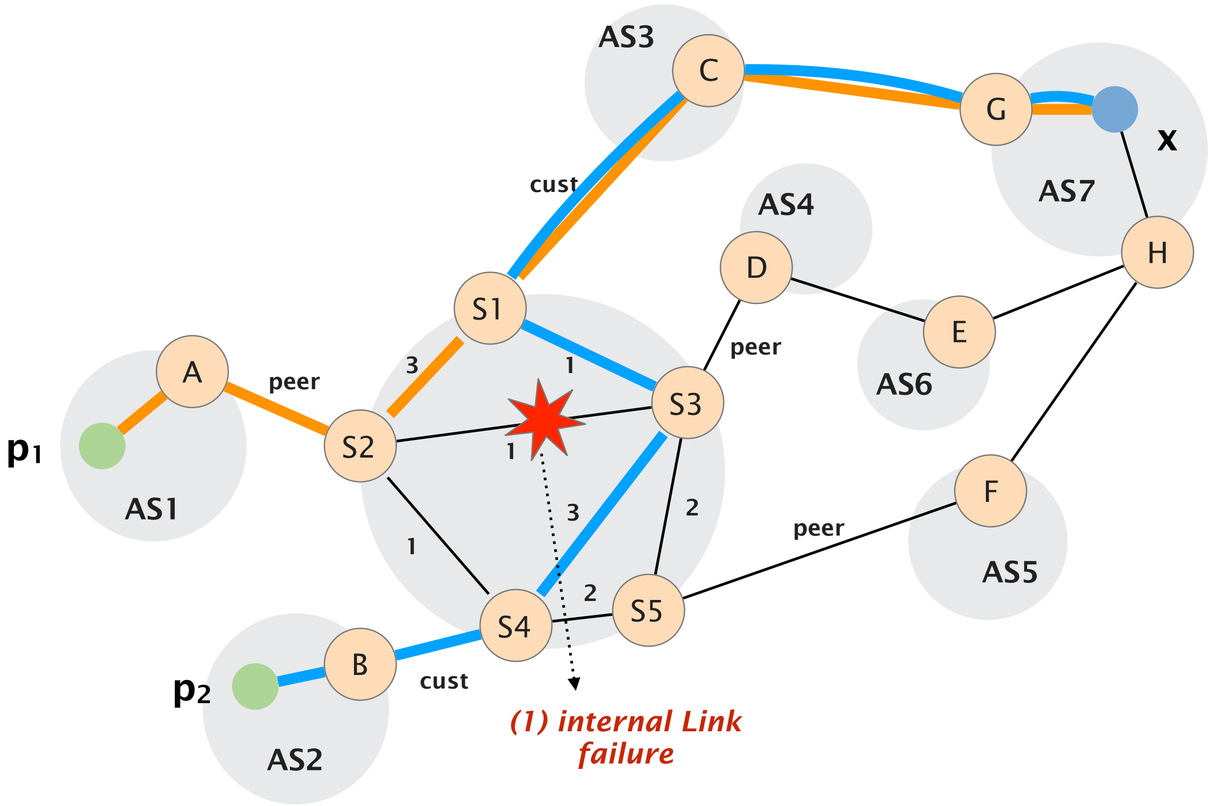


Internal link fails, triggering the path-vector algorithm

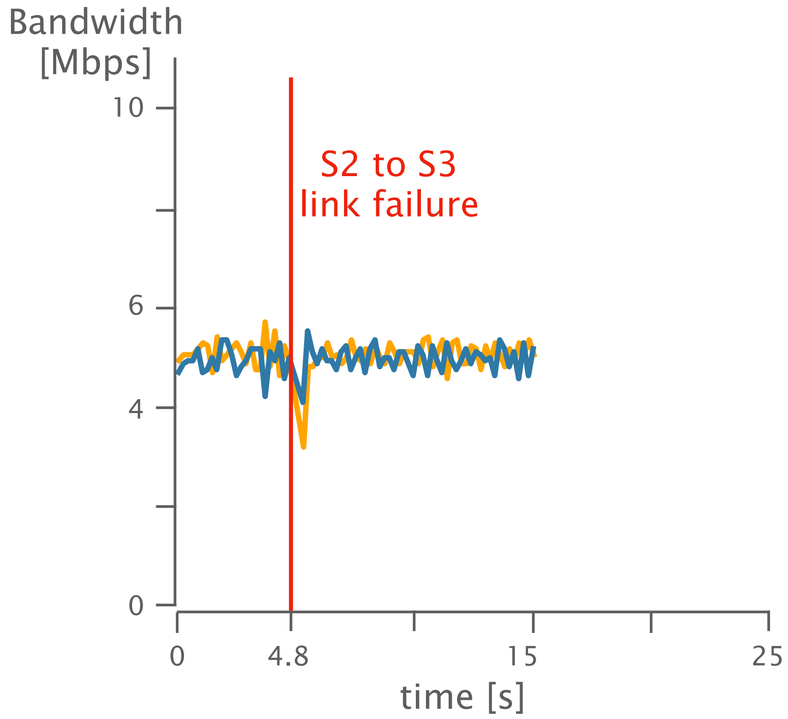


Traffic S1- AS3

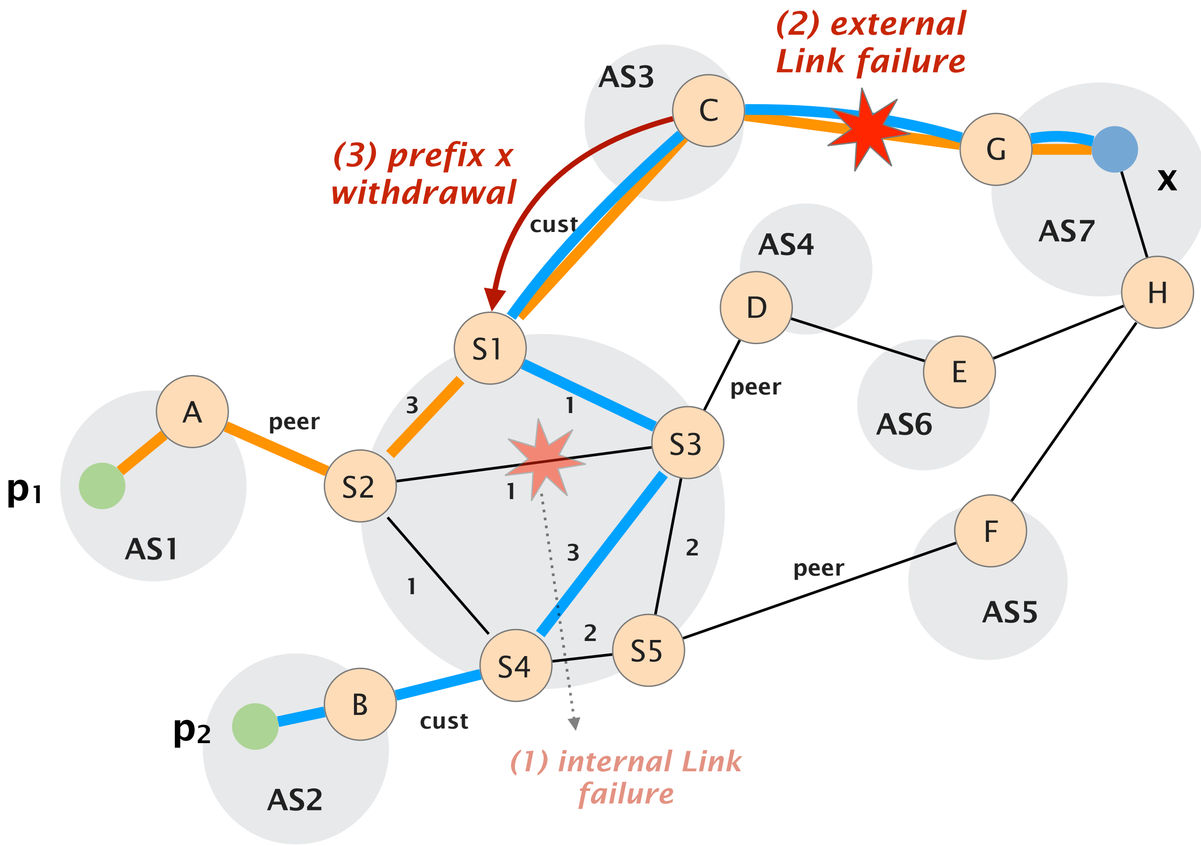




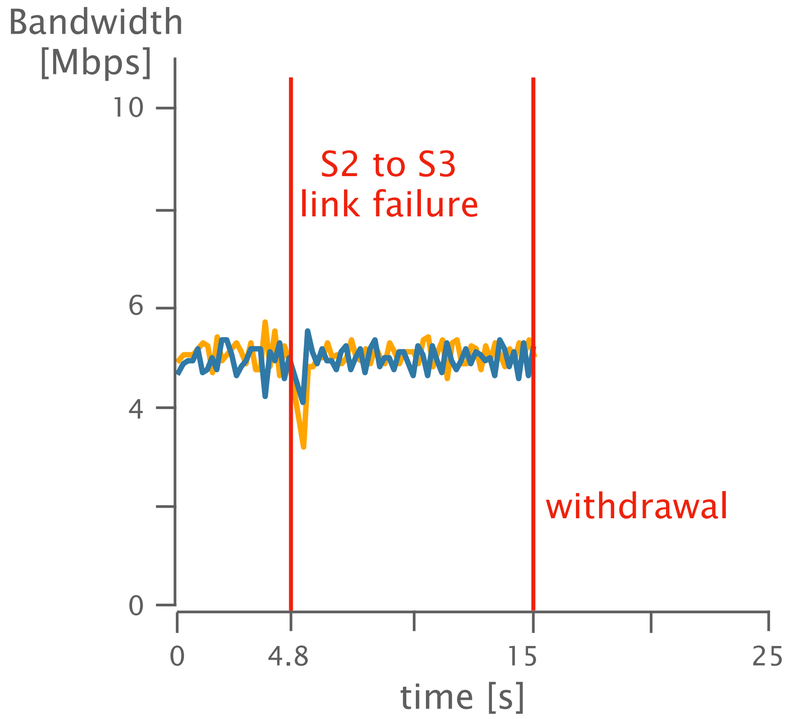
Traffic S1- AS3



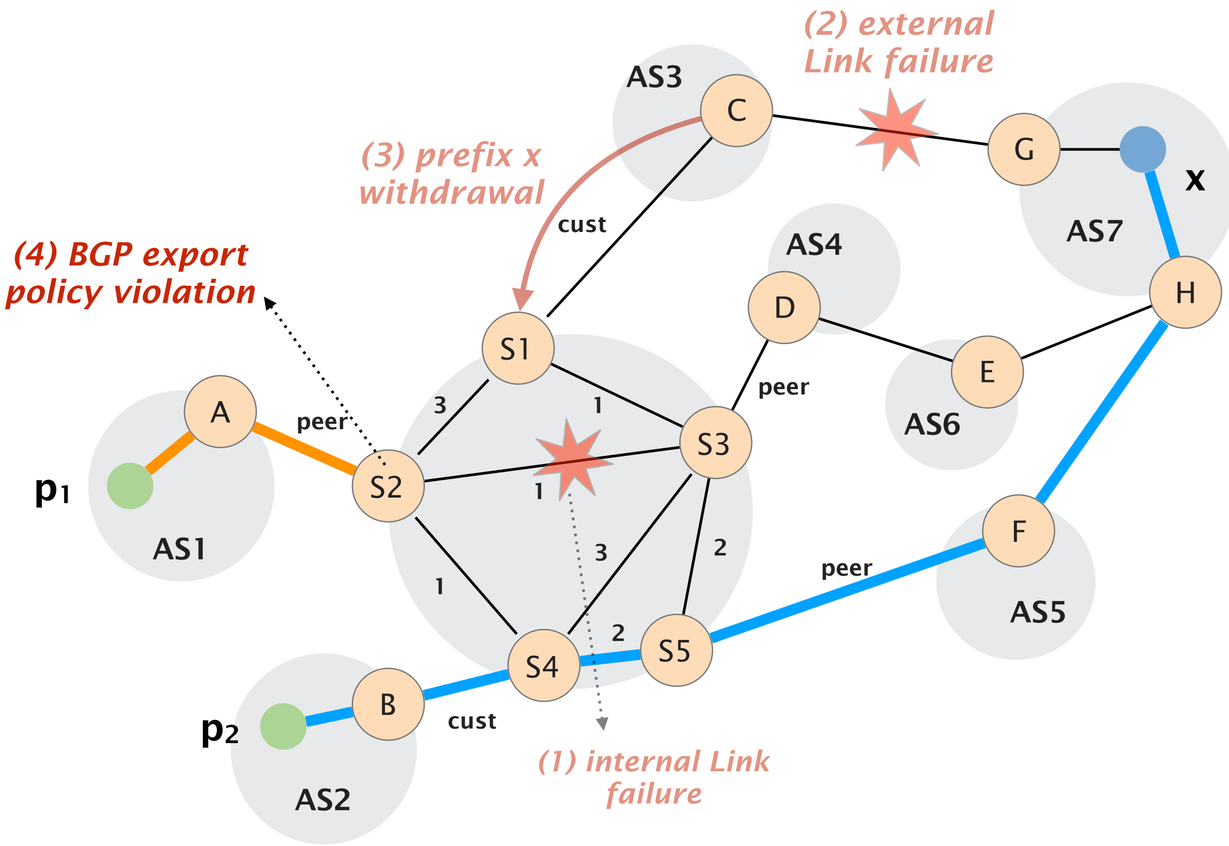
External link failure triggers a prefix withdrawal

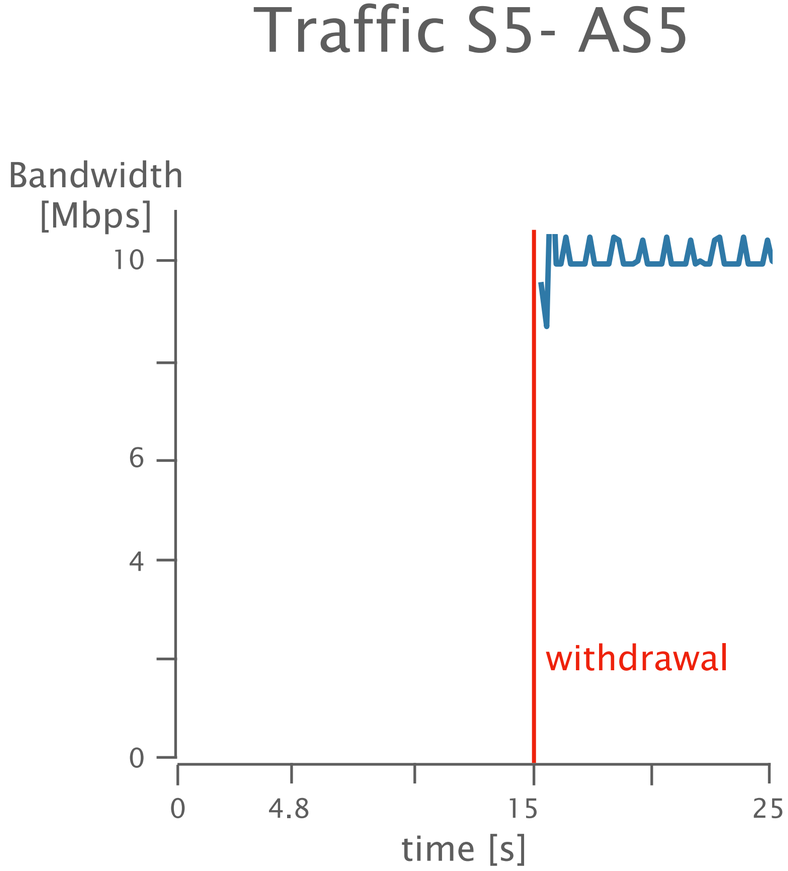


Traffic S1- AS3



Network computes new egress and applies new policies





Could we offload control-plane tasks to the data plane?

Yes… ***but…***

Programmable hardware is not limitless

Programmable hardware is not limitless

Some tasks *cannot* be offloaded

while offloading others is *not desirable*

- Reliable protocols

  e.g. TCP requires too much state

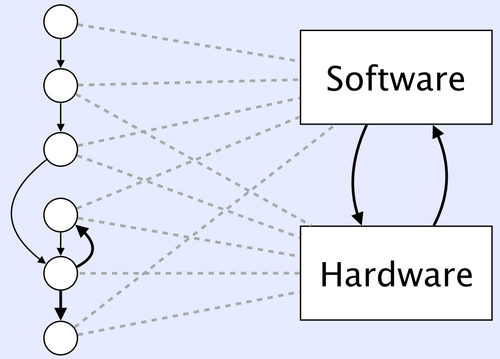
- Poor scalability of control plane tasks

  hardware memory is scarce and expensive

Can we have the best of both worlds?

Hardware-software codesign

Specification



problem graph *mapping set* *architecture graph*

functions

$Cost_i(.)$

$Performance_i(.)$

constraints

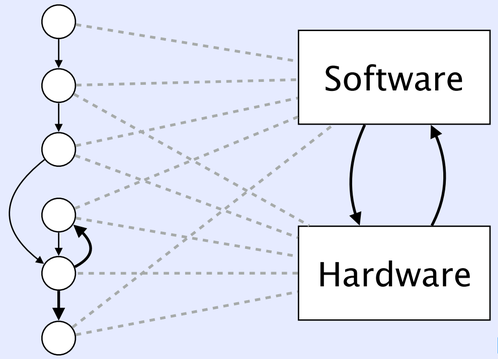
```
∀i:
  pred(i)<100
```

Optimization

Synthesis

Hardware-software codesign

Specification



problem graph *mapping set* *architecture graph*

functions

$Cost_i(.)$

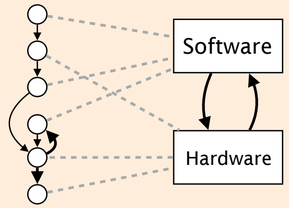
$Performance_i(.)$

constraints

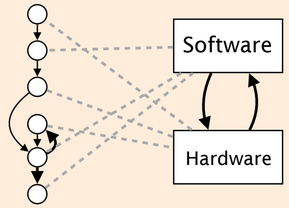
```
∀i:
  pred(i)<100
```

Optimization

```
cost(x):120
perf(x):200
```




```
cost(y):80
perf(y):200
```

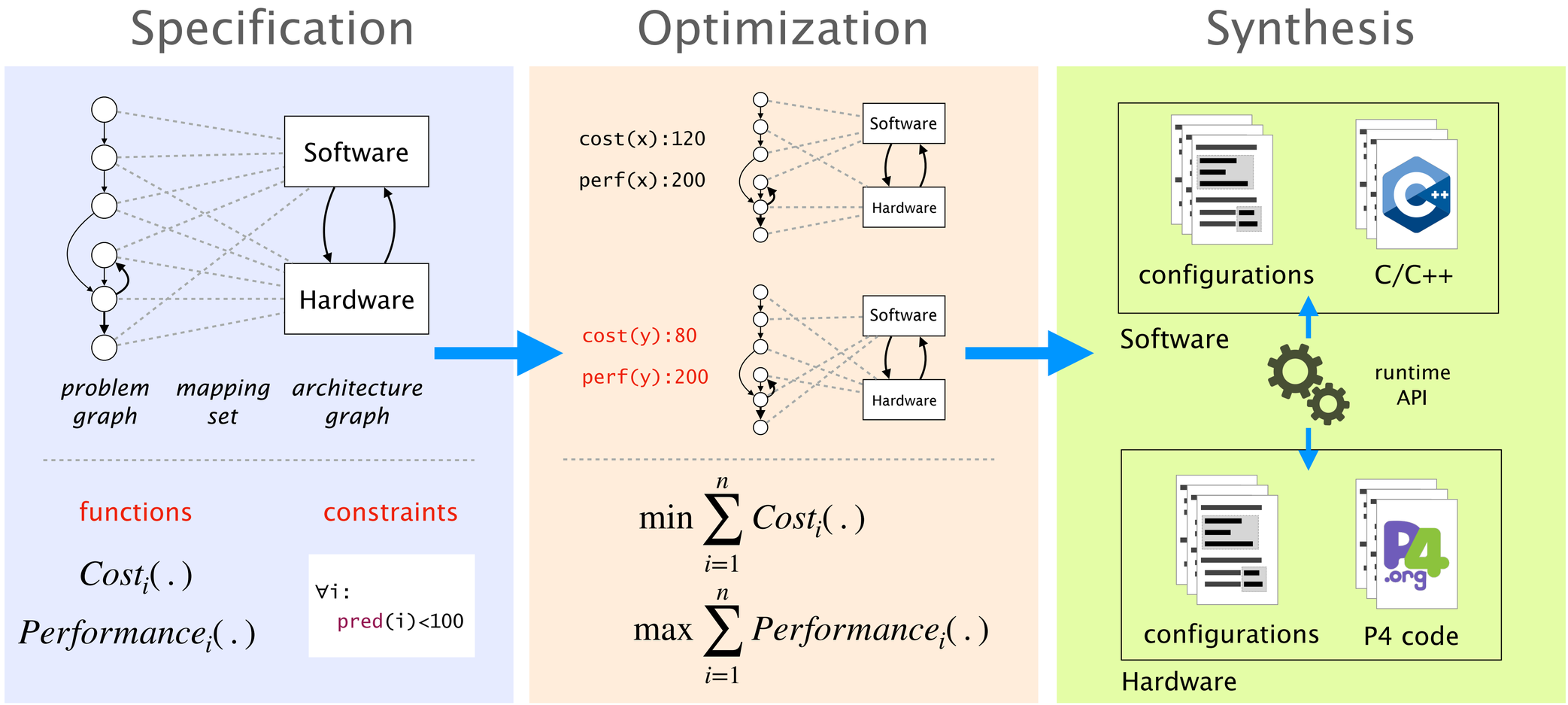


$$\min \sum_{i=1}^{n} Cost_i(.)$$

$$\max \sum_{i=1}^{n} Performance_i(.)$$

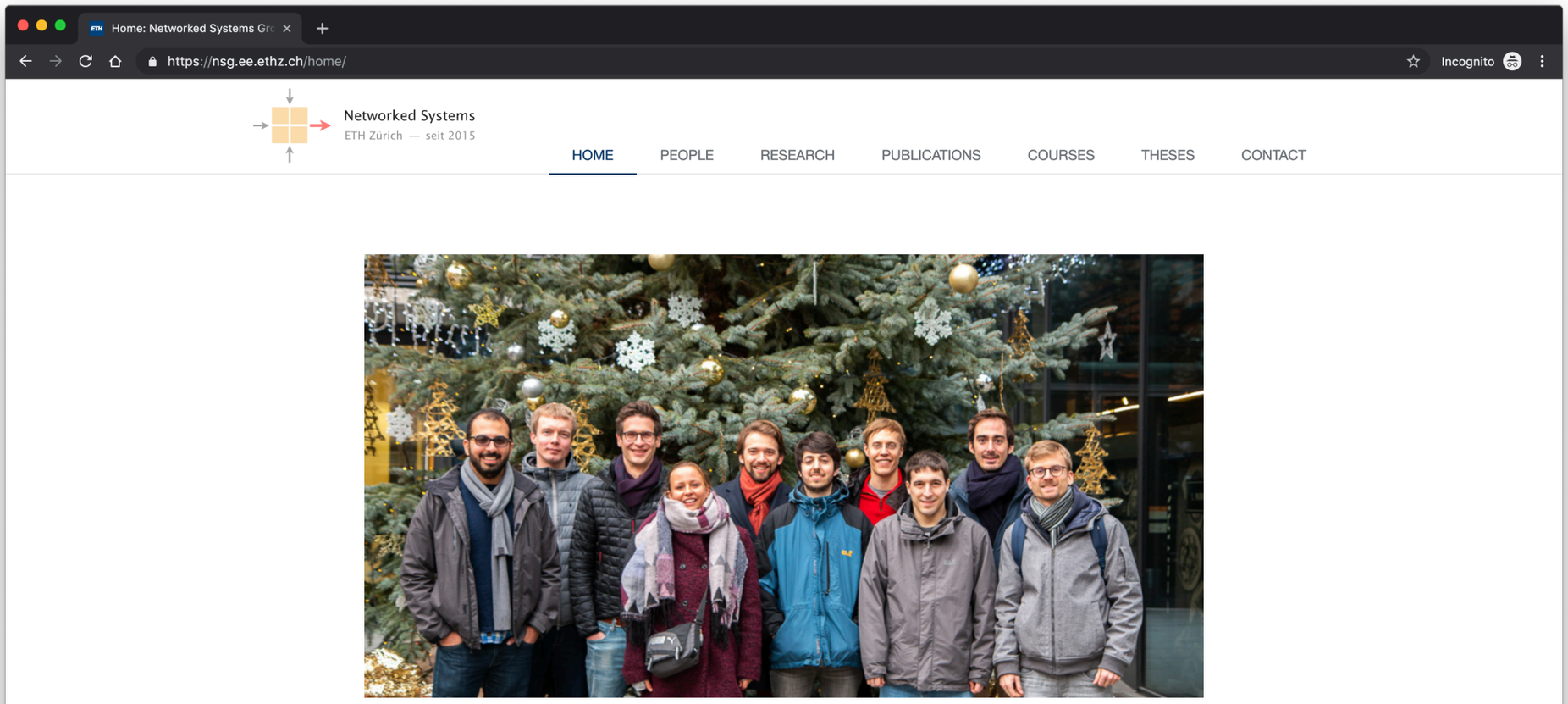
Synthesis

Hardware-software codesign



Find out more about our "quest"

https://nsg.ee.ethz.ch



The Networked Systems Group (NSG) is a research group in the Department of Information Technology and Electrical Engineering (D-ITET) at ETH Zürich led by Prof. Laurent Vanbever. We are also part of the ETH ICE center, a new group initiative on programmable and dependable networked systems.

Our research interests are centered around complex network management problems, with the larger goal of making current and future networks (especially the Internet) easier to design, understand and operate. We are currently active in multiple areas including network programmability, data-driven networking, verification, routing, and security. Most of our projects are inherently multidisciplinary and tend to involve recent advances in programming languages, algorithmics, and machine learning.

A few recent examples of practical systems we have built include: Bayonet, Fibbing, iTAP, Net2Text, NetComplete, NetHide, SDX, SyNET, SDNRacer, Stroboscope, and SWIFT. We are also currently looking at the impact of routing attacks on systems overlays such as cryptocurrencies and anonymity networks. To learn about our work, please check out our research and publications pages.

Our flagship lecture is Communication Networks offered in the Spring semester. We also offer a new lecture on Advanced Topics in Communication Networks in the Fall semester. Check our courses page for more information.

Network Control Planes

How? Where? Why?!

Laurent Vanbever

nsg.ee.ethz.ch

Dagstuhl Seminar

Wed Apr 3 2019