"Improving network flexibility"

Tilmans, Olivier

Abstract

Computer networks are deeply ingrained in our daily lives. We rely on them to place audio calls, to watch movies, or even to automate parts of our houses. Each of these use-cases comes with its own requirements to ensure its proper operation and generates unique traffic patterns. For example, video streams require a large amount of bandwidth from a server to the client, for the duration of the video. Efficiently supporting many requirements, potentially changing over time, requires networks to be flexible. In this thesis, we study and improve two key aspects of network flexibility. First, we tackle the issue of flexible network control by introducing Fibbing, a technique which achieves a central control over distributed routing protocols. We present the theory behind Fibbing using provably-correct algorithms, as well as a prototype controller which is compatible with unmodified commercial routers. Our algorithms scale to large Internet Service Provider (ISP) topologies, and measureme...

Document type : *Thèse (Dissertation)*

Improving network flexibility

Olivier Tilmans

Thesis submitted in partial fulfillment of the requirements
for the Degree of Doctor in Applied Sciences

December 2018

Pôle d'ingénierie informatique
ICTEAM
Université catholique de Louvain
Louvain-la-Neuve
Belgique

Thesis commitee:

Pr. Anja Feldmann	MPI-Inf, Germany
Pr. Charles Pecheur (President)	UCLouvain, Belgium
Pr. Ramin Sadre	UCLouvain, Belgium
Pr. Laurent Vanbever	ETH Zürich, Switzerland
Pr. Olivier Bonaventure (Advisor)	UCLouvain, Belgium

Improving network flexibility

by Olivier Tilmans

Université catholique de Louvain
ICTEAM
Pôle d'ingénierie informatique
Place Saint Barbe, 2
1348 Louvain-la-Neuve
Belgique

This work was partially supported by grants from the EC Seventh Framework Programme (FP7/2007-2013, grant no. 317647—Leone) and from Communauté française de Belgique (ARC grant 13/18-054). This work was also partially supported by a F.R.S.-FNRS FRIA scholarship (Fonds pour la formation à la Recherche dans l'Industrie et dans l'Agriculture, rue d'Egmont 5, B-1000, Bruxelles, Belgique).

Preamble

Projected to carry 3.3 Zetabyte[1] annually by 2021 [Cisd], the Internet has for long stopped to be a small network connecting universities. Indeed, it is now deeply ingrained in our daily lives, as we rely on it to place voice and video calls, watch movies, or get real-time road-congestion notifications which seamlessly update our commute itineraries. Additionally, it also connects automated factories to remote control centers, and enables enterprises to move to virtual office environments where everything is in the cloud.

This ever-growing list of applications comes with vastly different needs of connectivity for each of them, in terms of required bandwidth, expected latencies and losses, or traffic patterns (*e.g.,* bursts). In parallel, the traditional network boundaries are shifting, as end-hosts are getting more and more mobile, while the services they connect to are dynamically instantiated in the cloud. Implementing complex connectivity requirements in this context is thus a major challenge, as it requires network to continuously adapt to changes.

Today's networks are often statically configured to forward traffic on a best-effort basis, using historical traffic demands to optimize their forwarding paths. Indeed, most networks are composed of vertically-integrated routers (*e.g.,* routers from vendors such as Cisco or Juniper), use distributed routing protocols (*e.g.,* bundled as part of IOS, JunOS), and rely on coarse-grained randomly sampled measurements (*e.g.,* NetFlow). We say that these networks are inflexible, as adapting to new requirements is challenging without causing service disruptions [Jun08; Ver16].
While Software-Defined Networking [Kre+15] has been touted as the network architecture that will support tomorrow's use-cases due to its tremendous flexibility, we argue that current implementations bring challenges of their own. Moreover, deploying SDN through a clean-slate approach is unpractical for most networks, forcing an incremental deployment of SDN features co-existing with today's protocols.

[1] 10^{21} bytes

In this thesis, we study how to improve network flexibility. In particular, we focus on two research questions, which are key to enable tight control-loops as needed by future networks.

- *How can we adapt forwarding requirements in today's networks?*
- *How can we evaluate whether current requirements are met?*

Drawing inspiration from the strengths of the SDN paradigm, answering the first question requires to identify a new primitive enabling to centrally control distributed routing protocols. Answering the second implies to be able to monitor the behavior of a network in real-time, extracting fined-grained measurements which go beyond bandwidth estimation.

This thesis is structured in three parts. The first part introduces key concepts forming the background behind today's networks (§1).

Then, the second part tackles the issue of flexible network control. We begin by presenting a technique to achieve a centralized control over distributed routing protocols (§3). More specifically, we first motivate why today's networks have a hard time to adapt to change, and introduce Fibbing, a hybrid network architecture bringing SDN control to distributed routing protocols (§2). Then, we present the theoretical foundations of Fibbing (§3). In particular, we show how Fibbing supports advanced routing requirements in link-state routing protocols, by using provably correct algorithms to inject fake nodes and fake links in the routers' shared view of the network. Afterwards, we describe an implementation of Fibbing which controls OSPF networks (§4). We show that this implementation is able to control unmodified, off-the-shelf routers, and that it can scale to large networks and number of requirements with little overhead. We conclude this part by discussing new research perspectives opened by Fibbing (§5).

The last part of this thesis explores how to improve the flexibility of network monitoring systems. More specifically, we focus on two kinds of networks which have poor visibility over their traffic. On one hand, we introduce Stroboscope, a monitoring framework that combines the visibility of traffic mirroring with the scalability of traffic sampling (§6). Stroboscope enables ISP networks to collect fined-grained traffic samples, at scale, while actively controlling the monitoring overhead, in order to enable advanced analyzes such as estimating one-way delays. On the other hand, we present Flowcorder, a monitoring framework enabling enterprise networks to precisely monitor the performance of protocols as experienced by the end-hosts (§7). Flowcorder supports transparently encrypted and multipath protocols, as it directly instruments the protocol implementations on the end-hosts to collect connection performance profiles. Chapter (§8) concludes this third part.

We conclude this thesis and discuss future research in the last chapter (§9).

Bibliographic notes

Most of the work presented in this thesis appears in conference proceedings. The list of accepted and submitted publications is presented hereafter.

- Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. "Central Control Over Distributed Routing". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. 2015, pp. 43–56. DOI: 10.1145/2785956.2787497

  - Selected as best paper.

  - Received an IRTF Applied Networking Research Prize (ANRP).

- Olivier Tilmans, Stefano Vissicchio, Laurent Vanbever, and Jennifer Rexford. "Fibbing in Action: On-demand Load-balancing for Better Video Delivery". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. 2016, pp. 619–620. DOI: 10.1145/2934872.2959084

- Olivier Tilmans, Tobias Bühler, Stefano Vissicchio, and Laurent Vanbever. "Mille-Feuille: Putting ISP Traffic Under the Scalpel". In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. HotNets '16. 2016, pp. 113–119. DOI: 10.1145/3005745.3005762

- Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. "Stroboscope: Declarative Network Monitoring on a Budget". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 467–482. ISBN: 978-1-931971-43-0

- Olivier Tilmans and Olivier Bonaventure. "COP^2: Continuously Observing Protocol Performance". In: *Submitted to ACM EuroSys '19*. 2019

Additionally, while not strictly a part of this thesis, the following paper was its starting point.

- Olivier Tilmans and Stefano Vissicchio. "IGP-as-a-Backup for Robust SDN Networks". In: *Proceedings of the 10th International Conference on Network and Service Management*. CNSM '14. 2014, pp. 127–135. DOI: 10.1109/CNSM.2014.7014149

Finally, while implementations are not scientific contributions in themselves, all the systems and tools presented in this thesis have been fully implemented and thoroughly tested. To ensure that our results are reproducible, all these implementations are freely available under open-source licenses in the following repositories:

Fibbing (§II)	`https://github.com/Fibbing`
Stroboscope (§6)	`https://github.com/net-stroboscope`
Flowcorder (§7)	`https://github.com/oliviertilmans/flowcorder`

Acknowledgments

During the four years of research summarized in this thesis, I had the chance to meet, discuss, and be supported by several people.

First and foremost, I would like to thank Prof. Olivier Bonaventure for being a great advisor. Throughout the years, he continuously presented me with challenging ideas, as well as provided me with unique perspectives that enabled me to move forward and to improve my work. I am profoundly grateful for the considerable amount of time he spent following my progress, supporting me when needed, or proofreading. Moreover, I would like to thank him for giving me the opportunity to pursue a PhD.

I owe a special thanks to Stefano Vissicchio, who mentored me throughout my master thesis and then most of my PhD research. Through in-depth discussions, perpetual questioning of my results, and "blackboard sessions", Stefano taught me how to be a scientist.

I would like to thank Anja Feldmann, Charles Pecheur, Ramin Sadre, and Laurent Vanbever for their involvement in the jury of this thesis. Their insightful and sharp comments on my work enabled me to grasp the bigger picture around this thesis.

This thesis is the results of collaborations with other persons. I would like to thank my co-authors, Jennifer Rexford, Tobias Büehler, and Ingmar Poese for the work we jointly realized. Moreover, Tobias provided me with many helpful suggestions which turned my stays at ETH Zürich into a very enjoyable experience. Jo Segaert from BELNET and Dave Ward, Clarence Filsfils and Kris Michielsen from Cisco Systems for their support in testing Fibbing on real routers. Lynne Salameh and Roland Meier for their insightful comments and help on Stroboscope. Nicolas Detienne, Anthony Gego, François Michel, and Maxime Piraux for their help getting Flowcorder deployed.

I would like to thank my former colleagues from the IP Network Lab, Raphael Bauduin, Marco Chiesa, Quentin De Coninck, Fabien Duchêne, Benjamin Hesmans, Mathieu Jadin, Nicolas Laurent David Lebrun, and Viet-Hoang Tran for the many discussions we had, which indirectly influenced this thesis.

Additionally, I would like to thank the support staff from INGI, particularly Vanessa Maons and Sophie Renard, who helped me to navigate the realms of the administration. Finally, I would like to thank the INGI members as a whole for the informal (and sometimes surprisingly in-depth) discussions we shared during lunches at the cafeteria.

Last but not the least, I spent these last years surrounded by friends and my family who supported me and encouraged me. In particular, I would like to thank my wife, Lucile, who stood by my side day and night, coped with my impossible schedules, and whose support meant everything to me.

Contents

Part I

Background

1 Operating networks

The goal of computer networks is to enable the exchange of data between different machines, also known as end hosts. Most networks achieve this by using IPv4 and IPv6, the Internet Protocol versions 4 [RFC791] and 6 [RFC2460]. In this protocol, end-hosts are identified using an IP address. To send data to another host, a sender then first has to chunk it into packets, then has to write its own IP address and the one from the receiver as the source and destination addresses in an IP header. As end-hosts are rarely directly connected to one another, these packets are then relayed by intermediate nodes, routers, until they reach their destination, using a process called IP forwarding. The Internet is composed of the interconnection of multiple intermediate Autonomous Systems (ASes)—*i.e.,* networks operated by a single administrative entity. Exchanging data over the Internet then implies to forward IP packets across one or more ASes. This forwarding process is the keystone of the Internet. Indeed, to scale, each AS is able to autonomously define how each packet should be forwarded, and can selectively exchange reachability information with its neighbors. More importantly, this autonomy enables ASes to optimize their traffic flow, in order to maximize their resource utilization—*i.e.,* define along what exact succession of routers (path) and with what priority each packet should be forwarded. As such, one of the primary tasks when operating a network is to manage this forwarding process.

This chapter briefly presents the concepts behind IP forwarding, and how operators can optimize packet forwarding in their networks. To that end, we first present a high-level model of an IP router (§1.1), as found in most ISP networks. Then, we describe how forwarding paths are defined within an AS (§1.2). Next, we describe how networks are monitored, enabling them to be troubleshooted as well as providing the necessary data to optimize their forwarding paths (§1.3). Finally, we outline a recent network paradigm shift, Software-Defined Networking (SDN), which leverages programmable hardware to enhance network flexibility (§1.4)

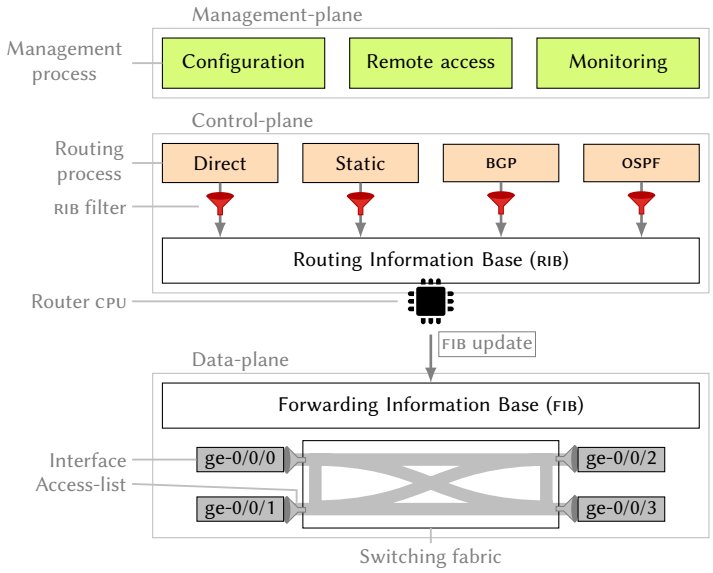


Figure 1.1: IP routers's functions and components can be organized in three sets of planes: data, control and management.

1.1 Modeling an IP router

Fig. 1.1 depicts a high-level architecture of an IP router. IP routers are made of three sets of components, referred to as planes.

Data-plane. This plane achieves the primary function of an IP router: switching IP packets received on one of its interface, the *ingress* interface, to output it on an *egress* interface. Beside forwarding packets, the data-plane can also filter them and/or modify them, by using *access-lists*—*e.g.,* to drop unauthorized packets, or to set the value of some fields in the IP header. All data-plane operations are controlled by corresponding entries in the Forwarding Information Base (FIB). For example, pure IP forwarding requires to perform a longest-prefix-match lookup in the FIB to extract the nexthop associated to the destination address of the packet. Note that the data-plane operates independently of the router's CPU as all its operations are hardware primitives.

Control-plane. The control-plane consists of all the algorithms and protocols that manage the content of the FIB. The control-plane components are mostly implemented in software, and often come bundled with the router Operating System (OS). To manage the content of the FIB, multiple routing processes add routes in the Routing Information base (RIB). This is a data structure stored in the router's memory which associates IP destination prefixes to IP nexthops or local interfaces. As the RIB is updated by the routing

processes, the router OS pushes these changes to the FIB.

Fig. 1.1 shows four routing processes often found on routers. The *Direct* routing process handles the directly connected routes—*i.e.,* the IP addresses and prefixes that are reachable on the local network attached to the routers' interfaces. The *Static* routing process keeps track of routes statically provisioned by the operator, through the router configuration. Finally, Fig. 1.1 lists two of the most popular routing protocols, BGP [RFC4271] and OSPF [RFC2328], each defining another routing process. Routing protocols enable routers to exchange information with one another, in order to learn how to reach various prefixes. On one hand, the *Border Gateway Protocol* (BGP) enables routers from one AS to exchange the routes they know with routers belonging to another AS. BGP is thus an inter-domain routing protocol. On the other hand, *Open Shortest-Path First* (OSPF) is an Interior Gateway Protocol (IGP), as it enables routers to learn how to reach prefixes located within the same AS (§1.2).

All of these routing processes result in routes which can be installed in the RIB. To control whether a given route can be used by a router or not (*i.e.,* installed in the RIB), *RIB filters* can be configured by operators. For example, to prevent BGP hijacks, an operator can filter out all BGP routes concerning its own prefixes or those using private IP address ranges (*e.g.,* `10.0.0.0/8` in IPv4). By default, two routes towards the same IP prefix are differentiated according to the Administrative Distance AD of the routing process that computed it (*e.g.,* to favor routes coming from more "reliable" routing protocols).

Management-plane. The management-plane controls the overall behavior of the routers, by the means of its global *configuration*. This configuration defines all the active routing protocols, their specific options, as well as other properties of the routers such as ACLs. The traditional way to configure routers is to use a Command-Line Interface (CLI), which is an interactive text-based interface supporting a vendor-specific configuration language. This CLI can be accessed on the router's serial port (*i.e.,* a management interface), or *remotely* (*e.g.,* using the Secure Shell protocol—SSH [RFC4251]). As large ISP networks can easily have more than a hundred routers, this manual configuration process hardly scales and often leads to operational issues [Ver16; Jun08]. As such, remote configuration protocols (*e.g.,* NETCONF [RFC6241]) combined with a data model (*e.g.,* YANG [**yang**]) have emerged (*e.g.,* OpenConfig [Ope]).

Last but not the least, the management-plane provides *monitoring* (§1.3) functions that provide statistics about both the router itself (*e.g.,* RAM and CPU usage) and the traffic flowing through its interfaces. These statistics can be either collected by a Network Management System (NMS)—pull-based monitoring, *e.g.,* using the Simple Network Management Protocol (SNMP [RFC1157])—, or automatically exported by the monitoring process—push-based monitoring, *e.g.,* NetFlow [RFC3954].

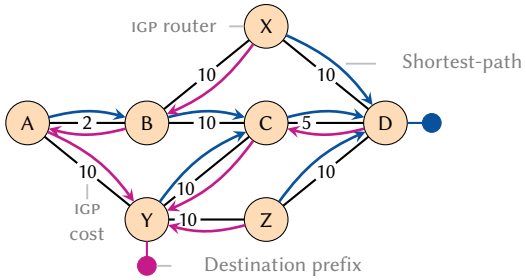


Figure 1.2: In link-state IGPs, routers discover the complete network topology and compute the shortest-paths to reach all prefixes.

1.2 Intra-domain routing

Interior Gateway Protocols (IGPs) enable routers to exchange reachability information, such that each of them is able to autonomously derive RIB entries for all prefixes located within their AS. We define two kinds of IGPs.

Distance-vector IGPs. In distance-vector protocols (*e.g.,* RIP [RFC2453]), nodes maintain a table associating each destination in the network with its relative cost, and the router it learned the route from. Periodically, each node sends a summary of that routing table to all its neighbors. Upon receiving routes on an interface, a node will add the cost of that interface to the cost of the received routes. If the resulting cost of one of the new routes is lower than the one in the node's routing table, then that route is updated to the newly received-one. In these protocols, nodes have no knowledge about the network topology, they only have reachability information.

Link-state IGPs. Link-state protocols (*e.g.,* OSPF, IS-IS [RFC1142]) work in two phases. First, routers discover their neighbors by periodically sending HELLO messages over their links, and wait for other routers to answer. This enables them to differentiate interfaces connected to end-hosts (*i.e.,* stub networks) from those connected to another router. Interfaces connecting routers enable them to establish an IGP adjacency, over which they can exchange reachability information. Beside enabling neighbor discovery, these HELLO messages also provide a primitive link failure detection as receiving them indicates that the link and the router "on the other side" are able to send data, *i.e.,* are up.

Once IGP adjacencies are established, each router can then periodically send Link-State Packets (LSP) to all its neighbors—*i.e.,* packets describing the prefixes associated to its interfaces and the status of its IGP adjacencies. In parallel, routers receiving LSP messages store them in a local database, and then immediately broadcast it to all their neighbors. This repeated LSP broadcast

enables all routers to eventually receive a copy of every router's LSP, and is called a flooding procedure. By analyzing all the LSPs it has received, a router then knows: *(i)* all prefixes located in the network; and *(ii)* the status of every IGP adjacency (*i.e.,* router-to-router links). This enables routers to build a graph describing the current, up-to-date, network topology. Routers then derive routes towards every prefix by executing the Dijkstra algorithm [FT84] to compute shortest-paths towards each of them. Operators can influence the results of this shortest-path computation by assigning a cost on every router interface. For example, Fig. 1.2 shows a network where the cost of the link between router A and B (A, B) has been set to 2, while the cost of (B, C) has been set to 10. Consequently, B has a single shortest-path towards the purple prefix, located at router Y, and transiting through router A. Similarly, A uses the path [A B C D] to reach the blue prefix located at router D.

Both distance-vector and link-state IGPs are inherently distributed, as routers can always derive RIB entries based on some information it exchanged with its peers. As such, IGPs are highly resilient to network failures, and can scale to very large networks. In practice, Link-state IGPs are however preferred to distance-vector ones as they converge much faster [Fra+05] and can easily be used as substrate for overlay protocols (*e.g.,* Segment Routing [RFC8402]).

1.3 Monitoring networks

Monitoring their networks enables operators to both optimize them (*e.g.,* plan forwarding paths according to past traffic demands) and to troubleshoot them (*e.g.,* identify which link is currently dropping packets). As such, monitoring systems are essential to operate networks, and can be divided in two categories.

Active measurements. The first approach to monitor networks consists in actively testing them. To that end, operators generate test traffic (*i.e.,* probe packets) and observe how the network handles it. For example, `ping` is a common program that lets operators test whether they can reach a target host, and also get rough latency estimates. Similarly, `traceroute` enables operators to iteratively discover all hops from a host towards a given destination IP address. Active measurements are often automated, building up complete monitoring frameworks—*e.g.,* IP-SLA [Cisa] checks whether Service Level Agreements (SLAs) are met, Pingmesh [Guo+15] aggregates `ping` measurements to identify potential issues in data-center networks. Operators are not the only ones measuring their networks. Indeed, researchers often use active measurements—*e.g.,* to infer Internet topologies [SMW02], or test whether new protocols can be deployed [Det+13; TDH+])—which are performed from globally deployed probes—*e.g.,* RIPE ATLAS [RIP15], SamKnows [Sun+11].

Passive measurements. Passive network monitoring consists in observing the actual traffic flowing in the network, in order to extract measurements. For example, routers maintain counters on a per-interface basis that describe current link utilization and can be queried using SNMP. Packet sampling techniques (*e.g.,* NetFlow [RFC3954]) go beyond interface-level statistics by dissecting packet headers. More specifically, sampling enables to collect statistics on a per-flow basis[1] such as the amount of bytes/packets, header flags (*e.g.,* TCP SYN), and start and end timestamps of flows. To cope with the high throughput observed in larger networks, this sampling process is often randomized (*e.g.,* sampling on average one packet randomly out of 1024). Random Packet Sampling can be used in ISP networks to estimate traffic demand matrices [Uhl+06], or to identify heavy-hitters [Zha+04].

1.4 Software-Defined Networking

By decoupling the data-plane from the control-plane, Software-Defined Networking (SDN) [Kre+15; FRZ13] is a recent network paradigm shift aiming at increasing network flexibility. More specifically, complex, vertically-integrated, routers are replaced in SDN by simpler programmable switches. Fig. 1.3 shows an high-level overview of such switch. Compared to traditional routers (*i.e.,* Fig. 1.1), SDN switches have no control-plane. Instead, the control-plane is logically centralized in a new network element, the SDN controller. This controller then uses well-defined APIs supported by the switches in order to dynamically program their processing pipeline (*e.g.,* using P4 [Bos+14]), or match-action tables (*e.g.,* using Openflow [McK+08]).

SDN enables a tremendous amount of network flexibility, as operators can program the switch behaviors to exactly fit their particular network needs. For example, as the controller has a global view over the complete network state, it can thus perform a global optimization and implement it using finer-grained constructs than the ones available in traditional networks—*e.g.,* forward traffic based on both the source *and* the destination address. SDN also enables to bypass the (long) standardization process that traditional protocols have to go through, enabling operators to quickly experiment and implement new routing protocols. Major cloud vendors have successfully transitioned to SDN backbones [Hon+13; Jai+13]. Furthermore, SDN supports more use-cases than pure routing. For example, it has been successfully used to build fined-grained network monitoring infrastructures [Kim+15; Zhu+15], coordinate network function [BHH16], cache frequent key-value lookups within switches [Jin+17], or speed-up distributed consensus [Jin+18].

[1] Traditionally identified using their 5-tuple:
(source IP, destination IP, protocol number, source transport port, destination transport port)

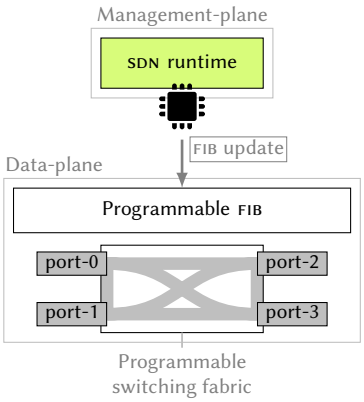


Figure 1.3: High-level architecture of a SDN switch.

Despite all of its promises, SDN has had a limited deployment. Indeed, while SDN improves network manageability and flexibility, it also brings new challenges. Indeed, centralizing the complete control-plane poses scalability challenges (*i.e.,* the SDN is in charge of *everything*) and resiliency challenges (*i.e.,* the controller is a single point of failure).

Part II

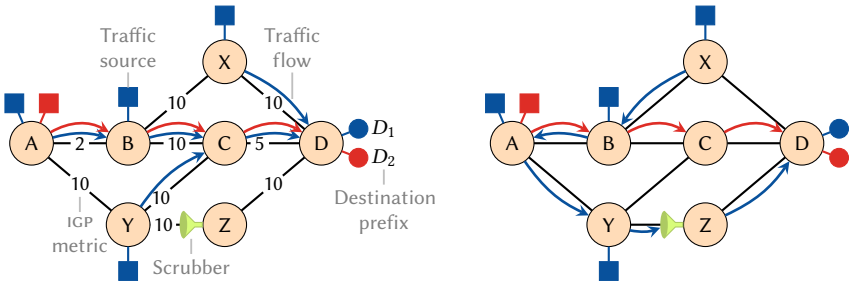
Central Control over Distributed Routing

2 Motivation

Adapting forwarding paths is one of the most daunting task when managing a network. Indeed, while such changes are mandatory to ensure that the network is operating efficiently, carrying these changes is an extremely complex and error-prone task. Multiple studies [Jun08; Ver16] have shown over the years that the vast majority of network outages are due to human errors when deploying new configurations. Networks are thus seldom reconfigured, and operators instead rely on over-provisioning, achieving a trade-off between profitability and the risk of service disruption. This results in inflexible networks, which can at the same time suffer from congestion (*e.g.,* due to flash crowds [Ari+03]) and have most of their capacity unused.

As an example, consider a large network with hundreds of devices, including those shown in Fig. 2.1a. This network provides a best-effort IP transit service, and serves multiple customers through a common router (D), acting as a Point-of-Presence (POP). More specifically, each customer prefix is routed according to the shortest-paths computed by the IGP. One such prefix (D_1) sees a sudden surge of traffic, from multiple ingress routers (A, B, X, and Y), threatening to congest the path used to reach router D. An operator's reaction to this traffic surge is twofold. First, to decrease the risk of congestion, the traffic towards the POP should be load-balanced on unused links, such as (Z, D). Second, to handle an eventual Distributed Denial-of-Service (DDOS) attack, the traffic flows causing the surge must be directed through a scrubber. This scrubber will analyze those flows, and drop those considered to be malicious.

Performing these two tasks is challenging in traditional networks. The first, adding new paths for load-balancing purposes, requires to carefully change the configuration of multiple routers. One possibility is to route traffic over both [A B C D] and [A Y Z D], which with IP forwarding requires their path cost to be equal. This requires to change the configuration of routers A and Y such that (A, Y)=3 and (Y, Z)=4. Note that a hidden constraint when performing this change is to pick metrics that will disallow the use of the link (Y, C) as it would create uneven load-balancing. The second task is more difficult. As the scrubber is not adjacent to router D, ensuring that the packets are forwarded via Y



(a) Initial network state.

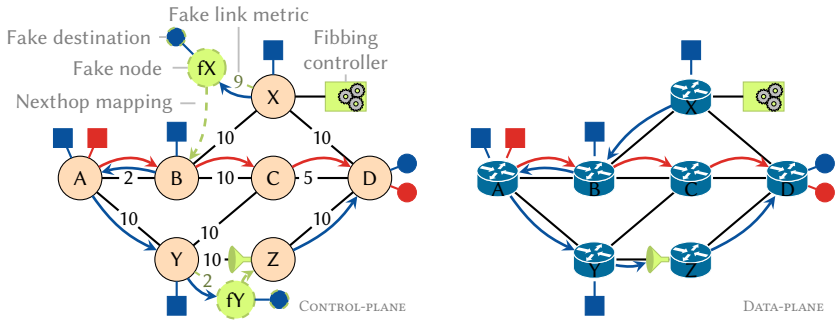
(b) Desired forwarding paths.

Figure 2.1: Steering all traffic towards destination address D_1 through the scrubber requires to change the FIB of every router.

requires to change the forwarding paths for all flows towards D_1 such that the scrubber is always reached before router D. Worse, because both D_1 and D_2 share the same egress router and thus the same set of shortest-paths, it is *impossible* to change the forwarding paths towards D_1 without also re-routing D_2. Therefore, achieving the second task would steer both prefixes through the scrubber, preventing any form of load-balancing.

Instead of relying on the IGP, additional protocols and techniques are needed to implement an acceptable set of forwarding paths, such as those visible on Fig. 2.1b. Tunneling protocols (*e.g.,* using a MPLS [RFC3031] data-plane, with a control-plane running LDP [RFC5036], RSVP-TE [RFC3209] or Segment Routing (SR) [RFC8402]) could be used to encapsulate and forward all flows towards D_1, implementing the forwarding paths shown on Fig. 2.1b. Unfortunately, as D_1 has four ingress routers, four tunnels would need to be configured. Depending on the tunneling mechanism, achieving the required paths would also require to add state on downstream routers (*e.g.,* RSVP-TE) and/or to encode such path in every forwarded packet (*e.g.,* Segment Routing). This increases the control-plane and/or the data-plane overhead.

Software Defined Networking (SDN) could easily solve the problem, as it provides a centralized and direct control over the FIB of every network device (*e.g.,* OpenFlow [McK+08], PCE [RFC4655]), thus to program all FIB entries needed to achieve Fig. 2.1b. However, SDN brings challenges of its own, as it trades away the scalability and resiliency aspect of distributed routing protocols. Indeed, a SDN controller has to compute and install all FIB entries for every flow, on every switch, and react quickly to topology changes. The controller is thus a bottleneck, due to both the size of the state to manage and the throughput at which it can update routes. In contrast, distributed routing protocols naturally split the network state across routers, and parallelize route computations. Distributing the controller across several nodes [Dix+13;



(a) A Fibbing controller augments the IGP topology with fake links and nodes.

(b) Routers map fake nodes to specific nexthops when computing FIB entries.

Figure 2.2: A Fibbing controller programs forwarding paths by injecting fake elements with specific metrics and nexthop mapping in the IGP topology.

Ber+14] to improve its reliability and scalability leads to additional challenges as it becomes a distributed system of its own, managing a shared state. Finally, the deployment of SDN as a whole is major hurdle due to the huge installed base of devices and management tools that need to be upgraded, as well as the lack of expertise with the technology of human operators. As a result, existing SDN deployments are often limited in scope, *e.g.,* for new deployments of private backbones [Jai+13; Hon+13], software deployments at the network edge [Cas+12], or restricted to data-centers [Tav+09; Sin+15].

This part presents Fibbing, a technique that provides a direct control over the routers' FIB by manipulating the input of distributed routing protocols. Indeed, the content of routers' FIB in a network using a link-state IGP (*e.g.,* OSPF [RFC2328], IS-IS [RFC1142]) is computed autonomously by each router as it computes shortest-paths over a synchronized view of the network. Leveraging this, a Fibbing controller introduces fake nodes (with fake prefix reachability information) and fake links (with specific metrics) to modify the shared view of the topology. This coaxes routers to compute desired shortest-paths without reconfiguring them, *i.e.,* a Fibbing controller *lies* to routers to control their FIB. In essence, Fibbing inverts the routing function: given the forwarding entries (*i.e.,* the output) and the routing protocol (*i.e.,* the function), Fibbing computes the corresponding routing messages to send to the routers (*i.e.,* the input).

A Fibbing controller can solve the problem in Fig. 2.1 by injecting two fake nodes (Fig. 2.2a), connected to routers X and Y with the depicted link metrics. Both fake nodes advertize reachability information exclusively towards D_1. When X recomputes its shortest-path towards D_1, it now prefers to use the one going through fX as it is cheaper (9) than the one going through D (10).

	Centralized/SDN *OpenFlow, PCE, SR*	Distributed/traditional *IGP, LDP*	**Hybrid** ***Fibbing***
Forwarding paths:			
configuration	▲(declarative & global)	▼(indirect & per-device)	▲(declarative & global)
manageability	▲(direct control)	▼(need for coordination)	▲(direct control)
path installation	▼(by controller, per-device)	▲(by device, distributed)	▲(by device, distributed)
Robustness:			
network failures	▼(by controller)	▲(local)	▲(local)
controller failures	▼(ad-hoc sync)	▲▲(distributed)	▲(sync via IGP)
partitions	▼(uncontrollable devices)	▲(distributed)	▲(fallback on IGP)
Routing policies:	▲▲(any path)	▼(shortest paths)	▲(any loop-free paths)

▲better ▼worse

Table 2.1: Fibbing combines the advantages of existing control planes and avoids their main drawbacks

Similarly, B, A, and Y, select the shortest-path going through fY. Routers can then update their FIB to the newly selected nexthops. As router X (resp. Y) uses a fake node as nexthop, it resolves it to the real nexthop B (resp. Z) using mapping information that was injected along the fake node. The resulting data-plane paths are then those shown in Fig. 2.2b. As no fake nodes announce a route towards D_2, the forwarding paths used to reach D_2 are unchanged.

Table 2.1 positions Fibbing against the two main networking paradigms. Namely, Fibbing adopts a hybrid approach, improving the flexibility and manageability of traditional IGPs by adopting a SDN-like approach, while preserving their inherent scalability and resiliency. More precisely, this hybrid approach gives Fibbing the following advantages.

Fibbing is expressive. A Fibbing controller can program the IGP to steer traffic along *any* set of loop-free paths, on a per-destination basis (*i.e.,* for any IP prefix length, up to a single address). This enables Fibbing to support advanced forwarding applications such as traffic engineering, load balancing, fast fail-over, and traffic steering through middleboxes. As it relies on destination-based routing protocols, Fibbing does not support finer-grained routing and forwarding policies such as matching on port numbers. However, those policies can easily be supported by integrating Fibbing in "flexible IGPs" [Pse+18] or by using middleboxes.

Fibbing scales and is robust to failures. Fibbing offloads most of the failure-recovery work to the IGP itself. Additionally, Fibbing can quickly compute augmented topologies to avoid loops and blackholes upon network failures. At the same time, Fibbing minimizes the size of these augmented topology to account for the limited CPU and memory resources of IP routers. Finally, Fibbing gracefully handles controller failures, by easily replicating controller instances and supporting both fail-open and fail-close semantics.

Fibbing keeps the path installation distributed. Prior approaches like the Routing Control Platform [Cae+05] rely on BGP as a "poor man's" SDN protocol enabling a controller to directly push FIB entries on each router. In contrast, Fibbing leverages the routing-protocol specifications to push modification on the view of the network shared by all routers. Doing so, Fibbing can trigger multiple forwarding changes at once, while letting routers compute on their own the content of their FIB. In other words, while the controller computes the routing *input* centrally, the routing *output* is still computed in a distributed fashion.

Fibbing works on existing routers. We present a Fibbing prototype, able to program real, unmodified, routers (we tested it with Cisco and Juniper routers). Our measurements show that these routers can install hundreds of thousands of FIB entries with an average installation time of less than 1ms per entry. This enables Fibbing to operate at much greater scale and with a faster convergence than what is achievable with recent SDN switches [Jin+14; Rot+12], without requiring new hardware. Furthermore, Fibbing enables to incrementally deploy in existing network SDN features which were typically restricted to clean-slate deployment in private backbones (*e.g.,* Google's B4 [Jai+13], Microsoft's SWAN [Hon+13]).

The rest of this part is divided in three chapters, structured as follows.

A first chapter presents the theory behind Fibbing (§3). Doing so, it highlights the abstraction enabled by the approach, formulates novel and provably correct algorithms to efficiently augmented topologies, and finally positions Fibbing in a broader context, *i.e.,* by showing its interactions with other protocols, discussing its applicability, and comparing it against prior art.

Then, a second chapter presents an implementation of Fibbing in a real IGP, namely OSPF (§4). More specifically, it begins by mapping core Fibbing concepts (*e.g.,* fake nodes and fake path costs) to OSPF messages and attributes, and then presents the architecture of a prototype Fibbing controller compatible with real unmodified routers. After that, the chapter demonstrates the applicability of Fibbing through measurements showing its low overhead on routers, a discussion of its failure-handling mechanism, and a case study where Fibbing performs real-time traffic engineering.

Finally, a summary (§5) concludes this part and discusses future research directions.

Flexible intra-domain routing with Fibbing

3

This chapter presents the theory behind Fibbing. To that end, we begin by describing the *abstraction* it enables, by showing how a network operator can express and realize high-level forwarding requirements (§3.1). Next, we formalize the *Topology Augmentation Problem*, and propose new algorithms to compute compact augmented topologies (§3.2). Among those algorithms, we discuss in detail the implementation of an efficient greedy heuristic which minimizes the size of the augmented topologies, *the Merger algorithm*, and prove its correctness (§3.3). To assess the practicality of the approach, we show that a Fibbing controller can quickly generate small augmented topologies through *benchmarks* conducted on realistic ISP topologies (§3.4). Then, we study the *interactions* between Fibbing and other protocols that use information from the IGP (§3.5), such as its influence on the BGP decision process (§3.5.2). After that, we *discuss* practical considerations about the approach (§3.6), such as its ability to enable incremental SDN deployment in legacy networks. Finally, we position Fibbing in a broader context by comparing it to related works (§3.7).

3.1 An abstraction to control paths in link-state IGPs

To control the forwarding paths computed by the IGP, Fibbing follows four consecutive stages, illustrated in Fig. 3.1, based on two inputs: *(i)* the network topology, which is automatically obtained from the IGP; and *(ii)* the desired forwarding requirements. More specifically, these forwarding requirements are either expressed directly as per-destination Directed Acyclic Graphs (DAGs), or indirectly using a high-level declarative language. In the latter case, the *Compilation* (§3.1.1) stage translates these high-level requirements into concrete forwarding DAGs. Then, the *Augmentation* (§3.2) stage solves the Topology Augmentation Problem, and computes an augmented topology that satisfies these forwarding DAGs in milliseconds. An *Optimization* (§3.3) stage can

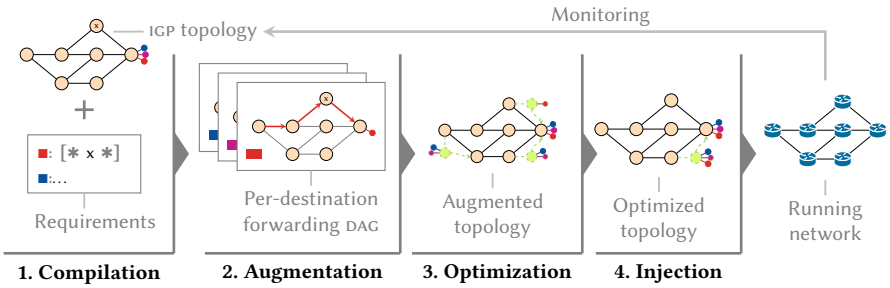


Figure 3.1: A Fibbing controller follows four steps to program paths in an IGP.

then reduce the size of the augmented topology while preserving the forwarding paths, minimizing the control-plane overhead at the expense of a slightly longer computation (hundreds of milliseconds §3.4). Finally, fake nodes and links contained in the augmented topology are transformed to actual IGP messages in the *Injection* (§4) stage. These messages are then flooded by the IGP, causing all routers to update their FIB to implement the operator requirements.

In this section, we first present the high-level language to express requirements and its compilation process (§3.1.1). Then, we show that Fibbing is expressive, enabling to program a wide-range of forwarding behaviors (§3.1.2).

3.1.1 A declarative language to express requirements

In Fibbing, operators configure a logically centralized controller in order to define network-wide requirements on the forwarding paths to be used. As These requirements can be expressed using the high-level language shown in Fig. 3.2. This language enables operators to follow a declarative approach when configuring their network, *i.e.,* to specify *what* their requirements are, simplifying their management. This process is in stark contrasts with the descriptive approach used in network running traditional distributed protocols, *i.e.,* specifying *how* to implement the requirements, where each device has to be configured individually.

Fibbing requirements are a collection of *per-destination requirements*. That is, Fibbing enables to specify requirements (USE) for any IP prefix, up to a single IP address (TOWARDS <*prefix*>), regardless of whether it is advertised in the IGP or not. These requirements fall in two categories: *(i)* forwarding requirements describe the paths along which the traffic should be forwarded to reach the destination; and *(ii) backup requirements* (identified by the ASBACKUPOF() operator) describe forwarding requirements that should be used when an IGP topology change causes one of the listed links to fail. In

req	::=	$d_1; \ldots; d_n$	Fibbing requirements
d	::=	USE (*fwd* | *backup*) TOWARDS <*prefix*>	Per-destination requirement
fwd	::=	$(fwd_i^{m_i}$ AND $fwd_j^{m_j})$ | $(fwd_i$ OR $fwd_j)$ | *p*	Forwarding requirement
p	::=	$[n^+]$	Path expression
n	::=	<*id*> | ∗	Node expression
backup	::=	*fwd* asBackupOf($link^+$)	Backup requirement
link	::=	$[$<*id*>$_i$, <*id*>$_j]$	Link expression
m	::=	<*integer*> (defaults to 1)	Multiplicity attribute

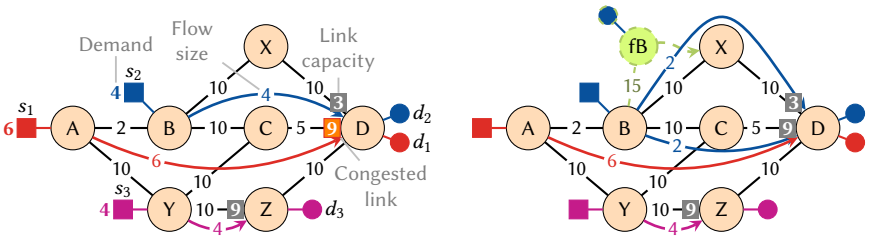
Figure 3.2: Abstract syntax of the Fibbing requirement language.

turn, each *forwarding requirement* is either recursively defined as a conjunction (AND) or as a disjunction (OR) of forwarding requirements, or describes a single path. A conjunction of n paths indicates that the traffic should be load-balanced across all n paths, optionally using the *multiplicity* attribute to control splitting ratios. More precisely, when load-balancing traffic across the set of paths and multiplicity attributes $p_1^{m_1}, \ldots, p_n^{m_n}$, a path $p_i^{m_i}$ will receive a fraction of the traffic equal to $\frac{m_i}{\sum_{k=1}^{n} m_k}$. A disjunction of n paths indicates that the traffic should be forwarded along one of those n paths (for example, to ensure the traversal of one of the replicas of a middlebox). Finally, a concrete *path* is a sequence of node identifiers (*e.g.,* IGP router id, loopback address). For flexibility, paths can be loosely defined using wildcards (∗) nodes, representing any sub-sequence of nodes.

To minimize the number of requirements that should be specified, Fibbing follows an exception-based approach. Nodes that are not part of a forwarding requirement have to keep their original set of nexthops, as computed by the IGP without a Fibbing controller.

Compilation. Translating requirements to per-destination forwarding DAGs is a three-step process.

First, the compiler ensures that all forwarding requirements only contain concrete paths by expanding the eventual wildcard nodes. A wildcard node expansion requires to replace the wildcard by one of the simple paths between its surrounding nodes, or any node in the network if the wildcard denotes a path tip. Each expansion then defines a concrete path, and the disjunction of all these concrete paths then replaces the original loosely defined path. This step can be computationally expensive as, in general, a network can have a number of paths that is exponential in the number of nodes. While unlikely in practice, especially for networks designed according to the current best practices, we bounded the number of paths that can be expanded out of a single requirement (10 by default). If no solution is found with the current set of paths during the topology augmentation, then those paths are expanded further.



(a) Initial topology (b) Augmented topology

Figure 3.3: Fibbing let operators program on-demand load-balancing.

Second, once all requirements are well-defined, the compiler groups them by destination and computes the Disjunctive Normal Form (DNF) of each requirement.

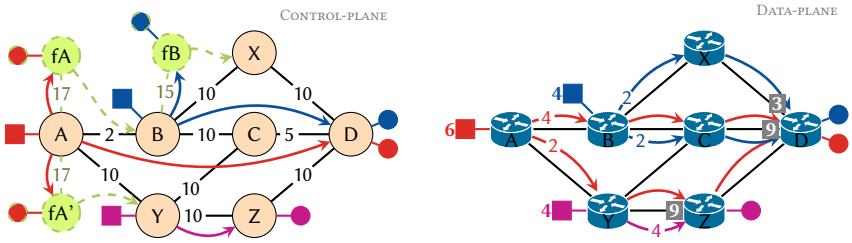
Finally, the compiler iterates over the resulting disjunction of path requirements, and builds a tentative forwarding graph for each clause. If the resulting graph is loop-free (*i.e.*, it is a DAG), the compiler then checks that every node present in the IGP topology is also present in the DAG. If a node is missing from a DAG specifying primary forwarding requirements, it is added in the DAG with its nexthops set to those from the unmodified IGP shortest-paths. Forwarding DAGs corresponding to backup requirements are also tagged with the set of links they protect.

3.1.2 Fibbing is expressive

Fibbing enables to steer any flow along a given path (§2). More precisely, Fibbing is able to enforce any forwarding DAG on a per-destination basis [VVR14]. We now highlight its expressive power through three high-level use cases.

Fibbing enables to program on-demand load-balancing.
Load-balancing traffic over multiple paths is a key primitive when designing networks, for example to maximize throughput, minimize response time, or increase reliability. Fibbing enables operators to dynamically add or remove paths on a per destination basis to configure ECMP-based load-balancing, and to control the resulting splitting ratios. For example, consider the network depicted in Fig. 3.3a, where three sources s_1, s_2, and s_3 send traffic to three corresponding destinations d_1, d_2, and d_3. Demands and link capacities are such that link (C, D) is congested. To alleviate congestion, load-balancing should be induced, for example by splitting on B traffic towards d_2 over two paths: via [B X D] and via [B C D].

This can be achieved in traditional IGPs by inducing additional equal-cost paths (*e.g.*, by re-weighting (B, X) to 5). However, as this would affect equally



(a) Fibbing induces two additional equal-cost paths from A towards d_1.

(b) A splits traffic unevenly as two shortest-paths out of three use the same nexthop.

Figure 3.4: Fibbing controls fractional traffic splitting ratios when load-balancing traffic, optimizing link utilization.

the traffic towards d_1 and d_2, this would congest the upper path. More generally, it is impossible to route the traffic destined to D_1 and D_2 using different set of paths purely in conventional IGP.

This simple requirement can easily be expressed with Fibbing as:

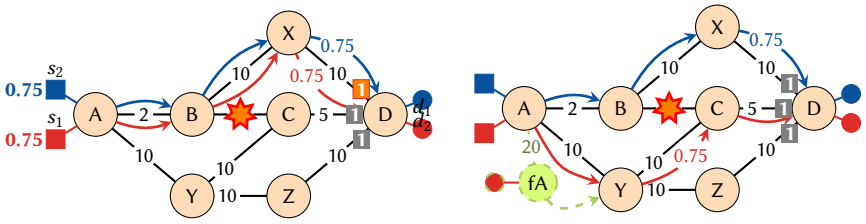
USE ([B X D] AND [B C D]) TOWARDS d_2

Fig. 3.3b shows an augmented topology which achieves this requirement. A fake node announcing d_2 is attached to B (with a metric of 5 on the fake link), and is mapped to X. After introducing this node, B has two shortest paths (of cost 15) to reach d_2, and splits its d_2 traffic (equally) over X and C.

Fibbing controls ECMP splitting ratios. While effectively removing the congestion, the augmented topology shown on Fig. 3.3b results in a link utilization imbalance as [B C D] is used to nearly its full capacity. Averaging out the link utilization ratios would reduce the risks of future congestion, should the demands change. Achieving this requires to send a third of the traffic destined to d_1 over the bottom path [A Y Z D]. Fibbing enables to program the ECMP splitting ratios through the *multiplicity* attribute that can be set on paths, leading to the following additional requirement:

USE ([A B X D]^2 AND [A Y Z D]) TOWARDS d_1

Fig. 3.4a shows an augmented topology achieving both load-balancing requirements. We see that the multiplicity attribute resulted in the addition of a redundant fake node fA mapping to B, as it has the same cost as the path [A B C D] and maps to the same nexthop. Combined with fA', this now causes A to have 3 equal-cost shortest-paths in the control-plane, and thus 3 FIB entries. As two of these shortest-paths use the same nexthop (B), we see on Fig. 3.4b that twice as much traffic towards d_1 goes over (A, B) than over (A, Y). Combined with the fake node for d_2 on B, this minimizes the maximum link load in the network.



(a) Initial topology (b) Augmented topology

Figure 3.5: Fibbing provisions backup paths to avoid post-failure congestion.

Fibbing can provision backup paths. While IGPs automatically rebuild new sets of shortest-paths after a link or node failure, these new paths might not be efficient and could have unwanted side effects (*e.g.,* causing congestion, increasing delays). Fibbing enables to precisely control the paths that will be used post-failures. More precisely, Fibbing can provision backup paths to protect specific traffic flows. As an illustration, consider the network in Figure 3.5a. The failure of a link of the sub-path [B C D] would lead to congestion since traffic flows towards d_1 and d_2 are both rerouted using the same alternate path ([B X D]), which includes a low capacity link (X, D). To prevent congestion, traffic towards d_1 and d_2 should use disjoint post-failure paths, yet only use those when the primary path has failed. This is impossible to achieve in conventional IGPs as both traffic flows share the same source and destination nodes, and would require significant control-plane overhead in MPLS. In contrast, Fibbing can achieve this easily using the following requirement:

USE [A B *] ASBACKUPOF([B, C] [C, D]) TOWARDS d_1
USE [A Y *] ASBACKUPOF([B, C] [C, D]) TOWARDS d_2

Fig. 3.5b shows the corresponding augmented topology, which has a single fake node advertising d_2. The metric to reach the fake node is such that: *(i)* it is more expensive to use the fake node than to use the default IGP shortest-path; and *(ii)* it will be the new shortest-path towards d_2 if any of the links of [B C D] fails. Unfortunately, while successful for this example, Fibbing cannot satisfy all possible backup paths requirements (§3.2.5).

3.2 Augmenting topologies

In this section, we formulate the augmentation problem (§3.2.1), and we show how the Fibbing controller quickly computes augmented topologies from a set of forwarding DAGs. We rely on a *divide-and-conquer* approach based on three consecutive steps.

1. **Topology initialization (§3.2.2).** We modify if necessary the initial

IGP metrics to guarantee that a Fibbing controller can always enforce any set of forwarding DAGs. This operation only needs to be done once, when Fibbing is first deployed in the network.

2. Per-destination augmentation (§3.2.3). We solve the Topology Augmentation Problem for each per-destination forwarding DAG on the initialized topology. We designed two algorithms for this step, achieving different tradeoffs between computation time and augmentation size. The fastest one, Simple, computes augmented topologies *within milliseconds*. Simple injects one dedicated fake node for every router that changes its nexthop. The relatively slower algorithm, Merger, aggressively reduces the size of the augmented topology by re-using the same fake nodes to program multiple routers. While the speed of Simple makes it well-suited to react quickly to transient events, such as failures, Merger is better suited to be run in background, progressively optimizing the augmented topology.

3. Optimization across destinations (§3.2.4). We further reduce the number of fake nodes and edges by merging the per-destination augmentations. Namely, whenever safe, we replace multiple fake nodes announcing different destinations with a single fake node which either announces all the destinations or creates a shortcut between routers in the augmented topology.

Finally, we discuss how backup requirements can be implemented using our augmentation algorithms (§3.2.5).

3.2.1 The Topology Augmentation Problem

We begin by defining three concepts that are key to formulate the problem we want to address. They will serve as basis when designing the algorithms solving it.

Nexthop selection process. We assume that routers select their nexthop set for a given prefix according to the decision process presented in (§1.1), *i.e.,* local routes are preferred to those computed by the IGP. For example, as D in Fig. 3.5 has local routes for d_1 and d_2 (*e.g.,* learned from another routing protocol, or directly connected routes), it will *never* prefer to use a route announced by a fake node, regardless of the announcement cost. In contrast, as A has no such route, it will select the IGP shortest-path towards each prefix, thus possibly decide to use the route announced by a fake node if it is shorter.

Fake edges nexthop mapping information. Fibbing injects fake nodes, attached to real nodes using fake edges, in the IGP. Alongside these fake elements, Fibbing also injects mapping information, linking each fake edge to a real forwarding nexthop. Throughout this part, we assume that a fake edge in the shortest path from any router R to any destination d corresponds to the

ability of R to resolve the fake edge to be any of its neighbors using this mapping information. In the introductory example in Fig. 2.2a, for instance, the fake edge between x and its adjacent fake node fx translates into x forwarding traffic to B.

Fake nodes scoping. A Fibbing controller can generate two types of fake nodes. First, *locally scoped* fake nodes are targeted to a single router. More precisely, they are ignored by all routers but the target one when they compute their shortest-paths. This enables to control the FIB of a target router without side effects for the other ones. Second, *globally scoped* fake nodes target all routers in the network. As such, all routers take them into account when computing their paths. Hence, if carefully computed, they can reduce the size of the augmented topology as they can affect more than one router. All of our previous examples used globally scoped fake nodes.

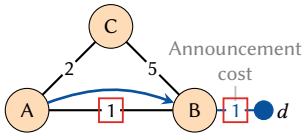
Problem 1 (Topology Augmentation Problem)**.** *Given an initial topology G and a set of forwarding DAGs, compute an augmented topology $G' \supset G$ such the following two properties hold:* (i) *for each path $[u\ v \dots\ d]$ in the forwarding DAG towards d, the nexthop of u in one of its shortest paths for d in G' is either v or a fake node f_u such that the fake edge linking u to f_u maps to v; and* (ii) *the number of shortest-paths towards d for any node R in G' is equal to the sum of the number of paths $[u \dots\ d]$ times their multiplicity attributes (§3.1.1) across all forwarding DAGs towards d.*

3.2.2 Topology Initialization

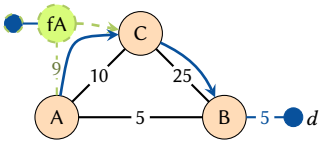
The goals of the topology initialization procedure are twofolds. First, it guarantees that Fibbing can always find an augmented topology implementing the requirements of any arbitrary forwarding per-destination DAG. Second, it increases the likelihood that the resulting augmented topologies can be efficiently reduced through the optimization algorithms (Merger (§3.3) or the cross-destination algorithms (§3.2.4)). To that end, we proportionally increase all link metrics (multiplying them by a constant factor) if any of them is too low. We also set high announcement cost for any destination (*e.g.,* the number of hops in the longest path in the network topology times the maximum link weight). More generally, our initialization procedure ensures the following property for the topology:

Property 1 (Fibbing compliance)**.** *A topology is Fibbing compliant if, for every destination d, the total cost of the shortest path from every router (including the ones announcing d) to d exceeds* 2*.*

This property ensures that for any router R and destination d, the controller can always compute a fake path P with two properties: *(i) P* is shorter



(a) A non-Fibbing compliant topology prevents A's preferred shortest-path from being changed using fake nodes or links.



(b) Multiplying all weights by 5 is sufficient to ensure Fibbing compliance, enabling to re-route A's traffic using Fibbing.

Figure 3.6: Every topology can be made Fibbing compliant.

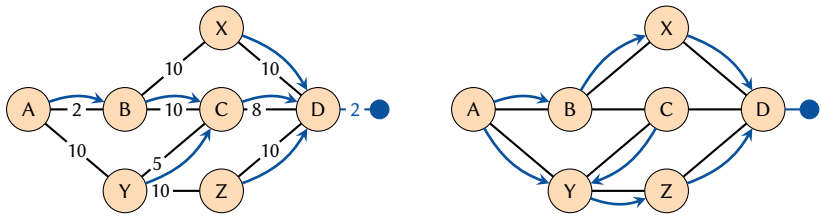
than the original shortest-path from R to d; and *(ii)* P is longer than the original shortest-path from any other router N ≠ R to d.

Topology initialization is non-intrusive. The initialization procedure only affects two sets of parameters (link metrics and prefix announcement costs), present in any link-state routing configuration. Additionally, multiplying them by a constant factors *preserves the original forwarding paths*. This re-weighting process can be carried out using known lossless reconfiguration techniques [Van+11]. This procedure is strictly required only once in the network lifetime. Indeed, Fibbing compliance is independent of the status of the links, or the forwarding requirements. Topologies growing in size can be easily kept Fibbing compliant by ensuring that the new destinations are announced with large enough costs and that the new links have weights consistent with those already made Fibbing compliant.

Fibbing compliance enables full Fibbing expressivity. Consider the IGP topology visible in Fig. 3.6a. As it is, this topology is not Fibbing compliant as there is one shortest-path that costs 2 ([A B d]). Changing A's nexthop would then require to add a fake node fA advertizing d. The new fake path [A fA D] would then need to have a lower cost than the current shortest-path (2). The cost of a path is computed as the sum of the traversed link metrics, as well as the announcement cost of the destination. As routing protocols typically require that these values are set to strictly positive integer, 2 is thus the minimal cost of a path. It is thus impossible to change A's nexthop using Fibbing in this topology.

Fig. 3.6b shows a possible output of our initialization procedure applied to the topology in Fig. 3.6a, where all metrics have been multiplied by 5. The topology is thus Fibbing compliant, and any per-destination forwarding DAG can be implemented with Fibbing in Fig. 3.6b. For example, re-routing A's traffic through C implies to inject a fake node such that the resulting fake path cost is less than 10.

We now prove with the following theorem that Fibbing compliance is a sufficient condition to implement any arbitrary set of Fibbing requirements.



(a) Shortest-paths computed by the IGP. (b) Desired new forwarding paths.

Figure 3.7: We compare the results of the per-destination algorithms on forwarding requirements causing four nodes to change their nexthops.

Theorem 1 (Fibbing expressiveness)**.** *Any set of per-destination forwarding DAGs can always be enforced by augmenting a Fibbing-compliant topology regardless of the fake node scoping type being used.*

Proof. We prove theorem 1 by describing a simple topology augmentation procedure that can place indiscriminately globally or locally visible fake nodes.

Let G be the initial topology. For every forwarding DAG with destination d, let $h_{R1}, \ldots, h_{Rn}$ be the set of nexthops of each node R in the DAG. For each nexthop h_{Ri} of R, we attach to R one fake node f_{Ri} announcing d. Each of these fake nodes generates a new fake path [R f_{Ri} D] in the augmented topology. We set the total cost of those newly added fake paths to 2. Since G is Fibbing compliant, then the cost of any other path from R to d in G whose nexthop is not a fake node is greater than 2. Hence, the shortest-paths of every node R in the augmented topology will be the set of paths $\{\forall i \in [1, n] \mid [R\ f_{Ri}\ D]\}$. The forwarding DAG is then implemented by mapping all fake links to their right physical links. □

Note that Theorem 1 applies to destinations in the augmented topology and does not necessarily match the destination prefixes announced in the original IGP. It thus implies that Fibbing controls the path for any IP prefix, including those resulting from the (de-)aggregation of existing prefixes (from a /0 to a /32 or /128) and prefixes not originally reachable in the IGP.

3.2.3 Per-destination augmentation

Two algorithms can be used to solve the Topology Augmentation Problem, Simple and Merger. As both can implement any set of requirements (See Theorem 1), we illustrate their operations and key differences on the same set of forwarding requirements, visible in Fig. 3.7

Algorithm 1 The Simple topology augmentation algorithm.

let G: IGP topology
let d: destination prefix
let D: forwarding DAG towards d

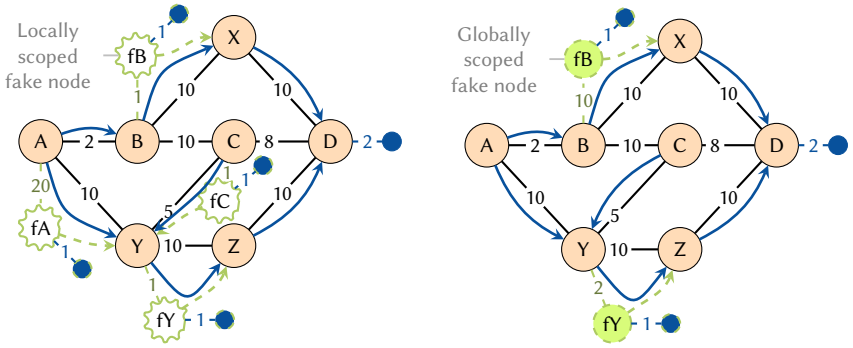
1: $F \leftarrow \emptyset$ — The fake paths to inject
2: **for all** R $\in G$ **do**
3: $N_D \leftarrow \{nh \in D \mid \exists\ (\text{R}, nh) \in D\}$ — Compute the nexthops of R in D
4: $N_G \leftarrow \{nh \in G \mid \exists\ (\text{R}, nh) \in G\}$
5: $N \leftarrow \emptyset$ — The nexthops for the fake paths of R
6: $cost \leftarrow 2$ — The total cost of the fake paths
7: **if** $N_G \subset N_D$ **then** — We add equal-cost paths
8: $cost \leftarrow dist(\text{R}, d, G)$ — Compute the shortest-path cost from R to d in G
9: $N \leftarrow N_D \setminus N_G$
10: **else if** $N_G \neq N_D$ **then** — We define new paths
11: $N \leftarrow N_D$
12: **for all** $nh \in N$ **do** — Add per-nexthop local fake paths towards d
13: **for** $i = 1, \ldots, multiplicity(D, \text{R}, nh)$ **do**
14: $F \leftarrow F \cup \{\ (\text{'local'}, \text{R}, nh, d, cost)\ \}$ — Enforce the multiplicity attribute
15: **return** F

Simple

The Simple algorithm relies solely on locally scoped fake nodes. For every destination d and corresponding forwarding DAG D, Simple adds fake nodes to each router whose nexthops in the original IGP topology G differ from those in D. More precisely, as shown on Alg. 1, Simple works on a per router basis.

For every router present in the IGP topology, the algorithm compares the desired set of nexthops in D, to the one computed by the IGP. If the two sets are equal, then the algorithm moves to the next router in G. Otherwise, Simple has to create locally scoped fake paths to implement the requirements. If the requirements add new shortest-paths to be used in conjunction with the ones computed by the IGP (line 7) then Simple only adds fake paths for the new nexthops (line 9), with a total cost set to the original shortest-path cost. In the more general case, the algorithm adds one fake path towards d per desired nexthop (line 11) with a total cost of 2. As the topology is Fibbing compliant, this ensures that R will recompute its nexthops to be the desired set. Finally, the algorithm (line 13) ensures that the eventual multiplicity attributes (see §3.1.1) in D are met by replicating fake paths accordingly.

Executing Simple to implement Fig. 3.7b leads to the augmented topology visible in Fig. 3.8a. Routers A, B, C, and Y are required to change their respective



(a) Simple quickly injects 4 locally scoped fake nodes: 3 changing their router's nexthop, and one adding a new equal cost path.

(b) Merger reduces the size of the augmented topology using two globally scoped fake nodes, with a longer computation.

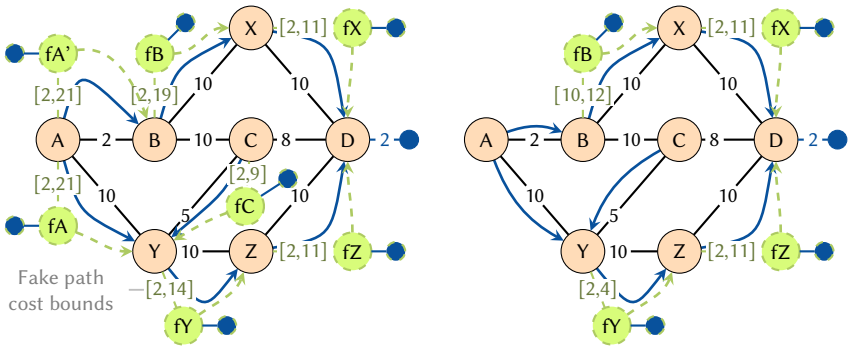
Figure 3.8: Simple and Merger perform different trade-offs in execution time and augmentation size to satisfy the forwarding requirements from Fig. 3.7b

nexthop set. Moreover, A needs to load-balance traffic over one additional nexthop (Y). Simple thus injects four locally scoped fake nodes. Three of these fake nodes change the preferred shortest-path of routers C, C and Y, as the new fake paths have a total cost of 2. The fourth fake node adds a new equal-cost path on router A, hence has the same total fake cost as the existing IGP path (21). Note that as the fake nodes are locally scoped, each of them only affects the routers on which they are attached. Consequently, none of these fake nodes can be removed, as doing so would cause routers to revert to the default IGP nexthops. For example, removing fC would cause C to fallback to using D as nexthop as it would exclude fY from its shortest-paths computation (*i.e.,* C cannot use the locally scoped fake node of Y).

Merger

The Merger algorithm aims to implement forwarding requirements with a lower control-plane overhead than Simple. Merger reduces the size of the augmented topology by relying on globally scoped fake nodes, where a single fake node can change the nexthops of multiple routers. In contrast to the four fake nodes created by Simple to implement the requirements of Fig. 3.7b, Merger only needs two (see Fig. 3.8b). The first one (fY) to change the nexthops of A, Y, and C. And the second one (fB) to change the nexthop of B. Additionally, the fake path costs are set such that A sees two equal-cost paths (with a total cost of 12), load-balancing its traffic over B and Y.

We now intuitively present the generic four steps followed by Merger to implement a forwarding DAG D towards a destination d in a Fibbing compliant



(a) Initial bounds and fake nodes.

(b) New bounds after 3 merges.

Figure 3.9: Merger iteratively merges successive fake nodes.

topology G. Additional details and correctness proofs are reported in Sec. 3.3.

Step 1—Initial fake node placement. Merger places an excessive amount of globally scoped fake nodes in the network. For illustrative purposes, a strawman approach (see §3.3) is to add one fake node for each nexthop of every non-terminal node in D (see Fig. 3.9a). For each of these fake nodes, Merger then computes *acceptable bounds* for the resulting fake path costs.

We define, the *upper bound* UB(fR) of a fake node fR as the maximal total fake path cost such that R will compute [R fR d] as (one of) its shortest-path(s). Initially, Merger sets UB(fR) = $dist$(R, d, G) − 1, *e.g.*, UB(Y)=14. Intuitively, this upper bound carries the constraint that a fake node has to be used. In contrast, the *lower bound* LB(fR) of a fake node fR carries the constraint that a fake node can only be used by a specific set of nodes. More precisely, LB(fR) is the minimal fake path cost such only R and its ancestors in D without a fake node can compute a shortest-path going through fR. In this example, as every node in the network has a fake node, Merger sets the initial value of any LB(fR) to 2. We say that a fake path cost is acceptable whenever LB(fR) ≤ UB(fR).

As G is Fibbing compliant, we can already compute an (inefficient) initial augmentation by setting the cost of all fake paths [R fR d] to LB(fR) (*i.e.,* 2).

Step 2—Fake node merging. Merger then combines successive fake nodes together. More specifically, the algorithm iteratively tries to merge consecutive fake nodes on any simple path in D. To determine whether any two fake nodes can be merged, the algorithm checks three conditions, formalized in (§3.3.4). To illustrate them, we now consider the merging attempt of fA and fY.

First, Merger ensures that if A loses its fake node and computes a shortest-path going through fY, the traffic will only flow along the desired paths. To that end, it verifies that all the shortest-paths in G' from A to Y are also in-

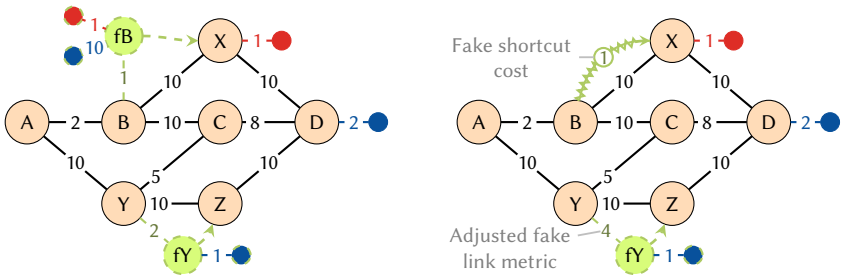
cluded in D.

The second condition ensures that fY can absorb the constraints of fA. More precisely, Merger computes new provisional bounds [LB'(fY), UB'(fY)] for fY that would be the result of merging fA into fY. Intuitively, these new bounds must ensure that fY is suitable for both Y and A. For example, UB'(fY) must guarantee that $dist(A, Y, G) + max\{dist(Y, fY, G)\} < dist(A, d, G)$, thus that A will prefer to use the path going through fY over its original IGP path. Similarly, the lower bound must be such that $LB(fA) \leq LB'(fY) + dist(A, Y, G)$, *i.e.,* the new lower bound of fY *"as seen from A"* does not attract unwanted nodes. The condition is then verified if those provisional bounds are acceptable, *i.e.,* if $LB' \leq UB'$, which is true in this example as LB'(fY)=2 and UB'(fY)=11.

Finally, as the bounds of fY have changed, Merger checks the consistency of the bounds of all other fake nodes. More specifically, it simulates the removal of fA and the new bounds of fY and compute the bounds adjustments needed using a *fake cost bounds propagation* procedure (§3.3.3). The condition is then verified and the changes applied if all adjustments yield acceptable bounds, and no locally scoped fake node is introduced by the procedure. This guarantees the absence of unwanted nexthop changes. In addition, this procedure also ensures that bounds across load-balanced paths are consistent with one-another, *i.e.,* that all possible fake path costs computed by the node splitting the traffic are equal. In our example, this requires to raise the lower bound of fA' to LB'(fA')=LB'(fY)+$dist$(A, Y, G)=12. Another iteration merges fc into fY, causing UB(fY) to be decreased to 4. Similarly, the upper bound of fA' thus becomes UB'(fA')=UB'(fY)+$dist$(A, Y, G)=14. Eventually, fA' is merged into fB, hence the final bounds of fB being $[12 - 2, 14 - 2]$ on Fig. 3.9b.

Step 3—Redundancy elimination. While smaller than Fig. 3.9a, Fig. 3.9b can still be reduced further. For example, fX creates a fake path that is redundant with the original IGP path of X, *i.e.,* maps to the same nexthop with an upper bound equal to its initial one. For each such fake node, Merger then attempts to remove it. Such removal is conceptually equivalent to setting LB(fX) = UB(fX) = $dist$(X, d, G). Similarly to the merging step, Merger then checks whether it can ensure the consistency of the lower bounds of all other fake nodes in the network and adjust them if needed. As this is the case for fX, it can thus be safely removed (and does not trigger any change of bounds in this particular example). This also applies to fZ, although removing it causes the lower bounds of fY to increase to 3 (which is then reflected on fB due to the equal-cost constraint).

Step 4—Fake path creation. Eventually, as the set of fake nodes F can no longer be reduced, Merger then creates one fake path per fake node. More specifically, it selects an integer $k \geq 0$ such that UB(fI) $\leq k \leq$ UB(fI) for any fake node fI $\in F$ and sets their total fake path cost to LB(fI)+k. Choosing a



(a) Multiple single-destination fake nodes can be merged together.

(b) A fake shortcut override the IGP link metric in one direction for all destinations.

Figure 3.10: Fake nodes present in multiple per-destination topology augmentation can be merged together to further reduce their number.

value for k is a tradeoff minimizing either the risk of unenforceable backup requirement (§3.2.5) if k is small, or minimizing the impact on other protocols' decision (§3.5.2) if k is high. As the final costs are now known, Merger then finally enforces all multiplicity attributes specified in the requirements. More specifically, for any router R which has a nexthop n with a multiplicity attribute m, Merger adds $m-1$ locally scoped fake nodes mapped to n with fake path costs set to create additional equal-cost paths for R in the augmented topology.

3.2.4 Cross-Destination optimization

Augmenting topologies on a per-destination basis may lead to the creation of redundant fake nodes. More specifically, fake nodes attached to the same router, mapping the same nexthop, but announcing different prefixes. In the cross-destination optimization step, we reduce such redundancy in two ways: *(i)* we merge redundant fake nodes; and *(ii)* we convert some fake nodes to fake shortcuts.

Cross-destination merging. Let G_1 and G_2 be the per-destination augmentation towards destinations d_1 and d_2. We iterate on every pair of fx_1 and fx_2, attached to the same router x mapping to the same nexthop n but announcing different destination prefixes d_1 and d_2. If fx_1 and fx_2 both have the same scoping type (*e.g.,* both are globally scoped), then we replace both of them by a single fake node fx', attached to x with a cost of 1, and also mapping to n. fx' then announces both destinations such that each d_i has an announcement cost of $dist(\mathsf{x}, d_i, G_i)-1$ for $i \in \{1, 2\}$, where $dist(\mathsf{x}, d_i, G_i)$ is, by construction, the total fake cost path of [x fx_i d].

For example, assume that an additional destination is attached to x in the network in Fig. 3.7a, with an associated forwarding requirement caus-

ing a fake node to be added on B and mapped to X. The result of the cross-destination merging is then shown in Fig. 3.10a, where B has a single fake neighbor announcing both destinations (rather than multiple fake neighbors each announcing a single destination), with different announcement costs for each destination.

Creating fake shortcuts. So far, all augmented topologies are composed of fake nodes able to attract multiple routers, for multiple specific destination prefixes. To reduce even further the size of the augmented topology, we replace when possible such fake nodes and destination announcements with fake paths connecting real routers, *i.e.,* fake shortcuts, without fake destination announcements.

Fake shortcuts are *directed* fake links. A fake shortcut enables to override the IGP metric of a link, in a given direction, for every destination in the network. Let R and N be two routers, and f_R be a fake node attached to R and mapped to N. f_R can be replaced if we can find a shortcut cost c such that no shortest path in the resulting augmented topology from R and from its neighbors to every destination is changed. This can require to adjust the lower bounds of the fake nodes still present in the topology.

We can add such fake shortcuts to our example from Fig. 3.10a. Indeed, as visible on Fig. 3.10b, we can find such cost $c = 1$ to replace f_B. In contrast, f_Y cannot be replaced by a fake shortcut as it is required to enforce C's desired nexthop, regardless of the cost of the link (Y, Z)[1].

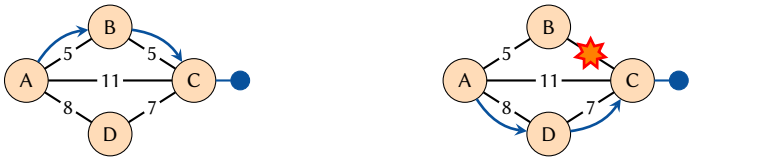
3.2.5 Implementing backup requirements

Similarly to primary requirements, backup requirements are represented using per-destination forwarding DAGs representing the desired forwarding paths after the failure of one its protected link set L. Satisfying them requires runing slightly modified versions of Simple and Merger. Let G be an initial topology, G' be an augmentation of G that satisfies primary requirements, and G'' a subsequent augmentation to implement the backup requirements.

Simple requires two modifications. First, to compute G', the algorithm always injects one fake path per nexthop for every router that has a changed nexthop set (*i.e.,* always executes line 11 in Alg. 1). Second, when computing G'', Simple sets the cost of the new fake paths to 3 instead of 2. This ensures that backup requirements can only be used if the primary ones have failed.

Implementing backup paths with Merger requires to modify its definition of upper and lower bounds when computing G''. More specifically, let f_R be a fake node towards a destination d and attached to a router R. The lower bound of f_R when computing G'' must be set to the total cost of the shortest-path

[1] Recall that link metrics in existing IGPs are strictly positive integers



(a) Primary forwarding requirement. (b) Impossible joint backup requirement.

Figure 3.11: Our current algorithms do not support all backup requirements.

P from R to d in G' plus one. This ensures that the fake node is only used as backup. Similarly, the upper bound of f_R is the minimal cost of all post-failure shortest-paths from R to d, different from P, minus one. Note that we only consider the paths computed on instances G' with one of the protected links removed (*i.e.,* single-failure cases). This ensures that the fake paths are the preferred post-failure paths.

Finally, once the per-destination algorithms have augmented the topology to implement the backup requirements, the cross-destination algorithms (§3.2.4) can then further optimize them without any modification.

Limitations. Our current augmentation algorithms may not always be able to *provision*[2], backup paths even in a Fibbing compliant topology. Indeed, our augmentation algorithms are unable to enforce a backup forwarding DAG D if there exists a post-failure path not included in D whose cost is the pre-failure shortest-path cost c, or $c + 1$. Consequently, it is impossible to restrict the set of post-failure paths to a subset of those used before the failure (*e.g.,* use only 1 path out of 3 equal-cost ones). Indeed, any such change would also implicitly restrict the set of paths during normal operations.

Consider the combination of primary and backup forwarding requirements visible in Fig. 3.11. As the primary requirement is already fulfilled by the default IGP shortest-paths (see Fig. 3.11a), no fake nodes will be added to the network by Simple nor Merger. In contrast, the backup requirement visible in Fig. 3.11b cannot be enforced. Indeed, to be used only after the failure of the primary path, the total cost of the new fake path [A f_A d] would need to be strictly higher than 10. At the same time, it should also be lower than 11 to be preferred to the default post-failure IGP path [A C D]. The backup requirement thus cannot be implemented. While a possible solution would be to attach a fake node on B to artificially decrease the cost of the primary path (*e.g.,* with a cost of 4), thus enabling to provision a backup fake node on A, this solution induces a greater control-plane overhead and is thus not applied automatically in our augmentation algorithms. Instead, if a backup requirement cannot be implemented, our algorithms report it to the operator.

[2]The controller can always enforce any backup forwarding DAG reactively (*e.g.,* as in §4.4).

3.3 The Merger algorithm

The description of Merger presented in (§3.2.3), while intuitive, is unpractical. Indeed, as the feasibility of any given merging attempt is constrained by the previous one, computing the *minimal* augmented topology requires to consider an exponential number of merging attempts.

For example, in Fig. 3.9a, merging first fy into fz causes the bounds fz to become [2,4]. This then prevents both fa and fc from being merged into fz. The resulting augmented topology is thus larger than the one in Fig. 3.9b, were fa and fc are merged before fy. More generally, identifying the minimal augmented topology requires to explore all permutations of merging attempts. As our example has 7 fake nodes, it defines up to 7! = 5040 possible merging sequences if all individual merging attempts could succeed (*e.g.,* if the requirement DAG is a subset of the current IGP paths).

The rest of this section details the greedy heuristic used by Merger to drastically reduce the number of merging attempts. More precisely, after introducing the notations used throughout this section (§3.3.1), we first present how the Merger heuristic (§3.3.2) guarantees a time complexity that is linear in the number of fake nodes. Then, we formalize the two key procedures at the heart of Merger: *(i)* the fake cost bounds propagation procedure (§3.3.3), which ensures that bounds are consistent across fake nodes; and *(ii)* the merging procedure (§3.3.4), which merges fake nodes when proved safe. Finally, we prove the correctness of Merger (§3.3.5), guaranteeing that the reduced topologies always implement the input forwarding requirements.

3.3.1 General notations

This sections uses the following notations.

Let G be a generic, non-augmented, weighted topology provided in input to Merger. $G', G'', \ldots$ are then the augmented topologies iteratively computed by Merger, *i.e.,* after each successive merging attempt.

For any graph G, $sp(s, d, G)$ and $dist(s, d, G)$ indicate the set of shortest-paths from s to d in G and their associated cost. Similarly, the cost of a path P in T is $cost(P, G)$, *i.e.,* $cost(sp(s, d, G), G) = dist(s, d, G)$. Likewise, $all\_paths(s, d, G)$ denotes all simple paths from s to d in G.

For any fake node fx, attached to a node x, nh(fx) is the node to which fx is mapped, *i.e.,* the real nexthop corresponding to the fake node.

For any topology G, if Merger successfully computes G' which implements a requirement DAG D towards a destination d, then all fake path costs in G' must be *consistently assigned*, *i.e.,* there exists an integer $k \geq 0$ such that for any globally scoped fake node fx, $cost([\text{x fx d}], G') = \text{LB}(\text{fx} + k)$.

Finally, a *guarantee* in an augmented topology (*e.g.,* about paths travers-

ing a specific fake node) is an assertion valid for any value of k used for consistently-assigned fake path costs.

3.3.2 A linear-time heuristic to reduce topologies

Merger limits the number of merging attempts to explore in two ways.

1. Bounding the number of possible merges. First, it limits the overall number of fake nodes, thus the total number of possible merges. To that end, in the initial fake node placement step (§3.2.3), it only adds fake nodes to routers whose required nexthop set is different from the one computed by the IGP. This effectively bounds the number of fake nodes of any augmented topology by the number of nexthop changes (instead of the total number of links in the requirement DAG). Note that, this restriction on the initial set of fake nodes effectively precludes the existence of redundant fake nodes, and thus removes the need to execute the third step presented in the overview of Merger (§3.2.3). In our example, the initial set of fake nodes is thus the same one as the one computed by Simple, shown in Fig. 3.8a, albeit with globally scoped fake nodes.

While the initial upper bounds of each of these fake nodes is as defined in (§3.2.3) (*i.e.,* set to $dist(\mathsf{R}, d, G) - 1$ for any fR attached to R in G), their lower bounds have to be computed carefully to ensure their consistency. Indeed, setting all of them to 2 no longer guarantees that a requirement DAG D towards destination d is correctly implemented. We address this by executing once the fake cost bounds propagation procedure (§3.3.3).

Note that Theorem 1 does not provide guarantees if fake nodes are connected only to nexthop changing nodes. As a result, some initial fake bounds could never be acceptable. We solve each of those cases by replacing the problematic fake node by a locally scoped one.

2. Attempting to merge a given fake node only once.

We now define the ordering in which merges are attempted. To ensure that Merger has a time complexity that is linear in the number of fake nodes, such ordering must consider the merging of each fake node at most once.

Let P=[$\mathsf{R}_1 \ldots \mathsf{R}_m$ d] be any source-sink path in D. For each such path, $F(P)$={$\mathsf{fN}_1, \ldots, \mathsf{fN}_k$} is then the set of fake nodes attached to nodes of P and implementing P in G'. We define a total order in $F(P)$ such that for any fN_i and fN_j in $F(P)$, $\mathsf{fN}_i < \mathsf{fN}_j$ if and only if the routers R_i and R_j to which fN_i and fN_j are respectively attached are such that R_i is an ancestor of R_j in D. In $F(P)$, fN_j is the successor of fN_i if $\mathsf{fN}_i < \mathsf{fN}_j$ and there are no fN_k such that $\mathsf{fN}_i < \mathsf{fN}_k < \mathsf{fN}_j$.

To define the ordering of its merging attempts, Merger iterates over all nodes in D, sorted according to a topological ordering [Kah62]. For each such

node R, Merger then checks whether there are any fake nodes fR, fR' … attached to R. For each such fR, let $\mathcal{P}$ be the set of paths from R to d in D that contains the sub-path [R nh(fR)]. Merger then tries to merge fR with its successor fS in any of the $F(P_i)$, with $P_i \in \mathcal{P}$. Irrespectively of the outcome of this procedure, fS will remain (possibly, with modified bounds if fR has been merged into it). When Merger reaches the last node of the topological order, it will have tried to merge every fake node exactly once.

Applied to the fake node placement from Fig. 3.8a, this yields the following ordering: {fA into fY, then fC into fY}. Note that as the topological sort defines a partial ordering, the reverse sequence would also be a valid ordering in this example. More generally, the intuition behind our heuristic is that if there exists a sequence of merging attempts such that all succeed, then any permutation of that sequence will yield the same augmented topology. Additionally, requirement DAGs tend to have more source-sink paths than they have sinks, *i.e.,* paths tend to collapse onto each other to end in common destinations. As such, merging from the sources towards the sinks reduces the risk of preventing multiple fake nodes from being merged into the same one (as would happen in Fig. 3.9a if fY is merged into fZ before trying fA or fC).

3.3.3 The fake cost bounds propagation procedure

The fake cost bounds propagation procedure takes as input the non-augmented topology G, the target forwarding DAG for D, a destination d, and a set of globally scoped fake nodes with their bounds, possibly initialized (§3.3.2). The procedure then ensures that the bounds are consistent across all fake nodes, by ensuring that three conditions are met for any fake node fX attached to a router X: *(i)* LB(fX) ≤ UB(fX); *(ii)* only X and its ancestors in D which do not have an attached fake node can compute a shortest-path in G' towards d ending by [X fX d]; and *(iii)* the range of consistently assigned fake path costs must be the same across equal cost paths.

To achieve this, the procedure first formalizes the constraints on any fake cost bounds as inequalities to be respected. Then, it fixes these bounds one by one, in a precise order, satisfying the computed inequalities. In the following, we provide a more detailed description of those two phases.

Formalization of the constraints on fake cost bounds

This is a static computation based on the link weights in G, and the set of input fake nodes. More specifically, consider any fake node fR attached to a node R. We first formalize the constraints between fR and any other fake node, and then between fR and nodes not attached to a fake node. In any case, we impose that bounds are sound, *i.e.,* that LB(fR) ≥ 2 and that UB(fR) < $dist$(R, d, G).

Let N ≠ R be any other node in G, to which Merger has also attached another fake node fN announcing d. We now distinguish between two cases, depending on the existence of a common ancestor M in D of both N and R (possibly N or R themselves), such that: *(i)* M is the root of at least two different paths P_1, P_2 in D; and *(ii)* fN (resp. fR) is the first fake node used to implement P_1 (resp. P_2) in G'.

If no such M exist, we then want to impose that the path [N … R fR d] is longer than [N fN d], ensuring that N uses its own fake node. Under the assumption of consistently assigned fake path costs, this property holds if LB(fN) < $dist$(N, R, G) + LB(fR). We therefore use this inequality as the formalization of the dependency between lower bounds LB(fR) and LB(fN).

Otherwise, fN and fR implement equal-cost paths for M. We then impose that the cost of both paths as computed by M is the same. This property is verified when both fake nodes have the same fake costs bounds relatively to M, *i.e.,* when both LB(fN) + $dist$(M, N, G) = LB(fR) + $dist$(M, R, G), and when UB(fN) + $dist$(M, N, G) = UB(fR) + $dist$(M, R, G). More generally, leveraging these equalities and taking into account their soundness constraints, we formalize the dependencies between bounds implementing equal-cost paths with the following inequalities: LB(fR) ≥ $max\{dist(\text{M}, \text{N}, G) + 2 - dist(\text{M}, \text{R}, G) \mid \text{fM} \in F\}$, and UB(fR) < $min\{dist(\text{M}, \text{N}, G) + dist(\text{N}, d, G) - dist(\text{M}, \text{R}, G) \mid \text{fM} \in F\}$, for any fake node fR ∈ F, such that F is the set of fake nodes implementing equal-cost paths for a node M

Similarly, let X be any node in G which does not have an attached fake node in G', and $\mathcal{P}$ be all simple paths from X to d in D. If at least one path in $\mathcal{P}$ is implemented in G' without any fake node, the shortest-path cost of X towards d is then fixed (*i.e.,* kept constant). This cost needs to be taken into account when computing the lower bound of fR. To that end, we treat X as if it had a fake node whose upper and lower bounds were equal to $dist$(X, d, G), and introduce the corresponding inequalities as detailed above. If all paths in $\mathcal{P}$ traverse at least one node attached to a fake node, then the final shortest-path cost from X to d depends on the values of those fake nodes. We can then safely ignore X when computing the bounds of fR, assuming that the set of input fake nodes was sound (*i.e.,* there exists a set of consistently assigned fake path costs able to implement D).

To come back to our example, using the same initial fake node locations as in Fig. 3.8a, the initial lower bound of fY (resp. fB) is 3 as it is constrained by Z (resp. X), which has a fixed shortest-path cost (12). On the other hand, fC's lower bound is 2 as it is not close enough to Z or X to attract them. Due to the ECMP constraint linking fA and fB, fA's lower bound is consequently set to 5. All of these fake nodes have the same upper bound as in Fig. 3.9a.

Computing concrete fake cost bounds

The fake cost bounds of a given fake node fx is likely to depends on the bounds of another fake node fn, which itself has constraints linking it back to fx, *i.e.,* fake nodes can have circular dependencies on their cost bounds. To break these cycles, this phase iterates over all of them and fixes them one a at time.

Let i be a generic iteration of the fake cost bounds propagation procedure. We split the set of input fake nodes in two, with $\mathcal{F}_i$ denoting the set fake nodes without concrete bounds at iteration i and $\mathcal{K}_i$ the one containing fake nodes with fixed bounds. For each fake node $\text{f}_\text{N} \in \mathcal{F}_i$, we define its *tentative bounds*, referred to as $\lfloor \text{LB}(\text{f}_\text{N})$ and $\lceil \text{UB}(\text{f}_\text{N})$, as the least constraining values of its bounds (*i.e.,* minimizing its lower bound and maximizing its upper bound) that satisfies its associated inequalities. More specifically, if a fake node $\text{f}_\text{R} \in \mathcal{K}_i$ appears in one the inequalities of f_N, we then use its concrete bounds to solve it (*i.e.,* $\text{UB}(\text{f}_\text{R})$ and $\text{LB}(\text{f}_\text{R})$ have been fixed in a previous iteration). Otherwise, we have $\text{f}_\text{N} \in \mathcal{F}_i, \text{f}_\text{N} \neq \text{f}_\text{X}$ and consider $\text{LB}(\text{f}_\text{R})$=2 and $\text{UB}(\text{f}_\text{R})=dist(\text{N}, d, G) - 1$ when solving inequalities involving f_R. Using these tentative bounds, the procedure then performs three steps.

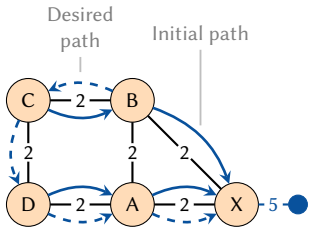
Step 1—Sorting fake nodes. We first sort all fake nodes in $\mathcal{F}_i$ according to the value of a specific function δ, in descending order. This function represents the relative influence of a fake node on the bounds of all the other ones. More precisely, the value of this function represents the minimal value that is imposed by a fake node on the lower bound of the closest fake node attached to a different node. That is, for a fake node fx, its δ value at iteration i is $\delta_i(\text{f}_\text{X}) = \lfloor \text{LB}(\text{f}_\text{X}) - min\{dist(\text{X}, \text{Y}, G) \mid y \in \mathcal{F}_i \cup \mathcal{K}_i\}$. If two nodes have the same δ_i value, any deterministic tie-breaker can then be used to have a total order in $\mathcal{F}_i$.

Step 2—Fixing one set of bounds. Let f_R be the first fake node in $\mathcal{F}_i$ (*i.e.,* the one with the highest δ_i). We fix both $\text{LB}(\text{f}_\text{R})$ and $\text{UB}(\text{f}_\text{R})$ to their current tentative values. We then remove f_R from $\mathcal{F}_i$ and add it to $\mathcal{K}_i$.

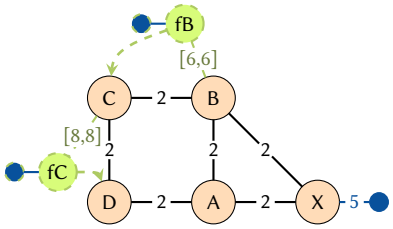
Step 3—Fixed bounds propagation. As f_R now has concrete bounds, we propagate these in the constraints of other fake nodes in $\mathcal{F}_i$ and update their tentative bounds accordingly. For example, consider any of the constraint inequalities ensuring that $[\text{A } \text{f}_\text{R} \; d]$ is the shortest-path in G' from R to d, *i.e.,* in the form of $\text{LB}(\text{f}_\text{R}) < dist(\text{R}, \text{N}, G) + \text{LB}(\text{f}_\text{N})$, with $\text{f}_\text{N} \in \mathcal{F}_i$. Such inequality is satisfied if $\text{LB}(\text{f}_\text{N})$ ends up having a value strictly greater than $\lfloor \text{LB}(\text{f}_\text{R}) - dist(\text{R}, \text{N}, G)$.

We thus update $\lfloor \text{LB}(\text{f}_\text{N})$ to $max\{\lfloor \text{LB}(\text{f}_\text{N}), \lfloor \text{LB}(\text{f}_\text{R}) - dist(\text{R}, \text{N}, G) + 1\}$.

Note that this propagation may render the bounds of f_N no longer acceptable, *i.e.,* $\lfloor \text{LB}(\text{f}_\text{N}) \leq \lceil \text{UB}(\text{f}_\text{N})$. In such cases, we remove all fake nodes attached to N, and implement its required nexthops using locally scoped fake nodes



(a) Implementing the desired forwarding paths requires to change the nexthop of B and C.



(b) Tentative fake cost bounds when considering only the constraints with nodes having a fixed cost shortest-path.

Figure 3.12: When propagating the fake cost bounds of fC, the lower bound of fB becomes greater than its upper bound, forcing Merger to replace fB by a locally scoped fake node.

with a cost of 2. Then, we restart the fake cost bounds computation from scratch, ignoring these locally scoped fake nodes when computing fake cost bounds. Fig. 3.12 shows a topology where this is the case. Indeed, due to D and A not having a fake node, fC and fB have to pick lower bounds high enough to not attract them. When executing the fake cost bounds propagation procedure with these tentative bounds, fC is then examined first since it has a larger delta value ($\delta_1(\text{fC}) = 6$; $\delta_1(\text{fB}) = 4$). The fake cost bounds of fC are thus fixed to [8,8]. Propagating the lower bound of fC then causes the constraint that fB should not attract C to evaluate as $\text{LB}(\text{fB}) > \text{LB}(\text{fC}) + dist(\text{C}, \text{B}, G)$, *i.e.,* $\text{LB}(\text{fB}) > 8 + 2$. This new tentative lower bound for fB is however greater than its upper bound (6), hence transform fB into a locally scoped fake node.

3.3.4 The merging procedure

The Merger algorithm reduces the size of an augmented topology by merging globally scoped fake nodes together, when deemed safe. Consider any two fake nodes fU and fV, part of the input set of fake nodes F. We now formally describe the three checks performed by Merger to determine if fU can be safely merged into fV, and what would be the resulting fake cost bounds changes across all fake nodes in the augmented topology.

Check 1—Shortest path compliance.
Merger first ensures that the post-merging paths are the ones in the forwarding requirements D. Let A be the set of nodes that will depend on fV to implement their required nexthops in D. That is, A includes U and all ancestors of U in D which do not have a fake node attached. Merger then checks that every node in S forwards its traffic towards V only along the desired paths, *i.e.,* that $\forall \text{N} \in A, sp(\text{N}, \text{J}, G) = all\_paths(\text{N}, \text{J}, D)$.

Check 2—Candidate compatibility.
Merger attempts to compute new provisional fake cost bounds for fv, that will meet both the original constraints on the bounds of fv and also those from fu. More precisely, these new bounds must guarantee that: *(i)* $sp(\text{u}, d, G')$ becomes the concatenation of $sp(\text{u}, \text{v}, G')$ and [v fv d]; and *(ii)* these new bounds must ensure that all nexthops in G' are kept unchanged before and after the merger.

Let lb'(fv) and ub'(fv) be the provisional lower and upper bounds of fv. Achieving the first constraint then implies that $\text{ub}'(\text{fv}) + dist(\text{u}, \text{v}, G) \leq \text{ub}(\text{fu})$, and thus that $\text{ub}'(\text{fv}) = min\{\text{ub}(\text{fb}), \text{ub}(\text{fu}) - dist(\text{u}, \text{v}, G)\}$, *i.e.,* the provisional upper bound must guarantee that the fake node will announce a total fake path cost that is preferred to the default igp paths of both u and v.

Similarly, assuming consistently assigned fake path costs, achieving the second constraint imposes that $\forall \text{fn} \in F \setminus A$, $\text{lb}(\text{fn}) < dist(\text{n}, \text{v}, G) + \text{lb}'(\text{fv})$, *i.e.,* no node but those in A can compute a shortest-path that goes through fv. Note that if we assume that the original lower bounds of fu and fv were consistent (*i.e.,* were the results of the fake cost bounds propagation procedure), we can then reformulate this as $\text{lb}'(\text{fv}) = max\{\text{lb}(\text{fv}), \text{lb}(\text{fu}) - dist(\text{u}, \text{v}, G)\}$. Indeed, as the shortest-path compliance check succeeded, subtracting $dist(\text{u}, \text{v}, G)$ from lb(fu) effectively preserves the constraints on the lower bound of fu while "moving" it on fv (*i.e.,* normalizing it). This enables to compare the two lower bounds in order to keep the most constrained one, meeting the constraints of both fake nodes.

Merger then finally checks that these provisional bounds ensures the existence of a fake path cost implementing D in G', *i.e.,* $\text{lb}'(\text{fv}) \leq \text{ub}'(\text{fv})$.

Check 3—Network-wide feasibility.
Merger then *simulates* the effects of merging fu into fv to assess the needed fake cost bounds changes, as well as their feasibility. Indeed, the removal of fu and the possibly changed bounds of fv could introduces some bounds inconsistencies in G'. More precisely, the second check only guarantees that fv proposes a path preferred to the igp one, and that no "unwanted" node uses fv. It does not guarantee that fv will be the preferred one by all nodes in A, or that equal-cost constraints are met.

To ensure the network-wide consistency of fake cost bounds, Merger creates g as a copy of G which does not contain fu and has the bounds of fv set to their provisional values. Then, it executes the fake cost bounds propagation procedure on g. If the fake cost bound propagation succeeds without adding locally scoped fake nodes, this third check then succeeds.

If any of those three checks fails, Merger immediately aborts the merging attempt. Otherwise, we set G' to g, *i.e.,* the augmentation of G now has one less fake node, and Merger attempts to merge another pair of fake nodes.

3.3.5 Merger is provably correct

We prove the correctness of Merger, *i.e.,* its ability to implement any forwarding DAG with a reduced augmented topology, in two steps. First, we show that the algorithm correctly implements any input forwarding DAG after the fake cost bounds computation procedure (see Theorem 2). Second, we prove that merges performed during the merging procedure never change the forwarding paths implemented in the pre-merging augmented graph (see Theorem 3). The correctness of Merger then follows by noting that it first executes the fake cost bounds propagation procedure, and then iteratively executes the merging procedure followed by the fake cost bounds propagation.

Note that as Merger never changes any link or link weight in G, for any pair of real nodes x and y in G, we always have $dist(\mathsf{x}, \mathsf{y}, G) = dist(\mathsf{x}, \mathsf{y}, G')$.

Correctness of the fake cost bounds propagation procedure

Let G be the original IGP topology, and G' the topology as computed by Merger after the fake cost bounds propagation procedure.

Lemma 1 (Once fixed, fake cost bounds are never changed during the procedure)**.** *If* LB(fx) *is fixed at an iteration i of the fake cost bounds propagation procedure, then $\delta_i(\mathsf{fx}) = \delta_j(\mathsf{fx})$ for any iteration j with $j > i$.*

Proof. Recall that the δ function is the difference between LB(fx) and the distance between x and the closest real node attached to a fake node. Proving that $\delta_i(\mathsf{fx}) = \delta_j(\mathsf{fx})$ for any iteration j with $j > i$ implies to prove that once fixed, LB(fx) stays constant. Assume by contradiction that there exists another iteration j, with $j > i$ in which LB(fx) is changed. More precisely, as lower bounds may only increase during the propagation phase in step 3, we then have $\text{LB}_j(\mathsf{fx}) > \text{LB}_i(\mathsf{fx})$. This implies that there exists a fake node fz at iteration j such that $\text{LB}_j(\mathsf{fz}) - dist(\mathsf{x}, \mathsf{z}, G) + 1 > \text{LB}_i(\mathsf{fx})$. By definition of the δ function, this then means that $\delta_j(\mathsf{fz}) \geq \text{LB}_i(\mathsf{fx}) > \delta_i(\mathsf{fx})$. This would cause fake node fz to be fixed before fx as $\delta_j(\mathsf{fz}) > \delta_j(\mathsf{fx})$, which contradicts the hypothesis that $j > i$ and thus proves the statement. □

Lemma 2 (The δ function defines a constant total ordering for fixed bounds)**.** *If* LB(fx) *is fixed at an iteration i of the fake cost bounds propagation procedure and $\delta_i(\mathsf{fx}) \geq \delta_i(\mathsf{fy})$ at that iteration, then $\delta_j(\mathsf{fx}) \geq \delta_j(\mathsf{fy})$ at any iteration $j > i$.*

Proof. Assume by contradiction that $\delta_i(\mathsf{fx}) \geq \delta_i(\mathsf{fy})$ but $\delta_j(\mathsf{fx}) < \delta_j(\mathsf{fy})$ in a latter iteration j, such that $j > i$ and $\text{LB}(\mathsf{fy}) \neq \text{LB}(\mathsf{fy})$. Let m be the closest iteration to i such that $\delta_m(\mathsf{fx}) < \delta_m(\mathsf{fy})$. By Lemma 1, if LB(fx) is fixed at i, then $\delta_l(\mathsf{fx}) = \delta_i(\mathsf{fx})$ for any $l > i$, hence $\delta_i(\mathsf{fx}) = \delta_m(\mathsf{fx})$. Let x and y be the real nodes respectively attached to x and y.

One of the following two cases must then apply.

- $m = i+1$. In this case, the delta function of f_Y must have been increased during the i-th iteration. Such increase then implies that the lower bound of f_Y gives us $\lfloor \text{LB}_m(f_Y) = max\{\lfloor \text{LB}_i(f_Y), \text{LB}_i(f_X) - dist(Y, X, G) + 1\}$ We have two sub-cases.

  If $\text{LB}_m(f_Y) = \text{LB}_i(f_Y)$, then $\delta_m(f_Y) = \delta_i(f_Y)$. As we have by hypothesis, $\delta_i(f_X) \geq \delta_i(f_Y)$ and $\delta_i(f_X) = \delta_m(f_X)$, we must then have $\delta_m(f_X) \geq \delta_m(f_Y)$.

  Otherwise, we have $\text{LB}_m(f_Y) = \text{LB}_i(f_X) - dist(Y, X, G) + 1$, which we can reformulate as $\text{LB}_m(f_Y) - 1 = \text{LB}_m(f_X) - dist(Y, X, G)$. Per its definition, the value of $\delta_m(f_Y)$ is $\text{LB}_m(f_Y) - cost(P, G)$, where P is a given path in G; as the cost of any IGP path is strictly positive, we thus have $\delta_m(f_Y) \leq \text{LB}_m(f_Y) - 1$. Moreover, $\delta_m(f_X) = \text{LB}_m(f_X) - dist(X, Z, G)$ with Z being the closest node to X also attached to a fake node; consequently, $\delta_m(f_X) \geq \text{LB}_m(f_X) - dist(X, Y, G)$. Combining those, we have $\delta_m(f_Y) \leq LB_m(f_Y) - 1 = \text{LB}_m(f_X) - dist(X, Y, G) \leq \delta_m(f_X)$, thus that $\delta_m(f_X) \geq \delta_m(f_Y)$.

  Both sub-cases generate a contradiction, as m was defined as an iteration in which $\delta_m(f_X) < \delta_m(f_Y)$.

- $m > i + 1$. In this case, there must be an iteration n, with $i < n \leq m$, such that $\delta_{n-1}(f_X) \geq \delta_{n-1}(f_Y)$ and $\delta_n(x) < \delta_n(y)$. As $\text{LB}(f_X)$ is fixed in iteration i, $\delta_{n-1}(f_X) = \delta_n(f_X) = \delta_i(f_X)$. $\text{LB}(f_Y)$ thus can only be fixed at iteration n and must increase by hypothesis between iteration $n-1$ and n, implying that $\delta_n(f_Y) > \delta_{n-1}(f_Y)$.

  Consider a lower bound $\text{LB}(f_Z) \neq \text{LB}(f_Y)$ being fixed at $n-1$. Fixing it at iteration $n-1$ must cause $\delta_n(f_Y) > \delta_{n-1}(f_Y)$. Additionally, if $\text{LB}(f_Z)$ is being fixed at $n-1$, then we have $\delta_{n-1}(f_Z) \geq \delta_{n-1}(f_Y)$. This leads to two subcases.

  If $\delta_{n-1}(f_Z) \geq \delta_{n-1}(f_Y)$ and $\delta_n(f_Z) < \delta_n(f_Y)$, we can consider Z instead of X and immediately generate a contradiction on Z and Y as the first case of our proof applies (*i.e.,* $m = n$ and $i = n-1$).

  Otherwise, if $\delta_{n-1}(f_Z) \geq \delta_{n-1}(f_Y)$ and $\delta_n(f_Z) \geq \delta_n(f_Y)$, we obtain by substitution $\delta_{n-1}(f_X) = \delta_n(f_X) < \delta_n(f_Y) \leq \delta_n(f_Z) = \delta_{n-1}(f_Z)$. Recall that $\text{LB}(f_X)$ is fixed at i, thus that we also have $\delta_i(f_X) \geq \delta_i(f_Z)$. As $\delta_{n-1}(f_X) < \delta_{n-1}(f_Z)$, we can then iterate our reasoning by contradiction on X and Z. Note that as $n-1 < m$, we have the guarantee that we will eventually fall in another subcase than this one, and thus finally generate a contradiction.

  All subcases thus generate a contradiction, which proves the statement. □

Lemma 3 (Nodes always use the fake nodes attached to them)**.** *The shortest path to d from any real node R attached to a fake node f_R which announces d in G' is $[R\ f_R\ d]$.*

Proof. The statement holds if the fake node is locally scoped, as the fake cost propagation procedure sets its cost to 2 and the topology is Fibbing compliant.

We now focus on globally scoped fake nodes.

Consider any pair of lower bounds LB(fx) and LB(fy). Let i_x and i_y be the iterations at which LB(fx) and LB(fy) are respectively fixed during the fake cost bounds propagation procedure. We denote the final value of δ(fx) (resp. δ(fy)) after the last iteration of the procedure as δ_f(fx) (resp. δ_f(fy)). Moreover, we denote by P_x (resp. P_y) the path [x fx d] (resp. [y fy d]).

By definition of the procedure, we have two cases.

- LB(fx) is fixed before LB(fy), that is, $i_x < i_y$. Assuming consistently assigned fake path costs, we have $cost(P_y, G') = \text{LB}_f(\text{fy}) + k$ with $k \geq 0$, thus $cost(P_y, G') \geq \text{LB}_f(\text{fy})$. Additionally, the bounds propagation step of the procedure ensures that $\text{LB}_f(\text{fy}) > \text{LB}_f(\text{fx}) - dist(\text{x}, \text{y}, G)$. The right side of the previous inequality can be rewritten as $cost(P_x, G') - k - dist(\text{x}, \text{y}, G)$. This implies that $\text{LB}_f(\text{fx}) - dist(\text{x}, \text{y}, G) \geq cost(P_x, G') - dist(\text{x}, \text{y}, G)$. Using all previous inequalities, we conclude that $cost(P_y, G') > cost(P_x, G') - dist(\text{x}, \text{y}, G)$.

- LB(fx) is fixed after LB(fy), *i.e.,* $i_x > i_y$. This implies that $\delta_{i_x}(\text{fx}) \leq \delta_{i_x}(\text{fy})$, thus that $\delta_f(\text{fx}) \leq \delta_f(\text{fy})$ by using Lemma 2. We now express δ_f(fx) and δ_f(fy) with respect to the cost of paths in G'.

By definition, $\delta_f(\text{fx}) = \text{LB}_f(\text{fx}) - dist(\text{x}, \text{n}, G)$, with n being the closest node to x that is attached to a fake node. Consequently, $\delta_f(\text{fx}) \geq \text{LB}_f(\text{fx}) - dist(\text{x}, \text{y}, G)$, *i.e.,* $\delta_f(\text{fx}) \geq cost(P_x, G') - k - dist(\text{x}, \text{y}, G)$. By using the consistently assigned fake path costs definition, *i.e.,* $k \geq 0$, we have $\delta_f(\text{fx}) \geq cost(P_x, G') - dist(\text{x}, \text{y}, G)$.

Similarly, we have $\delta_f(\text{fy}) = \text{LB}_f(\text{fy}) - dist(\text{y}, \text{z}, G')$ for some node z; thus $\delta_f(\text{fy}) = cost(P_y, G') - k - dist(\text{y}, \text{z}, G')$. As both $dist(\text{y}, \text{z}, G) > 0$ and $k \geq 0$, we thus have $\delta_f(\text{fy}) < cost(P_y, G')$.

Finally, we combine the previous inequalities around δ_f(fy) and δ_f(fx) to obtain $cost(P_y, G') > \delta_f(\text{fy}) \geq \delta_f(\text{fx}) \geq cost(P_x, G') - dist(\text{x}, \text{y}, G)$.

Both cases prove that $cost(P_y, G') > cost(P_x, G') - dist(\text{x}, \text{y}, G')$, *i.e.,* that the path P_x is always shorter for x to reach the destination, than going from x to y and then using P_y. The same argument can be applied to any other lower bound LB(fy), upholding the conclusion that P_x, *i.e.,* [x fx d], is the shortest path from x to d in G'.

The statement then follows by applying the same argument to all nodes x in G'. □

Lemma 4 (The procedure does not induce any nexthop change)**.** *For any real node* n *not attached to a fake node, if* $all\_paths(\text{n}, d, D)$ *is guaranteed not to contain a node* x *attached to a fake node* fx *before the fake cost bounds propagation procedure, then* $sp(\text{n}, d, G')$ *is guaranteed not to contain* x *after the procedure.*

Proof. Using the formalization of the constraints on the lower bound, we have $dist(\text{N}, \text{X}, G) + \text{LB}(\text{fX}) > dist(\text{N}, d, G')$. We then have two cases.

- $all\_paths(\text{N}, d, D)$ crosses no node attached to a fake node. X thus has a fixed shortest-path cost, and the statement then follows as the propagation procedure guarantees that LB(fX) will either satisfy the inequality, or be replaced by a locally scoped fake node which would then not affect N.

- $all\_paths(\text{N}, d, D)$ crosses a node Z attached to a fake node fZ. We then have $dist(\text{N}, d, G') = dist(\text{N}, \text{Z}, G) + \text{LB}(\text{fZ})$. By Lemma 3, and assuming consistently assigned fake path costs, the statement then holds. □

Theorem 2 (The fake cost bounds propagation procedure is correct)**.** *If fake path costs are consistently assigned, Merger correctly implements the input forwarding DAG D after the fake cost bounds propagation procedure.*

Proof. Consider any real node R in G'. We then have two cases.

- If R is not attached to any fake node, the assignment of lower bound values before the propagation procedure ensures that paths including any fake node connected to real ones not in $all\_paths(\text{R}, d, D)$ have a cost strictly greater than $dist(\text{R}, d, G')$. Lemma 4 then guarantees that R's shortest path $sp(\text{R}, d, G')$ in the augmented topology will not traverse any fake node attached to a real one not in $all\_paths(\text{R}, d, D)$. Per the definition of shortest-paths, $sp(\text{R}, d, G')$ can actually be written as the concatenation of $P = sp(\text{R}, \text{X}, G')$ and $Q = [\text{X fX } d]$, with either: *(i)* $Q \neq \emptyset$ and $\text{X} \in sp(\text{R}, d, G)$; or *(ii)* $Q = \emptyset$ and $\text{R} = d$ if no node in $sp(\text{R}, d, G)$ is connected to a fake one. In either case, R's next-hop is the same as in the original topology. This is consistent with the input forwarding DAG, given that Merger initially adds fake nodes to (at least) all nexthop changing nodes.

- Otherwise, if R is connected to a fake node, then Lemma 3 holds. The required nexthop is thus imposed via the fake path, yielding the statement. □

Correctness of the merging procedure

We now prove that the merging procedure does not trigger violations of previously-enforced forwarding DAGs. We begin by first proving an invariant preserved by the procedure.

Lemma 5 (The merging procedure does not induce any nexthop change)**.** *For any pair of real nodes (R, Z) and any fake node fZ attached to Z and announcing a destination d, if we have $dist(\text{R}, d, G^1) < dist(\text{R}, \text{Z}, G^1) + LB_1(\text{fZ})$ in the topology G^1 provided as input to the merging procedure, then we also have $dist(\text{R}, d, G^k) < dist(\text{R}, \text{Z}, G^k) + LB_k(\text{fZ})$ for any iteration k of the procedure.*

Proof. Consider any iteration k of the merging procedure, and assume that $dist(\text{R}, d, G^{k-1}) < dist(\text{R}, \text{Z}, G^{k-1}) + \text{LB}_{k-1}(\text{fZ})$ held. At iteration k, Merger picks two fake nodes fA and fB, and attempts to merge fA into fB by performing the three checks of the merging procedure.

- If the merging attempts fails, then no change has been applied to G^{k-1} has the set of fake nodes is unchanged and all fake cost bounds remain the same. In that case, $dist(\text{R}, d, G^{k-1}) < dist(\text{R}, \text{Z}, G^{k-1}) + \text{LB}_{k-1}(\text{fZ})$, directly implies $dist(\text{R}, d, G^{k}) < dist(\text{R}, \text{Z}, G^{k}) + \text{LB}_{k}(\text{fZ})$.

- Otherwise, fA is merged into fB. The candidate compatibility step sets LB'(fB) such that $dist(\text{R}, d, G^{k-1}) < dist(\text{R}, \text{B}, G^{k-1}) + \text{LB}'(\text{fB})$. All other lower bounds satisfy the corresponding inequality by hypothesis. By definition no lower bound is decreased during the fake cost bounds propagation procedure run in the network-wide feasibility step, *i.e.,* $\text{LB}_{k-1}(\text{fX}) \leq \text{LB}_{k}(\text{fX})$ for any fake node fX. As Merger does not change link weights, we thus have $dist(\text{R}, d, G^{k-1}) = dist(\text{R}, d, G^{k})$ and $dist(\text{R}, \text{Z}, G^{k-1}) = dist(\text{R}, \text{Z}, G^{k})$. Combining all of these, this implies that at the end of the iteration k, for any fake node fZ, $dist(\text{R}, d, G^{k}) < dist(\text{R}, \text{Z}, G^{k-1}) + \text{LB}_{k-1}(\text{fZ}) \leq dist(\text{R}, \text{Z}, G^{k}) + \text{LB}_{k}(\text{fZ})$, yielding the statement. □

Theorem 3 (The merging procedure is correct)**.** *In an augmented topology G' implementing a forwarding DAG D for a destination d in a topology G, successfully merging two fake nodes together using the merging procedure produces an augmented topology G'' which correctly implements D.*

Proof. Consider two fake nodes in G' fA and fB, such that the merging procedure successfully merged fA into fB to produce G''. Let A (resp. B) be the real node to which fA (resp. fB) is attached.

For any real node R, one of the following cases holds.

- $sp(\text{R}, d, G')$ includes fA before the merger, *i.e.,* R=A or R is one of the ancestors of A in D which is not attached to a fake node. The candidate compatibility check then ensures that $dist(\text{R}, \text{B}, G'') + \text{LB}''(\text{fB}) < dist(\text{R}, d, G)$, *i.e.,* the path created by reaching Y from R and then going through fY is shorter than R's original IGP shortest-path. Additionally, initial fake cost bounds propagation ensures that we have $dist(\text{R}, d, G^0) < dist(\text{R}, \text{X}, G^0) + \text{LB}_0(\text{fX})$ for any fake node $\text{fX} \notin \{\text{fA}, \text{fB}\}$ before the first execution of the merging procedure. Lemma 5 then guarantees that this property is preserved in subsequent merging attempts, *i.e.,* that $dist(\text{R}, d, G'') < dist(\text{R}, \text{X}, G'') + \text{LB}''(\text{fX})$ holds. Combining both inequalities, we conclude that $sp(\text{R}, \text{Y}, G'')+[\text{B fB } d]$ is guaranteed to be the post-merging shortest-path of X. Finally, the shortest-path compliance check guarantees that $sp(\text{R}, \text{B}, G'') = sp(\text{R}, \text{A}, G'') + sp(\text{A}, \text{B}, G'')$, thus R has the same set of nexthops in G'' and in G'.

- $sp(\text{R}, d, G')$ includes neither f_A nor f_B before the merger. We then have two subcases: *(i)* Either R is attached to a locally scoped fake node. R then keeps using this fake node as the associated total fake path cost (2) is always the lowest; or *(ii)* R implements its required nexthops in D by according to its shortest-path towards d in G'' (with or without traversing a fake node). By hypothesis as the merging attempt succeeded, the fake cost bounds propagation ran at the end of the merging procedure was then successful. Theorem 2 then implies that all fake cost bounds network-wide have been adjusted to preserve the required nexthops of R.

In both cases, R has the same nexthops in G' and G'', which proves the statement. □

3.4 Evaluation

This section presents experimental results confirming the feasibility of Fibbing. More specifically, we evaluate Simple and Merger (§3.2) according to two criteria. First, we record the time they take to compute an augmented topology for a given requirement (§3.4.1), and confirm that a Fibbing controller can quickly compute augmented topologies in reaction to network events. Then, we show that our optimization algorithms efficiently reduce the size of the augmented topology (§3.4.2), preserving the scalability of the underlying IGP by lowering the control-plane overhead.

Our evaluation is based on simulations performed on realistic ISP topologies [SMW02], whose sizes range from 80 nodes to over 300, with up to 2 000 links. We generated between 100 and 500 different forwarding requirements for each of these topologies, depending on their size. Generating one such forwarding requirements is a two steps process. First, we select a random node in the topology, to which we attach a particular destination prefix. Then, we introduce a controlled amount of random shortest-path violations (*i.e.,* nodes whose set of nexthops has been changed in the generated requirement DAG), such that the resulting set of forwarding paths towards the destination is still loop-free.

These benchmarks are performed using naive implementations of Simple and Merger in Python, spanning across about 1700 lines (comments included). As such, they should only be taken as baseline and could vastly be improved (*e.g.,* by switching to a more efficient language, caching intermediate computations more aggressively, or parallelizing some parts such as the merging attempts).

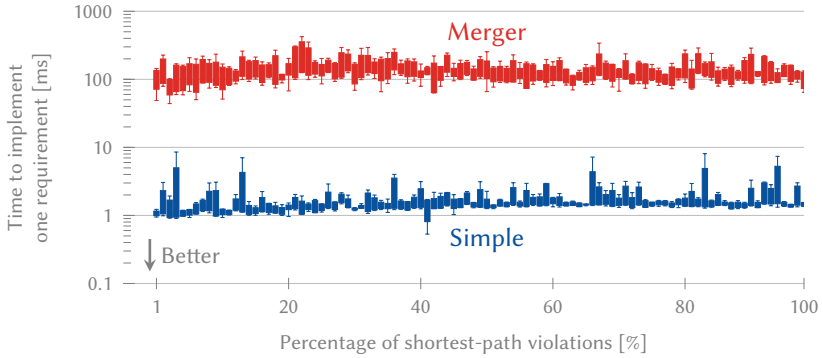


Figure 3.13: Simple and Merger quickly implement a forwarding requirement.

3.4.1 Fibbing augments network topologies within ms.

Fig. 3.13 plots the distributions of the time taken by Simple and Merger to compute an augmented topology (on the y-axis), against the number of nodes whose desired nexthops violate the original IGP shortest-paths (on the x-axis). Boxes contain the second and third quartiles, while and whiskers denote the minimal and maximal values. This graph only displays results from the benchmarks performed on the largest Rocketfuel topology (AS1239).

Simple was able to compute one per-destination augmentation in milliseconds, varying from 0.5ms to 8ms. As expected, using Merger to implement the requirements was comparatively slower as it resulted in running times in hundreds of milliseconds. Yet, these running times still allow using Merger to optimize augmented topologies in near real-time. Note that the execution time for both algorithms does not vary much with the number of shortest-paths violations.

3.4.2 Fibbing effectively optimizes augmented topologies.

Fig. 3.14 compares the relative size of the augmented topology (on the y-axis) between Simple and Merger, depending on the number of shortest-paths violations (on the x-axis), across all topologies. More precisely, the y-axis displays the ratios between the total number of fake nodes in the augmented topologies, and the total number of nexthop changes (which tends to be greater than the number of shortest-path violations due to equal-cost paths). The boxplots display the second and third quartiles, as well as the overall minimum and maximum values across all experiments.

On one hand, Simple always adds one fake node per nexthop change and is thus a baseline on the number of fake nodes needed. On the other hand, Merger was able to optimize some of those fake nodes across all exper-

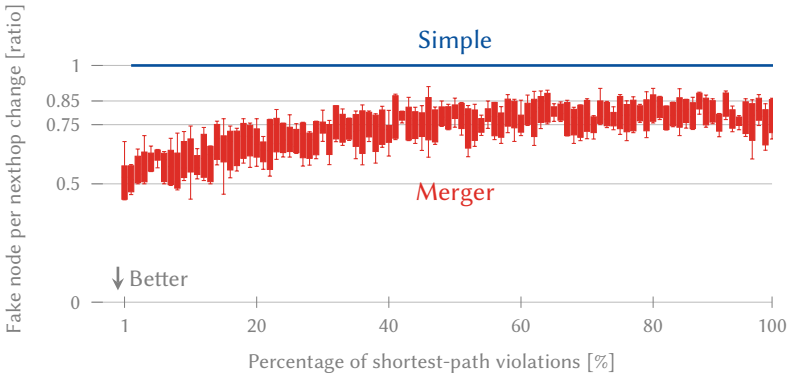


Figure 3.14: Merger reduced the augmented topology in all experiments.

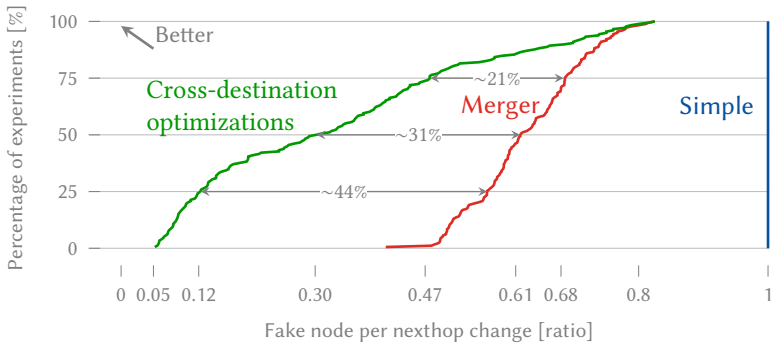


Figure 3.15: The cross-destination optimization algorithms drastically reduce augmented topologies implementing more than one forwarding DAG.

iments. This resulted in augmented topologies smaller by about 25% in the average case, and up to 66% in the best case (*i.e.,* Merger produced and augmented topology containing 44% of the fake nodes placed by Simple). Note that Merger's effectiveness tends to decrease as the number of shortest-path violations grows. This is expected as requiring more nexthop changes implicitly reduces the likelihood of Merger finding pairs of fake nodes that: *(i)* meet the shortest-path compliance criteria; and *(ii)* have compatible fake cost bounds despite the increasing amount of equal-cost constraints.

Finally, Fig. 3.15 confirms the effectiveness of our cross-destination optimizations algorithms (§3.2.4). To that end, it compares the cumulative distribution function (CDF) of the relative size of the augmented topology (on the x-axis) computed by Simple, Merger, and the cross-destination algorithms applied after Merger. More specifically, the plot aggregates the results of benchmarks concurrently implementing a random number of forwarding requirements on the same initial topology (between 1 and 100).

We see that in 75% of those benchmarks, the cross-destination optimizations were able to reduce the size of the augmented by at least an extra 20% with respect to Merger, which itself was using 32% less fake nodes than Simple. As the number the of concurrent requirements grows, so does the optimization gain of the cross-destination algorithms, as half of all experiments sees a gain of 31% less fake nodes compared to Merger, up to 44% in 25% of the experiments. More generally, in a network with L physical links, the cross-destination algorithms admit an upper bound on the total number of fake nodes equals to $2 \times L$ regardless of the number of forwarding requirements forwarding DAGs or of the number of nexthop changes (*i.e.,* one fake node for each direction of every link). In contrast, Simple and Merger only provide an upper bound on a per-destination basis.

3.5 Using Fibbing jointly with other protocols

IGPs are seldom the only routing protocols in use in a network, and often interact with the other ones in a way or another (sometimes unexpectedly [GW02b]). Fibbing thus inherits from these interactions. More specifically, using Fibbing has two effects on the underlying IGP topology (*i.e.,* the augmented topology): *(i)* some destinations use different shortest-paths; and *(ii)* the cost of these altered paths is lowered. As a result, any protocol depending on the path computed by the IGP, or on the cost associated to its routes, is also affected by the changes induced by Fibbing.

In this section, we present how Fibbing can leverage these effects on the IGP to control the forwarding behaviors of other protocols. We first discuss how Fibbing achieves a direct control over the paths used by any protocols

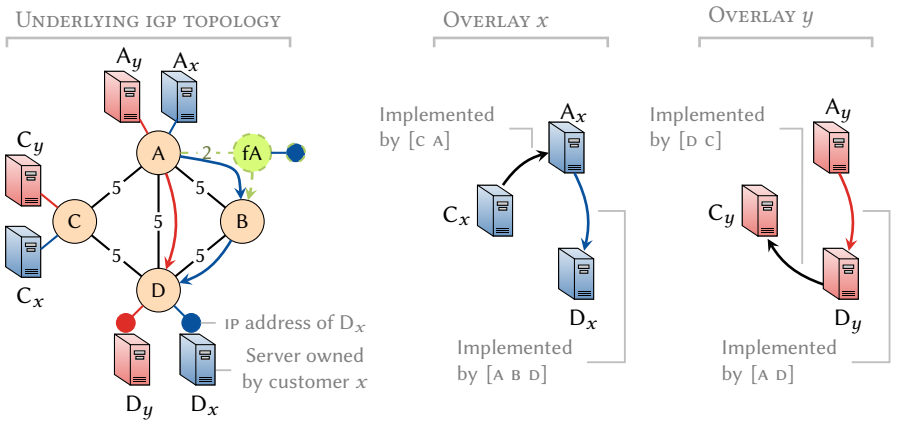


Figure 3.16: Fibbing controls the paths of tunnels building overlay networks.

building an overlay on top of the IGP (§3.5.1), such as LDP or GRE. In particular, we analyze the interactions between Fibbing and the BGP, highlighting how Fibbing controls the paths followed by transit traffic (§3.5.2) and how Fibbing enables an indirect control over the result of the BGP decision process.

3.5.1 Fibbing controls the routes used by overlay networks

Overlay networks are virtual network topologies built on top of an underlying (physical) topology. There are many use-cases for such overlay, ranging from multiplexing different data-planes on the same infrastructure (*e.g.*, routing MPLS packets in IP networks using LDP), to creating virtual networks spanning between multiple data-centers and isolating them across customers (*e.g.*, NVo3 [RFC8151], GRE [RFC2784], VxLAN [RFC7348], layer-2 VPN [RFC4664]), or even relaying application data across multiple logical nodes using proxies (*e.g.*, SSH gateways, HTTP load-balancers). In each of these cases, the traffic belonging to the overlay network is encapsulated, and then forwarded between the virtual nodes over a well-defined set of paths in the underlying topology, which we call tunnels. If the underlying topology is an IP network running an IGP, then the encapsulation specifies a source and a destination IP addresses that will act as tunnel endpoints, and the tunnels follow the shortest-paths between these endpoints.

Fibbing explicitly controls all underlying paths building up overlay networks. Indeed, as tunneling protocols rely on the IGP to determine the actual the paths between two tunnels endpoints, controlling those paths with Fibbing is achieved by specifying forwarding requirements towards each endpoint, as for any other destination. Such endpoints can be any IP address resolvable by the IGP, ranging from router loopbacks (*e.g.*, used by LDP to re-

solve MPLS labels), to servers' IP addresses (*e.g.,* for application proxies, GRE), including virtual machines' addresses. Additionally, if the network configuration allows it, multiple tunnels can be re-routed at once using Fibbing by specifying requirements towards the prefix aggregating all endpoints (*e.g.,* re-routing all virtual machines hosted on the same rack, assuming they all share a common prefix). Finally, note that the requirement language compilation process (§3.1.1) guarantees that Fibbing never unexpectedly alters the paths followed by tunneled traffic, *i.e.,* it only does so if an endpoint's IP is part of the requirements' destination prefixes.

For example, consider the IGP topology shown on the left part of Fig. 3.16, and hosting multi-tenants data-centers linked to routers A, C, and D. In these data-centers, two different customers have built an overlay connecting servers which implements the different processing steps of some data-intensive application. Assume that due to these customers, the operator of the network then detects that the link (A, D) is congested due to it being part of both overlays. The operator can then use Fibbing to re-route one of the tunnels implementing $[A_x\ D_x]$, by specifying that traffic towards the IP address of one the tunnel endpoints on D should be sent over an alternate path(*e.g.,* [A B D]). Accordingly, the controller will inject one fake node in the IGP to implement it. This causes the two tunnels connecting A to D to follow different IGP paths, while leaving their overlay topologies unchanged. Note that this process made no assumption on the actual protocol used to build the overlay.

3.5.2 Fibbing influences the BGP decision process

BGP routes transit traffic entering the network by first selecting an egress router and then relying on the IGP to forward it internally towards that egress, possibly encapsulated. BGP thus implicitly builds an overlay network, creating a fullmesh between all ingresses and egresses, dedicated to transit flows. As shown earlier (§3.5.1), specifying forwarding DAGs towards those egresses (*i.e.,* routers' loopback if the BGP routes are announced using the `next-hop-self` option) then controls the paths along which transit flows are forwarded.

The interactions between BGP and the underlying IGP, however, go beyond the paths followed by transit traffic. Indeed, every BGP router directly taps into the IGP state to extract the internal path cost to reach BGP egresses. These costs are then reflected in the MED attribute (exported to other ASes the to influence their preferred ingress routers), and also used as one of the later steps of the BGP decision process. Uncontrolled, this interaction between BGP and the IGP is a well-known source of issues (*e.g.,* forwarding loops, route oscillations) and has been widely researched [GW99; GW02a; GW02b; Van+13].

Fibbing controls transit paths. Consider the example network shown on Fig. 3.17. In this, network, two routers (D and Z) have established an eBGP

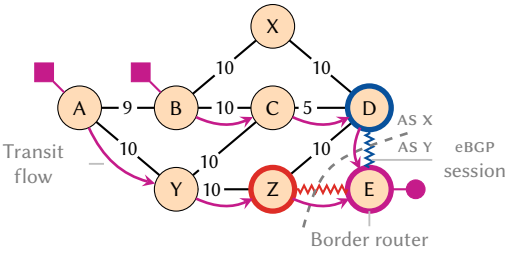
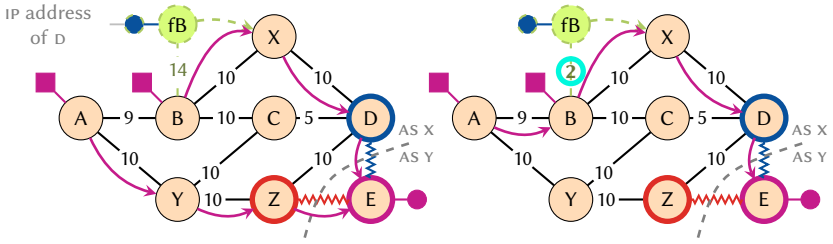


Figure 3.17: Routers A and B each selects a different egress router by using the IGP cost to reach them as tie-breaker in the BGP decision process.



(a) One globally scoped fake node announcing D's address. can re-route transit traffic between B and D to go through X.

(b) With a low enough fake path cost for fB, the BGP decision process A then prefers to use D as egress router instead of Z.

Figure 3.18: Fibbing changes the forwarding path between B and D by deceasing the cost of the desired path, hence influences the BGP decision process.

session towards AS Y. In AS X, all routers are part of an iBGP fullmesh. Assuming that no particular BGP filters are in use, all routers thus receive two routes towards the prefixes announced by AS Y and prefer the one with the closest nexthop. In this case, A decides to use Z as the IGP cost of [A Y Z] (20) is lower than the one of [A B C D] (24). Similarly, B opts to use D as egress router.

For operational reasons (*e.g.*, scheduled maintenance, traffic engineering), the operator of this network is tasked to divert the transit traffic crossing router C, *i.e.*, to use the path [B X D] instead. Upon reception of this new forwarding requirement, the Fibbing controller then implements it by injecting a globally scoped fake node fB announcing a route towards D, attached to B, as shown on Fig. 3.18a. While successfully steering the transit traffic between B and D, this change may however have additional side effects.

Fibbing controls the BGP egress selection process. Recall that the total fake path cost announced by a fake node is constrained by an upper and a lower bound, whose values ensure that the forwarding requirements are implemented properly. In this example, UB(fB)=14 guarantees that B will

always prefer the path going through fB and LB(fB)=2 is enough to guarantee that no node other than B and A (as B is its original IGP nexthop towards D) uses fB. Note that both of these values only consider the impact of fB in the forwarding DAG towards D. While sufficient to compute internal paths, this is no longer true for transit traffic. Indeed, in the presence of fB, A's BGP decision process compares the relative costs of [A Y Z] and [A B fB] and forwards transit traffic along the path with the lowest cost. Compare two possible fake path costs for fB, shown on Fig. 3.18a and Fig. 3.18b. In both cases, the forwarding requirement is implemented successfully. However, as shown on Fig. 3.18b, selecting the lower bound as fake path cost for fB also causes A's BGP decision process to prefer to use D as egress router instead of Z. In other words, the fake path cost of fB controls both the path of the transit traffic and which ingress nodes will select this path to forward transit traffic.

Fibbing enforces a ranking between egresses. Consider a network G with $\mathcal{N}$ routers, among which there are $\mathcal{E}$ border routers. Let fN_e be a fake node attached to router N and required to implement a forwarding DAG D_e towards a BGP egress $\text{E} \in \mathcal{E}$ (*i.e.,* fN_e announces one of E's addresses). Let $\mathcal{P}_n \subseteq \mathcal{E}$ be the desired set of egresses that N should prefer over E. Computing the fake path cost for fN_e requires to add the following two sets of constraints to the formalization presented in (§3.3.3)[3]

First, we constrain the fake bounds of fN_e according to the desired result of N's BGP decision process. More precisely, any egress router $\text{F} \in \mathcal{P}_n$ constrains the lower bound of fN_e such that the fake cost of fN_e will be higher than the cost to reach F, *i.e.,* $\text{LB}(\text{fN}_e) > dist(\text{N}, \text{F}, G')$. Similarly, any egress $\text{F} \notin \mathcal{P}_n, \text{F} \in \mathcal{N}$ constrains the upper bound of fN_e to ensure that BGP will prefer to use N over F, *i.e.,* $\text{UB}(\text{fN}_e) < dist(\text{N}, \text{F}, G')$. Note that depending on the required forwarding DAG towards F, $dist(\text{N}, \text{I}, G')$ is either a constant value (the path from N to F crosses no node with a fake node), or also depend on the fake cost of another fake node fG attached to router G. In the latter case, we express $dist(\text{N}, \text{F}, G')$ according to the fake cost bounds of fG, *i.e.,* $dist(\text{N}, \text{G}, G) + \text{LB}(\text{fG}) \leq dist(\text{N}, \text{F}, G') \leq dist(\text{N}, \text{G}, G) + \text{UB}(\text{fG})$.

The second set of constraints ensures that fN_e does not introduce unwanted BGP nexthop changes for any other router $\text{R} \in \mathcal{N}$. If R is an ancestor of N in D_e and N is the first node in $sp(\text{R}, \text{E}, D_e)$ that is attached to a fake node announcing E, then its final shortest-path cost towards E depends on the cost of fN_e. The upper and lower bounds of fN_e thus need to take into account the egress preferences of R, namely $dist(\text{R}, \text{N}, G) + \text{LB}(\text{fN}_e) > dist(\text{R}, \text{F}, G')$ for any $\text{F} \in \mathcal{P}_r$ or $dist(\text{R}, \text{N}, G) + \text{UB}(\text{fN}_e) < dist(\text{R}, \text{G}, G')$ for any $\text{G} \notin \mathcal{P}_r, \text{E} \in \mathcal{N}$ Other-

[3] For simplicity, we assume that only one egress can be preferred at a given time, *i.e.,* routers do use the ADD-PATH capability [RFC7911]. The formulation can however be extended to enable "equally-preferred" egresses by ensuring that their final path costs towards different egresses are the same.

wise, as R either has its own fake node or is not an ancestor of N, assuming that the fake cost of fN_e enables to implement D_e, then R's shortest-path towards E is, by definition, unaffected by fN_e. As such, its preferred BGP egress is unchanged, and R thus induces no additional constraint.

On the example of Fig. 3.18a, beside the constraints on the lower and upper bound needed to ensure that [B X D] is used instead of [B C D], applying the above formulation yields the following additional constraints on the fake cost bounds of fB: *(i)* UB(fB) < 25, to ensure that B selects D as egress; and *(i)* 9 + LB(fB) > 20, to ensure that A's BGP decision process keeps using Z over D. Changing this latter constraint to 9 + UB(fB) < 20, enforces that the preferred egress of A should be changed to D, and the result shown in Fig. 3.18b.

Fibbing BGP egresses has some control-plane overhead. While Fibbing is a good fit to alter few paths used for transit traffic and control the preferred egress routers, (ab)using it at large-scale faces three sources of overhead, challenging the scalability of the approach.

1—Controller overhead. When used towards $\mathcal{E}$ BGP egresses, Fibbing, our formulation requires considering at least $\mathcal{E}$ constraints on every node in the network, *i.e.,* the augmentation problem grows quadratically harder in the number of BGP egresses. As fake nodes have constraints recursively linking to other fake nodes across multiple DAGs, simple greedy heuristics like the Merger algorithm are less efficient at scheduling tentative merges. As a result, efficiently reducing the size of the augmented topology imposes to fallback to other approaches with a drastically higher computation cost on the controller, *e.g.,* solving it using ILP formulation is NP-Complete [Coo71].

2—IGP overhead. Beside being hard to compute, merging fake nodes is also less unlikely to happen. Indeed, as fake cost bounds are more constrained, the likelihood of successively merging the same fake node multiple times into its nexthop diminishes. Worse, experimental results showed that complex forwarding requirements tended to only be implementable using locally scoped fake nodes, preventing any optimizations. As such, augmented topologies implementing many requirements for transit traffic tend to be very large, with up to $\mathcal{N} * \mathcal{E}$ fake nodes in the absolute worst case (with $\mathcal{N} \geq \mathcal{E}$ the total number of BGP routers).

3—BGP overhead. Let fN be a fake node attached to a router N, and implementing a path towards a preferred egress router *e*. We distinguish three levels of BGP control-plane overhead induced by fN. First, if fN adds an equal-cost path towards *e*, then it does not result in any BGP control-plane activity. Instead, if fN causes N to see a decrease of its IGP cost towards *e* but *e* was already the preferred BGP egress, then fN causes BGP UPDATE messages to be sent over N's eBGP session, if any, reflecting the changes in the MED attribute. Finally, if fN changes the BGP nexthop of N, then UPDATE messages are sent

over all N's eBGP and iBGP sessions.

3.6 Discussion

This section covers qualitative aspects of Fibbing that go beyond the raw capabilities presented in previous sections. More specifically, we first highlight how Fibbing is naturally suited to drive incremental SDN deployment in existing networks (§3.6.1), then we discuss several high-level concerns raised by the community about the practicality of Fibbing (§3.6.2).

3.6.1 Fibbing enables incremental SDN deployment

The advent of SDN [FRZ13] makes it clear that network operators want their networks to be more programmable and easier to manage centrally. However, very few of them are able to deploy SDN using a clean-slate approach [Jai+13; Hon+13], notably due to the prohibitive cost of such approach. Instead, most networks rely on an incremental SDN deployment, where subset of devices will be upgraded at a time, gradually increasing SDN capabilities over time.

From a high-level viewpoint, Fibbing provides an API to program paths in an underlying IGP, *i.e.,* FIB entries, from a logically centralized controller. As such, Fibbing enables to control existing networks as if they were using fully-fledged SDN protocols. Consequently, Fibbing is a prime candidate to drive incremental SDN deployment in existing networks. More precisely, according to the classification presented in [VVB14], we distinguish two types of hybrid SDN models in which Fibbing naturally fits.

1—Integrated hybrid SDN. Fibbing lets operators plan the deployment of SDN in their existing network in two main steps. First, operators should update their control-plane in order to ensure that the IGP topology is Fibbing-compliant (§3.2.2). Then, management tools in the network can be upgraded to use the Fibbing controller instead of directly reconfiguring routers. Note that once this step is complete, the network can be fully operated as a SDN, keeping in mind the inherent limitations of the IGP, such as destination-based forwarding only, or the impossibility to express packet rewriting rules. From then on, operators can then start to progressively upgrade network devices to SDN-capable ones, transitioning the network to a service-based hybrid SDN.

Note that due to the programming capabilities it enables, incrementally deploying SDN using Fibbing maximizes the efficiency of the non-upgraded routers. This is in stark contrast to classical deployment strategies that treat the IGP routers as uncontrollable devices [Lev+14; Pou+17; Xu+17]

2—Service-based hybrid SDN. Such a network is composed of a mix of pure IGP routers, and SDN-enabled ones. In such a network, Fibbing can then

be used by the SDN controller as an API to control the non-upgraded routers, improving the overall flexibility of the network. Additionally, carefully selecting the location of the SDN devices is key to maximize their utility. For example, placing such SDN devices at the edges of the network would enable a controller to use them to classify the traffic and then encapsulate it, increasing the expressiveness of the Fibbing controller as this encapsulation would enable it to side-step the limitations of IGPs.

3.6.2 Practical considerations

Past discussions about Fibbing with researchers and operators often brought up a similar set of high-level concerns regarding the practicality of Fibbing, and its suitability as SDN. We hereafter qualitatively describe key points addressing such concerns.

Fibbing is a long-term solution. Beside enabling a low-cost incremental deployment, Fibbing enables a flexible and scalable network architecture (§3.6.1). Indeed, major industry [T K14] and academic actors [Cas+12; She14] advocate for an architecture where fine-grained SDN functionality is deployed at the edge, and solutions like Fibbing in the core. More importantly, as it combines the best of centralized control (manageability, flexibility) and distributed routing (resiliency, scalability), we believe that Fibbing fits the needs of the network core better than other approaches. Finally, even in pure-SDN networks, Fibbing can still flank the SDN control-plane with an IGP in order to provide a fast and scalable failure recovery mechanism [TV14], providing some backup policy guarantees.

Fibbing keeps operators' ability to troubleshoot networks intact. Fibbing leverages "*tried and true*" protocols, which carries several advantages. First, the implementations of IGPs on routers are mature and their eventual bugs are well-documented by router vendors. Second, as these protocols are well-known among operators, troubleshooting them fits their mental model, easing up the task. Additionally, as Fibbing relies on standard IGP messages (which can always be traced back as created by the controller (§4.1)), it is compatible with existing management, monitoring, and debugging tools. Moreover, Fibbing leverages simple IGP configurations (§3.3.3) (*i.e.,* links and weights), which are designed to be understood by humans. Finally, the Fibbing controller can be used as an additional source of truth to ease up debugging, for example to map injected fake nodes to the requirements they implement.

Fibbing simplifies the control-plane. On one hand, enabling to program arbitrary paths on a per-destination basis, Fibbing enables to simplify the protocol stack used in today's networks (*e.g.,* replaces RSVP-TE). Beside reducing operational complexity, this also reduces the risk of human errors

when configuring devices. On the other hand, predicting the result of a particular IGP message (*i.e.,* fake node) on the routers' FIB is simple. Additionally, as computationally complex tasks such as path computation, topology maintenance and installation of FIB entries are offloaded to the routers, the design of the Fibbing controller (§4.2) is significantly simpler than for other SDN protocols (*e.g.,* [ONOS; Kop+10; Floodlight; Fos+13]).

Fibbing has no impact on security. Fibbing is *not an attack* on the IGP (as opposed to [Nak+13]). The fake nodes injected by the controller are part of the IGP topology, and should be accepted by routers without causing stability issues (§4.1). Additionally, IGPs traditionally support the authentication of their control-plane message (*e.g.,* using MD5-based authentication [RFC2328; Cisb; Juna]), which can also be supported by the controller.

Fibbing partially supports middleboxes chaining. On its own, Fibbing is unable to program paths that loop over themselves, and as such would not be able to implement chains of middleboxes requiring such loops [Qaz+13; Fay+14]. Instead, implementing such chains with Fibbing requires some support from routers to break the loops in sub-paths (*e.g.,* by swapping encapsulation, leveraging policy-based routing to match on the input interface in addition to the destination IP address [Cise; Junb]), which can be provisioned centrally using BGP Flowspec [RFC5575; Cisf; Junc], or leveraging SDN-enabled nodes in a service-based hybrid SDN (§3.6.1).

3.7 Related Work

Fibbing contributes to the larger debate opposing fully distributed routing protocols to SDN protocols. More specifically, Fibbing performs a tradeoff between the centralization of routing decisions and the distribution of route computation. We can compare Fibbing to three main categories of earlier work towards achieving a centralized control over the behavior of the network.

Centralized control through routing protocols (re)configurations. As the individual router configurations define the overall behavior of a distributed routing protocol, managing these configurations from a centralized system enables to efficiently implement global requirements.

In [FT00; FRT02], such a system is used to implement complex traffic engineering policies in IGPs by performing a global optimization of link weights in order to approximate a solution to the Multi-Commodity Flow (MCF) problem. Comparatively, Fibbing is more general, as beside traffic engineering, it also enables to implement *any* forwarding paths on a per-destination basis, and this support more use-cases than pure TE. Furthermore, as this extra

flexibility controls the fractional splitting ratios used to perform uneven load-balancing, this enables Fibbing to implement a (near) optimal solution to the MCF problem [NKR13; SGD05].

More recently, Propane [Bec+16] provides a high-level language to express global objectives on the expected traffic flows in backbone networks or datacenters, automatically synthesized to BGP configurations. Compared to Fibbing, Propane suffers from three limitations: *(i)* while more expressive, its compilation process is order of magnitude slower, hurting its ability to react to online events; *(ii)* Propane relies on an external protocol (*e.g.,* NETCONF or, YANG) to actually push these configuration files on a per-router basis, causing a scalability bottleneck on the controller as it needs to configure each device individually; and *(iii)* Propane produces configurations that require operators to "trust" the compiler as they cannot be understood as-is, impacting their ability to troubleshoot the network.

Centralized control by leveraging routing protocols messages. The Routing Control Platform (RCP) [Cae+05] is a logically-centralized platform that uses BGP to install forwarding entries into routers. More precisely, a controller establishes an iBGP session with all routers in the AS, through which it then receives routes from other ASes as well as send routes that will be the ones preferred by the routers. Similarly, [Ver+07] presents the Intelligent Route Service Control Point (IRSCP), which improves RCP by enabling to control a ranking between egresses for each router, at scale. Both of these approaches require the controller to program the routes on all routers. Furthermore, routers completely rely on the RCP controller to be able to maintain up-to-date forwarding tables. In contrast, Fibbing can adapt the forwarding behavior of many routers at once (*i.e.,* using a globally scoped fake node). Furthermore, as Fibbing lets routers compute on their own the content of their FIB, the failure of the controller is comparatively less impactful than in RCP, as routers always can fallback to the default IGP paths.

Another approach to tweak the IGP topology is presented in [Car+15], where SDN switches are placed at specific locations in order to partition the IGP topology. As a result, these switches are then able to control the flooding procedure of the IGP, selectively isolating parts of the network and controlling route redistribution between the partition. Omitting practical considerations that would restrict the set of possible locations for those switches, Fibbing goes beyond this approach as it can change any path in the network (whereas the internal paths in the partitions created by [Car+15] are never changed). Finally, we argue that partitioning a network and actively controlling the redistribution across partitions poses inherent resiliency challenges, which are not applicable to Fibbing.

Centralized control by programming forwarding tables.
The Software-Defined Networking paradigm revolves around a centralized controller, directly programming entries in the routers' forwarding tables, possibly reacting to the reception of specific packets. Several protocols exists to that end (*e.g.,* Openflow [McK+08], P4 [Bos+14], I2RS [RFC7921]), with different set of tradeoffs (*e.g.,* targeting legacy hardware, supporting statefull processing logic). While more flexible than Fibbing, these protocols suffer from the same set of drawbacks, namely scarifying the scalability and resiliency of distributed protocols by putting the SDN controller in charge of maintaining all FIB entries on every router in the network.

Segment Routing (SR) [RFC8402] revisits the source-routing paradigm by encapsulating packets with an ordered list of nodes that should be crossed before the packet is delivered to its destination. It relies on a centralized controller to program ingress routers such that they properly classify ingress traffic and apply the proper encapsulation header. While both SR and Fibbing rely on an underlying IGP to forward the traffic, Fibbing lowers its data-plane overhead by programming paths directly in the control-plane. We argue that both approaches are complementary, as SR coupled with its Network Framework [Fil+18] perfectly fits the fined-grained SDN capabilities at the network edge presented in (§3.6.2).

Finally, Fibbing takes inspirations from Merlin [Sou+14] to capture operators' forwarding requirements. The mechanism satisfying them is, however, entirely different. We stress that our main contributions are the Fibbing techniques and algorithms, while the language only serves to encode forwarding DAG in a human-friendly way.

Fibbing real networks | 4

We now present an implementation of Fibbing in a *real IGP* (OSPF), compatible with Commercial Off-The-Shelf (COTS) routers. More precisely, we begin by specifying how fake nodes and links can be represented (§4.1) in OSPF, leveraging its *protocol specification* [RFC2328]. Then, we present the *key components* that are required to design and implement a Fibbing controller (§4.2) for OSPF. We confirm the scalability of the approach through *measurements* performed on real routers showing that Fibbing imposes almost no overhead (§4.3). Finally, we *discuss* the impact of failures (§4.4), and demonstrate that Fibbing can perform *real-time traffic engineering* in a case study (§4.5).

4.1 Fibbing OSPF networks

Open Shortest-Path First (OSPF) is a link-state IGP explicitly designed to operate in IP networks. Being almost 30 years old, its specifications have been through three major versions [RFC1131; RFC2328; RFC5340] and are completed by over 50 protocol extensions. Leveraging these specifications, we describe in this section how Fibbing can be implemented in a vanilla OSPF network. More precisely, we begin by introducing the core features that enable OSPF to establish a synchronized view of the network state across all routers (§4.1.1). Then, we show how Fibbing concepts such as fake nodes and fake links can be implemented according to the protocol specification (§4.1.2). Finally, we discuss the limitation of our approach and possible protocol extensions that would improve OSPF's support for Fibbing (§4.1.3).

Unless specified otherwise, this section focuses on OSPFv2 [RFC2328], instead of OSPFv3 [RFC5340]. While OSPFv3 is the latest version, OSPFv2 has the most mature open-source implementations [Quagga; BIRD], is widely deployed, and is still actively developed, *i.e.,* new protocol extensions such as Segment Routing are developed for both OSPFv2 and OSPFv3. Furthermore, all protocol features used to implement Fibbing in OSPFv2 networks are also found in OSPFv3.

4.1.1 Building a shared view of the network with OSPF

Two primitives are at the core of OSPF to build its link-state database, namely an adjacency detection mechanism to infer the state of a routers' links, and the flooding procedure to share this state with all other routers, synchronizing their link-state databases. We now describe both primitives, and then present key type of information found in a link-state database and how these can be used to build a view of the network topology.

Adjacency detection. In OSPF, routers send HELLO PDUs towards an IP multicast address over all their links[1]. This process enables any OSPF router to detect the presence of other OSPF routers connected on the same layer-2 domain. If multiple routers are connected over a broadcast domain (*e.g.,* ethernet), then one of them will be elected as *Designated Router* (DR), which is the sole router that will flood status information about that link, reducing control-plane churn. Beside automatically establishing full adjacencies (over which the link-state databases can then be synchronized), this mechanism also provides a failure detection mechanism. Indeed, if a router A stops receiving HELLO PDUs from one of its neighbor N for an extended amount of time (*e.g.,* three times the `hello-interval`), it can then assume either that N has failed, or that a link between A and N is down.

Flooding procedure. Flooding ensures all routers share the same view of the network by having each router broadcast its local state over each of its adjacencies, and also relay the state it learned from one of its neighbors to its other ones. OSPF names such pieces of the global network state "Link State Advertisements" (LSAs). All types of LSAs share a common header, visible in the first half of the PDU depicted in Fig. 4.1, whose main fields are:

- **Age.** A measure of the time (in seconds) since the LSA was originated. When this value reaches `max-age` (usually 3600s), the LSA is then evicted from the link-state database;
- **Type.** The type of information carried by the LSA;
- **Link ID.** The portion of the AS described by the LSA, its exact semantic depends on the *Type* field;
- **Advertizing router.** The OSPF `router-id` of the router that originated the LSA, *i.e.,* it is not changed when the LSA is flooded by other routers.
- **Sequence number.** Successive instances of a LSA sent by the same router, of the same type, and having the same link ID, are given successive sequence number. Routers only use the newest version of a LSA computing the content of their RIB.

[1] In IPv4, `224.0.0.5` is reserved for *all OSPF routers* and `224.0.0.6` for *all Designated Routers*.

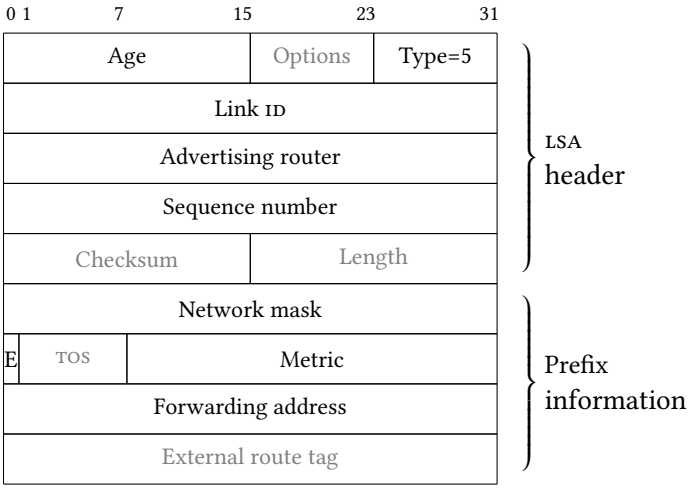


Figure 4.1: In OSPFv2, AS-external LSAs announce an IPv4 prefix reachable through the IGP nexthop specified by the Forwarding address field.

Whenever a router receives a LSA, it first checks in its link-state database whether it already had the same one (*i.e.,* same advertizing router, type and link ID). If that's the case, and if the sequence number of the received LSA is older than the one in its database, it then drops it. Otherwise, this LSA is either a brand-new one or a newer version, and it is installed in the router's link-state database before being flooded to its neighbors.

The link-state database. The network state is composed of different sets of information, *e.g.,* the routers present in the topology, the status of their links and adjacencies, or the prefixes reachable using the IGP. Therefore, OSPF defines several types of information that can be flooded by routers. To implement Fibbing in OSPF networks, we will use three types of LSAs:

- *Type=1:* A **Router LSA** announces the presence of a given router in the IGP topology. Additionally, it also lists all stub networks directly connected to the router (*i.e.,* layer-2 domains without any other router), point-to-point adjacencies, and interface connected to a broadcast layer-2 domain, each of them with a particular cost (*i.e.,* directed link weight). In Fig. 4.2, A, B, and C each originate one such LSA.
- *Type=2:* **Network LSAs** list all routers that have established adjacencies over a broadcast link, as well as the IP prefix of that link. It is only generated by the Designated Router of the link. In Fig. 4.2, assuming that A was elected as DR for the link (A, B), A will then generate a network LSA containing the prefix `x.y.z.0/p`, and listing both A and B as being connected to that broadcast domain.

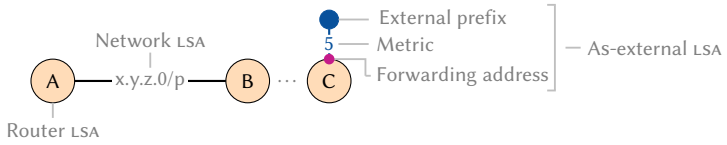


Figure 4.2: Each type of LSA represents a specific element of the network.

- *Type=5:* AS-external LSAs provide reachability information towards prefixes coming from another protocol (*e.g.,* a default route coming from a BGP border router). Fig. 4.1 shows the layout of such LSA, which we illustrate based on the example shown on the right of Fig. 4.2. OSPFv2 follows classful addressing conventions, hence encode the external prefix host bits (*i.e.,* "blue") in the *link ID* while the actual prefix length is encoded as a *network mask* (*e.g.,* `255.255.0.0` to denote a `/16`). The *forwarding address* (FA) field designates a router towards which traffic towards the blue prefix should be forwarded (*i.e.,* c). When creating a route r from this LSA, routers will first query their OSPF RIB to identify a route r' able to forward LSA's FA, then their FIB as a fallback. If no such r' is found, then the LSA is ignored. Otherwise, the nexthop of r is set to the nexthop of r'. Finally, the *metric* field (5) denotes the cost of the route. If the *E* bit is unset (E=0), this cost is absolute. If instead the E bit is set (E=1), then the cost of the route is the sum of the IGP shortest-path cost towards the FA and of the metric field.

Routers construct an up-to-date view of the network graph in two steps. First, they add one node for each router LSA. Second, they add one edge per point-to-point link between routers, and convert network LSAs to a full mesh of edges between the routers they contain. Finally, to derive the content of their OSPF RIB, routers then iterate over all LSAs and compute the total metric to reach the prefix(es) they contain (*e.g.,* the shortest-path cost to reach their advertizing router for router and network LSAs). Then, they apply the OSPF decision process. More precisely, if multiple LSAs announce the same prefix, the router selects its best route by first preferring those corresponding to internal routes (*i.e.,* created from router and network LSAs) to those corresponding to external routes (*i.e.,* AS-external LSAs, and other types not mentioned here)[2], then finally selecting those with the smallest metric.

Note that in the network graph computed by OSPF routers, links are OSPF adjacencies, *i.e.,* they may in reality map to a path made of multiple layer-2 links and switches. Additionally, as links cost are announced independently by each router, on a per interface basis, edges are always directed. For clarity

[2]Strictly-speaking, routers also differentiate between the metric types when comparing AS-external LSAs, preferring those whose E bit is unset.

in the figures, we represent with undirected edges links whose costs are the same in both direction (this has always been the case so far except for the fake shortcut shown in Fig. 3.10b).

4.1.2 Injecting fake nodes and links in OSPF

Implementing an augmented topology in an OSPF network, requires to specify a mapping between concepts specific to Fibbing (*e.g.,* fake nodes) and features supported by the message used by OSPF to distribute topology information, *i.e.,* LSAs. This mapping comes with two constraints.

First, to ensure stability of augmented topologies, the LSAs injected by Fibbing must not be considered as an attack or misconfiguration by the OSPF routers. Indeed, trying to impersonate routers to overwrite link-state database entries will result in the impersonated router "fighting back" (*i.e.,* sending newer LSAs to overwrite our injected LSAs) or potentially even erase its routing table, turning it into a blackhole [Nak+13]. As a result, the Fibbing controller must behave as a normal OSPF router, hence must possess its own `router-id` that will be set as advertizing router in the LSAs it injects, and establish at least one OSPF adjacency to connect to the IGP.

Second, to ensure that Fibbing works with current, unmodified, OSPF routers, our controller must only use fields found in standard LSAs, and rely on their specifications (*i.e.,* respect their specific semantic, as well as the general OSPF decision process). More importantly, this precludes the use of Opaque-LSAs (or their newer TLV variants [RFC7684]) which are the traditional way to deploy protocol extension. Indeed, these, by definition, require ad-hoc support on the routers and are thus not backward-compatible.

Accounting for the above two constraints, we implement Fibbing in OSPF by leveraging the *forwarding address* (FA) found in AS-external LSAs (see §4.1.1). More precisely, recall that the semantic of that field enables a router to announce a route towards a given prefix, mapped to a given IP nexthop, with a total cost depending on its location in the topology if its E bit is set. In other words, AS-external LSAs enable to distribute arbitrary routes whose nexthops are recursively resolved. This type of LSA thus corresponds to the mapping information (§2) needed to translate a fake node to a real nexthop. We experimentally confirmed that our use of AS-external OSPF LSAs to inject Fibbing fake nodes and links in a topology was working using physical Juniper and Cisco routers. We now describe how to use AS-external LSAs to implement the primitives of Fibbing, enabling to deploy augmented topologies in existing networks.

Implementing globally scoped fake nodes. Let G' be an augmented topology containing a globally scoped fake node f_R. Assume that f_R is attached to a real router R, has a total fake path cost of c, announces a prefix p, and

has mapping information such that N should be used as real nexthop when R forwards traffic towards p. A Fibbing controller can then implement this fake node using an AS-external LSA with the following properties *(i)* p's host bits and prefix length are encoded as link ID and host mask; *(ii)* the forwarding address is set to an IP address of N belonging to an interface connected to a broadcast domain, hence part of a prefix announced by a network LSA[3]; and *(iii)* the E bit is set to 1, and the metric is then set to $c - dist(\text{R}, \text{N}, G')$.

The controller then originates this LSA, flooding it through the OSPF domain to ensure its presence in every routers' link-state database, *i.e.,* be globally visible and affect all desired RIBs (at least the one of R). Upon receiving this crafted LSA, N will re-flood it to its neighbors and install it in its link-state database as it has an advertizing router different from himself. However, when computing its route towards p, it will then ignore this LSA as resolving up the forwarding address will not yield a nexthop.

Implementing locally scoped fake nodes. On the wire, we implement locally scoped fake nodes exactly as the globally scoped ones. Restricting their use to only the target node is, however, more challenging. To guarantee the consistency of link-state databases across routers, OSPF forbids from excluding specific LSAs from the flooding process, *i.e.,* over a given adjacency, all known LSAs are either flooded or none at all are. Consequently, a locally scoped fake node implemented by a LSA will always be in every router's link-state database. Fortunately, despite this limitation, locally scoped fake nodes can still be implemented.

Recall that an AS-external LSA can only be used if its forwarding address yields a routing table entry. We exploit this in two steps. First, during the initial re-configuration of the network to make it Fibbing-compliant (§3.2.2), we subnet an unused IP prefix P such that every link connected to a broadcast domain has an additional IP prefix. This enables router interfaces in the broadcast domain to receive additional addresses from that subnet, which we call private addresses. In parallel, we also filter the routes in the OSPF RIB (*i.e.,* using a `distribute-list`) to deny the installation of any route for P or its subnets in the OSPF RIB [Cish; June] (§1.1). Second, when injecting a locally scoped fake node whose goal is to set the nexthop of a router R to N, we then use as forwarding address an IP address of N in its private subnet over the link (R, N). On one hand, due to the static route filter configured earlier, no router in the network will have a route in its OSPF RIB able to resolve the FA of the newly

[3]This restriction comes from conditions in the OSPF specifications determining the validity of a forwarding address, as well as additional restrictions applied by vendors [Cisg]. This thus directly prevents from using a router's loopback address as forwarding address as it will never be part of a broadcast domain hence included in a network LSA. As such, controlling the paths followed by tunnels (as presented in §3.5.1) requires to establish them using IP addresses belonging to physical interfaces.

injected LSA. On the other hand, as R is directly connected to N, it will then have an entry in its FIB for the private address, *i.e.,* it has an additional source of knowledge about that particular subnet of P. Consequently, no router but R will be able to use the LSA corresponding to the locally scoped fake node, despite it being present in every link-state database.

For example, assume that we defined the private address range dedicated to Fibbing as `10.0.0.0/24`, and that we allocated `10.0.0.0/28` for the link (A, B) on Fig. 4.2, thus enabling routers A and B to each receive 4 additional addresses[4]. Assuming that A's private addresses belong to `10.0.0.0/29`, if a Fibbing controller then injects an AS-external LSA whose forwarding address is `10.0.0.1`, then only B will be able to use it (using its directly connected route) as: *(i)* A will ignore it as it is one of its own address; and *(ii)* C will not have a route towards it as `10.0.0.0/28` will be filtered out from OSPF, and as C is not directly connected to the broadcast domain formed by (A, B).

Enabling cross-destination optimizations. The first type of cross-destination optimization (§3.2.4) merges multiple fake nodes into a single one, announcing all their destinations at once. This is impossible to achieve in OSPF, as there are no ways to insert multiple destination prefixes in a single AS-external LSA[5]. Note that we can nevertheless limit some control-plane overhead when injecting several LSAs by batching up to 40 of them in a single OSPF LS-update message sent to the controller's neighbor (§4.3.2).

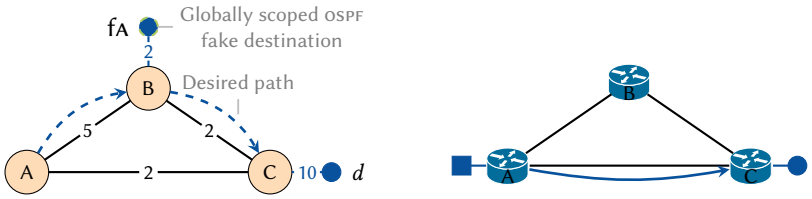
The second type of cross-destination optimization creates fake shortcuts, *i.e.,* overrides the metric of a link in a given direction. Such shortcut cannot be implemented by injecting a LSA. Indeed, this would require the Fibbing controller to impersonate a router to send a crafted router LSA with a modified link metric, hence triggering the fight back mechanism of the impersonated router, thus creating instability in the network. Instead, we argue that such feature should be implemented by enabling the Fibbing controller to remotely reconfigure the interface metric of the target OSPF router (*e.g.,* using NETCONF [RFC6241]), eventually prompting it to renew its own router LSA with the updated metric.

4.1.3 Limitations and possible IGP extensions solving them

Our approach to implement Fibbing with OSPF has two major limitations.

[4] $(2^{32-28}) - 2$, as IPv4 requires to reserve the network and broadcast subnet addresses.

[5] A feature of OSPFv2, removed in OSPFv3, enables to specify different host mask values, depending on the TOS associated with the route. Regardless of the fact that announcements would then be scoped by TOS values, it does not enable to change the host bits of a prefix, *i.e.,* it would enable to announce routes towards `a.b.c.0/24` and `a.b.0.0/16` in the same LSA, but not `a.b.c.0/24` and `x.y.z.0/24` hence be too restrictive for the cross-destination optimization algorithm.



(a) Forwarding requirements and augmented topology implementing them.

(b) Requirements violation due to an unwanted deflection.

Figure 4.3: This requirement cannot be implement in OSPF with a globally scoped fake node, as A uses it by forwarding traffic on its shortest-path to B.

1. Prefix compatibility. The OSPF decision process heavily restricts the set of prefixes that can be re-routed using Fibbing. More precisely, Fibbing can only alter paths towards prefixes that were announced in the IGP using AS-external LSAs in which the E bit was set. Consequently, this prevents us from using Fibbing to alter the path towards prefixes assigned over a link between two routers (*i.e.,* network LSAs), or to stub networks (that were announced in router LSAs).

A work-around consists in configuring every router such that it redistributes all its directly connected routes in OSPF using AS-external LSAs (*i.e.,* `redistribute connected <metric>`). While this enables Fibbing to control all prefixes announced in OSPF using router and network LSAs, this comes at the expense of an increased link-state database size.

2. Globally scoped fake nodes. Due to the semantic of the forwarding address field, globally scoped fake nodes in an augmented topology are translated to fake destination announcements in OSPF. In essence, this *merges* the fake node into its associated nexthop. This has two consequences.

First, this imposes to adapt the formulation of the constraints on a fake node's cost bounds in the Merger algorithm (§3.3.3). Consider the fake OSPF destination attached to the router B in Fig. 4.3a, which is needed to change A's nexthop towards the blue prefix. To guarantee that A will prefer to use the fake destination instead of d, the constraint on its upper bound thus needs to be changed to $\text{UB}(\text{fA}) < dist(\text{A}, d, G) - dist(\text{A}, \text{B}, G)$. Similarly, the lower bound constraints inequalities become $\text{LB}(\text{fA}) > \text{LB}(\text{fN}) - dist(\text{N}, \text{B}, G)$, for any other node N using a fake node fN (note that the constraint now depends on $dist(\text{N}, \text{B}, G)$ instead of $dist(\text{N}, \text{A}, G)$, as the fake destination is attached to B).

Second, the location of these fake destination announcement (*i.e.,* directly attached to the target nexthop) violates Theorem 1. Indeed, OSPF first computes paths between routers, and then creates routes for prefixes by mapping them to their advertizing router and using its previously computed nex-

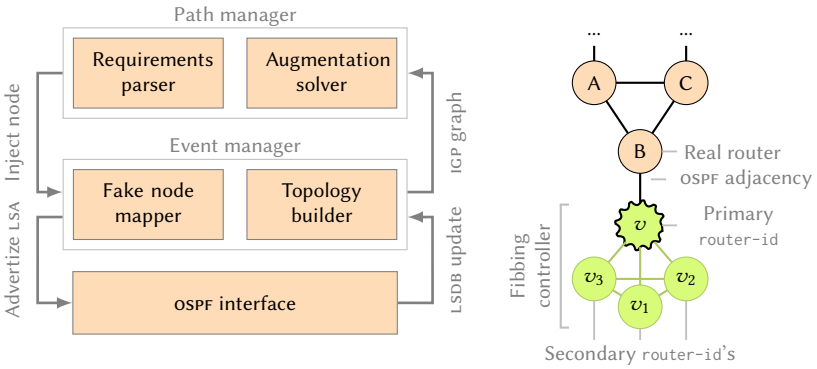
thop. Consequently, to identify the route for the forwarding address of fA in Fig. 4.3a, router A will query its OSPF RIB, and eventually find a route towards router B. However, as it was created by OSPF, this route follows the shortest-path from A to B, *i.e.,* uses C, which violates the requirement as shown on Fig. 4.3b. More generally, this forwarding requirement is impossible to implement using globally scoped OSPF fake destinations. Note that this issue does not apply to locally scoped fake destinations. Indeed, as these use forwarding addresses that are filtered out of the OSPF RIB, and resolved using directly connected routes present in the FIB, these are thus bound to use specific egress interfaces as nexthop. As such, we overcome the limitations of globally scoped fake destinations with locally scoped ones when needed.

Extending IGPs to cleanly support Fibbing. Our implementation of Fibbing in OSPF relies on the presence of forwarding addresses in the protocol specifications. Unfortunately, other link-state IGPs, such as IS-IS [RFC1142], have no equivalent mechanism. Enabling to use Fibbing in other protocols, or overcoming the limitations of the current OSPF implementation, thus requires to design a protocol extension. Enabling fully-fledged Fibbing would require two primitives from such extension: *(i)* creating virtual nodes, able to announce destination prefixes at specific metrics with specific "priority" in the IGP decision process; and *(ii)* a mechanism to create directed links from specific egress interface of real routers, towards such virtual nodes. Support for these functions can be added to protocol specifications without impacting current functionalities (*e.g.,* using the dedicated extension mechanisms such as IS-IS TLVs), and deployed through router software updates. Backward compatibility with non-upgraded routers is easily achieved as those routers would: *(i)* not announce their support for such extension; and *(ii)* ignore the unsupported LSAs, while still flooding them throughout the IGP. Our placement algorithms can account for them by considering that their nexthops will never be changed by any fake node.

4.2 Implementing a Fibbing controller

We built a complete prototype controller, that interfaces with the algorithms implemented and evaluated earlier (§3.4), able to implement forwarding requirements in real OSPF networks. The core logic of our controller, as well as its high-level interfaces are written in about 2200 lines of Python (comments included). Additionally, we extended Quagga [Quagga] with about 400 lines of C code, enabling our controller to interact with existing link-state routing protocols and thus COTS routers.

This section first presents the general architecture of our prototype (§4.2.1). Then, we describe how Fibbing can be made resilient against a controller fail-



(a) Our Fibbing controller exposes a high-level path manager framework to applications.

(b) The controller is perceived as a cluster of routers in the OSPF topology

Figure 4.4: Through its OSPF adjacency with a router, our Fibbing discovers the network topology and injects LSAs to implement forwarding requirements.

ure by distributing it across multiple replicas (§4.2.2).

4.2.1 Architecture of our Fibbing controller

Our prototype controller is composed of three main components, layered on top of each other and shown on Fig. 4.4a. As injecting a fake node requires our controller to participate as a router in the OSPF topology, these components act as interfaces between high-level Fibbing requirements and the underlying IGP. We now describe in more details each of these components.

Path manager. This is akin to the northbound interface of traditional SDN controllers. The path manager is a Python framework enabling applications to express forwarding requirements that should be implemented in the network. More specifically, we provide a *parser* that translates requirements expressed in the language presented in (§3.1.1) into forwarding DAGs. These DAGs can then be fed as input to an *augmentation solver*, which executes either the Merger or the Simple implementation (§3.4) to compute the set of required fake nodes and links. This set of fake elements can then be compared to those already injected in the network, leading to requests to the lower layers of the controller to add or remove particular fake nodes.

This path manager receives the complete IGP topology from the lower layers of the controller, and is notified whenever the network topology changes. This enables applications leveraging the path manager to define custom reactions to network events. Possible reactions could be to leverage *delta-databases* to react to a pre-computed network failure and ensure a *fast recovery*, to

change the set of forwarding requirements depending on the network topology, or even simply to raise an alarm in the network management system.

Event manager. The event manager in our controller is written in Python. It exposes a JSON-RPC interface, enabling the path manager to trigger the injection or the removal of a fake node, as well as to receive network topology changes. Two components are in charge of these tasks.

First, a *fake node mapper* translates fake nodes and links received from the path manager to their equivalent OSPF LSAs. More precisely, it first selects the forwarding address that matches both the fake node scope and the desired nexthop, and it then selects a suitable OSPF `router-id` for the LSAs. Indeed, OSPFv2 forbids a router from advertising the same prefix in multiple AS-external LSAs, viewing them as duplicate and ignoring all of them but the most recent one[6]. Supporting complex requirements that need multiple fake nodes (*e.g.,* see the motivational example in Fig. 2.2a) thus requires our controller to advertize the successive LSAs corresponding to these fake nodes, using the same prefix but different `router-id`'s. Finally, the fake node mapper translates the fake path cost to the desired metric (§4.1.2), and sets the age of the LSA. We *inject* a fake node in an underlying topology by using a low age value. Conversely, the fake node mapper *removes* a fake node by re-advertizing it with an age set to `max-age` (*i.e.,* 3600), effectively flushing it from every routers' link-state database (LSDB) (§4.1.1).

The second component of the event manager is the *topology builder*. This component watches for changes in the LSDB maintained by the OSPF interface from which it can build an up to date view of the network topology. Indeed, the flooding procedure of OSPF guarantees that every router will eventually have the latest version of the LSAs originated by all other routers in the network in its LSDB. Analyzing its LSAs (§4.1.1) is thus sufficient to derive the current network state, and detect changes that should be sent to the path manager. In other words, relying on the IGP enables our Fibbing controller to have built-in topology discovery and failure detection mechanisms.

OSPF interface. This component leverages a modified version of Quagga to establish an OSPF adjacency with at least one router in the network. Using this adjacency, our controller then joins the flooding domain, receiving and storing in its LSDB a copy of every LSA.

To support the multiple `router-id`'s required by the fake node mapper, the OSPF interface exposes the controller to the OSPF network as a small cluster of virtual routers, visible on Fig. 4.4b. To that end, it originates during its initialization one router LSA per desired `router-id`. Then, it minimizes the overall number of adjacencies having to be established with the real routers, by link-

[6]Note that this is no longer the case in OSPFv3, which would then simplify this part of our controller implementation.

ing all virtual routers in a full mesh, described by a single network LSA, and finally establishing a single adjacency between one of the virtual routers and the physical neighbor of the controller. This adjacency defines the *primary* `router-id` of our controller. Once completed, this bootstrapping phase enables our controller to inject multiple LSAs announcing the same prefix, using these provisioned `router-id`'s. Note that the number of virtual routers can be easily adjusted at runtime. For example, the controller shown in Fig. 4.4b is able to inject up to four fake nodes announcing the same destination prefix. Assuming that this controller receives a forwarding requirement needing five fake nodes to be implemented, it would then first allocate a new virtual router (*i.e.,* generate a new router LSA and update the network LSA), and then would inject the LSA corresponding to the fifth fake node using the newly provisioned *secondary* `router-id`.

4.2.2 Distributing the controller

The Fibbing controller introduces a single point of failure in a network. To protect against such failure (*e.g.,* crashes due to software bugs, hardware failures), our controller can be distributed across multiple nodes.

More precisely, we run multiple copies of our controller software, at different locations in the network. Leveraging the underlying IGP, no state needs to be synchronized between the controller replicas besides the input forwarding requirements to achieve an eventually consistent distributed system. Indeed, all our algorithms are deterministic. As the network topology across all controller replicas is eventually consistent thanks to the flooding procedure, these replicas thus all compute exactly the same augmented topology.

A naive approach would leverage this eventual consistency by having each replicas always implementing every forwarding requirement. Beside introducing some overhead during the flooding procedure as more LSAs would be originated and eventually discarded, this would also introduce small instabilities during network convergence as the replicas might become slightly out-of-sync.

Instead, we limit control-plane overhead using a simple primary-backup architecture, where only one replica actively sends LSAs, coupled with an inexpensive leader election process. Recall that OSPF defines a `router-id` as being an opaque 32bits identifier. We statically allocate a range of `router-id`'s that can be used as primary `router-id` of Fibbing controllers, such as each replica can receive a unique primary `router-id`, and another range for secondary `router-id`'s, shared across replicas. Leveraging this, a replica can then discover all the other ones by inspecting its network graph, looking for `router-id`'s matching the reserved range of primary `router-id`'s. If multiple replicas are present at the same time, we define the leader as the one with the small-

est primary `router-id`. If the controller elected as leader were to fail, its OSPF neighbors would then eventually detect it and flood the information in the network. This would result in every other replicas learning about the failure, eventually electing a new leader which would resume implementing the forwarding requirements.

4.3 Fibbing OSPF scales

To implement its requirements, a Fibbing controller injects crafted LSAs in the OSPF network. As complex requirements may require a large number of such LSAs, this section evaluates two possible bottlenecks that would hinder Fibbing's scalability. First, as these LSAs increase the size of every routers' LSDB, we measure (§4.3.1) on real routers the CPU and memory usage overhead induced by Fibbing, and its impact on OSPF convergence speed. Second, we evaluate the performance of our controller (§4.3.2) when it needs to inject large numbers of LSAs (*e.g.,* when reacting to failures).

Our experiments demonstrate that Fibbing scales, as it has a negligible impact on router's performance, and as our controller can trigger large topology updates in milliseconds.

4.3.1 Fibbing has no practical overhead on real routers.

We performed our measurements on two different routers: *(i)* a recent Cisco ASR9K running IOS XR v5.2.2, equipped with 12GB of DRAM assigned to the routing engine and Typhoon-based linecards; and *(ii)* a 7 years old (at the time) Juniper M120 running JunOS v9.2, equipped with 2GB of DRAM. Being aggregation routers, these devices are representative of edge devices commonly found in service provider networks. Both routers exhibited similar performance trends, and we report hereafter the ones collected on the Cisco device.

Fibbing has close to no CPU or memory overhead on routers. In a first set of experiments, we measured the resource usage on a router caused by an increasing number of fake nodes, announcing different destinations. Table 4.1 shows the memory usage of fake nodes implemented as OSPF AS-external LSAs. Two distinct memory structure are impacted by the presence of fake nodes: *(i)* the RIB, has it has to register the new routes created by the LSAs; and *(ii)* the memory consumed by the OSPF process itself, as it has to keep every LSA in its LSDB. Even with a huge number of fake nodes (100 000), the total overhead on both processes was only 154MB—a small fraction of the total memory available. Additionally, the RIB size increase only happened as a result of our experimental setting, as the router started with an empty RIB,

Fake node count	RIB size (MB)	OSPF memory (MB)
1 000	0.09	0.56
5 000	1.58	5.19
10 000	3.56	10.96
50 000	19.67	56.37
100 000	39.78	113.17

Table 4.1: Huge augmented topologies have a very limited memory overhead.

Fake node count	FIB update duration (s)	Average time/entry (μs)
1 000	0.89	886.00
5 000	4.46	891.40
10 000	8.96	894.50
50 000	44.74	894.78
100 000	89.50	894.98

Table 4.2: Programming FIB entries in a router with Fibbing is fast, sub 1ms.

i.e., Fibbing actually added routes towards previously unknown prefixes. In a scenario where Fibbing is used to re-route existing prefixes over new paths, the size of the RIB would stay constant as the routes created by Fibbing would replace the previous ones.

In parallel, we sampled the CPU utilization[7] of the router every five seconds immediately after we started injecting fake nodes. The utilization was systematically low, at most 4%. This covers the processing of the received LSAs, FIB updates, and re-flooding process.

Fibbing quickly programs forwarding entries. The second set of experiments measured the time required by a router to update its FIB in reaction to received LSAs, *i.e.,* measure the speed at which their FIB can be programmed by the Fibbing controller. Table 4.2 shows the total time taken by the router to update its FIB, depending on the number of injected number of fake nodes, each announcing different destinations. In each experiment, we measured the total installation time by recording the timestamps at which the first and last FIB entries where updated. On average, the time to process and install one FIB entry was constant (around 900μs), independently of the total number of updated entries. This is several orders of magnitude better than many most OpenFlow switches [Jin+14; Rot+12]. As the installation of FIB entries is dis-

[7] As reported by the router operating system, *i.e.,* as a usage percentage covering the last minute.

tributed, naturally parallelized across all routers, Fibbing is thus able to program 1 000 of network-wide FIB entries within a second.

Fibbing has no impact on OSPF convergence time. Finally, our last experiments measured the impact of Fibbing (*i.e.,* the presence of additional AS-external LSAs) on the convergence time of routers. To that end, we measured the total FIB update time when failing a link, causing the router to lose of all its routes and thus to recompute an alternate nexthop for all of them. By comparing the measured time when the alternate nexthop was due to LSAs injected by Fibbing, to the one when Fibbing was not in use, we can then evaluate the total overhead of Fibbing when computing a complete RIB, *i.e.,* computing the shortest-paths towards every prefix. As in previous experiments, we repeated our measurements for a growing number of destinations (from 100 to 100 000). In every experiment, the presence of LSAs injected by Fibbing did not have *any* visible impact. The total convergence times with or without Fibbing were systematically within 4ms, with the router being even faster to converge in the presence of Fibbing in some cases.

This result can easily be explained as OSPF operates a clear distinction between the inter-router paths, and the actual routes in a network. As such, routers derives the content of their RIB in two steps: *(i)* they first compute the shortest-paths between routers; and *(ii)* they walk through every prefix announcement contained in their LSDB, applying the OSPF decision process (§4.1.1) to identify the preferred one and creating the corresponding routes. Consequently, while Fibbing might slightly increase the second phase of this computation, the total duration is still dominated by the shortest-path computation[8].

4.3.2 Our controller efficiently injects large topologies

To efficiently implement large augmented topologies, our Fibbing controller has to be able to inject large numbers of LSAs in a short amount of time.

To achieve this, our controller minimizes the per-LSA transmission overhead to the minimum. Indeed, LSAs are carried between routers in LSupdate messages, themselves included as payloads of IP packets. As such message can contain up to 40 AS-external LSAs at once, our controller packs successive LSAs whenever possible. Additionally, OSPF ensures the reliability of its flooding procedure by requiring neighbors to send acknowledgments for every received LSA. Batching together LSAs thus also limits the number of packets containing such acknowledgments that are received and must be processed by our controller.

[8]Recall that computing the shortest-path between two nodes using the Dijkstra algorithm has a time complexity in $O(e + nlog(n))$, in a network with n nodes and e edges.

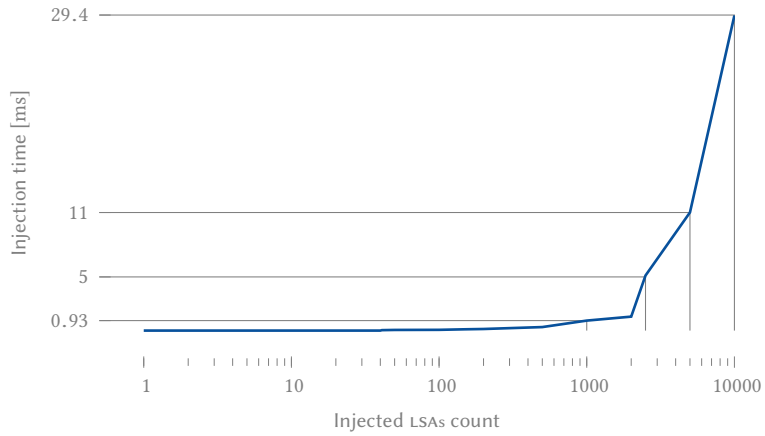


Figure 4.5: Our controller programs large topologies in milliseconds.

Fig. 4.5 shows the results of experiments where we recorded the time taken by our controller to inject up to 10 000 LSAs through an OSPF adjacency with a Cisco C7018 router. More precisely, the plot shows the time difference between the timestamp at which a request to inject LSAs was received by the fake node mapper, coming from the path manager of the controller (§4.2.1), and the timestamp at which the last LSupdate message was sent, implementing the request. We see that our controller was able to inject up to 10 000 LSAs (hence batched in 250 LSupdate messages) in less than 30 ms. Note that we never observed LSA retransmissions in our experiments, indicating that the router had no issue to cope with the sending rate of our LSAs.

4.4 Dealing with failures

Withstanding failures is critical for any network architecture. This section analyzes the impact of using Fibbing when facing different kinds of failures. More specifically, we distinguish between failures affecting the network itself (*i.e.,* routers and links) from those affecting the controller. We first describe the reactions of Fibbing to non-partitioning failures (§4.4.1), as those are by far the most common failure cases [Mar+08], then discuss how failures inducing network partitions affect the forwarding requirements implemented by the controller (§4.4.2). Finally, we conclude by an experiment (§4.4.3) which confirms the previous claims, by demonstrating that Fibbing is resilient to failures and recovers from them gracefully.

4.4.1 Fibbing quickly reacts to non-partitioning failures

Non-partitioning network failures impact traffic flows in three different ways.

First, some flows will be unaffected by the failures as their pre-failure forwarding paths do not cross any failed network element. Note that this does not preclude them from experiencing some service degradation as some other flows could be re-routed over their links, potentially congesting them.

Second, flows for which no Fibbing requirements have been specified rely on the IGP to establish a new path, without any action from the Fibbing controller. Tuned properly, IGPs have been showed to react extremely quickly to network failures, achieving sub-second convergence even in large networks [Fra+05], and leveraging advanced features commonly supported by current routers such as Loop-Free-Alternates (LFA) [RFC6571] and IP fast-reroute techniques [RI07; RFC5714].

The remaining flows have pre-failures paths controlled by Fibbing, and now crossing failed network elements. This requires the Fibbing controller to react as soon as it detects the failures, both to remove possible blackholes or loops due to previously injected lies [VVR14], and to avoid requirement violations due to the new IGP paths. Theoretically, the total failure recovery time is equal to the sum of three terms: *(i)* the time taken by the controller to detect the failure; *(ii)* its reaction time; and *(iii)* the time taken by the IGP to converge once the controller has injected new LSAs implementing its reaction. The time to detect the failure is bounded by the IGP convergence time, as flooding is faster than re-convergence. Our earlier measurements showed that the controller reaction time could be extremely small using the Simple algorithm (§3.4), and that the injection time was negligible. As such, the total recovery time for flows controlled by Fibbing is twice the IGP convergence time, thus below 2 seconds [Fra+05]. Additionally, as routers typically delay their shortest-path recomputation by a few tends of milliseconds when receiving new LSAs[9], our controller could in similar cases injects its reaction LSAs before the beginning of the first IGP convergence, further reducing the recovery time.

Controller failures lead to two different cases. If all controller replicas have failed, the consequence to the flows controlled by Fibbing are the same as if the network was experiencing a partition separating the controllers from the rest of the network, and are covered in the next section (§4.4.2). Otherwise, at least one or more replica is still running. We then distinguish two cases. Either the failed controllers were backup replicas, in which case, flows controlled by Fibbing do not experience any disruption. Or the failed controllers include the primary one, in which case the leader election procedure (§4.2.2) guarantees

[9] This reflects the fact that single topology changes usually results in multiple LSAs being sent, *e.g.*, a failed link causes at least two routers to report it.

that a new controller will eventually take over. Note that this leader election might cause some Fibbing LSAs to expire, violating the requirements they were implementing during a short time window. We argue that the short election duration, *i.e.,* the time to detect the controller failure and the flooding delay, makes this unlikely at best and constrains the duration of such disruptions to twice the IGP convergence time at worst (< 2s).

4.4.2 Fibbing supports both fail-open and fail-close semantics

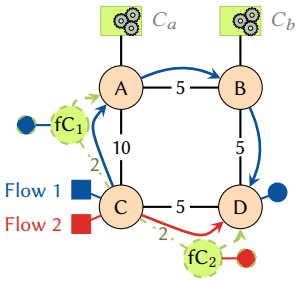
While very unlikely, catastrophic events such as the simultaneous failure of all controller replicas or a network partition isolating some routers from all controllers may still happen.

Unlike pure SDN solutions that leave the network uncontrolled in those cases, potentially turning it into a giant blackhole, Fibbing instead delegates by default the control to the underlying IGP. Indeed, recall that each LSA contains an age field (§4.1.1), set to be incremented every second by routers until it reaches the `max-age` value, causing it to expire and be ignored when computing the RIB. When a network partition occurs, LSAs injected by the Fibbing controller will then eventually expire and let the IGP in the partition restore the connectivity on its own, if possible. The Fibbing controller explicitly controls this field, and thus controls the speed at which the traffic will be handed back to the IGP. Faster hand-overs (high age values) come at the cost of additional control-plane overhead as the LSAs have to be refreshed more often.

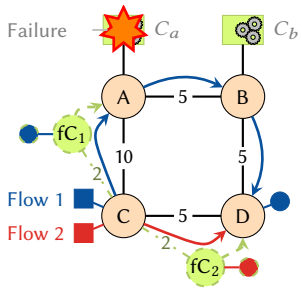
Leveraging this, Fibbing implements in the default case a fail-open failure semantic, best suited to non-critical requirements such as traffic engineering. Under this failure model, the corresponding destination prefixes are part as usual of the original IGP topology, and connectivity is restored by the IGP after the configured delay. For more stringent requirements, such as security ones (*e.g.,* firewall traversal), Fibbing can instead implement a fail-close semantic. In that case, the corresponding destination prefixes are not part of the original IGP topology, but introduced by the LSAs injected by the Fibbing controller. When a partition occurs, these LSAs will eventually disappear, blackholing the associated prefixes.

4.4.3 Our Fibbing controller gracefully handles failures

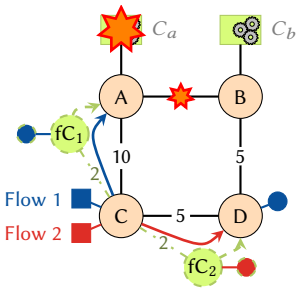
To demonstrate the failure resiliency of our controller, we emulated in GNS-3 [GNS3] the network shown in Fig. 4.6a, using images of Cisco IOS 12.4. In this network, two Fibbing controller replicas C_a and C_b are connected to routers A and B. Initially, the replica elected as leader is C_a. We connect two `iperf` [iperf] sources to A, and the corresponding sinks behind D. All links are configured with a maximal rate of 1 MB/s. Finally, we configure our controller replicas such that: *(i)* flow 2 has a fail-close failure semantic, and has



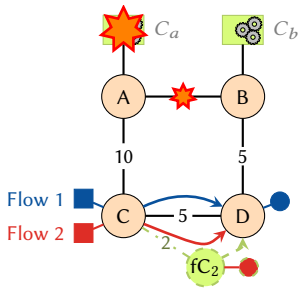
(a) Emulated topology from $t = 0$s to $t = 15$s.



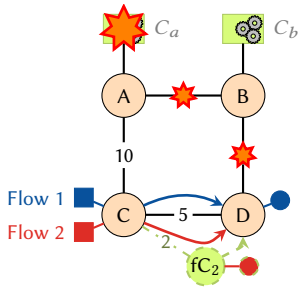
(b) Emulated topology from $t = 15$s to $t = 35$s and after $t = 145$s.



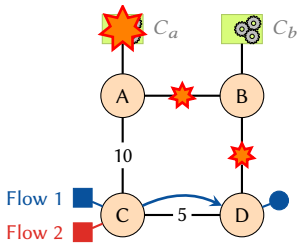
(c) Emulated topology from $t = 35$s to $t = 36$s.



(d) Emulated topology from $t = 36$s to $t = 60$s and from $t = 110$s to $t = 145$s.



(e) Emulated topology from $t = 60$s to $t = 65$s.



(f) Emulated topology from $t = 65$s to $t = 110$s.

Figure 4.6: Fibbing successfully implemented fail-close and fail-open failure semantics, and gracefully recovered upon the restoration of the connectivity.

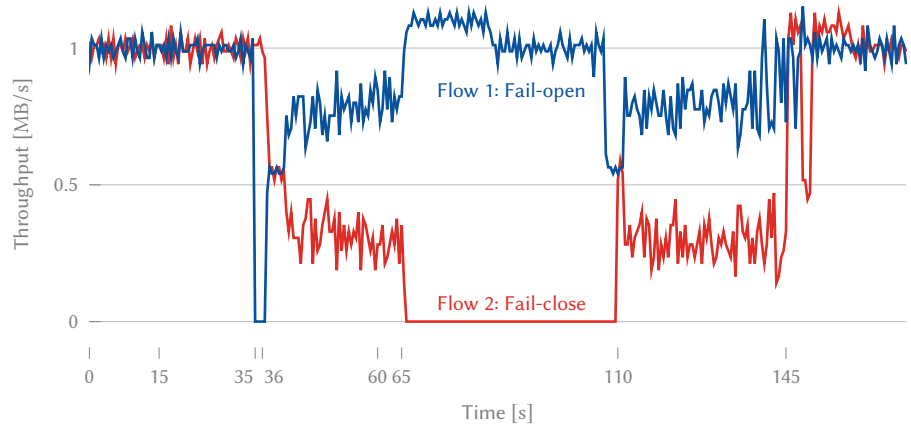


Figure 4.7: Measured throughput variations during multiple links failures and restorations

to cross link (C, D); and *(ii)* flow 1 has a fail-open failure semantic, and should preferably reach D via A, in order to maximize the throughput of both flows. Note that due to the fail-close failure semantic, the only LSA mentioning flow 2 is the one implementing fc_2. We then configured a static route on D to forward the destination prefix of flow 2 to the connected `iperf` sink, without redistributing it in the IGP.

Starting from a state in which both replicas and all links are up, we successively failed in this experiment *(i)* C_a at time $t = 15$; *(ii)* the link (A, B) at time $t = 35$s; ; and *(iii)* link (B, D) at $t = 60$. Finally, we re-establish both failed links, one at the time (at $t = 110$ and $t = 145$).

Fig. 4.7 shows the evolution of the reported `iperf` throughput during the experiment. In this plot, we see that the failure of the controller replica elected as leader (C_a) (Fig. 4.6b) has no impact on the forwarded flows. Indeed, as A quickly detects that C_a is unreachable and floods this information, C_b then detects the failure of its leader. Consequently, C_b starts refreshing the LSAs originally injected by C_a, preventing any disruption. When (A, B) fails, C_b then needs to remove fc_1 as it is creating a forwarding loop between A and C (Fig. 4.6c), hence blackholing the traffic of flow 2 at $t = 35$s in Fig. 4.7. The time to detect the failure and remove fc_1 was approximately 1*s*, causing both flows to be routed along (C, D) at $t = 36$s (Fig. 4.6d). Note that this time can be lowered by using fast failure detection mechanisms (*e.g.,* BFD [RFC5880]), disabled in this experiment. The failure of (B, D) results in a network partition (Fig. 4.6e) preventing C_b from controlling routers A, C and D. After about 5*s* ($t = 65$s), the LSA injected to enable routers to forward flow 2 expires,

blackholing it as required by the fail-close failure semantic (Fig. 4.6f). In contrast, as flow 1 is configured with a fail-open semantic, the IGP has sufficient knowledge to forward it without the Fibbing controller. Restoring (B, D) at $t = 110$s enables C_b to re-take control of the network, hence it re-inject fc_2 to enable routers to forward flow 2 (Fig. 4.6d). Finally, when (A, B) eventually comes back at $t = 145$s, C_b re-optimizes the distribution of both flows over the available paths (Fig. 4.6b).

4.5 Fibbing enables real-time traffic engineering

We conclude this chapter by demonstrating a practical application of Fibbing, where we use our controller to dynamically react to unexpected traffic surges.

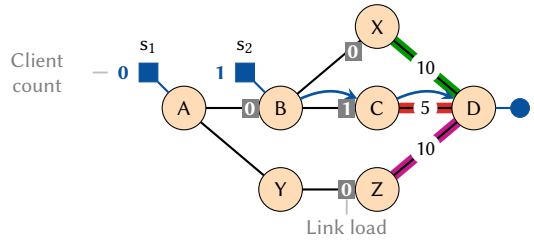
Bandwidth-intensive applications (*e.g.,* video streaming) impose hard-constraints on the performance of networks (*e.g.,* on available bandwidth, on queue sizes, and/or on loss rates) to ensure a good quality of experience for their users. To guarantee good network performance, operators typically optimize their network using traffic engineering techniques [Wan+08] (TE). Traditionally, these TE schemes pre-compute a network configuration for a predictable load (*e.g.,* using traffic matrices). Unfortunately, this is ineffective in the events of flash crowds [Ari+03]. For example, a sudden surge of traffic due to content shared over social networks could induce congestion, leading to service outages. Operators can thus either vastly over-provision their networks, hindering their profitability, or be at risk of service disruption.

Consider the network shown on Fig. 4.8a. In this network, two video servers (s_1 and s_2) are hosting content seldom accessed by clients located behind router D. As a result of a TE optimization, the operator of this network set the links weights in the network to 1, except for links (X, D), (C, D), and (Z, D). Such a network would suffer from two inefficiencies in the events of a flash crowd generating traffic from the servers to D: *(i)* many links would never be used; and *(ii)* all flows would systematically traverse the same path, competing for bandwidth

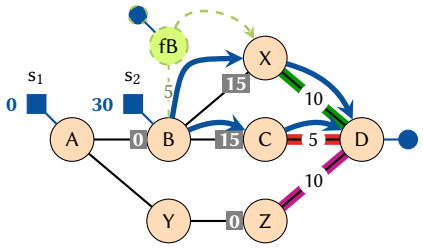
Fibbing provides better tools to react to flash crowds. Indeed, as it enables to program paths, on a per-destination basis, within milliseconds, it is thus able to react in near real-time to traffic demand fluctuations. We demonstrated this by emulating the network from Fig. 4.8a. In this network, we connected a Fibbing controller to Y. We developed an application running on top of the controller that monitors link loads in the network using SNMP, and is also notified by the video servers when they start or stop streaming content to a client. As it receives the up-to-date IGP topology from the controller, our application can then track the traffic demands, compute the overall network load, and op-

timize it [NKR13] if some link excess a configured threshold (35 MB/s in our experiment, with links configured with a maximal rate of 40 MB/s). Then, we simulated a flash crowd by progressively adding clients connected behind D and downloading videos from s_1 and s_2, and recorded the traffic crossing the three links connected to D. Fig. 4.9 plots the measured traffic volumes during our experiment.

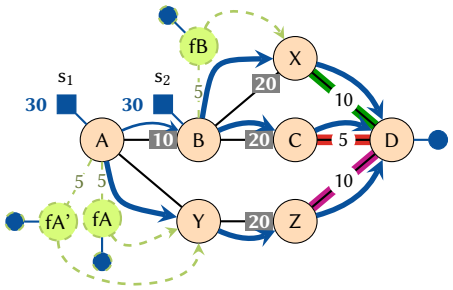
Originally ($t = 1$s), a single client connects to s_2 and downloads a video (Fig. 4.8a. At $t = 11$s, we started to add 29 more clients requesting videos from s_2. Eventually, these clients caused (C, D) to exceed the 35 MB/s threshold. In reaction, the Fibbing controller application started to load-balance at $t = 20$s the generated traffic over two paths (Fig. 4.8b). At $t = 33$s, we then started 30 more clients configured to request videos to s_1, causing first (X, D) then (C, D) to exceed 35 MB/s. To minimize the maximal link load, our controller then reacted at $t = 38$s (Fig. 4.8c) and introduced uneven load-balancing on A. This split 2/3 of traffic from s_1 to D over (A, Y) and the remaining 1/3 over (A, B). During this experiment, all clients experienced smooth video playbacks, as Fibbing temporally tripled the available bandwidth between D and the video servers. We performed the same experiment without Fibbing which, as expected, saw all the clients experiencing playback stutters, as (B, C) became congested after 20s.



(a) Initial topology, where a single client downloads a video from s_2.



(b) At $t = 20$s, Fibbing enables load-balancing for traffic between s_2 and D as the number of clients increases.



(c) At $t = 38$s, Fibbing introduces uneven load-balancing on A as s_1 experiences a flash crowd.

Figure 4.8: Fibbing performs on-demand load-balancing to react to unexpected traffic surges, minimizing the overall impact on the network.

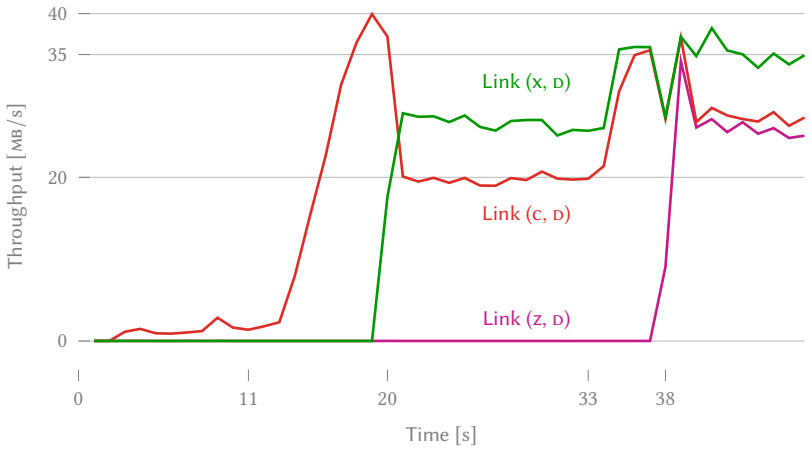


Figure 4.9: As the demand increases, Fibbing decreases the maximal link load by programming additional paths.

5 Summary

Centralizing routing decisions meets the flexibility needs of network operators, but often sacrifices the robustness and scalability of distributed protocols. In this part, we presented Fibbing, an architecture that achieves both flexibility and robustness through central control over distributed routing. Fibbing is expressive, and enables operators to easily implement traffic engineering schemes, on-demand load-balancing, or provision backup routes, with advanced failure semantics. Fibbing comes with algorithms that automatically translate high-level forwarding objectives to low-level primitive constructs exposed to routers. We showed that we could achieve this at scale, enabling Fibbing to be used in large networks. Fibbing works with any unmodified commercial router supporting OSPF, and has little performance overhead.

Both our implementation of the Fibbing algorithms and of the Fibbing controller have been released publicly, under an open-source license[1]. These have been successfully re-used and extended by students and researchers alike, confirming their maturity [CRS16; Zan+18; Arr16; Mol16]

Fibbing opens up several research perspectives. First, as Fibbing combines centralized routing decisions with distributed route computation, new research could investigate alternate tradeoffs (for example, would there be any benefits in partially distributing the Fibbing requirements?). A second direction would be to explore how Fibbing could benefit from (or benefit to) newer, actively developed protocols, *e.g.,* Segment Routing [RFC8402], Flexible IGP algorithms [Pse+18], or Multiple Topology Routing (MTR) [RFC4915], or from alternate routing platforms such as OpenR [openr]. Finally, a third thread of research would be to explore how Fibbing could be integrated with monitoring infrastructures, enabling self-driven networks.

[1] Available at `https://github.com/fibbing`.

Part III

Enabling fined-grained network monitoring

Deterministic traffic sampling in transit networks

6

In this chapter, we present a monitoring framework, Stroboscope, that combines the visibility of traffic mirroring with the scalability of traffic sampling. We first motivate (§6.1) why existing monitoring techniques are unable to provide fined-grained measurements in ISP networks. Then, we present a high-level overview (§6.2) of our framework, highlighting the key steps and challenges of our approach. Next, we formulate practical algorithms to: *(i)* adapt monitoring queries depending on the network state as well as estimate unknown traffic demands in real time (§6.3); *(ii)* compute optimally placed mirroring rules (§6.4); and *(iii)* schedule mirroring actions to adhere to a given budget (§6.5). Afterwards, we present a full implementation (§6.6) of Stroboscope, detailling different techniques to collect traffic slices. To demonstrate its scalability and practicality, we evaluate our framework using through benchmarks, simulations, and tests on Cisco routers (§6.7). We then conclude by showing practical examples using Stroboscope to collect fine-grained measurements (§6.8). Finally, we compare Stroboscope to alternative monitoring frameworks (§6.9).

6.1 Motivation

While essential to efficiently operate a network, acquiring an accurate and fine-grained visibility over the traffic is challenging in ISP networks. First, as they do not control the end-hosts, they depend on in-network solutions (*e.g.,* middbleboxes, or directly on routers). Second, due to the available monitoring tools (*e.g.,* NetFlow [RFC3954], sFlow [RFC3176]) and the traffic volume they carry, operators rely on packet sampling. By design, sampling provides no guarantee on which traffic flows will be sampled, by which router and at what time, as only few packets are randomly sampled (*e.g.,* 1 out of 1024). Except for few heavy-hitters [Zha+04], even minutes-long collections of random samples typically provide coarse-grained and inaccurate bandwidth estima-

tions for the large majority of the prefixes. Moreover, the likelihood of randomly sampling the same set of packets at different places as they traverse the network is extremely low. As such, analyzes consisting of measurements from different routers are inherently limited—*e.g.,* comparing measurement timestamps estimate latencies is unreliable at best, estimating forwarding paths by checking which routers saw the same flow 5-tuples might hide seldom used paths or routing oscillations. Consequently, reasoning on any network-wide forwarding behavior using random sampling is impossible. As such, these techniques are only used to provide bandwidth estimates, which causes ISP networks to suffer from an extremely poor visibility.

We experimentally confirmed these limitations by analyzing NetFlow data collected by hundreds of routers in a Tier-1 ISP. We looked at 10 minutes-long collection windows, and counted the number of NetFlow records associated to every prefix from the entire BGP table in such windows. We observed that the vast majority of prefixes (65%) have no record at all (*i.e.,* are "invisible" from NetFlow's point of view)[1]. 15% of the prefixes have only 2 NetFlow record in those 10 mins, and only 10% of all prefixes have more than 30 records. Worse, 75% of these observed flows were only reported by a single router, making it *impossible to reliably track flows network-wide*, even for the largest heavy hitters.

ISP operators are thus incapable of answering practical questions such as:

- *"What is the ingress router for a given packet seen at a specific node?"*
- *"Which paths do the traffic follow?"*
- *"Is the network-wide latency acceptable?"*
- *"Is traffic load-balanced as expected?"*

Answering these questions using a combination of control-plane measurements (*e.g.,* retrieving all routing tables) and active measurements (*i.e.,* probing) is insufficient for at least two reasons. First, control-plane measurements may not reflect what is happening on the data-plane (*e.g.,* software bug, degraded optical link causing framing errors and packet losses, or partial failure of a link bundle). Second, guaranteeing that active measurements are representative of the performance experienced by real user traffic is challenging in the presence of stateful middleboxes (*e.g.,* firewall, traffic shapers) and DiffServ [RFC2474] as it requires generating traffic closely mimicking the user one.

[1] This is expected for two reasons: *(i)* most of those prefixes were likely to not have any traffic at all transiting through the network; and *(ii)* the few that did have traffic were small enough to not be sampled as the routers have a sampling rate of 1 packet out of every 1024.

Stroboscope. This chapter presents Stroboscope, a scalable monitoring system that complements existing tools like NetFlow, by enabling fine-grained monitoring of *any* traffic flow. Stroboscope exploits the possibility to extract small traffic samples (*i.e.,* traffic slices) programmatically, by activating and deactivating traffic mirroring for any destination prefix, up to a single IP address, network-wide, and within milliseconds. Our tests confirm that this possibility is available today, on COTS routers, making Stroboscope immediately deployable.

By coordinating packet mirroring across routers, Stroboscope implements *deterministic packet sampling*: it collects copies of the same packets from multiple locations, during a given time window. This enables Stroboscope to follow such packets as they cross the network, hence to precisely measure network behaviors such as forwarding paths, one-way delays and load-balancing ratios, *as experienced by the actual user packets*. Traffic slices with no packets are also informative. Indeed, as these signal the absence of traffic, they enable Stroboscope to derive additional forwarding properties such as packet losses or isolation across regions.

6.2 Stroboscope

This section presents a description of Stroboscope, highlighting the different components of our monitoring framework. As visible on Fig. 6.1, Stroboscope layers two sets of building blocks. First, a compilation layer translates monitoring queries (§6.2.1) given by the operator into a schedule (§6.2.2) of low level mirroring actions. Beside providing accuracy guarantees on the measurements collected by Stroboscope, this compilation also ensures that the overhead of Stroboscope (*e.g.*,, bandwidth usage) falls within an operator-defined monitoring budget. Second, a runtime (§6.2.3) layer executes the schedule computed by the compilation process. As it starts collecting measurements, Stroboscope can then answer the operator's queries. Additionally, this runtime layer ensures that Stroboscope's guarantees are preserved even when facing unexpected events (*e.g.*,, routing changes, or flash crowds).

We now intuitively illustrate all components of the Stroboscope framework on a running example, shown on Fig. 6.2. More precisely, we consider a network operator who suddenly receives reports from some of its customers that a prefix (`1.2.3.0/24`) cannot be reached through its infrastructure. To troubleshoot these reports, the operator wants to use Stroboscope to: *(i)* check that the corresponding traffic flows follow the expected paths; and *(ii)* measure key performance indicators, such as packet loss rates and path latencies.

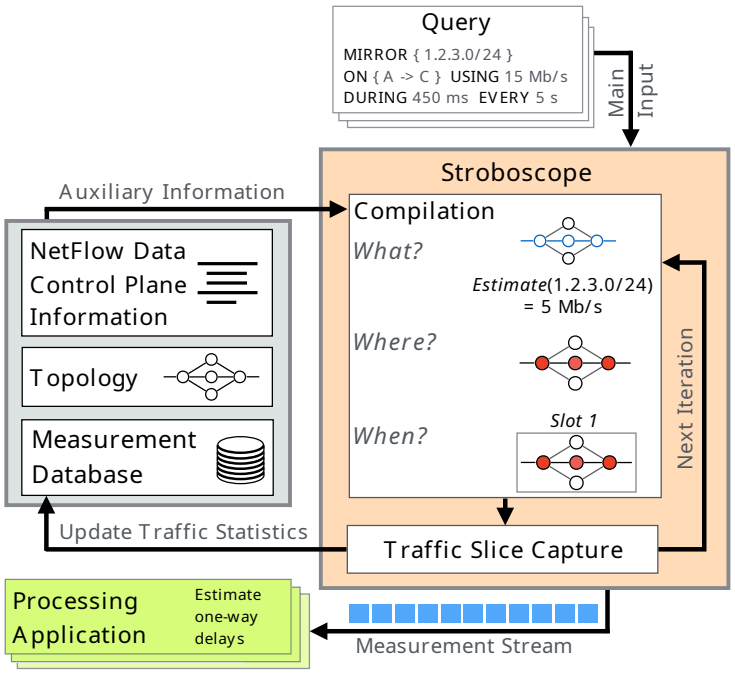
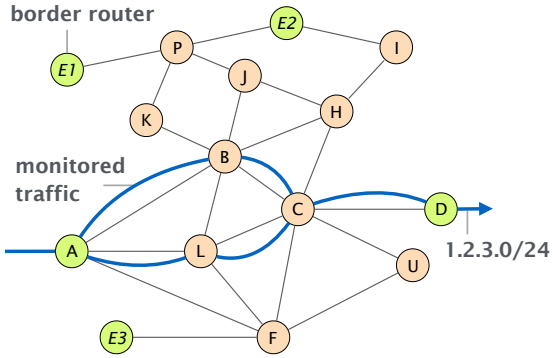
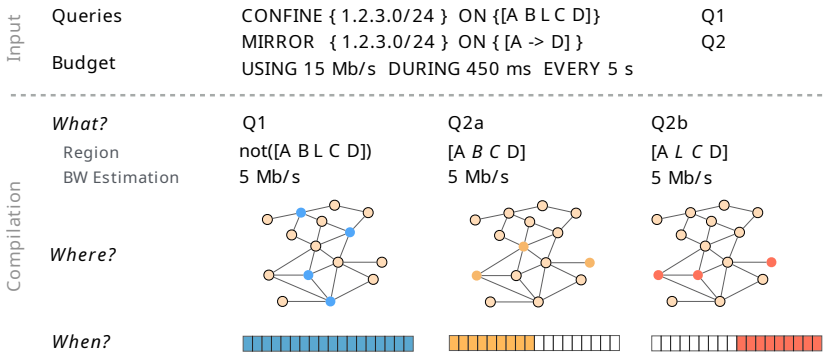


Figure 6.1: High-level overview of Stroboscope's components.

.



(a) Sample network.



(b) Compilation process translating high-level queries in a measurement campaign.

Figure 6.2: From high-level monitoring queries and the current network state, Stroboscope computes a measurement campaign schedule meeting the monitoring budget.

6.2.1 Expressing monitoring queries

Stroboscope defines a small, SQL-like, language that lets operators specify their monitoring goals through a succession of high-level queries.

((MIRROR | CONFINE) <*prefixes*> ON <*paths*>)+
USING <*Gbps*> DURING <*sec*> EVERY <*sec*>

Monitoring queries are defined on a per prefix basis. For each set of IP prefixes (up to a single address), a query then specifies whether traffic should be mirrored (MIRROR) or confined (CONFINE). Additionally, the operator also specifies where this traffic should be mirrored (or the region in which it should be confined) by specifying a set of nodes using the ON operator. For convenience, the $\longrightarrow$ operator can be used to indicate that this set of nodes should be dynamically computed according to the paths used by the routing protocols (*e.g.,* A $\longrightarrow$ B denotes all paths from A to B). MIRROR and CONFINE queries differ in when they mirror traffic: the former continuously mirrors traffic as it flows across the specified nodes, while the latter only mirrors traffic that *leaves* a specified region. Combined with the use of the $\longrightarrow$ operator, these two mirroring modes let operators verify that the behavior of the data-plane matches the one reported by the control-plane, *e.g.,* a MIRROR query detecting that a traffic flow is forwarded on paths other that those computed by the control-plane likely indicates either that a re-convergence is happening or that some FIB corruption is happening.

Finally, operators specify a monitoring budget as a set of constraints on: *(i)* the maximum rate of mirrored traffic (USING) allowed; *(ii)* the duration of any measurement campaign (DURING); and *(iii)* the frequency at which a measurements campaign should be repeated (EVERY).

Coming back to our example, the operator can instruct Stroboscope to mirror traffic along all IGP paths between A and D using a MIRROR and a $\longrightarrow$ construct (see Fig. 6.2b). Additionally, a CONFINE construct is specified to verify that these paths are the only ones carrying traffic towards `1.2.3.0/24` (*i.e.,* that the MIRROR queries see the complete traffic towards that prefix).

While simple, our language supports several practical use cases. Among others, MIRROR queries enable network-wide path tracing, *i.e.,* following a given packet as it traverses a sequence of nodes. Packet copies can then be analyzed by monitoring applications to estimate data-plane performance, like packet loss or path latency, or to inspect packet payloads. CONFINE queries are especially useful to detect unwanted forwarding behavior, (*e.g.,* traffic shifts, security policies) at runtime, and to complement information from MIRROR queries (*e.g.,* on paths not taken by given traffic flows). Finally, its simplicity enables our language to be easily integrated with automated network monitoring solutions, which generate new queries dynamically on be-

half of the operator to perform more complex analyzes (*e.g.,* monitoring a large IP prefix by iterating through its sub-prefixes).

6.2.2 A three-staged compilation process

Given a high-level query, determining which flows to mirror, where and when is both hard and potentially dangerous. Aggressive mirroring strategies can lead to significant congestion (*e.g.,*, if many routers mirror traffic for popular destinations) and inaccurate results (*e.g.,*, if congestion affects the mirrored traffic). Conversely, conservative strategies can lead to poor coverage and slow answers.

Stroboscope tackles those challenges on behalf of operators. From high-level queries, Stroboscope derives *measurement campaigns*, *i.e.,* schedules of mirroring rule (de-)activations that: *(i)* provide strong guarantees on budget compliance; *(ii)* maximize accuracy by activating mirroring rules as often as possible; *(iii)* minimize the number of mirroring locations to both lower the mirrored traffic volume and decrease the control-plane overhead. Stroboscope derives these measurement campaigns in three successive steps, each addressing a different aspect of the problem.

1. What? Resolving high-level queries (§6.3). Stroboscope begins by translating all input queries into concrete ones, defined on actual paths and flows. To this end, it collects routing (*e.g.,* IGP and BGP) feeds, enabling it to keep an up-to-date view of the forwarding paths defined by the control-plane, as well as the location of prefix announcements (*e.g.,* BGP border router). In parallel, it collects NetFlow data when available and maintains a *measurement database*, storing results from past measurement campaigns. Based on this information, Stroboscope then estimates per-prefix traffic volumes, which it passes along the concrete queries to the next compilation step.

In our example, Stroboscope estimates the traffic demand for `1.2.3.0/24` to be 5 Mbps. Resolving [A ⟶ D] then yields two paths, [A B C D] and [A L C D], causing (Q2) to be split into two subqueries (Q2a, Q2b), one for each actual path. Note that the confine query is not split due to its semantic (*i.e.,* it does not track a packet along specific paths).

2. Where? Optimizing mirroring locations (§6.4). Second, Stroboscope selects mirroring locations to answer the different queries. While a strawman approach would place such rules on every node specified in the input queries, this would cause the mirrored traffic to consume large amount of traffic, likely exceeding the monitoring budget. Instead, Stroboscope minimizes the total number of mirroring rules by optimizing their locations using two provably correct algorithms. Doing so, it minimizes the mirrored traffic and the control-plane overhead to deploy them.

The first algorithm (§6.4.1) optimizes the placement of MIRROR queries. The key insight is to leverage properties of the *complete* network topology to prune mirroring rules. For instance, for Q2a in Fig. 6.2b, no mirroring rule is required on router C, as C is the only 1-hop path between B and D. Consequently, by comparing the TTL of packets mirrored at B and D, we can infer whether the traffic traversed C without actually mirroring there.

The second algorithm (§6.4.2) place mirroring rules for CONFINE queries. The key insight is to place heavily rate-limited mirroring rules all around the region specified in the query. This way, CONFINE queries do not mirror any traffic by default, and only few packets per location are mirrored when the queries are violated. Our algorithm minimizes the number of surrounding rules by placing as close as possible to the egress routers of the monitored prefix. For example in Fig. 6.2b, the algorithm places only one mirroring rule on P for (Q1) to detect possible packets crossing [A B] and leaving the network at $E1$ or $E2$.

3. When? Scheduling mirroring actions (§6.5). Finally, Stroboscope assembles a measurement campaign by scheduling mirroring actions (*i.e.,* mirroring rule (de-)activation) over time. This schedule spreads these actions according to the estimated traffic volumes to meet the budget, while at the same time maximizes the monitoring accuracy by activating mirroring rules as much as possible. Computing this is a variant of the bin-packing problem, which is NP-hard. To scale, Stroboscope encompasses fast approximation heuristics ($O(n \log n)$ where n is the number of queries) whose results are close to optimal.

In our example, Q2a and Q2b in Fig. 6.2 cannot be scheduled at the same time given the specified budget of 15 Mbps. Indeed, with 4 different mirroring rules, they would require a total of 20 Mbps. Stroboscope therefore schedules Q2a and Q2b each for half of the timeslots. In addition, as Q1 does not mirror any traffic unless a violation is detected, Stroboscope schedules Q1 for all the timeslots, so that any violation to Q1 can be detected.

This compilation process results in a measurement campaign which achieves deterministic sampling, *i.e.,* packets for one specific query are mirrored from well-defined locations for a given amount of time, while adhering to the monitoring budget.

6.2.3 Carrying out measurement campaigns

Stroboscope's runtime (§6.6) executes the measurement campaigns, (de-)activating mirroring rules according to the previously computed schedule. As it receives mirrored packets, Stroboscope can then dynamically check that its accuracy and budget guarantees are achieved.

Meeting the monitoring budget requirements inherently depends on two assumptions. First, Stroboscope checks the correctness of its demand estimations by monitoring the total traffic being mirrored and stops the measurement campaign when detecting a budget violation. Such a premature termination is enforced within one schedule timeslot (a few milliseconds). Second, as CONFINE queries are not expected to mirror traffic, and only require one packet when violated, they are rate-limited. Similarly, to meet its accuracy guarantees, Stroboscope monitors the routing protocols feeds and flags measurements collected during convergence (*e.g.,* as the forwarding paths change, some MIRROR queries might report false positive of packets "disappearing").

Whenever Stroboscope detects that an assumption behind its guarantees no longer holds, it then trigger a new compilation of the monitoring queries.

As the measurement campaign progresses, Stroboscope outputs a stream of collected mirrored packets with their meta-data (*e.g.,*, timestamp, corresponding query, router at which it was mirrored), meant to be processed by the operators or external applications.

6.3 From abstract to concrete monitoring queries

To build a measurement campaign from high-level monitoring queries, Stroboscope begins by translating these queries to well-defined ones using knowledge about the current state of the network. More precisely, it first ensures that all input queries are defined over well-specified paths and regions, by translating all $\longrightarrow$ constructs found in the input queries (§6.3.1). Then, it estimates the traffic volumes of the monitored prefixes (§6.3.2).

6.3.1 Resolving loosely defined regions

When writing monitoring queries (§6.2.1), operators associate a set of prefixes to a set of nodes, hereafter referred to as regions (*i.e.,* connected components of the network graph) for CONFINE queries and paths for MIRROR queries. To enable queries to dynamically adapt to changes in the routing control-plane, hence its computed forwarding paths, the $\longrightarrow$ construct can be used as a placeholder to denote those paths. More precisely, Stroboscope replaces any expression $\mathtt{S}\longrightarrow\mathtt{T}$ with the forwarding paths from router S to router T as provided by the routing protocols (*e.g.,*, the IGP) running in the network. If no such path can be found, Stroboscope returns a compilation error. For example, for the query (Q2) in Fig. 6.2b, [A $\longrightarrow$ D] is translated as [A B C D] and [A L C D], assuming that the forwarding paths towards `1.2.3.0/24` are defined by the IGP and that all links in Fig. 6.2a have a cost of 1.

Whenever the $\longrightarrow$ operator is present at the start (resp. end) of a query, Stroboscope replaces it with the set of all ingress (resp. egress) routers that

receive traffic for the prefix in the query—*e.g.,* leveraging BGP information if present, or static knowledge of all network border routers. Using this feature, the queries from Fig. 6.2b can be generalized to encompass all paths terminating in D (instead of those starting at A and ending at D) by replacing [D ⟶ D] with [⟶ D]. This would then enable the operator to discover on the fly which ingresses are active. Those translations are updated at the start of each measurement campaign, so that Stroboscope performs the following measurements consistently with the latest available routing information, and flags the previous measurements if collected during routing changes.

6.3.2 Estimating traffic volumes

Beside specifying how quickly a measurement campaign should complete, the monitoring budget also specifies a maximum amount of bandwidth that can be used by the mirrored traffic. To compute a measurement campaign that adheres to this budget, Stroboscope thus needs information about traffic volumes for every prefix specified in the input queries. It is *fundamentally impossible* to exactly know how much traffic will be destined to any prefix ahead of measurements, *i.e.,* any flow can exhibit unexpected traffic variation at any point in time.

Stroboscope does not require exact traffic predictions as it includes runtime mechanisms bounding the amount of excessive traffic (§6.5.2). However, having traffic estimates that closely match the real demands is highly desirable as it increases the likelihood that multiple queries can be scheduled in the same timeslot. This maximizes their accuracy as they are scheduled more often, and decreases the chance that the schedule is infeasible—especially when using heuristics (§6.5). To dynamically estimate the traffic demand towards one prefix, Stroboscope leverages data collected in past measurement campaigns towards that prefix. Indeed, as Stroboscope's measurements consist of mirrored traffic, during a well-defined time window, they can be used to derive the traffic volume transiting through the network during that particular window. More precisely, Stroboscope stores in its measurement database the peak traffic demand measured over a customizable number of minutes (5, by default). Stroboscope then uses such value as a conservative estimate of the traffic demand towards that prefix.

If no such historical data is available, Stroboscope first attempts to leverage NetFlow data if available (or any other source of information). If there are enough records (*e.g.,* more than 30), collected during a reasonable time window (*e.g.,* 5 min), Stroboscope then uses the peak recorded value as traffic estimate. Otherwise, Stroboscope assumes that the query will require the complete bandwidth budget, and thus schedules it in its own slot. By analyzing the resulting mirrored traffic, Stroboscope will then be able to refine its

traffic prediction for the next measurement campaigns. We stress that the risk of significantly exceeding the budget by scheduling queries with no historical data is limited for two reasons. First, as those target prefixes are not present in NetFlow records, they are likely to carry a limited amount of traffic as they generated few or no records over minutes of random sampling. Second, the mirroring rules are only active during a single timeslot, limiting the duration in which the mirrored traffic would exceed the budget to tens of milliseconds at most—about 25 ms in our current implementation (§6.7).

6.4 Optimizing mirroring location

Stroboscope optimizes the location of mirroring rules using two algorithms, one per type of query. More specifically, MIRROR queries are optimized by the Key-points Sampling algorithm (§6.4.1) while CONFINE queries are handled by the surrounding algorithm (§6.4.2). Each algorithm minimizes the number of mirroring locations also providing *high accuracy guarantees* of the produced measurements (*e.g.,* packets violating CONFINE queries are never missed). Fewer mirroring locations (*i.e.,* mirroring rules) come with two benefits: *(i)* the cost of the query is minimized, enabling to schedule it more often; and *(ii)* the control-plane overhead is reduced as fewer mirroring actions have to be performed during the measurement campaign.

Stroboscope's algorithms take as input the operator-specified queries and the complete network topology—*i.e.,* including all links and nodes currently down. This ensures that the algorithms always guard against all possible network paths, and never select mirroring locations breaking the accuracy guarantees due to transient topology changes.

On one hand, a mirroring location pointing to an edge (N, M) defines a mirroring rule that mirrors matching packets leaving router N through its outgoing interface towards router M. On the other hand, a mirroring location pointing to a node N defines a mirroring rule that mirrors matching packets leaving router N regardless of its outgoing interface.

6.4.1 Key-points Sampling algorithm

The objective of MIRROR queries is to collect multiple copies of the same packet as it traverses the network over the specified path. Based on this, the Key-points Sampling algorithm (KPS) minimizes the number of mirroring locations needed to answer a given MIRROR query.

Goal. Given a MIRROR query on a path P, KPS selects the minimal set of routers which will capture traffic following P which enables to decide for any packet mirrored by the query whether it was forwarded along P or not.

While the absolute minimum set of router to achieve this would be to select both ends of the path, this is often insufficient. Indeed, consider the query (Q2) in Fig. 6.2a that mirrors traffic along [A B C D]. Mirroring only at A and D would prevent to distinguish packets forwarded over [A B C D] from those flowing over [A L C D].

General solution. By default, KPS returns all the routers in the path. This guarantees that the resulting measurement campaigns track all packets crossing any subset of routers in the path. Then, for each mirrored packet, Stroboscope checks if there exists a sequence of routers such that the Time-To-Live (TTL) of the packet found in multiple traffic slices is decreased exactly by 1 at each hop in the sequence. Assuming that every router decreases packets' TTL by 1[2], the existence of such a sequence then unambiguously indicates whether the packet followed the path or not.

Optimizations. To go beyond this general solution, and actually minimize the number of mirroring locations, KPS exploits the following theorem.

Theorem 4 (KPS only needs to mirror the ends of paths of unique length)**.** *Let a forwarding path P be the concatenation of sub-paths $Q_1, \ldots, Q_n$. MIRROR queries on P can be correctly answered by mirroring only on the endpoints s_i and t_i of all Q_i such that no other forwarding path from s_i to t_i has the same length as Q_i.*

Proof. First, we show that if any $Q \subseteq P$ is the only path of length x from p to s, we can always distinguish mirrored packets that have been forwarded over Q by just mirroring on p and s. Let l be the length of Q, and let t_p and t_s be the TTL values of any packet mirrored from p and s. Under the assumption that the TTL is properly decreased (by one) at each forwarding hop, we can unambiguously determine if the packet has been forwarded over Q: If $t_s = t_p - l$, then Q must be the traversed sub-path because Q is the only path of length l between p and s by hypothesis.

The statement of the theorem then follows by noting that the same property applies to any sub-path Q_i, as well as to their concatenation—*e.g., P.* □

As an illustration, consider Fig. 6.2a. The path [A B C D] can be seen as the concatenation of [A B] and [B C D]. As [B C D] is the only path in the topology of length 3 from B to D, Theorem 4 states that we can omit C from the mirroring locations. In other words, we can distinguish packets traversing [B C D] as the only ones whose TTL in D is equal to their TTL in B minus 2.

Algorithm. Given a path P, KPS iterates through all concatenations of sub-paths $Q \subseteq P$ that result in P, sorted in ascending order by the number of

[2]Which is the case if routers properly implement the specifications of IPv4, IPv6 and MPLS [RFC791; RFC2460; RFC3443].

sub-paths composing each of them. For each concatenation, KPS then verifies that Theorem 4 holds on all of its sub-paths. Checking whether a path [A ... B] is the only one of length l between A and D is done by performing a depth-first search on the network graph, rooted at a, and with a maximal search depth of l. As one given sub-path is likely to be present in multiple possible concatenations, we cache the results of such searches to speed up the next ones. If Theorem 4 holds on all sub-paths, KPS then returns as set of mirroring locations all endpoints of each sub-path.

For example, for the path [A B C D], KPS first considers the concatenation of sub-paths {[A B C D]} (*i.e.,* the path itself). As shown above, Theorem 4 does not hold for that path as it has the same length as [A L C D]. This causes KPS to then consider either [A B C][C D] or [A B][B C D]. As Theorem 4 applies for the latter concatenation (and not for the former one as [A B C] has the same length as [A L C]), this is thus the minimal concatenation. KPS thus selects as mirroring the three routers at the endpoints of its sub-paths, *i.e., {A,B,D}*.

KPS is theoretically inefficient, as any of the depth-first searches it runs can potentially explore an exponential number of paths. However, our evaluation (§6.7.2) shows that KPS takes milliseconds to process paths in real networks, due to their sparsity and the limited path lengths.

As the goal of MIRROR queries is to track a packet through the network, they can only be applied on simple paths. As such, if the ON operator of a MIRROR query defines a region (either by hand, or due to the use of the $\longrightarrow$ construct resolving (§6.3) to multiple paths), we create one sub-query for each path in the region, hence apply KPS one each sub-query.

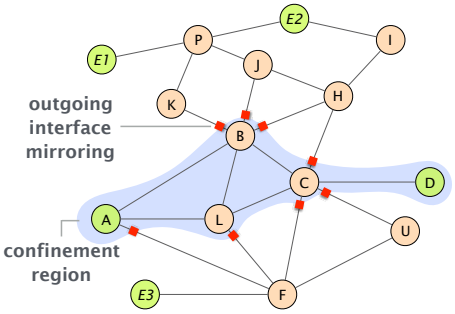
6.4.2 Surrounding algorithm

CONFINE queries aim at catching packets exiting a given confinement region (*i.e.,* connected component) of the network graph. To that end, Stroboscope runs the surrounding algorithm to minimize the required mirroring locations.

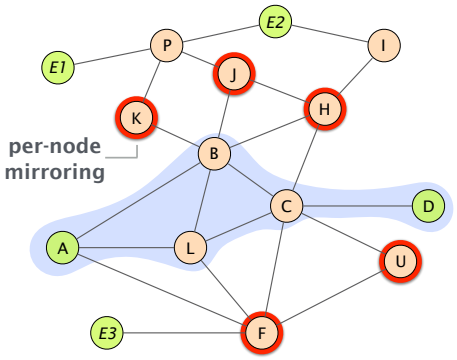
Goal. Given a CONFINE query on a region R, the surrounding algorithm selects mirroring locations (edges or routers) such that no packet can cross a link not belonging to the region without being mirrored.

One of the challenges faced by the surrounding algorithm is to avoid capturing interfering traffic, *i.e.,* packets for the prefix of the query which did not originate from the confinement region. For example, assume in Fig. 6.2 that there exists a transit traffic flow towards `1.2.3.0/24` along the path [E1 P E2]. In such case, P cannot be selected as mirroring location for (Q1) as it would incorrectly flag packets following that path as leaving the confinement region.

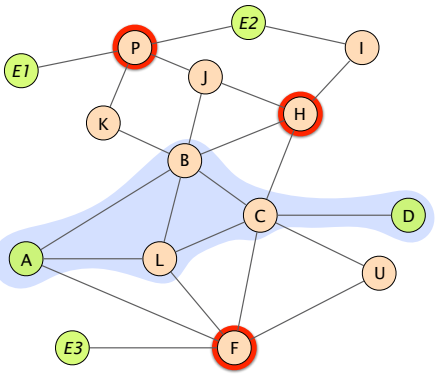
General solution. Given a confinement region R, we define the *edge surrounding* $E(R)$ of R as the set of directed edges (R, N) such that $\text{R} \in R$ and $\text{N} \notin R$.



(a) Default mirroring locations computed by the surrounding algorithm on CONFINE queries defined on the region A B C L D.



(b) Optimized placement if no flow for the queried destination crosses neighbors of the confinement region (no interfering traffic).



(c) Optimized placement in the absence of both interfering traffic and network anomalies (blackholes, forwarding loops).

Figure 6.3: Depending on properties of the network graph and knowledge about the correctness of the network, we can reduce the number of mirroring locations and keep the same guarantees.

By default, the surrounding algorithm returns as mirroring locations all edges in $E(R)$. Fig. 6.3a visualizes the output of this algorithm for the confinement region of (Q1) in Fig. 6.2.

We prove the correctness of the default output of the surrounding algorithm using the following theorem.

Theorem 5 (The edge surrounding is sufficient to answer a CONFINE query)**.** *A CONFINE query on a region R can be correctly answered if and only if the set of mirroring locations is the edge surrounding of R.*

Observe that the theorem intuitively holds because: *(i)* by definition, exiting R implies that a packet is forwarded from a router in R to another outside R, hence over a link in the edge surrounding; and *(ii)* only the packets exiting R are mirrored using the edge surrounding. Leveraging these observations, we prove Theorem 5 using the following lemma.

Lemma 6 (Mirroring rules on the edge surrounding answers a CONFINE query)**.** *Given a region R, its edge surrounding $E(R)$ and any prefix d, capturing all packets with a destination address included in d and only those packets is guaranteed if and only if mirroring rules matching d are active on every edge in $E(R)$.*

Proof. We first show that all packets exiting R are captured if and only if mirroring rules are placed on all edges in $E(R)$. Consider any packet p entering R through an ingress router $\mathrm{I} \in R$. For p to exit R, there must be a node $\mathrm{R} \in R$ which forwards p to a node $\mathrm{O} \notin R$, using the edge (R, O). If mirroring rules are active on all the edges in $E(R)$, then p is detected, as the definition of both $E(R)$ and (R, O) implies $(\mathrm{R}, \mathrm{O}) \in E(R)$.

Finally, consider a packet p is mirrored by a rule located on an edge $(\mathrm{X}, \mathrm{Y}) \in E(R)$. By definition of the mirroring rule: *(i)* p must have a destination address included in d; *(ii)* p must have crossed a node $\mathrm{X} \in R$ and be forwarded to a node $y \notin R$, by definition of $E(R)$. This implies that only packets leaving R and destined to the target prefix are mirrored, which yields the statement. □

First optimizations. Mirroring the surrounding of a confinement region does not generate any mirrored packet if the traffic is indeed confined to that region. Nevertheless, minimizing the number of mirroring locations would reduce both the control-plane overhead when activating the rule, and the time needed to deactivate it when it starts mirroring traffic.

Whenever available, the surrounding algorithm uses routing information to reduce the number of mirroring locations. More precisely, knowing the forwarding paths for the queried prefixes can be used to safely push mirroring locations one hop away, *i.e.,* to move them from outgoing interfaces of routers

in R to neighboring routers. For example in Fig. 6.3a, if no forwarding path for `1.2.3.0/24` crosses F, then F itself can be used as a mirroring location (at the router level), replacing the three mirroring rules previously located on the edges (A, F), (L, F), and (C, F), as shown on Fig. 6.3b.

We define the *node surrounding* $N(R)$ of a region R as the set of routers that are directly connected to at least one router in R and that are not part of any forwarding path for the prefix in the query. The surrounding algorithm then selects as mirroring locations first all nodes in the node surrounding, then all edges in the edge surrounding that do not end in a node belonging to the node surrounding. Fig. 6.3b depicts the best case of this optimization, where the location determined by the node surrounding completely replace the need for location given by the edge surrounding.

We can prove the correctness of this extension by extending Theorem 5 to take into account that: *(i)* for any edge (N, M) removed from the mirroring locations, M has been added to the mirroring locations; *(ii)* by the definition of mirroring location, mirroring on a node is equivalent to mirroring on all its outgoing edges; and *(iii)* by the definition of node surrounding, M is guaranteed to not be part of any forwarding path towards the prefix of the query, *i.e.,* M will mirror traffic only when it exits the confinement region.

Optimal solution. The surrounding algorithm further reduces the number of mirroring rules in the guaranteed absence of forwarding anomalies[3], that is, no blackholes and no forwarding loops within the monitored network.

We define a *mixed-egress path* for a region R as a simple path starting from a router in R, traversing at least one router outside R and ending in any egress point. Observe that in general, any packet exiting a region R either reaches an egress point (including those in R), or is dropped before. In the absence of forwarding anomalies, only the former case can happen. As such, the following theorem holds.

Theorem 6. *In the absence of forwarding anomalies, a* CONFINE *query on a region R can be correctly answered if and only if every mixed-egress path for R contains at least one mirroring location.* [4]

The following lemma proves Theorem 6.

Lemma 7. *In the absence of forwarding anomalies, any packet not confined to a region R is guaranteed to be mirrored if and only if every simple path starting from a node in R, traversing a node outside R and ending in any egress point crosses at least one active mirroring rule matching the packet destination.*

[3] This property can for example be checked by leveraging the results of other MIRROR queries given as input to our system.

[4] Note that by their definitions, the edge and node surroundings guarantee that the condition of Theorem 6 holds.

Proof. We separately prove sufficiency and necessity of the condition expressed by the theorem.

Sufficiency: Proof by contradiction. Assume that some packets not confined to R are not mirrored despite mirroring rules placed according to the condition in the statement. In the absence of forwarding anomalies, those packets are guaranteed to be delivered to an egress point. Not to be confined to R, they must follow a path $[\mathrm{R}_1 \ldots \mathrm{R}_n\ \mathrm{O}_1 \ldots \mathrm{O}_m \ldots \mathrm{E}]$ where E is an egress point, nodes $\mathrm{R}_i \in R\ \forall i = 1, \ldots, n$, and nodes $\mathrm{O}_j \notin R\ \forall j = 1, \ldots, l$. By hypothesis, an active mirroring rule must be on this path and must mirror the packets, contradicting the assumption that packets are not mirrored.

Necessity: Proof by contradiction. Assume that it is guaranteed to mirror all packets not confined to R but that there exists a path $P = [\mathrm{R}_m \ldots \mathrm{O} \ldots \mathrm{E}]$ which does not have an active mirroring rule, where E is an egress point, $\mathrm{R}_m \in R$, and $\mathrm{O} \notin R$ (possibly $\mathrm{O} = \mathrm{E}$). Consider now any path $[\mathrm{R}_1 \ldots \mathrm{R}_n\ \mathrm{R}_m]$ where $m \geq n$, and $r_i \in R\ \forall i = 1, \ldots, m$. This path must exist since a region is defined as a connected component. Packets forwarded on the concatenation of the previous two paths (*i.e.*,,$[\mathrm{R}_1 \ldots \mathrm{R}_n\ \mathrm{R}_m \ldots \mathrm{O} \ldots \mathrm{E}]$) are thus not confined to R as they cross $\mathrm{O} \notin R$. However, they are not mirrored as they exit the region through P, which contradicts the assumption. □

To illustrate this theorem, consider the example in Fig. 6.3c. If nodes P, H, and F mirror traffic, then no packet can exit the region [A B L C D] without traversing some mirroring location, or incurring a forwarding-anomaly—*e.g.*, looping on some routers to re-enter the region, or be incorrectly discarded by an internal router.

Algorithm. Determining the default set of mirroring locations in the presence and absence of interfering traffic mainly requires computing edge and node surroundings, respectively: both sets can be calculated by simply iterating over all the links of the input network.

In the absence of forwarding anomaly, the surrounding algorithm returns a minimal set of locations compliant with Theorem 6. To this end, it computes a set of nodes disconnecting the input region from every egress. This is a variant of the *minimal multi-terminals cut* problem. As the cardinality of the node surrounding provides an upper bound on the size of the cut to be computed, Stroboscope solves this variant in polynomial time using the algorithm described in [CLL09].

To further improve its efficiency, Stroboscope however computes a tighter upper bound by heuristically removing redundant elements from the node surrounding. First, it initializes the cut to the node surrounding. For every node N in the current cut, Stroboscope computes a simplified graph that does not include any node in the current cut except N, nor any link in the confinement region. For example, when considering router U in Fig. 6.3b, the algo-

rithm removes F, K, J, H, as they are in the node surrounding, and all the links in the region [A B L C D]. On this simplified graph, the algorithm computes the connected component including N—which is, [U C] in our example. If there is no path between any node in this component and an egress point (as it is for U in our example), it then indicates that all mixed-egress paths must include at least another router in the current cut. N is thus redundant, and can be safely removed from the current cut.

6.5 Computing measurement campaigns

The two previous steps of the compilation pipeline defined the amount of mirroring rules for each query (§6.4), as well the estimated traffic volume generated by the activation of one of those mirroring rule (§6.3). This section presents the last stage of the compilation pipeline, which schedules the queries over time to define a measurement campaign. More precisely, Stroboscope maximizes the accuracy of measurements across queries by activating them as much as possible, while also taking into account the traffic and time budget. We first describe how Stroboscope computes a mirroring schedule (§6.5.1), and then show how it adapts it at runtime (§6.5.2).

6.5.1 Scheduling mirroring rules

Answering monitoring queries requires to activate all its mirroring rules at the same time, in order to ensure sound measurements. As such, Stroboscope schedules mirroring rules on a per-query basis—*i.e.,* either all mirroring rules of a query are active at a given point in time, or none are. Any schedule computed by Stroboscope is composed of a finite number of timeslots, each containing one or more monitoring query. A timeslot represents a time interval, during which all of its associated queries are active. Timeslots do not overlap with each other.

Stroboscope ensures that activating all queries within a timeslot stays within the budget by assigning a *cost* to each of them. More precisely, this cost reflects the expected rate (*e.g.,* 5 Mb/s) of traffic mirrored when the query is active. On one hand, each MIRROR query has a cost corresponding to the number of mirroring rules it requires (§6.4.1), multiplied by the estimated traffic demand towards its prefix (§6.3.2). On the other hand, the cost of any CONFINE query is set to zero as they are not expected to mirror any traffic. Furthermore, mirroring rules used by CONFINE queries are heavily rate-limited, hence at most a few packets per mirroring location are mirrored in the worst case. As a result, Stroboscope always schedules CONFINE queries in all timeslots. Stroboscope's scheduling problem then consists in assigning MIRROR queries to every timeslot, so that the sum of the costs of all queries scheduled

in a timeslot does not exceed the traffic budget defined by the operator with the USING keyword.

Fig. 6.5 shows the different steps followed by Stroboscope to compute a measurement campaign. First, it derives the total number of timeslots, their duration and spacing from the monitoring budget defined in the requirements with the DURING keyword (§6.2.1) and latency estimates. Next, to scale to many queries and schedule sizes, Stroboscope splits the scheduling problem in two phases. It first computes a schedule of minimal duration, where each query is scheduled exactly once. Then, it maximizes the usage of the budget by replicating the minimal schedule as much as possible, hence maximizing the overall measurement accuracy. This scheduling pipeline is modular, and enables Stroboscope to switch between very fast but less optimized schedules (*e.g.,* to quickly add new queries to an existing schedule), and more optimized ones with a slower computation (*e.g.,* to maximize the measurement accuracy of long-lived measurement campaigns).

Timeslot duration and spacing. In-network latencies impose two constraints on the timeslots composing the schedule. First, the duration of timeslots has to be long enough to ensure that MIRROR queries can collect multiple copies of the same packet mirrored from different locations. This thus requires the timeslots to be greater than the maximal latency observed over paths monitored by MIRROR queries. This can either be configured statically, or dynamically estimated using results from prior measurement campaigns (§6.8). Note that the minimal timeslot duration is also constrained by the mirroring technique (§6.7.1). Second, to let in-flights packets arrive at the collector at the end of a timeslot, schedules generated by Stroboscope must include spacing between consecutive timeslots. This prevents mirrored traffic generated by different timeslots from overlapping, which would violate the traffic budget. We conservatively set this spacing to the maximum router-to-collector latency.

Minimal schedule extraction. Stroboscope solves its scheduling problem by first computing a minimal schedule. More precisely, it assigns exactly one timeslot to each query, with the goal of minimizing the number of timeslots. This process is a bin packing problem, and therefore NP-hard. To quickly approximate the schedule, Stroboscope first computes an upper bound on the size of the minimal schedule, using the well studied First-Fit-Decreasing heuristic—which has a worst-case time complexity of $O(n \log n)$ where n is the number of queries, and has been shown to approximate optimum solutions with a tight bound of $\sim 1.22\text{OPT}$ [JG85]. Computing a more optimized schedule can then be done by exploiting this upper bound to define an Integer Linear Program (ILP), visible in Alg. 2, which will compute the optimal bin-packing solution.

Note that if Stroboscope fails to compute such minimal sub-schedule, it

Algorithm 2 ILP formulation to compute the minimal subschedule

$$\text{minimize} \quad \sum_{s}^{S} U_s \tag{6.1a}$$

$$\text{subject to} \quad \sum_{q}^{Q} \left(R_{qs} a_q\right) \leq B U_s \qquad \forall s \in S, \tag{6.1b}$$

$$\sum_{s}^{S} R_{qs} = 1 \qquad \forall q \in Q, \tag{6.1c}$$

$$U_s \geq R_{qs} \qquad \forall q \in Q, \forall s \in S, \tag{6.1d}$$

$$U_s \leq S_{s'} \qquad \forall s \in S, \forall s' \in S, s < s' \tag{6.1e}$$

Objective Function. Minimize the length of the sub-schedule

Parameters.

Q The set of all input queries;

S The set of all timeslots;

B The maximal available bandwidth in a single timeslot;

a_q The cost of the query q.

Decision Variables.

R_{qs} Binary variable scheduling a query q in timeslot s when $R_{qs} = 1$;

U_s Binary variable tracking if the timeslot s has any assigned query.

Constraints.

(6.1b) The total query cost in any timeslot is at most B

(6.1c) All queries are scheduled once

(6.1d) Track the number of used timeslots

(6.1e) Open timeslots in sequence (tie-breaking constraint)

Algorithm 3 ILP formulation maximizing the monitoring budget utilization

$$\text{maximize} \quad \left[\sum_{q}^{Q} \left(\sum_{s}^{S} R_{qs} \right) w_q + M\Omega \right] \tag{6.2a}$$

$$\text{subject to} \quad \sum_{q}^{Q} \left(R_{qs} a_q \right) \leq \beta_s \qquad \forall s \in S, \tag{6.2b}$$

$$\sum_{s}^{S} R_{qs} > M \qquad \forall q \in Q \tag{6.2c}$$

Objective Function. Maximize the utilization of the budget, either by maximizing the number of allocations of some queries, according to their preference level, or by spreading the budget across all queries (thus maximizing the minimal allocation).

Parameters.

Q The set of all input queries whose cost enable them to be scheduled in at least one timeslot;

S The set of all timeslots having enough available bandwidth to be assigned at least one new query;

β_s The available bandwidth in the timeslot s, *i.e.*, $\beta_s \leq B$;

Ω Control whether all queries have a similar number of timeslots (high value) or the absolute number of allocation should be maximized (low value);

w_q The preference level of the query q;

a_q The cost of the query q.

Decision Variables.

R_{qs} Binary variable scheduling a query q in timeslot s when $R_{qs} = 1$;

M Continuous variable representing the minimal number of slots allocated to any query

Constraints.

(6.2b) The total query cost in any timeslot is at most β_s

(6.2c) Track the minimal number of timeslots assign for any query

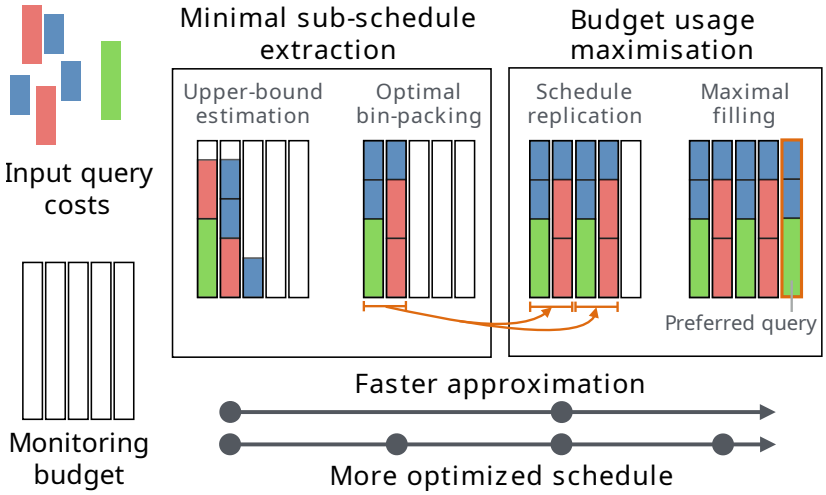


Figure 6.4: Stroboscope scheduling algorithm.

Figure 6.5: By first computing a minimal schedule to meet the monitoring budget and then expanding it to maximize the budget usage, the scheduling pipeline of Stroboscope is flexible and can be used to compute either quick approximations or optimized schedules.

then stops processing the monitoring requirements and reports an error to the operator as it implies that the monitoring budget is too small to fit all input queries.

Budget usage maximization. In the seconds phase, Stroboscope replicates the minimal schedule as much as possible. This increases the number of timeslots allocated to all queries uniformly, maximizing the overall measurement accuracy. This replication might not fully utilize the monitoring budget. For example, the schedule in Fig.6.5 encompasses 5 timeslots, which allows to replicate its minimal schedule at most twice while wasting 1 timeslot. This solution can be optimized further, filling the remaining timeslot(s), using another ILP (Alg. 3). More precisely, this last optimization makes a (configurable) trade-off between scheduling as many queries as possible using the remaining budget available in each timeslot and scheduling in priority queries which have been given a higher preference.

6.5.2 Adapting the schedule at runtime

The measurement campaign computed in the previous steps maximizes the utilization of the monitoring budget based on traffic estimates which can be wrong (*e.g.,* due to unpredictable traffic variations). While estimation errors

can balance across different prefixes, using a static schedule comes with the risk of mirroring much more traffic than the budget if the predictions are greatly underestimating the actual traffic volume for some prefixes.

Stroboscope tracks the total amount of traffic mirrored after every timeslot. By comparing it with the budget for 1 second (*e.g.,* 1 Gb if the budget is 1 Gbps), Stroboscope can then assess whether it is overflowing the budget. Whenever the total mirrored traffic exceeds the 1-second budget, Stroboscope stops the ongoing measurement campaign, waits for the remaining time in the 1-second interval while computing a new schedule using the update traffic estimates, and finally runs a new measurement campaign. For example, if it detects that 1.1 Gb of traffic have been mirrored in 0.7 seconds, for queries with a budget of 1 Gb/s, Stroboscope immediately stops the measurement campaign, waits for 0.3 seconds, and then starts a new campaign—which likely reduced the amount of times the queries were scheduled to account for the previous budget overflow.

As the inter-timeslot spacing ensures that the collector receives all the mirrored packets before starting the next measurements (§6.5.1)), the runtime behavior just described yields the following property.

Property 2. *Stroboscope exceeds the budget in any query for at most 1 timeslot per measurement campaign.*

Note that traffic estimates are updated after the stopped campaign, so the successive campaign is much more likely not to exceed the budget again.

6.6 Deploying Stroboscope in real networks

We built a complete prototype of Stroboscope in $\sim$ 5,000 lines of Python code, and 650 of C code. Our implementation covers the entire compilation pipeline along with the logic to trigger mirroring rules on routers (Cisco or Linux-based), as well as the benchmarks used in the evaluation (§6.7).

As the previous sections extensively described the various needed to build a Stroboscope collector and also implement its compilation pipeline, this section focuses one the last piece of the framework: the collection of traffic slices. We first present the general mechanisms used to mirror specific packets on routers, as well as how these are processed by the Stroboscope collector (§6.6.1). Then, we detail the two approaches supported by Stroboscope to execute mirroring actions (*i.e.,* mirroring rules (de-)activations), showcasing their respective strengths and weaknesses (§6.6.2).

6.6.1 Mirroring packets

Most commercial routers support packet mirroring capabilities [Cis16; Jun14]. At a high-level, such capabilities work in two steps, more-or-less flexibles depending on the router vendor, OS version, and hardware components. First, they enable operators to specify how to identify packets that should be duplicated, *e.g.,* using route-maps. Then, as packets matching the mirroring criteria are duplicated, they let operators specify how these duplicates should be handled: sent as-is over a given interface (*e.g.,* SPAN), sent towards a virtual interface designating a tunnel (*e.g.,* GRE), or delegated to a vendor-specific protocol implementation (*e.g.,* ERSPAN).

Packet mirroring primitives are typically implemented in hardware. This enables them to work at scale, able to duplicate traffic at line rate without degrading the forwarding performance of the routers, *i.e.,* with no effect on the packets being mirrored [Vis+10]. A naive approach to implement Stroboscope would thus consists on dynamically programming the mirroring criterias, in order to temporarily match specific flows hence create traffic slices. Unfortunately, most routers only support a limited amount of mirroring criterias (*e.g.,* 2 on our routers) that can be handled in hardware at the same time. Such a naive approach would thus be unable to mirror more than 2 flows at the same time on any router, hence be heavily limited both on the number of queries it could answer and on the accuracy of the measurements—*i.e.,* the location of mirroring rules would then become the primary bottleneck, instead of the monitoring budget.

To overcome this limitation, Stroboscope trigger mirroring actions in two steps. First, packet mirroring primitives are pre-configured to be always enabled, mirroring any packet carrying a mirroring tag (*e.g.,* a specific ingress VLAN, DSCP value, or firewall mark). Generating a traffic slice for a given flow can then be performed indirectly (*i.e.,* without reconfiguring the mirroring primitive) by applying the proper tag to its packets (§6.6.2). We use different mirroring tag values, in order to handle differently packets answering a MIRROR query from those answering a CONFINE query. Indeed, as CONFINE queries only need a few packet to report a violation (and no packets in the normal case), we heavily rate-limit them to minimize the risk of overflowing the traffic budget during unexpected traffic shifts.

Once created, mirrored packets are then forwarded towards the Stroboscope collector, alongside regular traffic. Despite Stroboscope's design revolving around adhering to a traffic budget, mirrored packets should never be prioritized over regular traffic to minimize the risk of congestion induced by the measurements. For each received mirrored packet, the Stroboscope collector then extracts: *(i)* the router ID that mirrored the packet (*e.g.,* based on the outer source IP address if using a GRE encapsulation); *(ii)* the destination ad-

dress of the mirrored packet (*e.g.,* the inner destination address); and *(iii)* the arrival NIC timestamp. Based on this 3-tuple, Stroboscope is then able to identify which query caused this packet to be mirrored. At the end of the timeslot, Stroboscope then walks through its scheduled queries, analyzing their traffic slices (*e.g.,* to find matching packets enabling to reconstruct forwarding paths) and updating the traffic statistics. Eventually, Stroboscope streams the collected mirrored packets with all their meta-data (*e.g.,* matches across traffic slices, corresponding query, timestamp) to external monitoring applications which can then post-process them—*e.g.,* build time-series, inspect packet payloads, or estimate latencies.

6.6.2 Carrying out mirroring actions

As described earlier (§6.6.1), Stroboscope triggers mirroring actions by applying (or removing) a mirroring tag to packets. Stroboscope comes with two approaches to achieve this.

Tagging packets by their ingress interface. VLANs encapsulate packets such that packets sent over the same physical link but using different VLAN tags are processed by routers as if they were coming from different layer-3 domains, *i.e.,* VLANs create multiple virtual interfaces multiplexed on a single physical interface. We can use VLANs as a way to encode multiple bits of information, signaling whether a packet should be mirrored or not, and by which type of query (CONFINE or MIRROR).

We configure two additional VLANs on every link in the network, and advertize them in the IGP with a high metric to ensure that are unused by default. Mirroring rules on all routers are then set to mirror all packets arriving at a router from one of these VLANs. Assume that we have packets towards a destination d crossing a link a from a router X to a router Y, which we want to mirror at Y. We then use Fibbing (§II) to start (resp. stop) mirroring these packets by re-routing X to forward d over the mirroring VLAN. The duration of the traffic slice is then determined by the duration between the activation and deactivation LSAs sent by the Fibbing controller. This approach has two main advantages: *(i)* the collector can activate up to 40 mirroring rules for a given query using a single IGP message (§4.3.2), enabling it to quickly (de-)activate all mirroring rules within a timeslot; *(ii)* our measurements confirmed that we could precisely control the slice duration, capturing traffic slices as small as 14 ms (§6.7). Unfortunately, this approach suffers from two limitations. First, as it relies on the IGP to activate mirroring, it can thus only activate mirroring on a per-destination basis[5]. Second, this approach implicitly relies on the correctness of the control-plane. Activating mirroring by instructing around

[5] All our algorithms could be applied as-is to more precise types of flows, *i.e.,* mirroring based on 5-tuples instead of destination prefixes

the target mirroring location to re-route traffic is only correct if the traffic was already flowing through the underlying physical link, or if the node re-routing the traffic was seeing no traffic at all for that prefix (*e.g.,* in the case of a CONFINE query). These conditions are challenging to verify in practice, and constraint the type of measurements achievable by Stroboscope.

Tagging packets using a local-agent. A second, safer, approach consist in instructing a local agent on the routers to (de-)activate mirroring by using the DSCP value of packets as mirroring tag. Indeed, routers often support scripting interfaces [Cisc; Jund], or event fully-fledged SDKs [KAB09], enabling to provision custom configuration actions or daemons on the routers. Stroboscope leverages this by uploading on every router during a configuration phase. Then, at runtime, the Stroboscope collector maintains SSH sessions with each router, enabling it to communicate with every agent. When it wants to update the set of active mirroring rules on a given router (*i.e.,* at the start of a timeslot), the collector then sends it an activation message containing: *(i)* the list of flows that should be mirrored (and which interfaces in the case of mirroring rules located on edges); and *(ii)* the duration of the timeslot. This local agent then updates an access-list (ACL), in order to change the DSCP value of packets that should be mirrored. Then, it sets a timer corresponding to the input duration. At the expiration of this timer, our agent then clears the ACL to stop all mirroring rules. Beside avoiding the drawbacks of the previous approach, this technique further reduces the number of control-plane mesages (*i.e.,* only one per timeslot, per router), and let routers autonomously deactivate their mirroring rules. This last behavior provides a strong guarantee that mirroring rules will never be active longer than a timeslot—*e.g.,* the collector crashing or becoming unreachable will not result in a complete network meltdown.

While other techniques can be used to remotely configure an ACL (*e.g.,* BGP Flowspec [RFC5575], or Netconf [RFC6241]), keeping the added flexibility of maching packets using ACLs, these do not provide the same safety guarantees as they would require the collector to deactivate the mirroring rules.

6.7 Evaluation

This sections presents results obtained when submitting the key components of Stroboscope to various benchmarks. More precisely, we first confirm the practicality of our approach using measurements performed on real routers (§6.7.1). These measurements show that our implementation can activate numerous mirroring rules in a short amount of time (*e.g.,*, consistently with [BDL04]), and that it could precisely control the traffic slice durations. Then, we demonstrate that Stroboscope's entire compilation pipeline (*i.e.,* the

placement algorithms (§6.7.2) and the scheduling pipelines (§6.7.3)) scales to large numbers of queries. More precisely, we ran synthetic benchmarks on realistic ISP topologies to demonstrate that: *(i)* measurement campaigns can be computed at a timescale suitable for online use; and *(ii)* the accuracy of each query can be efficiently maximized.

6.7.1 Real routers natively support traffic slicing

We performed experiments using two physical routers (Cisco C7018) to asses whether we could run Stroboscope at scale. To that end, we directly connected these routers and configured one of them to mirror incoming packets, sending the copies to the collector, while the other one was connected to a traffic generator.

Controlling slices duration. We first measured the level of control that Stroboscope has over the duration of the traffic slices it collects. To that end, we successively tried to collect traffic slice of increasing length, and measured how long was the resulting stream of mirrored packets. Fig. 6.6 shows the measured duration of the traffic slices when using the local agent to trigger mirroring, depending on the specified deactivation delay. Each experiment was repeated 50 times. We see that the slice durations increase linearly with the delay specified to the agent, and that the minimal slice duration is 24 ms. Repeating the experiment using the other approaches resulted in similar measurements, although the minimum achievable traffic slice in that case was 14 ms.

Mirroring processing delay. Additionally, we also measured the time needed by routers to mirror packets. To that end, we directly connected the Stroboscope collector to the router mirroring the traffic, and used the traffic generator to send traffic to the collector. We then computed the delay between the arrival time of every packet and their mirrored copy. The mean mirroring delay over roughly 100 000 measurements was $\mu = 2.6\,\mu s$, with a standard deviation of $\sigma = 1.6\,\mu s$. Such small values hint that our routers mirror packets in constant time.

6.7.2 Optimizing mirroring locations is fast and efficient

We benchmarked our placement algorithms on all Rocketfuel topologies [SMW02] and on the largest topologies from the Internet Topology Zoo [Kni+11]. We performed more than 4, 000 experiments for each algorithm. We stress that the speed of these algorithms directly is key to enable to adapt Stroboscope's measurement campaigns at runtime (*e.g.,* to react to routing changes), as each query requires to run one of them once. Additionally, their ability to minimize the overall number of mirroring rule is critical

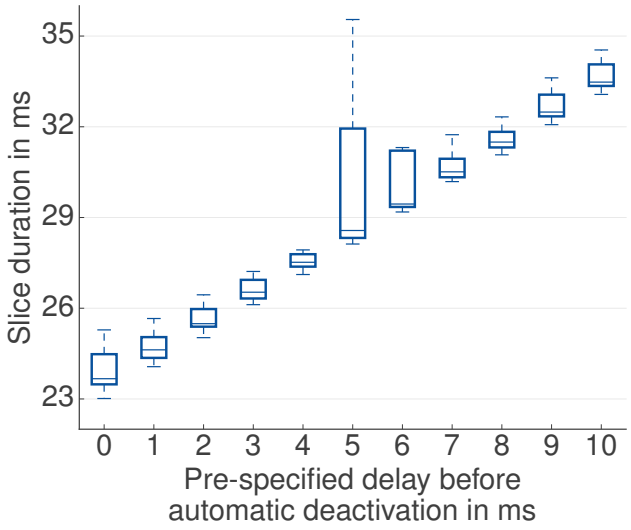


Figure 6.6: Local-agent running on Cisco routers (C7018) precisely control the traffic slice duration, and are able to collect slices as small as 24 ms.

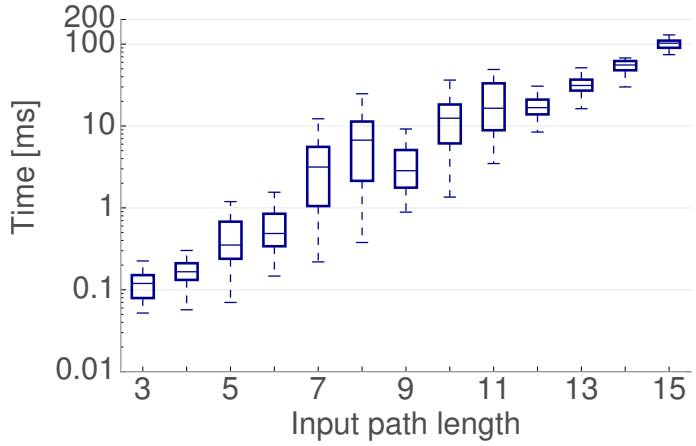
to allocate more timeslots per query, hence to maximize their measurements accuracy.

Key-points sampling (§6.4.1). We evaluate the algorithm by running it on random shortest-paths (according to the IGP weights for the Rocketfuel topologies, and edge count on the Topology Zoo ones), and random deviations from these (*i.e.*,, paths longer by up to 50% with the same end points).

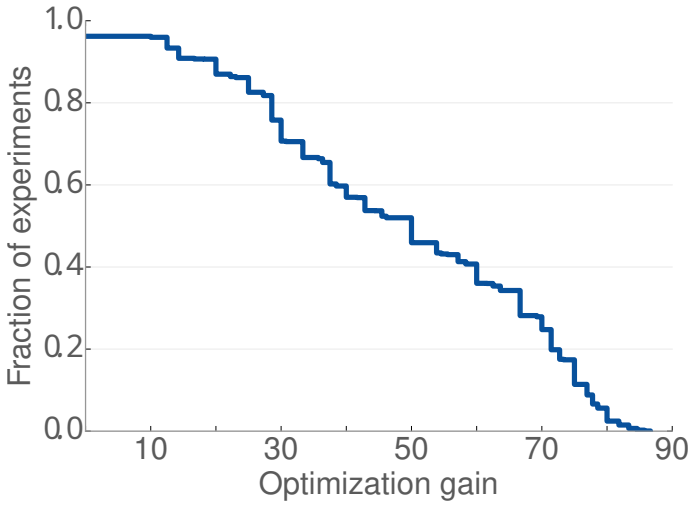
Fig. 6.7a shows box plots of the measured execution time in function of the input path lengths. As expected, the algorithm exhibits an exponential behavior, yet still completes in milliseconds for path of realistic lengths. The CDF on Fig. 6.7b quantifies the gain brought by KPS in terms of mirroring-rule reduction. More precisely, we compute the optimization by comparing the number of mirroring rules before running KPS (*i.e.*, one rule per hop) to the one after executing KPS. We see that $\sim$ 80% of the experiments resulted in a gain of over 30%. KPS returned only 2 to 4 mirroring rules in 64% of the experiments.

Surrounding algorithm (§6.4.2). Similarly, we benchmarked the surrounding algorithm using random connected components as confinement regions. Additionally, we randomly selected 25% of the nodes having 2 or less outgoing edges as egress points (to ensure that the minimal surrounding had to compute a non-trivial solution).

Fig. 6.8a shows the measured execution times, in function of the region size. As the node and edge surrounding are almost the same algorithms (*i.e.*,

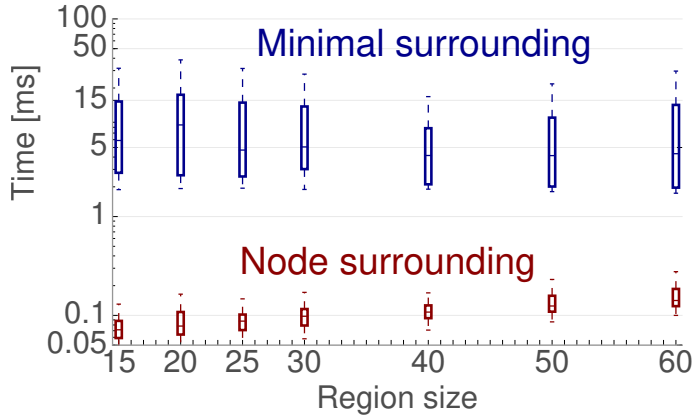


(a) KPS runs quickly even for longer input paths in large graphs.

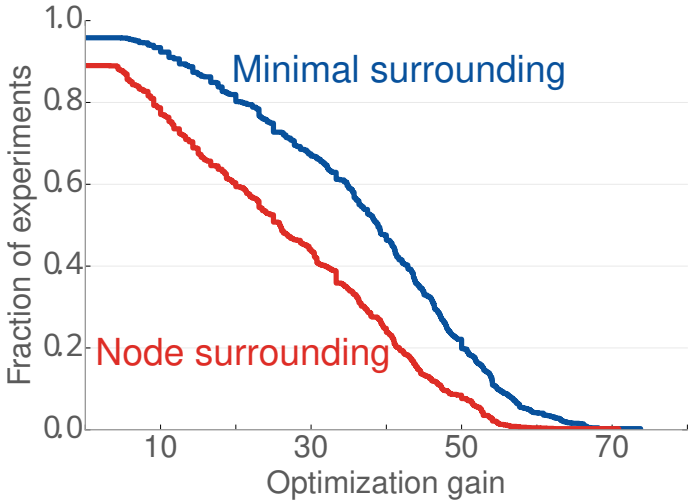


(b) KPS reduced the number of mirroring rules in all experiments.

Figure 6.7: The Key-points Sampling algorithm minimizes the number of mirroring rules in milliseconds, reducing their overall cost.



(a) The running times of the surrounding alogrithm depend on the size of the graph and not the input region.



(b) The surrounding algorithm drastically reduces the number of mirroring rules.

Figure 6.8: By quickly minimizing the number of mirroring rules, the surrounding algorithm reduces the control-plane overhead.

the node surrounding filters the results from the edge surrounding), we only reports the results of the node surrounding. We observe that computing the node surrounding runs in hundreds of microseconds, and is an order of magnitude faster than the further optimized placement. Additionally, execution times do not depend on the input regions but instead on the network size and its average node degree. Fig. 6.8b shows the reduction in the number of mirroring rules compared to the edge surrounding. Both algorithms reduce the number of mirroring locations by at least 30% in half of the experiments, and the optimal one can provide an extra gain of 20%.

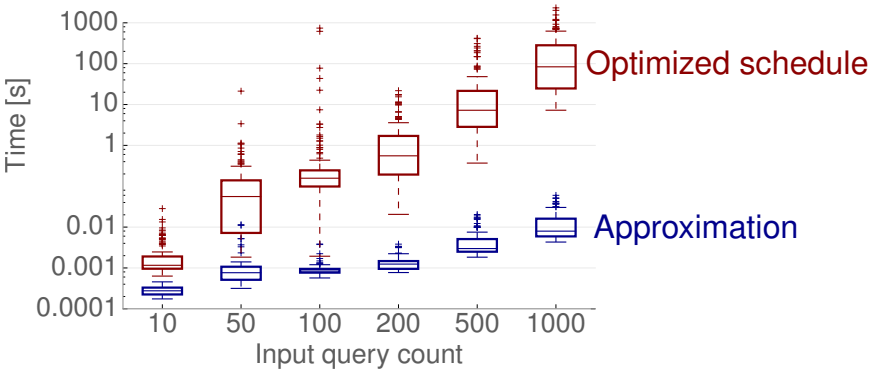
6.7.3 The scheduling pipeline is flexible

We conclude our evaluation by demonstrating the scalability and efficiency of our scheduling pipeline. To that end, we schedule an increasing number of queries using both the more optimized scheduling pipeline and its approximation. All queries have a random cost, normally distributed. Additionally, we varied the monitoring budgets used such that each schedule has between 20 and 400 timeslots, with a maximal bandwidth set between 2 to 100 times the average query cost.

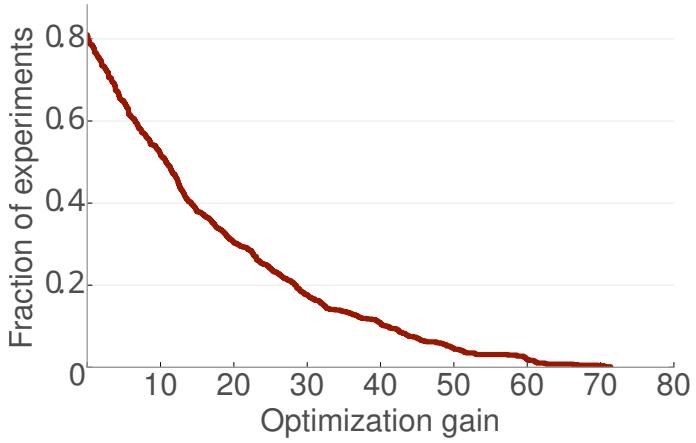
Fig. 6.9a shows the running times of the optimized scheduling algorithm and of its approximation. We confirm that the approximated schedule can be used to react to online events, as it is computed in microseconds. The large variance of the optimized pipeline is due to the variation of the maximal bandwidth usage across experiments. If this value is low, it increases the estimated upper bound for the bin-packing problem, which makes computing an optimized schedule exponentially slower. The optimized schedule, however, leads to improved accuracy. Indeed, Fig. 6.9b depicts the CDF of the relative increase of slot allocation (scheduling a query in a timeslot), when using the optimized pipeline instead of the approximation. For about half of the experiments the optimized schedule contains 15% more slot allocations than the approximated one, up to 40% for 10% of the experiments.

6.8 Using Stroboscope in practice

This section presents our experience when using Stroboscope in networks emulated using Mininet [LHM10]. We first describe monitoring applications built on top of Stroboscope's measurement stream, enabling to estimate loss rates, load-balancing ratios and one-way delay in a transit network (§6.8.1). Then, we report Stroboscope's reaction when facing an unexpected traffic increase, experimentally validating its ability to adapt its schedule at runtime (§6.8.2).



(a) The scheduling pipeline can quickly compute a measurement campaign, enabling to adapt it online, even on larger inputs.



(b) Optimizing the schedule maximizes the measurement campaign accuracy by increasing the number of timeslots allocated to all queries.

Figure 6.9: Stroboscope can compute a quick approximated schedule, or one that maximizes accuracy.

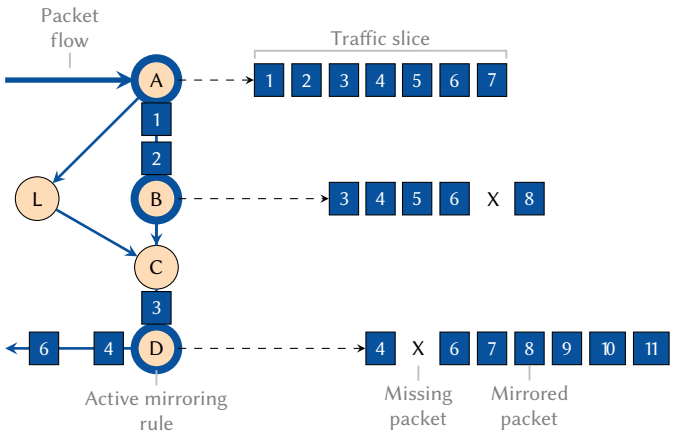


Figure 6.10: Identifying packets present in multiple traffic slices enables to reason about network-wide behaviors.

6.8.1 Analyzing Stroboscope's measurements

We emulated the network shown in Fig. 6.2a, and attached a Stroboscope collector to router U. Then, we configured it using the queries visible on the top of Fig. 6.2b. Finally, we configured all links to have a delay of 5 ms and a loss probability of 1%.

We now present three applications built on top of Stroboscope's measurements, which analyze the content of traffic slices.

Estimating loss rates. Stroboscope estimates losses over paths by combining MIRROR and CONFINE queries. Indeed, there are only three possible reasons causing a packet mirrored at the ingress of a path (A) to not have a matching copy at the egress (D): *(i)* the timeslot completed before the packet reached the egress, which only happens if no packet afterwards is seen at both A and D; *(ii)* The CONFINE query detected a violation; or *(iii)* the packet was dropped.

Consider the example traffic slices visible on Fig. 6.10, which are collected to answer the MIRROR queries from Fig. 6.2b. We see that packet "5" was found in the traffic slices collected by A and b, but was never mirrored by D. Assuming the CONFINE queries did mirror any packet either, we can then conclude that packet "5" was lost between routers B and D.

In our emulated network, we measured the rate on the path [A ⟶ D] to be 7%—slightly higher than the real value (5%) as some mirrored packets were also dropped by the network.

Estimating load-balancing ratios. Most networks load-balance traffic across similar paths by enabling ECMP on the routers. Whenever a router has

multiple possible equal-cost paths for a packet, it then hashes its header to select a nexthop. ECMP hash function polarization [Cis13] causes suboptimal network usage as it causes routers to only use a subset of the available equal-cost paths. It is also hard to detect in practice. We confirmed that Stroboscope can easily detect such issues by computing perceived load-balancing ratios. More specifically, the collector computes the ratio of matching packets seen on all hops of a path (*e.g.,* {A, B, D}), over those seen only at both endpoints (*e.g.,* {A, D}). This ratio can then be compared against the assumed number of equal-cost paths between the two endpoints to detect whether hash function polarization is occurring (*e.g.,* it should be close to 50% if there are two paths).

Coming back to the example on Fig. 6.10, we see that packet "7" was only seen in the traffic slices of A and D. While this indicates that the packet was successfully forwarded across the network, it also means that it followed another path than the expected one (*e.g.,* it might have transited through L).

In our emulated network, the monitored prefix had a single flow. This caused the computed ratio to be about 90% (recall that there are losses in the network. This unusual ratio should prompt the operator to observe more closely the captured packet headers across the different traffic slices.

Estimating one-way delays. Stroboscope can estimate one-way delays in a network without any form of clock synchronization between the routers and the collector. Recall that beside the mirrored packets, Stroboscope's measurement stream also add meta-data, such as the time at which each packet was received by the collected. Our third monitoring applications leverages this timestamp to estimate one-way delays in the network in two steps.

First, Stroboscope estimates router-to-collector latencies. To that end, it activates on each router a mirroring rule matching the router's loopback address. The collector then sends probes towards these loopbacks, and receives mirrored copies echoed by the NIC (*i.e.,* without any intervention from the routers' CPU). Assuming that paths between the controller and the routers are symmetrical[6], Stroboscope then computes router-to-collector latencies by comparing the time at which each probe was sent to the time at which its mirrored copy was received.

Second, Stroboscope estimates one-way delays over any given paths (*e.g.,* [A $\longrightarrow$ D]) by using measurements from a MIRROR query in three steps. First, Stroboscope builds a list of packets present in the traffic slices of both endpoints. Then, for each such matching packet, it infers the time at which this packet traversed each router by subtracting the router-to-collector latency from the time at which the mirrored copy was received at the collector. Fi-

[6]For practical reasons, the loopback addresses used for this should be dedicated to this monitoring application. As such, we can use Fibbing to re-route these specific addresses if the default routes are not symmetrical.

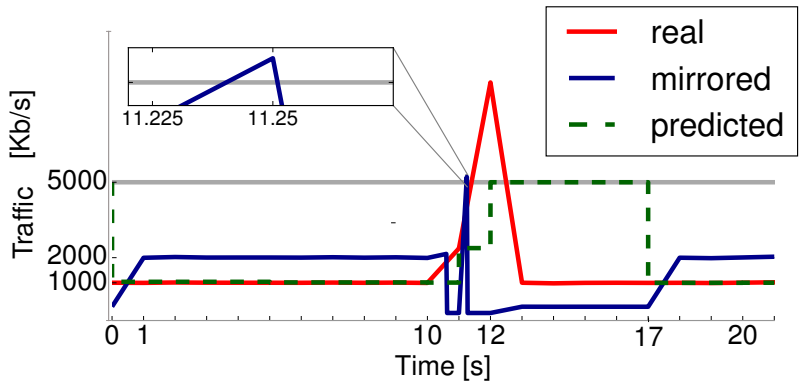


Figure 6.11: Stroboscope dynamically estimates traffic demands and swiftly reacts upon budget violation.

nally, Stroboscope can then compute the difference between these traversal times.

Using this procedure, our monitoring application measured that the latency of [A ⟶ D] in the emulated network was 15 ms, as expected.

6.8.2 Reaction to unexpected traffic volume

Measurement campaigns are computed to ensure that they adhere to the monitoring budget. Guaranteeing this in practice requires runtime mechanisms which measure the current budget usage and adjust the measurement campaign accordingly (§6.5.2). In a small emulated network, we configured Stroboscope to mirror a flow of 1 Mb/s at two locations, using at most 5 Mb/s. We then started a traffic generator for that flow, and configured it to generate a short-lived large of traffic after a few seconds. We configured Stroboscope to only take into account the last 5s of traffic when estimating traffic demands (to speed up the experiment), and to use 40 timeslots of 25ms.

Fig. 6.11 shows the evolution of *(i)* the configured rate of the generated traffic; *(ii)* the predicted traffic demand as computed by Stroboscope; and *(iii)* the rate of mirrored traffic. Initially, the prediction starts at the budget value, causing little mirrored traffic. After 1 s, the prediction is updated to reflect the last observed peak demand. This increases the amount of mirrored traffic as the query is scheduled more often (in all timeslots in this case). At $t = 10\,s$, the real traffic volume spikes, increasing the mirrored traffic. Eventually, the mirrored traffic exceeds the predicted volume, and the measurement campaign is then interrupted. A new schedule, with updated traffic statistics is then started at $t = 11\,s$. However, as the generated traffic volume kept in-

creasing, this causes the query to immediately exceed the budget in a single timeslot, halting the measurement campaign again. This causes the traffic estimation from $t = 12\,s$ to $t = 17s$ to be equal to the whole budget, hence scheduling the query in a single timeslot. At $t = 17\,s$, the last peak traffic rate is expunged from the measurement database. This causes Stroboscope to update its prediction for the monitored flow, scheduling the query in all timeslots. In total, the mirrored traffic exceeded the budget only during a single timeslot (25 ms).

6.9 Related Work

Network monitoring is an age-old topic, and has been extensively researched. Stroboscope can relate to many previous work, which we can organize in six threads of work.

Stream-based monitoring. One of Stroboscope's goal is to stream measurements collected at different vantage points to monitoring applications. This relates to Gigascope [Cra+03], a system providing a SQL-like query language to stream packet-based measurements from any router interface. Stroboscope goes beyond this by providing higher-level constructs such as path-based queries and the ability to adhere to a monitoring budget, enabling to coordinate measurements across multiple devices. Furthermore, Gigascope supports fewer concurrent queries as changing the packet dissectors they execute on the routers is slow.

Mirroring-based monitoring. Other approaches have been explored to monitor networks using packet mirroring, often for a specific purpose—*e.g.,* [Vis+10] relies on packet mirroring to selectively monitor control-plane traffic. Table 6.1 compares Stroboscope to the two closest mirroring-based systems, namely Everflow [Zhu+15] and Planck [Ras+14]. Stroboscope is the only approach that automatically controls the traffic volume generated by its measurement, making it possible to deploe in ISP networks.

Monitoring with programmable hardware. Programmable hardware (e.g., P4 [Bos+14], OpenFlow [McK+08]) and virtual network devices (e.g., Open vSwitch [Pfa+15]) enable to program new monitoring capabilities on-demand, directly in the network. For example, path queries can be performed by encoding the path traversed by packets in the packets header [Kaz+13; Nar+16]. Narayana et al. [Nar+17] introduce a performance query language, Marple, interacting with a key-value store running on the switches. SketchVisor [Hua+17] leverages virtual switches to sketch the traffic. Basat et al. [Ben+17] present a randomized constant time algorithm to identify hierarchical heavy hitters. NetQRE [Yua+17] uses regular expressions over packet

Feature	Stroboscope	Everflow [Zhu+15]	Planck [Ras+14]
Query-based mirroring	✓	✓	✗
Monitoring on a budget	✓	✗	✗
Runs on commodity hardware	✓	✓	✓
Independence from active probing	✓	✗	✓
Independence from header bits	✓	✗	✓

Table 6.1: Stroboscopeis the only mirroring-based monitoring system which is able to adhere to a budget.

streams to express flow-level and application-level policies. As Stroboscope exposes the mirrored packets to monitoring applications, the algorithms and functionalities of these approaches could directly be built on top of Stroboscope's measurements.

Monitoring flow statistics. Many techniques compute aggregated statistics directly inside the network before exporting them to a collector. Traditionally used in ISP setups, NetFlow [RFC3954] provide coarse-grained flow statistics by randomly sampling traffic. More recently, FlowRadar [Li+16] and ProgME [YCM11] provide per-flow packet counters. While these approaches implicitly limit their monitoring overhead, they lack the capability of Stroboscope to track individual packets across the network, hence cannot reason about network-wide behaviors (*e.g.,* one-way delays, path tracing).

Data-center monitoring. Data-center network offer additional degrees of freedom compared to ISP networks, such as controlling the end hosts or almost non-existent latencies. As such, many research contributions providing fined-grained visibility over data-center traffic cannot be applied in ISPs. For example, [Mos+16] collects fine-grained statistics on the end hosts, [Guo+15] leverages pings across end-hosts to estimate end-to-end latencies. In contrast, Stroboscope is a more general in-network solution, applicable in any network. Being modular, some of its building blocks could also be transferred to other systems, *e.g.,* it could be used to enable Everflow [Zhu+15] to control its amount of mirrored traffic.

Network Verification. Verifying network correctness is a very active research domain. More precisely, both the data-plane verification techniques [Mai+11; KVM12; Kaz+13; Khu+13; Lop+15; Sto+16] and the control-plane ones [Fog+15; Gem+16; Wei+16; Bec+17] build a model of the network on which they can then verify correctness properties. Stroboscope complements these approaches by enabling them to dynamically collect traffic sam-

ples, at controlled locations and times (*e.g.,* to verify assumptions made by the verification model). Similarly, Stroboscope can also be used as a complementary source of data for network debugging tools in SDNs [Wun+11; Han+14].

Observing protocol performance on the end hosts

7

This chapter presents Flowcorder, an enterprise network monitoring framework to measure the network performance experienced by the end-hosts. We begin by analyzing current enterprise network monitoring approaches, formulating where they fall short and why they cannot work with recent transport protocols (§7.1). Then, we describe how Flowcorder addresses the limitations of previous approaches, by instrumenting the transport protocol implementation on the end-hosts (§7.2). Flowcorder follows a generic approach to export performance profile of connections by transparently extracting Key Performance Indicators from existing protocol implementations (§7.3). We describe an application of our approach which realizes an event-based instrumentation of the Linux kernel TCP stack (§7.4). Next, we demonstrate the generality of the approach by extending it to support MPTCP (§7.5). Afterwards, we report the results of an evaluation which shows that Flowcorder imposes little to no overhead on the end-hosts (§7.6). We conclude by highlighting insights provided by Flowcorder when deployed in a campus network (§7.7), and finally compare Flowcorder to other approaches to which it can relate (§7.8).

7.1 Motivation

Network performance depends on a variety of factors such as link delays and bandwidth, router buffers, routing or transport protocols. Some of these are controlled by the network operators, others by the end-hosts. To detect potential issues, and ensure their proper operations, most network operators monitor a wide range of statistics on the health of their networks, which can be classified in three categories.

First, *health metrics* capture the status of network elements. Most networks record those using SNMP, polling their devices every few minutes to collect various statistics (*e.g.,* link load, CPU usage, size of forwarding tables). Operators often also collect *traffic statistics*, usually using NetFlow/IPFIX [San15;

Hof+14; TB11]. These provide more detailed information about the flows crossing the network (*e.g.,* layer-4 5-tuples, volumes in bytes and packet), and enable various management applications [Li+13] (*e.g.,* identifying heavy-hitters [GSF13], major source/destination pairs [YRW17], or detecting DDoS attacks [Sek+06; Ste+15]). Finally, operators monitor *key performance metrics* which are important for many end-to-end applications, such as delays, packet losses, and retransmissions. On one hand, active measurements techniques [Cisa; LCC09] collect these metrics by generating test traffic (*e.g.,* pings). On the other hand, passive measurements [Fin+11; JTO10] infer these performance metrics by analyzing the packets that traverse the network (*e.g.,* using network taps which maintain per-flow states to accurately measure Round-Trip-Times (RTT), retransmissions, packet losses and duplications [Mel02]).

Although widely deployed, passive monitoring suffers from several important limitations. First, as link speeds increase, it becomes more and more difficult to maintain the per-flow state that is required to collect detailed performance metrics [Tre+17]. Second, as multipath protocol deployment increases (*e.g.,* MPTCP [RFC6824] is used in iPhones [App17] and for other services [BS16b]), passive monitors only see a subset of the packets belonging to a connection. This compromises their ability to operate properly [Pea14]. Finally, the most important threat against the passive collection of network performance metrics is the deployment of encrypted protocols, such as QUIC [Lan+17]. QUIC replaces the HTTP/TLS/TCP stack with a simpler protocol that runs over UDP. Google estimates [Lan+17] that QUIC already represents more than 7% of the total Internet traffic. Recent measurements indicate that content providers have started to deploy QUIC massively [Rüt+18]. The IETF is currently finalizing a standardized version of QUIC [IT18]. From a performance monitoring viewpoint, an important feature of QUIC is that all the payload and most of the header of the packets are encrypted. This prevents the middlebox ossification problems that affect protocols such as TCP [Hon+11; Pap+17], but it also greatly decreases the ability for network operators to monitor network performance. This prompted some of them to ask to modify QUIC to be able to extract performance information from its headers [Emi+17]. The IETF answered those operational concerns by reserving one bit in the QUIC header (the spin-bit [Tra+18]), exposing limited delay information. Furthermore, multipath extensions to QUIC have already been proposed [Vie+18; DB17].

Losing the ability to monitor the performance of connection is problematic painful for enterprise network operators. Indeed, as they have little to no knowledge about the state of their ISP network, troubleshooting connectivity issues requires them to passively infer the performance of the network. More precisely, if a user reports a partial outage, operators have to: *(i)* identify the

set of flows affected; *(ii)* identify which part of the enterprise network is affected (*e.g.,* which uplink if the network is multihomed); and *(iii)* characterize the current and past performance of the affected services. Such information is useful for more than troubleshooting. Indeed, collecting performance metrics enable enterprise to evaluate the quality of the connection across providers, verify SLAs, or efficiently provision their network. If enterprise network operators want to continue to collect performance metrics on the end-to-end flows of their users, they need a different approach than passive monitoring to be future proof.

Problem statement. *How can we support the legitimate need of fine-grained performance information from enterprise network operators in presence of encrypted, multipath protocols?*

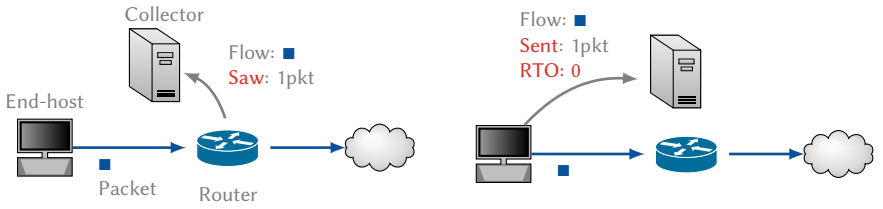
Key challenges. Designing a monitoring framework that answers this question raises at least four challenges. First, this framework must accurately depict the performance experienced by the end-hosts. This limits the applicability of active measurements, as this might hide issues specific to the used protocol (*e.g.,* TCP RTO). Second, it must support multipath protocols, and thus monitor the performance of all paths used by a given connection. This limits the possibility of using passive monitoring since this would require coordination among the monitors located on different paths. Third, supporting encrypted protocols prohibits such framework from analyzing packet headers or contents and prohibits the utilization of "transparent" proxies. Finally, it should operate with a low overhead, limiting the generated statistics to the minimum to establish a baseline for normal operation, while also enabling to quickly capture and detect performance issues.

Flowcorder. We introduce Flowcorder, a novel enterprise network monitoring framework which addresses the above challenges. The key insight behind Flowcorder is to leverage the per-connection information that is already maintained by the end-hosts themselves.

Instrumenting the transport stacks of the end-hosts enables Flowcorder to compute Key performance Indicators (KPIs) for each connection. By capturing such KPIs at specific moments of the connection life-cycle, Flowcorder can then build *performance profiles* of connections. Finally, Flowcorder aggregates those profiles and exports them over IPFIX, integrating with existing monitoring infrastructure and enabling analyzes across hosts, protocols, remote services and/or ISPs.

7.2 Flowcorder

Many networks monitor their traffic using in-network appliances that inspect packets crossing them, and eventually export statistics to measurement col-



(a) Most networks are monitored using passive measurements made by the network devices.

(b) With Flowcorder, end-hosts themselves export connection performance statistics.

Figure 7.1: Flowcorder enhances network monitoring with fine-grained measurements about connections performance as experienced by the end-hosts.

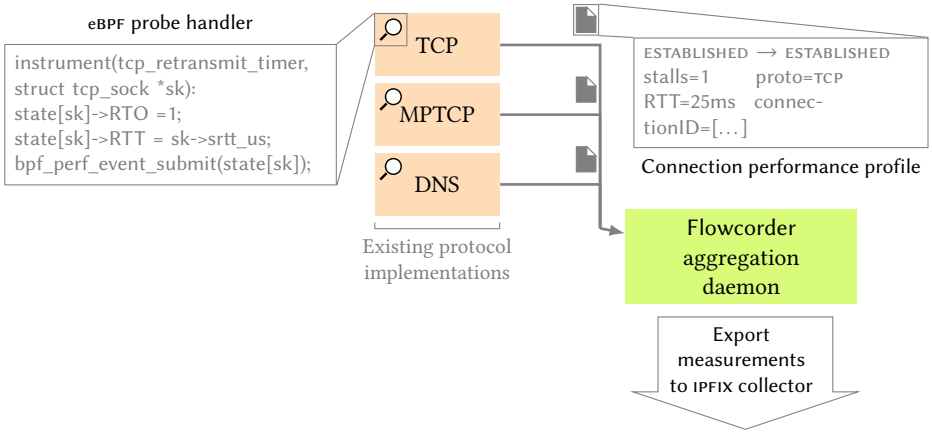


Figure 7.2: Flowcorder enables to evaluate network performance from generic Key Performance Indicators collected on the end-hosts for every connection.

lectors using a protocol such as IPFIX (Fig. 7.1a). While sufficient to track traffic demands, or collect rough traffic statistics through passive inference of the connection states, these techniques hardly scale if the operators requires fine-grained performance measurements on a per-connection basis. Flowcorder instead pushes the monitoring processes directly on the end-hosts (Fig. 7.1b). By monitoring the per-connection states, Flowcorder can then record the performance of the connections, as experienced by the end-users, and then export those over IPFIX to complement existing measurement infrastructure. The rest of this section illustrates the different building blocks making up Flowcorder, visible on Fig. 7.2. More specifically, we consider a network administrator who wants to use Flowcorder to answer the following a high-level question: *"Which provider performs the best to synchronize files with a remote storage service accessed over TCP?"*

Computing performance profiles. The first step to answer this high-level question is to identify KPIs (§7.3.1) that enable to characterize the performance of the instrumented protocol. Such KPIs should contain general statistics about the connection, as well as metrics indicating possible performance issues, specific to the protocol.

For example, high-level KPIs to answer our illustrative question could be: (*i*) the number of bytes transferred and assumed to be lost; (*ii*) the amount of reordering [Jai+07; AFT11; BS02] that occurred in the network; and (*iii*) signs of bufferbloat, such as the number of bytes received multiple times, thus signaling a retransmission timeout on the sender, or times where the connection stalled and was blocked from sending pending data for several RTTs (TCP RTO).

Continuously streaming the collected KPIs is inefficient as, beside wasting resources, it might hide the key performance outliers in the noise generated by the huge number of smaller variations. Instead, Flowcorder exports the KPIs of a connection only at specific moments in the connection lifecycle (§7.3.3). In-between these exports, the KPIs are buffered in a lightweight aggregation daemon, local to the end-host. Once the decision to export the measurement is made, this aggregation daemon computes a *performance profile* of the connection: statistics computed over KPIs (*e.g.,* moving averages, counter increases) during well-defined moments of the connection life-cycle. The performance profile is then serialized as an IPFIX record and added in a pending IPFIX message buffer. As we want to minimize the processing load on the collector and take advantage of the features provided by IPFIX, the message is only exported once its size reaches the local MTU.

In our example, a connection towards the remote storage service that would experience one retransmission timeout in its entire life-cycle would generate four performance profiles: (*i*) one describing the connection establishment; (*ii*) one describing the performance of the data transfer (*e.g.,* aver-

age RTT, byte counters, number of RTO experienced) up to the RTO; (*iii*) one describing the performance while the connection is considered as lossy; and (*iv*) a final one describing the performance since the end of the lossy state and how the connection ended (*e.g.,* did it abruptly end with a TCP RST?).

Collecting KPIs. Under the hood, Flowcorder instruments existing transport protocol implementations on the end-hosts. Many methods exist to collect such statistics, such as extracting them from a general purpose loggers [EWT; ChroLog] or polling [RFC1157]. Instead, Flowcorder uses an event-based method. More specifically, Flowcorder inserts eBPF probes at specific code paths in the transport protocol implementations (§7.3.2). When the end-host stack reaches one of these probes, the probe handler is executed, computes KPIs of the connection, exports them in an asynchronous channel to the aggregation daemon, and then resumes the normal execution of the protocol implementation. Beside minimizing the instrumentation overhead (§7.6), this approach is also extremely flexible as it does not require any support from the implementation (*e.g.,* MIBs), and is thus not restricted to a predefined set of metrics, computed in an opaque manner.

In the example of Fig. 7.2, we see that one such probe has been setup to intercept the expiration of the TCP retransmission timer. If any connection experiences a RTO, this handler then increases the KPI counting RTO's and updates the connection's RTT estimated by TCP, then exports it for processing in user-space.

Analyzing performance profiles. Flowcorder produces measurements that can be collected, parsed and analyzed by any IPFIX collector supporting custom Information Elements [RFC5610]. Performance profiles are independent views of the performance of a connection during a given window of time, and one can be analyzed separately from the others belonging to the same connection. These performance profiles thus enable the network operator to build several views of the network according to key metrics using simple database queries, and to analyze them (§7.7).

For example, to answer his question, our network administrator could compute generic statistics such as mean, variance and median of all performance profiles contained in a given time window, aggregated by provider, and run hypothesis tests. These results could also be split based on the IP version, or compared against the general trend to access all other remote services. Finally, beside numerical tests, one can also generate time series and plot them in monitoring dashboards.

KPI	Description
$\sum$[*] `Sent`	Data[†] sent towards the remote host
$\sum$ `Received`	Data received and processed by the end host
$\sum$ `Lost`	Data assumed to be lost in the network
$\sum$ `Errors`	Data received corrupted
$\mathcal{A}$[‡] `RTT`	Mean Round-Trip-Time and variance (*i.e.,* jitter)
$\sum$ `Duplicates`	Received data already acknowledged
$\sum$ `OFO`	Data received out-of-order
$\mathcal{A}$ `OFO-dist`	Distance of out-of-order data from the expected one
$\sum$ `Stalls`	Count when the connection delays the sending of any pending data during several RTTs

Table 7.1: Key Performance Indicators can answer most questions about transport protocol performance

[*] $\sum$ denotes a counter over a time window
[†] Most KPIs can be duplicated to track byte-counts and packets ('data')
[‡] $\mathcal{A}$ denotes an average and a variance over a time window

7.3 Recording protocol performance

Flowcorder records *performance profiles* of connections directly on end-hosts, and exports them to a collector for further analysis. Achieving this requires addressing three issues: (*i*) What should a performance profile contain to describe a connection and indicate performance issues (§7.3.1)?; (*ii*) How can we collect these key metrics from the protocol implementations?; and (*iii*) When should these profiles be computed to maximize the accuracy of the measurements while minimizing the overhead of Flowcorder (§7.3.3)?

7.3.1 Characterizing protocol performance

Connection-oriented transport protocols such as TCP maintain state and usually expose some debugging information (e.g. `struct tcp_info` [ker] on Linux or macOS). However, recording the entire state for each established connection is impractical. Most of this information is very specific to the protocol implementation and does not always relate to connection performance. For example, one can find the distance (in terms of TCP segments) between the last out-of-order segment and the expected sequence number or the value of the slow-start threshold in the `struct tcp_info`, both of which give almost no insight to qualify the connection performance. Finally, while Flowcorder aims to collect fine-grained measurements about protocol performance as experi-

enced by the end-hosts, recording every single data point would be counter-productive, as the more critical observations will end up buried in a huge pile of data.

Instead, we characterize protocol performance by recording the evolution of Key Performance Indicators (KPIs) during a connection. Example KPIs are listed in Table 7.1. Recording `Sent` and `Received` bytes quantifies the volume transported on a connection, while tracking the number of segments quantifies the packet rate (*e.g.,* an interactive SSH session produces many small TCP segments). Recording `Lost` segments or segments with a checksum error (`Errors`), enables to qualify the path used by the connection. Tracking the evolution of the RTT (and thus implicitly its jitter) can be used to estimate whether congestion is building up in the network (and is the main source of information of some congestion control algorithms such as BBR [Car+16]). Similarly, recording the reception of segments containing already acknowledged data is an indication that the remote host mistakenly assumed their loss, which could be a sign of a possible bufferbloat. Measuring the amount of packet reordering is also useful, especially in the context of transport protocols, as its occurrence often limits the maximum achievable throughput. Finally, recording when a connection is prevented from making progress is a strong signal that something bad happened in the network (*e.g.,* triggering a TCP RTO).

From these KPIs, network administrators can then answer complex high-level questions characterising the performance of the network, such as: (*i*) what is the best response time that can be expected when connecting to a remote server?; (*ii*) Is the connection suitable for bulk transfers?; or (*iii*) Is the network congested?

7.3.2 Collecting KPIs from implementations

Recording the evolution of the KPIs of a connection on the end-hosts requires to extract them directly from the protocol implementation. Achieving this is usually possible using poll-based techniques. For example, SNMP can be used to query the TCP Management Information Base (MIB) [RFC4022]. Some OS'es also define APIs to retrieve information [ker; Ana], or log events to a centralized journal [EWT] which can then be monitored.

These techniques however come with two limitations. First, the information they give is limited to the explicitly defined metrics. For example, counting TCP out-of-order packets, as well as characterizing their out-of-order distance is impossible on Linux with the existing API. Counting received duplicates is not feasible either. Second, by requiring the monitoring tool to poll them, getting more accurate information about performance changes imposes a polling frequency and thus a high resource usage on the end-hosts. For example, characterizing the connection establishment times requires to pre-

cisely track the first few packets of a connection, which could be exchanged within a few milliseconds.

To address these issues, Flowcorder bypasses these traditional techniques, and directly instruments the protocol implementation at runtime.

Dynamic tracing using eBPF. Flowcorder leverages the existing dynamic tracing tools such as *kernel probes* [Gos05], or DTrace [CLS]. These enable to insert lightweight probes at runtime at arbitrary locations in either kernel (*e.g.,* to instrument the TCP kernel implementation (§7.4)) or user-space code (*e.g.,* to instrument DNS resolution routines, for which we present collected measurements in (§7.7)), typically around function calls. Conceptually similar to breakpoints and debugging watches, these probes automatically call user-defined handlers before and after executing the probed instruction. These handlers have complete access to the memory, as well as to the content of the CPU registers. More recently, the Linux kernel added code to define such handlers using extended Berkeley Packet Filters (eBPF) [IO].

eBPF code is pre-loaded in the kernel using the `bpf()` system call. This eBPF code is executed in an in-kernel virtual machine that mimics a RISC 64-bits CPU architecture, with 11 registers and a 512 bytes stack. This code can be interpreted, but many architectures include a JIT that compiles the eBPF bytecode to native machine code. Before accepting to load an eBPF code, a verifier ensures safety guarantees such as proof of termination (*e.g.,* by limiting the overall number of instructions and disallowing non-unrollable loops) and checks memory-access. eBPF code executed within the kernel can asynchronously communicate with user-space processes using perf events (FIFO queues). Additionally, eBPF programs can defines maps, which let them maintain state in-between executions. When an eBPF probe handler is executed, it receives an instance of the `struct pt_regs`, which describes the content of the CPU registers when the probe was hit, including the value of the stack pointer. This enables the eBPF handler to inspect the function arguments, or to explore the memory of the instrumented code. These capabilities make eBPF a target of choice to write probe handlers, as they guarantee that the handlers will not cause crashes nor hang the instrumented code, while also enabling it to compute complex statistics and easily report them to user-space.

This approach has at least five advantages. First, by leveraging state transitions that are internal to the implementation, it ensures an accurate translation to KPIs. For example, by recording retransmission timer expirations, it easily distinguishes between a connection that had no data to send for a while and a connection that was stalled and had to wait a complete RTO before sending anything else. Second, it seamlessly adapts to settings local to the host—for example, the TCP duplicate ACK threshold, or the support for SACK on a per connection basis—that alter the behavior of the transport protocol. As such,

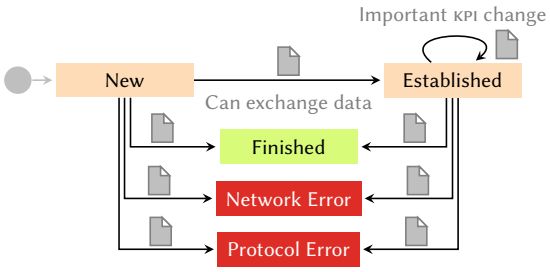


Figure 7.3: Transport protocol flows can be abstracted in a general FSM, where state-transitions act as signal to compute performance profiles.

it accurately captures the performance experienced by all instrumented end-hosts. Third, as it implements a push-based model where the transport stack itself calls Flowcorder, it minimizes the overhead on the end-hosts. Indeed, as the probe locations guarantee that all KPI changes will be detected, this avoids the need for constant, high-frequency, polling of the state-variables. Fourth, as it enables to both read per-connection states and to compute arbitrary statistics that can be stored in maps (thus defining custom ancillary state), this approach is highly flexible, as it does not rely on specific support from the protocol implementation. Finally, it could also be applied to encrypted transport protocols such as QUIC since it does not use the packet data but instead the state-variables of the protocol implementation.

7.3.3 Creating performance profiles

To use dynamic tracing and eBPF handlers to instrument a particular transport protocol, one needs to pick probe insertion locations to catch updates to the state of a connection. While a straw-man approach would pick the main functions involved in every send and receive operation, and continuously stream the connection KPIs after each sent and received packet, this would impose a high overhead without necessarily providing useful measurements. Indeed, once the probes are inserted, their handlers are executed for every connection hitting that code path. Instead, we aim at recording the evolution of KPIs between key events in the connection life-cycle. To this end, we place probes at locations that are seldom reached, yet catch all important events affecting the connection, and record statistics describing the evolution of the KPIs between two events. We call such set of statistics the *performance profile* of a connection.

A first set of events are defined by the protocol specifications. Such specification is usually composed of two different parts. The first is the syntax of the protocol messages, which can be expressed informally with packet de-

scriptions or more formally by using a grammar (*e.g.,* ABNF [RFC5234], ASN.1 [ASN.1]). The second part of the specification describes how and when these messages are sent and processed. Most Internet protocols specifications use Finite State Machines (FSM) to represent the interactions among the communicating hosts. Although implementations are usually not directly derived from their specification (e.g. for performance reasons or ease of maintenance), most implementations also include the key states and transitions of the protocol specifications. For example, most TCP implementations include the `SYN_RCVD`, `SYN_SENT` and `ESTABLISHED` state of the TCP specifications [RFC793]. While state transitions signal that a connection is making progress, not all of them provide similar information (*e.g.,* transitions into the TCP `TIMEWAIT` state give no information on the connection besides that "it is about to close"). Ultimately, these FSM describe the life-cycle of a connection. They can thus be abstracted by mapping their state and transitions to the three key phases in a connection life-cycle: (*i*) the connection establishment; (*ii*) the exchange of data; and (*iii*) the connection tear-down. These three stages enable us to define the abstract FSM visible on Fig. 7.3. When the state of a connection in this simplified FSM changes, it is a signal that Flowcorder needs to create a *performance profile* for the connection. Performance profiles should thus also contain the start and end states corresponding to their transition, enabling to compare the performance of connections for similar transitions (*e.g.,* characterize the connection establishment delay).

A second set of events that requires Flowcorder to generate a performance profile are the functions in the protocol implementation that indicate that an unexpected event occurred (*e.g.,* a retransmission timeout). We model this by a looping transition in the `ESTABLISHED` state in Fig. 7.3.

Finally, a third set of probe locations is defined by KPIs that are not computed by default by the protocol implementation. For example, metrics related to reordering for the TCP instrumentation. Tracking these KPIs then implies to create an ancillary state for the connection (*e.g.,* using an eBPF map), and updating it as the connection advances.

Once exported by the eBPF handlers, these performance profiles will eventually be received by an user-space aggregation daemon. This daemon then serializes these profiles to an IPFIX record, adding in the process information to identify both the connection (*e.g.,* the TCP 5-tuple) and the network path used (*e.g.,* the egress interface and source address). This record is then eventually exported to the collector.

7.4 Instrumenting the Linux TCP implementation

To demonstrate the applicability of our approach, we have applied it to the TCP implementation of the Linux kernel. This is a high-performance and widely used TCP implementation that has been tuned over more than a decade. We first introduce the KPIs building up the performance profiles of TCP connections (§7.4.1). Then, we describe the various eBPF handlers that are used, and illustrate their interactions (§7.4.2).

7.4.1 KPI selection

Instrumenting the Linux kernel TCP stack requires to map the chosen KPIs to TCP state variables. A TCP connection is represented in the kernel using the `struct tcp_sock`. As-is, this structure already contains most of the KPIs presented in Table 7.1. For example, `bytes_received` tracks the received bytes; `srtt_us` is a moving average of the estimated TCP RTT. Computing the statistics to create a performance profile from these state variables thus requires the eBPF handler to: (*i*) retrieve the address of the connection state from the parameters of the instrumented functions; (*ii*) copy the relevant state variables from the kernel memory to the eBPF stack; and (*iii*) compute the statistics on the evolution of the KPIs that these variables represent.

Unfortunately, not all KPIs from Table 7.1 are directly available in the TCP implementation. More specifically, four KPIs are missing. First, the number of duplicate incast bytes (`Duplicates`) is never recorded. If a connection receives a segment already (partially) acknowledged, the implementation ignores its payload. Second, the number of retransmission timeouts (`Stalls`) is not recorded. Similarly, the number of bytes and packets that arrived out of order (`OFO`) is not tracked. Finally, the existing `reordering` connection state variable is not sufficient to represent the distance between out-of-order packets (`OFO-dist`). Indeed, while it does express an out-of-order distance, it does so in terms of number of MSS-sized segments, and represents only the value computed for the last packet. Furthermore, it is clamped by a `sysctl` value.

Recording such "custom" KPIs thus requires to create an eBPF map alongside the probe handlers. This map can then be used to contain the ancillary state for each monitored connection (i.e., map a connection state to a data structure containing the value of the KPIs not provided by the protocol implementation). Managing this map has two implications. First, new entries must be added for any connection that will be monitored. This is especially important for connections initiated by the end-host itself. Indeed, if the TCP SYN they send is lost, the retransmission timer will expire, and the count of connection stalls will need to be increased. This does not apply for inbound connection requests, as creating state before their acceptance by user-space

Probe location	Pre	Post	Handler description
`tcp_v[46]_connect`	☑	☑	Register a new connection attempt and initialize its ancillary state; export KPIs to an error state if the function returns an error which indicates a cancellation of the connection.
`tcp_finish_connect`	☑	☐	Exports KPIs indicating the establishment of a new outbound connection.
`inet_csk_accept`	☐	☑	Exports KPIs for a new inbound connection accepted by user-space.
`tcp_set_state`	☑	☐	If a connection moves to `TCP_CLOSE`, compute its final state and exports its KPIs.
`tcp_retransmit_timer`	☑	☐	Export KPIs if the connection has stalled and enters a lossy state once established.
`tcp_fastretrans_alert`	☐	☑	If the connection congestion control state moves back to `TCP_CA_OPEN` (e.g., has recovered from an RTO), exports KPIs to mark the end of the lossy state.
`tcp_validate_incoming`	☑	☑	Detect incast duplicates; update the reordering KPIs if the packet enters the `ofo_queue`.

Table 7.2: A few probes in the Linux TCP implementation act as events to detect many performance changes.

application would provide a Denial-of-Service attack vector. Similarly, this ancillary state must be purged when the connection is over. The second implication of managing such ancillary state is that it imposes to insert eBPF code at every location where one of its value needs to be updated. Fortunately, as the missing KPIs represent very specific behaviors, these only require to instrument two extra locations (see §7.4.2). Note that the overhead of storing such ancillary is extremely low (*i.e.,* less than 100 bytes per connections in the current implementation).

7.4.2 Defining probe locations

Table 7.2 lists the functions of the Linux kernel where we insert our probes as well as their handler(s). These functions were chosen to minimize the over-

head induced by the probes, i.e., they are never executed in the context of the TCP "fast-path" processing. They fall into two categories.

First, we instrument the functions that correspond to state changes in the TCP FSM (i.e., from `tcp_v6_connect` to `tcp_set_state`). These indicate changes in the connection life-cycle and thus mandate computing KPIs. Second, we instrument functions that denote events which require us to update our ancillary connection state. More specifically, `tcp_retransmit_timer` let us track expirations of the retransmission timer. If a connection experiences a RTO, and its write queue is not empty or the user-space is blocked on a syscall, then it means that the connection has stalled. `tcp_fast_retrans_alert` may signal that a connection has recovered from a RTO (i.e., that the network is stable again) and moved back in the established state.

`tcp_validate_incoming`'s instrumentation is split into two handlers. First, it detects whether an incoming segment has already (partially) been acknowledged. Such a segment is an explicit signal that the other host experienced a retransmission timeout. Second, if the function accepts the received segment, this means that it is an out-of-order segment, and the handler updates the statistics tracking the reordering. Furthermore, as both `tcp_retransmit_timer` and `tcp_fast_retrans_alert` indicate that a significant performance event has occurred (a succession of losses in the network, and then a recovery), their handler also export KPIs. This eventually creates performance profiles looping on the `ESTABLISHED` state, enabling to describe the performance of the connection before, during, and after such transient events (e.g., a flash crowd causing congestion).

Collecting KPIs for a new outbound connection. We now illustrate how Flowcorder exports KPIs describing the establishment of a new outbound connection. In the example shown in Fig. 7.4, an application creates a regular TCP socket. Then, it tries to establish a TCP connection with the `connect()` system call. This system call is processed by the kernel, and eventually reaches the `tcp_v4_connect()` function, for which Flowcorder had registered a probe. This probe is executed before the instrumented function. It registers basic information about this connection establishment, such as its destination address and the time at which it started. Then, the kernel executes the `tcp_connect()` function, eventually sending a TCP SYN segment. When the function exits, the post handler is executed and immediately returns as the connection was successfully initiated and the kernel switches to other tasks. Unfortunately, this initial SYN does not reach the destination. After some time, the retransmission timer expires. This causes the kernel to execute the `tcp_retransmit_timer()` function. Again, Flowcorder intercepts that call using a probe, which increments the number of stalls. The kernel then sends a second TCP SYN.

When receiving the corresponding SYN+ACK, the kernel identifies the TCP

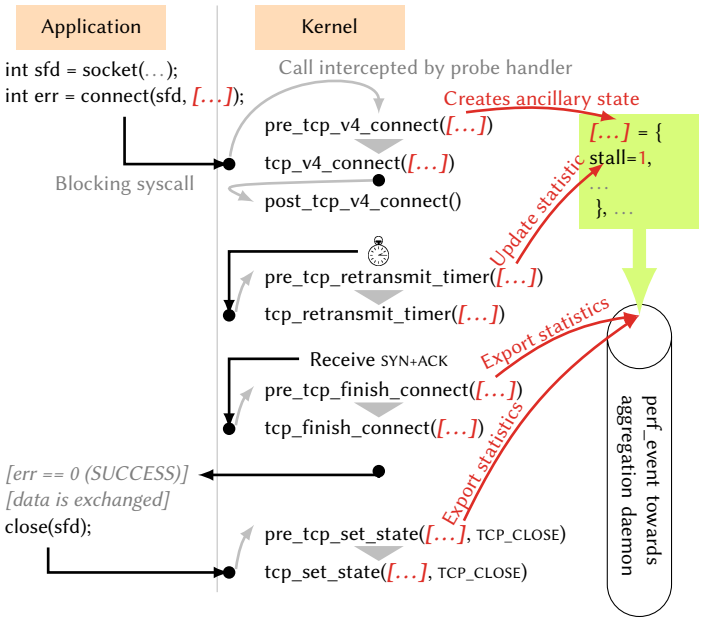


Figure 7.4: Abstract time-sequence diagram of the generated performance profiles of a TCP connection which loses its initial SYN, exchanges data, then closes. With a few kernel probes, our eBPF handlers trace the entire connection life-cycle and report it to an user-space daemon.

connection it corresponds to and reaches `tcp_finish_connect()`. As its corresponding eBPF handler is awoken, Flowcorder marks the connection as established, computes its KPIs and sends them to the user-space aggregation daemon using a `perf_event`. This daemon asynchronously fetches and analyzes the KPIs, builds the performance profile of this new connection and adds it in its IPFIX pending message buffer to send it later to the collector. In parallel, the `tcp_finish_connect()` kernel function completes and wakes up the application which can use the connection.

If the network then behaves perfectly (e.g., no reordering, and no losses), the probes placed in the kernel are never reached thus never executed for that connection. Finally, when the application closes its socket, the kernel eventually calls `tcp_set_state` to move the underlying connection to the `TCP_CLOSE` state. Flowcorder intercepts this call, computes the final set of KPIs for this connection, and exports a performance profile covering the entire connection and reaching a final state describing how the connection ended (e.g., FINISHED if both TCP FIN's were received and acknowledged).

7.5 Extending Flowcorder to support MPTCP

MPTCP is a new TCP extension which enables to operate a single TCP connection over multiple paths [RFC6824]. Two main implementations of this protocol exists: the reference one in the Linux kernel [PBa] and one deployed by Apple on iOS [App17]. We now demonstrate the genericity of Flowcorder, by enabling it to record performance profiles of MPTCP connections.

To instrument MPTCP, a few architectural details have to be taken into account. Despite being a relatively complex implementation (~18kLOC), it is heavily tied to the existing TCP implementation. At its heart, a MPTCP connection operating over two paths is composed in the kernel of two TCP connections, and of one meta-socket. This meta socket is the one exposed to user-space. It hijacks the socket API used by TCP (*i.e.,* user-space programs use MPTCP by default). Sending data using MPTCP requires to break the bytestream received on the meta-socket into chunks with a MPTCP sequence number (DSS), and then to send those over one of the subflows. The receiver's meta socket then reads the receive queues of its TCP subflows, and reassembles the original bytestream thanks to the DSS.

Instrumenting this implementation poses three challenges: (*i*) differentiating between a new MPTCP connection and regular TCP one can only be done once the SYN+ACK has been received, since MPTCP connection will contain a dedicated option (MP_CAPABLE); (*ii*) MPTCP subflows will trigger the same eBPF probes as regular TCP connections as they are implemented in the kernel using TCP connections; (*iii*) new subflows can be created directly by the

meta-socket, without any action from the user.

KPIs specific to MPTCP. As MPTCP subflows operate as regular TCP connections, we use the same set of KPIs as in §7.4.1 with one addition. When a retransmission timeout occurs on a subflow, its unacknowledged segments are retransmitted both on the subflow itself, as well as on another one (they are *reinjected* on another subflow). We record the number of reinjections done by a subflow in a new KPI present in the ancillary state of the MPTCP subflows.

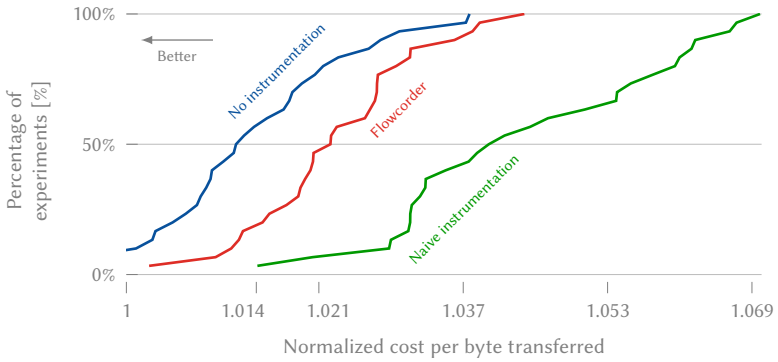
Recall that the meta-socket provides a bytestream service pretending to be TCP. As such, it supports most of the KPIs supported by TCP, with four tweaks. First, as it gets its segments from underlying TCP connections, it cannot receive corrupted segments and has no concept of latency, removing those KPIs. Second, segments arriving out-of-order on the meta-socket no longer indicate reordering happening in the network. Indeed, such reordering is handled by the subflows themselves, as part of their vanilla TCP implementation. Instead, reordering on the meta-socket is tied to the relative performance difference between the subflows[1]. Third, duplicate incast segments on the meta-socket indicate that the sender reinjected segments on another subflow as the duplicated ones timed out. Finally, retransmission timeouts at the meta-socket level indicate that the connection is suffering from head-of-line blocking (*e.g.,* a lossy subflow prevents all others from making progress). As one of the more common causes of such a behaviour are too small receive buffers, this defines a new KPI specific to the meta-socket.

eBPF probes handlers. All probes defined in §7.4.2 also record the performance of MPTCP subflows as-is. In addition to them, we update the ancillary state to track reinjection across subflows by instrumenting `__mptcp_reinject_data`. Recording the performance of the meta-socket also requires the addition of probes to record the expiration of its retransmission timer (`mptcp_meta_retransmit_timer`). New subflows initiated by the instrumented host are automatically detected by the probes handling the creation of TCP connections. Detecting the creation of new subflows initiated by the remote host (*e.g.,* when operating as server) requires instrumenting `mptcp_check_req_child`.

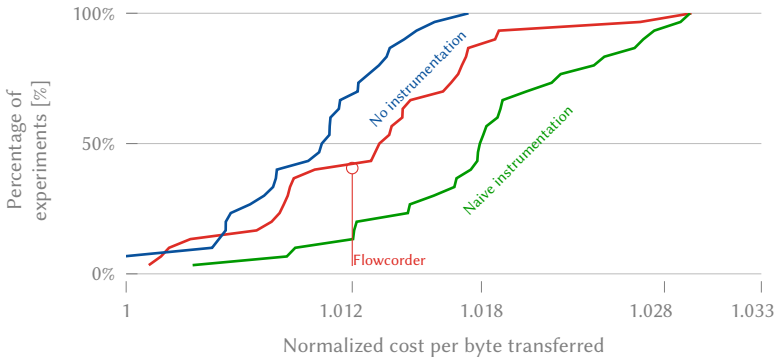
7.6 Flowcorder operates with little overhead

In this section, we begin by evaluating the overhead of Flowcorder when instrumenting the Linux TCP stack. We first run micro-benchmarks to estimate

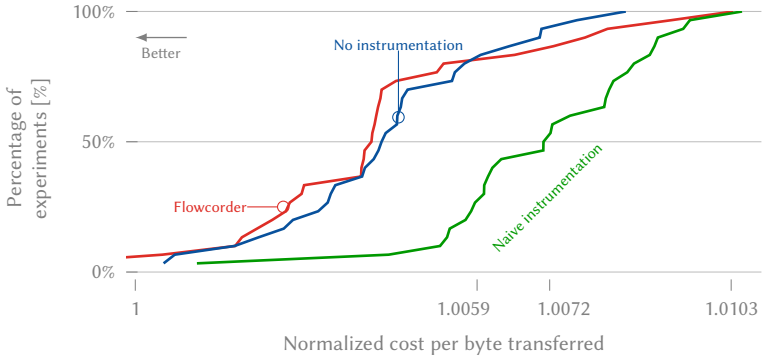
[1]Consider two successive segments A and B, such that A comes first in the MPTCP bytestream. If B arrives before A on the receiver's meta-socket, it then follows that: (*i*) A and B were sent over different subflows, as subflows guarantee in-order delivery; and (*ii*) the subflow of B was "better", e.g., had a lower latency, and/or fewer losses.



(a) Flowcorder induces a small overhead when used over a link with no loss and sub-ms RTT.



(b) Introducing a 10ms RTT and 0.1% loss rate reduces the overhead by an order of magnitude.



(c) 30ms RTT and 1% losses cause overhead of Flowcorder to become negligible.

Figure 7.5: Analyzing the number of instruction executed to saturate a 10G link shows that the overhead induced by the kernel probes is negligible, especially when the link exhibits losses or reordering.

the overhead of Flowcorder in function of the characteristics of the underlying network (§7.6.1). Then, we evaluate the application-visible performance impact of instrumenting the TCP stack (§7.6.2). Finally, we conclude the section by presenting how to evaluate that the performance profiles produced by Flowcorder are accurate, especially after kernel upgrades thus potential changes in the instrumented protocol implementation (§7.6.3).

7.6.1 Instrumentation overhead

To estimate the overhead induced by the monitoring daemons as well as the kernel probes injected in the TCP stack by Flowcorder, we execute a benchmark between two servers (each with 8-cores CPUs at 2.5Ghz and 8G of RAM) and connected using 10G interfaces. Intuitively, the overhead of running Flowcorder amounts to estimating how much extra work the machine being instrumented did to perform a given task. We use NTTTCP [ntttcp] to initiate multiple TCP connections from one server to the other, effectively saturating the 10G link. For each experiment, we record how many bytes were successfully transferred, and use `perf` [perf] to record how many instructions were executed during the experiment, as reported by the hardware CPU counters. Each experiment ran for 60 seconds, in order to average out measurement errors. We also varied the RTT applied over the link (from a few hundred μs to 100ms), its jitter (10% of the RTT), as well as its loss rate (from 0 to 1% of random losses). We performed 200 experiments per combination of RTT and loss rate, and had Flowcorder enabled on the servers in half of those experiments. Then, we compared the resulting cost (expressed as the average number of instructions executed on the servers, divided by the number of bytes successfully transferred) of using Flowcorder. Using the number of executed CPU instructions as metric has at least four advantages: (*i*) it is independent of the precise duration of the experiment (i.e., coarse-grained timers have no incidence on the results); (*ii*) it isolates the results from the transient states of TCP congestion control; (*iii*) it is independent of the CPU frequency, which is impossible to precisely control when running benchmarks; and (*iv*) it captures both the load induced by the kernel probes, and the load induced by the user-space daemons aggregating KPIs and exporting IPFIX records. We show a summary of the results in Fig. 7.5, which plots the cumulative distribution of the fraction of experiments according to their normalized cost (i.e., we normalize all costs by the lowest one).

When operating over a perfect link (Fig. 7.5a), we see that Flowcorder increases by about 1.1% the number of instructions executed during a test. As the experiments had almost no delay and no losses, this gives a baseline as to how expensive it is to run Flowcorder, when all connections are processed in the kernel fast path (i.e., the path levering as many optimizations as pos-

sible, such as hardware offload or skb coalescing, which decreases the overall CPU cost of the connection) thus trigger as few events as possible. When adding some delay (10ms of RTT, and 1ms of jitter), and a small random loss probability of 0.1%, we see in Fig. 7.5b that this per-byte instruction overhead decreases quite substantially, and is approximately 0.3%. Indeed, as segments starts to arrive out-of-order, or are lost, the TCP stack begins to process these segments in the slow path, which is much more expensive CPU-wise than the load induced by Flowcorder. This impact is even more visible as we reach a RTT of 30ms±3ms, with a loss rate of 0.5% (Fig. 7.5c) which puts the executed instructions per byte successfully transferred overhead at only 0.06%.

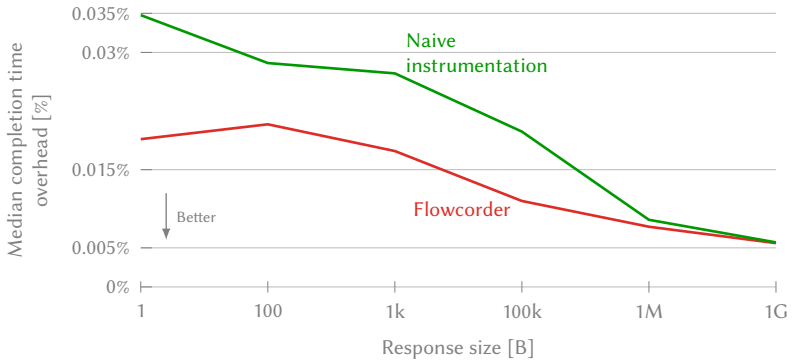
This indicates that the relative cost of using Flowcorder decreases when the network quality worsens, thus when Flowcorder starts to actually produce performance profiles. The handling of lost or out-of-order segments has a much larger impact on the performance than the kernel probes inserted by Flowcorder and associated monitoring daemons. The decrease in the number of instructions per byte transferred between Fig. 7.5a and Fig. 7.5b is expected, as increasing the RTT by several orders of magnitude increases the idle periods of connections as they wait for ACKs.

We performed the same experiments when instrumenting the MPTCP implementation (§7.5), and observed similar results (albeit with a smaller overhead as MPTCP disables the kernel TCP fast path processing).

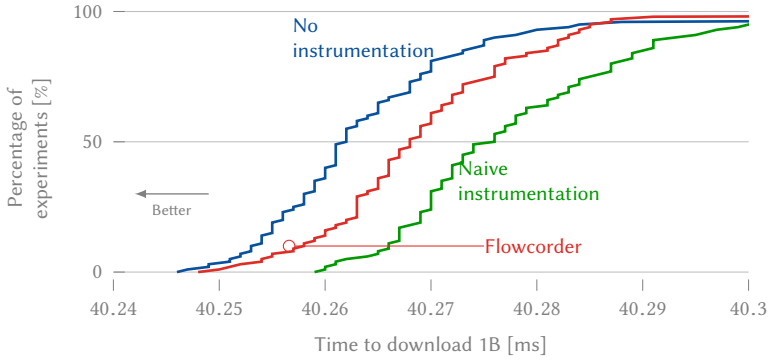
7.6.2 Impact on application performance

The previous section showed that Flowcorder was inducing some overhead on the instrumented end hosts. In this section, we evaluate whether this overhead can cause application-visible performance degradations. To this end, we configure one host to run a HTTP server. We then record the time to perform an HTTP GET to download a file of a given size from the server. As we saw in §7.6.1, the overhead of Flowcorder is maximum in a perfect network. Both client and server are directly connected, and Ethernet flow-control is enabled to ensure the absence of losses. We simulate the client requests using ApacheBench [apsf], with a variable number of parallel connections (up to 100). We repeated each experiment 2000 times (i.e., opened 2000 connections for each response size), and half of those had Flowcorder enabled on the client. We recorded for each experiment how quickly the connection completed (i.e., how long did it take to perform the TCP three-way handshake, the HTTP GET, then download the response and close the connection), and the results are visible in Fig. 7.6.

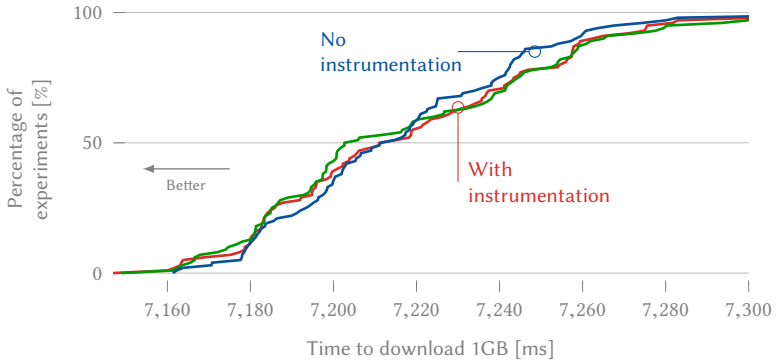
Fig. 7.6a shows the median overhead per response size, which is the observed increase in completion time when the end hosts was being instrumented by Flowcorder. We see that as the size of the HTTP response increases,



(a) Flowcorder has a negligible overhead on the overall completion time when downloading a file.



(b) The instrumentation overhead is comparatively higher when downloading one-byte files.



(c) Large transfers amortize the overhead as few performance profiles are generated per connection.

Figure 7.6: Using Flowcorder has almost no application-visible impact on the performance of the Linux TCP stack.

the overhead decreases. This result is expected. Indeed, recall that Flowcorder generates at least two performance profiles for each connection, and none in the established state if there are no performance degradations. If the response exceeds a few TCP segments, its completion time is thus dominated by the TCP data transfer, and not by the execution of kernel probes. Fig. 7.6b thus shows the absolute worst case for these experiments, as the response consists in a single segment. We see that the median increase in the response time in that case is about 0.15%. Fig. 7.6c shows the overhead with a 1GB response. We note that the overhead there was about 0.03%. We also performed experiments over a link with some delay and/or losses, and observed that the overhead in those case was even lower as the response time was completely dominated by the network characteristics.

These benchmarks, show that despite inducing some overhead, Flowcorder has a very low (if not negligible) impact on the performance of connections initiated by applications. This result also holds when instrumenting MPTCP.

7.6.3 Ensuring accurate measurements

The content of the performance profiles generated by Flowcorder, and thus the accuracy of the measurements, clearly depend on the correctness of the instrumentation of the protocol implementation.

Sources of measurement errors. As Flowcorder extracts most of its KPIs by performing raw memory reads directly in the per-connection states, this extraction process it thus a first possible source of errors. Values could be read at incorrect offsets, or be decoded incorrectly (e.g., reading only the first 32b of a 64b counter). The second source of possible errors are the assumptions the probes make on the status of the connection. For example, the TCP instrumentation assumes that a connection can be identified by the address at which its state resides, which is conveniently passed around as `struct sock *sk` in most functions. If such assumption is wrong (or no longer holds due to an update), then Flowcorder will produce incorrect measurements as it might mix up connections, wrongly assume that a connection received an out-of-order segment, ... The third source of errors is the set of probes and their locations. Indeed, as the implementation of the protocol improves over time, the set of functions called for each event (e.g., received segments, timer expiration) and their relative order might change. The most obvious effect of this on Flowcorder would be inconsistent performance profiles (e.g., increasing the number of bytes transferred of a closed connection), or missed events (e.g., connections never appearing in logs, missed RTOs).

Preventing measurement errors. To prevent the first source of errors, Flowcorder re-compiles its eBPF code every time its probes are inserted. As

this compilation process directly happens on the instrumented machine, it can use information local to the machine (e.g., headers matching the running kernel, values in procfs to enable or disable the MPTCP instrumentation). This source of measurement errors is thus prevented by design. Incidentally, this re-compilation process also ensures that probes are always inserted at their proper locations, as their offset are also dynamically computed during the eBPF compilation.

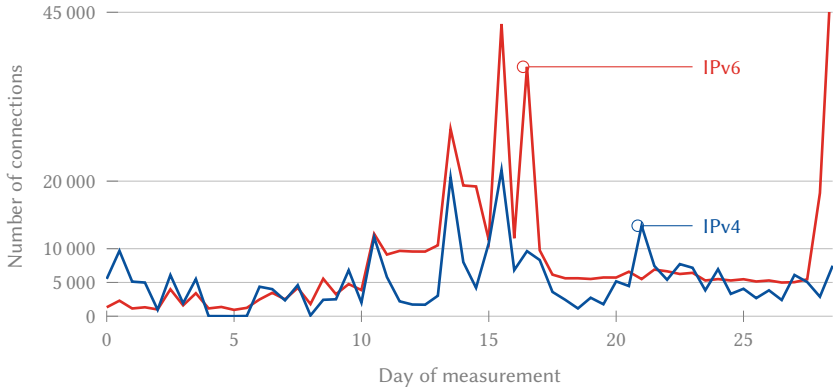
To prevent the seconds and third type of errors, we built a test suite using Packetdrill [Car+13]. Packetdrill enables us to test protocol implementation using scripts which describe connections. More specifically, those scripts inject crafted packets in a local interface at specific points in time, as well as specify the content of packet(s) that should be sent by an implementation in response to incoming packets or API calls. Packetdrill contains a set of edge test cases for the Linux TCP implementation, and similar test cases for MPTCP are available [Sch]. As each test case depicts a well-defined connection, we can statically predict the performance profiles that should be produced by Flowcorder when instrumenting that connection. This lets us build integration tests to validate that Flowcorder accurately instruments protocol implementations.

Using this test suite, we were able to ensure that Flowcorder accurately instruments the TCP stack of the Linux kernel from v4.5 to v4.17, and MPTCP v0.93.

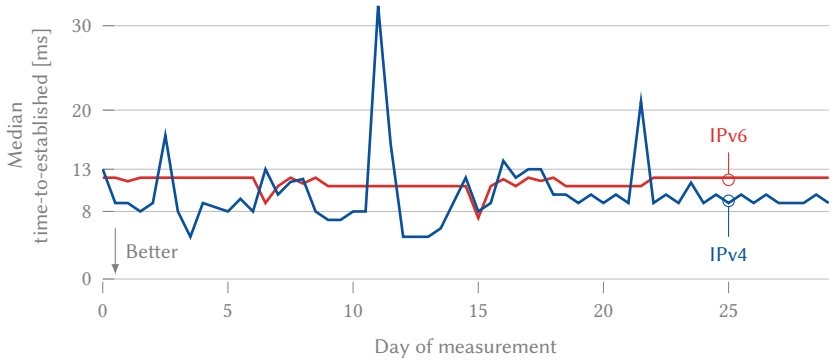
7.7 Flowcorder in a campus network

We now present measurements collected over one month with Flowcorder in a campus network. We deployed Flowcorder in student computer labs, where we run on every host monitoring daemons that instrument the Linux kernel TCP stack, presented in (§7.4), as well as DNS resolutions libraries. Each end-host is dual-stacked and has public addresses.

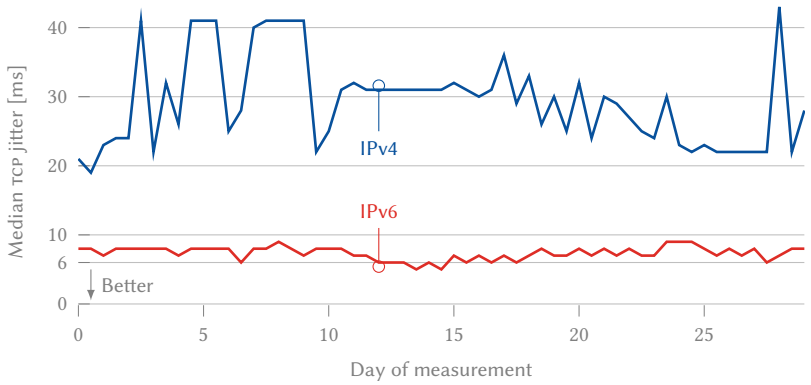
Viewing the effects of Happy Eyeballs. Fig. 7.7a shows the repartition of the TCP connections in function of the IP version used. We see that most of the connections are established using IPv6. As major cloud services are very popular amongst students and they all support IPv6, this could be due to Happy Eyeballs [RFC6555]. We can confirm that Happy Eyeballs indeed favors connections over IPv6 by looking at Fig. 7.7b. It compares the median time required to establish new TCP connections depending on the used address family. More specifically, it only contains connections established towards dual-stacked ASes. We see that the time to open a new connection is similar for both address families, despite IPv4 exhibiting many outliers. As Happy Eyeballs gives IPv6 connections a head start of usually 300ms (although some



(a) Number of TCP connections in the data set.

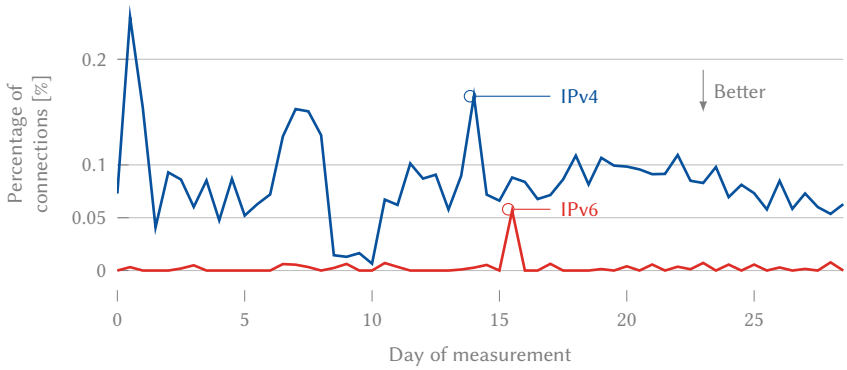


(b) IPv4 and IPv6 have similar connection establishment delays, causing Happy-Eyeballs to favor IPv6.

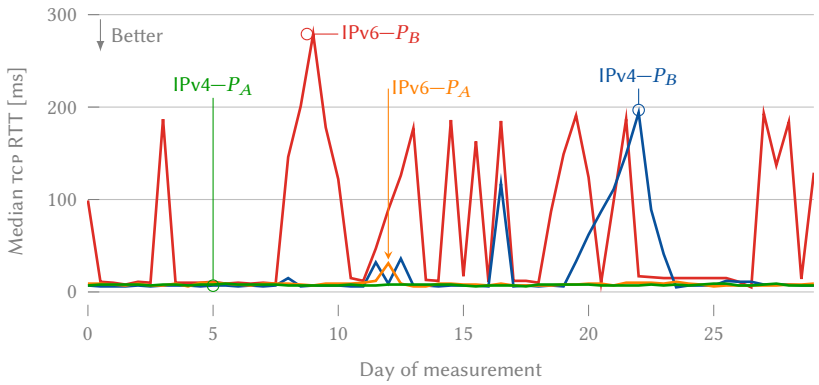


(c) IPv4 connection suffer from a larger jitter.

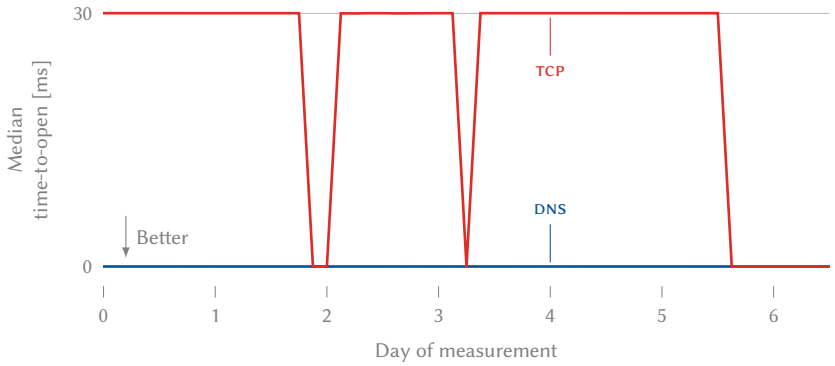
Figure 7.7: Network performance insights provided by Flowcorder in a dual-stacked, multi-homed, campus network.



(d) A disproportionate amount of IPv4 connections experiences at least one initial SYN retransmission.



(e) One large cloud service provider often routes requests towards datacenters inducing large RTTs.



(f) An issue in one datacenter caused colocated service to present vastly different response time.

Figure 7.7: Network performance insights provided by Flowcorder in a dual-stacked, multi-homed, campus network.

have called to reduce it [BS16a]), this explains why IPv6 is almost always used to reach popular services.

Comparing the performance of different uplinks. Our network is dual-homed. It uses different uplinks for IPv4 and IPv6. We leverage Flowcorder to analyze the difference between the two address families. Fig. 7.7c shows the median jitter observed for TCP connections. We observe that the jitter experienced by IPv4 connections is higher than for IPv6. This correlates with the trend from Fig. 7.7b, where IPv4 showed more variations. Finally, to better understand why the IPv4 connection establishment delay had a higher variance, Fig. 7.7d shows the ratio of connections that were successfully established after losing their initial TCP SYN. We see that this mainly occurs only for IPv4, which might point to an on-site issue with a firewall or congestion of the IPv4 uplink. Overall, these results show that IPv6 connections seem to perform better than IPv4 connections in our campus. This is expected, as only the IPv4 traffic is shaped by our provider.

Comparing the performance of remote cloud services. Another usage for the measurements collected by Flowcorder is to compare the performance when accessing different cloud services. Indeed, as an ISP might have different peering agreements with them, measuring the quality of the connections towards those service can be a factor to decide whether to subscribe to one service or another (or to select a different ISP). For example, Fig. 7.7e compares the median TCP RTT when accessing two popular cloud services. For these services, a low RTT is key to ensure a proper level of interactivity. We see that while both services tend to show similar RTT's over IPv4, one of them (P_B) performs much worse when accessed over IPv6[2]. Keep in mind that while Flowcorder uses TCP's estimates to report RTT and jitter, this might not completely reflect the true values to reach the actual server, as there could be middleboxes or TCP proxies present on the path, fiddling with segments.

Detecting a local operational issue. Beside providing external connectivity, our campus network also hosts services such as a DNS resolver or institutional web servers. During our measurement campaign, students were complaining that accessing those web servers was abnormally slow. As these web servers are collocated with the DNS servers, we can thus directly use Flowcorder to compare their performance. Fig. 7.7f shows the median time to establish a connection to any of these servers. Given that the servers are located a few hundreds of meters away, 30ms to receive a SYN+ACK is a clear performance anomaly, especially compared to the time required to receive a DNS reply. After talking with the network operators, we learned that this

[2]Further analyzes revealed that the provider's DNS was causing students' requests to use datacenters located on another continent.

problem was due to a faulty load-balancer that was fixed near the end of the observation period.

7.8 Related work

Monitoring network performance is an age-old topic. Flowcorder draws from three main threads of work.

Collecting transport performance metrics. Passive inference of transport protocol characteristics has been a primary source of measurements for a long time, e.g., inferring per-flow TCP states by analyzing packet headers provided by a network tap (tstat [Mel02]), or correlating packet traces collected on the end hosts (Deja-vu [Agg+11]). More recent approaches tailored to data-centers (e.g., Trumpet [Mos+16], Dapper [GBR17]) perform such analyzes in real-time, at the edges of the network (i.e., access switches or virtual machine hypervisors). While these techniques provide fine-grained measurements for TCP they will not be applicable to emerging encrypted protocols such as QUIC.

Instrumenting the end-hosts. SNAP [Yu+11] or NetPoirot [Arz+16] collect an enormous amount of statistics about TCP connections directly from datacenter hosts. By collecting those on a central management system, they can then correlate observations in order to identify the root causes of performance issues (e.g., bottleneck switch or link, or misconfigured of TCP delayed ACK's). Both tools poll event loggers (e.g., Windows EWT, or Linux syslog) every few milliseconds. As such, they are restricted to the measurements provided by those loggers (typically the TCP MIB [RFC4022]), with a higher CPU overhead than Flowcorder. Odin [Cal+18] is a framework injecting javascript when serving client requests from CDN to perform active measurements. While this approach collects performance metrics as experienced by end-hosts, the measurements that it can records are, by design, much more limited.

Instrumenting protocol implementations. Several tools provide some visibility over the internals of the Linux TCP stack. tcpprobe [Theb] is a kernel module which logs the evolution of the congestion control variable in response to incoming TCP segments. tcp-tracer [Wea] reports the TCP state changes (e.g., NEW→ESTABLISHED) for all connections. BCC [Thea] provides several small tools, enabling to log some aspects of TCP connections. All of these tools use the same primitives to instrument the TCP stack (*i.e.*,, kprobes, often combined with eBPF handlers), but they are not coupled with enterprise management systems and only monitor very specific aspects of the protocol implementation.

8 Summary

This part introduced two network monitoring frameworks, each tailored to answer the specific needs of different types of networks.

First, Stroboscope (§6) is an in-network solution to collect precise metrics about traffic flows, with a controlled monitoring overhead. As such, Stroboscope is particularly well-suited to provide visibility over the traffic of ISP networks. Stroboscope combines the visibility benefits of traffic mirroring with the scalability of traffic sampling. As such, Stroboscope can monitor traffic flows with very large traffic volumes, while also enabling to reason about network-wide behaviors. We implemented Stroboscope[1], and show that it scales well: it computes measurement campaigns for large networks and query sizes in few seconds, and produces a number of mirroring rules well within the limits of current routers. Stroboscope works on existing routers and is therefore immediately deployable.

Second, we presented Flowcorder (§7), a monitoring framework which directly extracts Key Performance Indicators from the end-hosts. By directly instrumenting protocol implementations, Flowcorder is able to collect fine-grained measurements transparently supporting fully-encrypted and multi-path protocols. We presented an implementation of Flowcorder[2] for the Linux kernel TCP and MPTCP stacks, and are now extending our approach to instrument QUIC libraries, as their implementations stabilize. We presented performance insights provided by Flowcorder when deployed in a campus network. Flowcorder operates with a negligible overhead, and integrates with existing measurement collection infrastructure.

Both of these monitoring frameworks enable to collect fine-grained measurements about different aspects of the networks. As such, one future research direction would be to use their measurements to drive tight control-loops. For example, Flowcorder's performance profiles could be used by an enterprise network controller to optimize the content of DNS replies. Such integration would enable to dynamically select the best performing provider or

[1] Available at `https://github.com/net-stroboscope`.
[2] Available at `https://github.com/flowcorder`.

address family for a given service. Similarly, as Stroboscope enables to collect end-to-end measurements on any path (including forwarding paths), it provides the necessary data to dynamically adjust the behavior of the network to reach target Service-Level Agreement.

Finally, the measurements collected by both monitoring frameworks could be processed by machine-learning algorithms. For example, Stroboscope 's monitoring queries could be automatically generated by such a system to confirm the existence of a predicted network outage. Flowcorder's performance profiles could be filtered, in order to automatically highlight measurements showing likely performance issues.

9 Conclusion

Today's networks are mostly static, as adapting them to new requirements is complex and error-prone. Furthermore, they have little visibility over their traffic as they rely on random packet sampling. More specifically, ISP networks are currently unable to measure network-wide behaviors, such as forwarding paths, or one-way delays. In parallel, enterprise networks are losing their ability to monitor the performance of their connections as newer, encrypted, and multipath protocols are deployed. As a result, today's networks are inflexible.

In this thesis, we explored how to improve their flexibility in two ways. First, we developed a new control primitive, Fibbing, which enables to centrally control the forwarding paths computed by distributed routing protocols (§3). More precisely, we formulated provably correct algorithms that augment link-state IGP topologies with fake nodes and links to implement forwarding requirements. These algorithms scale to large topologies, and complete in a time frame suitable for online use. Additionally, we discussed how Fibbing interacted with overlay protocols (*e.g.,* BGP, or MPLS), as well as how it could be used to enable incremental SDN deployment. Then, we presented a complete implementation of a Fibbing controller which is able to program paths in an OSPF network made of commercial, unmodified routers (§4). In particular, we showed how we could map Fibbing primitives to parts of the OSPF specifications, and identified the key building blocks to be able to implement a controller. We demonstrated that Fibbing induced almost no overhead on real routers, and that our implementation supported advanced failure semantics. Finally, we demonstrated the practicality of our implementation through a case study where a Fibbing controller is used to implement real-time traffic engineering policies.

In a second time, we explore how to improve the flexibility of network monitoring systems. First, we presented Stroboscope, a network monitoring framework enabling ISP operators to collect fined-grained measurements (§6). Stroboscope computes high-level queries into a schedule of low-level mirroring (de-)activations, in order to collect small traffic samples. To that end, it

features a compilation pipeline with provably correct algorithms which optimizes where and when to activate mirroring rules, in order to provide both strong accuracy guarantees, and to adhere to a monitoring budget. We implemented a complete Stroboscope prototype, and demonstrated that it could work with today's routers. Finally, we introduced Flowcorder, a monitoring framework aiming at capturing the performance of protocols in enterprise networks (§7). Flowcorder transparently supports encrypted and multipath protocols, unlike traditional passive monitoring techniques. Flowcorder instruments protocol implementations on the end-hosts in a generic way in order to collect Key Performance Indicators on a per-connection basis. These KPIs are aggregated into connection performance profiles, and eventually sent to an IPFIX collector for analyzes. We presented a complete implementation of Flowcorder which instruments the Linux TCP stack, and demonstrated its extensibility by adding the support for MPTCP. Flowcorder operates with almost no overhead on the end-hosts. We then presented hinsights provided by Flowcorder when deployed in a campus network. We believe that both Stroboscope's and Flowcorder's measurements can be used to drive tight control-loops for their respective kind of networks, as well as enhance their troubleshooting capabilities.

Open problems

Along the way, several future research directions have been opened in this thesis. First, Fibbing shows that some level of programmability can achieved using distributed protocols. As such, future work could explore further the primitives that can distributed in the IGP (*e.g.,* alternate path computation) and those that should be kept centralized. Furthermore, as the industry as been heavily deploying Segment-Routing, we believe that its interactions with Fibbing should be investigated further as both approaches can benefit from one another—*e.g.,* SR could use Fibbing in lieu of binding segments to reduce the label stack and ensure failure resiliency, while Fibbing could leverage SR to express function chains.

Second, Stroboscope shows that by controlling its overhead, packet mirroring is a useful monitoring primitives. Further research could explore how to enhance the current runtime aspects of Stroboscope, *e.g.,* to implement them on the router themselves. Additionally, integrating Stroboscope with machine-learning systems could automatically generate monitoring queries, enabling to monitor paths "about to have issues".

Finally, Flowcorder shows that recent advances in OS instrumentation can be used to collect fined-grained performance records of connections. Extending it to support a mature QUIC library implementation is one of the possi-

ble future research direction. Another one could be to extend it to be able to dynamically switch to more fined-grained analyzes as it detects a potential performance issue (*e.g.*, start to monitor all incoming TCP segments once one has been received out-of-order to detect whether the issue is sporadic or is constant).

References

Scientific papers

[AFT11] Brice Augustin, Timur Friedman, and Renata Teixeira. "Measuring multipath routing in the internet". In: *IEEE/ACM Transactions on Networking (TON)* 19.3 (2011), pp. 830–840.

[Agg+11] Bhavish Aggarwal et al. "Deja Vu: Fingerprinting Network Problems". In: *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*. CoNEXT '11. Tokyo, Japan, 2011, 28:1–28:12. DOI: 10.1145/2079296.2079324.

[Ari+03] I. Ari et al. "Managing flash crowds on the Internet". In: *MASCOTS, IEEE*. 2003.

[Arz+16] Behnaz Arzani et al. "Taking the Blame Game out of Data Centers Operations with NetPoirot". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil, 2016, pp. 440–453. DOI: 10.1145/2934872.2934884.

[Bec+16] Ryan Beckett et al. "Don'T Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil, 2016, pp. 328–341. DOI: 10.1145/2934872.2934909.

[Bec+17] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. "A General Approach to Network Configuration Verification". In: *SIGCOMM*. 2017, pp. 155–168.

[Ben+17] Ran Ben Basat et al. "Constant Time Updates in Hierarchical Heavy Hitters". In: *SIGCOMM*. 2017, pp. 127–140.

[Ber+14] Pankaj Berde et al. "ONOS: towards an open, distributed SDN OS". In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, pp. 1–6.

[BHH16] Anat Bremler-Barr, Yotam Harchol, and David Hay. "OpenBox: a software-defined framework for developing, deploying, and managing network functions". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM. 2016, pp. 511–524.

[Bos+14] Pat Bosshart et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[BS02] John Bellardo and Stefan Savage. "Measuring packet reordering". In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM. 2002, pp. 97–105.

[BS16a] Vaibhav Bajpai and Jürgen Schönwälder. "Measuring the effects of happy eyeballs". In: *Proceedings of the 2016 Applied Networking Research Workshop*. ACM. 2016, pp. 38–44.

[BS16b] O. Bonaventure and S. Seo. "Multipath TCP deployments". In: *IETF Journal* 12.2 (2016), pp. 24–27.

[Cae+05] M. Caesar et al. "Design and implementation of a routing control platform". In: *NSDI*. 2005.

[Cal+18] Matt Calder et al. "Odin: Microsoft's Scalable Fault-Tolerant CDN Measurement System". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association. 2018.

[Car+13] Neal Cardwell et al. "packetdrill: Scriptable Network Stack Testing, from Sockets to Packets." In: *USENIX Annual Technical Conference*. 2013, pp. 213–218.

[Car+15] Marcel Caria, Tamal Das, Admela Jukan, and Marco Hoffmann. "Divide and conquer: Partitioning OSPF networks with SDN". In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE. 2015, pp. 467–474.

[Car+16] Neal Cardwell et al. "BBR: Congestion-based congestion control". In: *Queue* 14.5 (2016), p. 50.

[CLL09] Jianer Chen, Yang Liu, and Songjian Lu. "An improved parameterized algorithm for the minimum node multiway cut problem". In: *Algorithmica* 55 (2009), pp. 1–13.

[Coo71] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM. 1971, pp. 151–158.

[Cra+03] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. "Gigascope: a stream database for network applications". In: *SIGMOD*. 2003, pp. 647–651.

[CRS16] Marco Chiesa, Gábor Rétvári, and Michael Schapira. "Lying your way to better traffic engineering". In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM. 2016, pp. 391–398.

[DB17] Quentin De Coninck and Olivier Bonaventure. "Multipath QUIC: Design and Evaluation". In: *Conext'17*. ACM. 2017, pp. 160–166.

[Det+13] Gregory Detal et al. "Revealing middlebox interference with tracebox". In: *Proceedings of the 2013 conference on Internet measurement conference*. ACM. 2013, pp. 1–8.

[Dix+13] Advait Dixit et al. "Towards an elastic distributed SDN controller". In: *ACM SIGCOMM computer communication review*. Vol. 43. 4. ACM. 2013, pp. 7–12.

[Fay+14] Seyed Kaveh Fayazbakhsh et al. "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags". In: *Proc. NSDI*. 2014.

[Fin+11] Alessandro Finamore et al. "Experiences of internet traffic monitoring with tstat". In: *IEEE Network* 25.3 (2011), pp. 8–14.

[Fog+15] Ari Fogel et al. "A General Approach to Network Configuration Analysis". In: *NSDI*. 2015, pp. 469–483.

[Fos+13] N. Foster et al. "Languages for software-defined networks". In: *Communications Magazine, IEEE* 51.2 (Feb. 2013), pp. 128–134. ISSN: 0163-6804. DOI: 10.1109/MCOM.2013.6461197.

[Fra+05] Pierre Francois, Clarence Filsfils, John Evans, and Olivie r Bonaventure. "Achieving Sub-second IGP Convergence in Large IP Networks". In: *SIGCOMM Comput. Commun. Rev.* 35.3 (July 2005).

[FRT02] B. Fortz, J. Rexford, and M. Thorup. "Traffic engineering with traditional IP routing protocols". In: *Communications Magazine, IEEE* 40.10 (2002), pp. 118–124. ISSN: 0163-6804. DOI: 10.1109/MCOM.2002.1039866.

[FRZ13] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The road to SDN". In: *Queue* 11.12 (2013), p. 20.

[FT00] B. Fortz and M. Thorup. "Internet traffic engineering by optimizing OSPF weights". In: *Proc. INFOCOM*. 2000.

[FT84] M. L. Fredman and R. E. Tarjan. "Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms". In: *25th Annual Symposium onFoundations of Computer Science, 1984.* Oct. 1984, pp. 338–346. DOI: 10.1109/SFCS.1984.715934.

[GBR17] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. "Dapper: Data plane performance diagnosis of tcp". In: *Proceedings of the Symposium on SDN Research*. ACM. 2017, pp. 61–74.

[Gem+16] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. "Fast Control Plane Analysis Using an Abstract Representation". In: *SIGCOMM*. 2016, pp. 300–313.

[GSF13] Sriharsha Gangam, Puneet Sharma, and Sonia Fahmy. "Pegasus: Precision hunting for icebergs and anomalies in network flows". In: *INFOCOM, 2013 Proceedings IEEE*. IEEE. 2013, pp. 1420–1428.

[Guo+15] Chuanxiong Guo et al. "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom, 2015, pp. 139–152. DOI: 10.1145/2785956.2787496.

[GW02a] Timothy G. Griffin and Gordon Wilfong. "On the Correctness of IBGP Configuration". In: *SIGCOMM Comput. Commun. Rev.* 32.4 (Aug. 2002), pp. 17–29. ISSN: 0146-4833. DOI: 10.1145/964725.633028.

[GW02b] Timothy G Griffin and Gordon Wilfong. "Analysis of the MED Oscillation Problem in BGP". In: *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on*. IEEE. 2002, pp. 90–99.

[GW99] Timothy G. Griffin and Gordon Wilfong. "An Analysis of BGP Convergence Properties". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '99. Cambridge, Massachusetts, USA, 1999, pp. 277–288. DOI: 10.1145/316188.316231.

[Han+14] Nikhil Handigol et al. "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks". In: *NSDI*. 2014, pp. 71–85.

[Hof+14] Rick Hofstede et al. "Flow monitoring explained: From packet capture to data analysis with netflow and ipfix". In: *IEEE Communications Surveys & Tutorials* 16.4 (2014), pp. 2037–2064.

[Hon+11] Michio Honda et al. "Is it still possible to extend TCP?" In: *IMC'11*. ACM. 2011, pp. 181–194.

[Hon+13] C. Hong et al. "Achieving High Utilization with Software-Driven WAN". In: *ACM SIGCOMM*. 2013.

[Hua+17] Qun Huang et al. “SketchVisor: Robust Network Measurement for Software Packet Processing”. In: *SIGCOMM*. 2017, pp. 113–126.

[Jai+07] Sharad Jaiswal et al. “Measurement and classification of out-of-sequence packets in a tier-1 IP backbone”. In: *IEEE/ACM Transactions on Networking (ToN)* 15.1 (2007), pp. 54–66.

[Jai+13] S. Jain et al. “B4: Experience with a Globally-Deployed Software Defined WAN”. In: *ACM SIGCOMM*. 2013.

[JG85] David S Johnson and Michael R Garey. “A 7160 theorem for bin packing”. In: *Journal of Complexity* 1.1 (1985), pp. 65–106.

[Jin+14] Xin Jin et al. “Dynamic scheduling of network updates”. In: *ACM SIGCOMM*. 2014.

[Jin+17] Xin Jin et al. “Netcache: Balancing key-value stores with fast in-network caching”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, pp. 121–136.

[Jin+18] Xin Jin et al. “NetChain: Scale-Free Sub-RTT Coordination”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA, 2018, pp. 35–49. ISBN: 978-1-931971-43-0.

[JTO10] Wolfgang John, Sven Tafvelin, and Tomas Olovsson. “Passive internet measurement: Overview and guidelines based on experiences”. In: *Computer Communications* 33.5 (2010), pp. 533–550.

[KAB09] James Kelly, Wladimir Araujo, and Kallol Banerjee. “Rapid service creation using the JUNOS SDK”. In: *Proceedings of the 2nd ACM SIGCOMM workshop on Programmable routers for extensible services of tomorrow*. ACM. 2009, pp. 7–12.

[Kah62] A. B. Kahn. “Topological Sorting of Large Networks”. In: *Commun. ACM* 5.11 (Nov. 1962), pp. 558–562. DOI: 10.1145/368996.369025.

[Kaz+13] Peyman Kazemian et al. “Real Time Network Policy Checking Using Header Space Analysis”. In: *NSDI*. 2013, pp. 99–111.

[Khu+13] Ahmed Khurshid et al. “VeriFlow: Verifying Network-Wide Invariants in Real Time”. In: *NSDI*. 2013, pp. 49–54.

[Kim+15] Changhoon Kim et al. “In-band network telemetry via programmable dataplanes”. In: *ACM SIGCOMM*. 2015.

[Kni+11] Simon Knight et al. “The Internet Topology Zoo”. In: *IEEE Journal on Selected Areas in Communications* 29 (2011), pp. 1765–1775.

[Kop+10] Teemu Koponen et al. “Onix: A Distributed Control Platform for Large-scale Production Networks”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. Vancouver, BC, Canada, 2010, pp. 1–6.

[Kre+15] Diego Kreutz et al. “Software-defined networking: A comprehensive survey”. In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76.

[KVM12] Peyman Kazemian, George Varghese, and Nick McKeown. “Header Space Analysis: Static Checking for Networks”. In: *NSDI*. 2012, pp. 113–126.

[Lan+17] Adam Langley et al. “The QUIC Transport Protocol: Design and Internet-Scale Deployment”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. Los Angeles, CA, USA, 2017, pp. 183–196. DOI: 10.1145/3098822.3098842.

[LCC09] Xiapu Luo, Edmond WW Chan, and Rocky KC Chang. “Design and Implementation of TCP Data Probes for Reliable and Metric-Rich Network Path Monitoring.” In: *USENIX Annual Technical Conference*. 2009.

[Lev+14] Dan Levin et al. “Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks”. In: *USENIX Annual Technical Conference*. USENIX Association. 2014, pp. 333–345.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. “A network in a laptop: rapid prototyping for software-defined networks”. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM. 2010, p. 19.

[Li+13] Bingdong Li, Jeff Springer, George Bebis, and Mehmet Hadi Gunes. “A survey of network flow applications”. In: *Journal of Network and Computer Applications* 36.2 (2013), pp. 567–581.

[Li+16] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. “FlowRadar: A Better NetFlow for Data Centers”. In: *NSDI*. 2016, pp. 311–324.

[Lop+15] Nuno P Lopes et al. “Checking Beliefs in Dynamic Networks.” In: *NSDI*. 2015, pp. 499–512.

[Mai+11] Haohui Mai et al. “Debugging the Data Plane with Anteater”. In: *SIGCOMM Comput. Commun. Rev.* 41 (2011), pp. 290–301.

[Mar+08] A. Markopoulou et al. “Characterization of failures in an operational IP backbone network”. In: *Trans. on Netw.* 16 (4 2008), pp. 749–762. ISSN: 1063-6692.

[McK+08] Nick McKeown et al. “OpenFlow: Enabling innovation in campus networks”. In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[Mel02] Marco Mellia. “TCP Statistic and Analysis Tool”. In: *IEEE Network* 16.5 (Sept. 2002).

[Mos+16] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. “Trumpet: Timely and Precise Triggers in Data Centers”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM ’16. Florianopolis, Brazil, 2016, pp. 129–143. DOI: 10.1145/2934872.2934879.

[Nar+16] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. “Compiling Path Queries”. In: *NSDI*. 2016, pp. 207–222.

[Nar+17] Srinivas Narayana et al. “Language-Directed Hardware Design for Network Performance Monitoring”. In: *SIGCOMM*. 2017, pp. 85–98.

[NKR13] Krisztian Nemeth, Attila Korosi, and Gabor Retvari. “Optimal OSPF traffic engineering using legacy Equal Cost Multipath load balancing”. In: *IFIP Networking Conference, 2013*. IEEE. 2013, pp. 1–9.

[Pap+17] Giorgos Papastergiou et al. “De-ossifying the internet transport layer: A survey and future perspectives”. In: *IEEE Communications Surveys & Tutorials* 19.1 (2017), pp. 619–639.

[Pfa+15] Ben Pfaff et al. “The Design and Implementation of Open vSwitch.” In: *NSDI*. 2015, pp. 117–130.

[Pou+17] Konstantinos Poularakis, George Iosifidis, Georgios Smaragdakis, and Leandros Tassiulas. “One step at a time: Optimizing SDN upgrades in ISP networks”. In: *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*. IEEE. 2017, pp. 1–9.

[Qaz+13] Zafar Ayyub Qazi et al. “SIMPLE-fying Middlebox Policy Enforcement Using SDN”. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM ’13. Hong Kong, China, 2013, pp. 27–38. DOI: 10.1145/2486001.2486022.

[Ras+14] Jeff Rasley et al. “Planck: Millisecond-scale Monitoring and Control for Commodity Networks”. In: *SIGCOMM*. 2014, pp. 407–418.

[RI07] Alex Raj and Oliver C Ibe. “A survey of IP and multiprotocol label switching fast reroute schemes”. In: *Computer Networks* 51.8 (2007), pp. 1882–1907.

[RIP15] RIPE NCC. “Ripe atlas: A global internet measurement network”. In: *Internet Protocol Journal* 18.3 (2015).

[Rot+12] Charalampos Rotsos et al. “OFLOPS: An Open Framework for Openflow Switch Evaluation”. In: *Conference on Passive and Active Measurement*. Vienna, Austria, 2012, pp. 85–95. DOI: 10.1007/978-3-642-28537-0_9.

[Rüt+18] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. “A First Look at QUIC in the Wild”. In: *Passive and Active Measurement*. Springer International Publishing, 2018, pp. 255–268. ISBN: 978-3-319-76481-8.

[Sek+06] Vyas Sekar et al. “LADS: Large-scale Automated DDoS Detection System.” In: *USENIX Annual Technical Conference, General Track*. 2006, pp. 171–184.

[SGD05] Ashwin Sridharan, Roch Guérin, and Christophe Diot. “Achieving near-optimal traffic engineering solutions for current OSPF/IS-IS networks”. In: *IEEE/ACM Transactions on Networking (TON)* 13.2 (2005), pp. 234–247.

[Sin+15] Arjun Singh et al. “Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 45. 4. ACM. 2015, pp. 183–197.

[SMW02] Neil Spring, Ratul Mahajan, and David Wetherall. “Measuring ISP Topologies with Rocketfuel”. In: *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’02. Pittsburgh, Pennsylvania, USA, 2002, pp. 133–145. DOI: 10.1145/633025.633039.

[Sou+14] Robert Soulé et al. “Merlin: A Language for Provisioning Network Resources”. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT ’14. Sydney, Australia, 2014, pp. 213–226. DOI: 10.1145/2674005.2674989.

[Ste+15] Daniël van der Steeg, Rick Hofstede, Anna Sperotto, and Aiko Pras. “Real-time DDoS attack detection for Cisco IOS using NetFlow”. In: *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE. 2015, pp. 972–977.

[Sto+16] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. “SymNet: Scalable Symbolic Execution for Modern Networks”. In: *SIGCOMM*. 2016, pp. 314–327.

[Sun+11] Srikanth Sundaresan et al. “Broadband internet performance: a view from the gateway”. In: *ACM SIGCOMM computer communication review*. Vol. 41. 4. ACM. 2011, pp. 134–145.

[T K14] T. Koponen et al. “Network Virtualization in Multi-tenant Datacenters”. In: *NSDI*. 2014.

[Tav+09] Arsalan Tavakoli, Martin Casado, Teemu Koponen, and Scott Shenker. “Applying NOX to the Datacenter.” In: *HotNets*. 2009.

[TB11] Brian Trammell and Elisa Boschi. “An introduction to IP flow information export (IPFIX)”. In: *IEEE Communications Magazine* 49.4 (2011).

[Tre+17] Martino Trevisan et al. “Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learne”. In: *IEEE Communications Magazine* (Mar. 2017).

[Uhl+06] Steve Uhlig, Bruno Quoitin, Jean Lepropre, and Simon Balon. “Providing public intradomain traffic matrices to the research community”. In: *ACM SIGCOMM Computer Communication Review* 36.1 (2006), pp. 83–86.

[Van+11] L. Vanbever et al. “Seamless Network-Wide IGP Migrations”. In: *Proc. SIGCOMM*. 2011.

[Van+13] L. Vanbever, S. Vissicchio, L. Cittadini, and O. Bonaventure. “When the cure is worse than the disease: The impact of graceful IGP operations on BGP”. In: *2013 Proceedings IEEE INFOCOM*. Apr. 2013, pp. 2220–2228. DOI: 10.1109/INFCOM.2013.6567025.

[Ver+07] Patrick Verkaik et al. “Wresting Control from BGP: Scalable Fine-Grained Route Control.” In: *USENIX Annual Technical Conference*. 2007, pp. 295–308.

[Vie+18] Tobias Viernickel et al. “Multipath QUIC: A Deployable Multipath Transport Protocol”. In: *2018 IEEE International Conference on Communications (ICC)*. IEEE. 2018, pp. 1–7.

[Vis+10] Stefano Vissicchio et al. “Beyond the Best: Real-time Non-invasive Collection of BGP Messages”. In: *INM/WREN*. 2010.

[VVB14] Stefano Vissicchio, Laurent Vanbever, and Olivier Bonaventure. "Opportunities and Research Challenges of Hybrid Software Defined Networks". In: *SIGCOMM Comput. Commun. Rev.* 44.2 (Apr. 2014), pp. 70–75. ISSN: 0146-4833. DOI: 10.1145/2602204.2602216.

[VVR14] Stefano Vissicchio, Laurent Vanbever, and Jennifer Rexford. "Sweet Little Lies: Fake Topologies for Flexible Routing". In: *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. HotNets-XIII. Los Angeles, CA, USA: ACM, 2014, 3:1–3:7. DOI: 10.1145/2670518.2673868.

[Wan+08] N. Wang et al. "An overview of routing optimization for internet traffic engineering". In: *Communications Surveys & Tutorials, IEEE* (2008).

[Wei+16] Konstantin Weitz et al. "Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver". In: *ACM SIGPLAN Notices* 51 (2016), pp. 765–780.

[Wun+11] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. "OFRewind: Enabling Record and Replay Troubleshooting for Networks". In: *USENIX ATC*. 2011, pp. 15–17.

[Xu+17] Hongli Xu et al. "Incremental deployment and throughput maximization routing for a hybrid SDN". In: *IEEE/ACM Transactions on Networking (TON)* 25.3 (2017), pp. 1861–1875.

[YCM11] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. "ProgME: towards programmable network measurement". In: *IEEE/ACM Transactions on Networking (TON)* 19 (2011), pp. 115–128.

[YRW17] Bahador Yeganeh, Reza Rejaie, and Walter Willinger. "A view from the edge: A stub-AS perspective of traffic localization and its implications". In: *Network Traffic Measurement and Analysis Conference (TMA), 2017*. IEEE. 2017, pp. 1–9.

[Yu+11] Minlan Yu et al. "Profiling Network Performance for Multi-tier Data Center Applications." In: *NSDI*. Vol. 11. 2011, pp. 5–5.

[Yua+17] Yifei Yuan et al. "Quantitative Network Monitoring with NetQRE". In: *SIGCOMM*. 2017, pp. 99–112.

[Zan+18] Saeed Barkabi Zanjani, Kuang-Yi Li, Steven SW Lee, and Yuan-Sun Chu. "Path Provisioning for Fibbing Controlled Load Balanced IP Networks". In: *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE. 2018, pp. 154–158.

[Zha+04] Yin Zhang et al. “Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications”. In: *SIGCOMM*. 2004, pp. 101–114.

[Zhu+15] Yibo Zhu et al. “Packet-Level Telemetry in Large Datacenter Networks”. In: *SIGCOMM*. 2015, pp. 479–491.

Internet standards and drafts

[RFC1131] J. Moy. *OSPF specification*. RFC 1131. RFC Editor, Oct. 1989.

[RFC1142] D. Oran. *OSI IS-IS Intra-domain Routing Protocol*. RFC 1142. RFC Editor, Feb. 1990.

[RFC1157] Jeffrey D. Case, Mark Fedor, Martin Lee Schoffstall, and James R. Davin. *Simple Network Management Protocol (SNMP)*. STD 15. RFC Editor, May 1990.

[RFC2328] John Moy. *OSPF Version 2*. STD 54. RFC Editor, Apr. 1998.

[RFC2453] Gary Scott Malkin. *RIP Version 2*. STD 56. RFC Editor, Nov. 1998.

[RFC2460] Stephen E. Deering and Robert M. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. RFC Editor, Dec. 1998.

[RFC2474] Kathleen Nichols, Steven Blake, Fred Baker, and David L. Black. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC 2474. RFC Editor, Dec. 1998.

[RFC2784] Dino Farinacci et al. *Generic Routing Encapsulation (GRE)*. RFC 2784. RFC Editor, Mar. 2000.

[RFC3031] E. Rosen, A. Viswanathan, and R. Callon. *Multiprotocol Label Switching Architecture*. RFC 3031. RFC Editor, Jan. 2001.

[RFC3176] P. Phaal, S. Panchen, and N. McKee. *InMon Corporation’s sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. RFC 3176. RFC Editor, Sept. 2001.

[RFC3209] D. Awduche et al. *RSVP-TE: Extensions to RSVP for LSP Tunnels*. RFC 3209 (Proposed Standard). Updated by RFCs 3936, 4420, 4874. Dec. 2001.

[RFC3443] P. Agarwal and B. Akyol. *Time To Live (TTL) Processing in Multi-Protocol Label Switching (MPLS) Networks*. RFC 3443. RFC Editor, Jan. 2003.

[RFC3954] B. Claise. *Cisco Systems NetFlow Services Export Version 9*. RFC 3954. RFC Editor, Oct. 2004.

[RFC4022] R. Raghunarayan. *Management Information Base for the Transmission Control Protocol (TCP)*. RFC 4022. RFC Editor, Mar. 2005.

[RFC4251] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. RFC Editor, Jan. 2006.

[RFC4271] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. RFC Editor, Jan. 2006.

[RFC4655] A. Farrel, J.-P. Vasseur, and J. Ash. *A Path Computation Element (PCE)-Based Architecture*. RFC 4655. RFC Editor, Aug. 2006.

[RFC4664] L. Andersson and E. Rosen. *Framework for Layer 2 Virtual Private Networks (L2VPNs)*. RFC 4664. RFC Editor, Sept. 2006.

[RFC4915] P. Psenak et al. *Multi-Topology (MT) Routing in OSPF*. RFC 4915. RFC Editor, June 2007.

[RFC5036] L. Andersson, I. Minei, and B. Thomas. *LDP Specification*. RFC 5036. RFC Editor, Oct. 2007.

[RFC5234] D. Crocker and P. Overell. *Augmented BNF for Syntax Specifications: ABNF*. STD 68. RFC Editor, Jan. 2008.

[RFC5340] R. Coltun, D. Ferguson, J. Moy, and A. Lindem. *OSPF for IPv6*. RFC 5340. RFC Editor, July 2008.

[RFC5575] P. Marques et al. *Dissemination of Flow Specification Rules*. RFC 5575. RFC Editor, Aug. 2009.

[RFC5610] E. Boschi, B. Trammell, L. Mark, and T. Zseby. *Exporting Type Information for IP Flow Information Export (IPFIX) Information Elements*. RFC 5610. RFC Editor, July 2009.

[RFC5714] M. Shand and S. Bryant. *IP Fast Reroute Framework*. RFC 5714. RFC Editor, Jan. 2010.

[RFC5880] D. Katz and D. Ward. *Bidirectional Forwarding Detection (BFD)*. RFC 5880. RFC Editor, June 2010.

[RFC6241] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. *Network Configuration Protocol (NETCONF)*. RFC 6241. RFC Editor, June 2011.

[RFC6555] D. Wing and A. Yourtchenko. *Happy Eyeballs: Success with Dual-Stack Hosts*. RFC 6555. RFC Editor, Apr. 2012.

[RFC6571] C. Filsfils et al. *Loop-Free Alternate (LFA) Applicability in Service Provider (SP) Networks*. RFC 6571. RFC Editor, June 2012.

[RFC6824] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824. RFC Editor, Jan. 2013.

[RFC7348] M. Mahalingam et al. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. RFC Editor, Aug. 2014.

[RFC7684] P. Psenak et al. *OSPFv2 Prefix/Link Attribute Advertisement*. RFC 7684. RFC Editor, Nov. 2015.

[RFC791] Jon Postel. *Internet Protocol*. STD 5. RFC Editor, Sept. 1981.

[RFC7911] D. Walton, A. Retana, E. Chen, and J. Scudder. *Advertisement of Multiple Paths in BGP*. RFC 7911. RFC Editor, July 2016.

[RFC7921] A. Atlas et al. *An Architecture for the Interface to the Routing System*. RFC 7921. RFC Editor, June 2016.

[RFC793] Jon Postel. *Transmission Control Protocol*. STD 7. RFC Editor, Sept. 1981.

[RFC8151] L. Yong et al. *Use Cases for Data Center Network Virtualization Overlay Networks*. RFC 8151. RFC Editor, May 2017.

[RFC8402] C. Filsfils et al. *Segment Routing Architecture*. RFC 8402. RFC Editor, July 2018.

[Emi+17] Stephan Emile, Mathilde Cayla, Arnaud Braud, and Frederic Fieau. *QUIC Interdomain Troubleshooting*. Internet-Draft draft-stephan-quic-interdomain-troubleshooting-00. IETF Secretariat, July 2017.

[Fil+18] Clarence Filsfils et al. *SRv6 Network Programming*. Internet-Draft draft-filsfils-spring-srv6-network-programming-05. IETF Secretariat, July 2018.

[IT18] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-13. IETF Secretariat, June 2018.

[Pse+18] Peter Psenak et al. *IGP Flexible Algorithm*. Internet-Draft draft-ietf-lsr-flex-algo-00. IETF Secretariat, May 2018.

[Tra+18] Brian Trammell et al. *Adding Explicit Passive Measurability of Two-Way Latency to the QUIC Transport Protocol*. Internet-Draft draft-trammell-quic-spin-03. IETF Secretariat, May 2018.

Online resources

[ChroLog] The Chromium project. *How to enable loggin in Chromium*. URL: `https://www.chromium.org/for-testers/enable-logging`.

[ASN.1] ITU. *Introduction to ASN.1*. URL: https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx.

[BIRD] Ondrej Filip, Pavel Machek, and Martin Mares. *BIRD Internet Routing Daemon*. URL: https://http://bird.network.cz/.

[EWT] Microsoft. *Network Tracing in Windows 7: Architecture*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dd569137(v=vs.85).aspx.

[GNS3] Galaxy Technologies LLC. *Graphical Network Simulator - 3*. URL: https://www.gns3.com/.

[ONOS] The Open Network Foundation (ONF). *ONOS: Open Network Operating System*. URL: http://onosproject.org//.

[Ana] Julian Anastasov. *ip-tcp_metrics - management for TCP Metrics*. URL: https://www.linux.org/docs/man8/ip-tcp_metrics.html.

[App17] Apple. *Advances in Networking*. 2017. URL: https://developer.apple.com/videos/play/wwdc2017/707/.

[apsf] The Apache Software Foundation. *ab - Apache HTTP server benchmarking tool*. URL: https://httpd.apache.org/docs/2.4/programs/ab.html.

[Arr16] Ferran Llamas Arroniz. "Flexible Traffic Engineering in existing networks with Fibbing". MA thesis. ETH Zurich, 2016.

[BDL04] A. Bobyshev, P. DeMar, and D. Lamore. "Effect of dynamic ACL (access control list) loading on performance of Cisco routers". In: *Computing in High Energy Physics*. 2004.

[Cisa] Cisco. *IP SLAs Configuration Guide*. URL: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipsla/configuration/xe-16/sla-xe-16-book.html.

[Cisb] Cisco Systems. *Cisco OSPF MD5 Authentication*. URL: http://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/13697-25.html.

[Cisc] Cisco Systems. *Cisco Python API*. URL: http://bit.ly/2fMgyKP.

[Cisd] Cisco Systems. *Cisco Visual Networking Index: Forecast and Methodology, 2016–2021*. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf.

[Cise] Cisco Systems. *Cisco. Configuring Policy-Based Routing*. URL: http://www.cisco.com/c/en/us/td/docs/ios/12_2/qos/configuration/guide/fqos_c/qcfpbr.html.

[Cisf] Cisco Systems. *Cisco. Implementing BGP Flowspec*. URL: http://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/asr9k_r5-2/routing/configuration/guide/b_routing_cg52xasr9k/b_routing_cg52xasr9k_chapter_011.html.

[Cisg] Cisco Systems. *Common Routing Problem with OSPF Forwarding Address*. URL: https://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/13682-10.html#topic1.

[Cish] Cisco Systems. *Route-Maps for IP Routing Protocol Redistribution Configuration*. URL: https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/49111-route-map-bestp.html.

[Cis13] Cisco Tech Support. *CEF Polarization*. https://goo.gl/b7ZSMy. 2013.

[Cis16] Cisco Systems. *Configuring ERSPAN*. https://goo.gl/h3qaGL. 2016.

[CLS] Bryan Cantrill, Adam Leventhal, and Mike Shapiro. *About DTrace*. URL: dtrace.org.

[Floodlight] Project Floodlight. *The Floodlight OpenFlow controller*. URL: http://www.projectfloodlight.org/floodlight/.

[Gos05] Sudhanshu Goswami. *An introduction to KProbes*. April 18, 2005. URL: https://lwn.net/Articles/132196/.

[IO] IO Visor Project. *eBPF, extended Berkeley Packet Filter*. URL: https://www.iovisor.org/technology/ebpf.

[iperf] Jon Dugan et al. *iPerf–The TCP, UDP, and SCTP network bandwidth measurement tool*. URL: https://iperf.fr/.

[Juna] Juniper. *Juniper OSPF MD5 Authentication*. URL: http://www.juniper.net/documentation/en_US/junos14.2/topics/topic-map/ospf-authentication.html.

[Junb] Juniper. *Juniper. Configuring Filter-Based Forwarding to a Specific Outgoing Interface or Destination IP Address*. URL: http://www.juniper.net/techpubs/en_US/junos12.2/topics/topic-map/filter-based-forwarding-policy-based-routing.html.

[Junc] Juniper. *Juniper. Enabling BGP to Carry Flow-Specification Routes.* URL: https://www.juniper.net/documentation/en_US/junos12.3/topics/example/routing-bgp-flow-specification-routes.html.

[Jund] Juniper. *Junos Automation Scripts Overview.* URL: https://goo.gl/WpjAcX.

[June] Juniper. *Route Map Overview.* URL: https://www.juniper.net/documentation/en_US/nsm2012.2/topics/concept/security-service-firewall-screenos-route-map-overview.html.

[Jun08] Juniper. *What's Behind Network Downtime?* 2008. URL: https://www-935.ibm.com/services/au/gts/pdf/200249.pdf.

[Jun14] Juniper Networks. *Layer 2 Port Mirroring Overview.* https://goo.gl/YxgZuY. 2014.

[ker] The Linux kernel. *struct tcp_info definition.* URL: https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/tcp.h#L168.

[Mol16] Edgar Costa Molero. "Improving Load-Balancing Decisions in Data Center Networks Using Software-Defined Networking". MA thesis. ETH Zurich, 2016.

[Nak+13] Gabi Nakibly, Eitan Menahem, Ariel Waizel, and Yuval Elovici. "Owning the routing table, part II". 2013.

[ntttcp] Microsoft. *NTTTCP-for-Linux.* URL: https://github.com/Microsoft/ntttcp-for-linux.

[Ope] OpenConfig. *Openconfig: Vendor-neutral, model-driven network management designed by users.* URL: https://openconfig.net.

[openr] Facebook Inc. *OpenR: Open Routing.* URL: https://github.com/facebook/openr.

[PBa] C. Paasch, S. Barre, and et al. *Multipath TCP in the Linux Kernel.* URL: https://www.multipath-tcp.org.

[Pea14] K. Pearce. "Multipath TCP Breaking today's networks with tomorrow's protocol". 2014.

[perf] The Linux kernel. *perf: Linux profiling with performance counters.* URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[Quagga] Paul Jakma and et al. *Quagga Routing Suite.* URL: https://www.quagga.net.

[San15] Omar Santos. *Network Security with NetFlow and IPFIX: Big Data Analytics for Information Security*. Cisco Press, 2015.

[Sch] Arnaud Schills. *Packetdrill test suite for MPTCP*. URL: https://github.com/aschils/packetdrill_mptcp.

[She14] Scott Shenker. *Time for an SDN Sequel? Scott Shenker Preaches SDN Version 2*. 2014. URL: https://www.sdxcentral.com/articles/news/scott-shenker-preaches-revised-sdn-sdnv2/2014/10/.

[TDH+] Olivier Tilmans, Gregory Detal, Benjamin Hesmans, et al. *tracebox: A middlebox detection tool*. URL: http://tracebox.org.

[Thea] The IOvisor project. *BPF Compiler Collection (bcc)*. URL: https://www.iovisor.org/technology/bcc.

[Theb] The Linux Foundation. *TCP Probe*. URL: https://wiki.linuxfoundation.org/networking/tcpprobe.

[Ver16] Veriflow. *Network Complexity, Change, and Human Factors Are Failing the Business*. Ed. by Dimensional Research. 2016. URL: https://www.veriflow.net/survey/.

[Wea] Weaverworks. *tcptracer-bpf*. URL: https://github.com/weaveworks/tcptracer-bpf.