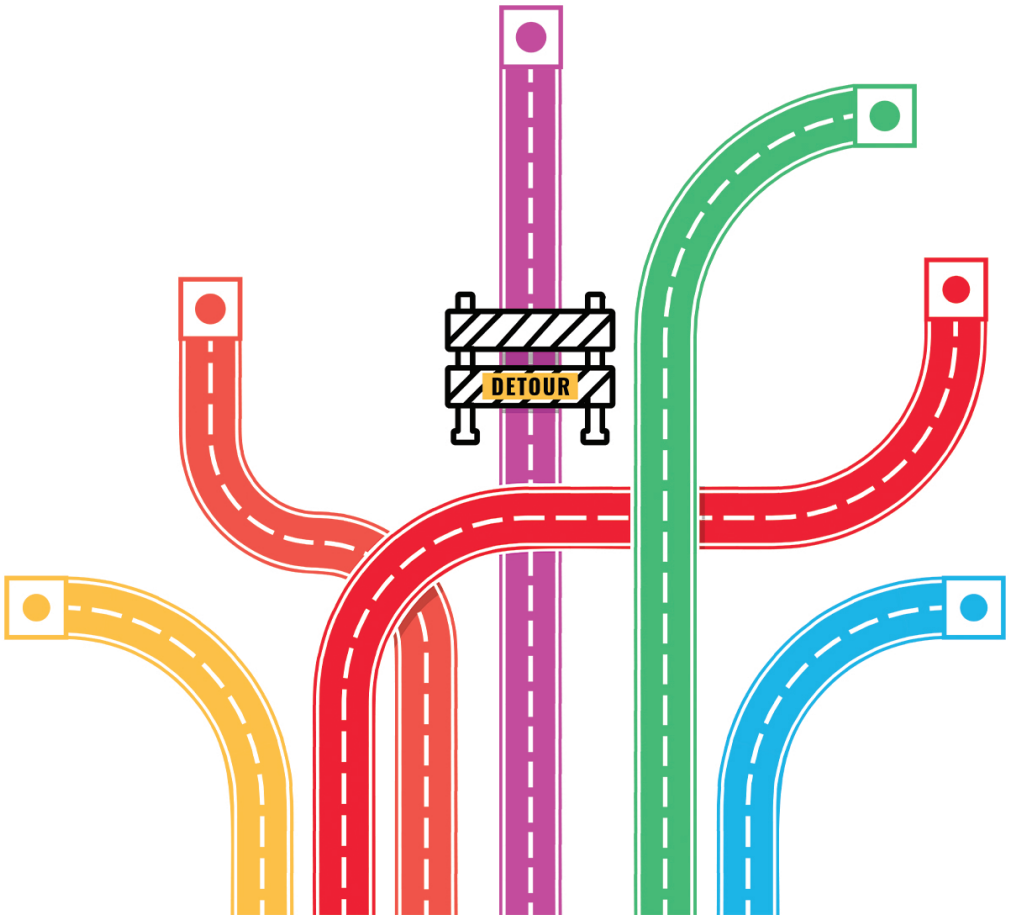
THOMAS HOLTERBACH

A FRAMEWORK TO FAST REROUTE TRAFFIC UPON REMOTE OUTAGES

DISS. ETH NO. 27857



Diss. ETH No. 27857

A Framework to Fast Reroute Traffic Upon Remote Outages

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich
(Dr. sc. ETH Zurich)

presented by
Thomas Holterbach

born on 31.07.1991
citizen of France

accepted on the recommendation of
Prof. Dr. Laurent Vanbever, examiner
Prof. Dr. Adrian Perrig, co-examiner
Prof. Dr. Kimberly Claffy, co-examiner

2021

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 191

Thomas Holterbach

A Framework to Fast Reroute Traffic Upon Remote Outages

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to ETH Zurich
for the degree of Doctor of Sciences

DISS. ETH NO. 27857

Prof. Dr. Laurent Vanbever, examiner
Prof. Dr. Adrian Perrig, co-examiner
Prof. Dr. Kimberly Claffy, co-examiner

Examination date: August 24, 2021

Abstract

Nowadays, so many services – including critical ones – rely on the Internet to work that even a few minutes of connectivity disruption make customers unhappy and cause sizeable financial loss for companies. Ensuring that customers are always connected to the Internet is thus a top priority for Internet service providers. However, this is harder than one may think because the Internet is often subject to network outages.

Network outages are a headache for network operators because they are unpredictable, can occur in any of the 70,000 independently operated networks composing the Internet, and can affect users' connectivity network-wide. Far too often, the only way to restore connectivity upon an outage is to wait that *(i)* BGP, the glue of the Internet, converges; and *(ii)* the routers update their forwarding decisions accordingly. Unfortunately, these two processes work on a per-destination basis and are thus inherently slow given the always-increasing number of destinations in the Internet. It is therefore not a surprise that network operators still experience minutes of downtime upon outages.

In this dissertation, we tackle the problem of fast connectivity recovery upon outages occurring in remote networks, without requiring network operators to change the standards, manufacture new devices, or cooperate with each other. The final result of our work is Snap, a framework that network operators can deploy on their routers and allows them to quickly detect outages and reroute traffic to working alternative paths that comply with the configured routing policies. Snap's design follows a two-step recipe. First, it uses an outage inference algorithm based on new fundamental results and which, instead of waiting for the slow control-plane (BGP) notifications, analyzes the fast data-plane signals. Second, it uses a rerouting scheme that allows routers to quickly reroute all the affected traffic to alternative paths circumventing the outage.

Snap's design takes advantage of the recent advances in network programmability and relies on a hardware-software codesign. To be fast, Snap collects data-plane signals at line-rate using programmable switches (*e.g.*, Tofino). The switches then mirror the signals to a controller, which accurately infers remote outages and triggers traffic rerouting. We implemented Snap in $P4_{16}$ and Python and show its effectiveness in many real-world situations. Our results indicate that Snap can restore connectivity within a few seconds only, which is much faster than the few minutes often needed by traditional routers.

Résumé

De nos jours, il y a tellement de services – dont certains essentiels – qui dépendent d'Internet que quelques minutes seulement de pertubation sur la connexion suffisent à rendre les utilisateurs mécontents et peuvent causer de lourdes pertes financières aux entreprises. S'assurer que les clients soient toujours connectés à l'Internet est donc une priorité absolue pour les opérateurs réseaux. Cependant, cela est plus difficile qu'il n'y paraît, parce que l'Internet est souvent sujet à des pannes réseaux.

Les pannes réseaux sont un casse-tête pour les opérateurs réseaux parce qu'elles sont imprévisibles, peuvent arriver dans chacun des 70000 réseaux autonomes formant l'Internet, et peuvent affecter la connectivité sur l'ensemble du réseau. Bien trop souvent, la seule façon de rétablir la connectivité lors d'une panne est d'attendre que *(i)* BGP, le protocole de routage qui fait fonctionner l'Internet, converge et que *(ii)* les routeurs mettent à jour leurs décisions pour envoyer les paquets vers un nouveau chemin. Malheureusement, ces deux tâches exigent la réalisation d'actions pour chacune des destinations affectées par la panne et sont par conséquent intrinséquement lentes étant donné que le nombre de destinations dans l'Internet ne fait qu'augmenter avec le temps.

Dans cette thèse, on s'attaque au problème de rétablissement rapide de la connectivité lors de pannes arrivant dans des réseaux distants, sans exiger des opérateurs réseaux de modifier les normes, de produire de nouveaux équipements, ou de coopérer entre eux. Le résultat final de cette thèse est Snap, un système que les opérateurs réseaux peuvent déployer sur leurs routeurs, leur permettant de rapidement détecter les pannes et rerouter le trafic vers des routes alternatives qui fonctionnent et qui respectent les politiques de routages configurées. Le design de Snap s'appuie sur deux ingrédients clés. Premièrement, il utilise un algorithme d'inférence de pannes basé sur de nouveaux résultats fondamentaux et qui, au lieu d'attendre les lentes notifications du plan de contrôle (BGP), analyse les signaux rapides du plan de données. Deuxièmement, il utilise un procédé de reroutage permettant aux routeurs de rapidement rerouter tout le trafic affecté vers des chemins alternatifs qui contournent la panne.

Le design de Snap tire profit des récents progrès qui rendent les réseaux plus programmables et utilise intelligement leurs capacités au niveau hardware et software. Pour être rapide, les signaux du plan de données sont collectés par des switches programmables à la même vitesse que le débit du trafic. Les switches envoient ensuite les signaux vers un controller, qui infère précisement les pannes

distantes et déclanche le reroutage du trafic. Nous avons implémenté Snap en $P4_{16}$ et Python et nous montrons son efficacité dans beaucoup de scénarios que l'on peut trouver dans la vraie vie. Nos résultats indiquent que Snap est capable de rétablir la connectivité en quelques secondes seulement, ce qui est bien plus rapide que les minutes souvent requises par les routeurs traditionnels.

Preamble

When I tell people that I research how to improve the Internet, they often wonder what I do and have no clue that there is still research in this area possible and needed. This reaction is likely a good sign, as it means that the Internet works just fine for the ordinary man.

Because people are relatively happy with it, the Internet is taking a more and more important part in our daily lives. Now, even people that used to be unfamiliar with computer science use it to buy, learn, work, play or drive, among others. We also see more and more critical services such as remote surgeries or banking systems relying on the Internet. It is not a surprise that half of the ten companies with the highest market capitalization are IT companies providing Internet services [6].

The problem is that by making all our services Internet-driven, it is not enough for the Internet to work just fine. Nowadays, even a few minutes of downtime can have serious effects. Imagine, for instance, how angry you would be if once in a while your virtual call with your clients failed or if you could not load the directions while driving in LA for the first time. And it can be far worse than that: no later than in June 2021, an outage in Orange's network made the emergency numbers in France unavailable for a few hours [29]. Of course, it goes without saying that Internet outages also cause large financial loss for companies [4].

To work better and keep up with the more and more demanding applications and the growing number of users, the Internet is constantly improved over time thanks to the effort provided by the community. With this thesis, we hope to contribute to that effort. Our contributions, which we describe in the following paragraphs, focus on improving the quality of the Internet upon outages.

Focus on the Internet reaction to failures. The Internet is a set of around 70,000 independently operated networks, also called Autonomous Systems (ASes), which are interconnected together. What makes end-to-end connectivity possible in this large network of networks is routing automation, using distributed protocols, along with dedicated devices to quickly forward packets. While this process is transparent for end-users, it involves a number of challenges which come from the distributed nature of the Internet and its business-driven decisions, among others.

This thesis focuses on improving the Internet convergence upon failures, which is a problem that has occupied the mind of researchers for decades. The Internet

convergence is the process during which the Internet automatically adjusts itself upon a change. We illustrate the Internet convergence by making an analogy with the road network. When the road map changes (*e.g.*, a road is closed because it needs to be fixed), the road network staff updates the signs on the streets to divert cars on new paths. In the Internet, packets (cars) are routed on links (roads) that are interconnected via routers (intersections). The routers rely on distributed routing protocols during the convergence to automatically learn the new routes and update their forwarding entries (signs).

Now, think about all the time you spent in a traffic jam while driving a car because of an accident. Unfortunately, the same issues arise in the Internet upon failures, except that instead of being delayed, packets are often dropped. Despite the effort of the road network staff trying to react swiftly, car accidents often result in traffic jams because they are unpredictable. Be it in the road network or in the Internet, unpredictable events trigger an inevitably long convergence process, which includes: reporting the incident, computing the new routes for all the destinations, and finally installing the new forwarding decisions.

Why is the Internet convergence so much of a problem? There is a fundamental difference between the road network and the Internet, which makes the Internet convergence process everything but easy. In the road network, as drivers are intelligent, only a few signs (*e.g.*, for a few of the nearby cities) are enough at every intersection to guide them to their final destination. Unfortunately, this is not the case in the Internet, where routers must perform routing and packets forwarding for *every* destination.

Take, for instance, the Border Gateway Protocol (BGP), which is the routing protocol used in the Internet to exchange reachability information between the different ASes. Upon a change, BGP works on a per-destination basis: it propagates routing information for every destination concerned by the change. The same goes in the forwarding plane of a router, where a flat forwarding table requires the router to maintain one forwarding entry for every destination.

In the Internet, we currently observe around 865k destinations, also known as IP prefixes, and this number continuously increases over time. As a result, a local change in an AS can trigger complex network-wide operations at the routing and forwarding level. In this thesis, we show that this process, *i.e.*, the Internet convergence, can easily take *minutes*, a time during which traffic can be lost. It is thus not a surprise that a survey which we conducted among 73 network operators indicates that they are concerned about the slow Internet convergence and looking for solutions to prevent the resulting downtime.

Why are the existing solutions not sufficient? The good news is that there are solutions to prevent connectivity loss upon failures. A widespread approach, which does not require modifying the standards, is fast rerouting the traffic towards working backup paths while the network is converging.

At the control-plane level, this involves pre-computing backup paths for every possible failure and destination. The backup paths must restore connectivity and thus avoid the failure and do not result in forwarding inconsistencies such as forwarding loops, which are the typical negative side effects when rerouting traffic without notifying the other routers in the network.

At the data-plane level, this involves *(i)* pre-populating the backup routes in the forwarding table of the routers; and *(ii)* using a tailored rerouting scheme to ensure fast activation of the backup routes, regardless of the number of IP prefixes concerned by the failure. In practice, this is often achieved with a hierarchical forwarding table, which allows grouping prefixes sharing the same primary and backup paths to reroute all of them with a single data-plane update.

Unfortunately, the currently deployed fast reroute solutions, *e.g.*, IP Fast Reroute for intra-domain routing and BGP Prefix Independent Convergence for inter-domain routing, only work upon local outages, not the remote ones, *i.e.*, those which occur in other networks. This limited scope is worrisome, as because of the architectural design of the Internet, a packet often traverses many networks before it reaches its destination. A failure in any of these networks can result in traffic loss, even if the sender deploys the latest failure protection mechanisms.

Our measurements, as well as our discussions with network operators, revealed that remote outages do indeed cause traffic loss in the Internet. Yet, this problem remains unsolved because fast rerouting upon remote outages is fundamentally harder than upon local failures, and this for two reasons. First, from sender's perspective, a remote network is not controllable, and cooperation is hard, often just impossible. Second, as a path-vector protocol, BGP does not advertise the status of the links but instead propagates per-prefix information. As a result, upon a remote failure, a router has no other option than to wait for the slow control-plane notifications and updating the forwarding entries for every affected prefix, one by one.

A local fast reroute framework for remote outages

Our main contribution is a framework that enables routers to fast reroute traffic on working and policy-compliant backup paths upon remote outages, without requiring network operators to change the standards, manufacture new devices, or cooperate.

Designing a fast reroute framework for remote outages requires *(i)* detecting the remote outage; *(ii)* assessing the affected traffic; and *(iii)* fast rerouting the affected traffic on backup paths avoiding the failure. Designing such a framework is challenging because there are two conflicting objectives. On the one hand, we need *accurate* heuristics to infer remote outages. On the other hand, the traffic rerouting must be *fast* to prevent packet loss.

To address both of these objectives, we propose new inference algorithms which analyze signals from the control and the data plane to detect and locate the remote outages. Our algorithms are fast and accurate, even with the partial and noisy signals often observed in practical cases. To quickly reroute the data traffic, we then propose a new rerouting scheme tailored for fast inter-domain traffic rerouting. More precisely, instead of rerouting packets based on their destination IP address, we propose to reroute packets based on pre-computed data-plane tags which contain information about the backup next-hop to use.

We always design our solutions with practicality in mind. First, the proposed framework is deployable on existing hardware, and does not require any modification to BGP. To do that, we rely on the new programmable switches and design algorithms that leverage their line-rate speed, scale to thousands of prefixes and are compilable to existing hardware targets. Second, we design algorithms that comply with the BGP policies and are provably safe and beneficial within reasonable assumptions. Those guarantees imply that the traffic is always rerouted on valid BGP paths without causing forwarding issues, regardless of where the framework is deployed, enabling network operators to deploy the framework incrementally.

Outline. We divide this thesis into six parts.

In §1, we first provide the necessary background information.

In §2, we introduce the problem of convergence in the Internet, and show with our own measurement study the negative effects caused by the slow Internet convergence on the end-to-end connectivity. We show that our results align with the feedback we received from the 73 operators participating in our survey.

In §3, we zoom in on the problem of local fast reroute upon remote Internet outages. We show why the existing fast reroute solutions do not work upon remote outages and why the state-of-the-art solutions on inter-domain outages detection, localization, or remediation are not suited for fast traffic rerouting. At the conceptual and practical levels, we then explain the challenges and requirements when designing a fast reroute framework for remote outages.

In §4, we present SWIFT, a fast reroute framework which looks at the control-plane BGP messages to infer and localize remote Internet outages. To achieve fast and safe traffic rerouting, SWIFT reroutes packets based on their pre-computed data-plane tags using the SDN and Openflow capabilities.

In §5, we present Blink, a fast reroute framework which leverages the programmable switches and works entirely in the data-plane. To work at line-rate, Blink relies on new probabilistic algorithms to detect TCP retransmissions, infer remote outages and reroute traffic.

In §6, we finally present Snap, a fast reroute framework which uses a hardware-software codesign to implement the best of SWIFT and Blink. More precisely,

Snap can run the complex SWIFT inference algorithms in a software controller using as input the fast data-plane signals used by Blink and which are carefully collected from the data traffic by the programmable switches. We conclude this section with discussions on possible improvements that this new hardware-software codesign makes possible.

Publications

Most of the work presented in this thesis appears in previously published conference proceedings or workshops. The list of accepted publications is presented hereafter.

- Conference publications

SWIFT: Predictive Fast Reroute.
Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti and Laurent Vanbever, In *Proc. of ACM SIGCOMM 2017*, Los Angeles, USA.

Blink: Fast Connectivity Recovery Entirely in the Data Plane.
Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Stefano Vissicchio, Alberto Dainotti and Laurent Vanbever In *Proc. of USENIX NSDI 2019*, Boston, USA.

- Workshop publications

(Self) Driving Under the Influence: Intoxicating Adversarial Network Inputs.
Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla and Laurent Vanbever In *Proc. of ACM HotNets 2019*, Princeton, USA.

- Posters and Demos

Boosting the BGP convergence in SDXes with SWIFT.
Philipp Mao, Rüdiger Birkner, Thomas Holterbach and Laurent Vanbever In *Proc. of ACM SIGCOMM 2017 (Demo session)*, Los Angeles, USA.

Supercharge me: Boost Router Convergence with SDN.
Michael Alan Chang, Thomas Holterbach, Markus Happe and Laurent Vanbever In *Proc. of ACM SIGCOMM 2015 (Poster session)*, London, UK.

Acknowledgments

I feel incredibly blessed to have been surrounded by many kind and talented people during the journey that led me to complete this dissertation. They made this experience so enriching to me that it will always remain in my mind. I am grateful to everyone who helped me through this process in one way or another. While I can't thank all of them, I would like to dedicate this section to the few that made this dissertation possible.

First and foremost, I would like to thank Prof. Laurent Vanbever for his exceptional guidance throughout the Ph.D. Laurent's knowledge is impressive at many different levels, and his commitment to consistently achieve the various research and teaching missions as well as possible was a great example and source of motivation. Besides, as one of the first members of the Networked Systems Group, I can say without any doubt that Laurent always makes sure that his researchers work in the best conditions and in an enjoyable atmosphere. Despite all his responsibilities as an ETH-Professor, Laurent was always willing to devote time and share his expertise with me over the years. What Laurent taught me is invaluable and will remain incredibly useful in my entire life, at work and outside of work.

I would like to thank Prof. Adrian Perrig and Prof. Kimberly Claffy for participating in the jury of this thesis. Aside from the interesting discussions that we had during the defense and after, they provided me very detailed feedback, which allowed me to improve the quality of the dissertation.

I would like to thank Dr. Stefano Vissicchio and his group for the pleasant and fruitful visit that I was able to do at UCL. Stefano gave me many insightful pieces of advice during my visit. His thoroughness when doing research, as well as his aptitude to tackle actual problems through theoretical abstractions will remain a source of inspiration to me. I also would like to thank Dr. Alberto Dainotti and the entire CAIDA group for the unforgettable internship that I was able to do at CAIDA. Alberto shared his unique expertise on Internet measurement problems with me and his feedback and help were precious.

Turning the often stressful and deadline-driven life of a Ph.D. student into a very enjoyable experience would not have been possible without my amazing colleagues from the Networked Systems Group. I would like to thank Ahmed, Alexander, Albert, Coralie, Edgar, Ege, Maria, Roland M., Roland S., Romain, Rüdiger, Rui, Tibor, and Tobias for their support and collaboration and for all the good times we had eating lunch at the "mensa" or doing entertaining social activities in Switzerland or elsewhere.

I also would like to thank Beat Futterknecht, who was always available to help me deal with all kinds of paperwork from the swiss administration.

I could not join ETH Zurich without the help and support of several great people. I would like to thank Prof. Cristel Pelsser and Randy Bush for the internship I could do at IIJ in 2014. Not only did they prepare me very well to tackle the Ph.D. challenge, but they also helped me to find this Ph.D. position at ETH Zurich that suited me so well. I also would like to thank Dr. Pascal Merindol for connecting me to the research world during my Master's studies at the University of Strasbourg. His excitement when doing research on computer networks motivated me to start a Ph.D.

I would like to thank my friends from Strasbourg, Zurich, and the different places I had the chance to live during my various visits and internships. They made my life more joyous, which helped me to do better research.

I am grateful to all my family for their unconditional love and support. Having such a lovely family, where we always try to help each other, is truly a blessing. I would like to thank my two siblings, Olivier and Lise, for transmitting their enthusiasm for science to me. My thanks also go to my nephew Arthur as well as my cousins, aunties, and uncles for all their encouragement throughout my studies. I also have a thought for my four grandparents, who always took care of me. I hope that they are proud of me from where they are.

Finally, I would like to sincerely thank my parents, who simply put my studies as a top priority in their lives. I will remain forever grateful for that.

Table of Contents

1

Background

Several large-scale transport and communication services have been designed in our history. In 1464, the King of France Louis IX designed the royal postal service and arranged stations or "relays" on the roads where couriers could change horses. In the 19th century, with the emergence of the automobile, highways were built to connect major cities. More recently, optical fiber cables have been installed between the main cities to provide Internet connectivity.

Although these services were designed at very different epochs and for different purposes, they share some principles and often have similar requirements. Besides, their infrastructures are somewhat comparable: it is a network composed of links that interconnect nodes. The links aim at providing a high capacity and being reliable. For instance, in the royal postal service, the couriers could take a new horse ready to gallop at every station to arrive faster to the destination. Highways are often high-speed roads with multiple lanes to prevent congestion. Similarly, the links in the Internet are often full-duplex fiber optical cables that can send more than 100Gbit/s at the speed of light.

Given the large-scale nature of all these services, they all opted for a shared infrastructure where nodes connect other nodes, and the couriers, cars, or Internet traffic move node by node until they finally reach the destination. Be it the road network or the Internet, a node receives traffic and forwards it to the best exit. In the road network, intersections such as interchanges allow cars coming from one road to reach another road quickly. In the Internet, routers forward packets coming from one interface to another interface at line-rate. The biggest interchanges can accommodate hundreds of thousands of vehicles daily, whereas the latest IP routers can forward several terabits per second.

This chapter explains how end-to-end connectivity is made possible in the Internet and shows the key design choices that allow the Internet to support more and more users and applications. We start in §1.1 with the routing basics, which are principles not only used in the Internet but also in many other communication services. We then explain in §1.2 how computer networks perform packets routing and introduce the latest forwarding technologies. Finally, we describe in §1.3 how the Internet has been designed at the logical and architectural levels.

1.1 Routing basics

Routing is the process of selecting the best paths in a network along which data can be forwarded to one or more destinations. Different types of networks, such as the postal service or the public transportation network, rely on routing to select the best paths. Consider the network in Figure 1.1, which is used to forward packets *between* different companies and *within* each company. A route must be determined to forward a packet from a node to another. For instance, a possible route from 5 to 18 is 5 — 11 — 15 — 16 — 18. Network topologies are often redundant to avoid a single point of a failure and enable traffic engineering. In Figure 1.1, there is another route from 5 to 18 via the company B. The best path selection depends on multiple criteria such as the capacity of the links, routing policies, and the current load of the network.

It is possible to perform routing either statically or dynamically. Static routing consists in manually selecting the paths to use within the network. Static routing is easy to design and implement but quickly becomes a hassle for larger networks interconnecting dozens or hundreds of nodes via multiple paths and undergoing frequent topological changes. Dynamic routing consists in automatically selecting the best paths based on the underlying topology and the available resources. It is responsive to network changes: the selected paths are automatically updated when the topology or the available resources change. Dynamic routing is suited for large networks and helps prevent forwarding issues such as dead ends or loops.

1.1.1 Routing algorithms

At its heart, dynamic routing relies on graph algorithms to select the best routes. The network topology is often modeled as a Directed Acyclic Graph (DAG) with a set of N nodes and L links. A weight function $w : L \to \mathbb{R}_{>0}$ can also associate a weight to every link in the graph. The weight of a link typically represents a metric such as the geographical distance or the link's capacity. The Dijkstra [56] and the Bellman-Ford [30] algorithms both aim to find the shortest path from an initial node to every other reachable node in a weighted DAG.

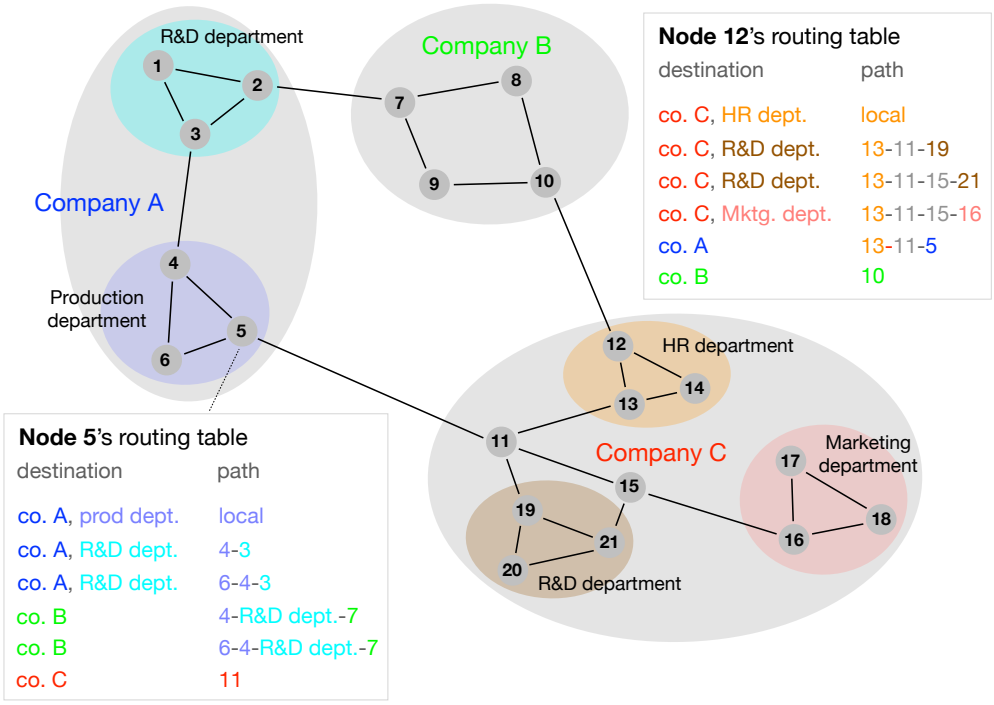


Figure 1.1 Example of a network topology modeled as a graph with nodes and links. The network is divided into regions allowing hierarchical routing, simplifying routing, and packet forwarding.

The Dijkstra algorithm resembles the breadth-first graph traversal algorithm. The main difference is that the order in which the algorithm visits the nodes depends on the link weights. More precisely, consider a graph G with positive weights, an initial node s, and a tree T initialized with s. At a given iteration, the candidate nodes to be visited are not in T but connected to a node in T. Among them, the Dijkstra algorithm selects the node with the minimal total distance from s, attaches it to T on the shortest path, and proceeds to the next iteration until T contains all the nodes in G. When the algorithm finishes, T is the shortest path tree. The shortest path from the initial node to another node in the graph is thus simply the path in T. The complexity of the Dijkstra algorithm resides in the selection of the next node to visit among the candidate nodes at each iteration. When the candidate nodes are stored in a standard list, the algorithm runs in $O(|N|^2)$. However, with a Fibonacci heap instead, the algorithm runs asymptotically in $O(|L| + |N|log|N|)$, the fastest known time complexity for this problem.

The Bellman-Ford algorithm relies on the principle of relaxation. Thus, more precise approximations gradually replace an approximation of the shortest distance until the algorithm reaches the optimal solution. More concretely, the algorithm first sets to infinity the approximations of the shortest distance between the source node s and the other nodes in the graph. The algorithm then iterates over the links and updates the approximations according to their

weight. For instance, the first shortest distance approximations to be updated are for the nodes adjacent to s. Then, the algorithm iterates over the links and refines the shortest distance approximations. The number of iterations required to reach the optimum solution depends on the order in which the links are processed during each iteration. In the worst-case scenario, the algorithm guarantees to return the optimal shortest distances after iterating $|N|-1$ times over all the links, which results in a theoretical complexity of $O(|N||L|)$.

1.1.2 Hierarchical routing

Routing in a large and shared infrastructure often poses two problems. First, even optimized versions of the Dijkstra or the Bellman-Ford algorithms would not scale well on graphs with hundreds of thousands of nodes. Second, routing policies might differ between different regions in the network. For instance, the traffic may follow the path with the least number of hops in a region. In contrast, in another, it may follow the path with the shortest geographical distance.

Hierarchical routing is a principle used in many transport and communication services to solve these problems. With hierarchical routing, nodes are aggregated into regions. The nodes within a region can be further aggregated into subregions. Hierarchical routing simplifies routing, as it is now performed *within* a region or *between* the regions (in which case a single "node" can represent an entire region), but not between all the nodes in the network. To illustrate hierarchical routing with a practical example, consider the swiss postal service. The postal code of an address indicates the destination regions to reach, for every level of aggregation, before it arrives at the post office responsible for delivering the package to the final destination. For instance, in Switzerland, a typical address looks like `Gloriastrasse 35, 8092 Zurich`. The first digit of the postcode indicates that the package must be sent to the "Zurich" district. The second digit indicates that the package must then be sent to the "Zurich city" area. Finally, the last two digits indicate the post office number that can deliver the package to `Gloriastrasse 35`.

With hierarchical routing, nodes can forward packets based on their destination region instead of their actual destination address. Consequently, a node only has to know the paths to reach few regions instead of each distinct destination. Because there are fewer destinations and shorter paths to consider, the best paths calculation (*e.g.*, using Dijkstra or Bellman-Ford) and the packet forwarding are greatly simplified. To showcase hierarchical routing, consider again Figure 1.1. Instead of calculating the best paths among all the existing paths and for every distinct destination in the network, 5 only stores the paths in its routing table (bottom right) to reach some of the regions in the network. More precisely, 5 only needs to know six paths: one for the local destinations (in the production department of company A), two for the destinations in the

R&D department of company A, two for the destinations in company B (which are simplified by aggregating the nodes in the R&D department into one single hop), and one for the destinations in company C. Similarly, 12 knows six paths only (top left), enough to calculate a route for every destination in the network.

Besides reducing the routing complexity, hierarchical routing allows each department to use its own routing strategy, regardless of how other companies and departments perform routing.

1.2 Routing in a computer network

A computer network is a set of connected systems that exchange data with each other. Two types of systems exist within a computer network. *End-systems* are devices such as a computer that generate traffic and send it to other end-systems. *Intermediate* systems are network devices such as a switch or a router that receive traffic and forward it onward towards the recipient. Each end-system has a unique 48-bit physical address called the Media Access Control (MAC) address. MAC addresses allow end-systems to send datagrams to each other directly or via network switches, which are intermediate systems that forward traffic based on the destination MAC address in each datagram.

To continue the analogy with the postal service we started in §1.1.2, a destination MAC address corresponds to the unique identifier of the recipient, *i.e.*, her name. MAC-based forwarding thus has limited scalability and is only used in small networks that are also named Local Area Networks (LAN). As with postal services, computer networks also use the principle of hierarchical routing to interconnect more users.

1.2.1 The Internet Protocol

Large computer networks that interconnect multiple networks use the Internet Protocol (IP) to transmit data in a scalable fashion. IP is a connectionless and unreliable protocol that end-systems use to exchange datagrams or packets. In an IP network, each end-system is assigned an IP address: a 32-bit number in IPv4 or a 128-bit number in IPv6. An end-system transmits data to another end-system by generating packets and labeling them with its IP address (the source IP address) and the IP address of the destination end-system (the destination IP address). End-systems in the same network share the same IP prefix, denoted as `IP/n` where n indicates the prefix length. The use of IP prefixes allows performing a more scalable hierarchical routing using prefix-based routing.

An IP address is thus assigned to an end-system based on the network in which it is directly connected. It can be assigned statically or dynamically using *e.g.*, the Dynamic Host Configuration Protocol (DHCP). When an end-system moves

to another network, it gets another IP address (but keeps its MAC address). Again, it is comparable to the postal service: when someone moves to another location, it keeps her name (MAC address) but changes his address (IP address).

When an end-system sends a packet to another end-system in the same network (be it the final destination or just a gateway to reach another network), it must learn the MAC address of the other end-system. This is possible with the Address Resolution Protocol (ARP), which enables an end-system to translate an IP address into the MAC address of the end-system assigned with that IP address. Network gateways, more commonly known as IP routers, are the intermediate systems that interconnect different networks together and forward packets between them along the best paths.

1.2.2 IP routers

A router has several interfaces from where packets can come in and out. Each interface of an IP router is connected to a different network. Thus a router interconnects different networks together. The goal of an IP router is to forward packets from one network to another such that they can eventually reach their final destination. A router must achieve two missions to accomplish its goal correctly and efficiently: *(i)* it must learn where to route packets according to their destination IP address; and *(ii)* it must forward them as fast as possible. Figure 1.2 depicts the high-level and simplified architecture of a traditional IP router, which includes two planes: the control and the forwarding plane (also known as data plane).

The control plane performs the routing operations and takes the forwarding decisions. For instance, network operators can configure routing protocols and static routes using a Command Line Interface (CLI) in the control plane. The router builds and maintains a Routing Information Base (RIB) in the control plane that stores all the static routes and the routes learned by the routing protocols for the different IP prefixes. The control plane also includes an ARP Table, built manually by the network operator or automatically with ARP.

The forwarding plane is responsible for forwarding packets. While the network functions in the control plane run on general-purpose Central Processing Units (CPUs), the forwarding plane has Application-Specific Integrated Circuits (ASICs) to forward packets at line rate. A router has one or more line cards, each providing multiple network interfaces along with a Forwarding Information Based (FIB) and an ARP Table. The FIB is derived from the RIB and only contains the best routes for each IP prefix. It is implemented with Ternary Content-Addressable Memory (TCAM) to enable fast lookups. When a packet arrives in an input interface, a lookup is performed in the FIB to find the next-hop IP address and determine the output interface. The packet is then switched to the output interface by the switch fabric. Finally, the destination

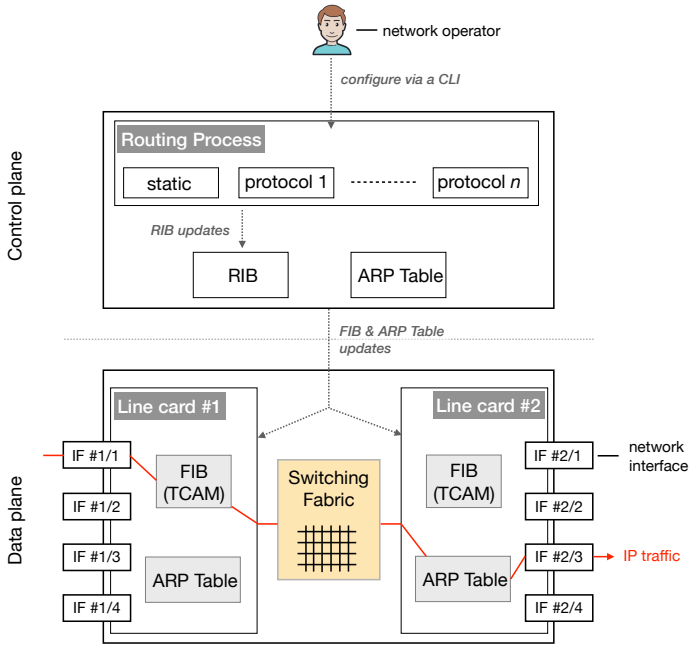


Figure 1.2 Simplified sketch of the architecture of an IP router. Traditional IP routers are divided into two planes. The control plan is in charge of the routing, whereas the data plane is in charge of forwarding the packets at line rate.

MAC address is derived from the ARP Table, and the output interface sends out the packet.

We only depict in Figure 1.2 the forwarding components of an IP router. However, the forwarding plane of a router is also capable of buffering, scheduling, and filtering packets according to rules and policies configured by the network operator in the control plane. Observe that as the memory used by the routers to buffer packets is finite, a router may drop packets if the rate at which they arrive exceeds the rate at which the router forwards them. Yet, modern routers are capable of forwarding terabits per second.

1.2.3 Distributed routing

In traditional IP routers, the control and forwarding plane are coupled in the same device. As a result, today's IP networks use distributed routing protocols for dynamic routing. A routing protocol specifies how routers communicate and which information they exchange for learning and agreeing on the routes used to reach the different destinations in the network. In practice, there are many distributed protocols, and traditional IP routers implement many of them. We observe two classes of routing protocols. With link-state routing protocols, topological information is exchanged between the routers such that each router can construct a map of the network and run the Dijkstra algorithm on it to

find the best paths. With vector-based protocols, routers exchange distance or path vectors and rely on the principle of relaxation as used in the Bellman-Ford algorithm to derive the best baths.

Routing protocols automatically allow the computing of new routes when the network topology changes. When a router is added, shut down, or fails, the neighboring routers detect the change, and all the routers in the network will automatically update their routing table. As long as the graph remains connected, the routers will always find a route to reach every destination. Unfortunately, when the network topology changes, it often takes time for the routers to exchange topological information or vectors and agree on the new routes to use. This time is known as the convergence time, which is a critical time during which routers may not update their routing and forwarding table at the same speed, resulting in inconsistent network-wide routing decisions. Inconsistent routing decisions result in routers dropping packets (blackhole) or, worse, routing loops. We elaborate on the convergence time in §2.

1.2.4 Software-defined Networks

Recently, new networking technologies have emerged with the goal to make networks more programmable and flexible. Software-defined Networks (SDN) [104] is the first attempt to make the control plane more programmable. SDN relies on new switches, which *(i)* enable users to bypass built-in control plane functions and introduce their own, and *(ii)* provide a set of more advanced – but fixed – forwarding functions leveraging a pipeline of match-action tables of arbitrary width and depth. In SDN, the data plane and the control plane are decoupled, and forwarding rules can be installed remotely through an Application Programming Interface (API) such as OpenFlow [125]. Besides the control plane being programmable, SDN also enables users to implement network-wide control plane functions in a centralized controller.

The centralized routing paradigm is both conceptually and operationally simpler than the traditional distributed control plane. The central controller knows the entire topology, computes routing decisions, and pushes the forwarding rules into the switches with *e.g.*, OpenFlow. SDNs are more manageable for network operators and mitigate some of the inherent problems of distributed routing, such as the inconsistent forwarding states during network convergence. The concept of SDN is used in many existing networks, for example, at Google [92]. However, one of the main drawbacks of centralized routing is that it is not adapted for routing between networks of different entities. For instance in Figure 1.1, centralized routing can be used within each company, but must be performed in a distributed fashion between them.

1.2.5 Programmable data planes

Traditional network hardware cannot be programmed as a general-purpose CPU because they rely on custom ASICs (see Section 1.2.2), which are tailored for fast packet processing. Yet, the very recent advances in switching ASIC design made it possible to program the data plane while keeping its line-rate packet processing [103, 35]. An example that illustrates this progress is the Reconfigurable Match Tables (RMT) model [35], which allows match-action tables to be defined using arbitrary header fields and provides a larger collection of packet processing actions. The Intel Tofino switch and its programmable Ethernet switch ASIC [5] implements the Protocol Independent Switch Architecture (PISA) architecture (a generalization of the MRT model) and allows for data plane programmability with maximum port bandwidth of 6.4 Tbit/s.

The PISA architecture consists of a programmable parser, a programmable match-action pipeline (which, besides the match-action tables, enables arithmetic operations and provide stateful objects), a traffic manager (*e.g.*, for packet scheduling) and finally a programmable deparser. The way the PISA architecture processes packets depends on *(i)* how the operator has programmed it *i.e.*, how she defines the match-action tables, performs arithmetic operations, or uses stateful objects *(ii)* the content of the match-action tables, which can be modified at runtime, and *(iii)* the state in which are the stateful objects.

Domain-specific ASICs are programmed with domain-specific programming languages. For instance, FPGAs are programmed with *e.g.*, VHDL, whereas GPUs are programmed with *e.g.*, OpenCL. In the case of PISA, the most widely used domain-specific programming language is $P4_{16}$ [34]. $P4_{16}$ is an open-sourced and high-level language that is protocol and target-independent. Alongside $P4_{16}$, the community has developed P4Runtime [135], an interface for controlling the data plane elements at runtime. We note that there exist other data plane programming languages, such as NPL [131], which is used to program the Broadcom Trident 4 [38], another programmable switch offering a different data-plane model than PISA. Although very recent, researchers and operators have already designed many $P4_{16}$ programs to improve computer networks' performance, resiliency, and security [82].

1.3 Internet routing

The Internet is a computer network where routing is anything but easy, and this for at least three reasons.

- The Internet is *large*. As of December 2020, we count more than 5 billion Internet users [90] and tens of billions of connected devices [91]. The Internet

is also large by the volume of traffic exchanged. Cisco estimated the global Internet traffic to be 1.1 Zettabytes per year [48]. To put this traffic volume in perspective, if each Gigabyte in a Zettabyte was a brick, we could build 258 Great Walls of China.

- The Internet is *heterogeneous*. It is heterogeneous at different levels. First, the Internet comprises many different network devices, which can run many different network protocols. These devices and protocols often evolve over time. Second, the Internet is composed of many networks, and a single autonomous entity administrates each. These networks have different goals. Some are data-center networks; some others are transit or edge networks. Finally, there is also various sort of communication over the Internet. For instance, Internet users stream videos, trade cryptocurrencies, or play video games. These different services require different Quality of Service (QoS).

- The Internet is a *business*. Among the most prominent companies, many are IT companies providing Internet services such as Amazon, Google, or Facebook. Where Internet traffic flows, money flows. Thus, network operators often have to deal with stringent Service Level Agreements (SLAs), as the cost of one minute of downtime for those companies can easily reach a 6-digit number [170].

1.3.1 Internet architecture

The Internet comprises Autonomous Systems (ASes), which are networks administrated by a single entity. Each AS has its own IP prefixes and is responsible for the routing within its network, also called intra-domain routing (see §1.3.2). ASes are interconnected to other ASes via inter-AS links and advertise their IP prefixes to be reachable from outside. This architecture is comparable to the one shown in Figure 1.1, where the companies are ASes, and the link between *e.g.*, 2 and 7 is an inter-AS link. As of February 2020, there are around 70,000 ASes in the Internet advertising a total of 875,000 prefixes [1].

ASes composing the Internet have different purposes. Some are Internet Service Providers (ISPs) and aim at providing Internet connectivity to their customers (which can be end-users or other ASes). Some others are Content Delivery Networks (CDNs) and aim to provide high availability and performance by distributing the content spatially relative to end-users. Finally, some others are Internet Exchange Points (IXPs), which are physical infrastructures where ISPs or CDNs can interconnect directly, rather than through third-party ASes.

At the AS level, we thus observe complex relationships. An AS maintains connections with other ASes, with which it has business and Service-Level Agreements (SLAs) that align with the type of relationship (customer, provider or peer). Network operators use the Border Gateway Protocol (BGP) to ensure that the traffic exchanged between the ASes follows the agreements. In a sense, BGP is like the "glue" of the Internet. It allows border routers, *i.e.*, the

ones directly connected to other ASes, to exchange routing and reachability information to the neighboring ASes and allows operators to express complex routing policies (see §1.3.3).

Over time the Internet architecture has evolved to adapt and provide better performance. However, it *still* relies on BGP and its routing principles. Although the community sometimes advocates for a new architecture [166] and proposes alternatives [28], changing the fundamental principles and protocols of the Internet is complicated and often takes time. A good illustration is IPv6, which started to be deployed in the mid-2000s and is only used by a relatively small fraction of the traffic several years after [89]. We can thus expect the current Internet architecture to remain for several years, even decades.

1.3.2 Intra-domain routing

Interior Gateway Protocols (IGPs) are the routing protocols used to calculate routes *within* an AS. We observe two types of IGPs: distance-vector and link-state protocols. Distance-vector protocols implement a distributed variant of the Bellman-Ford algorithm. More precisely, each router in the network maintains a vector containing the distance to every destination and periodically sends it to its neighboring routers. Upon reception of a distance-vector, a router increments all its values by the cost configured on the interface from where the vector was received, updates its routing table if it finds a path with a shortest distance, and transmits the updated distance vector to inform the other routers about the new distances.

The main drawback of distance-vector protocols is their convergence time. The routers take time to exchange and update distance vectors until they converge to the optimal solution, transiently hindering connectivity for some destinations in the network. As a result, distance vector protocols are only adapted for small or medium IP networks. Distance vector protocols include the Routing Information Protocol (RIP), Cisco's Internet Gateway Routing Protocol (IGRP), and others. For larger IP networks, link-state protocols are preferred.

With link-state protocols, each router advertises the state of its links to all the other routers in the network. From the link-state advertisements that it receives, a router then derives a connectivity map in the form of a graph, and on which it applies the Dijkstra algorithm to find the shortest paths to every other router in the network. Although every topological change requires the routers to recompute the shortest paths on the new topology, link-state protocols converge faster than distance-vector protocols. The Open Shortest Path First (OSPF) and Intermediate System to Intermediate System (IS-IS) algorithms are the link-state protocols often used in practice to configure intra-domain routing.

1.3.3 Inter-domain routing

Distance-vector and link-state routing protocols are not suited for inter-domain routing for two main reasons. First, because the AS-level topology includes around 70,000 ASes, far more than the number of routers in an AS. On such a large topology, those routing protocols would be impractical: distance-vector protocols would have issues to converge and link-state protocols would require a lot of resources in each router to compute the new paths upon every topological change. Second, ASes are individual entities that want control over the route selection process and may implement complex routing policies, which are not only based on the distance but also various other attributes.

BGP is the protocol used for inter-domain routing. It is a path-vector protocol and is policy-based. Path-vector protocols rely on the principles used by the Bellman-Ford algorithm as BGP routers advertise routes to their neighboring BGP routers. However, a BGP route contains not only the destination IP prefix but also the AS path (*i.e.*, the list of ASes that the route has traversed), a next-hop IP address, and a set of attributes that network operators can use to configure routing policies. We distinguish two types of routing information exchanged between BGP routers: advertisement and withdrawal. A BGP router advertises a route either when it learns a new best route for a destination or when a policy decision leads the router to prefer a new route. A router withdraws a route when the destination is no longer reachable. Observe that a router can implicitly withdraw a route for a destination by advertising another route for that destination, which takes precedence over the previously advertised route.

Figure 1.3 gives an overview of how BGP works in a traditional router and illustrates how network operators can use it to configure routing policies. In a nutshell, a BGP router receives the locally originated or redistributed routes as well as the routes advertised by its neighboring BGP routers with which it maintains BGP sessions. The BGP router then independently selects one best route for each IP prefix following the BGP Decision Process, updates its RIB accordingly and finally advertises the best routes to its neighboring BGP routers with their updated AS path. The BGP Decision Process is the algorithm used to select the best route for a prefix across all the received routes for that prefix and as a function of the attributes attached to each route.

While the organization of the BGP process and the algorithm used in the BGP Decision Process is fixed in a BGP router, the attributes attached to the routes and the input and output filters allow network operators to change the outcome of the process. More precisely, by manipulating the route attributes and defining route filters in the Input and Output Policy Engine, network operators can tune the route preference to modify the outcome of the BGP Decision Process, or decide to filter a route out and not advertise it to another BGP neighbor. Besides, some BGP route attributes also propagate network-wide, enabling network operators to implement network-wide routing policies. Putting

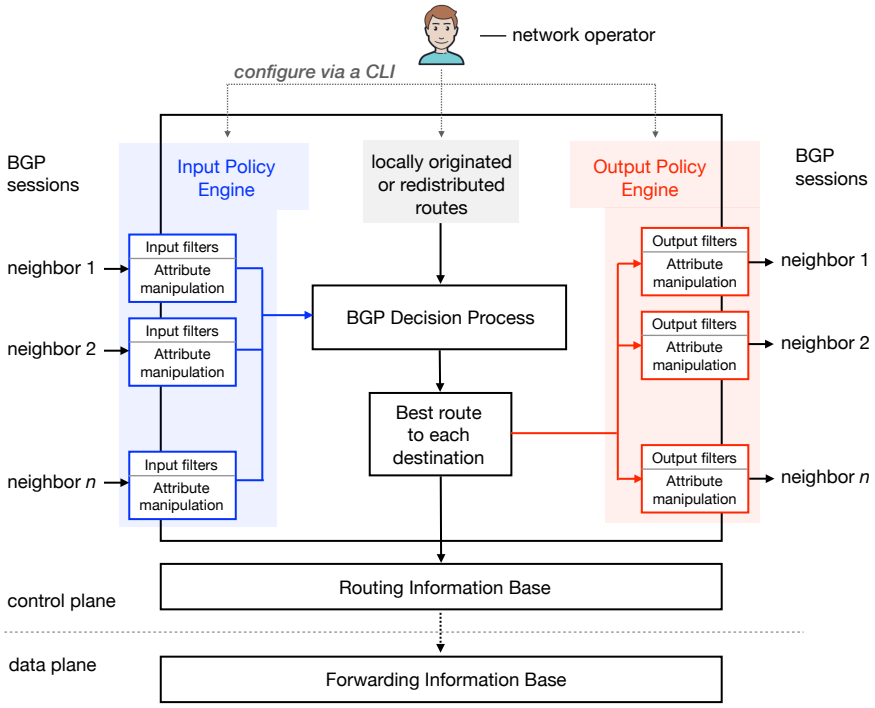


Figure 1.3 Overview of the BGP process in an IP router.

everything together, BGP enables operators to enforce business agreements (*e.g.*, the typical customer/provider business relationships [71]) and perform traffic engineering (*e.g.*, hot-potato routing).

In practice, we observe two types of BGP sessions. First, two border routers belonging to different ASes can establish an External BGP session (eBGP), allowing them to exchange BGP routes *between* ASes. Second, routers in the same AS can establish Internal BGP (iBGP) sessions, allowing them to exchange BGP routes between routers located *within* the same AS. While two eBGP peers are generally in the same LAN and thus directly reachable, two iBGP peers can be in different LANs (within the same AS) and thus rely on the IGP to communicate and send the traffic for external destinations towards the egress router in the AS.

A BGP router exhibits a slightly different behavior with eBGP and iBGP sessions. In particular, a router does not append its AS number to the AS path when it advertises a route over an iBGP session. Also, routes learned from iBGP neighbors are not propagated to other iBGP neighbors to prevent routing loops. Consequently, BGP routers within an AS must establish fully meshed iBGP sessions to redistribute external routes to all other routers within that AS. An alternative to iBGP full mesh is BGP Route Reflection [44]. We refer the reader to [85] to get hands-on experience on how to configure BGP in practice.

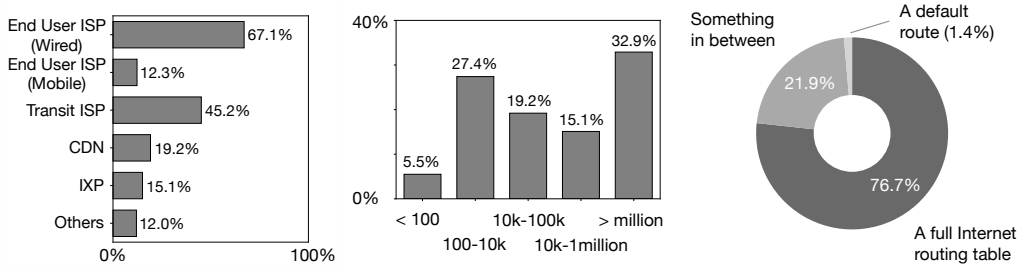
2

Internet (slow) convergence

Convergence in an IP network denotes the transition from a routing and forwarding state to another. A configuration change or a topology change typically triggers convergence. We define convergence *time* as the time between the change and the moment when the network has converged, *i.e.*, all the routers forward the traffic on the new best paths. The convergence process involves the following actions: *(i)* the routers on which a routing configuration change has been applied, or the ones adjacent to the topological change, detect the change; *(ii)* upon detection of the change, the routing protocol converges to a final and stable state, and the routers recompute their Routing Information Base (RIB); and *(iii)* the routers update their Forwarding Information Base (FIB) and send the traffic on the new best paths.

Except for the change detection, which is done locally, convergence is a network-wide and asynchronous process. Indeed, the distributed nature of routing protocols results in routers recomputing their RIB and updating their FIB at different times and speeds. Hence, during the convergence time, routers may have inconsistent states which cause issues such as a delayed convergence, routing loops, forwarding loops, or dead ends. These issues result in packet loss, which is detrimental for end-users, as many applications require continuous Internet connectivity, and for Internet service providers, as even the slightest downtime can cause large financial loss. For example, the cost of one minute of downtime for Amazon or Google easily reaches a 6-digit number [170] and almost any outage that they experience makes the news (*e.g.*, [123]). Smaller Internet players are not better off. Across the networking industry, the average cost of downtime is estimated to be about $8,000 per minute [54].

A 72-operators survey on the Internet convergence. To substantiate the effect of convergence issues on operational practices, we performed an anonymous survey among 72 operators. Figure 2.1 gives information about our respondents. Our respondents come from a wide variety of networks providing



(a) The type of network that our respondents operate (multiple-choice question)

(b) The number of Internet users that our respondents reach or connect

(c) The number of BGP routes the routers of our respondents maintain in their FIB

Figure 2.1 General information about the 72 operators participating in our survey

one or more services to a large customer base. The majority of the respondents (67%) provides Internet connectivity to wired end-users. 45% of them provide transit services. 19% of them work for Content Distribution Networks (CDN), while 15% of them work for an Internet Exchange Points (IXP). In terms of customer base, 33% of our respondents connect 1 million or more users to the Internet, 48% connect 100,000 users or more, while 67% connect at least 10k users. 77% of our respondents work with full Internet routing tables, meaning that their routers carry more than 865k prefixes in their forwarding tables [1].

Operators care about slow convergence. Among our respondents, 78% care about slow BGP convergence. The remaining 22% do not care because they: receive a single default route from their provider (3 of them); do not have stringent Service Level Agreements to meet (6 of them); or because they have never experienced a slow convergence before (4 of them). Because it is a problem for network operators, Internet convergence is an important research topic since the 90s [75, 52]. Researchers devoted much effort to measuring the convergence time, highlighting its effect on users' connectivity, and proposing mechanisms to make it more efficient. Among our respondents, 77% use at least one of these mechanisms.

In the following sections, we describe more precisely each action that is part of the convergence time and introduce the mechanisms proposed by the community to speed up convergence and mitigate the typical issues that arise during the convergence time. In §2.1, we focus on the topological change detection. In §2.2, we then focus on the Internet *routing* convergence. Finally, in §2.3, we focus on the FIB update.

2.1 Topological change detection

Two types of events can result in a topological change: the *notified* and *silent* changes. The *notified* changes can be manual reconfigurations (*e.g.*, via the router's CLI) that happen during planned maintenance operations. For instance, an operator can shutdown some interfaces or reconfigure some routing protocols (*e.g.*, change the OSPF weights) on a router before shutting it down for a software or hardware upgrade. The *silent* changes can be a link or a router failure or happen when a cable is plugged into a network interface card. Unlike *notified* changes, *silent* changes are not immediately detected and require detection techniques. In the rest of this section, we will focus on the *silent* changes, and more precisely, on changes caused by failures because these events likely result in packet loss during the detection time.

2.1.1 Basic failure detection

The first way to detect a link or a node failure is to look at the status of the interfaces. For example, today's network interface cards detect whether there is a signal on the wire or not. When the signal disappears, the router detects the failure. Looking at the signal on the wire works well when two routers are directly connected, without devices in-between. In practice, though, we often see network devices (*e.g.*, switches) between two logically connected routers rendering the failure detection based on the signal seen on the wire often inefficient. As a result, routing protocols include mechanisms to check whether two logically connected routers can still reach each other. For instance, OSPF relies on *Hello* packets that are sent at every *Hello Interval*, which is set to 10 seconds by default [129]. An OSPF router declares a neighbor dead when it does not receive a *Hello* packet within the *Router Dead Interval*, which is set to four times the *Hello Interval*, *i.e.*, 40 seconds. Similarly, BGP uses *Keepalive* messages, that a router sends every 60 seconds to its BGP neighbors, by default [11]. A router declares a BGP neighbor as down when it does not receive a *Keepalive* message during the *Hold time*, which is set to 180 seconds by default.

2.1.2 Fast failure detection

Upon a failure and with the default timers, it takes several seconds for a protocol to detect dead neighbors. During this time, routers may send traffic to the failed link or node, resulting in the traffic being lost without any notification. Fortunately, network operators can lower these timers to speed up the failure detection time. However, network operators need to be cautious when tuning these timers, as lower timers trigger more signalization traffic and increase the chances of peer flapping.

An alternative for fast failure detection is Bidirectional Forwarding Detection (BFD) [98], which aims to provide a sub-second failure detection. BFD establishes point-to-point sessions and relies on simple Hello and Echo packets. Yet, unlike the probing mechanisms used by the routing protocols, BFD runs directly on the line cards and thus avoids over-taxing the CPU of the router. Besides, multiple protocols (*e.g.*, OSPF and BGP) can simultaneously use a single point-to-point BFD session. Routing protocols need to register and use the BFD session as a service to be alerted upon a link or node failure.

Most of our 73 respondents do use fast failure detection mechanisms. Among the 73 network operators who answered our survey, 27 (37%) use non-default BGP timers, and 41 (56%) use a fast failure detection mechanism such as BFD.

2.1.3 Gray failure detection

A fundamental limitation with BFD is that it only detects hard failures that affect all the traffic. In practice, networks are also subject to "gray" failures, which affect only a fraction of the traffic and are typically caused by hardware bugs [87]. Gray failures are challenging to detect because both the basic *Hello* mechanism used by the routing protocols and the packet sampling techniques used for traffic monitoring are not fine-grained enough to detect gray failure [50, 137]. Recently, researchers designed several techniques levering data-plane programmability to accurately pinpoint gray failures in ISP environments. For instance, LossRadar allows capturing individual packet lost in the entire network and at fine time scale [116]. LossRadar places meters in programmable switches, which count the number of packets traversing each interface. The switches then regularly generate small digests with the content of the meters and send them to a controller, which decodes the digests to detect and locate individual packet lost. Similarly, NetSeer works entirely in the data plane and aims to detect and report data-plane events such as packet loss [181]. To do that, NetSeer implements a negative-acknowledgment-based protocol between the different switches.

2.2 The Internet *routing* convergence

Upon detection of a change, routers start to exchange information about the new topology or the new paths to use in a distributed fashion. In the Internet, end-to-end connectivity involves several routing protocols at different levels. Overall, the Internet routing convergence is a complex process during which forwarding issues can arise, such as forwarding loops or black holes.

We now show the typical routing convergence problems as well as the existing solutions to mitigate them. We naturally decompose the Internet routing convergence in two sub-problems: the routing convergence within an AS (§2.2.1) and between ASes (§2.2.2).

2.2.1 Convergence within an AS

This section mainly focuses on link-state protocols as they are often preferred in current ISP networks. However, we note that vector-based protocols exhibit similar routing inconsistencies during convergence.

2.2.1.1 Typical problems

A typical OSPF or IS-IS convergence includes the following phases: *(i)* a router detects the link-state change; *(ii)* the router signals the change to other routers by flooding link-state packets within the network; *(iii)* routers recompute the shortest-path tree; *(iv)* routers then recalculate their RIB; and *(v)* routers finally update their FIB. Putting everything together, the IGP convergence typically lasts for hundreds of milliseconds [69], a time during which routers may have inconsistent states resulting in packet loss.

Consider, for instance, the example in Figure 2.2, which depicts the forwarding state of four routers before and after the failure of the link 1 — 4 and for the traffic destined to an end-system connected to 1. During convergence, 4 may update its FIB before 3, in which case traffic will loop between 3 and 4 until both routers have updated their FIB (Figure 2.2b). Such transient loops have been observed and measured in existing IP networks [83, 127].

2.2.1.2 Solutions to improve the routing convergence

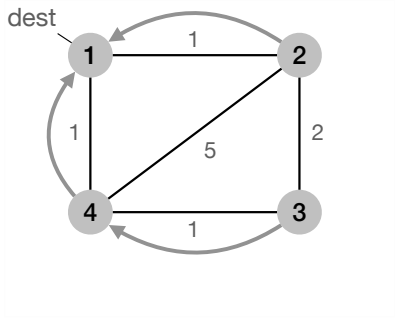
We divide the solutions into two categories: those tailored for *non-urgent* changes (*e.g.*, a change due to maintenance) and those tailored for *urgent* changes (*e.g.*, a failure).

For *non-urgent* changes: order the FIB updates. A solution to enable loop-free convergence upon a *non-urgent* change is to order the FIB updates between the different routers. For instance, in the scenario depicted in Figure 2.2, updating the FIB of 3 before the FIB of 4 prevents the forwarding loop between 3 and 4 (visible in Figure 2.2b). Ordering the FIB updates is possible using timers, dedicated control-plane messages, or by incrementally increasing the cost of the link to shut down until it can be safely removed [68, 49]. For instance, increasing the cost of the link 3 — 4 to 3 forces 3 to update its FIB and use the path 3 — 2 — 1 to reach the destination while 4 still uses its direct connection with 1. When 3 has updated its FIB, the link can be removed without creating any transient loop.

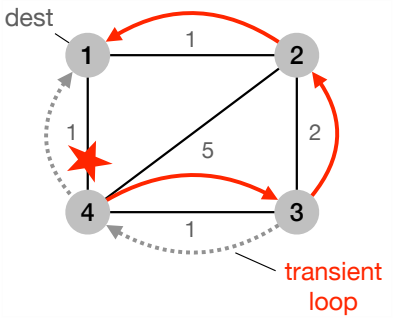
Ordering the FIB updates is also possible with vector-based protocols. For instance, the Enhanced Interior Gateway Routing Protocol (EIGRP) [151] relies on the DUAL algorithm to order the route updates between the routers [72]. The DUAL algorithm uses a diffusion algorithm that recursively allows a router to update its route only when its new next-hop has already updated its route. In the context of iBGP, the LOUP protocol disseminates route updates (or withdrawals) learned via eBGP in an order which provably never causes transient forwarding loops [81].

For *urgent* changes: fast-reroute traffic to loop-free alternates or tunnel the traffic. Ordering the FIB updates takes time because the network has to converge between each update. This strategy thus works for *non-urgent* changes but is not adapted for *urgent* changes, which may immediately result in packet loss. For *urgent* changes, a solution is to fast reroute the traffic towards a pre-computed backup path. Fast reroute techniques, however, break the congruence between the data and the control plane. When a router R fast reroutes the traffic to a backup path for a destination d, the new next-hop N is unaware of this path deflection. A forwarding loop is thus created if N forwards traffic destined to d on a path that includes R. For instance, in Figure 2.2, if 4 fast reroutes the traffic to 3 (the shortest path according to the link costs), a forwarding loop is created until 3 updates its FIB.

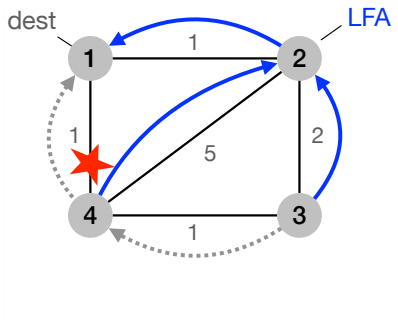
IP Fast Reroute (IPFRR) is a fast reroute framework for link-state IGP that prevents forwarding loops [158]. With IPFRR, each router knows the topology and thus can calculate Loop-Free Alternates (LFA) in anticipation of each failure of an adjacent link and can make them available for invocation with minimal delay [27]. LFAs are backup next-hops guaranteeing that when a failure of a particular link occurs, forwarding traffic through them will not result in a loop. Figure 2.2c illustrates the behavior of 4 when it uses IPFRR. Instead of using the shortest path, 4 fast reroutes the traffic to 2, immediately restoring connectivity. When all the routers have converged, 4 can safely fall back to the best backup path, via 3 (Figure 2.2d). Distance-vector protocols also use the concept of loop-free alternates. EIGRP, for instance, implements Loop-Free Alternate Fast Reroute.



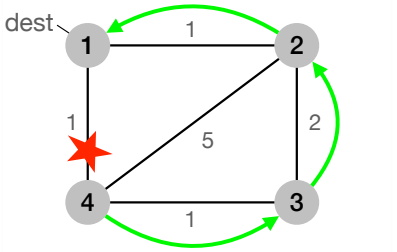
(a) Before failure.



(b) During convergence and without LFA.



(c) During convergence and with a LFA.



(d) After convergence

Figure 2.2 An illustration of routing inconsistencies during convergence when using a link-state IGP. Figure 2.2b (resp. 2.2c) shows the forwarding state when fast rerouting the traffic without (resp. with) an LFA. Each link is labeled with its cost. The arrows indicate the path used by the traffic to reach the destination.

An alternative to IPFRR is MPLS Fast Reroute, which creates backup tunnels using MPLS labels and activates them upon detection of the change [136]. Tunneling the traffic prevents transient loops but requires an additional control-plane mechanism to pre-compute the tunnels as well as data-plane tags (MPLS labels) that must be installed in the forwarding table of the routers. An AS can also tunnel the transit traffic to avoid problems during IGP or BGP convergence [33].

2.2.2 Convergence between ASes

Researchers measured that the BGP convergence time, defined as the time difference between the injection of a fault in an AS and the routing table of other ASes reaching a steady state, can last tens of seconds, even minutes [109, 108, 120]. In this section, we first explain in §2.2.2.1 what are the fundamental issues that make BGP converge so slowly. We then show in §2.2.2.2 the attempts to mitigate the issues caused by the slow BGP convergence.

2.2.2.1 The reasons why BGP converges so slowly

BGP can exhibit routing instabilities during convergence. While some routing protocols, such as OSPF, are proven to converge upon a topological or a configuration change, it is not the case with BGP, where conflicting policies may result in routing instabilities and persistent route oscillations [169, 110]. Whether the network finally converges after a period of routing instabilities depends on the order in which the routers exchange the BGP messages. Fortunately, in the early 2000s, Gao and Rexford have defined a set of guidelines that operators can follow when defining the routing policies to impose a partial order between the routes to each destination and ensure route convergence upon topology or policy changes [71]. These guidelines do not require coordination between ASes and follow the conventional traffic-engineering practices of ISPs. They are thus widely adopted by the community.

BGP is subject to path exploration. Path exploration is a phenomenon inherent to vector-based protocols. It occurs during routing convergence and consists of some routers trying several transient paths before selecting a new best path or declaring unreachability for a prefix [133]. In the worst-case scenario and with a complete graph, Labovitz et al. demonstrated that upon a failure, the number of route updates processed by a router for a destination is approximately $(n-1)!$, with n the total number of routers [109].

BGP works on a per-prefix basis. With BGP, routing information propagates on a per-prefix basis, allowing network operators to apply distinct routing policies and thus use different paths on a per-prefix basis. Besides, routing messages cannot specify network resources that failed because AS topologies and policies are hidden at the inter-domain level (mainly for scalability and AS-level privacy).

BGP design choices. Because routing oscillations add load on the routers and delay the convergence, BGP route flap damping techniques have been proposed and are deployed in actual routers to mitigate routing instabilities. These techniques consist in limiting the frequency of the route advertisements *e.g.*, using fixed timers associated with the Minimum Route Advertisement Interval (MRAI) [171, 11]. While beneficial to limit routing instabilities, these timers exacerbate the Internet routing convergence [121, 76]. Other design choices delay the routing convergence, such as the rate-limiting mechanism for BGP table transfer [31] or the TCP stack implementation [40].

2.2.2.2 Attempts to speed up the BGP convergence

Generally speaking, improving the BGP convergence is more complicated than with IGP because *(i)* unlike with IGP, there is no central entity: each AS performs routing on its own; *(ii)* BGP is a policy-based protocol and routing must always comply with the defined policies; and *(iii)* there are many more

destinations and routers in the Internet than within an AS. We classify the attempts to improve the Internet routing convergence in the following five categories.

Shrinking the BGP convergence time. The first option to shrink the BGP convergence time is to limit path exploration. Similarly to EIGRP, this can be achieved with a diffusion algorithm such as DUAL [72]. Routers can also leverage the BGP attributes (*e.g.*, the AS path) to check the consistency of the received routes (using the previous route updates received from the neighboring routers) and reject the ones assessed as infeasible [140]. Afek et al. propose to inject withdrawals for the prefixes for which transient and inconsistent route updates have been observed [20].

Reducing the number of messages disseminated during convergence is another solution to shrink the BGP convergence time. It can be achieved with BGP update packing [146], which consists of aggregating route updates sharing the same attributes into the same BGP update, or with route aggregation [161], which consists of filtering out routes for more specific prefixes while respecting the routing policies for data packets forwarding.

Finally, the BGP convergence can be improved by adding attributes in the BGP messages to indicate the source/origin of the route change event [96, 139], or simply by tuning its parameters [141].

Rerouting traffic to failover paths. Bonaventure et al. propose an extension to BGP that allows each router to automatically pre-compute an alternative next-hop for each of its BGP peering link and for each prefix [33]. Upon the failure of one of its peering link, a router can fast reroute the traffic through IP tunnels to the pre-computed alternative next-hops. Instead of using IP tunnels to reroute traffic to an alternative next-hop, Lakshminarayanan et al. propose to add in the header of the data packets the list of the encountered failed links, so that routers can use this information along with the network map (*e.g.*, learnt with the IGP) to reroute traffic to a working BGP next-hop [111].

R-BGP [107] allows the routers to advertise not only the best path to their neighbors, but also a failover path (ideally, with the most disjoint AS path compared to the primary path). Upon a BGP peering link failure, the routers can quickly reroute traffic on the failover paths learnt with R-BGP.

Using overlay networks. An overlay network runs on top of another network. For instance, Resilient Overlay Networks (RON) [25] include nodes deployed at various locations in the Internet and which cooperatively, using a custom routing protocol, route packets between each other. Overlay networks allow bypassing the routing decisions calculated by the underlying routing process and thus allow bypassing the slow Internet convergence. More precisely, each RON node monitors the quality of its virtual links using active probing and passive observations. It then reroutes traffic on another virtual path upon detection

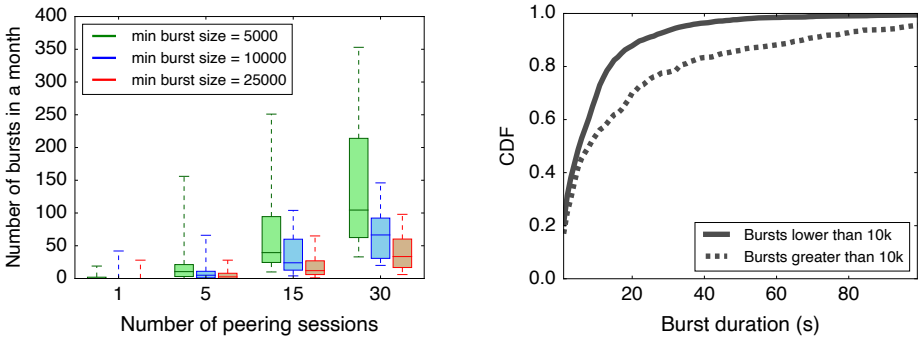
of a connectivity disruption. In the same line of thinking, Gummadi et al. [78] show that a simple overlay routing technique consisting of the source node attempting to recover from path failures by routing indirectly through a small set of randomly chosen intermediaries is beneficial and requires low overhead (no background path monitoring and custom routing protocol).

Finally, ARROW [142] also relies on an overlay network. With ARROW, ISPs can provide remote transit for a fee. Remote transit is achieved by provisioning tunnels, using a custom protocol, between end-points and the ISP providing the service. ARROWS tunnels include active probing mechanisms to detect and locate failures on the ARROW circuits quickly. An end-point can simultaneously establish multiple ARROW paths to the destination and use them in conjunction with the original path for fast failover. The authors show that an ARROW deployment can recover connectivity within a few hundred milliseconds by rerouting traffic to a backup tunnel upon an Internet outage.

Using new protocols and architectures. Consensus routing prevents the typical convergence issues by augmenting BGP with Paxos agreement to ensure route consistency, *i.e.*, a router forwards packets on a path only when the upstream routers have adopted the route [97]. Besides, consensus routing allows routers to be in a transient mode during which they can use backup routes (*e.g.*, learned with R-BGP).

Subramanian et al. propose to replace BGP with a Hybrid Link-state Path-vector protocol called HLP [163]. HLP relies on an additional level of hierarchy to divide the routing process and make it simpler. More precisely, multiple ASes can be grouped in one isolated region. Link-state routing is used within a region, whereas path-vector routing is used between the different regions. SCION shares similar principles: ASes are divided into independent trusted domains, and only a few trusted core ASes within each domain are connected to other core ASes in other domains [28]. SCION comes with new control-plane mechanisms for routing, which leverage the trusted domains to improve scalability, path selection (both on the sender and the receiver side) and make routing more secure.

Offloading routing tasks. Routing tasks can be offloaded to the data plane so that routers can converge without waiting for the slow control-plane notifications. For instance, DDC allows automatically finding an alternative working path upon a link failure using data-plane mechanisms only [117]. More precisely, upon detection of a failure, a router sends back a packet on the incoming link. When the downstream router receives this packet, it deactivates the failed path and forwards the packet to an alternative path instead. Costa Molero et al. show that it is possible to implement some of the routing tasks in the data plane using programmable switches [128]. They implemented a simple version of BGP in $P4_{16}$ and show that it speeds up inter-domain convergence. Routing can also be offloaded to the cloud, where more resources and less propagation delay help speeding up the routing convergence [143].



(a) Bursts of withdrawals are frequent and involve a high number of prefixes, including popular ones.

(b) Bursts take a long time to propagate; longer bursts even more.

Figure 2.3 Number, size and duration of bursts captured from 213 BGP vantage points in November 2016.

2.2.3 The Internet *still* takes minutes to converge

Unfortunately, despite network operators deploying some of the solutions listed above, the Internet still converges slowly. Indeed, the widely deployed solutions (*e.g.*, route flap damping [51]) only partially solve the convergence issues [121]. The other solutions require modifying the current standards (at the software or hardware level) or cooperation between ASes, making a network-wide deployment hard. For instance, SCION is only partially deployed in the Swisscom network [164]. In this section, we thus report evidence of the *still* slow Internet convergence with a measurement study.

We counted and measured the duration of bursts of BGP withdrawals extracted from 213 RouteViews [132] and RIPE RIS [16] peering sessions during November 2016. We look at bursts of BGP withdrawals because they typically coincide with path failures [61, 176]. We extracted the bursts using a 10 s sliding window: a burst starts (resp. stops) when the number of withdrawals contained in the window is above (resp. below) a given threshold. We choose 1,500 and 9 withdrawals for the start and stop threshold, respectively, which correspond to the 99.99$^{\text{th}}$ and the 90$^{\text{th}}$ percentile of the number of withdrawals received over any 10 s period. Overall, we found a total of 3,335 bursts; 16% of them (525) contained more than 10,000 withdrawals and 1.5% of them (49) contained more than 100,000 withdrawals. Our measurements expose the following four major observations.

BGP routers often see bursts of withdrawals. We computed the number of bursts observed by a router maintaining a growing number of peering sessions randomly selected amongst the 213 RouteViews and RIPE RIS peering sessions. Figure 2.3a shows our results. The line in the box represents the median value, while the whiskers map to the 5th and the 95th percentile. In the median case, a router maintaining 30 peering sessions would see 104 (resp. 33) bursts of at

least 5k (resp. 25k) withdrawals over a month. Even if a router maintains a single session, it would likely see a few large bursts each month. Indeed, 62% of the individual BGP sessions we considered saw between 1 and 10 bursts of withdrawals, 24% saw more than 10 bursts. Only 14% of the sessions did not see any. As a comparison, single routers in transit networks routinely maintain tens to hundreds of sessions [70] – even if not all those sessions might carry the same number of prefixes as the ones in our dataset.

Learning the full extent of an outage is slow. While most of the bursts arrived within 10 s, 37% (1,239) of them lasted more than 10 s, and 9.7% (314) lasted more than 30 s (see Figure 2.3b). This also means that withdrawals within bursts tend to take a long time to be received. In the median case (resp. 75$^{\text{th}}$ percentile), BGP takes 13 s (resp. 32 s) to receive a withdrawal.

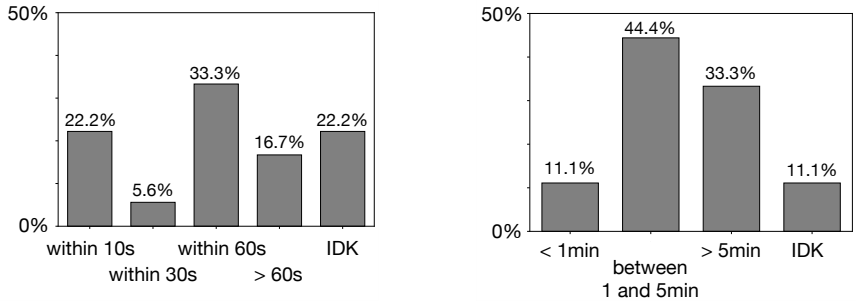
Large bursts take more time to be learned. Unsurprisingly, large bursts take more time to propagate than smaller ones (see Figure 2.3b). Overall, we found that 98 bursts took more than 1 min to arrive, with an average size of ≈81k withdrawals.

A significant portion of the withdrawals arrives at the end of the bursts. We took the bursts lasting 10 s or more and divided each of them into three periods of equal duration: the head, the middle, and the tail. We found that although most of the withdrawals tend to be in the head, 50% of the bursts have at least 26% (resp. 10%) of their withdrawals in the middle (resp. in the tail). For 25% of the bursts, at least 32% of the withdrawals are in the tail.

84% of the bursts include withdrawals of prefixes announced by "popular" ASes. We examined the Cisco "Umbrella 1 Million" dataset [15], listing the top 1 million most popular DNS domains. From there, we extracted the organizations (15 in total) responsible for the top 100 domains, such as Google, Akamai, Amazon, Apple, Microsoft, Facebook. 84% of the bursts we observed included at least one withdrawal of a prefix announced by these organizations. This demonstrates that slow Internet convergence can affect a significant fraction of the data traffic.

2.2.4 Slow routing convergence leads to significant traffic losses

Bursts of withdrawals result in downtime. While countless studies have shown that slow Internet routing convergence can cause long downtime on data-plane connectivity [106, 138, 162, 175, 167, 41], we confirmed the data-plane impact of a few bursts of withdrawals propagated by a national ISP (with more than 50 routers). Specifically, we analyzed the bursts sent by the ISP to its BGP neighbors over a period of three months. Among them, we selected three bursts that included more than 10k withdrawals and matched with an event logged by the ISP. By checking their logs, the operators identified the root causes of the



(a) The average case **(b)** The worse case

Figure 2.4 The downtime induced by the slow BGP convergence and measured by 18 network operators upon remote outages.

bursts: two maintenance operations and a peering failure at one of their IXPs. At least two of these three bursts induced downtime for transit traffic towards up to 68k prefixes, including popular destinations.

Network operators observe minutes of downtime during routing convergence. To emphasize the effect of the slow Internet routing convergence on users' connectivity, we also ask the 73 respondents of our survey about the actual downtime typically observed in their network during the BGP convergence. Among them, 17 (23.6%) collect statistics about BGP convergence and induced downtime (9 of which are transit ISPs), and 18 answered our questions about the average and worse case measured BGP convergence time upon remote outages. Here, we ask about *remote* outages because, from the perspective of the responding network operators, they require BGP to converge to restore connectivity (as opposed to local outages). Figure 2.4 shows the BGP convergence time measured by the 18 respondents. Half of them observe an average BGP convergence time upon remote outages above 30 s. Only 4 (22%) experience an average convergence time below 10 s (Figure 2.4a). In addition, 14 (78%) of the respondents observe a worst-case convergence time above 1 min, and 6 (33%) above 5 min (Figure 2.4b).

2.3 The FIB update

Once a router has learned the new paths, thanks to the routing protocols, and has recomputed its RIB, it must update its FIB to forward the traffic on the new paths. The FIB maps an IP destination to the MAC address of the calculated IP next-hop (in conjunction with the ARP table, see §1.2.2) as well as the output interface. This mapping is executed at line rate when the FIB is implemented in hardware.

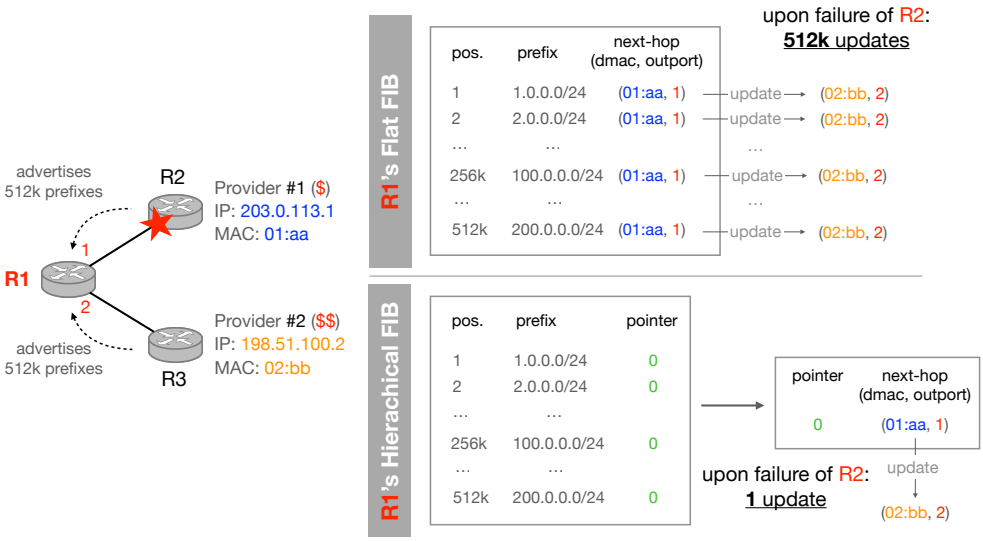


Figure 2.5 A simple scenario where R1 has two providers: R2 (the primary) and R3 (the backup). R2 and R3 both advertise 512k prefixes to R1. Upon failure of R2, a flat FIB requires 512k updates (top-right) to reroute all the traffic whereas a hierarchical FIB only requires one update (bottom-right).

In many routers, the FIB is flat, meaning it has one individual forwarding entry for each destination prefix, even though many of them share the same content and thus forward traffic to the same next-hop. We illustrate the effect of a flat forwarding table in Figure 2.5, which depicts a simple network where R1 is an edge router connected to R2 and R3, two routers in two different providers. Each of these provider routers advertises 512,000 IPv4 prefixes, which was the size of the Internet routing table in 2015 [1]. As R2 is cheaper than R3, R1 is configured to prefer R2 for all destinations. In such a case, each of the 512k FIB entries in R1 is associated with a distinct next-hop entry which all contain the MAC address of R2 as well as the outport 1. Upon the failure of a R2, every single entry of R1's FIB has to be updated, creating a significant downtime.

2.3.1 We measure the FIB update time on current routers

We measure the FIB update time of the recent Cisco Nexus 7k C7018 routers (running NX-OS v6.2, with no hierarchical FIB) that were interconnected via an HP E3800 J9575A Openflow-enabled switch. Our setup is depicted in Figure 2.6. Using this setup, we measure the FIB update time of R1 upon the failure of the link between R1 and R2. Note that we unplugged the cable directly from the router interface. Thus the failure detection is instantaneous (see §2.1), and the measured downtime corresponds to the FIB update. We rely on a hardware-based measurement using FPGA boards to measure the downtime induced by the failure with a precision of 70 μs. Such precision would be impossible to achieve using software-based measurements. We measured

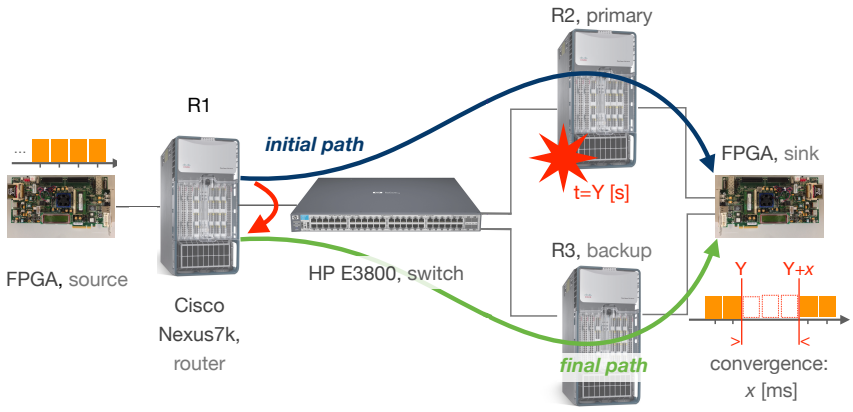


Figure 2.6 The setup we used to measure the FIB update time of a recent Cisco Nexus 7k. Our custom hierarchical FIB spans the router and the Openflow-enabled switch directly connected to the router.

the downtime by monitoring the maximum inter-packet delays each flow saw between two FPGA boards: a source that continuously sends a stream of 64-byte UDP packets towards 100 different destination IPs randomly selected, and a sink that receives the packets and reports the per-destination inter-packet delays. We implemented both the source and the sink on Xilinx ML605 evaluation boards featuring a Virtex-6 XC6VLX240T-1FFG1156 FPGA.

A flat FIB takes 2.5 minutes to reroute traffic in the worse case. Using the methodology above, we measured the FIB update time for an increasing number of prefixes (from 1k to 500k). We repeated the experiment 3 times per number of advertised prefixes. Since we measured the downtime of 100 prefixes for each experiment, we ended up with 300 statistically representative data points per measurement. Figure 2.7 depicts, in blue, the distribution of the measured downtime time using box plots. Each box shows the inter-quartile range of the FIB update time; the line in the box depicts the median value; and the whiskers show 5th and 95th percentiles. The numbers on top are the maximal FIB update time recorded.

We can see that the FIB update time is roughly linear in the number of entries to update in the FIB (not directly perceived in the figure because of the non-uniform scaling of the x-axis). This linear increase occurs because FIB entries are updated one by one; while the first FIB entry is updated immediately, regardless of the total number of prefixes, the last entry must wait for all the preceding FIB entries to be updated. This worse case highlights the undesirability of a flat FIB: as the FIB grows, so does its update time. Here, we see that `R1` takes close to 2.5min to converge when loaded with 512k prefixes. This result aligns with previous works, which also pointed out and measured the slow FIB update time [66, 69, 174].

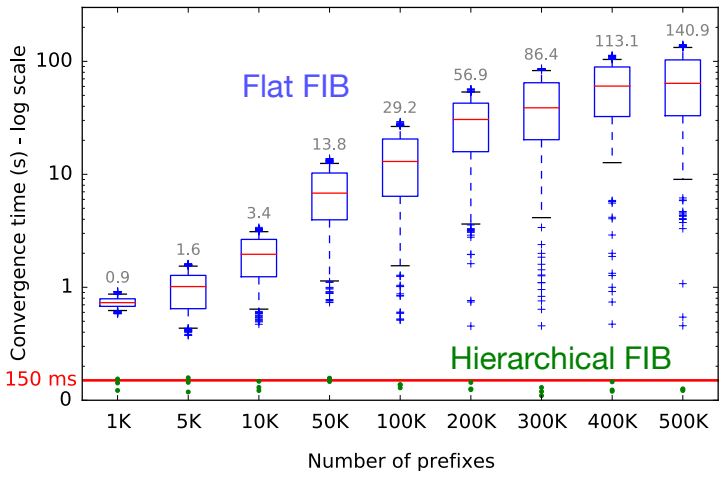


Figure 2.7 With a flat FIB, the FIB update time increases linearly with the number of entries to update. However, with our hierarchical FIB, the traffic is always rerouted within 150ms, regardless of the number of FIB entries to update.

2.3.2 Hierarchical FIB comes to the rescue

The solution to prevent downtime because of the slow FIB update is a hierarchical FIB. In its simple form, a hierarchical FIB adds a level of indirection: a first table maps a destination IP prefix to a pointer, which then resolves itself into the actual next-hop in a second table. Upon failure of a next-hop, only pointer values in the second table need to be updated to reroute all the affected traffic to a backup path. In a hierarchical FIB, the destination IP prefixes which share the same primary and backup next-hops are grouped using the same pointer. Since the number of destinations is by far greater than the number of primary and backup next-hop pairs, many destination IP prefixes will share the same pointer, enabling fast traffic rerouting.

Figure 2.5 (right-bottom) illustrates a hierarchical FIB. Because R1 selects R2 as primary and R3 as backup next-hop for the 512k destination prefixes, each entry maps to the pointer 0, which resolves itself into R2's MAC address $01:aa$ and outport 1. Upon the failure of R2, a single update in the second table is required to resolve the pointer 0 into R3's MAC address $02:bb$ and outport 2, which will immediately reroute all the traffic R3.

Hierarchical FIB in practice. Hierarchical FIBs have been successfully implemented in actual hardware routers with BGP PIC [66, 64]. Besides working in the simple case described in Figure 2.5, where the failure of a direct eBGP neighbor requires the traffic to be fast rerouted to another directly connected BGP next-hop, PIC also enables fast rerouting upon internal failures. In this case, the BGP next-hop does not change, but the IGP next-hop does. With PIC, we thus observe two levels of indirection: each prefix points to a BGP next-hop, and each BGP next-hop points to an IGP next-hop. Note that in practice, there

exist other mechanisms to quickly reroute the traffic upon internal failures, such as MPLS Fast Reroute [136]. Among the 73 network operators who participated in our survey, 17 (23%) use BGP PIC in their network, and 12 (16%) use MPLS Fast Reroute.

To illustrate the benefit of a hierarchical FIB, we implemented one which spans across two devices, the router and an SDN switch directly connected to it [22]. The router performs the first lookup, whereas the switch performs the second lookup. To provision forwarding entries in this hierarchical FIB, we built a custom controller. While the controller can rely on (typically) OpenFlow to provision forwarding entries in an SDN switch, dynamically provisioning custom forwarding entries in a router is trickier. The key insight is that our controller can use any routing protocol spoken by the router as a provisioning interface. Indeed, FIB entries in a router forward traffic to the MAC address and the outport associated with the IP next-hop learned via the routing protocol. To provision custom forwarding entries in the router, our controller first interposes itself between the router and its peers. Then, it computes primary and backup next-hops for all IP destinations. Finally, it provisions "pointers" by setting the IP next-hop field to a virtual IP next-hop that gets resolved by the router into a fake next-hop MAC address (the pointer) using ARP. Upon failure of R2 in Figure 2.6, all the controller has to do to converge is to modify the pointer value by adding a table entry in the switch with Openflow, which matches on the fake destination MAC address (the pointer) and set the outport to 2, in order to reroute all traffic to R3.

With a hierarchical FIB, R1 systematically converges within 150ms. We measured the update time of our hierarchical FIB spanning two devices using the same methodology as in §2.3.1, and report the results in Figure 2.7 (green dots). The time to update our hierarchical FIB is constant irrespective of the number of prefixes. This constitutes a 900× improvement factor compared to the worst-case measured with a flat FIB.

2.3.3 Recent advances on fast forwarding updates

One limitation with a simple hierarchical table is that it only allows installing one backup route for every destination prefix. Depending on the failure, the backup routes can thus be suboptimal or worse, can be affected by the failure too. Therefore, A fast reroute framework should pre-compute several backup routes for different potential failures and install all of them in the switch so that the best backup route for a given failure can quickly be activated. However, handling more backup routes for each prefix requires more complex data structures than *e.g.*, BGP PIC, at the control and the forwarding levels.

At the switch level, Openflow supports fast-failovers to alternative links without control-plane consultation and with minimal packet loss [134]. However, the decision about which alternative path to use for a packet is limited to a sequence of ports such that the packet is forwarded to the first active (*i.e.*, non-failed) port in the sequence. PURR shows how to implement such a fast failover mechanism in P4 switches, using ternary matches to avoid packet recirculation and maximize the switch throughput [45].

At the ISP level, OPTIC aims to fast reroute transit traffic towards the optimal BGP route after an internal event [118]. Hence, several backup routes need to be pre-computed and installed in the FIB of the routers for each prefix to protect them against any possible IGP event optimally.

3

Local fast reroute upon remote outages

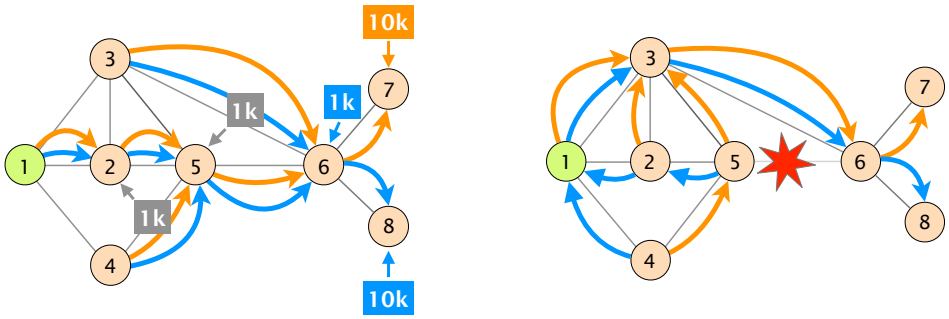
In Chapter 2, we detailed the Internet convergence process and showed evidence that it can still take *minutes*. We showed solutions to quickly recover connectivity upon a silent change such as a link or router failure. Among others, network operators can configure BFD on the routers to detect an outage and configure BGP PIC to quickly reroute traffic on working backup paths.

Unfortunately, we also showed that these solutions only work locally, and speeding up the Internet convergence network-wide remains an open problem. From the perspective of a network operator, the lack of solutions that work network-wide means that upon a remote outage, there is currently no other way for its routers to wait for the slow control-plane notifications (BGP messages) before they can update their FIB, prefix by prefix, to reroute the traffic to alternative working paths.

In this dissertation, our goal is to implement a framework to ***locally*** fast reroute traffic upon ***remote*** transit outages. In this chapter, we start by introducing the problem of local fast reroute upon remote outages (§3.1). We then show why a fast reroute framework targeting remote Internet outages is useful (§3.2). Finally, we highlight the key conceptual challenges (§3.3) as well as the key operational requirements (§3.4).

3.1 Definition and illustration of a remote outage

From the perspective of a particular AS, we define a *remote* outage as an outage occurring in another AS that affects one or multiple BGP peering links. The AS on which we focus thus has no control over the remote AS undergoing the outage and on the affected BGP peering links.



(a) Initial forwarding paths **(b)** Post-failure paths

Figure 3.1 Example of slow convergence upon a remote outage. Routing policies and the absence of information about physical connectivity force AS1 to wait for 11k BGP withdrawals, one per prefix owned by AS6 or AS8.

We illustrate a remote outage and its effect using the network depicted in Figure 3.1. For the sake of exposition, we use an AS as the basic routing element. Each AS i originates a distinct set of prefixes S_i. We focus on the 21k prefixes of S_6, S_7, and S_8, before and after the failure of the link $(5, 6)$. Figure 3.1a and Figure 3.1b respectively show pre- and post-failure AS paths. AS5 knows an alternate path for S_7 (via AS3) before the failure. However, because of inter-domain policies (*e.g.*, partial transit [147]), it does not know any backup path for S_6 and S_8. For those prefixes, AS 5 recovers connectivity after the failure via AS2.

After the failure of $(5, 6)$, AS5 restores connectivity for S_7 almost immediately by rerouting traffic to its alternate path (through AS3). However, since AS5 does not have backup paths for S_6 and S_8, a black hole is created for flows directed to the corresponding 11k prefixes. In the control plane, the failure causes AS5 to send 10k path updates to notify that it now uses new paths to reach S_7, along with 11k path withdrawals to communicate the unavailability of path $(5, 6, 8)$. For AS1 and AS2, the failure of $(5, 6)$ is a remote outage, which comes with a loss of traffic towards S_6 and S_8. These two ASes have no other option than waiting for the 11k BGP withdrawals and updating the corresponding 11k FIB entries to reroute the traffic for S_6 and S_8 to a working backup path.

In practice, BGP peering link failures are everyday events [33]. Besides, an outage can affect multiple BGP peering links when they share the same physical infrastructure (*e.g.*, the same layer-2 switch). The links sharing the same physical infrastructure can be grouped in a Shared Risk Link Group (SRLG). A study showed that 30% of the unplanned failures affect multiple links [122].

3.2 Motivation to fast reroute upon remote outages

The problem with the outage illustrated in Figure 3.1 is that there is currently no way for AS1 and AS2 to restore connectivity quickly. Indeed, the current deployed BGP fast reroute techniques (*e.g.*, BGP PIC [66]) only work for local failures, not the remote ones.

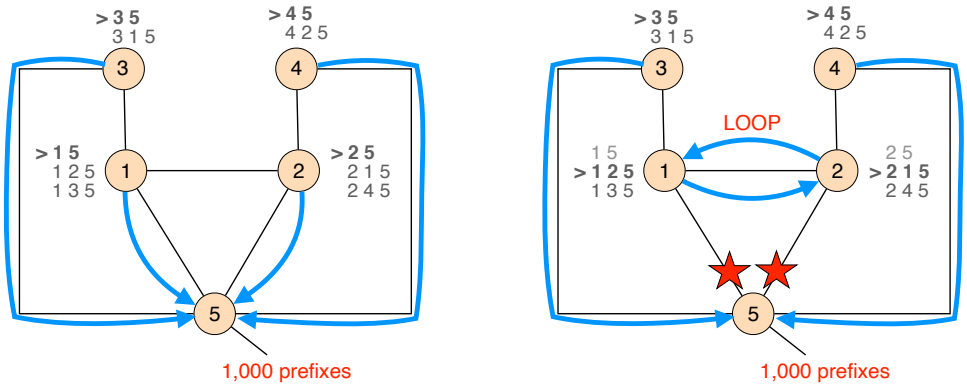
In principle, though, if all ASes can locally fast reroute traffic to a working backup path upon any local failure (including SRLG failures), fast rerouting upon a remote outage is unnecessary. Unfortunately, fast traffic rerouting upon a local failure is not always possible in practice, motivating the deployment of a fast reroute framework targeting the remote outages. We explain why the deployment of such a framework is beneficial in the following paragraphs.

There is not always a working backup path when the failure occurs. One key challenge of any fast reroute technique is to find a *working* backup path. For instance, IPFRR [158] reroutes the traffic to LFAs to prevent forwarding loops. As a vector-based routing protocol, BGP inherently hides some of the backup paths that would result in forwarding loops in case of traffic rerouting. Take, for instance, the case depicted in Figure 3.1. AS5 only learns the backup path via AS2 for prefixes in S_6 and S_8 after it sends withdrawals for these prefixes and AS1 and AS2 converge and use the path via AS3 to reach these prefixes.

The problem is that depending on the topology, there might not exist working backup paths. In the case of IPFRR, the per-link LFA coverage on actual core ISP topologies ranges from 16% to 98% [65]. This number can be improved using Remote LFAs, which consists in using IP tunnels to have additional logical links and increase the LFA coverage [39, 58]. In the case of BGP, AS5 in Figure 3.1 does not have a LFA for prefixes in S_6 and S_8. Using remote LFAs with BGP is difficult, as it would require cooperation between ASes. Hence, in this example, AS5 cannot fast reroute traffic for prefixes in S_6 and S_8.

Naively rerouting traffic can result in forwarding loops. Forwarding loops can happen upon an SRLG failure because multiple ASes may fast reroute traffic independently, thus possibly resulting in inconsistent forwarding states. Consider the example depicted in Figure 3.2. There are 5 ASes, and we focus on the paths towards the 1,000 prefixes advertised by AS5, a multi-homed stub AS. Here again, for the sake of exposition, we use an AS as the basic routing element. We indicate next to each AS the possible paths to reach the 1,000 prefixes and order them based on their preference. For instance, AS1 prefers path $1-5$ over path $1-2-5$, which is preferred over path $1-3-5$.

Consider now that the two links between $1-5$ and $2-5$ fail, for instance, because a line card of the router in AS5 is broken or because the traffic between the two pairs of ASes traverses a broken layer-2 switch. If both AS1 and AS2 fast reroute the traffic for the 1,000 prefixes towards their best backup path, *i.e.*, the path $1-2-5$ for AS1 and the path $2-1-5$ for AS2, then the traffic



(a) Initial forwarding paths **(b)** Naive traffic rerouting upon failure

Figure 3.2 Naively fast rerouting traffic at the inter-domain level can cause forwarding loops. Blue arrows indicate forwarding paths towards the 1,000 prefixes.

would loop between AS1 and AS2 until BGP converges and AS1 (resp. AS2) reroutes traffic to AS3 (resp. AS4). In this example, AS1 should instead use AS3 as backup and AS2 should use AS4 as backup to quickly restore connectivity. To prevent the cure from being worse than the disease, operators might stay cautious and wait for BGP to converge.

Fast traffic rerouting upon BGP peering link failures is tricky. There are BGP fast reroute techniques that prevent forwarding issues. For instance, Bonaventure et al. propose a set of techniques to enable fast traffic rerouting upon BGP peering links failures and without forwarding anomalies [33]. They consider SRLG failures and propose solutions to protect the *outgoing* and *incoming* traffic of an AS. Consider again the stub AS5 in Figure 3.2. The goal for AS5 is to protect both the *outgoing* traffic as well as the the *incoming* traffic. For the *outgoing* traffic, the solution is rather simple: the primary egress router in AS5 can fast reroute traffic to a backup egress router not in the same SRLG using IP tunnels and a hierarchical FIB. For the *incoming* traffic, the problem is trickier as the provider's network must reroute traffic. In this case, naively rerouting to the best backup path could result in forwarding loops or black holes, as we show in Figure 3.2b.

Now, consider the scenario in Figure 3.2 again and imagine there is one distinct router at each end of the BGP peering links. Assume that network operators follow the standard practices and configure their routers to maintain iBGP sessions fully mesh. Following the idea in [33], upon the failure of the link $(1, 5)$ and $(2, 5)$, the primary egress router in AS1 should fast reroute the traffic to a secondary ingress router in AS5 that is not in the same SRLG than the primary ingress router in AS5. In other (and simpler) words, AS1 should fast reroute the traffic to AS3, which will then send the traffic to AS5, bypassing the failure and restoring connectivity.

The problem is that setting up these backup paths at the inter-domain level requires cooperation between ASes. First, the provider (here, AS1) needs to learn which secondary ingress router to use in AS5 and use it. A possible solution for AS5 is to announce the secondary ingress router over the eBGP session with the provider. Second, the primary egress router in the provider's network should reach the secondary ingress router without using the failed link. For instance, in Figure 3.1, the primary egress router in AS1 should reach the secondary ingress router in AS5 via AS3. This backup path can be used by configuring on the secondary ingress router in AS5 an IP address that belongs to an IP prefix owned and advertised by the provider directly connected to that router (here AS3).

Many networks are not designed for fast traffic rerouting upon BGP peering link failures. To simplify the solution, AS5 could install multiple parallel peering links ending at different locations with its providers. In this case, each provider would know a backup path that it can directly use instead of rerouting traffic to a remote secondary ingress router reachable via other ASes. However, installing peering links is expensive and not always possible for small stub ASes. Among the 73 operators who participated in our survey, 18 (24.7%) do not have redundant peering links, 9 (12,3%) do establish parallel physical connections with their peers, but which end on the same router on their end, and 5 (7%) said that it depends. The remaining 41 operators (56%) establish parallel physical connections with their peers, which necessarily end on different routers on their end. Those results highlight that an important fraction of the ASes is not prepared for fast traffic rerouting upon BGP peering link failures. To reinforce this statement, we note that only 17 of our respondents (23%) uses BGP PIC [66], one of the main existing solutions to fast reroute traffic on *e.g.*, IP tunnels to recover connectivity upon BGP peering link failures quickly.

The vast majority of the operators would be interested in a fast reroute solution for remote outages. Namely, 95% of our respondents indicated that they would consider adopting a fast reroute solution to speed up convergence upon remote outages.

3.3 Key conceptual challenges

There are three main conceptual challenges when building a fast reroute framework targeting remote transit outages: *(i)* detecting the outage; *(ii)* assessing the affected traffic; and *(iii)* rerouting the affected traffic towards working backup paths. This section shows the state-of-the-art solutions that try to solve these challenges and explains why a fast reroute framework targeting remote outages need new tailored mechanisms to solve these challenges.

3.3.1 Detection of the remote outage

As a path-vector protocol, BGP does not explicitly notify the status of the AS links but instead (slowly) propagates announcements or withdrawals on a per-prefix basis. Besides, the currently deployed fast failure detection mechanisms such as BFD [98] rely on a point-to-point connection and thus only work upon local failures, not the remote ones. Consequently, remote Internet outages must be heuristically detected.

State-of-the-art solutions. Remote outages can be detected from the control plane because BGP instabilities and, more particularly, bursts of BGP updates and withdrawals coincide with Internet failures [61, 176].

They can also be detected from the data traffic. For instance, FACT looks at NetFlow [50] data and analyzes on a per-flow basis the cases where hosts in possible remote ASes are unresponsive [152]. Chocolatine looks at the Internet background radiation [53] and uses a statistical model to infer outages from the number of different IP addresses observed for each AS and over time [77]. Remote outages can also be detected from the state of the TCP connections. For instance, Disco looks at the long-running TCP connections between the RIPE Atlas probes and the RIPE Atlas infrastructure [16] and identifies bursts of disconnections, which are signs of a severe outage [156].

Finally, active probing can detect remote outages. Active probing purposely injects traffic (typically ICMP probes) into the network to infer whether two end-points can communicate and through which IP path. A good example is Trinocular, which performs ping measurements at regular intervals and towards each IP block in the Internet to infer outages [144]. Similarly, ThunderPing sends ICMP probes towards residential Internet hosts to measure the correlation between bad weather conditions and Internet outages [154].

As injecting packets adds load on the network, active probing solutions often include a mechanism to adapt the probing rate depending on the likelihood of detecting an outage. For instance, Trinocular minimizes the probing traffic by adapting the probing rate based on a Bayesian inference model. Odd behaviors reported from passive measurements can also trigger active measurements. For instance, PlanetSeer monitors traffic between PlanetLab nodes [47] and its users [180]. Upon detection of anomalous behavior, PlanetSeer triggers active probing from PlanetLab nodes to further investigate the issue. Hubble triggers `ping` and `traceroute` measurements when it detects changes in the BGP control-plane [100]. With this approach, Hubble reduces the probing traffic by 94.5% compared to a naive probing strategy while still detecting most outages.

What a fast reroute framework targeting remote outages needs. The detection mechanism must be fast, ideally within few seconds only. This is far from the detection speed of existing solutions, which need *minutes* to detect remote outages. For instance, Trinocular probes in rounds of 11 minutes whereas

Hubble monitors at a 15 minutes granularity. Chocolatine and FACT both detect outages within with a 5 minutes timescale.

The detection mechanism must also detect many of the remote outages. From the perspective of a particular AS, many remote outages that may affect its connectivity should be detected, with a higher priority given to the outages that would affect a significant part of the traffic. This requirement hinders the use of public measurement platforms, such as RIPE Atlas or PlanetLab, which only cover a subset of the paths in the Internet.

3.3.2 Assessment of the affected traffic

As BGP works on a per-prefix basis, our goal is to infer the affected traffic on a per-prefix basis too. The first obvious solution is to monitor each prefix individually using passive measurements and active probing. However, this can be very resource-hungry, especially given the always-increasing number of prefixes advertised within the Internet [1]. An alternative solution is to locate each outage and use this information to assess the affected traffic, *e.g.*, using the AS paths learned from BGP.

State-of-the-art solutions. Because Internet outages correlate with BGP routing instability [61] and BGP routes contain useful information such as the AS path, many existing techniques use BGP data (*e.g.*, obtained from the BGP route collectors [16, 132]) to localize Internet outages. For instance, Caesar et al. look at BGP routes observed at multiple vantage points to pinpoint the location of the issues [43]. They observe that if a large number of BGP updates, for many prefixes, exhibit the same AS link in the AS path, then this link is a potential candidate for the location of the issue. Feldmann et al. propose to locate the ASes responsible for a routing change by correlating BGP information received from multiple vantage points across time, views, and prefixes [63]. PoiRoot improves the performance of the localization by using a set of measurement techniques to *(i)* estimate the set of ASes to monitor to perform the route cause analysis and *(ii)* collect the actual AS-level paths used towards the targeted ASes [93]. Besides active probing with `traceroute` and passive BGP monitoring with data collected from multiple vantage points, PoiRoot uses BGP poisoning (on dedicated IP prefixes) to learn which alternative paths the monitored ASes use towards the poisoned prefixes (which are otherwise invisible). With the collected data, PoiRoot then uses a recursive algorithm to isolate the root cause of the path changes accurately.

NetDiagnoser relies on the principle of binary network tomography to identify the location of network failures [55]. More precisely, NetDiagnoser infers the IP-level topology of the network using `traceroute` measurements performed by sensors located in multiple ASes and uses it to pinpoint the set of links affected by an outage. To mitigate typical issues with `traceroute` measurements (such as

non-responding hops), NetDiagnoser also uses information collected from BGP Looking Glasses. Finally, Fontugne et al. propose a statistical model to pinpoint in near real-time Internet outages using the RTTs and packet loss extracted from the publicly available `traceroute` data of the RIPE Atlas platform [67]. For instance, they detected the Amsterdam Internet Exchange (AMS-IX) outage in 2015 and were able to determine the peers that could not exchange traffic during the outage [13].

What a fast reroute framework targeting remote outages needs. Unfortunately, the current outage localization techniques focus on the quality and precision and not on the speed of the localization. They thus benefit from more flexibility in terms of inference algorithms and input sources (*e.g.*, active probing from multiple vantage points) but need *minutes* to localize an outage.

As speed is one of the key requirements for fast rerouting, we thus envision a different approach which "trades" accuracy to gain speed. Should the assessment not be perfectly accurate, some of the affected traffic will not be rerouted and will keep using the non-working primary routes until BGP converges, whereas some of the non-affected traffic might be unnecessarily rerouted, resulting in short-lived path suboptimality.

3.3.3 Remediation

When an outage is detected and the affected traffic assessed, the goal is to restore connectivity by quickly rerouting the affected traffic towards working backup paths. Rerouting traffic on working backup paths is challenging because multiple prefixes can be affected by an outage, and for each of them, the set of working backup paths depends on the location of the outage. An obvious solution is to monitor, for each prefix, each backup path individually and use the ones that are still working. However, monitoring all the backup paths takes resources. We thus prefer to leverage both path diversity and failure localization to reroute the traffic on backup paths bypassing the failure and restoring connectivity.

State-of-the-art solutions. A simple approach is to look at the AS path of the BGP routes and use the backup paths that avoid the outage. As an illustration, Bremler-Barr et al. [36] look at the Round Trip Time (RTT) deviation from logs of HTTP requests collected from different servers to detect degradations and combine these measurements with BGP data to locate the degradations. The authors then show that predicting degradations enables "intelligent routing," which consists in leveraging path diversity by selecting the best path for each destination according to the predictions.

LIFEGUARD [101] is an example of a system that automatically localizes Internet outages and restores connectivity dynamically by rerouting traffic

around the outages. Unlike the other remediation techniques, LIFEGUARD shows how an AS can restore connectivity for reverse paths, *i.e.*, for the inbound traffic. The case of reverse paths is more challenging than for the forward paths as there is no explicit mechanism to influence the routing decisions made by remote networks (the MED attribute is only helpful with the neighboring ASes). After inferring potential candidates for failure locations using `traceroute` and reverse `traceroute` [99] and isolating the direction of the failure using spoofed pings, LIFEGUARD restores connectivity on the reverse path by poisoning the BGP advertisements [160] so that the inbound traffic circumvents the outage.

What a fast reroute framework targeting remote outages needs. The existing outage remediation techniques such as LIFEGUARD are not tailored for fast traffic rerouting but aim to address the long-lasting outages, *i.e.*, the ones that are often not accompanied by routing updates and take hours to be fixed manually. LIFEGUARD aims to repair these outages within *minutes* automatically. On the contrary, our goal is to prevent connectivity issues caused by the slow Internet convergence by rerouting traffic within few seconds. Thus, besides rerouting on working backup paths, our solution must include a mechanism to quickly update the FIB of the routers for all the affected prefixes.

3.4 Key operational requirements

In this section, we list the key operational requirements required for practical deployment. To draw this list, we take into account the feedback from the 73 network operators who participated in our survey. Specifically, we asked them about the most relevant characteristics of a fast reroute framework for remote outages. Sixty-nine operators responded and results are summarized in Figure 3.3.

Easily deployable. Deployability is one of the main requirements expressed by network operators. More precisely, 43 (62.3%) of the participants care about the ease of deployment, 27 (39.1%) about the cost, and 16 (23.2%) about the ease of maintainability. Two respondents specifically said the solution should be compatible with the BGP standards. Deployability matters because changing the Internet standards is impracticable in the short term. Ease of deployment and cost also means that the framework should only require a software update or the reconfiguration of a device but should not require the manufacturing and installation of new expensive hardware.

Incrementally Deployable. A single, partial or complete deployment of the framework should be possible. The rerouting should not cause forwarding loops, irrespective of the number of networks deploying the framework. A complete deployment precludes intensive active probing techniques, which would likely add too much load to the network. Ideally, deploying the framework should always

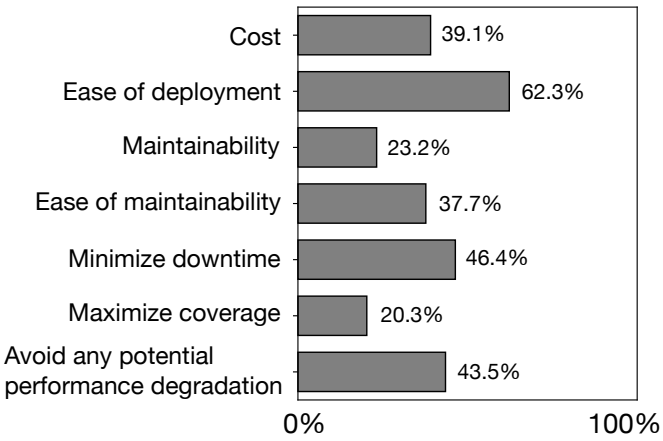


Figure 3.3 The most relevant characteristics a fast reroute framework for remote outages should have according to network operators.

be beneficial: the higher is the number of networks deploying the framework, the better is the protection against Internet outages.

Performant and precise. Unsurprisingly, network operators care about the performance and precision of the system. More specifically, 32 (46.4%) care about minimizing downtime, and 30 (43.5%) care about potential performance degradation. Unfortunately, as our heuristics rely on partial and noisy signals, they cannot be perfectly accurate. As a result, we must find a tradeoff between minimizing downtime (*i.e.*, rerouting as much of the affected traffic as possible) and avoiding potential performance degradation (*i.e.*, avoiding rerouting non-affected traffic).

We thus asked network operators whether they would mind if a fast reroute solution for remote outages would temporarily reroute traffic on a suboptimal (but working) backup path for prefixes not affected by the outage. Sixty-eight network operators responded. Among them, 33 (48.5%) operators answered they would not mind, 27 (39.7%) answered they would mind, and 8 (11.7%) said that it depends. We see that the plurality of the network operators is willing to trade path suboptimality for less downtime. Yet an important fraction of the operators does mind about path suboptimality. We thus tolerate path suboptimality only if there is a significant reduction of downtime.

Secure. As the framework controls the selection of the forwarding paths, malicious Internet players may have high interests in manipulating the system. The main threat is that malicious users could manipulate the framework and reroute the traffic destined to the victim prefixes to their network. In addition, there are other possible threats, such as manipulating the framework to prevent fast rerouting upon failures or deliberately adding load on the routers to impact its routing and forwarding performance. Ideally, the framework should detect, report and thwart these attacks.

4

SWIFT: Predictive Fast Reroute

In this chapter, we describe SWIFT, a locally-deployable framework which enables existing routers to restore connectivity in *few seconds* upon remote Internet outages.

SWIFT is based on two main ingredients. Immediately after receiving the first few BGP withdrawals of a burst, which correspond to the first signs of a remote outage, a SWIFT router runs an inference algorithm (§4.2) to localize the outage and predict which prefixes will be affected—a sort of time-bound Root Cause Analysis (RCA). Based on this inference, the SWIFTED router reroutes the potentially affected prefixes on paths unaffected by the inferred failure. As many prefixes may have to be rerouted at once, SWIFT also includes a data-plane encoding scheme (§4.3) that enables the router to flexibly match and reroute all prefixes affected by a remote failure with few data-plane updates.

Balancing inference accuracy & speed, with correctness & performance in mind. The key insight of the SWIFT inference algorithm compared to the prior (slow) RCA studies is that some accuracy can be traded for a significant gain in speed. Identifying the topological region where an outage is happening is indeed much faster than precisely locating the outage within that region. By rerouting traffic around the region, a SWIFTED router immediately restores connectivity for the affected prefixes at the cost of temporarily forwarding a few (according to our results) unaffected prefixes on alternate working paths.

SWIFT makes sure that the effect of diverting non-affected traffic does not trump the benefit of saving traffic towards the affected prefixes. First of all, we prove that rerouting non-affected traffic is safe: SWIFT does not lead to forwarding anomalies, even if multiple routers and ASes deploy it. Second, SWIFT selects the alternate paths taking into account the operator's policies (*e.g.*, type of peers, cost model) and performance criteria (*e.g.*, by preventing to reroute a large amount of traffic to low-bandwidth paths).

Deployment. SWIFT is deployable on a per-router basis and does not require cooperation between ASes, nor changes to BGP. SWIFT can be deployed with a simple software update since the only hardware requirement, a two-stage forwarding table, is readily available in recent router platforms [66].

Whenever a SWIFTED router fast-reroutes upon an outage, it guarantees connectivity to all the traffic sources passing through it. Hence, deploying SWIFT in a few central ASes would benefit the entire Internet, since these ASes would also protect their (non-SWIFTED) customers. The same applies within a network: deploying a few SWIFTED routers at the edge boosts convergence network-wide. A full Internet SWIFT deployment would achieve the utmost advantages of our scheme, as it guarantees ASes to reroute quickly, independently, and consistently with their policies.

Performance. We implemented SWIFT and used our implementation to perform extensive experiments using both real and synthetic BGP traces. Across all our experiments, SWIFT correctly identified 90% of the affected prefixes within 2 seconds. Moreover, a SWIFTED router can fast reroute 99% of the predicted prefixes with few data-plane rule updates, *i.e.*, in milliseconds. Finally, we show that our implementation is practical by using it to reduce the convergence time of a recent Cisco router by more than 98%.

Our implementation of SWIFT is open-source and available at `www.swift.ethz.ch`.

4.1 Overview

Figure 4.1 shows the workflow implemented by a SWIFTED router. We now describe the result of implementing such workflow on a BGP border router of `AS1` in Figure 3.1. Here, for simplicity and without loss of generality, we assume that a single router in `AS1` maintains all the BGP sessions with `AS2`, `AS3` and `AS4`.

Before the outage. The SWIFTED router in `AS1` continuously pre-computes backup next-hops (consistently with BGP routes) to use upon remote outages. This computation is done for each prefix and considering any possible inter-AS link failure on the corresponding AS path. For example, the `AS1` router chooses `AS3` or `AS4` as backup next-hop for rerouting the 20k prefixes advertised by `AS7` and `AS8` upon the failure of link $(1, 2)$. In contrast, it can only use `AS3` as backup to protect against the failure of link $(5, 6)$ for the same set of prefixes, since AS 4 also uses $(5, 6)$ prior to the failure. SWIFT then embeds a data-plane tag into each incoming packet. Each SWIFT tag contains the list of AS links to be traversed by the packet, along with the backup next-hop to use in the case of any link failure.

Upon the outage. After receiving a few BGP withdrawals caused by the failure of $(5, 6)$, the SWIFTED router in AS1 runs an inference algorithm that quickly identifies a set of possibly disrupted AS links and affected prefixes. The router then redirects the traffic for all the affected prefixes to the pre-computed backup next-hops. To do so, it uses a single forwarding rule matching the data-plane tags installed on the packets. As a result, AS1 reroutes the affected traffic in less than 2 s (independently from the number of affected prefixes), a small fraction of the time needed by a router (see Figure 2.7). When rerouting, SWIFT does not propagate any message in BGP. We proved that this is safe provided that the SWIFT inference is sufficiently accurate (§4.1.3). When BGP has converged, *i.e.*, the burst of withdrawals has been fully received and BGP routes have been installed in the forwarding table, the router removes the forwarding rules installed by SWIFT and falls back to the BGP ones.

In the following, we provide more details about the main components of SWIFT. In §4.1.1, we overview the inference algorithm that we then fully describe in §4.2. In §4.1.2, we illustrate how SWIFT quickly reroutes data-plane packets on the basis of tags pre-computed by the encoding algorithm detailed in §4.3. We finally report about SWIFT guarantees in §4.1.3.

4.1.1 Inferring outages from few BGP messages

The SWIFT inference algorithm looks for peaks of activity in the incoming stream of BGP messages. Each detected burst triggers an analysis of its root cause. To identify the set of links with the highest probability of being affected by an outage, the algorithm combines the implicit and explicit information carried by BGP messages about active and inactive paths. For example, the failure of $(5, 6)$ in Figure 3.1 may cause BGP withdrawals indicating the unavailability of paths $(1, 2, 5, 6)$ and $(1, 2, 5, 6, 8)$ for all the prefixes originated by AS 6 and 8. Receiving these withdrawals makes the algorithm assign a progressively higher failure probability to links in $\{(1, 2), (2, 5), (5, 6), (6, 8)\}$. Over time, the algorithm decreases the probability of links $(1, 2)$ and $(2, 5)$, because prefixes originated by AS2 and AS5 are not affected, and the probability of link $(6, 8)$, because not all the withdrawn paths traverse $(6, 8)$.

SWIFT aims at inferring failures quickly, yet keeping an eye on accuracy. Inference accuracy and speed are conflicting objectives. Indeed, precisely inferring the set of affected AS links might be impossible with few BGP messages, as they might not carry enough information. For instance, SWIFT cannot reduce the set of likely failed links any further than the entire path $(1, 2, 5, 6, 8)$ until it receives other messages than withdrawals for that path. Rerouting based on partial information can unnecessarily shift non-affected traffic, *e.g.*, all the prefixes originated by AS2 and AS5. In contrast, waiting for BGP messages takes precious time (§2.2) during which traffic towards actually-affected prefixes can be dropped.

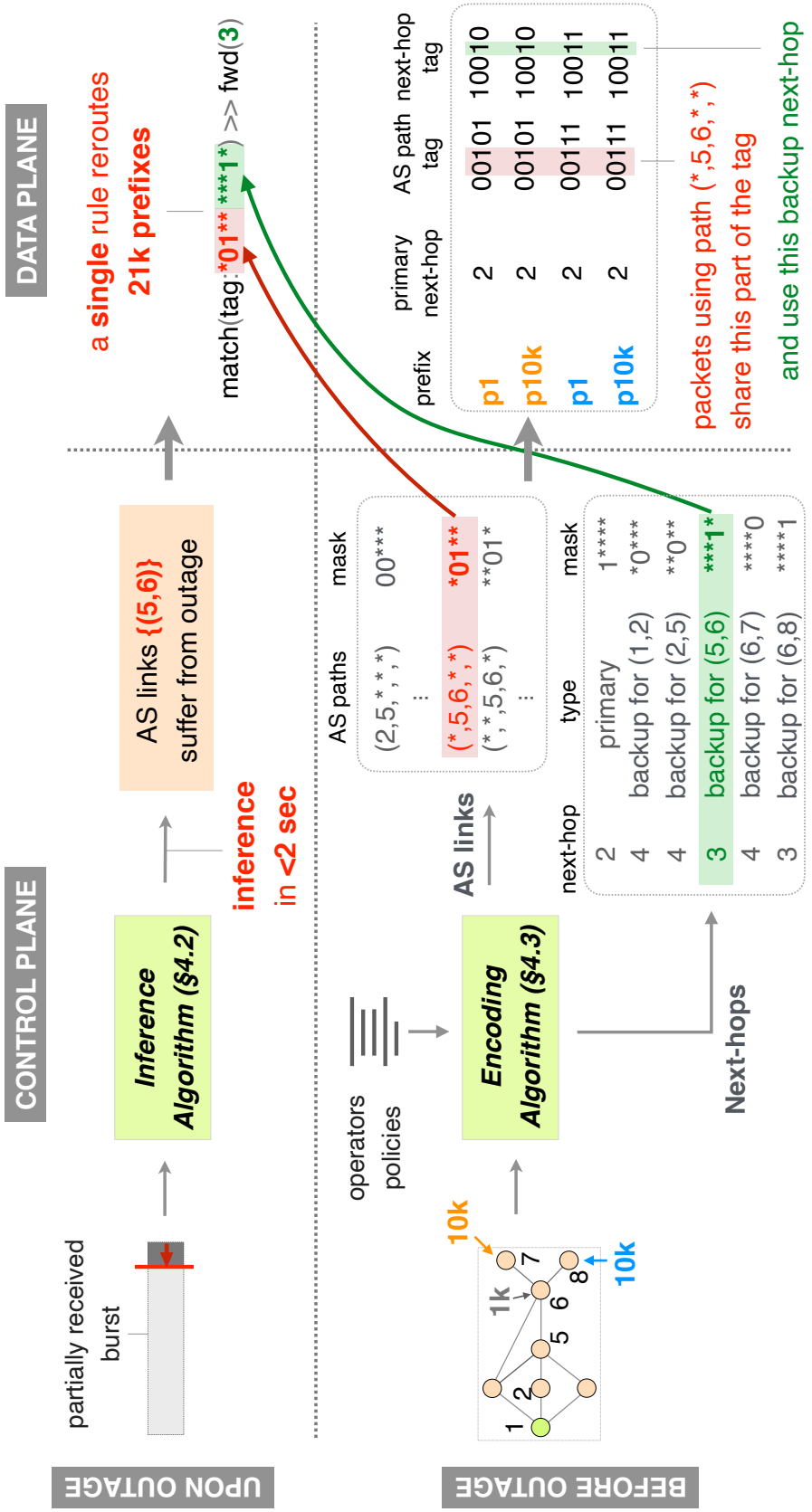


Figure 4.1 SWIFT workflow.

To avoid unnecessary traffic shifts, SWIFT evaluates the likelihood that its inferences are realistic (*e.g.*, using historical data). For instance, SWIFT evaluates the probability that a burst including withdrawals for all the prefixes originated by AS6, AS7 and AS8 happens. If a burst of similar size is unlikely, SWIFT waits for the reception of more messages to confirm its inference. Given that withdrawals for prefixes from AS7 and AS8 will likely be interleaved with path updates for AS6, this strategy quickly converges to an accurate inference, as we show in §4.4.

SWIFT uses a conservative approach to translate inferences into predictions of affected prefixes. Remote failures are often partial, that is, an outage can cause traffic loss for a subset of the prefixes traversing the affected link(s). For instance, a subset of the prefixes traversing the failed link $(5, 6)$ in Figure 3.1 can remain active because of physical link redundancy between AS5 and AS6, or be rerouted by intermediate ASes (*e.g.*, AS5) to a known backup path (like the prefixes originated by AS7). As BGP messages do not contain enough information to pinpoint the set of prefixes affected by an outage, SWIFT reroutes all the prefixes traversing the inferred links. Doing so minimizes downtime at the potential cost of short-lived path sub-optimality (for a few minutes at most).

SWIFT inference works well in practice. Our experiments on real BGP traces (see §4.4) show that SWIFT enables to reroute 90% (median) of the affected prefixes after having received a small fraction of the burst, and less than 0.60% of the non-affected prefixes.

4.1.2 Fast data-plane updates independently of the number of affected destinations

Upon an inference, a SWIFTED router might need to update its FIB for thousands of prefixes. As we show in Section 2.3, routers are slow to perform such large rerouting operations as they update their data-plane rules on a per-prefix basis. Besides, as the outage affects remote AS links, local fast-rerouting techniques [66] cannot be applied. According to our measurements (Section 2.3.1) and the previous studies, the median update time per-prefix is between 128 and 282 μs. Hence, current routers would take between 2.7 and 5.9 seconds to reroute 21k prefixes and *more than 1 minute* for the full Internet table (650k prefixes) – even if BGP could converge instantaneously.

SWIFT speeds up data-plane updates by rerouting according to packet tags instead of prefixes. A SWIFTED router relies on a two-stage forwarding table to speed up data-plane updates. The first stage contains rules for tagging traversing packets. SWIFT tags carry two pieces of information: *(i)* the AS paths along which they are currently forwarded; and *(ii)* the next-hops to use in the absence (primary next-hops) or presence (backup next-hops) of any AS-link

failure. The second stage contains rules for forwarding the packets according to these tags. By matching on portions of the tags, a SWIFTED router can quickly select packets passing through any given AS link(s), and reroute them to a pre-computed next-hop. Since tags are only used within the SWIFTED router, they have local meaning and are not propagated in the Internet (they are removed at the egress of the SWIFTED router).

Using again Figure 3.1, we now describe the rules in the forwarding table of the SWIFTED router in AS1. Figure 4.1 shows the tags returned by the SWIFT encoding algorithm. The first stage of the forwarding table contains rules to add tags consistently with the used BGP paths. Since prefixes in AS8 are forwarded on path $(2, 5, 6, 8)$, it contains the following rule.

```
match(dst_prefix:in AS8) >> set(tag:00111 10011)
```

The first part of the tag identifies the AS path. It maps specific subsets of bits to AS links in a given position of the AS path. The first two bits represent the first link in the AS path, which is link $(2, 5)$. Consistently with Figure 4.1, those bits are therefore set to 00. Similarly the second and third bits represent link $(5, 6)$ when it is the second link in the AS path, etc.

The second part of the tag (in green) encodes the primary and backup next-hops. Namely, the first bit identifies the primary next-hop, the second bit indicates the backup next-hop to use if link $(1, 2)$ fails, etc. This part of the tag enables SWIFT to match on traffic that may have to be redirected to potentially different next-hops depending on the link that fails and the destination prefix.

Before the failure of $(5, 6)$, the second stage only contains the forwarding rules consistent with BGP. Specifically,

```
match(tag:***** 1****) >> fwd(2)
```

Upon the failure of $(5, 6)$, SWIFT adds a *single* high-priority rule to the second stage – while not modifying at all the first stage.

```
match(tag:*01** ***1*) >> fwd(3)
```

The added rule exploits the structure of SWIFT tags to reroute traffic for all the affected 21k prefixes, at once.

The regular expression in it matches all the packets such that: $(5, 6)$ appears as the second link in their AS path (*i.e.*, the tag starts with *01**); and the backup next-hop is 3 (*i.e.*, the tag ends with ***1*). This includes traffic for prefixes in AS6, AS7 and AS8. Note that one rule is sufficient in our example, because the SWIFTED router does not use any AS path where $(5, 6)$ appears in other positions before the failure (otherwise, one rule per position is needed).

SWIFT compresses tags efficiently. Assigning subsets of bits for any AS link and possible position in the AS path does not scale for the Internet AS graph that currently includes >220,000 AS links. SWIFT encoding algorithm squeezes such graph in few bits by leveraging two insights. *First*, many links in the AS graph are crossed by few prefixes, and their failure does not lead to bursts large enough to even require SWIFT fast-rerouting. SWIFT therefore does not encode those links at all. *Second*, the AS paths used by a single router at any given time tend to exhibit a limited number of AS links per position. SWIFT therefore only encodes AS links and positions that are present in the used BGP paths.

SWIFT supports rerouting policies. SWIFT complies with *rerouting policies* specified by the operators when computing backup next-hops. Indeed, rerouting to a safe path might not always be desirable in practice (*e.g.*, because economically disadvantageous). Rerouting policies express the preferences between backup next-hops or forbid the usage of specific ones (*e.g.*, to mimic business and peering agreements). For example, operators can prevent SWIFT from: *(i)* using an expensive link with a provider rather than a more convenient one with a customer; *(ii)* rerouting to a link where free traffic is close to depletion (*e.g.*, according to the 95$^{\text{th}}$ percentile rule [130]); or *(iii)* moving high volumes of traffic to geographically distant regions (*e.g.*, by sending to a remote egress point).

SWIFT supports both local and remote backup next-hops. In addition to reroute locally to a directly connected next-hop announcing an alternate route, a SWIFTED router can also fast-reroute to remote next-hops, potentially at the other side of the network, by using tunnels (*e.g.*, IP or MPLS ones). Remote backup next-hops are learned via plain iBGP sessions.

SWIFT is easy to deploy. Only a software update is required to deploy SWIFT since recent router platforms readily support a two-stage forwarding table [66]. In §4.5 we show that SWIFT can also be deployed on any existing router by interposing a SWIFT controller and an SDN switch between the SWIFTED router and its peers. The two-stage forwarding table in that case spans two devices, similarly to an SDX platform [80, 79].

4.1.3 Guarantees and limitations

We now show that SWIFT is beneficial and safe. More particularly, we prove that SWIFT rerouting strictly improves Internet-wide connectivity, proportionally to the number of SWIFTED routers, and despite not notifying path changes in the control plane. This translates into incentives for both partial and long-term Internet-scale deployment (*e.g.*, on all AS border routers).

Theorem 4.1. *The number of disrupted paths is decreased by every SWIFTED router which is on a path affected by an outage.*

Theorem 4.2. *SWIFT rerouting causes no forwarding loop, irrespective of the set of SWIFTED routers.*

Both theorems are based on the following lemma.

Lemma 4.1. *When any SWIFTED router fast-reroutes, it sends packets over paths with no blackhole and loops.*

Safety conditions for predictive fast rerouting are formalized by the following two assumptions.

Routing stability assumption. During an outage, BGP paths do not arbitrarily change during rerouting. This assumption means that *(i)* routers only change inter-domain forwarding paths that are affected by the outage; and *(ii)* only one single outage happens at a time.

If this assumption is violated, then inter-domain loops can be generated. Let s be a SWIFTED router and n the next-hop to which s fast reroutes to avoid a certain outage. If n switches path for some fast-rerouted prefixes (*e.g.*, to reflect a policy change uncorrelated with the outage), it may choose the BGP path used by s before the outage (not updated by SWIFT): this would lead to a loop between n and s.

Nevertheless, SWIFT can quickly detect and mitigate such a loop: s can monitor whether n stops offering the BGP path to which it has fast-rerouted, and select another backup next-hop.

Reasonable inference assumption. SWIFT inferences enable the SWIFTED routers to avoid paths affected by an outage. The SWIFT inference algorithm implements a conservative approach for inferring links and selecting backup paths. Still, we cannot guarantee the validity of such assumption, since SWIFT inferences are based on the partial and potentially noisy information provided by BGP (and withdrawals that reach different ASes at different times). Inferences that cause SWIFT not to rule out all paths affected by an outage might induce packet loss: in these cases, a SWIFTED router could reroute traffic to a disrupted backup, and multiple SWIFTED routers could create an inter-domain loop (if the selected backup next-hop actually uses exactly one of the disrupted paths missed by the inference). In both cases, packets will be dropped, as it would have happened for the affected prefixes without SWIFT (*i.e.*, using vanilla BGP). However, our evaluation with both real BGP traces and controlled simulations (§4.4), suggests that very few SWIFT inferences lead to the selection of disrupted backup next-hops.

Under the reasonable inference assumption, we now show that lemma Lemma 4.1 holds.

Proof of Lemma 4.1. Consider any SWIFTED router s that fast reroutes at a given time t, to avoid an inferred outage. Let n be the router to which s fast reroutes. By definition of SWIFT (and any existing fast rerouting technique that s can apply if next to a disrupted link), the following properties hold at t:

- n must offer a BGP path P_n to s, otherwise s would have not fast rerouted to n. By definition of BGP, P_n does not contain loops.

- P_n must not include any of the links affected by the outage, *i.e.*, it does not include blackholes. This is a direct consequence of the reasonable inference assumption.

- n and all routers in P_n keep forwarding packets over P_n. This is always true for routers that do not fast reroute (using SWIFT or any other fast rerouting technique), by definition of BGP. Also, since P_n does not include any link affected by the outage (see previous property), routers in P_n that can fast reroute do not receive any withdrawal for path P_n (nor any path update, since SWIFT and other fast rerouting techniques do not generate BGP messages). As a consequence, they all also maintain P_n as forwarding path.

Combining these properties together, the packets fast rerouted by s at time t are forwarded over the path $(s\ n) \cup P_n$, which does not contain loops nor blackholes – which proves the statement. ■

We now use Lemma 4.1 to prove the theorems.

Proof of Theorem 4.1. The statement follows by observing that every SWIFTED router on a disrupted path *(i)* will fast reroute, if the reasonable inference assumption holds, and *(ii)* will redirect traffic over a non-disrupted path by Lemma 4.1. ■

Proof of Theorem 4.2. Assume by contradiction that upon an outage (affecting one or more inter-domain links) a forwarding path for a certain prefix contains a loop L at a given time during the BGP convergence. Since BGP is guaranteed to compute non-loopy paths, at least one router s in L must fast reroute. However, s cannot fast reroute to a path including a loop, by Lemma 4.1.

This contradicts the hypothesis, and yields the statement. ■

4.2 SWIFT Inference Algorithm

We now detail the SWIFT outage inference algorithm, its basics along with a proof of its correctness (§4.2.1) and how it accounts for real-world factors (§4.2.3).

4.2.1 Sound inference

In the following, we consider the stream of messages received on a single BGP session since the algorithm run on a per-session basis (enabling parallelism). We also initially assume that the algorithm aims at inferring an outage produced by a *single* inter-AS failed link.

Burst detection. SWIFT monitors the received input stream of BGP messages, looking for significant increases in the frequency of withdrawals. It classifies a set of messages as the beginning of a burst when such frequency (say, number of withdrawals per 10 seconds) in the input stream is higher than the 99.99$^{\text{th}}$ percentile recorded in the recent history (*e.g.*, during the previous month).

Failure localization. When detecting a burst, SWIFT infers the corresponding failed link as the one maximizing a metric called *Fit Score (FS)*. Let t be the time at which this inference is done. For any link l, the value of FS for l is the *weighted geometric mean* of the *Withdrawal Share* (WS) and *Path Share* (PS):

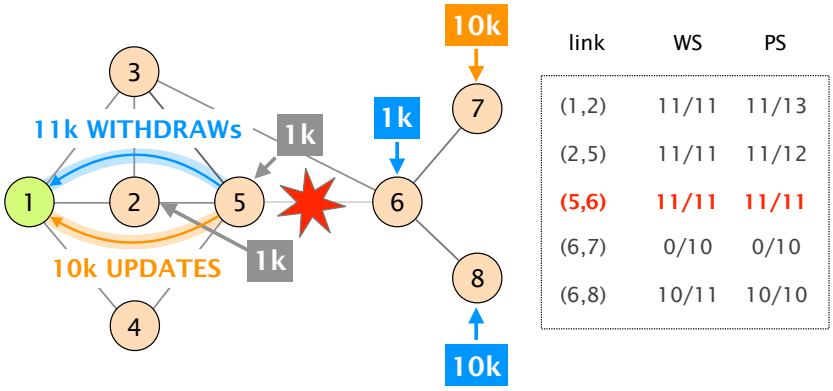
$$FS(l,t) = (WS(l,t)^{w_{WS}} * PS(l,t)^{w_{PS}})^{1/(w_{WS}+w_{PS})}$$

WS is the fraction of prefixes forwarded over l that have been withdrawn at t over all the received withdrawals. PS is the fraction of withdrawn prefixes with a path via l at t over the prefixes with a path via l at t. More precisely,

$$WS(l,t) = \frac{W(l,t)}{W(t)} \qquad PS(l,t) = \frac{W(l,t)}{W(l,t)+P(l,t)}$$

where $W(l,t)$ is the number of prefixes whose paths include l and have been withdrawn at t; $W(t)$ is the *total* number of withdrawals received as of t; $P(l,t)$ is the number of prefixes whose paths still traverse l at t. w_{WS} and w_{PS} are the weights we assign to WS and PS. By relying on WS and PS, the fit score aims at quantifying the relative probability that a link is responsible for the received withdrawals while being robust to real-world factors such as BGP noise (§4.2.3).

Example. Figure 4.2 reports the WS and PS values at the end of the burst of withdrawals generated by the failure of $(5,6)$ in Figure 3.1. Link $(5,6)$ is the only one with both WS and PS equal to 1, since all the AS paths traversing it have been either withdrawn or changed with another path not crossing $(5,6)$. In contrast, the PS values for links $(1,2)$ and $(2,5)$ are smaller than 1 (11k/13k and 11k/12k), because paths for the prefixes of AS 2 and AS 5 have not been



link	WS	PS
(1,2)	11/11	11/13
(2,5)	11/11	11/12
(5,6)	**11/11**	**11/11**
(6,7)	0/10	0/10
(6,8)	10/11	10/10

Figure 4.2 WS and PS metrics at the end of the burst of withdrawals caused by the failure of (5,6).

modified by the burst. The WS of $(6, 8)$ is smaller than 1 because not all the withdrawals pertain to that link. At the end, $(5, 6)$ is therefore correctly inferred as failed.

Correctness. SWIFT inference algorithm is always correct under ideal conditions. The following theorem holds.

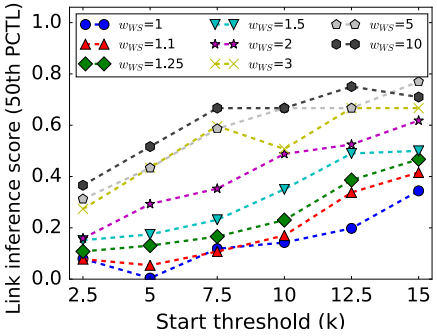
Theorem 4.3. *If all ASes inject at least one prefix on every adjacent link, SWIFT inference returns a set of links including the failed link if run at the end of the stream of BGP messages triggered by the failure.*

Proof of Theorem 4.3. Assume that a single link f fails and that the inference algorithm makes a prediction at time t when fed with all and only the BGP messages generated by f.

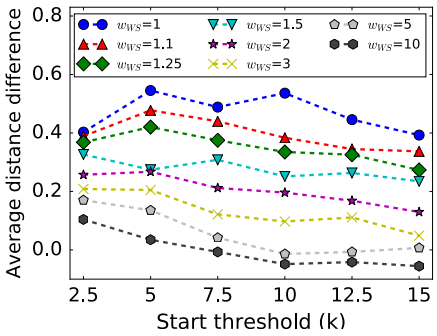
We now show that the inference algorithm assigns the highest possible values of both WS and PS to f.

Indeed, all the paths traversing f before the burst are either explicitly withdrawn or updated (to avoid f): This implies that the number $P(f, t)$ of paths traversing f at t is 0. Moreover, since only BGP messages generated by f's failure are in the inference input by hypothesis, all the received withdrawals must have crossed f, that is, $W(f, t) = W(t)$. As a consequence, $PS(f, t) = W(l, t)/(W(l, t) + 0)$ and $WS(f, t) = W(l, t)/W(l, t)$ are equal to their maximum value 1.

This implies that the fit score of f is the highest possible one, hence the SWIFT inference algorithm will return it in the set of failed links. ■



(a) Ratio of links inferred.



(b) Difference of distance from the SWIFTED router.

Figure 4.3 Behavior of the SWIFT inference algorithm as a function of when it is triggered during the burst and the ratio between the weight of WS and PS.

4.2.2 Tailored for fast inference

SWIFT makes inference *during* the bursts to be fast. However, the downside of being fast is that only partial information is available to make the inference. In this section, we show how we designed the SWIFT outage inference algorithm to be accurate with partial information.

SWIFT makes accurate inferences *during* the burst. Contrary to the assumptions of Theorem 4.3, SWIFT runs its inference algorithm at the beginning of a burst. Lack of information (*i.e.*, carried by not yet received withdrawals) can therefore affect its accuracy. Being aware of this lack of information, we adapt two parameters of SWIFT: *(i)* the *triggering threshold* which indicates the number of withdrawals after which SWIFT inference is triggered; and *(ii)* the difference between the weight assigned to WS and PS in the geometric mean computation. We pick the values for those two parameters based on observations we made on 375 bursts greater than 20k withdrawals and collected on 10 RouteViews collectors during the last two weeks of July 2016 (which is another dataset that we use to evaluate SWIFT in §4.4, so as to ensure that our results are not dataset-driven). Figure 4.3a shows the median inference score as a function of the *triggering threshold* and the weight assigned to WS (w_{WS}). The inference score is the ratio of links inferred at a given *triggering threshold* and which are the same than the ones inferred at the end of the burst.

SWIFT adapts the *triggering threshold* based on the likelihood of inferring the correct outage. Intuitively, the higher is the number of withdrawals received, the better is the inference. We can see in Figure 4.3a that with $w_{WS} \geq 3$, the median inference score is between 0.3 and 0.4 after 2.5k withdrawals, and between 0.7 and 0.8 after 15k withdrawals. These results reveal that after 2.5k withdrawals, SWIFT already localizes (sometimes

partially) many outages. As a result, we configure SWIFT to run the inference algorithm after only 2.5k withdrawals. However, we configure SWIFT to wait for more information if the number of withdrawals received is too low compared to the number prefixes that would be rerouted, as this would more likely result in an inaccurate inference. Specifically, we reject an inference that would reroute more than 10k, 20k, 50k, and 100k prefixes, if respectively inferred after 2.5k, 5k, 7.5k, and 10k withdrawals. After having received 20k withdrawals, SWIFT returns the inferred link regardless of the number of predicted prefixes.

SWIFT increases the weight of WS for better inference early on during the burst. The key intuition is that early on during the burst, a large number of prefixes are not yet withdrawn and are still using the failed link. As a result, the PS for that link may not be the highest one. The PS for the failed link actually increases when SWIFT runs the inference later in the burst. However, the WS for the failed link will always be greater or equal than the WS of any other link, provided that SWIFT does not receive unrelated withdrawals and that the outage is produced by a single link failure. SWIFT thus performs better when $w_{WS} > w_{PS}$.

The results in Figure 4.3a confirm this intuition. We can see that increasing the weight of WS (while keeping the weight of PS equal to 1) improves the inference. Figure 4.3b helps to understand this behavior by showing the difference, in the distance from the SWIFTED router, between the links inferred before and at the end of each burst. With equal weights for WS and PS, the algorithm tends to infer links further than the one inferred at the end of the burst. This is likely because at the beginning of the burst, the PS of the failed link is lower than 1, and thus further links may have a higher PS, since fewer prefixes are using them. To bring the distance difference closer to 0 (ideal case, if we assume that the inference at the end of the burst is the correct one), we can increase the weight of WS. Figure 4.3b shows that when choosing 3, 5, or 10 for the WS weight, the average distance difference is very close to 0, and can even be negative for 5 and 10. As a result, we set the ratio between the weights of WS and PS to 3.

4.2.3 Robust to real-world factors

In practice, actual streams of BGP messages do not always match the ideal conditions assumed in Theorem 4.3. In this section, we show how we designed the SWIFT inference algorithm to make it robust against real-world factors.

SWIFT quantitative metrics mitigate the effect of BGP noise. Some received BGP messages may be unrelated to the outage causing a burst but due to contingent factors (*e.g.*, misconfiguration, router bugs). They constitute noise that can negatively affect the accuracy of any inference algorithm. In SWIFT, noise can distort FS values. In Figure 4.2, for instance, withdrawals

for prefixes originated by AS5 can be received by AS1 during the depicted burst. This would increase the likelihood that the FS of $(2, 5)$ is higher than the one of $(5, 6)$, especially at the beginning of the burst.

In practice, SWIFT is robust to realistic noise as the level of BGP noise is usually much lower than a burst. Hence, its effect on quantitative metrics such FS, WS, and PS, tends to rapidly drop. This feature distinguishes our inference algorithm from simpler approaches, *e.g.*, based on AS-path intersection, which are much more sensible to single unrelated withdrawals.

SWIFT applies a conservative strategy if failed links cannot be univocally determined. It may happen that SWIFT cannot distinguish precisely which link has failed. For example, in Figure 4.2, assuming that the 1k prefixes from AS6 are updated and not withdrawn, SWIFT cannot distinguish if $(5, 6)$ or $(6, 8)$ failed. Whenever a failed link cannot be univocally determined, SWIFT inference returns all the links with maximum FS, *i.e.*, both $(5, 6)$ and $(6, 8)$ in the previous example.

SWIFT can infer concurrent link failures. To cover cases like router failures that affect multiple inter-AS links at the same time, the inference algorithm computes the FS value for sets of links sharing one endpoint. More precisely, the algorithm aggregates greedily links with a common endpoint (from links with the highest FS to those with the lowest one), until the FS for all the aggregated links does not increase anymore. The fit score FS for any set S of links is computed by extending the definition of WS and PS as follows.

$$WS(S,t) = \frac{\sum_{l \in S} W(l,t)}{W(t)} \qquad PS(S,t) = \frac{\sum_{l \in S} W(l,t)}{\sum_{l \in S} W(l,t) + P(l,t)}$$

The set of links (potentially, with a single element) with the highest FS is returned. To ensure safety (see §4.1.3), for each link inferred, SWIFT must choose a backup route that does not traverse the common endpoint of the links.

To put that into practice, SWIFT chooses a backup path for a prefix and a link in the AS path which avoids both endpoints of the link. This is required as SWIFT computes the backup next-hops in advance, *i.e.*, before the failure (see §4.1.2), and thus SWIFT does not know which endpoint of a link will be the common endpoint. This prevents SWIFT to reroute a prefix to a backup next-hop that uses another inferred link (because all the inferred links have a common endpoint). By choosing backup paths bypassing a superset of the inferred links, SWIFT also ensures safety in case the inference algorithm correctly localizes the ASes involved in the outage instead of the precise links.

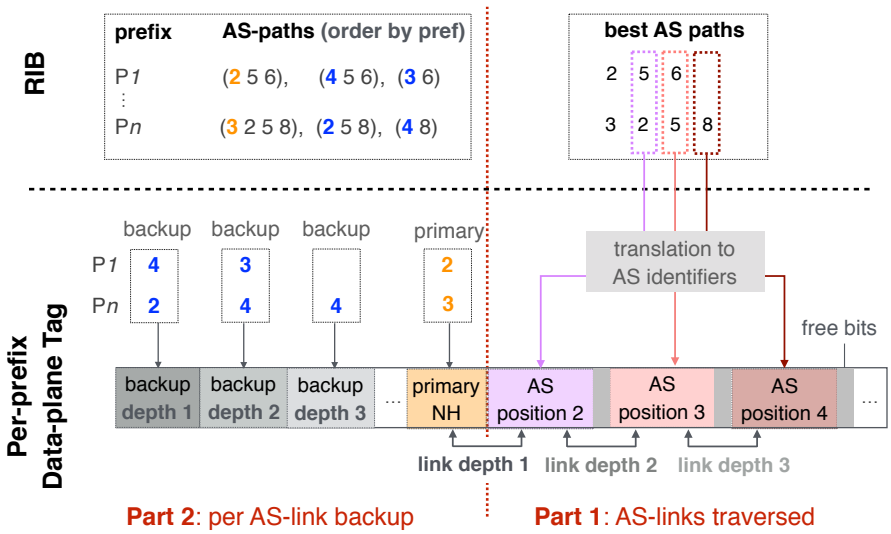


Figure 4.4 A SWIFTED router embeds a tag into incoming packets. The tag encodes the links traversed by the packet (Part 1) along with backup next-hops for each of the encoded links (Part 2).

4.3 SWIFT Encoding Algorithm

In this section, we describe how SWIFT tags are computed. Recall that these tags are embedded onto the incoming packets in the first stage of the forwarding table and are split in two parts: one which encodes the AS links used by the packet, and another which encodes the next-hops to reroute to should any of these links fail. Thanks to these embedded tags, a SWIFT router can quickly reroute traffic on working backup paths upon an inference, independently on the number of prefixes affected.

In Section 4.5, and similarly to [79, 80], we show how SWIFT can leverage the destination MAC to tag incoming traffic. The destination MAC is indeed a good "tag carrier" as it provides a significant number of bits (48), and can easily be removed in the second stage of the forwarding table by rewriting it to the MAC address of the actual next-hop, as any IP router would do.

Encoding AS links. The first part of the tag (right side of Figure 4.4) encodes the AS path along which each packet will flow. For each prefix, we consider the AS path associated with the best route for it, and we store the position of ASes in that path. Namely, we define m sets, with m being the length of the longest AS path, and we call the i-th set *position* i. For any AS path $(u_0\ u_1 \dots u_k)$, with $k \leq m$, we then add the AS identifier of u_i to position i, for every $i = 1, \dots, k$. Note that we do not model position 1 because the first hop in any AS path is already represented as the primary next hop (see Part 2 of Figure 4.4). This AS-path encoding is computed for every SWIFT's neighbor and allows AS paths to be encoded using AS identifiers for every position.

Encoding all used AS paths may not be possible. Not only can thousands of distinct ASes be seen for each position, but also the AS paths may be very long (>10 hops). Fortunately, two observations enable SWIFT to considerably reduce the required number of bits. *First*, from the perspective of one router, many AS links carry few prefixes. A failure of these links will therefore produce small bursts (if any), which allows for per-prefix update. Thus, we ignore any link that carries less than 1,500 prefixes in our SWIFT encoding. *Second*, links that are far away from the SWIFTED node are less likely to produce bursts of withdrawals than closer ones. Indeed, for distant links, it is more likely that intermediate nodes know a backup path. Our measurements (§4.4) confirm this. Consequently, SWIFT only encodes the first few hops of the AS paths (up to position 5).

For the remaining AS links, SWIFT encodes first the links with the highest number of prefixes traversing them. To do that, SWIFT uses an adaptive number of bits for each AS position: each position is implemented by a different bit group, whose length depends on the number of ASes in this position. For each position P, we map all the ASes in P to a specific value (the AS identifier) of the corresponding bit group. Hence, the size of this group is equal to the number of bits needed to represent all the values in P.

Encoding backup next-hops. The second part of the tag (left side of Fig. 4.4) identifies the primary next-hop as well as backup next-hops for each encoded AS link. For each prefix p, the primary next-hop is directly extracted as the first hop in the AS path for p. For instance, the primary next-hop for prefix $p1$ in Figure 4.2 is 2. Backup next-hops are explicitly represented to both reflect rerouting policies and prevent rerouting to disrupted backup paths. Consider again $p1$. The primary path is $(2, 5, 6)$. To protect against a failure of the first AS link $(2, 5)$, we can select `AS3` or `AS4`, since neither of the two uses $(2, 5)$ to reach $p1$. In contrast, for $(5, 6)$, only `AS3` can be used as a backup next-hop, since the AS paths received from `AS4` also uses $(5, 6)$.

Partitioning bits across the two parts of the tag. A fundamental tradeoff exists between the amount of paths and the number of backup next-hops that any SWIFT router can encode. On the one hand, allocating more bits to represent AS links (first part of the tag) allows a SWIFTED router to cover more remote failures. On the other hand, allocating more bits to represent (backup) next-hops (second part of the tag) allows a SWIFTED router to reroute traffic to a higher number of backup paths.

In §4.4.4, we show that allocating 18 bits to AS paths encoding is sufficient to reroute more than 98% of the prefixes. Assuming 48-bits tags (*i.e.*,, using the destination MAC), 30 bits are left to encode backup next-hops. If we configure SWIFT to support remote failures up to depth 4, the bits allocated for the backup next-hops needs to be divided by 5 (1 primary + 4 backup next-hops). As a result, $30/5 = 6$ bits are reserved for each depth, which translates into

$2^6 = 64$ possible next-hops. If one wants to consider remote failures only up to depth 3, then the number of next-hops is $2^7 = 128$ and two more bits can be allocated to the AS links encoding. Operators can fine-tune such decision, *e.g.*, based on the (expected) number of backup next-hops reachable by each SWIFTED router.

4.4 Evaluation

We now evaluate our Python-based implementation ($\approx$ 3,000 lines of code) of the SWIFT inference algorithm (§4.2) and the encoding scheme (§4.3). We first describe our datasets (§4.4.1). We then evaluate the accuracy of the inference algorithm, both in terms of failure localization (§4.4.2) and withdrawals prediction (§4.4.3).

We also evaluate the efficiency of SWIFT data-plane encoding (§4.4.4). Finally, we show that the combination of the inference algorithm and the encoding scheme leads to much faster convergence than BGP (§4.4.5).

4.4.1 Datasets

We evaluate SWIFT using two sources of bursts of BGP withdrawals.

Bursts from real BGP data, without outage ground truth. To evaluate how SWIFT would work in the wild, we use sets of actual bursts extracted from the same dataset used in §2.2.3. It consists of BGP messages dumped by 10 RouteViews [132] and 5 RIPE RIS [16] collectors during the full month of November 2016. These collectors received BGP messages from 213 peers. Note that, we found 5 routers peering with these collectors that exhibit a flapping behavior, with an anomalous large number of bursts of similar pattern; when including them, we obtain a minimal change in overall results ($\approx$2%), but since SWIFT performs uniformly on similar bursts, their large number ($\approx$500 bursts) causes a significant skew in the population of bursts. We therefore omit these peers from our analysis.

Our evaluation is based on 1,802 bursts with more than 1,500 withdrawals. Amongst them, 942 (resp. 339) have more than 2,500 (resp. 15,000) withdrawals.

Bursts from simulations, with outage ground truth. To validate the accuracy and the robustness of our inference algorithm, we use bursts extracted from control-plane simulations conducted with C-BGP [145]. We created a topology composed of 1,000 ASes using the Hyperbolic Graph Generator [23]. We set the average node degree to 8.4, which is the value observed in the CAIDA AS-level topology [14] in October 2016, and use as degree distribution

a power law with exponent 2.1 [105]. We defined the AS relationships as follows. The three ASes with highest degree are Tier1 ASes and are fully-meshed. ASes directly connected to a Tier1 are Tier2s. ASes directly connected to a Tier2 but not to a Tier1 are Tier3s, etc. Two connected ASes have a peer-to-peer relationship if they are on the same level, otherwise they have a customer-provider relationship. We configured each AS to originate 20 prefixes, for a total of 20k prefixes. Using C-BGP, we simulated random link failures, and recorded the BGP messages seen on each BGP session in the network. We collected a total of 2,183 bursts of at least 1k withdrawals. The median (resp. max) size of the bursts is 2,184 (resp. 19,215) withdrawals.

4.4.2 Failure localization accuracy

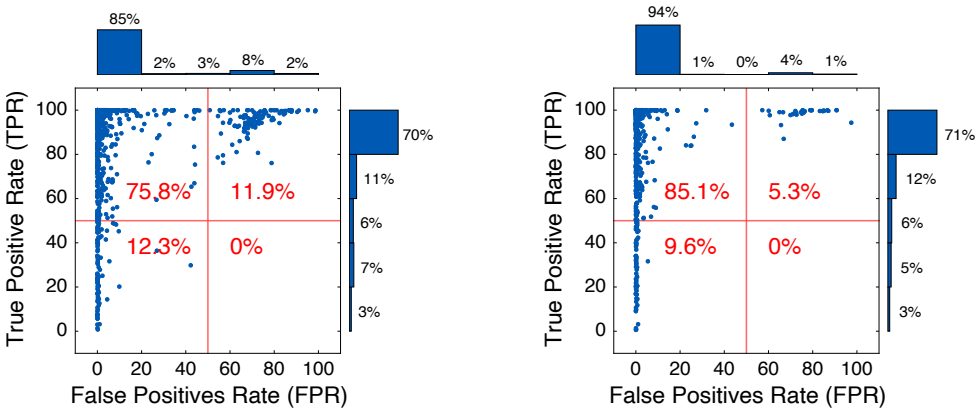
In the following, we evaluate the accuracy of the SWIFT inference algorithm on both datasets.

4.4.2.1 Validation on real BGP data

Since real BGP traces do not provide the ground truth on burst root causes, we estimate the accuracy of the inference algorithm indirectly: we evaluate the match between the prefixes withdrawn in the entire burst W and the prefixes W' whose path traversed the links inferred by SWIFT as failed. This can be formalized as a binary classification problem, in which the true and false positives are the prefixes in $W' \cap W$ and $W' - W$, respectively. We therefore evaluate the accuracy of SWIFT inference in terms of True Positive Rate (TPR) and False Positive Rate (FPR). More precisely, we have $TPR = TP/(TP + FN)$, $FPR = FP/(FP + TN)$. The negatives are all the prefixes announced in the session before the burst starts and not withdrawn during the burst.

Figure 4.5 shows the TPR and FPR on a per-burst basis. It is divided into quadrants. The top left quadrant corresponds to very good inferences, *i.e.*, for each burst, the links that SWIFT infers as failed are traversed by most of the withdrawn prefixes (high TPR) and few of the non-affected prefixes (low FPR). The top right quadrant contains inferences that overestimate the extent of a failure (high TPR and FPR): rerouting upon such inferences is still beneficial as the TPR is high (*i.e.*, connectivity is restored for many prefixes actually disrupted). The bottom left quadrant corresponds to inferences that underestimate the extent of a burst. Finally, the bottom right quadrant includes *bad* inferences (with low TPR and high FPR).

We evaluate two scenarios for SWIFT. In the first one (Figure 4.5a), the inference algorithm runs only once, after 2.5k withdrawals—without adapting



(a) Static *triggering threshold*.

(b) Adaptive *triggering threshold*.

Figure 4.5 Despite having little information, SWIFT inference is accurate. The vast majority of prefixes are correctly inferred as failed (top half quadrants). While some affected prefixes are missed (bottom left), no prediction is significantly inaccurate (bottom right).

the *triggering threshold*. In the second scenario (Fig. 4.5b), the inference algorithm runs every 2.5k withdrawals while following the simple model we described in §4.2.3 to decide when to reroute traffic. In this case, SWIFT waits for more withdrawals to arrive before rerouting large numbers of prefixes early on in the burst.

SWIFT makes accurate inferences in the majority of the cases, and never makes *bad* inferences. Even when using only 2.5k withdrawals (Figure 4.5a), SWIFT makes accurate inferences in the vast majority of the cases: TPR is more than 60% for more than 81% of the bursts. However, it also overestimates the extent of the burst (FPR is higher than 50%) for about 12% of the bursts. SWIFT inference algorithm performs sensibly better with an adaptive *triggering threshold* (Figure 4.5b). Better performance comes at the price of missing some bursts because of the extra delay. Specifically, it missed a total of 256 bursts (53% of them smaller than 5k) compared to the static *triggering threshold* version. Despite this, the adaptive *triggering threshold* version of the inference algorithm still completes the inference at the lowest threshold (2.5k) for the majority of the bursts (65%). The increased density of the top left quadrant in Figure 4.5b is a clear indication of the gain obtained by trading a bit of speed for better accuracy. Finally, we stress that SWIFT *never* falls into the bottom right quadrant, irrespective of whether an adaptive *triggering threshold* is used or not.

4.4.2.2 Validation through simulation

We now describe the results obtained by SWIFT inference algorithm when run on the bursts generated in C-BGP (see §4.4.1).

Under ideal conditions, SWIFT inference is *always* correct. We ran our inference algorithm at the end of each burst and found that the inference is always correct, consistently with Theorem 4.3.

SWIFT inference is accurate enough to ensure safety, even early on during the bursts. When we ran the inference algorithm after only 200 withdrawals (1% of the total number of prefixes advertised, see §4.4.1), SWIFT identified a superset of the failed link for 9% of the bursts. For the remaining 91%, it returned a set of links adjacent to the failed one. Nevertheless, for *all the 2,183 bursts but one*, SWIFT selected a backup path that bypasses the actual failed link. This is because SWIFT chooses a backup route that does not traverse the common endpoint of the inferred links (see §4.2.3).

SWIFT inference is robust to noise. We simulated BGP noise by adding, in each burst, 1,000 withdrawals of prefixes that are not affected by the failure. This number is much greater than what we observe in real BGP data, both in absolute terms (9 withdrawals only during 10 second periods in the 90th percentile, see §2.2.3) and as a percentage (since we only advertise 20k prefixes in C-BGP, whereas there are more than 865k prefixes advertised in the real world [1]). When we triggered the inference at the end of the burst, SWIFT identified the failed link for 91% of the bursts (1991), a superset for 9% bursts (188), a set of links adjacent to the failed one for 1 burst and did a wrong inference for 3 bursts. When we triggered the inference after 200 withdrawals, SWIFT still selected backup paths that bypass the actual failed link for *all the bursts but one*. More precisely, SWIFT identified a superset of the failed link for 12% of the bursts, while for the remaining 88%, it returned a set of links adjacent to the failed one.

4.4.3 Withdrawals prediction accuracy

In the previous section (§4.4.2), we showed that SWIFT inference algorithm is indeed able to identify the failed link, even with limited information. In this section, we evaluate the ability of SWIFT to predict withdrawals and we also give the absolute number of prefixes fast rerouted upon such inference, enabling us to quantify the benefit of SWIFT as well as the possible under/overshooting induced.

	percentile of bursts						
	10th	20th	30th	50th	70th	80th	90th
Burst size between 2.5k and 15k							
CPR	24.6%	48.9%	72.6%	89.5%	98.5%	99.7%	99.9%
FPR	0%	0.03%	0.07%	0.22%	0.50%	0.81%	1.8%
CP	47	178	349	901	2.1k	3.0k	4.3k
FP	24	125	301	802	2.2k	3.1k	5.0k
Burst size greater than 15k							
CPR	5.6%	39.3%	80.4%	93.0%	98.1%	99.7%	99.9%
FPR	0%	0%	0.04%	0.60%	5.42%	13.9%	74.9%
CP	1.7k	5.7k	11.0k	19.6k	53.2k	78.1k	193k
FP	0	6	110	2.4k	19.8k	50k	402k

Table 4.1 Inference algorithm with history model: performance of the prediction of future withdrawals.

Differently from the previous section, in order to evaluate specifically the *prediction*, we consider as "positives" only the prefixes withdrawn *after* the inference was made. This change affects the definition of TP (and TPR) but leaves FP (FPR) unaltered. Since we already used TPR in §4.4.2, we denote with CPR (for Correctly Predicted Rate) the true positive rate of the prediction. We also denote with CP and FP, the total numbers of prefixes correctly predicted or not, respectively.

4.4.3.1 Validation on real BGP data

Table 4.1 shows results obtained by running the SWIFT inference algorithm with the history model. Results for small ($\leq$15k) and large ($>$15k) bursts are shown separately.

SWIFT correctly fast-reroutes a large number of affected prefixes. For half (resp. 80%) of the small bursts, SWIFT correctly predicts at least 89.5% (resp. 48.9%) of the future prefix withdrawals. For half (resp. 80%) of the large bursts, SWIFT correctly predicts at least 93% (resp. 39.3%) of the future prefix withdrawals. In terms of absolute numbers, SWIFT correctly fast-reroutes a significant amount of prefixes, especially for larger ($>$15k) bursts, where the number of prefixes predicted is in the order of tens of thousands for 60% of the bursts and in the order of hundreds of thousands for more than 10%.

SWIFT only reroutes a small number of non-affected prefixes. Both for small and large bursts, the fraction of fast-rerouted prefixes that were not affected by the failure is small in most of the cases. In few cases (*e.g.*, 90-th percentile of the large bursts) however, the algorithm significantly overestimates the number of prefixes to be rerouted (FP). This is because we deliberately designed and tuned the algorithm to not minimize incorrectly rerouted prefixes in order to avoid missing prefixes that should be rerouted and have significant gain in network uptime. Incorrectly rerouted prefixes are indeed forwarded to a backup path which is sub-optimal but not disrupted, just for the few minutes needed for BGP to reconverge. This is in line with the key requirements we list in §3.4. Consistently, we note that less aggressive weights do reduce the FPR (see §4.2.2).

4.4.3.2 Validation through simulation

We now evaluate the accuracy of the prefixes prediction on the bursts generated by C-BGP.

SWIFT accurately predicts prefix withdrawals, even when considering noise. When inferring the affected prefixes after only 200 withdrawals, the FPR is equal to 0% for 98% of the bursts. The highest FPR observed is only 13%. In the median case (resp. 25th percentile), the CPR is equal to 88% (resp. 84%). The lowest CPR observed is 37%.

To consider the impact of BGP noise on these numbers, we added to each burst 1,000 withdrawals unrelated to the failure (as in §4.4.2.2). We found that, for 53% of the bursts, the FPR is still 0%. The FPR is greater than 9% for only 1% of the bursts. In the median case (resp. 25th percentile), the CPR is 53% (resp. 50%). The CPR is far from 100% because the withdrawals unrelated to the failure count as positives. In practice, we observe that the CPR is less affected by BGP noise, as the level of noise is usually much lower (see §2.2.3).

4.4.4 Encoding effectiveness

We now experimentally evaluate SWIFT encoding scheme (§4.3) by quantifying how many prefixes can effectively be rerouted in the data plane by matching on the pre-provisioned tags. For each burst, we define the *encoding performance*, as the fraction of predicted prefixes that can be rerouted by the encoding scheme. The performance depends on the number of bits allocated to the AS path part of the tag (see §4.3). For this part of the evaluation, we rely on the inference algorithm with the adaptive *triggering threshold* and consider the bursts obtained from the real BGP data.

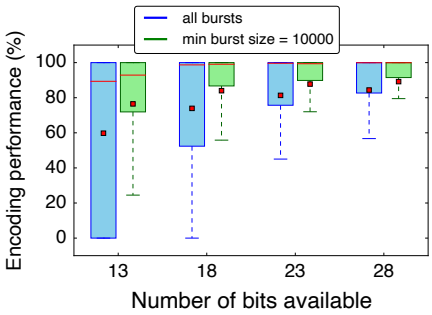


Figure 4.6 With only 18 bits available for the AS paths encoding, SWIFT can reroute more than 98.7% of the predicted prefixes in the median case.

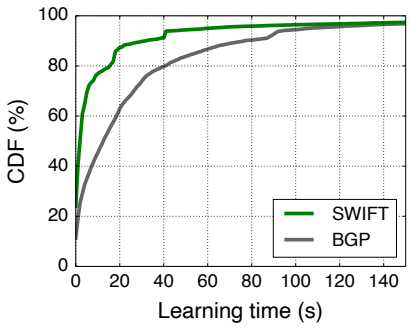


Figure 4.7 SWIFT quickly learns about remote outages. In 2 (resp. 9) seconds, SWIFT learns more than 50% (resp. 75%) of the withdrawals, BGP needs 13 seconds (resp. 32 seconds).

Allocating 18 bits to the AS-path part of the tag enables to reroute 98.7% of the predicted prefixes.. Figure 4.6 shows the encoding performance (over all bursts) as a function of the number of bits reserved for the AS-path part of the tag. Each box shows the inter-quartile range of the encoding performance: the line in the box depicts the median value; the dot depicts the mean; and the whiskers show the 5th and 95th percentiles. As the number of bits allocated to the AS paths encoded increases, so does the encoding performance. We see that 18 bits are already sufficient to reroute 98.7% of the predicted prefixes in the median case (73.9% in average). These results illustrate that the compression done by the encoding algorithm is efficient and manages to encode the vast majority of the relevant AS links. In addition, Figure 4.6 shows that for the large bursts of at least 10k withdrawals, the encoding performance is better (84.0% on average with 18 bits). This is explained by the design of our encoding algorithm, which encodes with highest priority the AS links with the largest number of prefixes traversing them (and which may cause large bursts in case of a failure).

Assuming a tag of 48 bits (*e.g.*, using the destination MAC), the remaining 30 bits can be used to encode the backup next-hops. If SWIFT encodes up to depth 4 (*i.e.*, position 5 in the AS path), 64 different next-hops can therefore be used. This suggests that SWIFT encoding can work well even if the SWIFT device is connected to a large number of external neighbors, like in IXPs §4.5.2. The number of backup next-hops can even be increased by reducing the number of AS hops encoded (*e.g.*, up to depth 3 instead of 4).

4.4.5 Rerouting speed

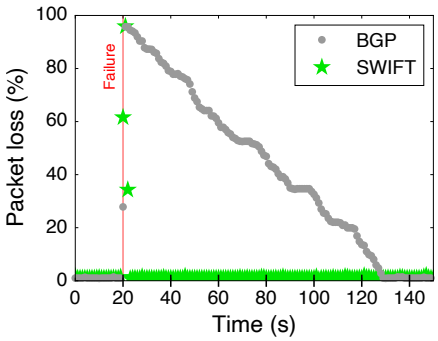
In this section, we show that the combination of the SWIFT inference algorithm and the encoding scheme enables fast convergence in practice (within 2 s) by quantifying: *(i)* the learning time required for a prediction; and *(ii)* the number of rules updates to perform in the data plane. Our results are computed on the bursts in the real BGP data.

SWIFT learns enough information to converge within 2 seconds (median). Compared to vanilla BGP, SWIFT converges much faster than a BGP router working at the per-prefix level. Figure 4.7 shows the CDF of the time elapsed between the beginning of the burst and the actual time at which every withdrawal in the burst is learned. For BGP, the learning time corresponds to the withdrawal timestamp. For SWIFT, it corresponds to the prediction time if the withdrawal is predicted, otherwise the withdrawal timestamp. The plot highlights that, in the median case, SWIFT learns a withdrawal within 2 s, while BGP needs 13 s. We can observe a shift at 41 s in the SWIFT curve. After investigation, we found that this is due to a very large burst of 570k withdrawals which took a total of 105 s to arrive. The first 20k withdrawals (needed for SWIFT to launch the prediction) took 41 s to arrive. Observe that, even in such a case, SWIFT was still able to shave off more than 1 min of potential downtime.

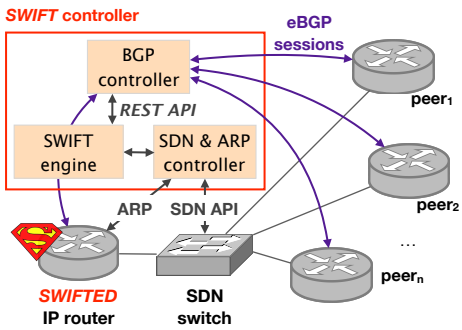
SWIFT requires few data-plane updates to reroute all the predicted prefixes. The number of data-plane updates required to reroute all the predicted prefixes depends on the number of failed AS links reported by the inference algorithm. When executing the inference algorithm after 2.5k withdrawals, in 29% of the cases, the number of links predicted is 1 and the median number (resp. 90th percentile) is 4 (resp. 29). For each reported link, one data-plane update is required for each backup next-hop (§4.3). As a result, in the median case (resp. 90-th percentile) and with 16 backup next-hops, 64 (resp. 464) data-plane updates are required. Considering a median update time per-prefix between 128 and 282 μs [173, 64], SWIFT can update all the forwarding entries within 130 ms.

4.5 Case Study

In this section, we showcase the benefits of SWIFT when deployed in two different scenarios. In §4.5.1 we deploy SWIFT on recent Cisco router to boost it convergence time. In §4.5.2, we then deploy SWIFT in a Software-defined Internet Exchange Points (SDX) [80, 79] to boost the convergence time of its participants.



(a) A SWIFTED Cisco router converge *98% faster*.



(b) Alternative SWIFTED implementation on existing router.

Figure 4.8 While a recent router takes 110 seconds to converge upon a large remote outage (left), the corresponding SWIFTED router (using the alternative deployment scheme depicted on the right) converges within 2 seconds.

4.5.1 SWIFT on a recent Cisco router

As mentioned in §4.1.2, SWIFT can be implemented directly on existing routers via a simple software update, since the only hardware requirement, a two-stage forwarding table, is readily available in recent platforms [66] (we confirmed this implementation through discussion with a major router vendor). Yet, to evaluate SWIFT without waiting for vendors to implement it, we developed an alternative deployment scheme that we describe in Figure 4.8b.

How to SWIFT any existing router. In our alternative deployment scheme, we interpose a SWIFT controller and an SDN switch between the SWIFTED router and its peers, respectively at the control- and data-plane level (as in §2.3.2). The setup is akin to the SDX platform [80, 79]. It enables to deploy SWIFT on *any* router that supports BGP and ARP, that is, virtually any router.

Upon reception of the BGP updates coming from the peers of the SWIFTED router, the controller assigns 48-bit tags according to the SWIFT encoding scheme (see §4.3). The controller programs the SWIFTED router to embed the data-plane tags in the destination MAC field in the header of incoming packets, using the same technique as in a SDX [80] (*i.e.*, with BGP and ARP). It also programs the SDN switch to route the traffic based on the tags, and rewrites the destination MAC address with the one of the actual next-hop. The two-stage forwarding table used by SWIFT then spans two devices: the SWIFTED router (first stage) and the SDN switch (second stage).

Upon the detection of a burst coming from a peer, the SWIFT controller runs the inference algorithm (§4.2), and provisions data-plane rules to the SDN switch for rerouting the traffic. Our SWIFT controller uses ExaBGP [3] to maintain BGP sessions.

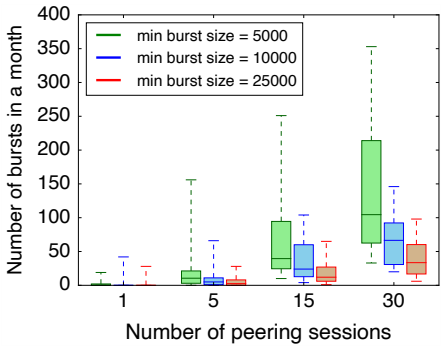


Figure 4.9 An SDX needs to handle large bursts of BGP withdrawals on a daily basis.

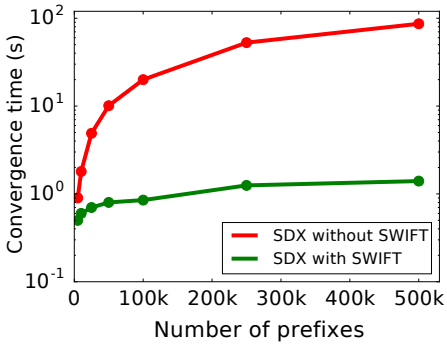


Figure 4.10 SWIFT greatly speeds up the BGP convergence in SDXes.

Methodology. We reproduced the topology in Figure 3.1a with a recent router (Cisco Nexus 7k C7018, running NX-OS v6.2) acting as `AS1`, which we connected to its peers via a laptop running a (software-based) OpenFlow switch (OpenVSwitch 2.1.3). We configured `AS6` to announce 290k prefixes. Then, we failed the link $(5, 6)$, and we measured the downtime using the same technique as in §2.3.1 (sending traffic to 100 randomly selected IP addresses).

A 98% speed-up. Figure 4.8a reports the downtime observed by the SWIFTED and non-SWIFTED Cisco router. While the vanilla Cisco router takes 109 s to converge, the SWIFTED Cisco router systematically converges within 2 s—a 98% speed-up.

4.5.2 SWIFT in an SDX

We now show that the deployment of SWIFT is not limited to single routers and that its architecture naturally fits in SDXes as well.

Minimal changes, maximal impact. An SDX controller essentially acts as BGP Route Server (RS) receiving, processing and propagating BGP messages between potentially hundreds of participants. Few other locations besides Internet eXchange Points interconnect so many networks in a single point. As large outages occur in the Internet, these RSes can be hit with huge bursts of BGP route updates that they have to process. SDXes are thus a good place to deploy SWIFT: with just a simple software update in the RS, all the SDX participants can benefit from protection against remote outages without requiring any changes on their side.

To demonstrate the benefit of deploying SWIFT in a SDX, we show the number of bursts a RS would see depending on the number of participants. Figure 4.9 shows the number of bursts of BGP withdrawals detected in November 2016 for groups of BGP sessions of different sizes and computed randomly from the

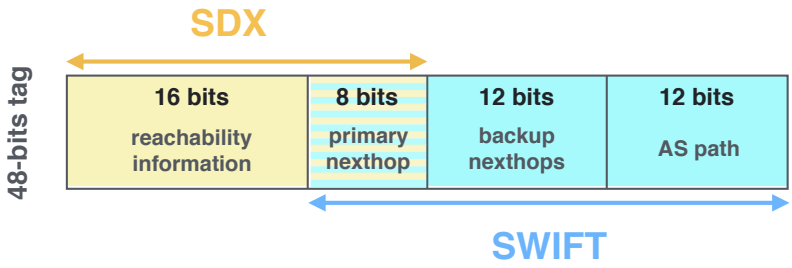


Figure 4.11 Embedding SWIFT within the SDX platform requires the two to share the 48-bits tag. Yet, we found this does not significantly impact their performance.

pool of 213 BGP sessions we used in §4.4.1. In the median case and with 30 peering sessions, we detected, in a month, 104 (resp. 33) bursts of at least 5k (resp. 25k) withdrawals.

Integrating SWIFT and SDX. SWIFT is a natural fit for the SDX as they both rely on a 2-stage forwarding table. For both projects, the first stage groups packets using a tag; the second stage then forwards packets according to their tag's values. The main difference is how SWIFT and SDX group packets. SWIFT groups packets based on the common Internet resources they are using (e.g. AS path) whereas SDX groups packets based on their forwarding equivalence class (which depends on the SDX policies). Both systems also rely on the 48-bits destination MAC address as a tag which is provisioned using BGP and ARP (as in §4.5.1).

As both the SDX and SWIFT use the destination MAC as data-plane tag, integrating the two projects require them to share this tag space. While this means that each project has fewer bits to work with, we show that we can still maintain SWIFT convergence properties and the ability of SDX to express many policies.

Sharing the 48-bits tag between SDX and SWIFT. As both SWIFT and SDX rely on the same data plane tag, the partitioning of the 48-bits between them has an impact on the performance of SWIFT and determines the number of participants the SDX may support. The operator is free to partition the tag according to her requirements. Figure 4.11 shows how we partition the 48-bits between SWIFT and SDX for this case study. 16 bits are reserved for the SDX, and 24 for SWIFT. 8 bits are used to encode the primary next-hop, which is used by both the SDX and SWIFT. The current partitioning supports up to 256 participants, which is sufficient for most IXPs. We configured SWIFT to only reroute traffic for outages happening in the first four ASes on the AS path, own AS excluded (*i.e.*, on the first three remote AS links). With this configuration, SWIFT computes three backup next-hops for each prefix, one for each of the first three remote AS links on the AS path. Hence, $12/3 = 4$ bits are available to encode each backup next-hop, which makes a total of $2^4 = 16$ possible backup next-hops for each participant.

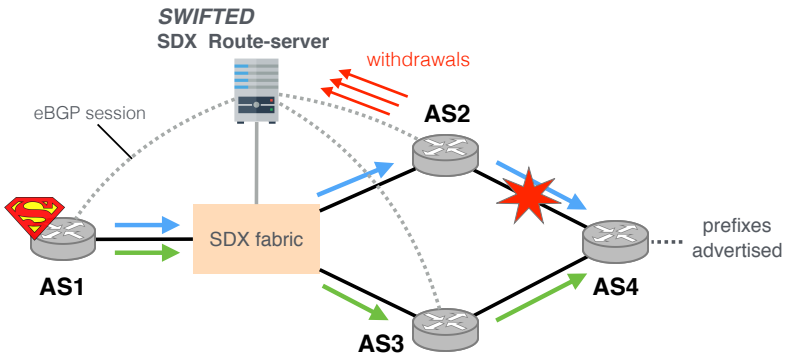


Figure 4.12 Our Quagga-based demonstration in which we showcase SWIFT running in the SDX controller. Blue (resp. green) arrows indicate the path used before (resp. after) we simulate a failure on the link $(2, 4)$.

Setup and experiment. We built the network depicted in Figure 4.12 with Mininet [114] and used Quagga for the routing software [7]. Each router is in a different AS, and AS1, AS2 and AS3 are connected to the SDX. We modified the iSDX [79] implementation to support SWIFT. We configured AS4 to advertise 5k, 10k, 25k, 50k, 100k, 250k and 500k prefixes, and made sure the primary path between AS1 and AS4 traverses AS2 and the backup path traverses AS3.

We then simulated a link failure on the link between AS2 and AS4, and measured the time AS1 takes to reroute the traffic for all the prefixes on the backup path. The failure generated a burst of BGP withdrawals that SWIFT processed in the SDX controller to trigger the fast reroute. We repeated this experiment with different number of prefixes.

Results. Figure 4.10 shows the convergence time of AS1. The convergence time without SWIFT increases linearly with the number of prefixes, and can be close to 90 second for 500k prefixes. When SWIFT is deployed, the convergence time is nearly constant and always within 1.4 second, as only few data plane rule updates are required to converge, irrespective of the number of prefixes affected by the outage. In practice, we expect the difference to be even higher as bursts take time to arrive and hardware-based routers update their forwarding table more slowly than software-based router §2.3.1.

5

Blink: Fast Connectivity Recovery Entirely in the Data Plane

In this chapter, we present Blink, a data-driven system that uses programmable data planes (see §1.2.5) to detect failures directly in the data plane by leveraging TCP-induced signals. By looking at data-plane signals, Blink is inherently faster than control-plane-driven solutions such as SWIFT.

Indeed, the fundamental problem with control-plane-driven solutions is that it can take $O(minutes)$ for the *first* BGP update to propagate after the corresponding data-plane failure. We illustrate this problem through a case study, by measuring the time the *first* BGP updates took to propagate after the Time Warner Cable (TWC) networks were affected by an outage on August 27 2014 [12]. We consider as outage time t_0, the time at which traffic originated by TWC ASes and observed at a large darknet [10] suddenly dropped to zero. We then collect, for each of the routers peering with RouteViews [132] and RIPE RIS [16], the timestamp t_1 of the first BGP withdrawal they received from the same TWC ASes. Figure 5.1 depicts the CDFs of $(t_1 - t_0)$ over all the BGP peers (100+ routers, in most cases) that received withdrawals for 7 TWC ASes: more than half of the peers took *more than a minute* to receive the first update (continuous lines). In addition, the CDFs of the time difference between the outage and the *last* prefix withdrawal for each AS, show that BGP convergence can be as slow as several minutes (dashed lines).

In short, a fundamental question is still open: *Is it possible to build a fast-reroute framework for ISPs that can converge in $O(seconds)$ for both local and remote failures?*

Blink: *fast, data-driven* convergence upon remote failures. We answer this question affirmatively by developing Blink, a *data-driven* fast-reroute framework built on top of programmable data planes. The key intuition behind

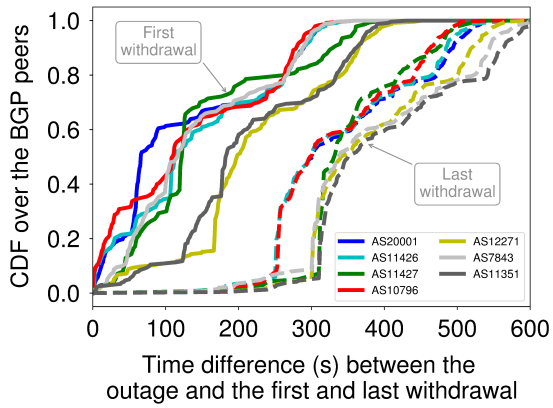


Figure 5.1 It can take minutes to receive the *first* BGP update following data-plane failures during which traffic is lost.

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

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

5.1 Key Principles and Challenges

In this section, we first show that TCP traffic exhibits a characteristic pattern upon failures (§5.1.1). We then discuss the key challenges and requirements to detect such a pattern, and recover connectivity by rerouting the affected prefixes, while operating entirely in the data plane, at line rate (§5.1.2).

5.1.1 Data-plane signals upon failures

Consider an Internet path (A, B, C, D) carrying tens of thousands of TCP flows, destined to thousand prefixes, in which the link (B, C) suddenly fails. We are interested in monitoring the data-plane "failure signal" perceived at A, with the goal of enabling A to detect it and to also recover connectivity by rerouting traffic through a different path (if any). Observe that A is not adjacent to the failure, *i.e.*, the failure is remote.

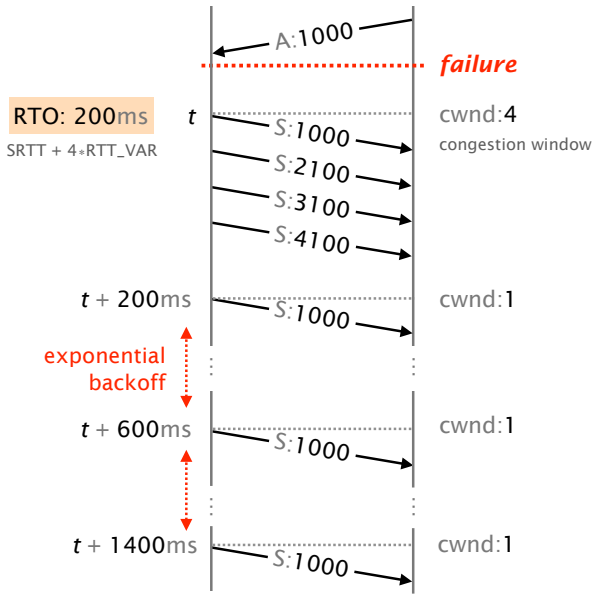
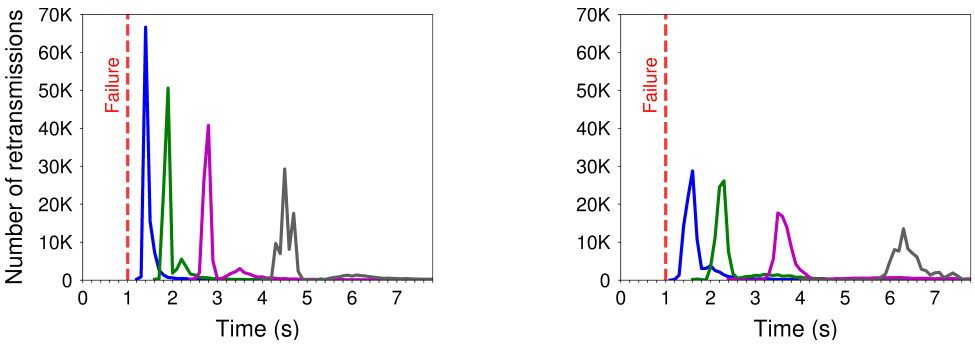


Figure 5.2 After the failure, a TCP flow keeps retransmitting the last unacknowledged segment according to an exponential backoff. The exact timing of the retransmissions depends on the estimated RTO before the failure (here 200ms).

As the link (B, C) fails, the TCP endpoints stop receiving acknowledgements (ACKs), and each of them will timeout after its retransmission timeout (RTO) expires, which will cause it to reset its congestion window to one segment and start retransmitting the first unacknowledged segment. Since the RTO is computed according to the RTT observed, each TCP endpoint will retransmit at a different time. Specifically, each TCP endpoint adjusts its RTO using the following relation: RTO $=$ sRTT $+$ 4*RTTVAR (see [149]), where sRTT corresponds to the smoothed RTT, and RTTVAR corresponds to the RTT variation. After each retransmission, each TCP endpoint further doubles its RTO (exponential backoff).

We illustrate the behavior of a TCP flow experiencing a failure in Figure 5.2. We assume that the TCP endpoint has an estimated RTO of 200 ms and that its congestion window can hold 4 packets. We denote by t the time at which the TCP endpoint transmits the first packet following the failure. The TCP endpoint experiences consecutive RTO expirations and retransmits the packet with sequence number 1000 at time $t + 200ms$, $t + 600ms$, $t + 1400ms$, etc. We experimentally verified that this behavior is similar across all TCP flavors implemented in the latest Linux kernel.

When multiple flows experience the same failure, the signal obtained by counting the overall retransmissions consists of "retransmission waves". Since this behavior is systematic, pronounced, and quick, we leverage it in Blink to perform failure detection in the data plane. This suggests Blink does not depend on specific TCP implementation details and would keep working effectively with



(a) Failure Signal (RTT 1×) **(b)** Failure Signal (RTT 1.5×)

Figure 5.3 The signal generated by TCP flows experiencing a connectivity problem is characteristic and composed of subsequent waves of retransmissions (in different colors). The waves have decreasing amplitude and increasing width.

future congestion control algorithms as long as they exhibit a similar behavior upon failures.

Note however the shape of these retransmission waves, *i.e.*, their amplitude and width, depends on the distribution of the estimated RTTs. As an illustration, Figure 5.3 shows the retransmission count for a trace that we generated with the ns-3 simulator [17] after simulating a link failure (according to the methodology in §5.5.1). In the left diagram, we used the distribution of the average RTTs of the TCP flows from an actual traffic trace (#8 in Table 5.1). In the right diagram, we increased the RTTs of this distribution by 1.5 to obtain a larger standard deviation.

We can clearly see the waves of retransmissions appearing within a second after each failure. RTT distributions with small variance make the flows more synchronized they will be when retransmitting. This translates into narrow peaks of retransmissions with a high amplitude. Conversely, if the flows have very different RTTs (*i.e.*, the variance is high), the peaks will have a smaller amplitude and will spread over a longer time. We elaborate on the challenges deriving from these observations hereafter.

5.1.2 Key challenges and requirements when fast rerouting using data-plane signals

We now highlight four key challenges and requirements that must be addressed to: (i) efficiently capture the failure signal we just described; and (ii) recover connectivity. We describe in §5.2 how does Blink address them entirely in the data plane.

Dealing with noisy signal. To discover its fair share of bandwidth, a TCP endpoint keeps increasing its transmission rate until a packet loss is detected, triggering a retransmission. TCP retransmissions therefore occur naturally, even without network failures. Likewise, minor temporary congestion events can also lead to bursts of packet drops, which will trigger subsequent bursts of retransmissions, again, without necessarily implying a failure.

Requirement 1: A data-plane-driven fast-reroute system should only react to major disruptive events while being immune to noise and ordinary protocol behavior.

Dealing with fading signals. As shown in Figure 5.3, the amplitude of the signal (*i.e.*, the count of TCP retransmissions) quickly fades with the backoff round as the compounded signal spreads over longer and longer periods.

Requirement 2: A data-plane-driven fast-reroute system should catch the failure signal within the first retransmission rounds.

Mitigating the effect of sampling. As tracking retransmissions in real-time requires state, monitoring *all* flows is not possible. As such, a fast-reroute system will necessarily have to track and detect failures using a subset of the flows. Yet, not all flows are equally useful when it comes to failure reaction: intuitively, highly active flows will retransmit almost immediately, while long-lived flows might not retransmit at all (if no packet was sent recently). From a fast-reroute viewpoint, tracking non-active flows is useless.

Requirement 3: A data-plane-driven fast-reroute system should select the flows it tracks according to their activity.

Ensuring forwarding correctness without control plane. While data-plane signals are faster to propagate than control-plane ones, they carry no information about the cause of the failure and how to avoid it. As such, simply rerouting to a backup next-hop upon detecting a problem might not work, as it might also be affected by the failure. Worse, the problem can even be at the destination itself, in which case no alternative next-hop will actually work. Given this lack of precise information, a fully data-plane-driven fast-reroute system has no other choice but trying and observing.

Requirement 4: A fully data-plane-driven fast-reroute system should select its backup next-hops in a data-driven manner, verifying that traffic resumes.

5.2 Overview

In this section, we provide a high-level description of Blink. We first focus on its data-plane implementation at the node level (§5.2.1). We then describe how Blink can be deployed at the network level (§5.2.2).

5.2.1 Blink, at the node level

Figure 5.4 describes the overall workflow of a Blink data-plane pipeline. The pipeline is essentially composed of three consecutive stages: (i) a *selection* stage that efficiently identifies active flows to monitor; (ii) a *detection* stage that analyzes RTO-induced retransmissions across the monitored flows and looks for any significant increase; and (iii) a *rerouting* stage that aims to retrieve connectivity by probing alternative next-hops upon failure. We now briefly describe the key ingredients behind each stage and provide details in §5.3.

Selecting flows to track. For efficiency and scalability, a Blink node cannot track all possible 865k+ IP prefixes or even all the flows destined to some prefixes. An initial design choice thus concerns which prefixes to track, and which specific flows to track for the selected prefixes.

Any approach based on data-plane signals is able to effectively monitor only the prefixes carrying a certain amount of packets. Blink is no exception, and therefore focuses on the most popular destination prefixes. While this might seem a limitation, it is actually a feature: the Internet traffic is typically skewed and a very limited fraction of prefixes carry most of the traffic, while the rest of the prefixes see little to none [150]. Blink can thus reroute the vast majority of the traffic by tracking a limited number of prefixes. We designed Blink to accommodate at least 10k prefixes in current programmable switches (§5.4).

Regarding which flows to track for a prefix, Blink adopts a simple but effective strategy. For each monitored prefix, the Flow Selector tracks a very small subset (64, by default) of *active flows*—*i.e.*, flows that send at least one packet within a moving time window (2 s by default). Tracked flows are replaced as soon as they become inactive, or after a given timeout (8.5 min by default) even if they remain active. We did not reuse heavy hitter detection algorithms such as [159], since they are designed to offer a higher accuracy than we need (heaviest flows instead of just active ones) at the expense of additional complexity and resources.

Detecting failures. A central idea of Blink is to infer a remote failure affecting a destination prefix from the loss of connectivity for a statistically significant number of previously active flows towards that destination. While possible in principle, Blink does not look at the flows progression (*i.e.*, if the flows continue to send new data packets) to detect a failure, as only a subset of the flows may be affected, *e.g.*, because of load-balancing.

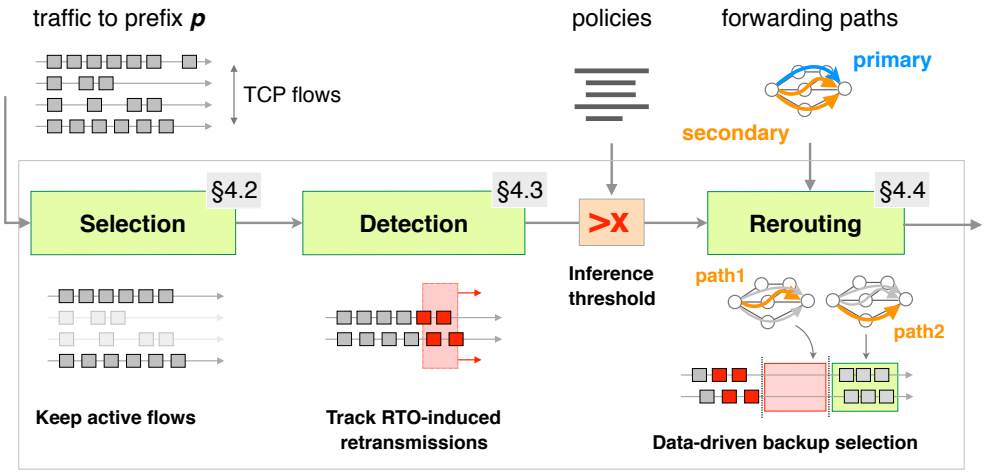


Figure 5.4 Blink data-plane workflow and key ingredients.

The detection stage looks for evidence of connectivity disruption across the flows identified by the Flow Selector. It stores key information on the last seen packet for each flow and determines if a new packet traversing the data-plane pipeline is a duplicate of the last seen one – an hallmark of RTO-induced TCP retransmissions (see Figure 5.2). Based on this check, for each destination prefix, it monitors the number of flows with at least one recent retransmission over a sliding time window of limited size (800 ms, by default). When the majority of the monitored flows experience at least one retransmission in the same time window, Blink infers a failure.

Rerouting quickly. When Blink detects a failure, the Rerouting Module quickly reroutes traffic by modifying the next-hop to which packets are forwarded, at line rate. In Blink's current implementation, the decisions of both when to reroute and to which backup next-hop to reroute are configurable by the operator based on their policies, as we believe that operators want to be in charge of this critical, network-specific operation.

When rerouting, the Rerouting Module sends few flows to each backup path to check which one is able to restore connectivity. It then uses the best and working one for all the traffic. The next-hops are configured at runtime by the operator to re-align the data-plane forwarding to the control-plane (*e.g.*, BGP) routes when the control plane has converged.

5.2.2 Blink, at the network level

The "textbook" deployment of Blink consists in deploying it on all the border routers of the ISP to track all the transit traffic. In this deployment, border routers either reroute traffic locally (if possible) or direct it to another border router (*e.g.*, through an MPLS tunnel). Of course, nothing prevents the

deployment of Blink inside the ISP as well. In fact, Blink also works for intra-ISP failures, *e.g.*, local to the Blink node or on the path from the Blink switch and an ISP egress point.

Blink is partially deployable. Deploying Blink on a single node already enables to speed up connectivity recovery for all traffic traversing that particular node. Also, Blink requires no coordination with other devices: each Blink node autonomously extracts data-plane signals from the traversing packets, infers major connectivity disruptions, and fast reroutes accordingly. To avoid forwarding issues, Blink verifies the recovery of connectivity for the rerouted packets by monitoring the data plane (see §5.3.4.2).

When rerouting, Blink also notifies the control plane, and possibly the ISP operator. This enables coordination with the control plane (*e.g.*, future SDN controllers), such as imposing the next-hop upon control-plane convergence, or discarding routes that are not working in the data plane.

5.3 Data-plane design

In this section, we describe the data-plane pipeline that runs on a Blink node, its internal algorithms, design choices and parameter values. Figure 5.5 depicts the four main components of the Blink data-plane pipeline: the Prefix Filter (§5.3.1), the Flow Selector (§5.3.2), the Failure Inference (§5.3.3), and the Rerouting Module (§5.3.4).

5.3.1 Monitoring the most important prefixes

To limit the resources used by Blink, the operator should activate Blink only for a set of important prefixes. A sensible approach would be to activate it for the most popular destination prefixes, as they carry most traffic, although nothing prevents the operator to select other prefixes – as long as there is enough TCP traffic destined to each of them (§5.5.1.1). To activate Blink for a prefix, the control plane adds an entry in the metadata table at runtime which matches the traffic destined to this prefix using a longest prefix match. Traffic destined to a prefix for which Blink is not active goes directly to the last stage of the data-plane pipeline and is forwarded normally (*i.e.*, find the next-hop and replace the layer 2 header).

The metadata table attaches to the matched packets a distinct ID according to their destination prefix. As memory (*e.g.*, register arrays) is often shared between the prefixes, this ID is used as an index to the memory. Observe that Blink could combine prefixes with common attributes (*e.g.*, origin AS or AS path) and which are likely to fail at the same time by mapping them to the same ID. This would increase the intensity of the signal, and would allow Blink

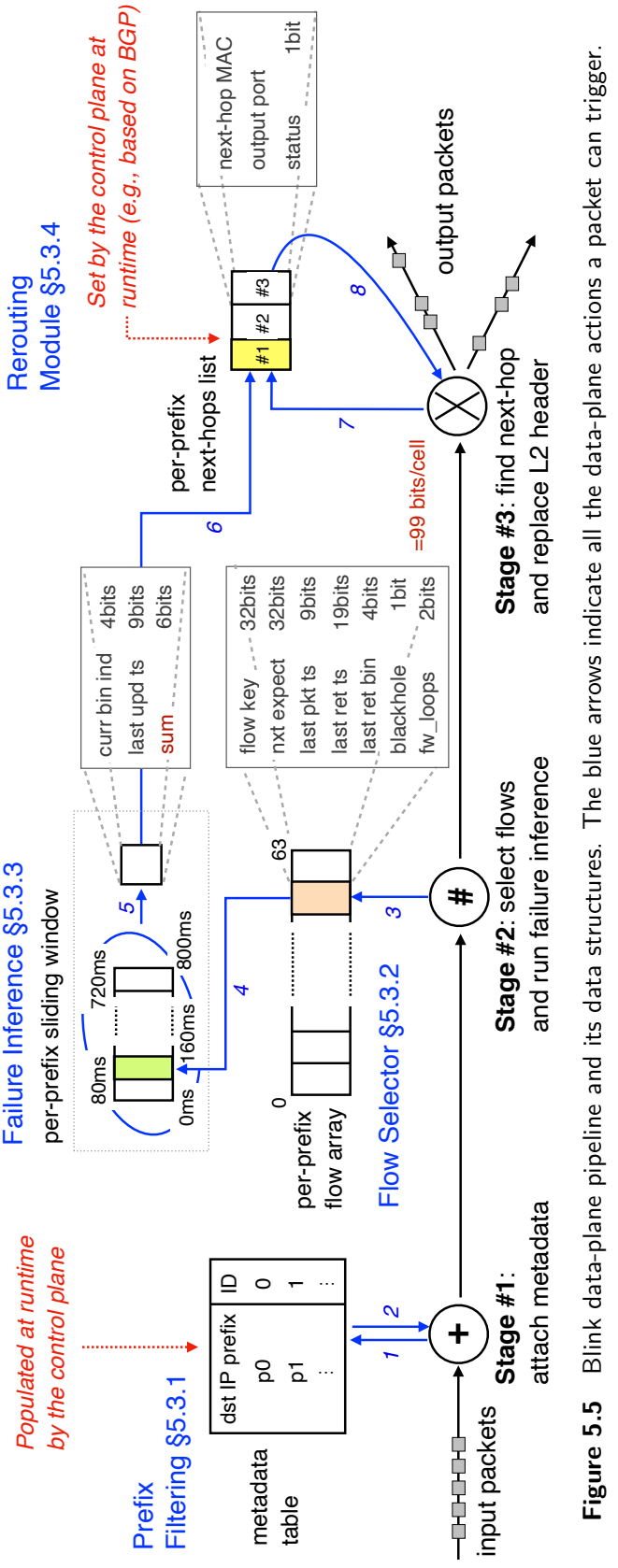


Figure 5.5 Blink data-plane pipeline and its data structures. The blue arrows indicate all the data-plane actions a packet can trigger.

to cover more traffic. Additionally, packets that do not carry useful information, *i.e.*, non-TCP traffic, and certain signaling-only packets such as SYN and ACK packets with no payload are not considered by Blink and are directly sent to the final stage.

5.3.2 Selecting active flows to monitor

Packets destined to a monitored prefix go to the Flow Selector, which will select a limited number of active flows (64 per prefix), and keep information about each of them.

Limiting the number of selected flows. Each flow for a given prefix is mapped to one of the 64 cells of a per-prefix flow array using a 6-bit hash of the 4-tuple (which includes source and destination IP and the port numbers). While we expect many flows to collide in the same cell, only one occupies a cell. This is enforced by storing the flow_key, namely a 32-bit hash of the 4-tuple in each cell of the flow array.

Flows colliding in the same cell are possible candidates to substitute the flow currently occupying that cell when it becomes inactive. It can happen that two flows mapped to the same cell have the same flow_key, in which case both would end up occupying the same cell, causing Blink to mix packets from two distinct flows and thus preventing it to correctly detect retransmissions for either flows. However, since we use a total of 38 bits (6 bits to identify the cell and 32 bits for the flow_key) to identify each flow, such collisions will rarely happen. The probability of collision can be computed from a generalization of the *birthday problem*: given n flows, what is the probability that none of them returns the same 38 bit hash value? This probability is equal to $\frac{2^{38}!}{(2^{38}-n)!} * \frac{1}{2^{38^n}} \approx e^{-\frac{n(n-1)}{2*2^{38}}}$. With $n = 10,000$ flows for a given prefix, the probability to have a collision is only 0.02%.

Replacing inactive flows. The challenge behind selecting active flows is that flows have different packet rates, which also change over time, *e.g.*, an active flow at time t may not be active anymore at time $t+1s$. A naive *first-seen*, *first-selected* strategy would clearly not work, because the selected flows might send packets at such a low rate that they would not provide any timely information upon a connectivity disruption – simply because there is no packet to retransmit. Our experimental evaluation in §5.5 confirms this intuition.

The Flow Selector monitors the activity of each selected flow by tracking the timestamp of the last packet seen in the register last_pkt_ts. As soon as the difference between the current timestamp and last_pkt_ts is greater than an eviction timeout, the flow is evicted and immediately replaced by another flow colliding in the same cell. A TCP FIN packet also causes immediate eviction. Intuitively, flow eviction makes the Flow Selector work very well for

prefixes which have many high-rate flows at any moment in time, or a decent fraction of long-living ones – which we expect to be often the case for traffic towards popular destinations. Our evaluation on real traffic patterns (see §5.5) confirms that this simple strategy is sufficient to quickly infer major connectivity disruptions.

Calibrating the eviction timeout. A remaining question for this component of the pipeline is how to dimension the eviction timeout. On one hand, we would like to evict flows as soon as their current packet rate is not amongst the highest for that prefix. On the other hand though, Blink needs to keep track of the flows long enough to see the first few packet retransmissions induced by a RTO expiration upon connectivity interruptions. Indeed, an eviction timeout of few hundred milliseconds is likely to be too low in many cases, since a flow takes *at least* 200ms to issue the first pair of duplicate packets. Note that 200ms only happens if there is no new packet between the first unacknowledged packet and the first retransmission. Also, remember that Blink considers only consecutive duplicates as packet retransmissions (see §5.1). By default, the eviction timeout is set to 2 s, which ensures to detect up to two pairs of consecutive duplicates for typical TCP implementations.

5.3.3 Detecting failures

We now describe how Blink detects RTO-induced retransmissions on the set of selected flows, and uses this information to accurately infer failures.

Detecting RTO-induced retransmissions. A partial or full retransmission of payload of the TCP packet can be detected by comparing the sum of its sequence number and payload length to the corresponding sum of the previous packet of the same flow. For example, in Figure 5.2 when the packet S:1000 (packet with sequence number 1000) arrives Blink will store 2100 (sum of sequence number and payload). If the next arriving packet triggers storing 2100 as well, a retransmission is detected. Observe that we store the expected sequence number per flow instead of the current one to account for cases where a packet is only partially acked.

Our design targets consecutive retransmissions for two reasons. First, RTO-induced retransmissions are consecutive (see Figure 5.2), whereas congestion-induced retransmissions (*i.e.*, noise) are likely to be interleaved by non-retransmissions, and hence will (correctly) not be detected. Second, this detection mechanism requires a fixed number of memory per flow, regardless the flow's packet rate.

Counting the number of flows experiencing retransmissions over time. Figure 5.3 shows that the TCP signal upon a failure is short and fading over time. To quickly and accurately detect the compounded signal across multiple

flows, we use a per-prefix sliding window. To implement a sliding window of size k seconds in P4, we divide it in 10 consecutive bins of 6-bit each, each storing the number of selected flows experiencing retransmissions during $k/10$ seconds. As a result, instead of sliding for every packet received, the sliding window moves every $k/10$ seconds period. More bins can improve the precision but would require more memory. This design enables us to implement the sliding window in P4 using only three information per prefix: (i) current_index, the index of the bin focusing on the current period of time, (ii) sum, the sum of all the 10 bins, and (iii) last_ts_sliding, the timestamp in millisecond precision of the last time the window slid. The additional 19-bit and 4-bit per-flow information last_ret_ts and last_ret_bin are also required to ensure that a flow is counted maximum one time during a time window. We provide more details about the implementation in §5.4.

Calibrating the sliding window. The duration of the sliding window affects the failure detection mechanism. A long time window (*e.g.*, spanning several seconds) has more chance to include unrelated retransmissions (*e.g.*, caused by congestions), whereas a short time window (*e.g.*, 100ms) may miss a large portion of the retransmissions induced by the same failure because of the different RTO timers. We set the duration of the sliding window to 800ms, with 10 bins of 80ms. First, because the minimum RTO is 200ms, a 800ms sliding window ensures to include all the retransmissions induced by the failure within the first second after the failure. Second, because under realistic conditions (in terms of RTT [84, 21, 157] and RTT variation [21]), flows would often send their first two retransmissions within the first second after the failure.

Inferring failures. A naive strategy consisting in inferring a failure when *all* the selected flows experience retransmissions would result in a high number of false negatives due to the fact that some flows may not send traffic during the failure, or simply because some flows have a very high RTT (*e.g.*, >1s). On the other hand, inferring a failure when only few flows experience retransmissions may result in many false positives because of the noise. As a result, by default Blink infers a failure for a prefix if the majority of the monitored flows (*i.e.*, 32) destined to that prefix experience retransmissions.

5.3.4 Rerouting at line rate

As soon as Blink detects a failure for a prefix, it immediately reroutes the traffic destined to it, at line rate. We first show in §5.3.4.1 how Blink maintains the per-prefix next-hops list used for (re)routing traffic. Then, we show in §5.3.4.2 how Blink avoids forwarding issues when it reroutes traffic.

5.3.4.1 Maintaining the per-prefix next-hops list

To reroute at line rate, Blink relies on pre-computed per-prefix backup next-hops. The control plane computes the next-hops consistently with BGP routes and specific policies defined by the operator. For each prefix, Blink maintains a list of next-hops, which are sorted according to their preference (see Figure 5.2). Each next-hop has a status bit. To reroute at line rate, Blink deactivates the primary next-hop by setting its status bit to 1 (*i.e.*, not working). Per-prefix next-hops are stored in register arrays and are updated at runtime by the controller, *e.g.*, when a new BGP route is learned or withdrawn. If a next-hop is not directly connected to the Blink node, Blink can translate it into a forwarding next-hop using IGP (or MPLS) information, as a normal router would do.

Falling back to the primary next-hop after rerouting.

After an outage, BGP eventually converges and Blink updates the primary next-hop and use it for routing traffic. However, Blink cannot know when BGP has fully converged. Our current implementation waits for a fixed time (*e.g.*, few minutes, so that BGP is likely to have converged) after rerouting before falling back to the new primary next-hop. We acknowledge that this approach might not be optimal (*e.g.*, it potentially sacrifices path optimality), but it guarantees packet delivery by using policy-compatible routes and avoids possible disruptions caused by BGP path exploration [133]. Investigating a better interaction with the control plane is left for future work.

5.3.4.2 Avoiding forwarding issues

Since Blink runs entirely in the data plane, it likely reroutes traffic before receiving any control-plane information possibly triggered by the disruption. In addition, even when carefully selecting backup next-hops (*e.g.*, by taking the most disjoint AS path with respect to the primary path), we fundamentally cannot have a-priori information about where the root cause of a future disruption is, or where the backup next-hop sends the rerouted traffic after the disruption. As a result, Blink *fundamentally cannot prevent* forwarding issues such as blackholes (*i.e.*, when the next-hop is not able to deliver traffic to the destination) or forwarding loops to happen. The good news, though, is that Blink includes mechanisms to *quickly react to forwarding issues* that may inevitably occur upon rerouting.

Probing the backup next-hops to detect anomalies. When rerouting, Blink reacts to forwarding anomalies by probing each backup next-hop with a fraction of the selected flows in order to assess whether they are working or not. For example, with 2 backup next-hops, one half of the selected flows is rerouted to each of them. The non-selected flows destined to this prefix are rerouted to

the preferred and working backup next-hop. Blink does this in the data plane using the per-prefix next-hops list.

When a backup next-hop is assessed as not working, Blink updates its status bit. After a fixed period of time since rerouting (1s, by default), Blink stops probing the backup paths and uses the preferred and working one for all the traffic, including the selected flows. If all the backup next-hops are assessed as not working, Blink reroutes to the primary next-hop and falls back to waiting for the control plane to converge.

Avoiding blackholes. Blink detects blackholes by looking at the proportion of restarted flows. After rerouting, Blink tags a flow as restarted by switching its blackhole bit to 1 as soon as it sees a packet for this flow which is not a retransmission. When the probing period is over, Blink assesses a backup next-hop as not working if less than half of the flows routed to that next-hop have restarted. The duration of the probing period (1s) is motivated by our goal of restoring connectivity at a second-level time scale, while also providing retransmissions with a reasonable time for reaching the destination through the backup next-hop and triggering the restart of the flows. For example, if Blink reroutes 778 ms after a failure (the median case, see §5.5.1.1) and assuming a reasonable RTT (*e.g.*, the median case in [84, 21, 157]), it is likely that the rerouted flows will send a retransmission and receive the acknowledgment (if the next-hop is working) within the following 1 s period.

Breaking forwarding loops. Blink detects forwarding loops by counting the number of duplicate packets for each flow. The key intuition is that forwarding loops have a quite strong signature: the same packets are seen over and over again by the same devices. This signature is very similar to the TCP signature upon a failure, where TCP traffic sources start resending duplicate copies of the same packets for every affected flow, at increasingly spaced epochs. As a result, the algorithm used by Blink to detect retransmissions also detects looping packets. To differentiate between normal retransmissions and looping packets, Blink relies on the delay between each duplicate packet. TCP can send for a flow up to 2 retransmissions in 1 s because of the exponential backoff (see §5.1.1), whereas a packet trapped in a forwarding loop can be seen many more times by the Blink node. Hence, Blink counts the number of duplicate packets it detects for each flow after the rerouting using the information fw_loops stored in each cell of the flow array, and tags a backup next-hop as not working by switching its status bit as soon as it detects more than 3 duplicate packets for a flow rerouted to this backup next-hop.

Observe that this mechanism reacts very quickly to the most dangerous loops, *i.e.*, the ones that recirculate packets very fast and hence are most likely to overload network links and devices. Longer and slower loops are mitigated in at most 1 s as Blink assumes the respective next-hop cannot deliver packets to the destination (*i.e.*, there is a blackhole).

5.4 Implementation

We have fully implemented the data-plane pipeline of Blink as described in §5.3 in ≈900 lines of $P4_{16}$ [165] code and in Python. We have also developed a $P4_{Tofino}$ implementation of Blink that runs on a Barefoot Tofino switch [5]. However, our $P4_{Tofino}$ implementation currently only supports two next-hops, one primary and one backup. Unlike our $P4_{16}$ implementation, our $P4_{Tofino}$ implementation uses the resubmission primitive in two cases: whenever the Flow Selector evicts a flow, or if two retransmissions from the same flow are reported within 800ms, *i.e.*, the duration of the sliding window. When resubmitting a packet, it is processed twice by the ingress pipeline of the switch and thus triggers more actions while being forwarded by the switch. The drawback is that the switch has to process more packets, reducing its overall bandwidth. However, the two cases for which packets are resubmitted only occur for the set of selected flows (*i.e.*, only 64 flows per prefix, see §5.3.2), and not for all the flows, thus limiting the impact on the bandwidth of the switch.

We now describe in more detail how we implemented the sliding window in $P4_{16}$ (§5.4.1) and then show how much hardware resources Blink needs in a Tofino switch (§5.4.2).

5.4.1 Implementation of a sliding window in $P4_{16}$

Blink uses one sliding window per prefix to count the number of flows experiencing retransmissions over time among the selected flows (§5.3.3). Besides the ten bins, Blink needs three meta information for each sliding window: (i) current_index, the index of the bin focusing on the current period of time, (ii) sum, the sum of all the 10 bins and (iii) last_ts_sliding, the timestamp in millisecond precision indicating when Blink has incremented the current_index to slide the window. When Blink detects retransmissions, it increments both the value associated with the bin at the index current_index and the sum. Upon reception of a packet at timestamp t, and assuming the window covers a period of k millisecond, if $t - \text{last\_ts\_sliding} > k/10$, Blink slides the window with the following operations. First, Blink finds the index of the expired bin by computing $(\text{current\_index} + 1) \bmod 10$. Note that because the modulo operator is not available in $P4_{16}$, we implement this calculation with if-else conditions. Second, it subtracts the value stored in the expired bin to sum. Third, it resets the value in the expired bin. Fourth, Blink makes current_index point to the expired bin. Finally, Blink updates last_ts_sliding to last_ts_sliding $+ k$. As a result, the counter sum always returns the number of flows experiencing retransmissions during the last $9/10k$ to k seconds.

A flow may send several retransmissions within a time window. Thus, Blink uses two additional per-flow metadata to avoid summing several retransmissions from the same flow in the same time window. The metadata last_ret_ts stores the

timestamp of the last retransmission reported for the corresponding flow. The metadata last_ret_bin stores the bin index corresponding to this timestamp. Consider that a retransmission for a flow is reported at time t, then if $t -$ last_ret_ts $< k$, Blink decrements the value in the bin at the index last_ret_bin and increments the value associated with the current bin. The sum remains the same, and last_ret_ts is set to the current timestamp and last_ret_bin is set to the current bin index.

5.4.2 Hardware resources usage

Blink is intended to run on programmable switches with limited resources. As a result, we designed Blink to scale based on the number prefixes it monitors and not on the actual amount of traffic destined to those prefixes. In this section, we derive the resources required by Blink to work for one prefix and show that it can easily scale to thousands of prefixes.

First, for every prefix, Blink needs one entry in the metadata table. Then, for each selected flow, the Flow Selector needs 99 bits (see Figure 5.5). As Blink monitors 64 flows per prefix, it needs a total of $64 * 99 = 6336$ bits to monitor a prefix. Blink does not store the timestamps (*e.g.*, last_pkt_ts and last_ret_ts) in 48 bits (the original size of the metadata) but instead approximates them using only 9 bits for a second precision and 19 bits for a millisecond precision to save memory. Blink shifts the original 48-bit timestamp to the right to obtain the second (resp. millisecond) approximation of the current timestamp. To fit in 9 (resp. 19) bits, Blink also resets the timestamps every 512 s ($\approx$ 8.5min) by subtracting to the original 48-bit timestamp a reference_timestamp. The reference_timestamp is simply a copy of the original 48-bit timestamp (stored in a register shared by all the prefixes) that is updated only every 512 s. Note that the Flow Selector evicts a flow if the current timestamp is lower than the timestamp of the last packet seen for that flow, which happens whenever Blink resets the timestamps (*i.e.*, every $\approx$ 8.5min).

The sliding window requires ten bins of 6 bits each, as well as $4 + 9 + 6 = 19$ bits to store additional information (see Figure 5.5), making a total of 79 bits. For the rerouting, Blink only requires one status bit per next-hop. With three next-hops, three extra bits are thus required. In total, for one prefix, Blink requires $6336 + 79 + 3 = 6418$ bits. As current programmable switches have few megabytes of memory, we expect Blink to support up to 10k prefixes, possibly even more.

5.5 Evaluation

We evaluate Blink's accuracy, speed, and effectiveness in selecting a working next-hop based on simulations and synthetic data (§5.5.1, §5.5.2). We then evaluate Blink using real traces and actual hardware (§5.5.3).

5.5.1 Blink's failure detection algorithm

Packet traces of real Internet traffic are hard to gather, and for the few traces publicly available [9, 46], there is no ground truth about possible remote failures on which Blink should reroute. Still, it makes little sense to evaluate Blink on traffic with non-realistic characteristics, or without knowing if Blink is correctly or incorrectly rerouting packets. We therefore adopt the following evaluation methodology.

Methodology. We consider 15 publicly available traces [9, 46] listed in Table 5.1, accounting for a total of 15.8 h of traffic and 1.5 TB of data.

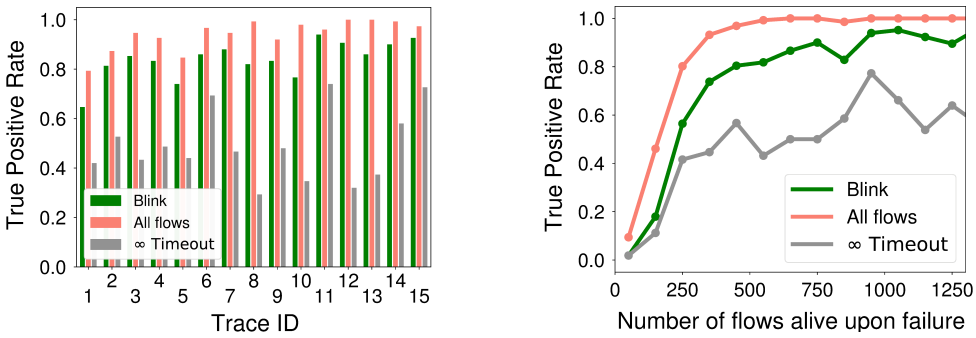
For each prefix, we extract the distributions of flow size, duration, average packet size, and RTT. To measure the RTT of the flows from the traces, we use the time difference between the SYN and ACK packets sent by the initiator of a connection as described in [94, 157]. We then run simulations with ns-3 [17] on a dumbbell topology similar to [172], where traffic sources generate flows exhibiting the same distribution of parameters than the one extracted from the real traces.

In some of our simulations, we introduce a failure after 10 s on the single link connecting the sources with the destinations, thus affecting all flows. In other experiments, we introduce random packet drops and no failure. We collect traffic traces for all simulations, feed them to our Python-based implementation of Blink one by one, and check if and when our system would fast reroute traffic.

Baselines. Since we are not aware of any previous work on real-time failures detection on the basis of TCP-generated signals, we compare Blink against two baseline strategies. The first strategy, *All flows*, consists in monitoring up to 10k flows for each prefix, and rerouting if any 32 of them sees retransmissions within the same time window. This strategy provides an upper bound on Blink's ability to reroute upon actual failures while ignoring memory constraints. The second strategy, ∞ *Timeout*, is a variant of Blink where flows are only evicted when they terminate (with a FIN packet), and never because of the eviction timeout. This strategy assesses the effectiveness of Blink's flow eviction policy.

Trace ID	Name	Date	Duration	Bit Rate	Trace Size	RTT median
1	caida-equinix-chicago.dirA	29-05-2013	3719 s	1631 Mbps	67 GB	200.22 ms
2	caida-equinix-chicago.dirB	29-05-2013	3719 s	2119 Mbps	73 GB	155.65 ms
3	caida-equinix-sanjose.dirA	21-03-2013	3719 s	2920 Mbps	122 GB	104.94 ms
4	caida-equinix-sanjose.dirB	21-03-2013	3719 s	1618 Mbps	82 GB	191.11 ms
5	caida-equinix-chicago.dirA	19-06-2014	3719 s	1629 Mbps	60 GB	209.72 ms
6	caida-equinix-chicago.dirB	19-06-2014	3719 s	6271 Mbps	163 GB	160.91 ms
7	caida-equinix-sanjose.dirA	19-06-2014	3719 s	3722 Mbps	144 GB	169.71 ms
8	caida-equinix-chicago.dirA	17-12-2015	3776 s	2540 Mbps	111 GB	240.18 ms
9	caida-equinix-chicago.dirB	17-12-2015	3776 s	3151 Mbps	99 GB	68.30 ms
10	caida-equinix-chicago.dirA	21-01-2016	3819 s	2250 Mbps	126 GB	224.09 ms
11	caida-equinix-chicago.dirB	21-01-2016	3819 s	4959 Mbps	143 GB	69.57 ms
12	caida-equinix-nyc.dirA	15-03-2018	3719 s	3027 Mbps	94 GB	306.76 ms
13	caida-equinix-nyc.dirA	19-04-2018	3719 s	3893 Mbps	125 GB	283.82 ms
14	mawi-samplepoint-F	12-04-2017	7199 s	878Mbps	74 GB	124.14 ms
15	mawi-samplepoint-F	07-05-2018	900 s	1098 Mbps	10 GB	156.20 ms
Total			15.8 h		1.5 TB	

Table 5.1 List of 15 real traces that we use to evaluate Blink. All together, they cover a total of 15.8 hour of traffic.



(a) Different traffic patterns **(b)** Different number of flows

Figure 5.6 Blink has high TPR when relatively few flows (*e.g.*, more than 350) are active upon the failure.

5.5.1.1 Blink often detects actual failures, quickly

We first evaluate Blink's ability to detect connectivity disruptions. For each real trace in our dataset, we randomly consider 30 prefixes which see a large number of flows ($>$ 1000 flows in the trace), and we generate 5 synthetic traces per prefix, each with a different number of flows starting every second (from 100 to 500 flows generated *per second*) and each containing a failure at a preconfigured moment in time.

We then compute the True Positive Rate (TPR) of Blink on these traces. For each synthetic trace, we check whether Blink detects the failure (True Positive or TP) or not (False Negative or FN). The TPR is computed over all the tested synthetic traces, and is equal to $TP/(TP+FN)$.

Figure 5.6a shows the TPR of Blink and our baseline strategies as a function of the real trace used to generate the synthetic ones. As expected, the *All flows* strategy exhibits the best TPR among the three considered strategies at the cost of impractical memory usage. We see that Blink has a TPR which is very close to (*i.e.*, less than 10% lower than) the *All flows* strategy—while tracking only 64 flows. Overall, Blink correctly reroutes more than 80% of the times for 13 traces out of 15, with a minimum at 65% and a peak at 94%.

At the other extreme, the TPR of the ∞ *Timeout* strategy is much lower than Blink, below 50% for most traces, highlighting the importance of Blink's flow eviction policy.

As the RTT of the flows affects the failure signal used by Blink to detect failures (see §5.1.1), we also look at the TPR as a function of the RTT. On the synthetic traces with a median RTT below 50 ms (resp. above 300 millisecond), Blink has a TPR of 90.6% (resp. 76.0%). This shows that Blink is useful even when flows have a high RTT.

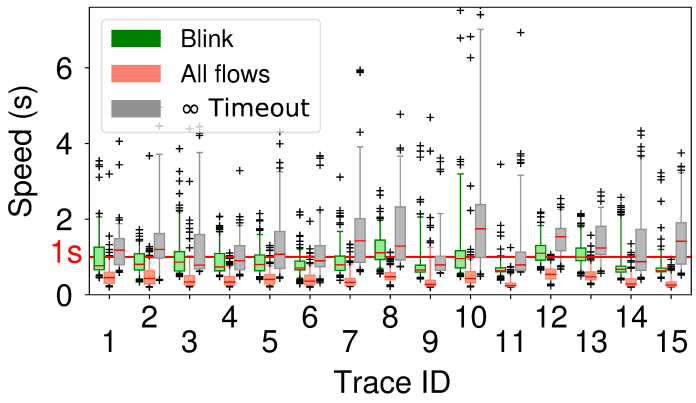


Figure 5.7 Blink is fast, for all traffic patterns.

As a follow-up, we then analyzed how much Blink's TPR varies with the number of flows active upon the failure (an important factor for Blink's performance). Figure 5.6b shows that Blink's TPR unsurprisingly increases if there are more flows active during the failure. With very few active flows, Blink cannot perform well, since the data-plane signal is too weak. However, Blink's TPR is already around 74% when about 350 flows are active, and reaches high values (more than 90%) with about 750 active flows. Again, Blink is much closer to the *All flows* strategy than to the ∞ *Timeout* one, although the *All flows* strategy reaches higher levels of TPR for lower number of active flows. These results suggest that Blink is likely to have a high TPR in a real deployment since we expect to see $\gg$750 active flows for popular destinations.

Not only does Blink detect failures in most cases, but it also recovers connectivity quickly upon failure detection. Figure 5.7 shows the time needed for Blink to restore connectivity for each of the real traces used to generate the synthetic ones, restricting on the cases where Blink detects the failure. Each box shows the inter-quartile range of Blink's reaction time. The line in each box depicts the median value; the whiskers show the 5th and 95th percentile.

Blink retrieves connectivity in less than 778 ms for 50% of the traces, and within 1 s for 69% of the traces. The *All flows* strategy restores connectivity within 365 millisecond in the median case, whereas the ∞ *Timeout* strategy needs 1.07 s (median). Naturally, Blink is faster when the RTT of the flows is low. On the synthetic traces with a median RTT below 50 ms, Blink reroutes within 625 ms in the median case. Yet, when the median RTT is above 300 ms, Blink is still fast and reroutes within 1.2 s in the median case.

5.5.1.2 Blink distinguishes failures from noise

One may wonder if Blink's ability to detect failures may not be due to it overestimating disruptions. By design, Blink cannot detect a failure without

packet loss %	1	2	3	4	5	6	7	8	9
					False Positive Rate (%)				
Blink	0	0	0	0.67	0.67	0.67	0.67	1.3	2.0
All flows	59	85	93	94	95	96	97	97	98
∞ timeout	0	0	0	0	0.67	0.67	0.67	0.67	0.67

Table 5.2 Blink avoids incorrectly inferring failures when packet loss is below 4%.

TCP retransmissions. Hence, the question is if Blink tends to overreact to relatively few, unrelated retransmissions, *e.g.*, induced by random packet loss.

To verify this, we generate synthetic traces with no failure but with an increasing level of random packet loss (from 1% to 9%) for all traffic. The trace synthesis follows our methodology of mimicking characteristics of real traffic for one prefix. For each real trace and loss percentage, we repeat the trace generation for 10 randomly extracted prefixes which see a large number of flows. For this experiment, we generate traces from 1-minute simulations where many (*i.e.*, 500) new flows start every second to ensure that Blink's Flow Selector is filled with flows, all potentially sending retransmissions.

For all these synthetic traces, we check whether Blink detects a failure (FP) or not (TN) and compute the False Positive Rate (FPR) as $FP/(FP+TN)$. Contrary to what happens for the TPR in §5.5.1.1, we expect *All flows* to be a worst case scenario as it sees all the retransmissions across all flows. On the other hand, ∞ *Timeout* should perform better than Blink because inactive flows (which are not evicted) do not contribute to the number of observed retransmissions.

Table 5.2 shows the FPR as a function of packet loss. Below 4% packet loss, Blink never detects failures. Between 4% and 7%, Blink incorrectly detected a failure for *one* synthetic trace out of the 150 generated. This indicates that Blink would work well under realistic traffic (we confirm this in §5.5.3.1), where the packet loss is often below these values. As a reference, the *All flows* strategy has an extremely high FPR, around 60% (resp. 85%) for traces with 1% (resp. 2%) packet loss. The ∞ *Timeout* strategy has only one false positive when the packet loss is between 5% and 10%, which, rather than a feature, is an artifact of tracking non-active flows.

Summary. Our results show that Blink strikes a good balance between detection of actual failures and robustness to noise (*i.e.*, TCP retransmissions not originated from a prefix-wide connectivity disruption). Blink's tradeoff is much better than the naive strategies: not evicting flows would significantly lower Blink's ability to correctly reroute upon failures, while monitoring all crossing flows comes with high sensitivity to noise (in addition to a likely impractical memory cost). Blink also recovers connectivity quickly (often within 1 s in our experiments) when it detects a disruption.

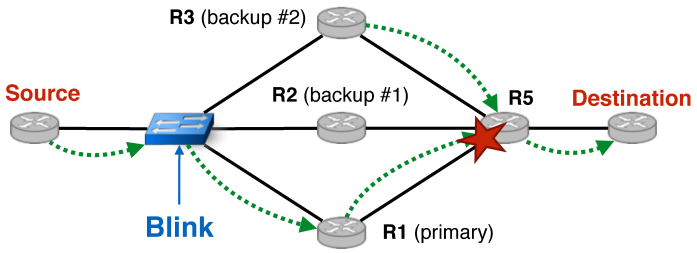


Figure 5.8 The network used to evaluate Blink's rerouting. Arrows indicate forwarding next-hops (R2 uses different next-hops depending on the experiment).

5.5.2 Blink's rerouting algorithm

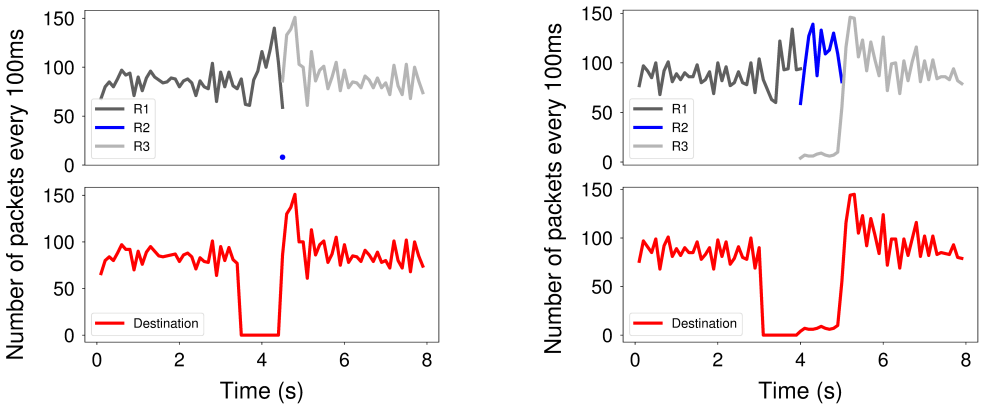
We now focus on the Blink's Rerouting Module. Our design ensures that rerouting is done entirely in the data plane, at line rate —we confirm this by experimenting with a Tofino switch, as described in §5.5.3. In this section, we therefore evaluate whether Blink is effective in rerouting to a working next-hop.

Methodology. We emulate the network shown in Figure 5.8 in a virtual machine attached to 12 cores (2.2 GHz). The P4 switch has three possible next-hops to reach the destination, R1 being the primary next-hop, R2 the most preferred backup next-hop and R3 the less preferred backup one. R1 and R3 use R5 as next-hop to reach the destination. R2 uses a different next-hop to reach the destination depending on the experiment, thus we do not depict it in the figure.

We emulate the P4 switch by running our $P4_{16}$ implementation of Blink in the P4 behavioral model software switch [18]. The P4 switch running Blink is linked to a Mininet network emulating the other switches. The source and the destination are Mininet hosts running TCP cubic.

We start 1000 TCP flows from the source towards the destination. To show the effectiveness of the Flow Selector, 900 flows have a low packet rate (chosen uniformly at random between 0.2 and 1 packet/s) while only 100 have a high packet rate (chosen uniformly at random between 2.5 and 20 packet/s). We use `tc` to control the per-flow RTT (chosen uniformly at random between 10 ms and 300 ms), and to drop 1% of the packets on the link between R5 and the destination in order to add a moderate level of noise. We first start the 900 flows with a low packet rate, so that the Flow Selector first selects them. Right after, we start the 100 remaining flows. Finally, after 20 s to 30 s, we fail the link between R1 and R5.

Blink quickly detects and breaks loops. We configure R2 to use the P4 switch as next-hop to reach the destination so that it creates a forwarding loop (by sending traffic back to the source) when Blink reroutes traffic to R2. Figure 5.9a shows the traffic captured at R1, R2 and R3 (top) and at the destination (bottom). Prior the failure, the traffic goes through R1, the primary



(a) Blink quickly breaks loops

(b) Blink reacts to blackholes

Figure 5.9 Traffic measurements quantifying Blink's speed in reacting to forwarding anomalies upon rerouting.

next-hop. Upon the failure, Blink probes if any of the available next-hops can recover connectivity: it sends half of the flows in the Flow Selector to R2 and the other half to R3. All the remaining flows go to R2 (preferred over R3). Blink detects the forwarding loop induced by R2 very quickly (only 8 packets were captured on R2) and immediately deactivates this next-hop to reroute all the traffic to R3, restoring connectivity within a total of 800 ms.

Blink quickly detects and routes around blackholes. In a separate experiment, we configure R2 to use R5 as next-hop, and we fail the link between R2 and R5 in addition to the one between R1 and R5. Figure 5.9b shows the traffic captured at R1, R2 and R3 (top) and at the destination (bottom). Upon the failure, Blink reroutes to R3 half of the selected flows, and to R2 the other half of the selected flows plus all the non-selected ones (since R2 is preferred over R3). However, because the link between R2 and R5 is down, the packets sent to R2 are just dropped by R2. After 1 s, Blink detects the blackhole and reroutes all the traffic to R3, restoring connectivity. The total downtime induced by the failure is 1.7 s.

5.5.3 Blink in the real world

So far, we have evaluated Blink with simulations and emulations. We now report on experiments that we run on real traffic traces and on a Barefoot Tofino switch.

5.5.3.1 Running Blink on real traces

In §5.5.1, we use simulated (but realistic) traffic traces to gain some confidence on Blink's accuracy in detecting connectivity disruptions. An objection might

be that our synthetic traces are not fully realistic. We therefore run Blink on the original real traces listed in §5.1 and tracked when it detects a failure. Note that we omitted failures detected for 73 prefixes (out of 2.28M) which *constantly* showed high-level of retransmissions (>20% of the flows retransmitting >50% of the time). In practice, Blink could detect such outliers at runtime and exclude those prefixes.

Here, unlike in §5.5.1 where simulate failures, we use real traces, for which we do not have ground truth. To measure Blink's correctness at detecting failures, we thus manually checked each case for which Blink detected a failure so as to confirm the connectivity disruption.

Over the 15.8 h of real traces, Blink detected 6 failures. In these 6 cases, the retransmitting flows represent 42%, 57%, 71%, 82%, 82% and 85% of all the flows active at that time and destined to the affected prefix. These numbers confirms that Blink is *not* sensitive to normal congestion events, and only reroutes in cases where a large fraction of flows experience retransmissions at the same time.

5.5.3.2 Deploying Blink on Barefoot Tofino switches

We finally evaluate our $P4_{Tofino}$ implementation of Blink on a Barefoot Tofino Wedge 100BF-32X. To do so, we generate TCP traffic between two servers connected via our Tofino switch running Blink. The server receiving the traffic has a primary and a backup physical link with the Tofino switch. We generate 1000 flows, 900 of which have a low packet rate and 100 a high one (similarly to §5.5.2). To show the influence of the RTTs on Blink when running on Tofino, we run two experiments, one with sub-1ms RTT, and another one in which we use `tc` to simulate for each flow an RTT chosen uniformly at random between 10 ms and 300 ms. After 30 s, we simulate a failure on the primary path, and measure the time Blink takes to retrieve connectivity via the backup link.

Blink-Tofino managed to restore connectivity in <1s. Blink retrieves connectivity in only 460 ms with sub-1ms RTT, and in about 1.1 s when the RTT of the flows is between 10 ms and 300 ms. We obtain comparable results (470 ms of downtime with sub-1 ms RTT) when running the same experiments with 3,000 flows among which 300 has a high packet rate.

5.6 Deployment considerations

We now discuss two possible operational concerns when deploying Blink in a real ISP network: adaptability and interaction with deployments of Blink in other ISPs.

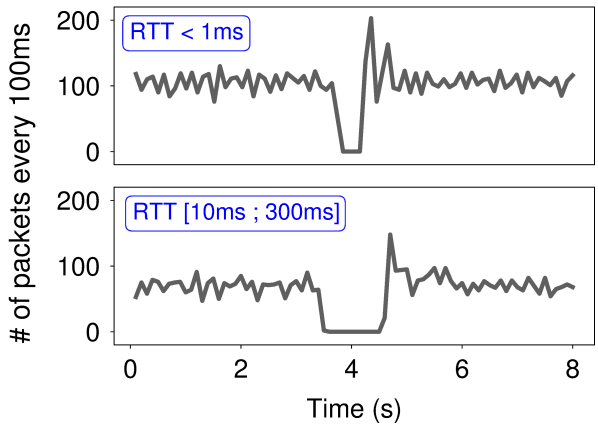


Figure 5.10 A Tofino switch running Blink retrieves connectivity within 460 ms if the RTT of the affected flows is below 1 ms (top figure); or within 1.1 s if the per-flow RTT ranges between 10 ms and 300 ms (bottom figure).

Adaptability. Clearly, a challenge in a real deployment of Blink is how to set its parameters correctly. This is hard because operators have different requirements, and traffic can exhibit varying characteristics. In §5.5, we show that it is definitely possible to set the parameter values so that Blink works well in real situations. Yet, in a real deployment, we envision that Blink could first be run in "learning mode", where it sends notifications to the controller instead of rerouting traffic. The controller then evaluates the accuracy of the system, for instance using control-plane data, and turns Blink on if the accuracy is good, or tune some parameters otherwise.

To guide operators, we list in Table 5.3 the main parameters used by Blink, and show how each of them can affect the performance of the system. We denote as TPR the True Positive Rate over all the synthetic traces generated from the 15 real traces listed in Table 5.1 and used in §5.5.1.1. The TPR with Blink's default values over all the synthetic traces is 83.9%. FPR denotes the False Positive Rate and is computed similarly as in §5.5.1.2.

Internet scale deployment. So far, we have described Blink's deployment in a single network (see §5.2). Of course, all ISPs have the same incentives to deploy Blink (*i.e.*, for fast connectivity recovery), so we envision that multiple, possibly all, ISPs might deploy Blink. Multi-AS deployment of Blink makes rerouting trickier. For example, if an Internet path traverses multiple Blink switches, it is not clear which ones will reroute, and whether the resulting backup path will be optimal. Blink switches can also interfere with each other. For example, if a Blink switch reroutes traffic to a backup path, a downstream Blink switch in the original path may lose part of the data-plane signal, preventing it to detect the failure. Finally, Blink's rerouting also increase the likelihood of creating inter-domain loops, since Blink selects backup next-hops based on BGP information, which might not be truthful if the downstream switches also run Blink.

Component	Name	Default value	Tradeoff
Flow Selector §5.3.2	Eviction timeout	2s	With a short eviction time (*e.g.*, 0.5s) flows can be evicted while they are retransmitting, reducing the TPR to 66.3%. With a longer eviction time (*e.g.*, 3s) inactive flows take more time to be substituted by active ones, reducing the TPR to 77.7%.
	Number of cells per prefix	64	Monitoring a small fraction of flows may result in a FPR increase. For example, with only 16 cells, the FPR is 2% for only a 3% of packet loss. However, the bigger the number of cells the smaller the amount of prefixes we can monitor due to memory constrains. With 64 cells (=64 flows monitored per prefix) Blink can support at least 10k prefixes.
Failure Inference §5.3.3	Sliding window duration	800ms	A long time window (*e.g.*, 1.6s) is more likely to report all the retransmitting flows, increasing the TPR to 89.8%, but also reports more unrelated retransmissions, increasing the FPR (0.67% for 3% of packet loss). A shorter time window (*e.g.*, 400ms) limits the FPR (0% for 9% of packet loss) but decreases the TPR to 49.4%.
	Sliding window number of bins	10	More bins increases the precision, at the price of using slightly more of memory. 10 bins give a precision $> 90\%$.
	Inference threshold	50%	A lower threshold, such as 25% (*i.e.*, 16 flows retransmitting when using 64 cells) gives a better TPR (94.8%), but increases the FPR to 7.3% for 4% of packet loss.
Rerouting Module §5.3.4	Backup next-hop probing time	1s	A longer probing period better prevents wrongly assessing a next-hop as not working, at the price of waiting more time to reroute.

Table 5.3 Parameters used by Blink, with their default values, and how they can affect the performance of the system.

While a full characterization of Blink's behavior in an Internet-scale deployment is outside the scope of this paper, Blink's design already guarantees some basic correctness properties. Blink already monitors for possible forwarding loops, and quickly breaks them by using additional backup next-hops (see §5.3.4.2). After having explored all possible backup next-hops, Blink also falls back to the primary next-hop indicated by the control-plane, even if not working: this would prevent oscillations where two or more Blink switches keep changing their respective next-hops in the attempt to restore connectivity. Finally, Blink switches do use BGP next-hops after BGP convergence.

Path optimality is much harder to guarantee within a system like Blink, where network nodes independently reroute traffic, without any coordination. However, we believe that path optimality can be transiently sacrificed in the interest of restoring Internet connectivity as quickly as possible. Besides, even with an Internet-wide deployment of Blink, path optimality will be restored when the control plane converges to post-failure paths.

5.7 Related Work

We now discuss related work beyond the Internet routing convergence studies and works focusing on Internet outages, which we already discuss in §2 and §3.

Traffic monitoring in the data plane. Few approaches monitor traffic using a programmable data plane. DAPPER is an in-network solution using TCP-based signals to identify the cause of misbehaving flows (whether the problem is in the network or not) [73]. Blink does not aim at identifying the cause of a particular flow failing but rather that many flows (for the same prefix) fail at the same time. In addition, unlike Blink, DAPPER requires symmetric routing for its analysis, which is often not the case in ISP environments. Sivaraman et al. [159] propose a heavy-hitter detection mechanism running entirely in the data plane. As Blink, it stores flows in an array and relies on flow eviction to keep track of the heaviest ones. However, unlike [159], Blink looks for active flows instead of the heaviest ones, on a per-prefix basis.

Data-driven networks. Motivated by the emergence of programmable data planes, many researchers have been arguing that network control should be made traffic-aware, as with Blink. Among others, such data-driven networks [62, 95] have been proposed for: optimizing routing strategies [168, 179, 153]; improving traffic engineering [178], streaming quality [119], or throughput in general [57]. Like with Blink, many of these data-driven solutions leverage programmable hardware and run directly in the data plane. For instance, Contra enables performance-aware routing within a network and entirely in the data plane [86]. With Contra, each node in the network runs a distance-vector protocol which propagates link metrics (*e.g.*, latency) via periodic probes. The nodes then

update their forwarding behavior based on the gathered metrics as well as the specified routing policies. RouteScout is a hybrid hardware/software system which enables performance-aware routing at the BGP level [26]. RouteScout calculates in the data plane and for each policy-compliant path metrics such as delay and loss and use them to synthetize a performance aware forwarding policy.

Outage inference from data traffic. Data-plane traffic has also been widely used in the past for ex-post measurement analyses. For example, WIND [88] infers network performance problems, including outages, from traffic traces by leveraging (among others) structural characteristics of TCP flows. In Blink, we perform online packet analysis, at line rate, but only to infer major connectivity disruptions – with simple yet effective algorithms that fit the limited resources of real switches.

Connection Path Reselection (CPR) is a system that identifies outages based on transport layer performance (TCP flows) in a timely manner and reroutes traffic on a per-flow basis and on healthy paths upon detection of an outage [112]. Its scope differs from Blink because it is a software-based approach tailored to run on edge networks such as CDNs. Blink is designed to run at line rate and on any programable switch, being at the edge or in the core of the network.

6

Snap: Taking the best of both worlds

Blink, as a fast reroute mechanism working *entirely* in the data-plane, is inherently fast but has to rely on simple algorithms because of the programmable hardware constraints. These algorithms work well under ideal conditions. However, they are often too simplistic for real scenarios, which exhibit complex patterns, weaker signals, or adversarial inputs. For instance, Blink simply triggers the rerouting for a prefix when half of the monitored flows experience retransmissions for that prefix. Besides, as being entirely in the data plane, Blink relies on a simple but naive and expensive (resource-wise) monitoring strategy: it monitors the prefixes individually and does not combine the information it receives from multiple of them, like SWIFT with its failure inference algorithm.

In this section, we present a hybrid hardware-software design for fast traffic rerouting upon remote outages. This hybrid design leverages the line-rate speed of the programmable data planes and the flexibility and resources provided by a controller running on a commodity server. We illustrate the benefits of a hybrid design with Snap, a fast reroute framework which combines the best of SWIFT and Blink.

Snap: the speed of Blink along with the smartness of SWIFT. Snap takes his inspiration from the hardware-software codesign problem, which is at the origin of a new design paradigm for networking applications and which has recently triggered discussions [124, 128] and resulted in many new frameworks [24, 37, 115]. Like Blink, Snap enables tracking the fast data-plane signals, such as the TCP retransmissions. However, as opposed to Blink, Snap takes the rerouting decision in software, allowing it to use much more advanced algorithms, such as the ones used by SWIFT. At its heart, Snap relies on a "smart" mirroring, which consists in mirroring to the controller only a fraction of the flows, the most useful ones according to the objectives. For instance, Snap focuses on mirroring active flows that are useful to detect remote outages. The fact that the rerouting decisions are now made in software enables using

more complex, thus precise, algorithms to detect Internet outages, finding the best alternatives paths, thwarting potential attacks, etc.

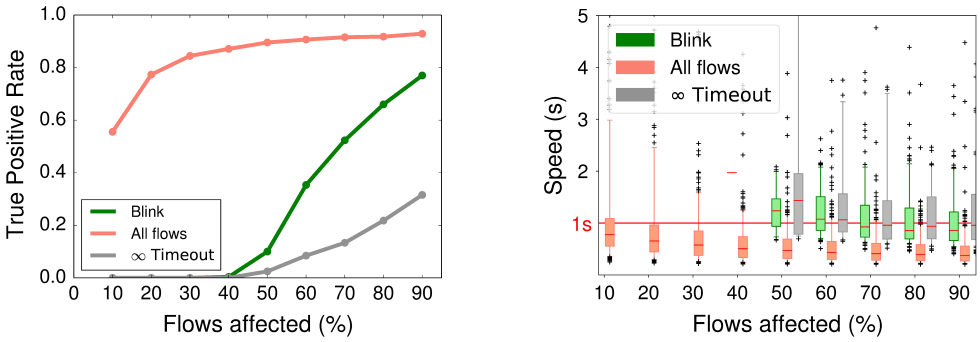
The benefits of the hardware-software codesign go beyond Snap. This chapter aims not just to present Snap but to highlight the benefits of its hardware-software codesign, which allows the rerouting process to be based on complex and custom decision algorithms that take as input the fast and carefully selected data-plane signals. In this chapter, we thus also discuss the potential algorithms that this hardware-software codesign allows implementing, and how they would improve the fast reroute process.

Yet, Snap already exhibits clear benefits compared to Blink and SWIFT in its current state. By combining the SWIFT logic for failure detection and localization with the data-plane signals used by Blink, Snap illustrates the capabilities offered by this hardware-software codesign. This combination of algorithms and inputs allows Snap to locate an outage quickly ($\approx$ 1 second); a capability that neither SWIFT nor Blink has. Localizing the outage enables using the SWIFT proactive rerouting strategy, *i.e.*, rerouting *all* the affected traffic towards backup paths bypassing the outage; a better strategy than the reactive one used by Blink (probing the backup paths to assess the working ones, for each prefix).

Outline. In this chapter, we first motivate in §6.1 our proposed hardware-software codesign by showing the limitations of Blink. We then show in §6.2 the key ingredients behind our proposed hardware-software codesign, with a focus on Snap. In §6.3, we focus on the design of Snap with its internal algorithms. In §6.4, we explain how we implemented Snap in $P4_{16}$ (data plane) and Python (controller), and show that it can run on current hardware and with existing traffic patterns. In §6.5, we evaluate Snap. Finally, we discuss the future work in §6.6 and show how we could leverage the hardware-software codesign to include more efficient algorithms and improve Snap.

6.1 Motivation

By looking at data-plane signals and running entirely in the data plane, Blink is fast. The downside, however, is that Blink has to trade precision and accuracy to gain simplicity so that its algorithms can run at line rate on the programmable switches. Unsurprisingly, by being too simple, Blink only works on the simple cases, missing many of the real (and not so simple) cases. We now show three concrete examples to illustrate why Blink is too simple to work in real life.



(a) Accuracy **(b)** Speed

Figure 6.1 The performance of the outage detection algorithm used by Blink decreases when the proportion of the flows affected by the outage decreases.

6.1.1 Blink misses many partial outages

The outage detection algorithm used by Blink is only based on one sort of input (TCP retransmissions) and uses a single conditional statement with only one threshold: if more than half of the monitored flows experience retransmissions, Blink reports an outage. As a comparison, SWIFT detects and locates an outage by running more complex algorithms which require iterating over the AS graph.

The problem with the simple detection algorithm used by Blink is that it may miss or needs more time to detect partial outages, *i.e.*, the ones that only affect a subset of the traffic destined to a prefix (*e.g.*, due to load-balancing). To illustrate this problem, we evaluate the performance of Blink upon partial failures and compare it to the two baselines we used to evaluate Blink: the *All flows* and ∞ *Timeout* strategies (see §5.5.1).

Methodology. For this evaluation, we randomly picked ten prefixes from each real trace listed in Table 5.1, and generated one synthetic trace for each of them, following the guidelines described in §5.5.1. For each trace, we simulated partial failures with nine different intensities (from 10% to 90% of the flows being affected). For these synthetic traces, 1223 flows (resp. 264) were active upon the failures in the median case (resp. 10th percentile).

Blink accuracy drops as the proportion of affected flows decreases. Figure 6.1a shows the TPR of Blink as a function of the percentage of flows affected by the failure. Unsurprisingly, because Blink needs to detect at least 32 flows experiencing retransmissions to detect the failure, the TPR is close to 0% if the failure affects less than 50% of the flows. For failures affecting 70% of the flows (resp. 90%), Blink works for 53% (resp. 77%) of the failures.

To improve Blink accuracy upon partial failures, one could lower the triggering threshold (initially set to 50%, see Table 5.3). Unfortunately, while this does increase the TPR, it also increases the FPR. For instance, this effect is visible

with the *All flows* strategy, with which only a small proportion of the flows needs to experience retransmissions to trigger the rerouting. The *All flows* strategy does exhibit a high TPR even for partial failures (Figure 6.1a), but also exhibits a very high FPR (see Table 5.2).

Blink speed decreases as the proportion of affected flows decreases. Figure 6.1b shows the time needed for Blink to restore connectivity upon a partial failure. Logically, as we decrease the number of affected flows, the detection speed of Blink decreases. For instance, it takes more than 1 second to detect most failures when less than 60% of the flows are affected.

6.1.2 Blink rerouting strategy cannot leverage path diversity

Blink's strategy to find a working backup path relies on a rather basic probing mechanism. Upon detecting a failure, Blink sends a fraction of the selected flows to each possible backup path to check which one can restore connectivity. While applicable, this reactive approach requires Blink to probe each backup path with a sufficiently large set of flows to assess whether they are working or not. Unfortunately, this requires precious hardware resources, especially when there are many different backup paths. Thus, the only option to prevent Blink from spending an unreasonably high amount of resources just for probing many backup paths, is to ignore some of them. For instance, the default implementation of Blink only considers two backup paths (§5.4).

Through a measurement study based on actual BGP data collected from BGP looking glasses, we now show that BGP routers often learn many different routes (*e.g.*, more than four) to reach a destination prefix. These results highlight that Blink would often have to ignore (potentially working) backup paths and thus cannot leverage path diversity.

Methodoloy. We used the Periscope API [74] to access BGP looking glasses and collect information about the routes learned by real routers to reach the prefixes advertised in the Internet. More precisely, we first took 100 random prefixes advertised in the Internet. Then, for each prefix, we collected the BGP routes from 30 routers, randomly picked from the set of 251 routers available from the Periscope API. Finally, we removed the cases for which only one route was available to reach the destination prefix and ending up with 2060 cases.

A router may learn many (> 10) backup routes to reach a prefix. In the medium case, a router knows three routes to reach a destination prefix. In the 75th percentile, a router knows five routes to reach a prefix, whereas, in the 90th percentile, it knows ten routes. The maximum is 92. Clearly, probing *e.g.*, ten or more different routes, each with a sufficiently large set of flows, requires monitoring more than just 64 flows per prefix, the default number of flows selected by Blink per prefix. Besides, due to the limitations imposed by

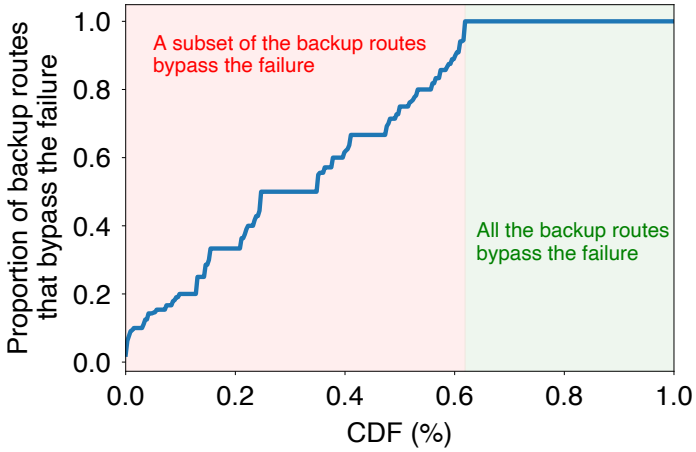


Figure 6.2 For a significant proportion of the simulated failures, only a subset of the available backup routes bypass the failure.

the hardware, the number of backup paths that can be used for each prefix in the Blink implementation is hardcoded and identical for all prefixes. Hence, to monitor *e.g.*, ten backup routes for a prefix, Blink would have to reserve enough resources to monitor ten backup routes for *all* the prefixes.

Blink is likely to miss working backup paths. We now show why a fast reroute framework targeting remote outages should consider *all* the available backup routes when rerouting traffic. Here, we first removed the cases for which only two routes are available to reach a prefix (*e.g.*, one primary and one backup) and focused on the cases for which multiple backup routes exist. Then, we randomly took one route for each router and prefix and assumed it is the primary one. Finally, for each case, we simulated a remote failure between two ASes in the AS path of the primary route and computed the proportion of backup routes that circumvent the failure, *i.e.*, for which the failure is not in the AS path. Note that we obviously omitted the cases where the AS path of the primary route has only one AS, and we also removed the prepended AS in the AS paths. We ended up with a total of 928 cases.

Figure 6.2 shows a CDF of the proportion of the backup routes that bypass the simulated failures. We omit the cases for which none of the backup routes bypass the failure (592 cases, *i.e.*, 64%). These cases represent most of the cases likely because we take each router individually and do not consider the routes learned by the other border routers and that are not advertised in iBGP. Among the remaining cases (336, *i.e.*, 36%), we can see in Figure 6.2 that for 62% of them, only a subset of the backup paths circumvent the failure. In some cases, only a small fraction of the backup routes circumvent the failure. This result highlights the need for a careful selection of the backup route to use across *all* the available ones, and not just a few of them, as is the case in the current Blink implementation.

6.1.3 Blink is prone to attacks

By eschewing the control plane's careful but slow decision-making in favor of the data plane's speed, Blink is fast. However, this is a significant risk because the data plane's inputs are not tightly controlled like the control plane's: packets drive the data plane. The promise of a fast and automatic reaction to data thus comes with new risks: malicious inputs designed towards negative outcomes [126, 177].

In this section, we illustrate the vulnerabilities of Blink with a case study in which an adversary sends crafted packets from a single host to fake link failures and hijack traffic. Note that this is not the only possible attack against Blink: one could also send crafted packets that are not retransmissions, preventing Blink from rerouting upon actual failures.

Attack. Manipulating Blink may seem trivial since generating TCP retransmissions is easy. While this is an essential ingredient, Blink's sampling strategy necessitates more care. The attacker needs to generate flows that always remain active so that once Blink samples a malicious flow, it keeps monitoring it. Thus, the number of sampled malicious flows increases over time until the sample is reset. Below, we show that an attacker can often ensure that her flows are the majority of the sampled flows for a prefix. Once this is the case, the attacker can easily trick Blink into rerouting traffic, possibly onto a path that she controls. In practice, one can perform this attack by sending the fake TCP retransmissions from a set of hosts that reach the victim prefix via Blink. Observe that the attacker does not need to establish TCP connections with the victim network, making the attack easier to set up and harder to prevent.

Theoretical analysis. Let t_R be the average time a legitimate flow remains sampled. We assume a malicious flow is always active, and thus once being sampled, it is never evicted unless the sample is entirely reset. We define t_B as the frequency at which Blink resets the sample ($t_B = 8.5$ min, by default). Thus, t_B is the attacker's time budget until all the sampled flows are evicted. We define q_m as the fraction of traffic that is malicious. For a particular cell of the array used for sampling, the probability p that it is occupied by a malicious flow at the end of the time budget t_B is $p = 1-(1-q_m)^{(t_B/t_R)}$. Now, consider X as a random variable corresponding to the number of malicious flows monitored across all cells at the end of the time budget, t_B. As each of the n cells acts independently, X is binomially distributed with parameters n and p. We use this distribution to calculate the practicality of the attack.

Figure 6.3 shows the number of monitored malicious flows over time. We consider $t_R = 8.37\ s$, the value computed for one prefix of a CAIDA trace [9] used in Blink's analysis, and $q_m = 0.0525$ (*i.e.*, 5.25% of the flows are malicious). After $200\ s$, there is a high chance that at least 32 monitored flows (*i.e.*, half) are malicious (red lines), enabling a successful attack. With

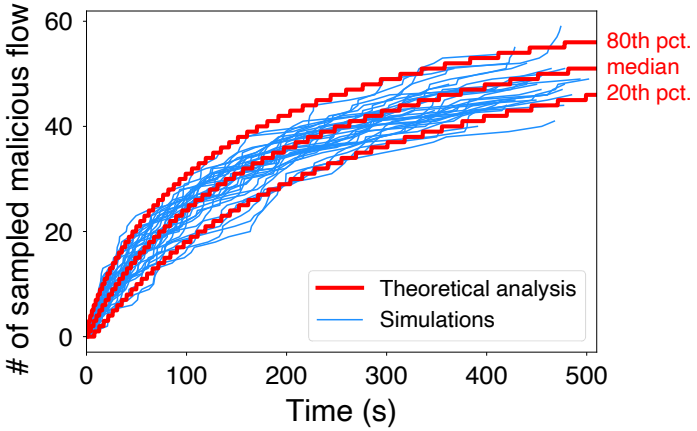


Figure 6.3 Malicious flows sampled by Blink over time ($t_R = 8.37s$, $q_m = 0.0525$). On average, it takes 172 s until the sample contains enough (*i.e.*, 32) malicious flows to allow the attacker to trick the system.

longer t_R, the attack is harder, *i.e.*, requires higher q_m. We analyzed the top-20 prefixes of each CAIDA trace used to evaluate Blink (see Table 5.1) and found that for half of them, the average time a flow remains sampled is $10\ s$ (the median is ∼5 s). The example in Figure 6.3 is therefore representative. While 5% of traffic to a destination is substantial, it is within reach, even for the most popular destinations, using a small botnet.

Experimental results. To confirm our theoretical results, we simulated a network with mininet [114] and the $P4_{16}$ implementation of Blink. We generated 2000 legitimate and 105 malicious flows ($q_m = 0.0525$), and used the same $t_R = 8.37\ s$. The thin blue lines in Figure 6.3 show the results of each experiment. As expected from the theoretical results, half of the sampled flows are malicious after ∼200 s. We did experiments with many values for t_R, and the results always match the theoretical analysis. It confirms that an attacker can manipulate Blink *quickly* and with a *small* amount of traffic.

6.2 Key Ingredients

This section gives an overview of the key components and principles behind Snap and its hardware-software codesign. We show that this codesign avoids Snap being as constrained by the programmable switches limitations as Blink. This will help Snap to address the Blink problems we identify in §6.1.

6.2.1 Involving the controller smartly

By involving the controller, Snap is less limited by the constraints imposed by the programmable switches and can use more advanced, thus more efficient, algorithms. Besides, running tasks on the controller enables saving data-plane resources, which can then be used to monitor more prefixes to improve scalability, monitor more AS links to improve coverage, or run other applications in the switches. However, the drawback of involving the controller is that a round trip between the Snap router and the Snap controller is needed to detect a failure and trigger the rerouting.

Trading a little bit of speed for a significant performance improvement. We illustrate this round trip time by showing a typical deployment of Snap within an AS (Figure 6.4). In this example, a single central controller runs the Snap control-plane functions for all the border routers, which we assume are $P4_{16}$ programmable routers that run Snap. The round trip consists in: *(i)* the Snap routers mirror traffic to the Snap controller, for failure detection (see Section 6.2.2); and *(ii)* when the Snap controller detects a remote failure, it updates one of the Snap router which reroutes traffic to a working backup path restoring connectivity

Fortunately, we expect this round trip time to be low. Indeed, in practice, the controller can be either directly in the switch CPU, in which case we expect only a few microseconds delay, or in a controller nearby the switch (as in Figure 6.4), in which case we expect only a few milliseconds delay. By involving the control plane, Snap is thus obviously (slightly) slower than Blink, but performs significantly better than Blink and is still an order of magnitude faster than the full control-plane-based solutions such as SWIFT.

Smart mirroring. Of course, if all the Snap routers naively mirror all the traffic towards a central controller, it may *(i)* generate an unreasonably high amount of traffic which may overload the network; and *(ii)* overload the Snap controller, which has a lower packet processing rate than the hardware-based routers. Snap solves these issues by mirroring only a fraction of the flows, focusing on the most useful ones, *i.e.*, the active ones. The Snap router performs this flow selection using Blink's Flow Selector, which aims to select a set of active flows and mirror them to the controller (see Figure 6.4). As a result, the amount of mirrored traffic does not depend on the actual volume of traffic forwarded by the Snap router but instead relies on the number of monitored AS links, on the number of flows mirrored per AS link, and on the actual amount of traffic sent by the mirrored flows.

Snap can also limit the total volume of mirrored traffic by performing distributed mirroring over multiple Snap routers in the same AS. Indeed, the traffic going to a destination prefix and traversing two Snap routers within the same AS (*e.g.*, R1 and R2 in Figure 6.4) may follow the same AS path, in which case it makes

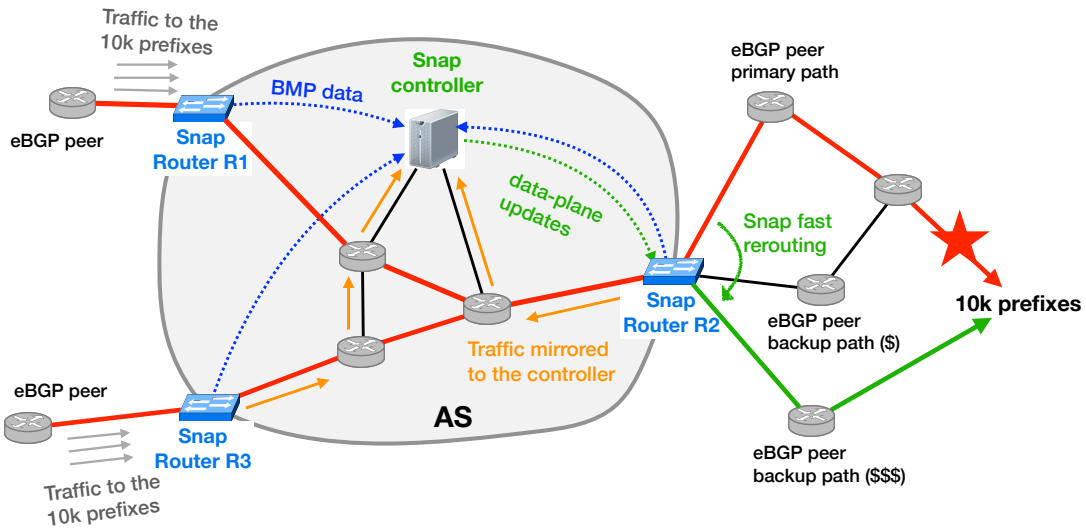


Figure 6.4 Overview of Snap when deployed within in AS. In this example, the Snap controller runs on a server. Snap quickly reroutes the traffic on a working backup path upon the remote failure affecting the traffic to the 10k prefixes.

no difference for the failure detection algorithm to process flows coming from one of the Snap routers or the other. Because the controller thus aggregates the mirrored flows coming from the different Snap routers, each Snap router has to mirror fewer flows, reducing the load on the network and the resources needed by the Snap routers and the Snap controller.

6.2.2 SWIFT failure inference with data-plane signals

The Snap controller uses the mirrored flows to run a failure inference algorithm akin to SWIFT. The key difference is that Snap uses fast data-plane signals (TCP retransmissions) as input, whereas SWIFT uses the slow control-plane signals (BGP updates and withdrawals). Despite this difference, the algorithm is essentially the same. The Snap controller collects BGP information from the Snap routers using, for instance, the BGP Monitoring Protocol [155], and maintains the AS-level graph. In the meantime, the Snap controller monitors TCP flows from the mirrored traffic and computes the proportion of them experiencing retransmissions over time and on a per AS-link basis.

6.2.2.1 Flow selection

To accurately pinpoint a failed AS link, Snap must monitor flows traversing that AS link, *i.e.*, for which the AS path of the routes to the destination prefixes of the flows include this AS link. Besides, and similarly to Blink, Snap needs to monitor active flows, *i.e.*, the ones constantly sending packets. As a result,

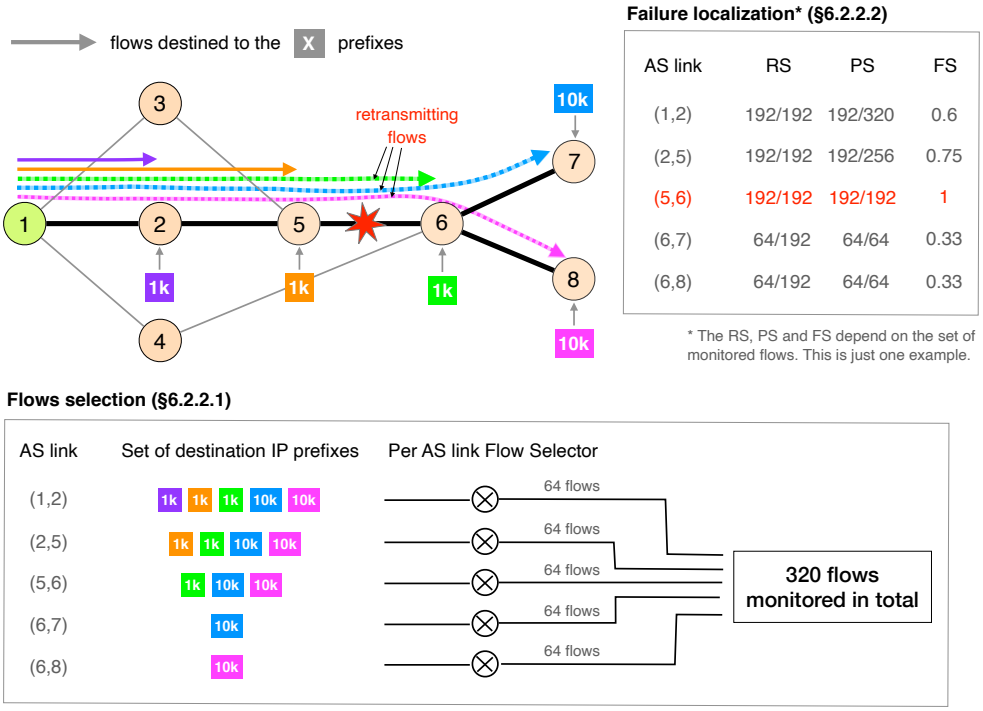


Figure 6.5 Illustration of the flow selection strategy and the failure inference algorithm used by Snap upon the failure of $(5, 6)$.

the Snap flow selection strategy ensures to select at least a certain number of active flows traversing each monitored AS link. This flow selection strategy is made possible by the Snap routers, which use a distinct Flow Selector for every monitored AS link, each selecting and mirroring 64 flows (by default) to the Snap controller.

Example. To illustrate this flow selection strategy, consider the scenario depicted in Figure 6.5. AS2, AS5 and AS6 advertise 1k prefixes each, whereas AS7 and AS8 advertise 10k prefixes each. AS3 and AS4 are only used as backup and are not in the primary paths. Each AS (but the backup ones) advertises a set of prefixes, denoted as P_x for ASX. On this example, Snap thus selects: 64 flows among the flows traversing $(1, 2)$ and that are destined to prefixes in $P_2 \cup P_5 \cup P_6 \cup P_7 \cup P_8$; 64 flows among the flows traversing $(2, 5)$ and that are destined to prefixes in $P_5 \cup P_6 \cup P_7 \cup P_8$; 64 flows among the flows traversing $(5, 6)$ and that are destined to prefixes in $P_6 \cup P_7 \cup P_8$; 64 flows among the flows traversing $(6, 7)$ and that are destined to prefixes in P_7; and finally 64 flows among the flows traversing $(6, 8)$ and that are destined to prefixes in P_8. In total, 320 flows are selected and monitored in this example.

6.2.2.2 Sound inference

For each AS link, Snap computes the proportion of flows experiencing retransmissions over time among the ones traversing the AS link. Snap infers a failure and triggers an analysis of its root cause when it detects a peak of TCP retransmissions for the flows traversing an AS link.

Failure localization. Upon detection of a failure, Snap infers the corresponding failed AS link as the one maximizing the *Fit Score (FS)*, a metric that we also use in SWIFT (see 4.2.1). Let t be the time at which this inference is done. For any link l, the value of FS for l is the *geometric mean* of the *Retransmission Share* (RS) and *Path Share* (PS):

$$FS(l,t) = \sqrt{RS(l,t) * PS(l,t)}$$

RS is the fraction of the flows forwarded over l that experience retransmissions at t over the total number of flows experiencing retransmissions. PS is the fraction of flows experiencing retransmissions and traversing l at t over the total number of flows traversing l at t. More precisely,

$$RS(l,t) = \frac{R(l,t)}{R(t)} \qquad PS(l,t) = \frac{R(l,t)}{R(l,t) + F(l,t)}$$

where $R(l,t)$ is the number of flows traversing l and that are experiencing retransmissions at t; $R(t)$ is the *total* number of flows experiencing retransmissions at t; $F(l,t)$ is the number of monitored flows traversing l at t and that are still working fine. Note that for the RS to be useful, Snap considers all the monitored flows traversing the AS link l when computing $R(l,t)$, and not just the 64 flows selected for that link. If only the 64 flows were considered, then the failed AS link and all the downstream AS links would all have the same RS value.

Example. Figure 6.5 reports possible RS and PS values upon detection of the retransmissions caused by the failure of $(5,6)$. Link $(5,6)$ is the only one with both RS and PS equal to one since all the flows traversing this link, and only those are experiencing retransmissions. In contrast, the PS values for links $(1,2)$ and $(2,5)$ are smaller than 1 (192/320 and 192/256) because some flows traversing these AS links are still working fine. The WS of $(6,7)$ and $(6,8)$ are smaller than one because not all the monitored flows experiencing retransmissions pertain to that link. In the end, $(5,6)$ is therefore correctly inferred as failed.

Correctness. The Snap inference algorithm is always correct under ideal conditions. The following theorem holds.

Theorem 6.1. *Snap inference returns a set of links including the failed link, if it runs when all the flows traversing the failed AS link exhibit retransmissions.*

Proof of Theorem 4.3. Assume that a single link f fails and that the inference algorithm makes a prediction at time t after detecting RTO-induced retransmissions.

We now show that the inference algorithm assigns the highest possible values of both RS and PS to f.

Indeed, all the flows traversing f are affected by the failure and will thus start to retransmit packets: This implies that the number $F(f,t)$ of flows traversing f and still working (*i.e.*, sending packets) at t is 0. Moreover, since the detected retransmissions are all caused by f's failure by hypothesis, all the flows experiencing retransmissions must have crossed f, that is, $R(f,t) = R(t)$. As a consequence, $PS(f,t) = R(l,t)/(R(l,t)+0)$ and $RS(f,t) = R(l,t)/R(l,t)$ are equal to their maximum value 1.

This implies that the fit score of f is the highest possible one. Hence the Snap inference algorithm will return it in the set of failed links. ■

6.2.2.3 Robust to real-world factors

In practice, the ideal conditions assumed in Theorem 6.1 do not always hold. This section shows that the Snap inference algorithm includes mechanisms to make it robust against real-world factors.

Snap quantitative metrics mitigate the effect of noise. Some of the detected retransmissions may be unrelated to the outage but caused by other factors such as congestions. They constitute noise that can negatively affect the accuracy of any inference algorithm. In Figure 6.5, for instance, flows destined to prefixes advertise by `AS2` or `AS5` may also experience retransmissions which distort the FS value. In this example, these unrelated retransmissions would increase the likelihood that the FS of $(1,2)$ and $(2,5)$ is higher than the one of $(5,6)$.

Fortunately, Snap follows Blink's strategy (§5.3.3) to detect retransmissions and only reports *consecutive* retransmissions, *i.e.*, the ones that are likely induced by the TCP retransmission timeout (RTO). The congestion-induced retransmissions are likely to be interleaved by non-retransmissions and are thus (correctly) not detected and not counted in the computation of the FS score. In practice, Snap is thus robust to real-world noise as the level of RTO-induced retransmissions is usually low. Hence, its effect on quantitative metrics such as FS, RS, and PS tends to drop rapidly.

Snap per-AS-link flows mirroring mitigates the effect of skewed traffic distribution. By following Blink's flow selection strategy, the probability that a flow is mirrored and monitored only depends on its packet rate: flows with a high packet rate are more likely to be selected than the others. This is intended

as Snap needs active flows to detect failures. However, to detect many remote outages, Snap not only needs active flows but also needs flows that traverse many of the AS links, thus destined to different destination prefixes advertised by different ASes.

The problem is that Internet traffic is typically skewed and a small portion of the Internet prefixes carry the most traffic [150]. Hence, if the probability of a flow to be monitored only depends on its packet rate, Snap performance would be skewed too: it would work very well on the path towards the top prefixes and not so well on the others. For example, consider the scenario in Figure 6.5 again, and assume that among the 320 selected flows, none of them are destined to the 10k prefixes advertised by AS7, because those destinations are not as popular as the others. In this case, a failure on the link $(6, 7)$ would be off the radar.

To increase Snap coverage, Snap runs Blink's flow selection on a per-AS-link basis. More precisely, for each monitored AS link, Snap mirrors 64 flows (by default) traversing it, regardless of the total number of flows traversing it. In Figure 6.5, it means that Snap mirrors 64 flows for every AS link, including $(6, 7)$, enabling it to detect any possible remote outages in this example.

Snap applies a conservative strategy if failed links cannot be univocally determined. It may happen that Snap cannot precisely distinguish which link has failed. For example, consider the scenario in Figure 6.5 again, and assume that among the 320 selected flows, none of them are destined to prefixes advertised by AS5, which can happen if not many flows (or no flows at all) are destined to prefixes advertised by AS5. In this case, both links $(2, 5)$ and $(5, 6)$ have the same FS score, and Snap cannot precisely determine which link has failed.

Whenever a failed link cannot be univocally determined, Snap inference returns all the links with maximum FS, *i.e.*, both $(5, 6)$ and $(6, 8)$ in the previous example.

Snap can infer concurrent link failures. To cover cases like router failures that affect multiple inter-AS links simultaneously, the inference algorithm used by Snap follows the same strategy as in SWIFT and computes the FS value for sets of links sharing one endpoint. To do that, Snap follows the same algorithm as the one used by SWIFT (see §4.2.3). More precisely, the algorithm aggregates greedily links with a common endpoint (from links with the highest FS to those with the lowest one) until the FS for all the aggregated links does not increase anymore. The fit score FS for any set S of links is computed by extending the definition of RS and PS as follows.

$$RS(S,t) = \frac{\sum\limits_{l \in S} R(l,t)}{R(t)} \qquad PS(S,t) = \frac{\sum\limits_{l \in S} R(l,t)}{\sum\limits_{l \in S} R(l,t) + F(l,t)}$$

The algorithm returns the set of links (potentially, with a single element) with the highest FS value.

To ensure safety (see §6.2.3), for each link inferred, Snap must choose a backup route that does not traverse the common endpoint of the links. This prevents Snap to reroute traffic destined to a prefix towards a backup next-hop that uses another inferred link (because all the inferred links have a common endpoint). Furthermore, by choosing backup paths bypassing a superset of the inferred links, Snap also ensures safety in case the inference algorithm correctly localizes the ASes involved in the outage instead of the precise links.

6.2.3 SWIFT rerouting with P4$_{16}$ variable-length metadata

Snap follows the same rerouting strategy as SWIFT: after localizing the outage, it reroutes traffic on backup paths that circumvent the outage. This section explains how Snap achieves this rerouting.

Data-plane tags with P4$_{16}$ variable-length metadata. Snap encodes in data-plane tags the AS links traversed by each packet as well as the backup next-hop to use should any of them fails (see §4.3). Unlike SWIFT, Snap does not have to compress the data-plane tags in the 48-bit destination MAC address but instead can use the P4$_{16}$ variable-length metadata. P4$_{16}$ metadata greatly simplifies the process of tagging each packet (*e.g.*, Snap does not need to use ARP to tag the packets) and enables Snap to use more bits per prefix to tag packets with more next-hops and AS links, as long as it does not exceed the capacity of the switch.

Besides, the variable-length metadata allows Snap to leverage a tradeoff between the number of monitored prefixes and the system's performance. More precisely, Snap can filter out the less essential prefixes to use more bits for the most important ones and perform better for them. The traffic destined to a non-monitored prefix simply goes to the last stage of the pipeline and is forwarded normally.

Snap is beneficial and safe. As soon as a failure is detected, Snap matches on the data-plane tags (*i.e.*, the metadata) to reroute traffic around the failure. Given that Snap uses the same rerouting strategy as SWIFT, Theorem 4.1 and Theorem 4.2 also hold for Snap, assuming routing stability and reasonable inference (see §4.1.3).

Despite not notifying path changes in the control plane, Snap is thus beneficial and safe. More precisely, Snap causes no forwarding loops and strictly improves Internet-wide connectivity, proportionally to the number of Snap routers.

6.3 Design

This section describes Snap's pipeline and its internal algorithms, design choices, and parameter values. Figure 6.6 gives an overview of the pipeline running in a Snap router, at the control- and the data-plane level. We will use it as an illustration throughout this section.

We divide the pipeline in four parts: the metadata installation (§6.3.1), the smart traffic mirroring (§6.3.2), the failure inference (§6.3.3) and the fast traffic rerouting around the failure (§6.3.4).

6.3.1 Per-packet metadata installation

The first step of the Snap data-plane pipeline consists of the metadata_table attaching a set of metadata to each packet based on their destination IP address.

The SWIFT tags in P4$_{16}$. First, the metadata use_snap allows activating Snap for some particular destination IP prefixes only. Packets destined to a prefix for which Snap is not active (*i.e.*, use_snap is equal to 0) go directly to the last stage of the pipeline and are forwarded normally.

If Snap is activated, the rest of the metadata attached to each packet is similar to the information carried in the data-plane tags used by SWIFT (§4.3). More precisely, the metadata_table sets the primary_next-hop metadata, a value that uniquely identifies the primary next-hop to use. Then, the metadata_table sets, for each AS link traversed by the packet, the link_ID metadata, a value that uniquely identifies he AS link, as well as the backup_next-hop metadata, a value that uniquely identifies the backup next-hop to use if the AS link fails.

The metadata_table is updated by the Snap controller at runtime. The controller maintains a RIB with the BGP routes to compute the primary and backup next-hops. The controller learns the BGP routes either from the BGP peers should it maintain BGP sessions with them or from the information collected with *e.g.*, BMP. The controller also maintains the AS graph, derived from the AS path of the BGP routes, and from which it computes a unique link_ID for every AS link.

Size of the metadata. By default, we follow the SWIFT parameters and consider AS links up to depth four in the AS graph (*i.e.*, position five in the AS path). An analysis on ten full BGP routing tables ($> 800k$ prefixes) collected from RIPE RIS [16] indicates that the highest number of distinct AS links up to depth four is 35,793. We thus reserve 16 bits for the link_ID, allowing Snap to encode $2^{16} = 65536$ AS links. As in SWIFT, Snap reserves 6 bits to encode a next-hop to work with up to $2^6 = 64$ distinct next-hops.

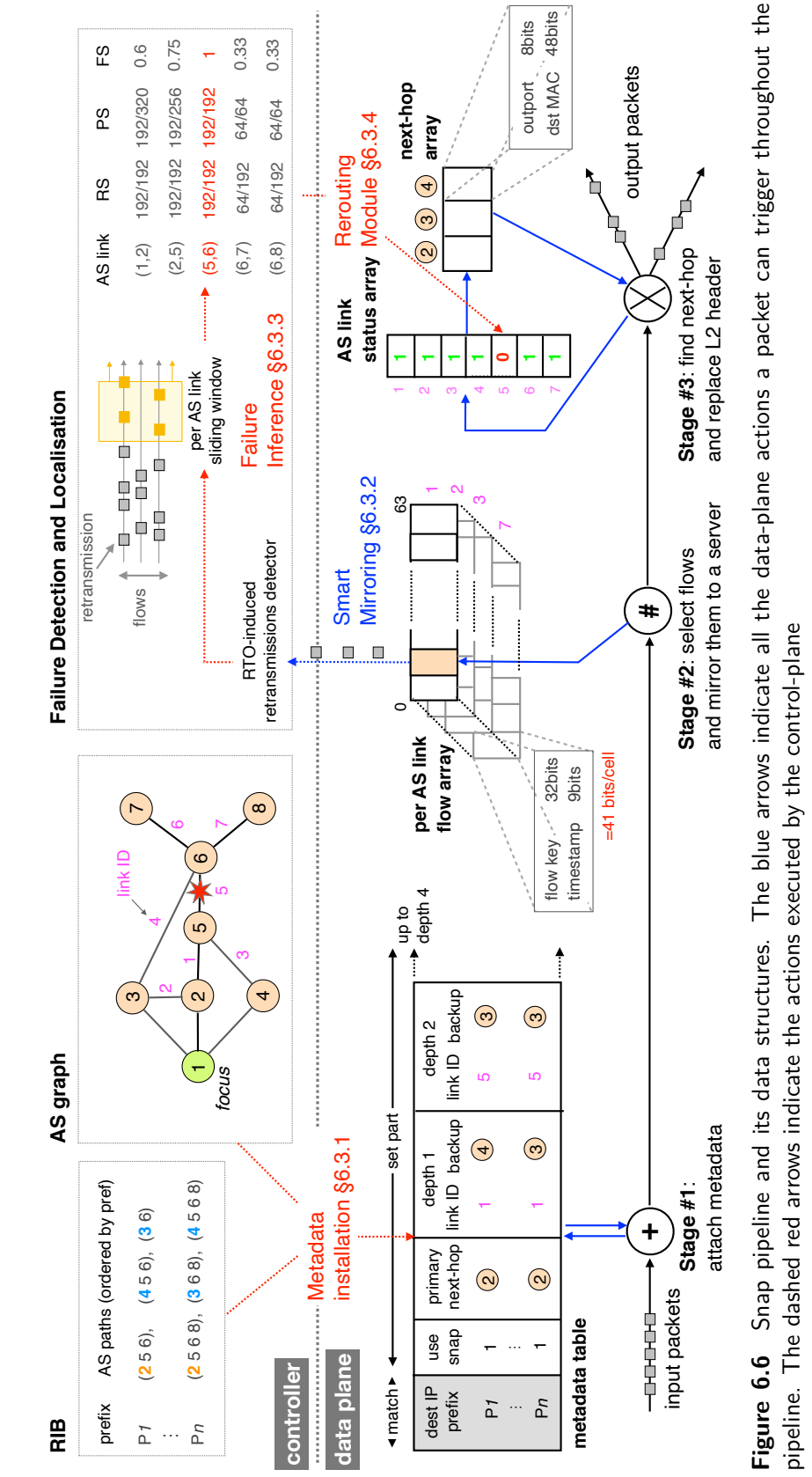


Figure 6.6 Snap pipeline and its data structures. The blue arrows indicate all the data-plane actions a packet can trigger throughout the pipeline. The dashed red arrows indicate the actions executed by the control-plane

6.3.2 Smart mirroring

A fraction of the packets destined to prefixes for which Snap is activated are then mirrored to the Snap controller. The controller runs either directly on the switch control plane, in which case the implementation relies on the copy_to_cpu primitive, or in a remote server, in which case Snap encapsulates each mirrored packet in a new IP header with the IP destination of the server.

Mirroring a distinct set of active flows for every monitored AS link. To decide which flows to mirror, Snap uses the same flow selection strategy as the one used in Blink: it relies on a Flow Selector to detect active flows, *i.e.*, the ones that are sending packets (see §5.3.2). However, Snap selects a set of flows per AS link instead of per destination prefix as in Blink. Snap thus uses a flow_array for each AS link that it monitors. Snap ensures that a flow is only mapped to a single flow_array to select disjoint sets of flows for each AS link. Snap randomly maps a flow to a flow_array, using a hash of the 5-tuple.

Following the Blink implementation, Snap uses 64 cells in each flow_array. Each cell in a flow_array contains a 32-bit flow_key that Snap uses to enforce that only one flow occupies a cell at a given time, as well as a 9-bit timestamp, to detect inactive flows. The eviction timeout, *i.e.*, the time after which a flow is considered inactive if it does not send any packet, is set to two seconds, the default value also used in Blink. We refer the reader to §5.3.2 for a detailed description of our $P4_{16}$-based Flow Selector.

Limiting the volume of traffic mirrored. To limit the volume of traffic mirrored, Snap uses a custom header that only contains the required information to track RTO-induced retransmissions. The custom header includes the destination IP, the flow_id, the timestamp, the TCP sequence number, the TCP payload size, and the previous flow_id (in case the packet triggered the eviction of the flow that occupied its cell in the flow_array).

6.3.3 Detecting and locating failures

The Snap controller uses the mirrored traffic to detect and locate remote outages using the following three steps.

Detecting RTO-induced retransmissions. As in Blink, Snap focuses on consecutive retransmissions because they are likely triggered by the RTO timeout (non-retransmissions more likely interleave the congestion-induced retransmissions). Snap reports a packet as RTO-induced retransmission if it retransmits a part of the full content of the directly preceding packet of the same flow. As in Blink (see §5.3.3), Snap detects RTO-induced retransmissions by comparing the sum of the packet sequence number and payload length to the corresponding sum of the directly preceding packet of the same flow.

Computing the proportion of flows experiencing retransmissions per AS-link and over time. To do that, Snap uses a sliding window for every AS link. Each sliding window reports the number of flows experiencing retransmissions per AS link and within the last 800 milliseconds (same value as in Blink, see §5.3.3). However, unlike Blink, which uses a probabilistic implementation of the sliding window because of the hardware constraints, Snap leverages the resources and flexibility offered by the controller and uses a deterministic – thus more accurate – sliding window, which slides every time Snap detects a RTO-induced retransmission.

Inferring the outage. Finally, Snap runs its outage inference algorithm, inherited from SWIFT (see §6.2.2). When half of the flows experience retransmissions for an AS link, Snap computes the Fit Score FS for every monitored AS link, and reports as failed the one with the highest score.

6.3.4 Fast traffic rerouting around the failure

When the Snap controller detects a failure, it quickly updates the data-plane decisions to reroute all the affected traffic on working backup paths.

Triggering traffic rerouting to working backup paths with a single data-plane update. Snap uses the AS_link_status_array to reroute traffic. This array stores the status of each of the monitored AS links. Each cell only contains a status bit that indicates whether the corresponding AS link is functional (1) or not (0). When the controller detects a remote outage, it updates the AS_link_status_array for the affected AS links and sets their status bit to 0, automatically activating the backup paths for the affected traffic.

Snap looks at the status of every AS link that a packet traverses (attached to the packet as metadata, see §6.3.1) to decide which next-hop to use. If all the traversed AS links are marked as functional, Snap sends the packet to the primary next-hop. On the contrary, if one of them is marked as not working, then the backup next-hop providing a path circumventing the faulty AS link (also attached to the packet as metadata) is used.

Sending the packets out. Within the Snap pipeline, the available next-hops are uniquely identified by a 6-bit integer value (see §6.3.1). The last operation to perform in the pipeline is thus to translate the next-hop identifier into the corresponding outport and destination MAC address. Snap achieves this operation using the next-hop_array, which is updated by the controller at runtime and stores the outport and destination MAC to use for every next-hop.

In addition to rerouting locally to a directly connected next-hop announcing an alternate route, a Snap router could fast-reroute to remote next-hops, potentially at the other side of the network, by using tunnels (*e.g.*, IP or MPLS ones). In this case, the next-hop_array must store the next-hop IP address.

6.4 Implementation

We have implemented the data-plane pipeline of Snap as described in §6.3 in 459 lines of $P4_{16}$ code. Our current implementation detects remote outages up to depth four in the AS graph. For now, we implemented the control plane in $\approx$ 1100 lines of Python code, and use Scapy [8] to sniff the mirrored packets. We also only focus on IPv4, although Snap would work with IPv6 too.

This control-plane implementation is enough for testing Snap on simple scenarios. However, for an actual deployment, we envision implementing a control plane capable of processing higher packet throughput, using C along with frameworks dedicated for fast packet processing such as DPDK [2].

The rest of this section shows that Snap is implementable on real hardware and practicable on real-world scenarios.

6.4.1 Memory usage in a Snap switch

In the metadata_table, for each prefix, Snap uses 8 bits to store the primary next-hop, $4 * 6 = 24$ bits to store the four backup next-hops, and $4 * 16 = 64$ bits to store the first four AS links. In total, activating Snap for a prefix requires 88 bits. Besides that, Snap also uses one flow_array for every monitored AS link. By default, each flow_array requires $(32+9)*64 = 2464$ bits. Finally, the AS_link_status_array needs one bit per monitored AS and the next-hop_array needs $8 + 48$ bits for each next-hop.

Snap memory usage thus depends on two main factors: the number of monitored prefixes and AS links. Given that current programmable switches such as Tofino [5] have tens of megabytes of memory available, we expect Snap switches to work with 100,000 prefixes ($\approx$ 1.1 MB) – an order of magnitude more than Blink– and tens of thousands of AS links (*e.g.*, 10,000 AS links would require $\approx$ 3MB).

6.4.2 The Snap control plane can run in commodity servers

In its current state, the per-packet computations performed by the Snap control-plane are rather basic. Upon reception of a packet, the controller *(i)* uses a radix tree to insert or retrieve information about the corresponding IP prefix; *(ii)* checks if the packet is a retransmission (performed in $O(1)$); and *(iii)* updates the number of flows experiencing retransmissions on the AS links traversed by the packet. When a failure is detected, the controller just iterates over all the AS links, updates the Fit Score for each of them, and returns the one with the highest value.

The challenge for the controller is doing this process in real-time, on general-purpose CPUs. The smart mirroring performed by Snap helps to address this problem because only a few flows are mirrored per AS-link (maximum 64, by default), limiting the volume of traffic that has to be processed by the Snap controller. The following paragraphs show the number of AS links we expect a Snap controller to be able to monitor.

Methodoly. For every traffic trace used in the Blink evaluation (Table 5.1), we computed the packet rate of each flow using SiLK [19], filtered out flows sending less than one packet every two seconds on average (as they are unlikely to be selected by Snap) and computed the average packet rate over all the remaining flows. The highest average packet rate measured for a flow is 91 pkts/s and appears on trace one. Assuming a worse-case scenario where each flow sends 91 pkts/s on average, we can compute the expected volume of traffic received by the controller as a function of the number of monitored AS links.

The Snap controller can monitor thousands of AS links. In our implementation, the custom Snap header used by the mirrored packets contains the original source and destination IP of the packet, the current flow_key as well as the old one (to inform the Snap controller that a flow has been evicted), the TCP payload length and the TCP sequence number (to detect retransmissions). Altogether, this corresponds to 28 bytes, which means that a mirrored packet has a total size of 64 bytes (including the Ethernet and IP header).

For a given AS link, the Snap controller thus receives $64 * 91 = 5824$ pkts/s, which corresponds to 372KB/s. As the current commodity servers are now able to process >100Mpps and >100Gbit/s of traffic using fast packet processing frameworks such as DPDK (*e.g.*, [60]), we expect the Snap controller to be able to monitor thousands of AS links.

We note that the mirroring scheme can be adjusted to mirror less traffic at the cost of more calculations in the Snap switch. For instance, the RTO-induced retransmissions can be tracked in the Snap switch (like in Blink) and only the detected retransmissions are then mirrored to the Snap controller.

6.5 Evaluation

In this section, we first evaluate the Snap outage inference in §6.5.1 and show that it is accurate in many scenarios exhibiting real-world characteristics. We then show Snap in action in §6.5.2.

6.5.1 Snap failure inference is accurate

The results in the Blink evaluation (§5.5.1.1) suggest that Snap can detect outages with good accuracy and quickly, despite the noise. We now supplement these results with simulations in a controlled environment and with outage ground truth to show that Snap is not only able to detect outages but can also locate them with high accuracy, despite the Internet traffic often being skewed [150].

Methodology. We implement a Python-based discrete-event simulator that *(i)* builds an AS-level topology and assumes one AS runs Snap; *(ii)* simulates a flow selection on every AS link assuming either a uniform traffic distribution or a skewed one, where a large proportion of the flows target a small proportion of the ASes; *(iii)* simulates a failure of an AS link l by failing flows traversing l one by one until Snap detects a failure; and *(iv)* runs the Snap failure inference to verify whether Snap correctly identifies l as the failed link.

We measured the performance of Snap over many different scenarios that correspond to real-life situations. More precisely, we used different AS-level topologies that are built from the BGP data collected by RouteViews BGP peers [132] (we only consider peers receiving more than 800k prefixes). For each peer, we first build the AS-level topology as seen from the BGP updates collected at a precise time in December 2020, and assume that the BGP peer runs Snap. We then rank the AS links based on the number of prefixes traversing them and assign 64 flows to each of the top-1000 AS links among the ones that do not exceed depth four in the AS graph, following the settings used in §6.4. Finally, we run 1000 distinct simulations for each peer wherein each of them we simulate a failure of a different AS link among the top-1000 AS links. Overall, we end up with a total of 100,000 simulations. Note that even though we only assign flows to the top-1000 AS links, Snap can detect an outage on *any* AS link in the AS graph.

We made each simulation twice. First assuming a uniform traffic distribution, where the destination AS of a flow is picked randomly and uniformly. Second, to simulate the effect of the skewed traffic distribution on the flow selection strategy used by Snap, we ensure that the destination ASes of the selected flows follows a Zipf distribution with a distribution parameter set to 1.8, which aligns with previous observations [150].

Snap failure inference is precise enough in 98.5% of the cases. We divide the inferences into three categories. *Correct* inferences return a set of links including the failed link. *Satisfactory* inferences return a set of links adjacent to the failed one. While not correct, these inferences ensure that Snap reroutes traffic to backup routes that do not traverse any of the failed links, as Snap picks a backup route that does not traverse both endpoints of the inferred failed link (see §6.2.2.3). We label these inferences as satisfactory as they enable Snap

		Traffic distribution	
		Uniform	Skewed
Inference	*Correct*	55.0%	55.8%
	Satisfactory	43.7%	42.7%
	Wrong	1.4%	1.5%

Table 6.1 Accuracy of the Snap outage inference with a uniform and skewed traffic distribution.

to restore connectivity. Finally, *wrong* inferences return a set of links that are remote from the failed link and thus do not enable Snap to safely reroute traffic.

Table 6.1 shows the accuracy of the Snap outage inference. With a uniform or a skewed traffic distribution, Snap outage inference returns the correct link in the majority of the cases. Among the correct inferences and with a uniform (resp. skewed) traffic distribution, 98% (resp. 93%) returned a single link – the correct one. With a uniform traffic distribution, only 1.4% of the inferences are wrong. The accuracy of the inferences with a skewed traffic distribution is only slightly lower (1.5% of wrong inferences). Besides, our results indicate that the accuracy of the inferences is high for every BGP peer. In the worst-case scenario and with a skewed traffic distribution, a BGP peer running the Snap outage inference makes wrong inferences in 2.4% of the cases only. These results highlight that the Snap outage inference enables fast and safe traffic rerouting in the majority of the cases.

6.5.2 Snap in action

We illustrate Snap on a virtual network built with p4-utils [59] and following the example used in Figure 6.5. `AS1` runs our Snap implementation described in §6.4. We start 100 TCP flows from `AS1` towards each AS in the topology (except `AS3` and `AS4`), making a total of 500 flows. Each flow sends packets at a regular interval, randomly selected between 1 s and 3 s, so as to have active and inactive flows according to the Snap settings. After a few seconds, we fail the link $(5, 6)$, a remote link for `AS1`, and report in Figure 6.7 the volume of traffic received by the destination ASes over time.

Upon the outage, the Snap router in `AS1` detects the outage and correctly localizes the event on the link $(5, 6)$. Despite `AS3` being the preferred backup path, Snap reroutes the traffic destined to `AS6`, `AS7` and `AS8` to `AS4`, which uses a path circumventing the outage and thus restores connectivity. In our experiment, the measured downtime is thus only 1.05 s. The traffic towards `AS2` and `AS5` is not affected by the outage, and thus not rerouted by Snap.

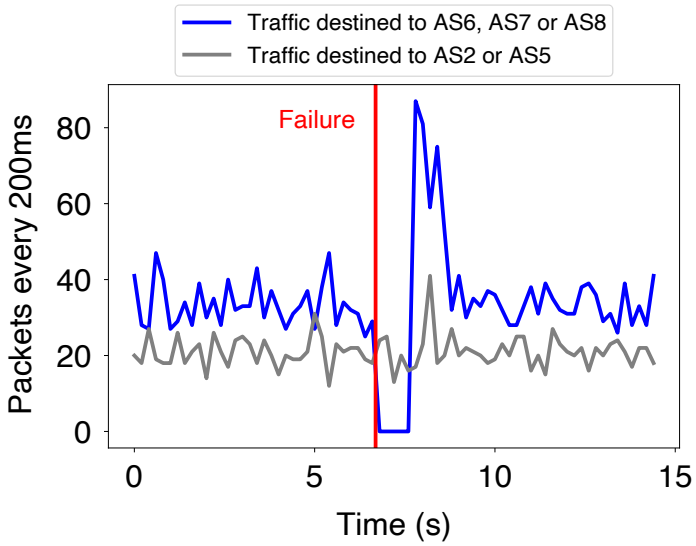


Figure 6.7 Illustration of Snap on the example depicted in Figure 6.5. Upon the failure of the link $(5, 6)$, the measured downtime is only 1.05 s for the traffic destined to AS6, AS7 or AS8.

6.6 Future work and open problems

Snap combines the best of SWIFT and Blink. Yet, we believe that its hardware-software codesign makes possible further improvements. In this section, we present several ideas on how to improve various aspects of Snap. We believe these ideas are a good starting point to solve the problem of partial outages and the security issues pointed out in §6.1, among others.

6.6.1 Better inference algorithms

In its current form, Snap follows the Blink approach and infers an outage when the proportion of flows experiencing retransmissions exceeds a certain threshold (50%, by default). We showed in §6.1 that this simple algorithm misses many outages, especially the partial ones, *i.e.*, those affecting only a subset of the traffic traversing the faulty AS link (*e.g.*, due to load-balancing).

With the decisions being taken in the controller, we envision using more accurate algorithms, *e.g.*, capable of detecting these partial outages and inferring which flows are affected by the outage and which flows are not. In the following paragraphs, we present ideas to improve the inference algorithm.

Anomaly detection algorithms. Remote outages often coincide with a narrow peak of retransmissions in the time series built from the reported RTO-induced retransmissions (see §5.1.1). We believe such patterns can be detected with anomaly detection algorithms. Looking at a particular pattern instead of using a

single threshold allows Snap to detect more events, especially the ones exhibiting a weaker signal, such as partial failures.

Because our framework aims to quickly reroute traffic upon outages, the anomaly detection algorithm must be fast and run in real time. The PELT changepoint detection algorithm appears to be a possible candidate to detect those peaks of retransmissions [102]. However, particular attention should be given when configuring the sensitivity, a typical parameter used in many anomaly detection algorithms. Indeed, a low sensitivity might result in many outages being off the radar whereas a high sensitivity might result in many false positives (*e.g.*, due to congestions or path changes).

Multiple inputs. Other signals can help to infer a remote outage. For instance, a remote outage is likely to coincide with a drop in the total throughput, as all the TCP flows quickly stop sending new packets. On the contrary, if most of the TCP flows are still sending new packets, there is likely no outage.

Anomaly detection algorithms on multivariate time series are potential candidates to detect outages from multiple inputs. Supervised learning algorithms such as decision trees or neural networks could also be potential candidates, but they require historical data of past failures for the learning phase, which is not always easy to obtain. We could build a synthetic dataset by generating traffic following real-world patterns and simulating failures, as we did to evaluate Blink (§5.5.1). However, this is challenging as traffic may exhibit different patterns depending on the routers on which it is observed.

Besides improving the performance of the inference, using other signals such as the throughput enables the inference to be less dependent on TCP, which can be useful in the future as TCP appears to be less used with time and replaced by other transport protocols, often encrypted, such as QUIC [113, 148].

Active probing. Active probing can also be used in conjunction with passive observations, to validate the outage inference and assess the severity of the failure, which is useful information to decide whether it is beneficial to reroute the traffic or not. For instance, upon the inference of an outage, the controller can send ICMP probes to verify if the destination prefixes assessed as failed are indeed not reachable anymore.

However, active probing has some drawbacks: *(i)* it may inject a lot of packets in the network, especially if many prefixes are assessed as failed; *(ii)* it delays the outage inference, as the controller needs to wait for the ICMP replies or the expiration of a timer; and *(iii)* the controller needs to know destination IP addresses in remote networks that respond to ICMP probes, which is not something straightforward to learn. As a result, active probing must be used with parsimony and must be accompanied by passive observations, as done in previous works on Internet failure detection such as Hubble [100].

6.6.2 Robust to adversarial inputs

There is clearly value in extracting actionable network control information from data-plane signals, but for a system like Snap to be practical, we must guarantee that this does not compromise security. We now show possible countermeasures that could prevent malicious users to trick a data-driven system such as Snap.

Using independent signals. Most attacks against Snap are possible because the system blindly trusts the inputs it receives, and the inputs are vulnerable to manipulations. A possible countermeasure is to improve the input's robustness against manipulations using many independent signals. For instance, Snap could monitor the RTT distribution over a large number of flows, approximate the expected RTO distribution upon a failure, and use it to distinguish between actual failures and malicious events. Manipulating Snap would then require an attacker to know the RTT distribution of the legitimate flows forwarded by the Snap router, information that is hard to obtain for an attacker.

The challenge when implementing these countermeasures is that it is difficult to assess whether two signals are truly independent and no adversary can manipulate both of them. Besides, combining signals increases the decision time and thus conflicts with immediate reactions to events required in the case of Snap.

Obfuscating control logic. Successful attacks require a model of the control logic used in a data-driven system. Obfuscating this logic, or varying it over time, can thus hinder attacks. This security-by-obscurity method can form part of a defense-in-depth approach.

In the Snap case, a simple approach is to use random and variable seeds when mapping a flow into a cell of the Flow Selector, along with a flow eviction strategy not only based on a fixed timer. Hence, it becomes harder for an attacker to derive the volume of traffic to send and the estimated time needed to corrupt the system.

Detecting malicious behavior. The signals used for failure detection often follow some patterns, such as the typical day and night or the weekday and weekend cycles. By looking at historical data, these patterns can often be learned and used to detect suspicious events.

Malicious behavior can also be detected using passive measurements or active probing. For instance, a sudden peak of traffic is likely to be malicious if it is not accompanied by a path change at the control- or the data-plane level.

However, when designing such countermeasures, it is important to find the sweet spot for maximizing the detection of malicious behavior given the cost of modifying the application and its impact on the decision time.

6.6.3 Higher-quality inputs

Unsurprisingly, the accuracy of the failure inference depends on the quality of the inputs. Currently, the input of the failure inference is a set of 64 flows per monitored AS links, selected based on their activity. In the following paragraphs, we show possible ideas to improve the quality of the inputs.

Better in-network flow selection. The Snap switch follows the same approach as in Blink to mirror active flows: a flow is evicted and replaced by a new one if it does not send a packet during at least 2 seconds. While useful, this flow selection strategy is far from being optimal. For instance, a flow sending exactly two packets per second will remain selected, preventing more active flows from being selected instead.

In the future, we thus envision improving the flow selection strategy and derive the activity of a flow based on the last n observed packets for that flow, with $n > 1$. For instance, selecting a flow based on its estimated average number of packets sent per second would already be beneficial. Previous works on real-time flow classification can also supplement our flow selection strategy [32]. For instance, flows belonging to certain types of traffic can be taken out of the selection process early on because they are likely to be inactive in the future.

However, the main challenge is that the flow selection is done in the programmable switches, which have very limited resources. Yet, it is certainly possible to implement more advanced algorithms than the simple eviction strategy used by Blink and Snap. For instance, Busse-Grawitz et al. implement random forests in $P4_{16}$ to perform real-time and in-network packet classification [42]. Observe that the cost, in terms of resources, to select a flow when using more advanced algorithms is likely to be higher than with our current simple selection strategy. Hence, it is important to find a good balance between the number and the quality of the selected flows.

Controller-driven flow selection. Constantly mirroring 64 (by default) flows for each of the monitored AS links to detect and locate the rather infrequent (but very damaging) remote outages can be too excessive and resource-hungry. We thus envision exploring how the mirroring can be adapted over time, according to the situation. In particular, as the inference algorithms are executed in the controller, we envision letting the Snap controller adapt the flow selection by updating some of the registers in the memory of the Snap switch.

A possible idea is to adapt the flow selection to quickly zoom in on a part of the AS graph or on a particular AS link upon detection of a suspicious event. Imagine, for instance, that the Snap controller has detected a high level of retransmissions on a particular AS link. The controller could then adapt the flow selection performed in the Snap switch to mirror more flows traversing the potential faulty AS link to the controller, for further and more precise analysis. However, one challenge with this reactive approach is to zoom in quickly, as

traffic can be lost during the extra time needed to zoom in and infer the outage.

Finally, we believe that letting the Snap controller evict flows in the Flow Selector of the Snap switch can be beneficial. We see two motivations for that. First, the controller can compute more per-flow features than the switch (*e.g.*, the distribution of the packets' inter-arrival time) and thus can better infer which flow is likely to help for detecting remote outages. If the controller detects that a selected flow is not useful for failure inference, it could evict it directly from the Flow Selector, instead of waiting for the switch to do so. Second, the controller can evict flows that exhibit undesirable or suspicious behavior. For instance, the controller can evict a flow that sends a particularly high number of packets per second, a behavior that is more of a burden for the controller.

7

Conclusions

In this dissertation, we shed light on the problem of connectivity disruptions upon remote Internet outages, which is the consequence of the slow Internet convergence. We show that it is possible to quickly restore connectivity upon remote outages, although BGP, as a path-vector protocol, does not explicitly notify the status of the links in the network but instead slowly propagates per-prefix reachability information.

Our solution relies on *(i)* an inference algorithm that quickly and accurately infers and locates remote outages from control- and data-plane signals; and *(ii)* a rerouting scheme which relies on pre-computed data-plane tags to quickly reroute all the affected traffic towards backup paths avoiding the failure, regardless of the number of IP prefixes for which traffic must be rerouted. The final result of our work is Snap, a fast reroute framework for remote Internet outages that is deployable on existing devices, provides forwarding guarantees, complies with routing policies configured by the network operators, and is incrementally deployable without requiring any modification to BGP. Snap reduces the time a router takes to restore connectivity from minutes (with BGP) to a few seconds.

Snap, which we present in §6, is based on the key principles used by SWIFT and Blink, which we describe in §4 and §5, respectively. Overall, the contribution of this dissertation is fourfold.

We highlight the effects of remote outages on users' connectivity. We supplement the previous works focusing on Internet convergence with an extensive measurement study highlighting the effects of remote outages on users' connectivity. More precisely, we measure the frequency of remote outages and estimate their resulting downtime. We confirm our observations with an anonymous survey that we conducted with 73 network operators and with experiments that we performed on a recent Cisco router.

We make fast rerouting upon remote Internet outages possible. We show that it is possible to find a good balance between speed and accuracy when inferring remote outages from control- and data-plane signals. More precisely, our inference algorithm quickly infers the location of the failure at the AS level – a level of precision sufficient to guarantee a safe, beneficial, and fast traffic rerouting within reasonable assumptions.

We develop a fast reroute framework for remote Internet outages. We implement the key components of our fast reroute framework and manage to deploy it on existing devices. To do that, we leverage network programmability (*i.e.*, SDN and P4) and rely on a hardware-software codesign, where the controller runs the outage inference while the switches take care of carefully mirroring active flows to the controller and fast rerouting the traffic using our pre-computed data-plane tags upon the inference of an outage.

We extensively evaluate our framework. We evaluate our framework on actual devices (*e.g.*, a Tofino switch) and with real and synthetic inputs to show its practicality in many different real-world situations. Our results indicate that our framework can infer and locate a vast majority of the remote outages and restore connectivity within a few seconds only.

Concluding remarks

Despite the problem of connectivity disruptions caused by the slow Internet convergence being an extensively studied topic already, the design of a framework that network operators can deploy locally without cooperation with other networks and that enables routers to quickly restore connectivity upon remote Internet outages was still an unsolved problem.

In this dissertation, we identify the key challenges for fast traffic rerouting upon remote Internet outages and present a set of key findings and fundamental guidelines that help to solve this problem. Of course, as we discussed in §6.6, there are still many problems that remain open. Yet, we hope that our work will inspire future fast outage detection and fast traffic rerouting solutions.

Credits

The work presented in this dissertation would not have been possible without the contributions of many people that I would like to acknowledge in this section. First of all, I would like to thank all my co-authors and, more precisely, Laurent Vanbever, Stefano Vissicchio, and Alberto Dainotti for their insightful ideas and feedback and their substantial contributions to the writing of the SWIFT and Blink paper. I would like to give credit to Stefano Vissicchio for sketching the SWIFT proof of correctness. Furthermore, I would like to thank Edgar Costa Molero, who devoted a lot of effort to the Blink evaluation and, more precisely, on generating the synthetic traces we used to evaluate the performance of Blink (§5.5). I also would like to thank Maria Apostolaki, who came up with the idea of detecting connectivity disruptions from the sequence number of consecutive TCP packets of the same flow; Stephan Keck, who worked on identifying possible attacks against Blink and produced Figure 6.3 during his semester thesis; and Markus Happe, who programmed the FPGA boards we used in §2.3.1 to measure the FIB update time of a Cisco router. Finally, I would like to thank Rüdiger Birkner, Tobias Bühler, Edgar Costa Molero, Roland Meier, and Ahmed El-Hassany for their valuable feedback on this dissertation's draft.

Bibliography

[1] CIDR REPORT for 16 Feb 21. https://www.cidr-report.org/as2.0/.

[2] Data Plane Development Kit. https://www.dpdk.org/.

[3] ExaBGP. The BGP swiss army knife of networking. https://github.com/Exa-Networks/exabgp.

[4] INSIDER. Amazon's one hour of downtime on Prime Day may have cost it up to $100 million in lost sales. https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7?IR=T.

[5] Intel Tofino, P4-programmable Ethernet Switch ASIC that Delivers Better Performance at Lower Power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[6] Largest Companies by Market Cap. https://companiesmarketcap.com/.

[7] Quagga Routing Suite. quagga.net.

[8] Scapy: Packet crafting for Python2 and Python3. https://scapy.net/.

[9] The CAIDA UCSD Anonymized 2013/2014/2015/2016/2018 Internet Traces. http://www.caida.org/data/passive/passive_2013_dataset.xml.

[10] The UCSD Network Telescope. https://www.caida.org/projects/network_telescope/.

[11] A Border Gateway Protocol 4 (BGP-4). RFC 1771, 1995. URL: https://rfc-editor.org/rfc/rfc1771.txt, doi:10.17487/RFC1771.

[12] CNN - Time Warner Cable comes back from nationwide Internet outage. https://money.cnn.com/2014/08/27/media/time-warner-cable-outage/index.html, 2014.

[13] Outage at Amsterdam internet hub affects much of Netherlands. https://nltimes.nl/2015/05/13/outage-amsterdam-internet-hub-affects-much-netherlands, 2015.

[14] The caida as relationships dataset. http://www.caida.org/data/active/as-relationships/, 2016.

[15] Cisco Umbrella 1 Million. https://blog.opendns.com/2016/12/14/cisco-umbrella-1-million/, 2016.

[16] RIPE RIS Raw Data. https://www.ripe.net/data-tools/stats/ris/, 2016.

[17] Network Simulator 3. https://www.nsnam.org/, 2018.

[18] P4 behavioral model. https://github.com/p4lang/behavioral-model, 2018.

[19] SiLK. https://tools.netsa.cert.org/index.html, 2018.

[20] Y. Afek, A. Bremler-Barr, and S. Schwarz. Improved bgp convergence via ghost flushing. *IEEE Journal on Selected Areas in Communications*, 2004. doi:10.1109/JSAC.2004.836002.

[21] Jay Aikat, Jasleen Kaur, F. Donelson Smith, and Kevin Jeffay. Variability in tcp round-trip times. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, 2003. doi:10.1145/948205.948241.

[22] Michael Alan Chang, Thomas Holterbach, Markus Happe, and Laurent Vanbever. Supercharge me: Boost router convergence with sdn. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015. doi:10.1145/2785956.2790007.

[23] Rodrigo Aldecoa, Chiara Orsini, and Dmitri Krioukov. Hyperbolic graph generator. *Computer Physics Communications*, 2015.

[24] Johanna Amann and Robin Sommer. Viable protection of high-performance networks through hardware/software co-design. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks; Network Function Virtualization*, 2017. doi:10.1145/3040992.3041003.

[25] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. *SIGOPS Oper. Syst. Rev.*, 2001. doi:10.1145/502059.502048.

[26] Maria Apostolaki, Ankit Singla, and Laurent Vanbever. Performance-driven internet path selection. In *Proceedings of the 2021 ACM Symposium on SDN Research*, 2021.

[27] Alia Atlas and Alex D. Zinin. Basic Specification for IP Fast Reroute: Loop-Free Alternates. RFC 5286, 2008. URL: https://rfc-editor.org/rfc/rfc5286.txt, doi:10.17487/RFC5286.

[28] David Barrera, Laurent Chuat, Adrian Perrig, Raphael M. Reischuk, and Pawel Szalachowski. The SCION internet architecture. *Communications of the ACM*. URL: /publications/papers/SCION-CACM.pdf, doi:10.1145/3085591.

[29] BBC. France emergency service number disrupted after network outage. https://www.bbc.com/news/world-europe-57341526, 2021.

[30] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 1958.

[31] Zied Ben Houidi, Mickael Meulle, and Renata Teixeira. Understanding slow bgp routing table transfers. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, 2009. doi:10.1145/1644893.1644935.

[32] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. Traffic classification on the fly. *SIGCOMM CRR*, 2006. doi:10.1145/1129582.1129589.

[33] Olivier Bonaventure, Clarence Filsfils, and Pierre Francois. Achieving sub-50 milliseconds recovery upon bgp peering link failures. *IEEE/ACM Transactions on Networking*, 2007. doi:10.1109/TNET.2007.906045.

[34] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CRR*, 2014. doi:10.1145/2656877.2656890.

[35] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM*, 2013. doi:10.1145/2486001.2486011.

[36] Anat Bremler-Barr, Edith Cohen, Haim Kaplan, and Yishay Mansour. Predicting and bypassing end-to-end internet service degradations. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, 2002. doi:10.1145/637201.637248.

[37] Pietro Bressana, Noa Zilberman, Dejan Vucinic, and Robert Soulé. Trading latency for compute in the network. In *Proceedings of the Workshop on Network Application Integration/CoDesign*, 2020. doi:10.1145/3405672.3405807.

[38] Broadcom. Broadcom Trident4, High-Capacity StrataXGS Trident4 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[39] Stewart Bryant, Clarence Filsfils, Stefano Previdi, Mike Shand, and Ning So. Remote Loop-Free Alternate (LFA) Fast Reroute (FRR). RFC 7490. https://rfc-editor.org/rfc/rfc7490.txt. doi:10.17487/RFC7490.

[40] Bush, Randy and Allman, Mark and Patel, Keyur and Pitta Venkatachalapathy, Balaji and Pandey, Manoj and Paxson, Vern and Pelsser, Cristel and Kern, Ed. Happy Packets: Some Initial Results. Presented at NANOG 2004. https://archive.psg.com/121009.nag-bgp-tcp.pdf.

[41] Bush, Randy and Griffin, Timothy and Mao, Morley and Purpus, Eric and Stutsbach, Dan. TCP Behavior of BGP . https://archive.psg.com/040524.nanog-happy.pdf.

[42] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pforest: In-network inference with random forests, 2019. arXiv:1909.05680.

[43] Matthew Caesar, Lakshminarayanan Subramanian, and Randy H. Katz. Towards localizing root causes of bgp dynamics. Technical report, EECS Department, University of California, Berkeley, 2003. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2003/6364.html.

[44] Enke Chen, Tony J. Bates, and Ravi Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456, 2006. https://rfc-editor.org/rfc/rfc4456.txt. doi:10.17487/RFC4456.

[45] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. Purr: A primitive for reconfigurable fast reroute: Hope for the best and program for the worst. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019. doi:10.1145/3359989.3365410.

[46] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. Traffic data repository at the wide project. In *USENIX ATEC'00*. http://dl.acm.org/citation.cfm?id=1267724.1267775.

[47] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: An overlay testbed for broad-coverage services. *SIGCOMM CRR*, 2003. doi:10.1145/956993.956995.

[48] Cisco. The Zettabyte Era Officially Begins (How Much is That?). https://blogs.cisco.com/sp/the-zettabyte-era-officially-begins-how-much-is-that.

[49] François Clad, Pascal Mérindol, Jean-Jacques Pansiot, Pierre Francois, and Olivier Bonaventure. Graceful convergence in link-state ip networks: A lightweight algorithm ensuring minimal operational impact. *IEEE/ACM Transactions on Networking*, 2014. doi:10.1109/TNET.2013.2255891.

[50] Benoît Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, 2004. https://rfc-editor.org/rfc/rfc3954.txt. doi:10.17487/RFC3954.

[51] Clemens Mosig and Randy Bush and Cristel Plesser and Thomas C Schmidt and Matthias Wählisch. Route Flap Damping in the Wild?! https://labs.ripe.net/author/clemens_mosig/route-flap-damping-in-the-wild/.

[52] Ricardo Bennesby da Silva and Edjard Souza Mota. A survey on approaches to reduce bgp interdomain routing convergence delay on the internet. *IEEE Communications Surveys Tutorials*, 2017. doi:10.1109/COMST.2017.2722380.

[53] Alberto Dainotti, Roman Amman, Emile Aben, and Kimberly C. Claffy. Extracting benefit from harm: Using malware pollution to analyze the impact of political and geophysical events on the internet. *SIGCOMM CRR*, 2012. doi:10.1145/2096149.2096154.

[54] Datacenterfrontier. Cost of Data Center Outages. https://datacenterfrontier.com/white-paper/cost-data-center-outages/.

[55] Amogh Dhamdhere, Renata Teixeira, Constantine Dovrolis, and Christophe Diot. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proceedings of the 2007 ACM CoNEXT Conference*, 2007. doi:10.1145/1364654.1364677.

[56] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1959.

[57] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. PCC: Re-architecting congestion control for consistent high performance. In *USENIX NSDI*, 2015.

[58] A. Atlas Ed and Google Inc. U-turn Alternates for IP/LDP Fast-Reroute, 2006. https://tools.ietf.org/html/draft-atlas-ip-local-protect-uturn-03.

[59] Edgar Costa Molero. Extension to mininet that makes p4 networks easier to build. https://github.com/nsg-ethz/p4-utils.

[60] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, 2015. doi:10.1145/2815675.2815692.

[61] Nick Feamster, David G. Andersen, Hari Balakrishnan, and M. Frans Kaashoek. Measuring the effects of internet path faults on reactive routing. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2003. doi:10.1145/781027.781043.

[62] Nick Feamster and Jennifer Rexford. Why (and how) networks should run themselves. *CoRR*, 2017. http://arxiv.org/abs/1710.11583. arXiv:1710.11583.

[63] Anja Feldmann, Olaf Maennel, Z. Morley Mao, Arthur Berger, and Bruce Maggs. Locating internet routing instabilities. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2004. doi:10.1145/1015467.1015491.

[64] Clarence FilsFils. BGP Convergence in much less than a second. NANOG 40, 2007. https://archive.nanog.org/meetings/nanog40/presentations/ClarenceFilsfils-BGP.pdf.

[65] Clarence Filsfils, Pierre Francois, Mike Shand, Bruno Decraene, Jim Uttaro, Nicolai Leymann, and Martin Horneffer. Loop-Free Alternate (LFA) Applicability in Service Provider (SP) Networks. RFC 6571, 2012. https://rfc-editor.org/rfc/rfc6571.txt. doi:10.17487/RFC6571.

[66] Clarence FilsFils, Pradosh Mohapatra, John Bettink, Pranav Dharwadkar, Peter De Vriendt, Yuri Tsier, Virginie Van Den Schrieck, Olivier Bonaventure, and Pierre Francois. Bgp prefix independent convergence (pic) technical report. Technical report, Cisco, 2011. http://www.cisco.com/en/US/prod/collateral/routers/ps5763/bgp_pic_technical_report.pdf.

[67] Romain Fontugne, Cristel Pelsser, Emile Aben, and Randy Bush. Pinpointing delay and forwarding anomalies using large-scale traceroute measurements. In *Proceedings of the 2017 Internet Measurement Conference*, 2017. doi:10.1145/3131365.3131384.

[68] Pierre Francois and Olivier Bonaventure. Avoiding transient loops during the convergence of link-state routing protocols. *IEEE/ACM Transactions on Networking*, 2007. doi:10.1109/TNET.2007.902686.

[69] Pierre Francois, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving sub-second igp convergence in large ip networks. *SIGCOMM CRR*, 2005. doi:10.1145/1070873.1070877.

[70] Lixin Gao. On inferring autonomous system relationships in the Internet. *IEEE/ACM Transactions on Networking*, 2001. doi:10.1109/90.974527.

[71] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. *IEEE/ACM Transactions on Networking*, 2001. doi:10.1109/90.974523.

[72] J.J. Garcia-Lunes-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking*, 1993. doi:10.1109/90.222913.

[73] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research*, 2017. doi:10.1145/3050220.3050228.

[74] V. Giotsas, A. Dhamdhere, and k. claffy. Periscope: Unifying looking glass querying. In *Passive and Active Network Measurement Workshop (PAM)*, 2016.

[75] M. Goyal, M. Soperi, E. Baccelli, G. Choudhury, A. Shaikh, H. Hosseini, and K. Trivedi. Improving convergence speed and scalability in ospf: A survey. *IEEE Communications Surveys Tutorials*, 2012. doi:10.1109/SURV.2011.011411.00065.

[76] T. G. Griffin and B. J. Premore. An experimental analysis of bgp convergence time. In *Proceedings Ninth International Conference on Network Protocols (ICNP)*, 2001. doi:10.1109/ICNP.2001.992760.

[77] A. Guillot, R. Fontugne, P. Winter, P. Merindol, A. King, A. Dainotti, and C. Pelsser. Chocolatine: Outage detection for internet background radiation. In *Network Traffic Measurement and Analysis Conference (TMA)*, 2019. http://icube-publis.unistra.fr/4-GFWM19.

[78] Krishna P. Gummadi, Harsha V. Madhyastha, and David Wetherall. Improving the reliability of internet paths with one-hop source routing. In *6th Symposium on Operating Systems Design & Implementation (OSDI)*, 2004. https://www.usenix.org/conference/osdi-04/improving-reliability-internet-paths-one-hop-source-routing.

[79] Arpit Gupta, Robert MacDavid, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. An industrial-scale software defined internet exchange point. In *USENIX NSDI*, 2016.

[80] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A Software Defined Internet eXchange. In *SIGCOMM 2014*. doi:10.1145/2740070.2626300.

[81] Nikola Gvozdiev, Brad Karp, and Mark Handley. LOUP: The principles and practice of intra-domain route dissemination. In *USENIX NSDI*, 2013.

[82] F. Hauser, M. Häberle, Daniel Merling, S. Lindner, V. Gurevich, Florian Zeiger, R. Frank, and M. Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *ArXiv*, 2021.

[83] Urs Hengartner, Sue Moon, Richard Mortier, and Christophe Diot. Detection and analysis of routing loops in packet traces. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, 2002. doi:10.1145/637201.637217.

[84] Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunstrom. Measuring latency variation in the internet. In *ACM CoNEXT '16*. doi:10.1145/2999572.2999603.

[85] Thomas Holterbach, Tobias Bühler, Tino Rellstab, and Laurent Vanbever. An open platform to teach how the internet practically works. *SIGCOMM CRR*, 2020. doi:10.1145/3402413.3402420.

[86] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *USENIX NSDI*, 2020.

[87] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017. doi:10.1145/3102980.3103005.

[88] Polly Huang, Anja Feldmann, and Walter Willinger. A non-instrusive, wavelet-based approach to detecting network performance problems. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, 2001. doi:10.1145/505202.505229.

[89] Internet Society. State of IPv6 Deployment 2018. https://www.internetsociety.org/wp-content/uploads/2018/06/2018-ISOC-Report-IPv6-Deployment.pdf.

[90] Internet World Stats. Internet Growth Statistics. https://www.internetworldstats.com/emarketing.htm.

[91] IoT-Analytics. State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time. https://www.helpnetsecurity.com/2019/05/23/connected-devices-growth/.

[92] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. *SIGCOMM CRR*, 2013. doi:10.1145/2534169.2486019.

[93] Umar Javed, Italo Cunha, David Choffnes, Ethan Katz-Bassett, Thomas Anderson, and Arvind Krishnamurthy. Poiroot: Investigating the root cause of interdomain path changes. In *Proceedings of the ACM SIGCOMM*, 2013. doi:10.1145/2486001.2486036.

[94] Hao Jiang and Constantinos Dovrolis. Passive estimation of tcp round-trip times. *SIGCOMM CRR*, 2002. doi:10.1145/571697.571725.

[95] Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. Unleashing the potential of data-driven networking. 2017. doi:10.1007/978-3-319-67235-9_9.

[96] Jiazeng Luo, Junqing Xie, Ruibing Hao, and Xing Li. An approach to accelerate convergence for path vector protocol. In *Global Telecommunications Conference (GLOBECOM)*, 2002. doi:10.1109/GLOCOM.2002.1189059.

[97] John P John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus routing: The Internet as a distributed system. In *USENIX NSDI*, 2001.

[98] Dave Katz and David Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, 2010. https://rfc-editor.org/rfc/rfc5880.txt. doi:10.17487/RFC5880.

[99] Ethan Katz-Bassett, Harsha V. Madhyastha, Vijay Kumar Adhikari, Colin Scott, Justine Sherry, Peter van Wesep, Thomas Anderson, and Arvind Krishnamurthy. Reverse traceroute. In *USENIX NSDI*, 2010.

[100] Ethan Katz-Bassett, Harsha V. Madhyastha, John P. John, David Wetherall, and homas Anderson. Studying black holes in the internet with hubble. In *USENIX NSDI*, 2008.

[101] Ethan Katz-Bassett, Colin Scott, David R. Choffnes, Ítalo Cunha, Vytautas Valancius, Nick Feamster, Harsha V. Madhyastha, Thomas Anderson, and Arvind Krishnamurthy. Lifeguard: Practical repair of persistent route failures. *SIGCOMM CRR*, 2012. doi:10.1145/2377677.2377756.

[102] R. Killick, P. Fearnhead, and I. A. Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 2012. doi:10.1080/01621459.2012.737745.

[103] Christos Kozanitis, John Huber, Sushil Singh, and George Varghese. Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system. In *Proceedings of the 29th Conference on Information Communications*, 2010.

[104] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 2015. doi:10.1109/JPROC.2014.2371999.

[105] Dmitri V. Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *CoRR*, 2010. http://arxiv.org/abs/1006.5169.

[106] Nate Kushman, Srikanth Kandula, and Dina Katabi. Can You Hear Me Now?!: It Must Be BGP. *ACM SIGCOMM CCR*, 2007. doi:10.1145/1232919.1232927.

[107] Nate Kushman, Srikanth Kandula, Dina Katabi, and Bruce M Maggs. R-BGP: Staying connected in a connected world. In *USENIX NSDI*, 2007.

[108] C. Labovitz, A. Ahuja, R. Wattenhofer, and S. Venkatachary. The impact of internet policy and topology on delayed routing convergence. In *Proceedings IEEE INFOCOM.*, 2001. doi:10.1109/INFCOM.2001.916775.

[109] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed internet routing convergence. *SIGCOMM CRR*, 2000. doi:10.1145/347057.347428.

[110] Craig Labovitz, G. Robert Malan, and Farnam Jahanian. Internet routing instability. *SIGCOMM CRR*, 1997. doi:10.1145/263109.263151.

[111] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. Achieving convergence-free routing using failure-carrying packets. *SIGCOMM CRR*, 2007. doi:10.1145/1282427.1282408.

[112] Raul Landa, Lorenzo Saino, Lennert Buytenhek, and Joao Taveira Araujo. Staying alive: Connection path reselection at the edge. In *USENIX NSDI*, 2021.

[113] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade,

Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM*, 2017. doi:10.1145/3098822.3098842.

[114] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *ACM SIGCOMM Hotnets 2010*.

[115] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *USENIX NSDI*, 2016.

[116] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *ACM CoNEXT*, 2016. doi:10.1145/2999572.2999609.

[117] Junda Liu, Baohua Yan, Scott Shenker, and Michael Schapira. Data-driven network connectivity. In *ACM SIGCOMM HotNets*, 2011. doi:10.1145/2070562.2070570.

[118] Jean-Romain Luttringer, Quentin Bramas, Cristel Pelsser, and Pascal Mérindol. A fast-convergence routing of the hot-potato. *CoRR*, 2021. https://arxiv.org/abs/2101.09002. arXiv:2101.09002.

[119] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *ACM SIGCOMM*, 2017. doi:10.1145/3098822.3098843.

[120] Z. Morley Mao, Randy Bush, Timothy G. Griffin, and Matthew Roughan. Bgp beacons. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, 2003. doi:10.1145/948205.948207.

[121] Zhuoqing Morley Mao, Ramesh Govindan, George Varghese, and Randy H. Katz. Route flap damping exacerbates internet routing convergence. *SIGCOMM CRR*, 2002. doi:10.1145/964725.633047.

[122] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. Characterization of failures in an operational ip backbone network. *IEEE/ACM Trans. Netw.*, 2008. doi:10.1109/TNET.2007.902727.

[123] Mashable. Amazon.com went down for about 20 minutes, and the world freaked out. https://mashable.com/2016/03/10/amazon-is-down-2/.

[124] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on load distribution and the role of programmable switches. *SIGCOMM CRR*, 2019. doi:10.1145/3314212.3314216.

[125] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CRR*, 2008. doi:10.1145/1355734.1355746.

[126] Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla, and Laurent Vanbever. (self) driving under the influence: Intoxicating adversarial network inputs. In *ACM HotNets*, 2019. doi:10.1145/3365609.3365850.

[127] Pascal Merindol, Pierre David, Jean-Jacques Pansiot, Francois Clad, and Stefano Vissicchio. A fine-grained multi-source measurement platform correlating routing transitions with packet losses. *Computer Communications*, 2018.

[128] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. Hardware-accelerated network control planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, 2018. doi:10.1145/3286062.3286080.

[129] John Moy. OSPF Version 2. RFC 2328, 1998. doi:10.17487/RFC2328.

[130] W.B. Norton. *The Internet Peering Playbook: Connecting to the Core of the Internet*. 2011.

[131] NPL. Open, High-Level language for developing feature-rich solutions for programmable networking platforms. https://nplang.org/.

[132] University of Oregon. Route Views Project, 2016. www.routeviews.org/.

[133] Ricardo Oliveira, Beichuan Zhang, Dan Pei, Rafit Izhak-Ratzin, and Lixia Zhang. Quantifying path exploration in the internet. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, 2006. doi:10.1145/1177080.1177116.

[134] Open Networking Foundation. OpenFlow Switch Specification, 2012. https://opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf.

[135] P4.org's API Working Group. P4Runtime. https://p4.org/p4-runtime/.

[136] P. Pan, G. Swallow, and A. Atlas. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. RFC 4090, 2005.

[137] Sonia Panchen, Neil McKee, and Peter Phaal. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176, 2001. doi:10.17487/RFC3176.

[138] Vern Paxson. End-to-end routing behavior in the internet. *SIGCOMM CRR*, 1996. https://doi.org/10.1145/248157.248160.

[139] Dan Pei, Matt Azuma, Dan Massey, and Lixia Zhang. Bgp-rcn: improving bgp convergence through root cause notification. *Computer Networks*, 2005. doi:https://doi.org/10.1016/j.comnet.2004.09.008.

[140] Dan Pei, Xiaoliang Zhao, Lan Wang, Daniel Massey, Allison Mankin, Shyhtsun Wu, and Lixia Zhang. Improving bgp convergence through consistency assertions. 2002. doi:10.1109/INFCOM.2002.1019337.

[141] Cristel Pelsser, Olaf Maennel, Pradosh Mohapatra, Randy Bush, and Keyur Patel. Route flap damping made usable. In Neil Spring and George F. Riley, editors, *Passive and Active Measurement*, 2011.

[142] Simon Peter, Umar Javed, Qiao Zhang, Doug Woos, Thomas Anderson, and Arvind Krishnamurthy. One tunnel is (often) enough. In *ACM SIGCOMM*, 2014. doi:10.1145/2619239.2626318.

[143] Shahrooz Pouryousef, Lixin Gao, and Arun Venkataramani. Towards logically centralized interdomain routing. In *USENIX NSDI*, 2020.

[144] Lin Quan, John Heidemann, and Yuri Pradkin. Trinocular: Understanding internet reliability through adaptive probing. In *ACM SIGCOMM*, 2013. doi:10.1145/2486001.2486017.

[145] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with c-bgp. *IEEE Network Magazine of Global Internetworking*, 2005.

[146] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, 2006. doi:10.17487/RFC4271.

[147] M. Roughan, W. Willinger, O. Maennel, D. Perouli, and R. Bush. 10 Lessons from 10 Years of Measuring and Modeling the Internet's Autonomous Systems. *IEEE Journal on Selected Areas in Communications*, 2011.

[148] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. *A First Look at QUIC in the Wild*. 2018. doi:10.1007/978-3-319-76481-8_19.

[149] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. RFC 6298, 2011. doi:10.17487/RFC6298.

[150] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging zipf's law for traffic offloading. *ACM SIGCOMM CCR*, 2012. doi:10.1145/2096149.2096152.

[151] Donnie Savage, James Ng, Steven Moore, Donald Slice, Peter Paluch, and Russ White. Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP). RFC 7868, 2016. doi:10.17487/RFC7868.

[152] Dominik Schatzmann, Simon Leinen, Jochen Kögel, and Wolfgang Mühlbauer. Fact: Flow-based approach for connectivity tracking. In *Passive and Active Measurement*, 2011.

[153] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *ACM SIGCOMM*, 2017. doi:10.1145/3098822.3098853.

[154] Aaron Schulman and Neil Spring. Pingin' in the rain. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, 2011. doi:10.1145/2068816.2068819.

[155] John Scudder, Rex Fernando, and Stephen Stuart. BGP Monitoring Protocol (BMP). RFC 7854, 2016. doi:10.17487/RFC7854.

[156] A. Shah, R. Fontugne, E. Aben, C. Pelsser, and R. Bush. Disco: Fast, good, and cheap outage detection. In *Network Traffic Measurement and Analysis Conference (TMA)*, 2017. doi:10.23919/TMA.2017.8002902.

[157] S. Shakkottai, N. Brownlee, A. Broido, and k. claffy. The RTT distribution of TCP flows on the Internet and its impact on TCP based flow control. Technical report, Cooperative Association for Internet Data Analysis (CAIDA), 2004.

[158] Mike Shand and Stewart Bryant. IP Fast Reroute Framework. RFC 5714, 2010. doi:10.17487/RFC5714.

[159] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SOSR'17*. doi:10.1145/3050220.3063772.

[160] Jared Smith, Kyle Birkeland, Tyler McDaniel, and Max Schuchard. Withdrawing the bgp re-routing curtain: Understanding the security impact of bgp poisoning through real-world measurements. 2020. doi:10.14722/ndss.2020.24240.

[161] Joao Luis Sobrinho, Laurent Vanbever, Franck Le, and Jennifer Rexford. Distributed Route Aggregation on the Global Network. In *ACM CoNEXT*, 2014.

[162] Ashwin Sridharan, Sue B. Moon, and Christophe Diot. On the Correlation Between Route Dynamics and Routing Loops. In *ACM IMC, 2003*.

[163] Lakshminarayanan Subramanian, Matthew Caesar, Cheng Tien Ee, Mark Handley, Morley Mao, Scott Shenker, and Ion Stoica. Hlp: A next generation inter-domain routing protocol. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2005. doi:10.1145/1080091.1080095.

[164] Swisscom. The secure, high-speed Internet under your control. https://www.swisscom.ch/en/business/enterprise/offer/wireline/scion.html.

[165] The P4 Language Consortium. P4 16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[166] Roscoe Timothy. The end of internet architecture. In *ACM HotNets*, 2006.

[167] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. California Fault Lines: Understanding the Causes and Impact of Network Failures. In *ACM SIGCOMM, 2010*.

[168] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to route. In *ACM HotNets*, 2017. doi:10.1145/3152434.3152441.

[169] Kannan Varadhan, Ramesh Govindan, and Deborah Estrin. Persistent route oscillations in inter-domain routing. *Computer Networks*, 2000. doi:10.1016/S1389-1286(99)00108-5.

[170] VentureBeat. 5-minute outage costs Google $545,000 in revenue. https://venturebeat.com/2013/08/16/3-minute-outage-costs-google-545000-in-revenue/.

[171] Curtis Villamizar, Ravi Chandra, and Dr. Ramesh Govindan. BGP Route Flap Damping. RFC 2439, 1998. doi:10.17487/RFC2439.

[172] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. *ACM SIGCOMM CCR*, 2006. doi:10.1145/1151659.1159928.

[173] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central control over distributed routing. In *ACM SIGCOMM, 2015*.

[174] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central control over distributed routing. In *ACM SIGCOMM*, 2015. doi:10.1145/2785956.2787497.

[175] Feng Wang, Zhuoqing Morley Mao, Jia Wang, Lixin Gao, and Randy Bush. A Measurement Study on the Impact of Routing Events on End-to-end Internet Path Performance. In *ACM SIGCOMM, 2006*.

[176] Lan Wang, Xiaoliang Zhao, Dan Pei, Randy Bush, Daniel Massey, Allison Mankin, S. Felix Wu, and Lixia Zhang. Observation and analysis of bgp behavior under stress. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, 2002. doi:10.1145/637201.637231.

[177] Jiarong Xing, Wenqing Wu, and Ang Chen. Architecting programmable data plane defenses into the network with fastflex. In *ACM HotNets*, 2019. doi:10.1145/3365609.3365860.

[178] Zhiyuan Xu, Jian Tang, Jingsong Meng, Weiyi Zhang, Yanzhi Wang, Chi Harold Liu, and Dejun Yang. Experience-driven networking: A deep reinforcement learning based approach. 2018. doi:10.1109/INFOCOM.2018.8485853.

[179] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *ACM SIGCOMM*, 2017. doi:10.1145/3098822.3098854.

[180] Ming Zhang, Chi Zhang, Vivek Pai, Larry Peterson, and Randy Wang. Planetseer: Internet path failure monitoring and characterization in wide-area services. In *USENIX OSDI*, 2004.

[181] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*, 2020. doi:10.1145/3387514.3406214.

Thomas Holterbach

born in July 1991, French

Main research interests

Internet routing, architecture and security

Network traffic measurements and analysis

Software-defined networking

Programmable network hardware

Education

2016–2021	Doctoral Studies in Computer Science Supervised by Prof. Laurent Vanbever ETH Zurich, Switzerland
2012–2014	Master in Computer Science Specialized in Networks and Embedded Systems University of Strasbourg, France
2009–2012	Bachelor in Computer Science University of Strasbourg, France
2009	Baccalauréat (French secondary school diploma)

Experience

2015–2019	Doctoral Student and Scientific Assistant Supervised by Prof. Laurent Vanbever Networked Systems Group ETH Zurich, Switzerland
2018	Visiting Scholar (3 months) Supervised by Dr. Stefano Vissicchio Computer Science Department University College London, UK
2016	Junior Researcher (6 months) Supervised by Dr. Alberto Dainotti Center for Applied Internet Data Analysis University of California San Diego, CA, USA

2014	Research Intern (6 months) Supervised by Dr. Cristel Pelsser and Randy Bush Internet Initiative Japan, Tokyo, Japan
2013-2014	Research Intern (9 months) Supervised by Dr. Pascal Merindol Network Research Team ICube laboratory, Strasbourg, France