Bringing Path Diversity to the Endhosts using Software Defined Networks and IPv6 Segment Routing

Mathieu Jadin

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

October 2022

ICTEAM
UCLouvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Charles **Pecheur** (Chair)	UCLouvain/ICTEAM, Belgium
Pr. Olivier **Bonaventure** (Supervisor)	UCLouvain/ICTEAM, Belgium
Pr. Per **Hurtig**	Karlstad University, Sweden
Pr. Ramin **Sadre**	UCLouvain/ICTEAM, Belgium
Pr. Stefano **Vissicchio**	University College London, United Kingdom

Bringing Path Diversity to the Endhosts using Software Defined Networks and IPv6 Segment Routing

by Mathieu Jadin

ICTEAM
UCLouvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was partially supported by the F.R.S-FNRS.

Preamble

Since the eve of the Internet, performance and robustness are the main goal to its evolution. In parallel, the number of users and machines increased, increasing the challenge of providing performance and robustness. In tackling robustness, network operators learned that every single thing can fail in a network. Failures can be caused by human errors or by software. The age of a component can cause it to malfunction. Natural disasters such as flood or earthquakes can damage equipment. Denial of Service attacks can render part of a network too loaded to actually be used.

Because everything can fail, network operators avoid keeping a single point of failure when designing their networks. They duplicate links and routers to be able to cope with at least one failure. When a failure occurs, the impact for the users is limited. An alternative is available so that network connectivity is ensured.

This good practice has another positive consequence. When nothing fails, a variety of paths exist to reach a given destination over the Internet. As the number of connected devices continues to increase, these additional paths represent an interesting resource to increase performance at a lower cost.

Researchers investigated how to load balance traffic through different paths to provide good performance. They tried to improve the control of network operators on their network and run complex optimization algorithm to route the traffic. This is called Traffic Engineering. These techniques harbor good average results but consider aggregates of connections over extended periods of time. Indeed, aggregation is crucial to prevent optimization problems to becoming too complex. Other solutions extended the endhosts protocols to be able to use multiple paths. Each data flow can grade its performance of any path and choose the best paths for its use case. These multipath protocols require support on both ends, and usually have limited access to the path diversity because of the lack of communication between the network and the endhosts. We claim that IPv6 Segment Routing [Fil+20] is a promising technology to give endhosts partial control of their paths.

This thesis designs strategies to allow the network and the endhosts to collaborate and use this collaboration to reap path diversity benefits while using single-path protocols. We focus on networks regrouping many endhosts: typically enterprise networks and data centers. We assume the trust of the endhosts allows them to ask for a path or choose between different paths.

This trust is achievable for IT-controlled endhosts: laptops given to employees without an admin access or servers. In particular, the main contributions of this thesis are the following.

- *An architecture for collaboration between a network controller and endhosts with IPv6 Segment Routing and DNS.*

  Chapter 2 introduces collaboration between the network and the endhosts by proposing an SDN-like architecture: the Software Resolved Network (SRN) architecture. Like OpenFlow-based SDN solutions, SRNs enable the network operators to specify policies that control the network paths that are used by applications. For this, Software Resolved Networks build upon the IPv6 Segment Routing architecture (SRv6) and the DNS protocol. Applications use the DNS as a signalling protocol to request the creation of end-to-end network paths. Those paths are computed by a controller that interacts with the DNS resolver and returns the selected path to the requesting application as a segmented SRv6 path.

  There are two important differences between SRNs and traditional SDNs. First, SRNs enable the applications to directly interact with the controller to specify path requirements like delay or bandwidth. Second, SRNs do not require per-flow state in all network nodes. The controller installs state on the access routers but not on the core routers. This improves the scalability of SRNs.

  This work was carried out in collaboration with David Lebrun who implemented the core of the controller and the path selection process. For the sake of completeness, his contribution is explained in this thesis.

- *A near-optimal traffic engineering algorithm for IPv6 Segment Routing through column generation.*

  In Chapter 3 we contribute to network-based optimization algorithms that can be used in our SRN architecture. We focus on the development of efficient traffic engineering algorithms that leverage the unique characteristics of Segment Routing. More precisely, we propose a new scalable approach that relies on column generation [DL05]: CG4SR. Moreover, it can provide strong guarantees on the quality of the solutions by computing a lower-bound that is in most cases tighter than the one obtained with a multicommodity flow relaxation.

  This work was carried out in collaboration with François Aubry who designed the algorithm solving the pricing problem of the column generation formulation. This algorithm chooses the most promising paths

to consider. For the sake of completeness, his contribution is explained in this thesis.

- *Studying and extending TCP modularity in the Linux kernel.*

  Chapter 4 takes a broader look at the modularity of the TCP stack. Through kernel updates, applications were able to configure more and more the TCP behavior to their needs, using the socket API or system-wide configuration (i.e., `sysctl` in Linux). However, this did not address the main issue: kernel development is a slow process and the aggregation of all the TCP extensions makes it even harder to include more changes.

  To allow more modularity, Brakmo [Bra17] introduced eBPF in the Linux kernel to dynamically program the TCP stack. We use eBPF to implement TPC. This addition of eBPF was the first step towards a truly modular implementation of TCP and many steps were taken afterwards. We survey the progress of the Linux stack towards a truly extensible TCP, where most of its optional features would be injected from user space through eBPF. We also improve this modularity through three new features and illustrate through concrete use cases.

- *Allowing TCP connections to recover themselves from remote network failures.*

  Chapter 5 extends the SRN architecture. We empower the trusted end-hosts even more in the path selection process while allowing applications to benefit from our solution even if they are unaware of SRN. We link the flow information of TCP to perform a path selection.

  TCP considers the network layer as a black box. All the packets sent by a host follow a path chosen by the network. If a TCP source detects severe packet losses due to link failure or congestion, it simply waits before retransmitting the lost packets hoping that after some idle time the router buffers will have drained or routing protocols will have converged. As networks contain many redundant paths, this temporal reaction is not the only possibly one.

  We advocate for an end-to-end approach where a TCP sender can explore alternate paths when it experiences severe losses on its current path. We propose the TCP Path Changer (TPC). TPC's design is oriented at providing high responsiveness by rerouting traffic.

- *Allowing TCP connections to dynamically explore several paths based on*

network performance.

Also in Chapter 5, we use TPC to realize measurement-informed source routing. We devise techniques to support online performance-based path selection. In designing our schemes, we grapple with various challenges pertaining to stability and performance. We show that machinery from the theory of online learning (namely, multi-armed bandit theory [VBW15]) is naturally applicable to this context and can be employed to address these challenges. To demonstrate the feasibility of our scheme, we perform path selection for delay-sensitive TCP connections.

The evaluations in Chapter 2 and Chapter 5 are emulation-based. This makes them easily reproducible because they do not require specific setup or hardware. However, to fully demonstrate these contributions' performance in an unstable environment, a deployment on the Internet is required. These evaluations are left for future work.

Bibliographic notes

Conferences and Journals

1. D. Lebrun, M. Jadin, F. Clad, C. Filsfils, and O. Bonaventure. "Software resolved networks: Rethinking enterprise networks with ipv6 segment routing". In: *Proceedings of the Symposium on SDN Research*. ACM. 2018, p. 6.

2. M. Jadin, F. Aubry, P. Schaus, and O. Bonaventure. "CG4SR: Near Optimal Traffic Engineering for Segment Routing with Column Generation". In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE. 2019, pp. 1333–1341.

3. M. Jadin, Q. De Coninck, L. Navarre, M. Schapira, and O. Bonaventure. "Leveraging eBPF to make TCP path-aware". In: *IEEE Transactions on Network and Service Management* (2022).

Workshops

M. Jadin, O. Tilmans, M. Mawait, and O. Bonaventure. "Educational Virtual Routing Labs with IPMininet". In: *ACM SIGCOMM Education Workshop*. 2020.

Magazines

O. Bonaventure, Q. De Coninck, F. Duchêne, A. Gego, M. Jadin, F. Michel, M. Piraux, C. Poncin, and O. Tilmans. "Open educational resources for computer networking". In: *ACM SIGCOMM Computer Communication Review* 50.3 (2020), pp. 38–45.

Posters and Demos

F. Duchene, M. Jadin, and O. Bonaventure. "Exploring various use cases for IPv6 Segment Routing". In: *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. ACM. 2018, pp. 129–131.

Acknowledgments

First, I would like to thank my advisor, Prof. Olivier Bonaventure who was key to the success of my PhD. His guidance, a savvy mix of wisdom, encouragement and eccentric ideas, allowed me to grow as a researcher. I am sincerely grateful for the amount of time we spent discussing research together.

This thesis is the results of collaborations with other persons. I would like to thank my co-authors: Fabien Duchêne, Florentin Rochet, François Aubry, François Clad, Louis Navarre, Michael Schapira, Olivier Tilmans, Pierre Schaus, and Quentin De Coninck. It's thanks to their insight, hard work, and our discussions that we managed to publish the papers that this thesis is based on.

I would like to thank my former coworkers that shared an office with me: Fabien Duchêne, François Michel, Gorby Kabasele, Louis Navarre, Maxime Piraux, Nicolas Rybowski, Olivier Tilmans, Quentin De Coninck, Thomas Wirtgen, Tom Barbette, Raziel Carvajal Gómez, and Viet Hoang Tran. More than once, they brightened the mood through casual or work discussions (e.g., during the cleverly named cafec).

I am very thankful to my former colleagues of INGI for the department's warm atmosphere. Some of them became or already were friends: Anthony Gégo, Alexandre Dubray, Axel Legay, Benoît Duhoux, François Michel, Guillaume Derval, Nicolas Laurent, Quentin De Coninck, Rosana Veroneze, Raziel Carvajal Gómez, and Vanessa Maons. I am especially thankful to my friends from our "rpg club" for all the time that we spent together: Charles Thomas, Eduard Baranov, Fabien Duchêne, Gorby Kabasele, Hélène Verhaeghe, Ludovic Taffin, Marie-Marie van der Beek, Olivier Goletti, Pierre Reinbold. Our time together really helped me getting through the confinement that we faced a few years ago.

I would like to thank the staff from INGI: Vanessa Maons, Sophie Renard, Chantal Poncin, Pierre Reinbold, Nicolas Detienne, Anthony Gego, Ludovic Taffin for their availability that helped me conduct my research in the best possible conditions.

I want to express my gratitude to my family, especially my mother Pénélope, that supported me and helped me pursue my studies. I also owe a special thank you to my friends: Amandine Willame, Cyril de Vogelaere, Guillaume Moyson, Nicolas Dublet-Meis, and Quentin Meyers for their support. Many thanks go as well to all the people that have been close to me during these last years. They all contributed to this thesis in their own way.

Finally, I wish to express a heartful thank you to my boyfriend Gaëtan. We met only six months ago and yet, his kind words gave me the strength to go through tough times and to complete this thesis.

Mathieu

Contents

Background 1

In this chapter, we cover the concepts and the protocols of networking that are necessary to understand the thesis. A reader familiar with them can dive into the next chapters. Section 1.1 presents IPv6 and its addressing. In Section 1.2, we cover packet forwarding based on shortest path routing. Section 1.7 highlights a logically centralized routing architecture: the Software Defined Networks. Section 1.3 and Section 1.4 present, respectively, the transport protocols TCP and UDP. Section 1.5 covers the DNS architecture that assigns human-readable names to IP addresses. Finally, Section 1.9 concludes this chapter.

1.1 Internet Protocol version 6 (IPv6)

At the very basis of the Internet, the Internet Protocol, or IP, plays one of the most important roles. As in a postal system, it carries the source and destination of any data packet. Based on these pieces of information, the Internet finds the destination machine and forwards the packet to it. Destinations and sources are identified by an address, an IP address. Just as postal addresses, IP addresses are hierarchical. Typically, the first part of the address, identifies a group of machines connected to the same entity (Proximus, Google, Facebook, Cloudflare, UCLouvain,...). The middle part identifies a part of the network of the entity (called subnet) and the final part identifies the machine in this subnet.

Two versions of IP coexist: IP version 4 (IPv4) [DAR81] and IP version 6 (IPv6) [DH17]. The main drive to the existence of IPv6 is the exhaustion of the IPv4 address space. IPv4 addresses are encoded using 32 bits, thus limiting the Internet to $2^{32} \approx 4$ billions publicly accessible devices. Foreseeing the issue, IPv6 was designed with addresses encoded on 128 bits. The acceptance of IPv6 has increased through years and in this thesis we focus on IPv6 and do not consider IPv4. Indeed, we believe that this protocol is doomed to disappear.

The format of the IPv6 header attached to each packet is detailed in Figure 1.1. The version is always 6 for IPv6. It is used to differentiate IPv4 and IPv6 packets. The traffic class is used in some networks to give a priority to a given class of traffic. Solutions such as DiffServ [BBC06] use that field. The flow label [Ama+11] is an attempt to speed up the process of flow identification. It

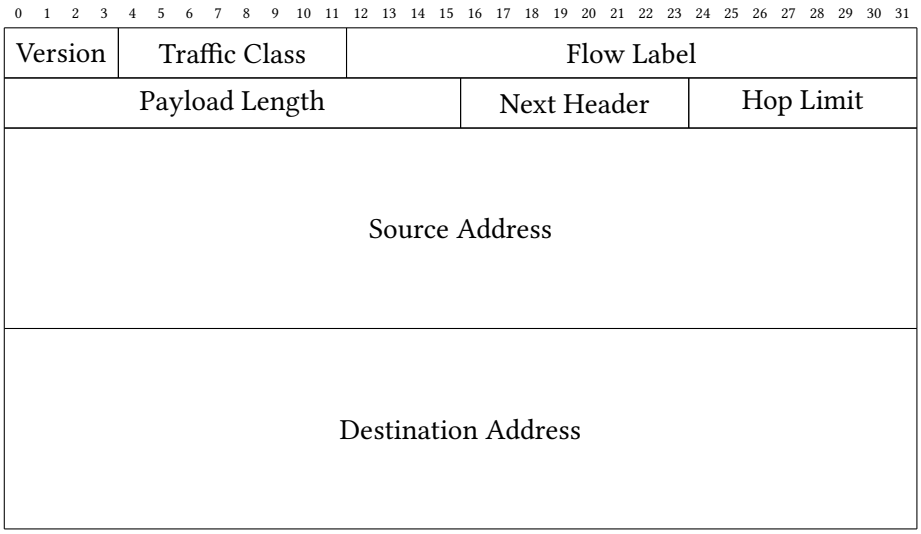

Figure 1.1: The IPv6 header format.

is read by some load balancers or classifiers to balance the traffic or prioritize it. The payload length indicates the length of the data part of the IP packet beyond this header. The hop limit field constraints the maximum number of routers that this packet can cross. At each router, this hop limit is decremented. When it reaches 0, the packet is discarded and an ICMPv6 message is returned to the sender of the packet. The next header field indicates the type of the next header after this IPv6 header. This can be a transport-layer header or an extension header.

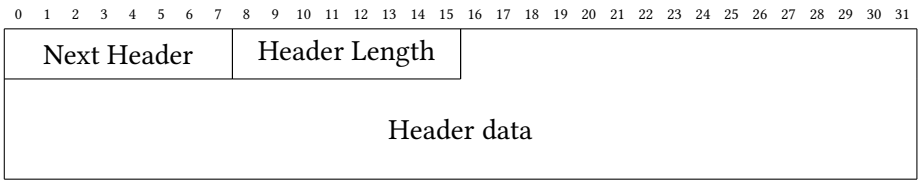

Figure 1.2: The IPv6 extension header format.

Extension headers are optional functionalities that are not present by default in the IPv6 header. If present, they are always placed before the transport-layer protocol header. Figure 1.2 shows the generic IPv6 extension header format. The first field gives the type of the next header in the chain, which can also be another extension header or a transport layer header. The header length helps the parser to identify the position of the next header, even if it cannot parse this extension header. By default, unknown extension headers are ignored. Each extension has its own format for its header data. One of the extension, IPv6 Segment Routing is extensively used in this thesis. It is

detailed in Section 1.8.

1.2 Routers and packet forwarding

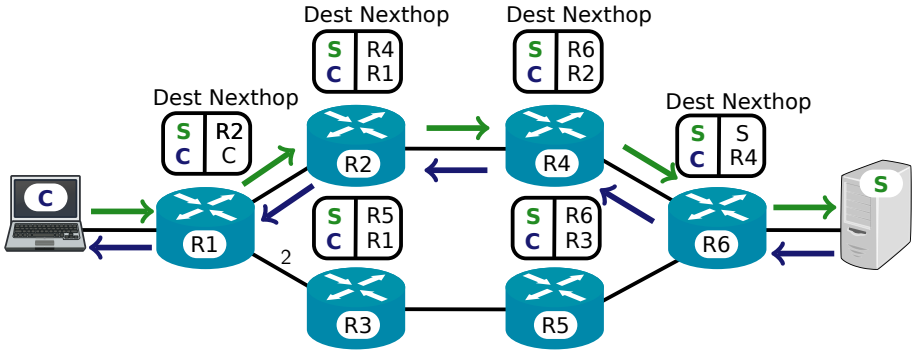

Figure 1.3: Packet forwarding through Link State routing (e.g., OSPF or IS-IS).

Packets are forwarded in the network through devices called routers. These routers have routes to forward them to destination. The process of finding routes is automated and distributed using specialized protocols, such as OSPF [Moy13] or IS-IS [ISO08]. They exchange connectivity information so that every router in the network can compute the shortest path from itself to any destination (using for instance shortest-path algorithms such as Dijkstra [Dij59]). Network operators can set any strictly positive weight on the links, e.g., to avoid using intercontinental links unless there is no other way.

Figure 1.3 shows the packet forwarding process. At each router, green packets, destined to S, are forwarded to the nexthop in the routing table. For R1, it is R2. R2's routing table identifies R4 as the nexthop, and so on until S is reached. Link weights are by default 1, except for the link between R1 and R3. This is why a routing protocol based on shortest path algorithm chooses the upper path to forward the green packets. If R4 fails, the packets will switch to the bottom path when the loss of connectivity is propagated to the other routers.

It is possible for a destination to have two or more paths with the same sum of link weights. For instance, in Figure 1.3, if the link weight of R1-R3 is 1, R1 could use both paths and load balance traffic using Equal-Cost Multiple Paths [Hop00] (or ECMP). Routers compute a hash of the information identifying packets coming from a given flow and they use modulo operator to select the index of the path based on this hash. In IPv6, it uses the source and destination addresses, as well as the flow label. If the underlying protocol is TCP or UDP, it also uses the source and destination port numbers.

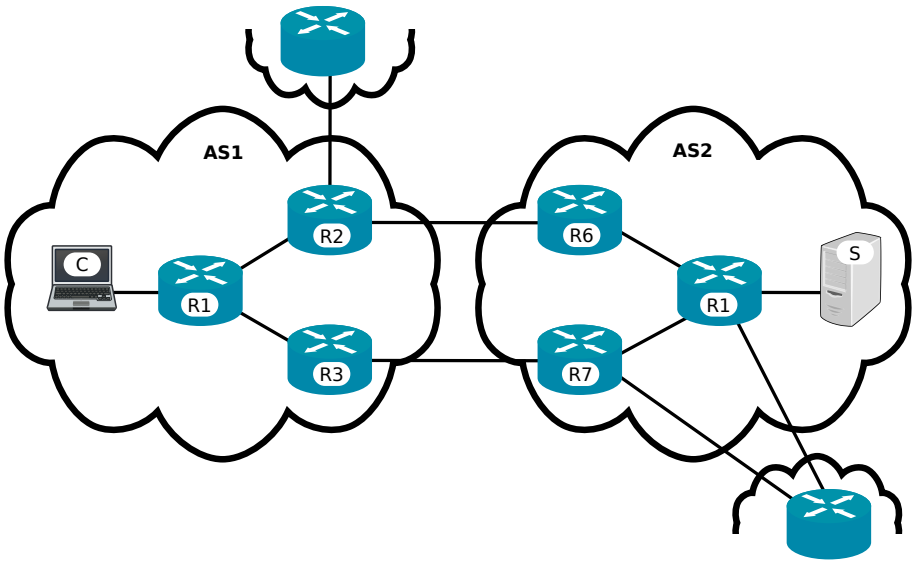

Figure 1.4: Interdomain routing with BGP.

Internet is an inter-connection of networks controlled by different organizations (e.g., Proximus, Google,...) that agree to share connectivity with each other. These networks are called Autonomous Systems (AS). They use a specific routing protocol to express their complex business policies: the Border-Gateway Protocol [RLH06] (BGP). This protocol selects one route for each destination prefix. The routes are distributed to the intradomain routing protocol so that endhosts (Laptops, phones, servers,...) in the network can connect to the Internet.

Figure 1.4 shows a small interdomain network with two ASes that share two peering links through their border routers (i.e., R2, R3, R6 and R7). These border routers run BGP to learn the set of IP prefixes that are reachable via each AS or even the prefixes it learned from other ASes.

1.3 Transmission Control Protocol (TCP)

IPv6 carry data until a destination across the Internet. However, routers can drop packets because of memory pressure, data transmission can be faulty,... Some IPv6 packets can be delayed and transmitted out-of-order. For all these reasons, it is necessary to have a reliable protocol that retransmits lost packets and reorders them at the destination as they were initially sent by the source. The Transmission Control Protocol [Pos81b] (TCP) is such a reliable transport protocol.

1.3.1 TCP Header

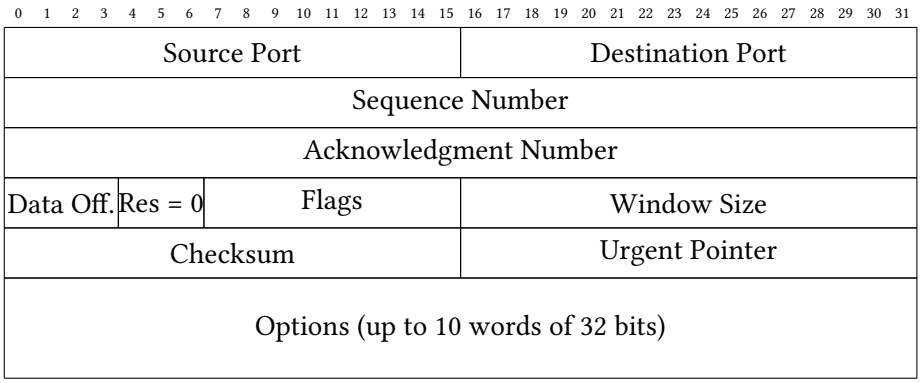

Figure 1.5: The TCP header format.

IPv6 addresses identify machines, but an incoming packet could be for the browser while the other one is for the mailing agent. In order to identify the destination and source applications of a packet, machines use an identifier called port. The source and destination ports are detailed in the TCP header shown in Figure 1.6.

To be able to detect losses or packet reorderings, TCP maps each byte of data with a sequence number. The header carries the sequence number of the first data byte in the packet. The receiver thus always knows which byte comes first. It can also detect missing sequence numbers. It can notify the sending-side of received sequence numbers with the Acknowledgment Number. This number acknowledges the reception of all the bytes prior to this value. The sender thus knows when it needs to retransmit missing data and recover from losses.

The data offset is the number of 32-bit words separating the start of the TCP header from the actual application data. The size of 4 bits restricts the TCP Option Space to 10 words of 32 bits. Indeed, the data offset value cannot exceed 15.

The reserved field means that these bits are reserved for a future use and should be set to 0. As of this writing, there are 9 defined TCP flags. The first three bits are used for Explicit Congestion Notification [RFB01] (ECN). Then the URG flag indicates to the other endpoint whether the urgent pointer's field is significant. The latter field indicates the sequence number of the last urgent byte. The ACK flag has the same purpose for the acknowledgment number's field. The PSH flag asks the TCP implementation to push the buffered data to the receiving application without waiting. The last three flags (RST, SYN and FIN) are used to initiate and tear down TCP connections.

Because the destination does not have an infinite memory, the source cannot send more than what the other end is ready to accept. Otherwise, the destination will discard data. The amount of data that the destination is ready to received is encoded in the Window Size. The checksum [Pos81b] is an error detection code to detect transmission errors affecting the IPv6 addresses, TCP header and application data.

TCP Options were designed to improve performance or add features to TCP. One of them changes the unit of the Window Size: the TCP Window Scale option [Bor+14].

1.3.2 TCP Connection

To explicitly control the existence and the availability of the other endpoint, TCP establishes a connection to the other end. The existence of this connection means that the other end agrees to receive data.

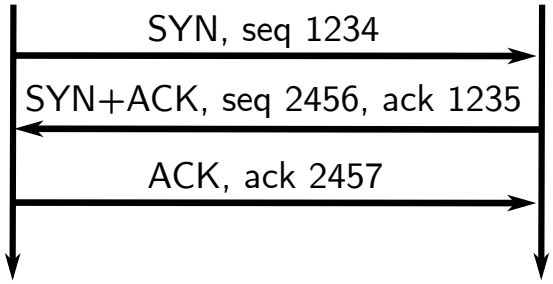

Figure 1.6: TCP Three-way Handshake.

The establishment of a connection forms an exchange of three messages, called the TCP three-way handshake. Figure 1.6 shows the time sequence diagram of the exchanged messages. These are regular packets containing a TCP header, but the header has particular flags set to signal the intent to start a connection. The endhost actively starting the connection sets the SYN bit to 1 and chooses the starting sequence number of its future data stream. The receiving endhost can accept, refuse or ignore the connection. To accept it, it sends a packet with the SYN and the ACK flags set to 1. It also picks its own starting sequence number because the data can be sent from both endpoints. The connection is finally established when the first endpoint acknowledges the SYN+ACK packet.

After transmitting the data, the connection can then be torn down in two ways: a graceful connection release or an abrupt one. Figure 1.7 shows the time diagram of the exchanged messages in the graceful connection release. FIN is another flag of the TCP header. The abrupt connection release is simpler and preferred on overloaded servers. It sends a packet with the RST flag

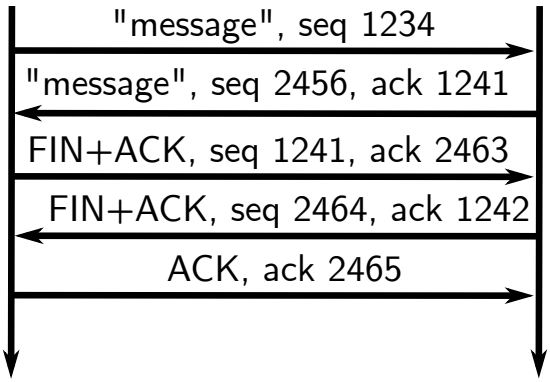

Figure 1.7: TCP Graceful Connection Release.

set to 1. This connection release requires fewer messages and is quicker than the graceful release.

1.4 User Datagram Protocol (UDP)

Some applications do not require a reliable transfer, for instance online games. In this instance, a player will not notice a few missing frames. However, we still need to identify applications with port numbers. For these use cases, we can use the User Datagram Protocol [Pos80] (UDP).

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Source Port	Destination Port
UDP Length	Checksum

Figure 1.8: The UDP Header format.

Figure 1.8 shows the format of its header. The first two fields are the port numbers. Then, the UDP length that gives the length of the UDP header and its data. Thus, they are limited to 65,535 bytes. The last field is for the checksum to detect modifications of the IPv6 addresses, UDP header and application data.

1.5 Domain Name System (DNS)

Hosts are identified by their IP and IPv6 addresses, but this is impracticable for humans. Moreover, machines change their IP addresses if servers are moved to a datacenter in another country and users should still be able to access the service. For that, they can use the Domain Name System [Moc87] (DNS).

1.5.1 DNS hierarchy

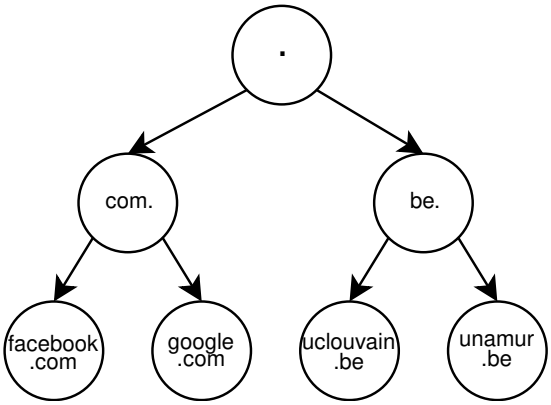

Figure 1.9: The DNS hierarchy example.

As for the IP addressing, DNS names are hierarchical. Figure 1.9 shows a small subset of the real DNS hierarchy. Each domain is divided in subdomains that are more specific than their parents. For instance, the domain name "`uclouvain.be.`" is controlled by the university but is registered to the owners of the domain name "`be.`". This domain is registered on the servers of the root domain.

Each domain is responsible for storing the mapping between its domain names and their IPv4 and IPv6 addresses. These pieces of information are called records.

To learn the mapping between a domain name and its IP address, machines use the DNS protocol. For instance, for the "`google.com.`" domain, it queries the root domain DNS servers for the DNS servers of the domain "`com.`". Then, it queries these servers for "`google.com.`" domain's DNS servers. These servers know the mapping between the name and the IP addresses, so its last query is for them. DNS records are cached by DNS resolvers so that they don't need to perform this whole search every time a user loads a web page.

1.5.2 DNS Packet

The DNS queries and replies were originally designed to be carried in UDP packets. However, it can also be carried in other protocols, such as TCP [Dic+16] or HTTPS [HM18]. The DNS header format, shown in Figure 1.10 is the same no matter the underlying protocol. Each request contains an ID to link the reply with the query. The R bit indicates whether the header is a request (0) or a reply (1). The opcode identifies the type of query. A standard

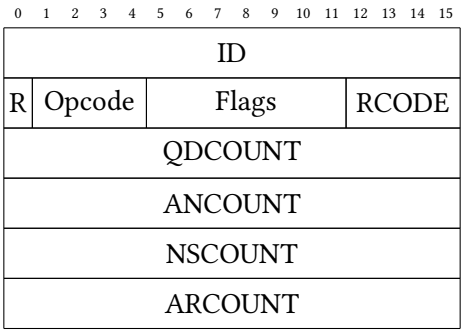

Figure 1.10: The DNS Header format.

query tries to get records attached to domain names but it is also possible to do a reverse query to get the domain name of an IP address.

Because of DNS caches, it is possible for records to be found in DNS servers that are not the DNS servers of the domain. When a reply is coming back from the DNS servers responsible for a domain, they set one of the DNS flags, the AA bit to 1 to express that they are the authoritative servers for the domain. There are other flags such as the TC bit marks the truncation of the reply. This can happen when DNS runs over UDP but not over bytestream protocols such as TCP.

The RCODE is the response code that indicates whether the query triggered an error or not. The error can be from the server's fault or due to the query of a nonexistent domain.

The next fields are the number of records, for each record section, inside the DNS packet. QDCOUNT counts the number of records in the question section, ANCOUNT for the answer question, NSCOUNT for the authority section (including authoritative DNS name server for the domain), ARCOUNT for the additional records section.

The payload (i.e., data) of a DNS packet is the list of records for each of the section. These sections are listed in the same order as their counter. If a counter is set to 0, then the section is absent from the DNS payload.

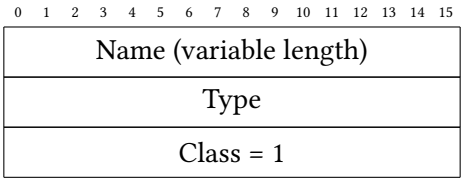

Figure 1.11: The DNS Question format.

DNS questions have a different format from other records. This format is shown in Figure 1.11. The name is the domain name of the query. The type indicates the desired type of records. DNS servers hold the mapping towards

IP addresses but also the domain names of mail servers and the DNS servers of the subdomains. Class was designed to allow the existence of multiple DNS hierarchies but nowadays, we only use one DNS hierarchy.

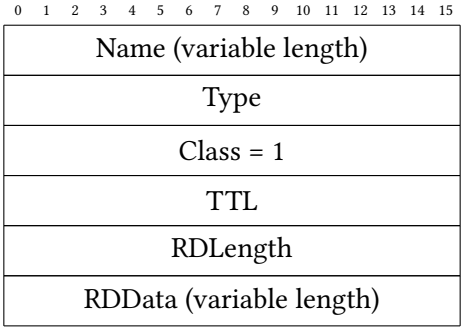

Figure 1.12: The DNS Resource Record format.

The format of the other records is presented in Figure 1.12. The first three fields have the same purpose as in a DNS question. In addition to that, DNS servers indicate in the TTL, the number of seconds for which this record can be cached. After this delay, a new DNS query should be made to download the updated record. RDData content depends on the type of records. For instance, it can be an IPv6 address or the domain name of the subdomain DNS server. RDLength indicates the length of this field. Thus, unknown record types can be skipped.

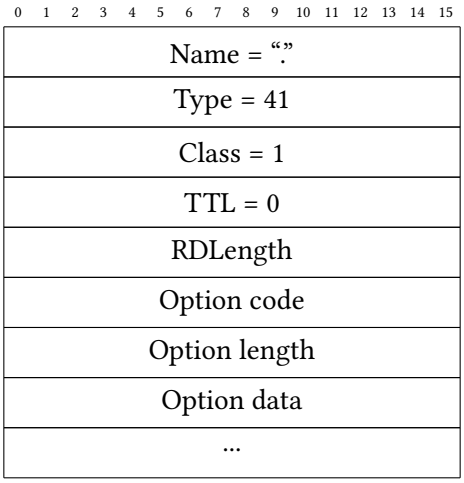

Figure 1.13: The DNS EXT0 Pseudo-Resource Record format.

[DGV15] specifies a special type of record shown in Figure 1.13. This record extends a DNS request to pass additional information to the DNS server. Indeed, a DNS reply can be extended by inventing new DNS resource records

but the DNS question format does not have fields for arbitrary data. This record is placed in the additional record section of a DNS request. Its data includes a list of options. Each option is specified with three fields: the option code indicating the type, the actual data and its length.

1.6 Quality of Service

In Section 1.2, each flow gets the same treatment along the path. This is not necessarily the best approach. Some applications are more delay-sensitive than others, e.g., Voice over IP [Ros+02]. To give some guarantees of service, Integrated Services [BCS13] and Differentiated Services [Wan+13] were designed.

1.6.1 Integrated Services

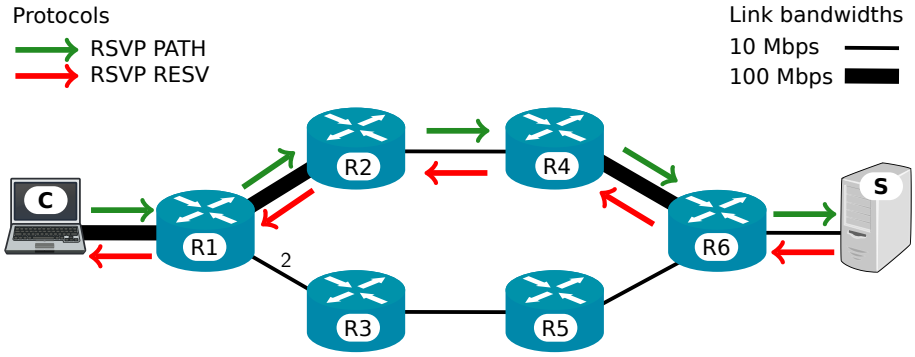

Figure 1.14: Resource Reservation for Integrated Services.

In Integrated Services, each flow can be allocated a given share of network resources, for instance a guaranteed share of the bandwidth. The advantage is that the flow will not have to compete with the rest of the traffic to get that bandwidth.

Figure 1.14 shows the resource reservation process in a small network. Each time that a client starts a new flow, it reserves resources on the network that matches the applications' requirements.

Let's assume that the flow can use up to 50 Mbps, but the flow can work with less bandwidth. The client, C tries to reserve this bandwidth by using the Resource Reservation Protocol [Bra+13]. The client does not know the current bandwidth available on the routers. C sends a PATH message to its access router, R1, with a bandwidth request of 50 Mbps. R1 reserves 50 Mbps for this flow and forwards the message to the next hop that it will use to reach the server: R2. R2 cannot guarantee 50 Mbps because the link R2-R4 only has

10 Mbps available. Since the 50 Mbps are not required, instead of returning an error to R1, it forwards the PATH message to R4 after updating the requested bandwidth to 10 Mbps. R4 reserves 10 Mbps and forwards the message to R6. Finally, R6 the final router, forwards the PATH message to the server, S. It sends a RESV request to confirm the final bandwidth reserved: 10 Mbps. The message is forwarded backwards to R6, R4, R2 and R1 which update their reservations to 10 Mbps instead of 50 Mbps. When the RESV request reaches C, the endhost knows that the flow's reserved bandwidth. Reservations time out if they are not renewed regularly by the client (C in Figure 1.14). Each router needs to remember the state that was reserved to free its resources at the end of the reservation.

Integrated Services are not deployable in a large network or over the Internet because all the routers need to maintain the state of each flow going through them. This number is significantly large for routers at the center of big networks: they would need to hold several million of reservations at the same time, in addition to their routing tables.

1.6.2 Differentiated Services

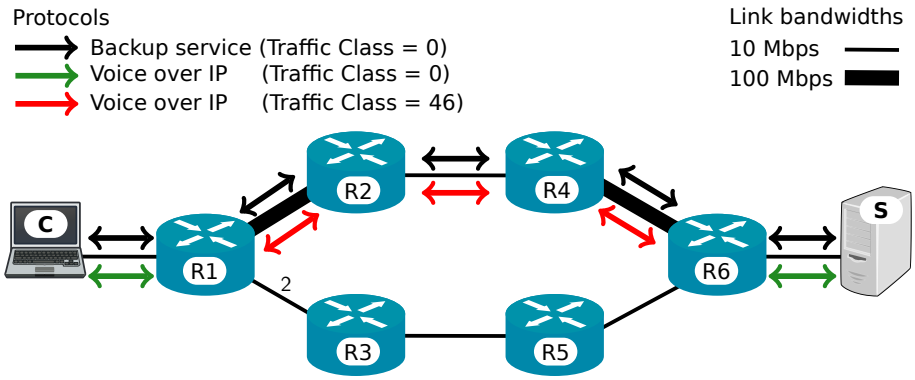

Figure 1.15: Traffic is classified by the access routers for Differentiated Services.

Differentiated Services [Wan+13] take a more scalable approach than Integrated Services (Section 1.6.1). Instead of reserving resources for each flow. It applies a different service per class of flows. For instance, Expedited Forwarding provides low-loss and low-latency transfers for Voice over IP. Differentiated Services' architecture does not change the routing. Both regular traffic, called Best-Effort traffic, and Expedited Forwarding take the same path if they have the same destination.

Figure 1.15 shows a network using differentiated services to provide fewer losses and a low latency to Voice over IP. The access router, R1, is responsible

for identifying this type of traffic and changing the traffic class field in the IPv6 header (see Figure 1.1) and set it to 46, the Expedited Forwarding class value. The core routers, R2 and R4, only read the traffic class value and prioritize traffic class set to Expedited Forwarding. They have to apply at most 64 different behaviors, at worst, which is much smaller than the number of flows in large networks. Therefore, Differentiated Services' architecture is actually deployed in production networks while Integrated Services' architecture is not. The exit router can reset the traffic class field before it reaches the destination to hide these details to the endhosts.

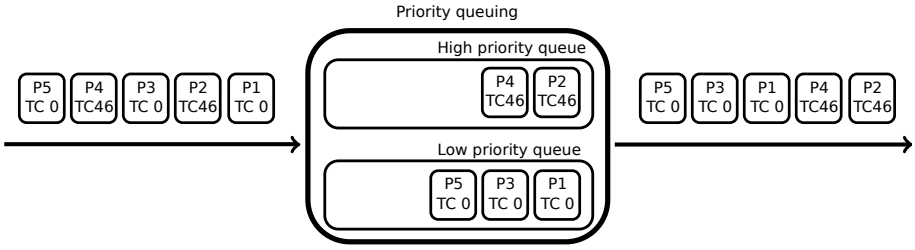

Figure 1.16: The priority queuing favors packets tagged with the Expedited Forwarding class (46) to provide a low latency and few losses.

To offer few losses and low latency for Expedited Forwarding, routers can use a priority queuing strategy: they place the packet of a queue depending on its traffic class. Figure 1.16 shows an example of priority queuing. In our example with only Best Effort and Expedited Forwarding classes, only two queues are required. The numbers on the packets indicate their order of arrival. Only P2 and P4 are placed in the queue with the highest priority. A queue can only be emptied if the queue of higher priority is empty. Otherwise, the non-empty queue with the highest priority is processed. Therefore, the order of the packets forwarding on the wire will be P2, P4, P1, P3, P5.

This behavior provides a lower latency because it reduces the sequential delay and reduces the chance of losses because a queue emptied quickly is less likely to grow beyond its limits and cause packet drop. However, Differentiated Services assume that the support for a class exists on all the routers on the path. Partial support is possible but less effective. The Differentiated Services architecture does not try to find a better path for prioritized traffic. It tunes the experience of the traffic on the default path.

More details about the existing classes and their implementation on intermediate routers are found in [FDC02; Wei+99].

1.7 Software Defined Networks (SDN)

The routing decision in Section 1.2 is distributed. This is great for scalability but is hard to manage because every device has to be correctly configured to support more complex policies than shortest path routing. From this need, Software Defined Networks [McK+08; Cas+07; Kre+15] (SDN) were born. In this section, we present SDN networks and some selected architectures that are related to this thesis.

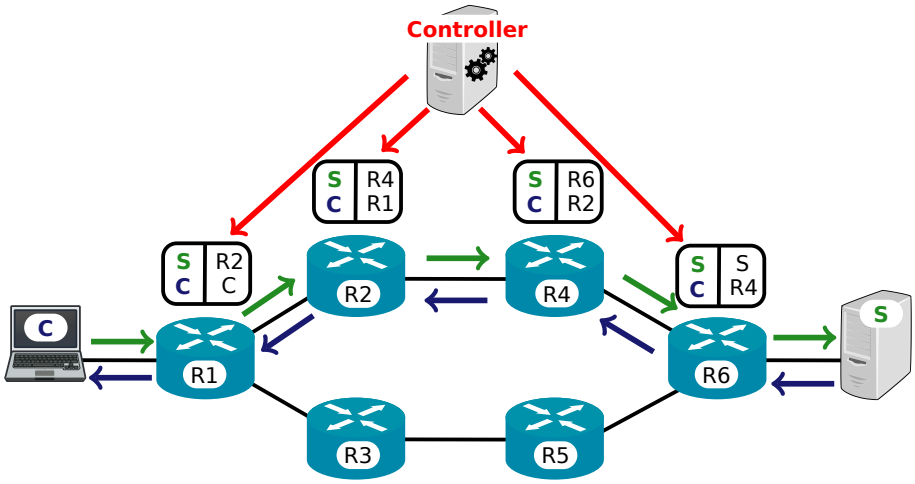

Figure 1.17: Intradomain routing with SDN.

Several types of SDN networks have been proposed (see [Kre+15] for a detailed survey). They typically rely on a logically centralized controller as shown in Figure 1.17. The controller interacts with the network devices (routers, switches and sometimes middleboxes) to support the network policies defined by the operator. The network devices are kept simple and the controller pushes their packet forwarding rules. OpenFlow is a popular protocol to push those routes [McK+08]. The controller can implement complex policies, for instance, not using the shortest path.

One of the first SDN architectures was Ethane [Cas+07; McK+08] to dynamically tune the routing of each incoming flow. A flow can be a new TCP connection or traffic going to the same destination.

Figure 1.18 shows the creation of a new route in an example network. When a packet from a new flow reaches an OpenFlow switch, there is no specific forwarding route programmed by the controller. However, a default behavior, programmed by the controller, sends all the new packets to the controller. The controller reads the incoming packet and computes a forwarding route through the network. Access control policies or optimization objectives can be set by the network operators. In our example, the controller forces this

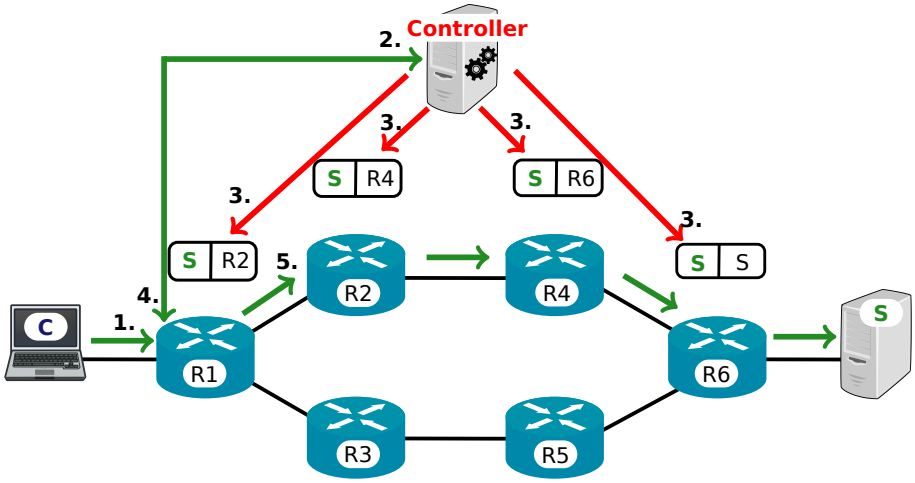

Figure 1.18: Dynamic routing of new flows with Ethane.

connection through the upper path. The controller first inserts the forwarding rules through each of the OpenFlow switches on the network. Finally, it forwards the buffered packets back to the first OpenFlow switch which forwards it to the next hop of the upper path. Eventually, the packets reach their destination.

PANE [Fer+13] introduces an API for the endhosts' applications to interact with an OpenFlow controller. This API is implemented with a new protocol. Applications can reserve a given bandwidth or transit through a particular middlebox. They can also give a hint to the controller of the duration of their traffic flow, which helps with the bandwidth reservation. Figure 1.19 shows the creation of a new route in an example network. Instead of waiting for the first packet of the connection to enter the network, the endhost applications first uses PANE's API to reserve a path through the network. After computation, the traffic starts.

1.8 IPv6 Segment Routing (SRv6)

Shortest Path Routing is simple to compute for routers, even on large topologies. However, network administrators might want to route traffic along non-shortest paths. For instance, they could route traffic away from congested paths or use specific paths for business critical traffic without installing OpenFlow switches. Segment Routing [Fil+18b] (SR) encodes segments that are instructions to be interpreted in SR-capable routers. SR can be applied to MPLS or IPv6. This thesis focuses on IPv6 Segment Routing [Fil+20] (SRv6).

In SRv6, the segment list is encoded as a type of Routing Header, an IPv6 Extension Header, called the Segment Routing Header (SRH). Figure 1.20

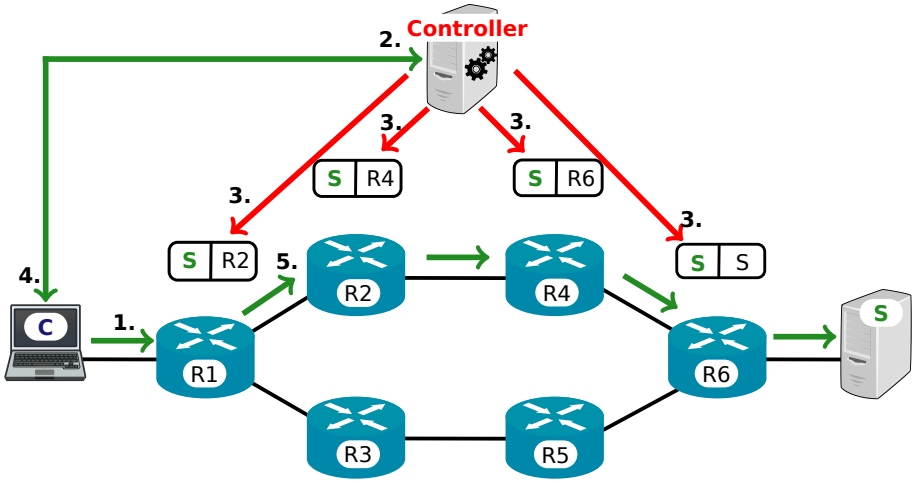

Figure 1.19: Endhost interactions with PANE.

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Next Header	Header Length	Routing Type = 4	Segment Left
Last Entry = N	Flags	Tag	
Segment List[0]			
...			
Segment List[N]			
Optional Type Length Value objects (variable)			

Figure 1.20: The IPv6 Segment Routing Header format.

shows the format of this extension header. Next Header and Header Length are the necessary fields for any extension header, as explained in Section 1.1. The routing type, set to 4, indicates that the routing extension header contains a Segment Routing Header (SRH). The Segment List is encoded in reverse order, i.e., Segment List[N] is the first segment to apply and Segment List[0] the last one. Segment Left gives the number of segments that are left to apply, excluding the current one. The Last Entry field gives the index of the last segment which is N since we have N+1 segments. This value is not supposed to change while Segment Left is decremented each time a segment is applied. In SRv6, the current segment to apply replaces the IPv6 header destination address. The Segment List[0] is therefore used to remember the destination and apply it after all the other segments.

In addition to the segment list, SRH can include additional information in Type-Length-Value (TLV) format. For instance, the SRH can be authenticated using an HMAC [KBC97].

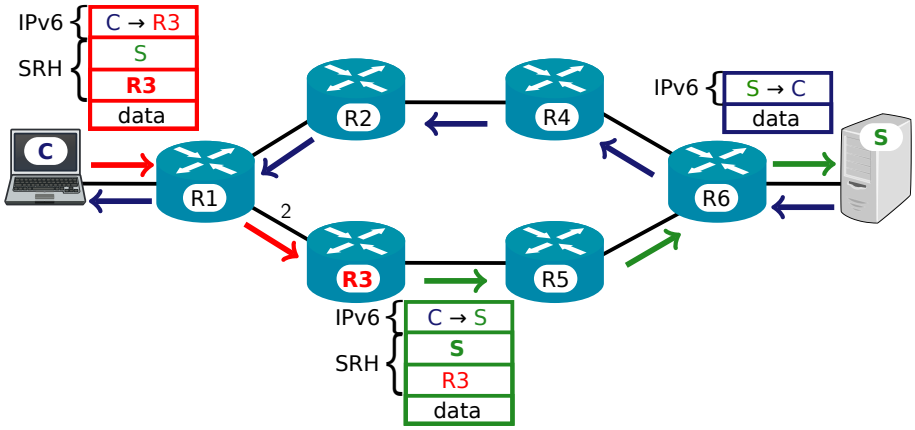

Figure 1.21: Non shortest-path routing with IPv6 Segment Routing.

The most basic type of segment is called a node segment. This segment is an IPv6 address of a node in the network. Only this intermediate destination needs to be able to parse an SRH. The rest of the routers only read the destination in the IPv6 header which was replaced by the current segment and forward it along the shortest path. Figure 1.21 shows an example of a segment list containing one node segment. The first segment forces the packet to first go through R3 before reaching the destination. Only router R3 needs to parse the SRH to replace R3 by the initial destination of the packet: the server S.

Figure 1.22 showcases another type of segment: the adjacency segment. This segment identifies a link to use in particular. Even if the link has a weight of 100, an adjacency segment can force the traffic through Link R1-R3. A node segment would not be able to use this link since the shortest path towards

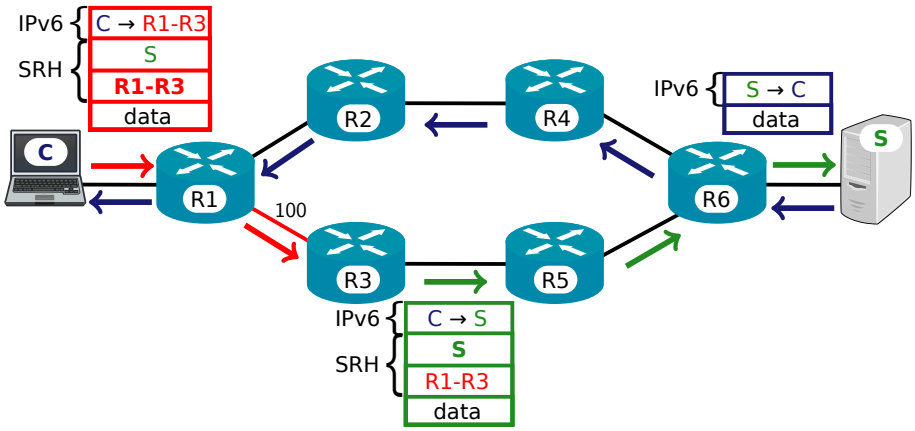

Figure 1.22: An adjacency segment identifies a link in particular.

R3 is not through Link R1-R3 in this topology. Other types of segments are defined [Fil+21]. But ultimately, we can imagine any type of segment that triggers any type of behavior.

1.9 Conclusion

In this chapter, we present the networking concepts required to understand the following chapters. Section 1.1 presents the IPv6 protocol [DH17] that adds a source and a destination address to packets of data. Routers forward these packets to their destination by following routes. These routes can be derived through protocols such as OSPF [Moy13], IS-IS [ISO08] for intra-AS routing. BGP [RLH06] is used for inter-AS routing. Section 1.3 and Section 1.4 introduce two transport protocols: TCP and UDP. Transport protocols use port numbers to identify applications on a particular host, which is identified by one or more IP addresses. Section 1.5 present DNS [Moc87] that was designed to assign human-readable names to hosts and map them to their IP addresses for actual packet forwarding. In Section 1.7 and Section 1.8, we see two alternative processes of intra-AS routing, namely, Software Defined Networks [McK+08; Kre+15] and IPv6 Segment Routing [Fil+20].

Software Resolved Networks with IPv6 Segment Routing

2

IPv6 adoption in the global Internet has grown in a spectacular fashion. Pushed by the increasing pressure of the IPv4 addressing space exhaustion, Content Delivery Networks (CDN) and Internet Service Providers (ISP) have deployed IPv6 at a large scale [NG16]. Today, a growing fraction of mobile and home users rely on IPv6 to access web-based services [Wu+13; Col+10; Czy+15] and some mobile providers have deployed IPv6-only networks. This IPv6 wave has not yet reached the majority of enterprise networks but the most advanced ones have already pledged to move to an IPv6-only architecture [BNC11; Hol16; Mar16; Kea17]. With significantly fewer users than large providers, small and middle-sized enterprises do not feel the same pressure to move to IPv6 as ISPs. Many consider IPv6 as simply a variant of IPv4 with more addresses and have difficulties in justifying the cost of an IPv6 deployment. This incorrect assumption plays a key role in the current *status quo* of IPv6 deployment in enterprise networks.

In parallel, many enterprises are seduced by Software Defined Networks (SDN) [McK+08; Cas+07; Kre+15] that promise to simplify the management of their networks. These are often more complex than ISP networks, given the need to support a variety of business policies [Mal+04; Sun+11]. These policies correspond to various business objectives that need to be met by the network. A first example is Quality of Service. Many enterprise networks have deployed Voice over IP and video services that require special QoS in particular on low bandwidth wide area links. A second example is the need for fine-grained access control. It is frequent in enterprises to restrict access to parts of the network for some classes of users. These restrictions can be implemented by using firewalls, routing policies and other techniques. A third example is the need to support specific paths for specific applications, e.g., to use dedicated links or support extranet services. A fourth example is the large number of middleboxes that are deployed in many enterprise networks [She+12]. Several types of SDN networks have been proposed (see [Kre+15] for a detailed survey). They typically rely on a logically central-

ized controller that interacts with the network devices (routers, switches and sometimes middleboxes) to support the network policies defined by the operator. OpenFlow is a popular protocol that enables SDN controllers to interact with network devices [McK+08].

In this chapter, we propose an architecture to instantiate the SDN vision in IPv6 enterprise networks. To achieve this, we leverage IPv6 Segment Routing (SRv6) [Fil+18b; Fil+20] to steer packets through an ordered list of instructions. We call our architecture **Software Resolved Networks** (SRN). An SRN is a network that is managed by a logically centralized controller. The name SRN originates from the fact that our architecture co-locates its controller with a DNS resolver and uses extensions of the DNS protocol to interact with trusted endhosts. In developing Software Resolved Networks, we make the following contributions.

An architecture implementing an application-centric SDN paradigm suitable for enterprise networks (Section 2.1, Section 2.2 and Section 2.3). This architecture supports *conversations* between applications, regulated by their interactions with the controller through the DNS protocol.

DNS extensions enabling (*i*) applications to embed traffic and/or path requirements in their DNS requests and (*ii*) the controller to return the appropriate path to the applications (Section 2.2).

A prototype implementation running on Linux that demonstrates the feasibility of our approach and enables other researchers to replicate and expand our results. We implement a modular controller, and extend DNS libraries (Section 2.4).

Measurements demonstrating the flexibility of our prototype and its performance through various microbenchmarks and experiments (Section 2.5).

2.1 Enterprise Networks

Most enterprises, notably within a campus, have a dense network, *i.e.*, there are many possible paths between a given pair of hosts through the network. From a high level viewpoint, a mathematician could represent the network engineer's job as a function $\mathcal{M}$ that maps the end-to-end communication flows on specific network paths, or the empty path when a flow is prohibited by the network policies. The output of this mapping function typically depends on a mix of configuration parameters (link weights, IP addresses, VLANs, access lists, etc.), the state of the network links and nodes, and the characteristics (*e.g.*, source and destination addresses and ports) of the packets exchanged for each flow.

In practice, function $\mathcal{M}$ can be realized by using a variety of technologies. A very simple approach in a small network is to use the Spanning Tree protocol. In this case, the active network topology is restricted to a tree and there is only one path for each end-to-end flow. Several variants of this approach have been deployed. IP routing protocols such as OSPF are other popular examples. In this case, function $\mathcal{M}$ becomes a shortest-path algorithm whose main parameters are the link weights and the status of the links and nodes. Packets follow the shortest path, possibly with per-flow load balancing when there are equal cost paths. Enterprise network operators have deployed a variety of solutions to leverage other paths than the shortest ones. Some have used several routing protocols [Mal+04], configured VLANs or access lists [Sun+11] or even used BGP inside the enterprise [LP16]. Others have relied on policy routing to provide finer control on the selection of the end-to-end paths. Integrated Services [BCS13] was introduced as an architecture to add resource reservations for end-to-end flows. With this architecture, packets are still forwarded along the shortest path and the RSVP protocol [Bra+13] is used to maintain per-flow state on the intermediate nodes.

Various realizations of the Software Defined Networks (SDN) paradigm have revisited this problem. Instead of using a distributed protocol with configuration parameters that sometimes indirectly influence the end-to-end paths, SDN relies on a logically centralized controller that implements function $\mathcal{M}$ and programs the intermediate nodes to realize the chosen network path for a given end-to-end flow. A simple realization is to intercept the first packet of each flow on the edge node, forward it to the controller that selects the path and programs the intermediate nodes by using the OpenFlow protocol [McK+08]. A wide range of solutions have been proposed under the generic umbrella of a logically centralized controller [Kre+15].

Any realization of function $\mathcal{M}$ is always a tradeoff between the number of parameters that the network operators can tune and the amount of state that needs to be maintained on the network nodes. Some enterprises require solutions that provide a very fine-grained control on the mapping of traffic flows onto network paths, possibly on a per connection or a per source/destination pair basis. On the other hand, network operators need to minimize the amount of state that must be installed and maintained on each node for obvious scalability reasons.

Enterprise networks are composed of three main types of network devices: layer-2 switches, layer-3 routers and middleboxes [She+12]. We distinguish three types of layer-3 routers. At the edge are *access* and *border* routers. Hosts are connected to access routers. Border routers are connected to upstream providers. *Core* routers are only connected to other enterprise routers. We use Router Advertisements [Sim+07] (RAs) and StateLess Address AutoConfiguration (SLAAC) to assign one or more IPv6 addresses to

the hosts. The DNS configuration of the hosts can be distributed through RAs or DHCPv6 [Mad+10].

Routers exchange routing information by using a link state routing protocol such as OSPFv3 [Col+15]. In particular, we assume that the link state routing protocol used in the enterprise supports traffic engineering extensions that distribute unidirectional link metrics such as bandwidth utilization and link delay [Gia+15].

2.2 Software Resolved Networks

A key principle of Software Resolved Networks (SRN) is to let applications have an active role in the management of their flows.

When an application running on a client host communicates with a server, the packets carried from one endpoint to the other are usually called a unidirectional flow. In this chapter, we always associate this flow with a return flow. In other terms, we consider that two applications always communicate in a bidirectional fashion. We call this pair of flows a *conversation*. We distinguish the two endpoints of a conversation. The application that initiates a conversation is called a *client application*. The process of initiating a conversation is referred to as *establishing a conversation*. Conversely, an application accepting conversations is called a *server application*. We also distinguish applications with respect to their location. An application whose endpoints are located inside the enterprise is called an *internal application*. Likewise, an application whose endpoints are located outside the enterprise is called an *external application*.

We make two reasonable assumptions about the enterprise network: (*i*) all endpoints are reachable over IPv6 and (*ii*) all endpoints are identified by DNS names. Each device in the network is properly named according to a DNS naming plan. Furthermore, we leverage the large number of IPv6 addresses [Col+16]. For example, each server may receive several IPv6 addresses, and each address is only used by a particular application. Furthermore, we assume that server applications are never referred to by their IPv6 address at the application layer, but rather by their DNS name[1]. The applications use the DNS protocol to interact with the controller. We prefer to use DNS instead of a signalling protocol such as RSVP or NSIS for two main reasons. First, our architecture does not create state on core routers. Second, DNS is easy to extend, widely implemented and well-known by network operators. Furthermore, using the DNS minimizes the additional delay caused

[1]The only exception is the DNS resolver whose IP address is distributed by DHCPv6 or Router Advertisements. We also assume that enterprise applications will be written in a high-level programming language that provides a `connect-by name` API instead of the `connect-by address` of the C socket API.

by the creation of the network path. To facilitate this interaction, the default resolver configured for these applications is the controller itself (the SRN), acting transparently as a regular DNS resolver.

When a client application initiates a conversation to a server application in a Software Resolved Network, it performs the following operations. First, it issues a DNS request to resolve the name of the server application into an IPv6 address. Then, it sends data to the resolved address. As the DNS resolver is actually the controller, it may automatically perform appropriate actions, such as allocating a network path. Furthermore, the client can embed in the DNS request requirements about the conversation, using existing DNS extensions [DGV15]. For example, those requirements can list the expected bandwidth and latency. We name such a DNS request a *conversation request.* Along with the resolved IPv6 address, the controller returns a *Path ID* in the DNS reply. This *Path ID* is an opaque string that maps to the allocated network path and is the key to enabling the application to use this selected path. It is important to note that a *Path ID* uniquely identifies *one half* of a conversation, *i.e.*, the packet flow starting from the client application that issued the conversation request and going to the other endpoint. An application may reissue a conversation request at any time during the lifetime of a conversation. This enables the application to request a new network path corresponding to updated requirements.

As such, the SRN provides a mechanism for the dynamic registration of server applications, namely *server registration.* A server registration is very similar to a conversation request. Instead of resolving the name of an application, the server attempts to resolve a name that is pre-configured by the operator. This name is not attached to any particular application. Rather, its resolution signals a registration request to the controller. If the server has the credentials to register this name, the request is translated into a DNS update message [RB97; Wel00] that updates the corresponding entry.

When the server receives a connection from a client, half of the conversation is established. To establish the other half, the server issues a conversation request to the controller in order to fetch a Path ID. This is realized upon reception of the first packet, and before returning any packet to the client. The server must have some way to identify the other end of the conversation. Using the classical IP 5-tuple is not sufficient as the controller does not have protocol-level information such as the source and destination ports. The only simple, unique identifier of that particular conversation is the Path ID used by the client. To enable the server to use it as reference, the client's Path ID is embedded in the connection request. This is realized using the Segment Routing Header (SRH), further discussed in Section 2.2. Instead of resolving an application name, the server issues a conversation request for the Path ID. This request may also include traffic requirements. The controller allocates

a network path for the other direction of the conversation. A new Path ID is mapped to this network path and returned to the server.

Figure 2.1 shows an illustration of the conversation request and server registration workflows. In exchange (A), the server issues a server registration request to the controller. In exchange (B), the client issues a conversation request towards that same server, with a requested bandwidth of 2 Mbps. The controller replies with the IPv6 address of the server, and with a Path ID. In exchange (C), the server has received a connection request from the client. The client's Path ID is embedded in the connection request. The server then issues a conversation request to the controller, stating the original Path ID and requesting the corresponding reverse Path ID, with a bandwidth requirement of 6 Mbps. The controller replies with the relevant Path ID and the exchange of packets continues.

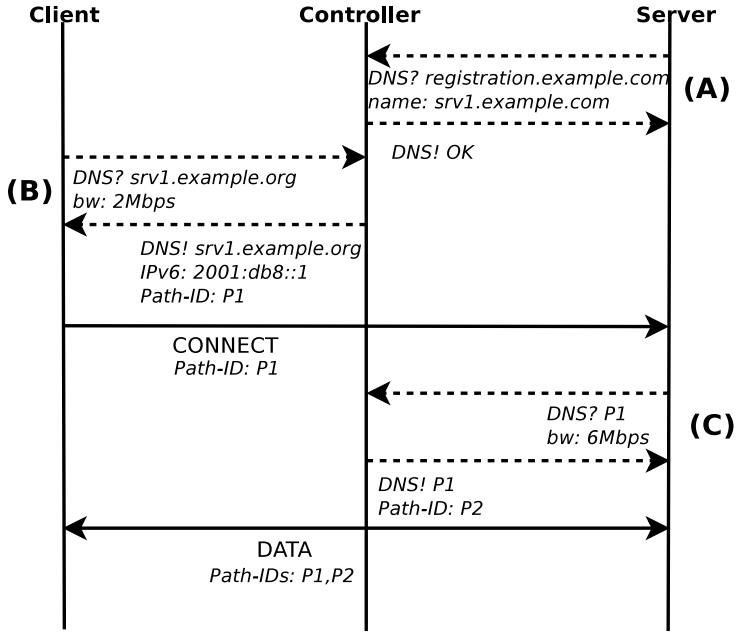

Figure 2.1: Workflow for server registration and connection establishment.

The enterprise network can be connected to one or more upstream providers. As such, client applications may initiate connections towards external servers. Conversely, external applications may initiate connections towards internal servers. We consider three types of conversations with respect to the application locations.

Internal client communicating with internal server. This is the main kind of conversation we focus on. When a client initiates the connection, it sends a conversation request to the controller. The request states

the server application and optional traffic requirements. The controller allocates a path in the network for the client-server direction of the conversation and returns a corresponding Path ID `P1` to the client. This Path ID is embedded in the subsequent connection request packet sent to the server using Segment Routing features that are independent of the transport protocol. Upon reception of the connection request packet, the server issues a conversation request. This request includes the received Path ID, `P1`, and asks for the reverse one. The controller replies with a Path ID `P2`, corresponding to the server-client direction of the conversation. Packets can now be exchanged and each application can re-issue a conversation request at any time to update the requirements of their direction of the conversation.

Internal client communicating with external server. The controller can only control one direction of the conversation because the server resides outside the enterprise network. The connection establishment procedure does not change. However, the other direction of the conversation, *i.e.*, the return traffic, will have to follow some default policies set up at the edge of the network. Such policies might include a detour via a firewall to ensure the traffic is legit. The operator defines those default policies in the controller, which configures the border routers to implement them.

External client communicating with internal server. In this case, the connection establishment originates from outside the enterprise network. As the external client uses its own DNS resolver rather than the enterprise resolver, the controller cannot handle this part of the conversation. The ingress traffic follows default network policies configured at the border routers (*e.g.*, firewall traversal). The server has the opportunity to issue a conversation request, retrieving a Path ID for the server-client direction of the conversation.

2.3 The SDN Resolver

The SRN is the logical controller that manages an SRN. It must accept, process and maintain conversation requests issued by applications. To realize this, it exposes the network state and the conversation requests to externally pluggable path selection algorithms. Those algorithms then select a path that matches the requested conversation properties. This path is transformed into Segment Routing (SR) instructions by the SRN and enforced in the network.

We leverage the *binding segments* [Fil+18b] to implement the Path IDs presented in Section 2.1. Indeed, a binding segment is the unique identifier of an SR policy, which encodes a path in the network. Each host is configured

to send its packets with an SRH containing two segments: the first one is a binding segment (*i.e.*, its Path ID) mapped to an SR policy on the access router and the second one is the final destination of the packet (*e.g.*, a server or client application). The packets sent by the host thus follow the shortest path up to the access router, using regular IPv6 forwarding. Then, they are encapsulated within an outer IPv6 header and the SRH computed by the controller. Hence, the conversation state, implemented with SR policies, is only maintained by edge routers. The core routers only need to be configured with stateless segments that can then be used by the controller to enforce specific paths in the network. These segments can be shared by any number of SR policies. Such a stateless core has the advantage of reducing memory requirements and avoiding conversation state synchronization between routers.

2.3.1 Path segmentation

Network paths implementing application policies are enforced using Segment Routing. Once an appropriate path is selected, it must be transformed into a list of segments. To realize this, we implement a controller that leverages the MinSegECMP path segmentation algorithm proposed in [Aub+16]. It implements the segmented path construction function buildSegpath. From a graph G, a source node s, a destination node t, and a set of policies $\mathcal{P}$, this algorithm computes (i) the full path from the source to the destination that matches the policies (*path computation*), and (ii) the minimal list of segments that implements the previously computed path (*path segmentation*). The path computation is realized through a generic selectPath function. This function must be extended by any external algorithm that provides path selection features. Then, path segmentation is achieved using MinSegECMP. This algorithm computes the minimal segmentation of a path without ECMP.

2.3.2 SRN Control plane

The control plane of an SRN consists of several components. At the core is the logically centralized controller. It does not communicate directly with applications. Instead, the exchanges between the applications and the controller are mediated by a DNS proxy. The proxy receives DNS requests from applications. It performs the actual DNS resolution, using the enterprise's resolver, and forwards the conversation requests to the controller. The controller processes the request, then instructs the access router to insert the resulting SR policy into its Forwarding Information Base (FIB). The controller explicitly waits for the SR policy to be inserted. This synchronization is necessary to ensure that the router does not receive legitimate packets with a binding segment that is not yet recognized in the FIB. Afterwards, the controller returns the generated binding segment (*i.e.*, Path ID) to the DNS proxy. Ultimately,

the proxy crafts the corresponding DNS reply, using the resolved address and the binding segment. This reply is transferred to the application. Assuming that the controller, proxy, and resolver are in the same network vicinity, this setup allows grouping most of the transactions within a low-delay network radius.

The last component of an SRN is the Network State Daemon (NSD) which gathers the network state as exposed by OSPF-TE and forwards it to the controller. The rate of network state updates depends on the OSPF refresh timer defined by the operator. The value should be a trade-off between control traffic overhead and up-to-date metrics. A high-level illustration of an SRN architecture is shown in Figure 2.2.

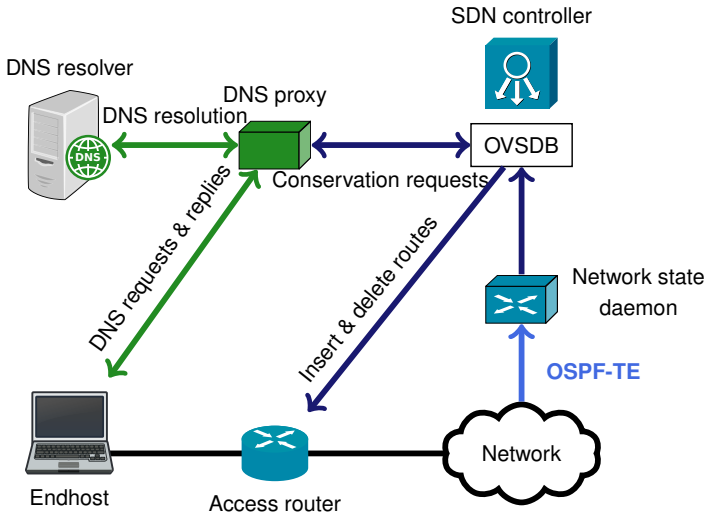

Figure 2.2: Illustration of the components of a SRN. The figure shows the exchanges involved in a conversation request.

2.3.2.1 Interfaces

The controller's northbound interface (*i.e.*, facing the applications) leverages the DNS protocol. We extend DNS with a new type of *Resource Record* called BSID. This record carries a binding segment implementing a Path ID encoded as an IPv6 address. We also use EDNS0 [DGV15] to carry metadata in DNS messages. We define three new option codes to carry the requested bandwidth, requested latency and application identifier. The flexibility of EDNS0 enables to easily define new options.

In our prototype, the communications between the controller and its components (*i.e.*, southbound interface) are realized through the OVSDB protocol [PD13]. Originally designed for Open vSwitch, OVSDB is a generic,

JSON-RPC based protocol, supporting transactional queries on NoSQL-like databases. However, any per-flow configuration protocol can be used. One can make a different choice to match the protocols available on his access routers (*e.g.*, BGP-LS [Gre+16; Pre+17] or XMPP [Sai11]).

We define five database tables for SRN. We call this set of tables the *Segment Routing Database* (SRDB). `ConvReq` stores the conversation requests generated by the applications and translated by the DNS proxy. It contains the source and destination of the request, the resolved destination address, traffic requirements, and a status. The status is set to `PENDING` when the entry is inserted by the DNS proxy. If the request is accepted by the controller and the conversation is created, the controller changes the status to `ALLOWED`. Otherwise, the status is set to a value that reflects the reason why the conversation was not created (*e.g.*, administrative deny, impossible to satisfy the traffic requirements, etc.). This table is written and read by both the DNS proxy and the controller. `ConvState` stores the state for half-conversations. Each entry contains the source and destination applications, traffic requirements, Path ID and mapped segments, expiration timers, etc. It is written by the controller and read and written by the routing daemon. The write access of the routing daemon is required to enable the removal of expired conversations (*e.g.*, due to idle timeout). `ServReg` stores the server registration requests. It contains the name and address of the requesting servers, and a status field that has the same semantics as for the `ConvReq` table. It is written and read by both the DNS proxy and the controller. `LinkState` and `NodeState` store topology data gathered through OSPF-TE, such as announced prefixes, link utilization, etc. They are written by the NSD and read by the controller.

2.3.2.2 Operations

To ensure the correct operation of the network according to the principles described in Section 2.1, our controller includes at least two processes: conversation requests and server registrations (see Figure 2.1). We also support a third operation: reactions to network events. Those processes operate as follows.

Conversation request. The controller matches each conversation request against the rules defined by the operator. The last matching rule wins. When a rule matches, the controller applies the rule's main decision: accept or deny. In the latter case, an error is returned to the application and the process stops. In the former case, the controller combines the policies defined in the rule and the path constraints provided by the conversation request into a final set of policies. Once the final set of policies is defined, it is translated into a list of segments. Then, the

controller generates a binding segment and creates a route that implements the SR policy. This route matches packets for that particular binding segment and encapsulates them with the previously computed list of segments. This route is immediately inserted using the chosen per-flow configuration protocol (*e.g.*, OVSDB, BGP-LS, XMPP,...) into the access router of the initial requesting application. Note that if the application is susceptible to use more than one access router (*e.g.*, with VRRP [Nad10]), then the route would be inserted in all concerned routers. Finally, the binding segment is returned to the application. The controller keeps the resulting state in memory until expiration. For resiliency purposes, the controller could leverage a dedicated algorithm for backup path computation. Those backup paths should be configured as such in the access router and associated to the same binding segment.

Server registration. When the controller receives a server registration request, it uses the DNS update mechanism [RB97; Wel00]. If there is no pre-existing DNS record for the server name, then a new record is created with a given TTL. If the server is already part of an existing record, then its TTL is refreshed. Otherwise, the server is added to the list of entries associated with this name. The server receives a DNS reply with an associated TTL. The server must refresh its registration before the expiration of the TTL.

Network event. We consider as a *network event* any link or node failure that affects active conversations. We also consider sudden increases in link utilization or delay that would break conversation requirements. The controller does not have the same reaction time for these events. A link failure is quickly propagated by OSPF [Fra+05]. A node failure is detected when all its neighbors have reported the loss of the adjacency, which can take some time. The link bandwidth utilization and delays are updated by using adaptive timers and thresholds by OSPF-TE implementations [Sal+06]. When the controller detects a network event, it scans its conversation state database and builds a list of affected conversations. In case of link or node failure, all conversations that traverse the failed link or node are impacted. In case of congestion or delay increase, the affected conversations are those whose path no longer matches the traffic requirements. For each affected conversation, the controller looks for pre-computed backup paths. If such a path exists and satisfies the requirements, it is selected as replacement. Otherwise, the controller recomputes a new path that suits the conversation policies. The controller updates the conversation and pushes the new state to the impacted routers. In turn, they change their FIB. Note that the

controller can change a path without even interacting with the application, as Path IDs provide a level of indirection to the actual segmented path implemented on the access router. This is the key benefit of such indirection. We described a very basic algorithm for online path recomputation. More advanced approaches (*e.g.*, recompute non-affected paths to reach a more optimal state) discussed in the literature [Egi+12; Qaz+13] could also be included in the controller.

If the controller fails, then many features become unavailable. However, active conversations are not impacted. New conversations can be handled by default SR policies, configured on access routers for packets lacking a binding segment. To avoid switching too quickly to degraded mode, more controllers may be added to the network. Two or more controllers can act in a master-slave fashion.

2.3.3 Fault tolerance

Let us consider the resilience of SRN. If the controller fails, then obviously, many features become unavailable. Applications cannot issue new conversation requests, servers cannot register or refresh their registration and if an adverse network event happens, the affected paths are not updated. Active conversations are not impacted (a controller failure does not affect the routers' FIBs) and server registrations do not immediately expire. However, new conversations can appear at any time but will not be able to complete without the controller. We consider that the network should be able to operate in *degraded mode* without the controller. To realize this, the DNS proxy can be configured to return a default binding segment if the controller is unavailable. If the DNS proxy is itself unavailable, the access routers can be configured with a default policy for ingress packets lacking a binding segment. The behavior of this default SR policy is operator-defined. For example, it may simply forward the packets to their destination, following the IGP's shortest path. Packets may also be forced to pass through a middlebox such as a firewall. Such a mechanism enables the network to function in a temporary degraded mode in case of controller failure.

Operating in degraded mode is never desirable, even if better than a complete network outage. To prevent from switching too rapidly to this mode, more controllers may be added to the network. For example, two or more controllers can act in a master-slave fashion. Furthermore, large or multi-site networks can be partitioned, each partition being assigned its own controller [Ber+14; Dix+14]. In the latter case, a controller failure will affect only its designated network partition, avoiding network-wide degraded mode.

In multi-site networks, assigning a controller to each site prevents the requests to traverse an inter-site WAN link, which could add a significant

latency. State synchronization between controllers of different partitions is minimal, as long as each access router is operated by at most one controller to prevent possible write conflicts. The controllers still need to access each other's conversation state (read-only) to answer reverse Path ID requests, if the two access routers of a conversation are handled by different controllers.

2.3.4 Security

From a security viewpoint, an SRN is exposed to the same security risks as the enterprise's internal DNS resolver. As such, packets from untrusted sources should not be able to reach the controller, as it is already protected by the enterprise firewall.

The introduction of SRv6 in the enterprise network implies the support of the IPv6 Segment Routing header extension. As such, it is crucial that SR-enabled IPv6 packets originating from outside the enterprise network are filtered by the border routers. Otherwise, an external attacker could gain unauthorized access to the enterprise network resources. The mitigation for this threat is simply to configure the border routers to drop all SR-enabled packets originating from an external interface [Fil+21].

Internal threats come from, *e.g.*, malicious employees or applications. They want to get more services from the SDN Resolver than allowed. Two actions can counter such abuses. First, to prevent misuses of network resources, only authorized equipment should be allowed to emit arbitrary SRHs on the network. Network administrators can leverage existing ACLs to prevent unauthorized usage of segments. Access control schemes such as 802.1x can be deployed to control the access to the layer-2 network. Furthermore, the interactions between the clients and the SDN Resolver can be protected by using techniques such as DNS over TLS [Hu+16].

Trusted sources get more services from the SDN Resolver. Malicious applications running on a trusted endhost have access to more services, should they leverage SRN to access them. SRN does not allow differentiating one untrusted application from a trusted one running on a trusted endhost. A potential fix would be to use DNS over TLS [Hu+16]: each application on each endhost would have a client certificate that identifies it to the SDN Resolver.

2.3.5 Supporting legacy devices

To fully leverage the capabilities of SRN, applications need to explicitly interact with the controller. There are legacy applications that do not understand our DNS extensions and/or cannot install the Path ID. In this case, instead of using a binding segment, the Path ID can be implemented as special destination address returned in the DNS reply. Packets using this particular destination are mapped to an SR policy on the access router. Before applying

the policy, the destination address would be translated to the real destination address, using NAT rules dynamically inserted by the controller.

2.4 Prototype Implementation

To assess the performance of our proposed architecture, we developed a fully functional prototype implementation. It runs on Linux clients, routers, servers and controllers. Overall, our prototype comprises about 10,000 lines of C code. We describe the main components of this prototype in this section.

2.4.1 Path ID propagation

In Figure 2.1, after the client has established the connection, the server requests a binding segment for the server-client direction of the conversation. If the controller is able to infer the full requirements of the conversation from the client request, then the server-side request is superfluous. This can speed up the connection establishment process. To realize this, we implement the following solution, illustrated in Figure 2.3.

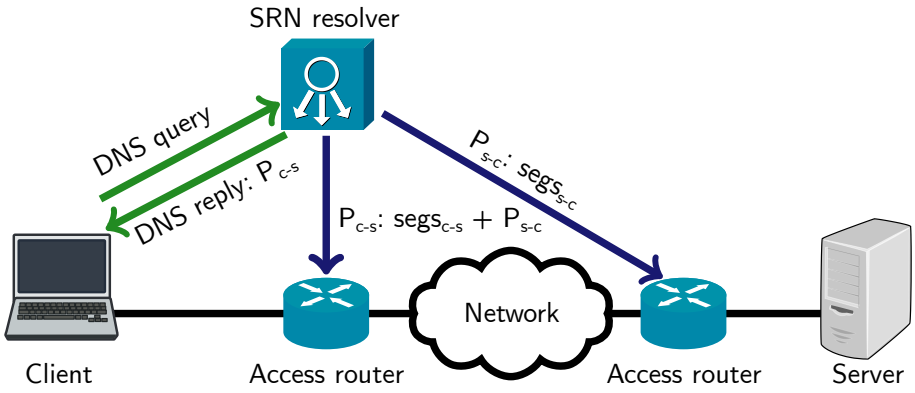

Figure 2.3: Illustration of the return path ID propagation with one conversation request.

When the client issues its conversation request, the controller immediately computes a network path for both directions of the conversation and maps them to two Path IDs (resp. P_{c-s} and P_{s-c} for the client and the server). The controller then inserts a particular SR policy into the client's access router. This SR policy maps P_{c-s} to the corresponding encapsulation, but also instructs the router to *overwrite* P_{c-s} with P_{s-c} in the SRH before the encapsulation. The server then receives a packet with its own Path ID present in the SRH, instead of the client's Path ID. Then, it simply needs to echo the binding segment. Once the conversation is fully established, the server is free to request an update of the conversation with its Path ID, *e.g.*, to reflect

changes in traffic requirements. This technique has obvious security implications. Blindly echoing a binding segment is a process that must be strictly controlled. Allowing only authorized network equipment to emit arbitrary SRHs as explained in Section 2.3.4 should prevent misuses of the feature. Additionally, the HMAC feature of SRv6 can be leveraged to ensure the authenticity and integrity of the SRH.

2.4.2 Controller implementation

The controller is implemented as a heavily multithreaded program. It consists of three main subsystems.

First, the *graph* subsystem implements all the structures, operations and algorithms that deal with the network topology represented as graphs. A graph is represented by a list of nodes and a list of edges. They contain only the minimum information needed to compute the shortest paths with the Dijkstra algorithm. For example, an edge only stores the two nodes it connects as well as its associated metric. The graph nodes and edges contain a generic `data` pointer that can provide additional information (*e.g.*, metric, link latency, bandwidth, etc.). Three helper hashmaps are precomputed to speed up the path computations. The first one (`neighs`) maps each node to its neighbors. The second one (`min_edges`) maps each pair of neighbors to the lowest cost edge that connects them. The last one (`dcache`) maps each node to its precomputed shortest-path Directed Acyclic Graph (SP-DAG). The first two hashmaps speed up the Dijkstra algorithm. The third one speeds up the MinSegECMP algorithm, which requires switching between multiple SP-DAGs. To make graphs friendly to multithreaded environments, each graph is protected by a read-write lock. Moreover, each node and edge structure contains a reference counter. This enables to hold references to them outside critical sections (*i.e.*, when the per-graph lock is not taken).

Second, the *Segment Routing Database* (*SRDB*) subsystem implements an abstraction layer to interact with the OVSDB server. It provides an API for two classes of operations: watching for OVSDB table updates (*monitors*) and performing insertions, updates, and deletions on OVSDB tables (*transactions*). A dedicated thread monitors each table and calls a user-defined callback function whenever a change happens. Transactions are performed by a pool of threads, each maintaining a permanent TCP connection with the OVSDB server. A shared thread-safe buffer is used to pass the transactions to the thread pool. Each transaction consists of the data to transmit as well as a one-element thread-safe result buffer. When the result of the transaction is available, it is pushed into the buffer by the processing thread. Such a producer-consumer architecture enables to (i) scale the number of transaction threads with the database load and (ii) support asynchronous transactions. This sub-

system is also used by other components of the SRN, such as the DNS proxy.

The third subsystem is the *core* of the controller. Using the *graph* and *SRDB* subsystems, the *core* implements the necessary features to run a *Software Resolved Network*. It maintains a *global state* that consists of three sets of data: (*i*) the operator-defined policies, (*ii*) the current network state, and (*iii*) the active conversations. The network state consists of two graphs. A *production* graph is used to perform the path computations, and a *staging* graph is used as a buffer to store the link-state changes. This global state is processed and updated by three major components. The first component is the set of monitoring threads, each of them watching an OVSDB table and calling an associated callback function when necessary. The callbacks for the `NodeState` and `LinkState` tables update the staging graph accordingly. The callback for the `ConvReq` table extracts the conversation request and stores it in a shared thread-safe buffer. This buffer is consumed by the second component, which is a pool of *worker threads*. The worker threads handle most of the controller's workload. Concurrently, they match the requests against the operator policies, compute a path between the source and destination, generate an associated binding segment, create an internal conversation state and commit the newly created conversation to the `ConvState` table. The third component is the network monitoring thread. At configurable intervals, it sets the staging graph as the production graph if the network state changed, it recomputes potentially affected conversations, and garbage collects expired conversations.

2.4.3 Application API

To facilitate the deployment of SR-aware applications, we implement a user API that abstracts the utilization of the DNS extensions and the interactions with the kernel. To support our DNS extensions, we modify the `c-ares` DNS library [C-A]. We implement a custom library, `libsrdns`, that provides an API to handle SR-enabled sockets. For example, the library exposes the `sr_socket()` function that takes as input the socket type (*e.g.*, TCP or UDP), the destination name and port, the local application name, and optional bandwidth and latency parameters. Using this input, the function (*i*) resolves the destination name and fetches any associated binding segment, (*ii*) creates a socket with the corresponding type and (*iii*) attaches an SRH with the binding segment to the socket.

2.5 Evaluation

In this section, we evaluate our SRN through two angles. First, we present the microbenchmarks to measure several aspects of the core controller per-

formance. Then, we evaluate how the SRN behaves when all its components are integrated in a test bed network.

2.5.1 Microbenchmarks

Those evaluations are performed on a four years old laptop using a Core i7-3740QM running at 2.7 GHz with 8 GB of RAM.

2.5.1.1 Conversation requests

The fundamental operations that the controller needs to support is handling conversation requests and generating the corresponding conversation state. They measure how the controller handles conversation requests under a high load. They proceed as follows. First, we initialize an empty OVSDB database with the SRDB schema. They populate the `NodeState` and `LinkState` tables using the Abilene topology [Int], composed of 11 nodes and 15 links. This topology can represent the core of a middle-sized enterprise network. With a benchmarking tool that leverages our SRDB subsystem, we generate batches of conversation requests, at a configurable uniform rate, and without path constraints. The source and destination of each request are selected at random. They measure the delay between the insertion of the request in the `ConvReq` table and the insertion of the conversation state in the `ConvState` table The request completion time is defined as the time elapsed between those two events. As such, it measures (*i*) the insertion of the conversation request in OVSDB, (*ii*) the reception of the `ConvReq` entry by the controller's monitor, (*iii*) the processing of each request and the generation of the conversation state, (*iv*) its insertion in OVSDB, and (*v*) the reception of the `ConvState` entry by the benchmarking tool's monitor.

To estimate the load that an SRN would need to sustain, we looked at the main DNS resolver of our university campus. This campus gathers about 5,000 employees and 28,000 students. Three DNS resolvers serve the campus and the load is well-balanced between them. Their statistics show that the DNS resolvers load never exceeds 1,000 requests per second. We consider this as a baseline for our benchmarks.

In a first batch of measurements, we evaluate how the controller performs over various request rates, using a single worker. The results are shown in Figure 2.4. At 2,000 requests per second, the requests are consistently processed within less than one millisecond. At 4,000 reqs/s, the completion time noticeably increases for more than half of the requests but stays below 10 milliseconds. At 10,000 reqs/s, the controller is unable to cope with the load and the completion time increases by three orders of magnitude.

In a second batch of measurements, we measure the horizontal scaling of the controller with additional workers. They load this controller with 10,000

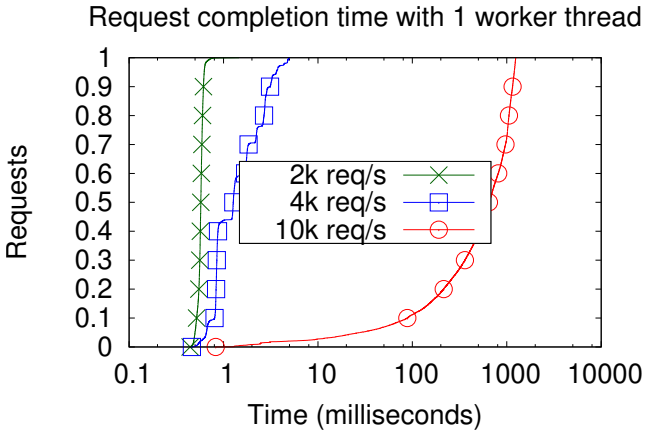

Figure 2.4: A single worker can handle loads of several thousand requests per second.

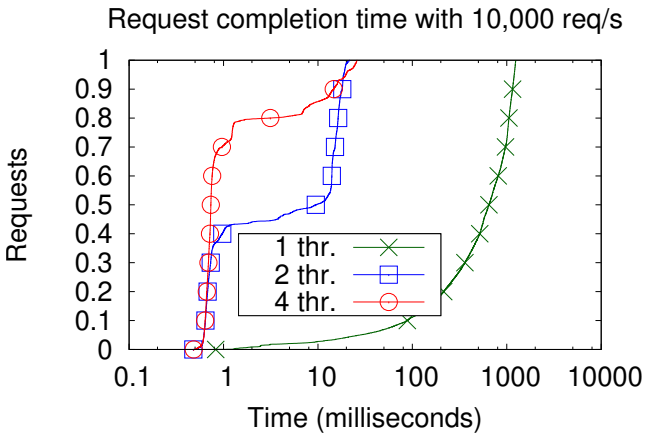

Figure 2.5: Request completion times for various worker threads experiencing 10,000 reqs/s.

reqs/s. The results are shown in Figure 2.5. As previously shown, a single worker is unable to cope with this load and the completion time quickly reaches about one second. Doubling the number of workers (*i.e.*, 2 workers) consistently improves the completion time. About 40% of the requests are completed within less than a millisecond. The remaining 60% require between 10 and 25 milliseconds. By doubling again the number of workers (*i.e.*, 4 workers), about 70% of the requests complete within less than a millisecond. The remaining 30% take between 10 and 25 milliseconds.

In Figure 2.6, we show the evolution of the request completion time for various rates using 4 workers. At 2,000 and 4,000 reqs/s, the request completion time remains below one millisecond. As previously shown, 70% of the

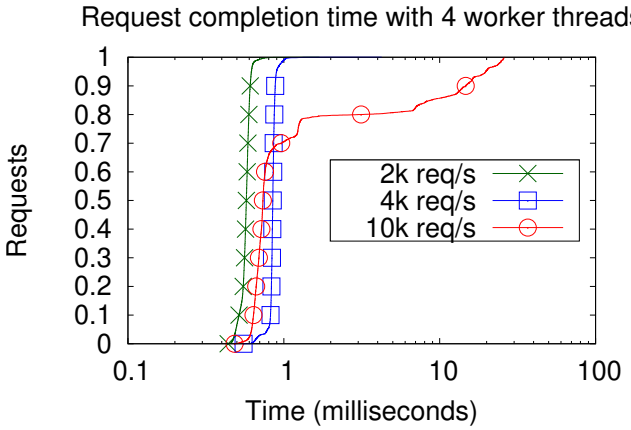

Figure 2.6: Request completion times for 4 various worker threads experiencing various loads.

requests at 10,000 reqs/s also completed within a millisecond.

Existing OpenFlow controllers such as Beacon and NOX have been shown to return responses at a higher rate [Too+12; Eri13; Sha+13]. However, SRN will receive several orders of magnitude less requests than a standard OpenFlow controller. Indeed, only conversation requests (*i.e.*, DNS requests) are handled by SRN. In contrast with an OpenFlow controller, an SRN does not need to act on a per-flow basis. Furthermore, DNS caching is known to perform very well [Jun+02] and directly applies to an SRN.

Those benchmarks are significant for two reasons. First, they show that the controller is able to cope with the typical load of a DNS resolver in a large enterprise network. Furthermore, it efficiently scales with respect to the available CPU. The benchmarks also show that the reference implementation of the OVSDB protocol can sustain such a load. This result is important as it shows that, while OVSDB was not originally designed as a signalling protocol, it can still be used as such without major performance issues, as proposed in [Dav+17].

2.5.2 End-to-end experiments

To evaluate our SRN architecture in an end-to-end setting, we instantiated a test bed with 6 machines with 8 CPU (AMD Opteron 6128). The machines all run on the Linux kernel 4.19. The topology is shown in Figure 2.7. Each link has a bandwidth of 1 Gbps and a sub-millisecond latency. We emulate a latency of 1 ms with tc-netem. Each node is assigned an IPv6 prefix and contains pre-computed shortest-path routes towards all other nodes. The `Controller` node contains the main components of an SRN, *i.e.*, the controller, DNS proxy, and OVSDB server. It also hosts a regular DNS server.

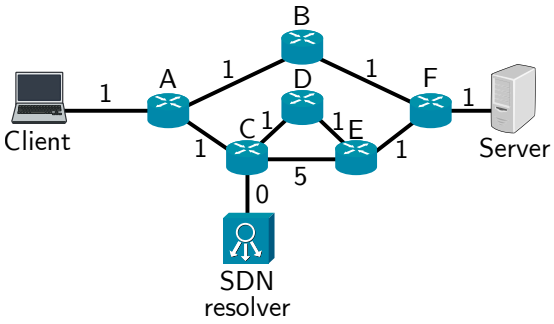

Figure 2.7: Network topology. Unit link weights. Numbers are link delays (ms).

The DNS server maps the `server.test.sr` domain name to the main IPv6 address of the server node. Nodes A and F are access routers and each of them contains a routing daemon.

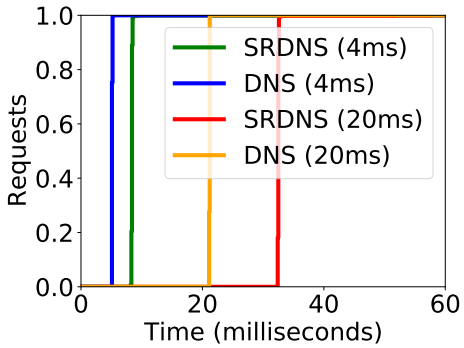

Figure 2.8: Conversation request completion time for various RTTs over 1000 requests.

We generate conversation requests for `server.test.sr` from the client. We measure the time needed to complete the request, as seen by the client. Then, we compare this delay against the time needed to complete a regular DNS request. In a first batch of measurements, we use the network topology as shown in Figure 2.7. The round-trip time between the client and the controller is 4 milliseconds. In a second batch of measurements, we increase the delay of each link between the client and the controller to 5 milliseconds. As a result, the round-trip time between the client and the controller increases to 20 milliseconds. The results are shown in Figure 2.8. We observe that the conversation request completion time consists of one RTT between the client and the controller (*i.e.*, the DNS conversation), and one RTT between the access router and the controller (*i.e.*, the route installation), plus, on average, a constant overhead of 1.3 milliseconds. This overhead can be explained by

the component communication. Note that 20ms for the RTT between the controller and the client are not realistic in an enterprise network but this is useful to identify the causes of the delay.

2.6 Related Work

Software Defined Networks have been a hot topic in the research community since the publication of [McK+08]. Several survey papers have analyzed this vast literature in details [Kre+15; FRZ13; Xia+15]. In this section, we compare Software Resolved Networks (SRN) with many of the key related work, namely, Integrated Services [BCS13], Differentiated Services [Wan+13], Ethane (as described in [McK+08]), and PANE [Fer+13]. This related work is explained in Chapter 1.

Integrated Services and Differentiated Services offer Quality of Service but do not change the paths of the traffic. They are limited by the capacity of the shortest path and not the capacity of the whole network. The other solutions, including our own, use the whole network to provide Quality of Service. Ethane and PANE use the OpenFlow protocol to be able to choose any path in their network. SRN uses IPv6 Segment Routing for that.

An important difference with the state of the art is the source of overheads. Integrated Services use RSVP to reserve resources for each privileged flow on every router. Therefore, Integrated Services does not scale on network with a lot of traffic. Differentiated Services instead classifies the flows into pre-defined classes. The state on the core routers scales with the number of classes (12 are defined in [BBC06]). Ethane and PANE rely on the OpenFlow protocol to configure flow tables on the switches. In SRNs, the controller interacts with the routers through OVSDB tables. Note that while traditional SDN networks require the installation of flow tables on all network devices, in SRNs the controller only interacts with the edge routers. The controller does not need to interact with the other routers which improves the scalability of SRNs. Because SRNs use IPv6 Segment Routing, only one of the edge routers needs an extra route for each accepted conversation request. In Ethane or PANE, every OpenFlow switch on the path needs an extra forwarding rule. The header added by SRv6 creates an overhead that slightly reduces the usable bandwidth. To keep this overhead low, SRNs use MinSegECMP path segmentation algorithm that bounds the number of segments used to generate the paths (see Section 2.3.1).

There are other alternatives to OpenFlow or IPv6 Segment Routing. Several SDN solutions also encode paths inside packets. Hari et al. propose in [HLW15] to encode network paths inside the layer-2 MAC addresses of the packets on the first switch of the path. Jeyakumar et al. propose in [Jey+15] to leverage the 20-bit flow label field in the IPv6 header and the 6-bit DS field

in the IPv4 header to dynamically parametrize middleboxes. IPv6 Segment Routing has the advantage to be available in the Linux kernel since its version 4.12. It would be possible to replace IPv6 Segment Routing by MPLS Segment Routing. However, this is sounder to deploy IPv6 in a IPv4 dataplane to tackle the exhaustion of the IPv4 address space and to benefit from Segment Routing. OpenFlow is more costly to deploy on existing networks than IPv6 Segment Routing because the operators cannot reuse existing routers, they need to buy OpenFlow switches.

Another important difference is that the endhosts participate actively in SRNs with their DNS requests. This implies that our SDN resolvers can use policies based on DNS names and not only addresses and ports. Other SDN solutions such as PANE [Fer+13] use a related approach. PANE offers an API enabling the applications to interact with the controller. This API is implemented with a new protocol, while SRNs extend the DNS protocol. Beyond SDN and enterprise networks, researchers have proposed several architectures where content is retrieved directly with names [Kop+07; Jac+09].

It is more efficient for SRN to use DNS to translate the DNS names of the conversation requests into IP addresses. If SRN used a custom protocol, the DNS request would need to be made by either the client or the controller. In the first case, this lengthens the request latency duration because we need to have the reply before sending the conversation request. In the second case, this doubles the number of requests on the DNS servers which is inefficient.

2.7 Conclusion

The goal of this thesis is to allow the network and the endhosts to collaborate and use this collaboration to reap path diversity benefits while using single-path protocols. In this chapter, we proposed Software Resolved Networks, an SDN-like architecture for enterprise networks. Like OpenFlow-based SDN solutions, SRNs enable the network operators to specify policies that control the network paths that are used by trusted applications. For this, Software Resolved Networks build upon the IPv6 Segment Routing architecture (SRv6) and the DNS protocol. Applications use the DNS as a signalling protocol to request the creation of end-to-end network paths. Those paths are computed by a controller that interacts with the DNS resolver and returns the selected path to the requesting application as a segmented SRv6 path.

There are two important differences between SRNs and traditional SDNs. First, SRNs enable the applications to directly interact with the controller to specify path requirements like delay or bandwidth. However, this requires to modify the code of these applications to include our library. Second, SRNs do not require per-flow state in all network nodes. The controller installs state on the access routers but not on the core routers. This improves the scalability

of SRNs.

Even if this chapter focused on enterprise networks, SRN's architecture is applicable to datacenters as well. They also have applications that could benefit from signalling their requirements to the controller.

We implement a complete Software Resolved Network [2] on Linux hosts, routers and servers. Our performance evaluation demonstrates that our prototype meets the performance expectations of enterprise networks.

[2] The code of our prototype is available at `https://github.com/segment-routing/sdnres`.

Near-Optimal Traffic Engineering with CG4SR

3

3.1 Introduction

Wide area networks are a key element of our Internet-centric society. Most of them are managed by Internet Service Providers (ISPs) and large enterprises that use them to interconnect their datacenters. Building and operating these networks is costly, and their owners rely on a variety of traffic engineering techniques [Awd+02; Wan+08] to optimize them. These techniques range from adjusting link weights based on traffic patterns [FRT02], using a centralized controller in a Software Defined Network [Jai+13; BM15] to using MPLS to control routing paths [Xia+00; Auk+00].

IPv6 Segment Routing [Brz+18; Fil+18a] encodes these paths inside each packet as a series of segments. The added routing flexibility of IPv6 Segment Routing makes it possible to better utilize the network to its full capacity without incurring the overhead that one gets by using MPLS as there is no need to maintain a global state or use specialized signalling protocols such as LDP or RSVP-TE. However, traffic engineering with Segment Routing requires the development of specific algorithms. In practice, deployed routers have hardware limitations on the number of segments that they can forward [Tan17].

The goal of this thesis is to propose and explore the use cases of a deployable architecture allowing endhosts to control their paths. In this chapter, we focus on the development of efficient traffic engineering algorithms that leverage the unique characteristics of Segment Routing that can be used in the SRN controller presented in Chapter 2. More precisely, we propose a new scalable approach that relies on column generation [DL05]: CG4SR. Moreover it can provide strong guarantees on the quality of the solutions by computing a lower-bound that is in most of the cases tighter than the one obtained with a multicommodity flow relaxation. We evaluate CG4SR in ISP networks because they are the most challenging due to the high number of available paths to choose from.

In developing CG4SR, we make the following contributions.

CG framework for SR: We propose the first column generation framework for solving TE with Segment Routing.

Fully leverage SR: Our solution is able to leverage Segment Routing to the full extent of its potential as we have full control over the number of segments that are used, and we support both node and adjacency segments.

Improved lower bound: We improve the lower bound provided by traditional Multi-Commodity Flow approaches by up to 15% for TE with Segment Routing.

Min cost path SR-problem: As a by-product of our solution, we developed a general and efficient algorithm for computing Segment Routing paths of minimum cost.

The remainder of this chapter is organized as follows. In Section 3.2, we give describe the main traffic engineering techniques that have been proposed. Then, in Section 3.3, we present the intuition behind column generation. In Section 3.4, we formalize the traffic engineering problem and propose a column generation formulation to solve it. In Section 3.5, we provide efficient algorithms to solve the subproblems involved in our formulation and a detailed description of our column generation algorithm. In Section 3.6, we provide a thorough evaluation of our solution, comparing it to the state-of-the-art traffic engineering algorithms for Segment Routing.

3.2 Traffic Engineering for Segment Routing

A simple formulation of the Traffic Engineering (TE) problem is the Multi-Commodity Flow (MCF) linear programming formulation [AMO93]. Unfortunately, formulating TE as an MCF has several drawbacks. First, the formulation uses $O(|V|^2 \cdot |\mathcal{D}|)$ variables making it unusable for large topologies. Second, this formulation is not designed to be used with Segment Routing and can thus output any set of network paths. In practice, deployed routers have hardware limitations on the number of segments that they can forward [Tan17].

In the recent years, a few SR centric approaches for optimizing TE have been proposed [Har+15b; Har+15a; GHV17; Bha+15]. Complete approaches to optimize exactly traffic engineering using Segment Routing are unable to tackle the general problem as they suffer from the same scaling problem as the MCF formulation mentioned above. Bhatia et al. propose a solution addressing the scalability issue by considering a subset of the possible Segment Routing paths (sr-paths) with one intermediate segment only [Bha+15]. Trimponias et al. propose to restrict the set of allowed segments to a subset of the network [Tri+17]. Other researchers have proposed heuristic techniques. Hartert et al. proposed in DEFO [Har+15b; Har+15a] a Large Neighborhood Search technique combined with Constraint Programming. Later, Gay et al. proposed [GHV17] to use standard local search to iteratively improve the current solution. These heuristic approaches are rather efficient but provide no

way of knowing how far from the optimal value they end up. Moreover, all the existing approaches only consider node segments in their formulation. They are thus not able to fully exploit the flexibility of Segment Routing with adjacency segments.

3.3 Column Generation Intuition

This section details the intuition behind Column Generation (CG) framework [DL05]. Its goal is to solve linear programs with numerous variables faster by carefully selecting the interesting variables to consider. Let us consider the following linear program:

$$
\begin{array}{lccccccccc}
LP(\mathcal{X}) & & & & & & & & & \\
\max & c_1x_1 & + & c_2x_2 & + & \dots & + & c_nx_n & & \\
\text{s.t.} & a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & \leq & b_1 \\
& a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & \leq & b_2 \\
& \vdots & & \vdots & & & & \vdots & & \vdots \\
& a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & \leq & b_m
\end{array}
$$

The idea is to start from a subset $\mathcal{X}'$ of all variables $\mathcal{X}$ when trying to optimize this very large linear program. For instance, if $\mathcal{X}'$ is a set of 3 variables, the linear program becomes:

$$
\begin{array}{lccccccc}
LP(\mathcal{X}') & & & & & & & \\
\max & c_1x_1 & + & c_2x_2 & + & c_3x_3 & & \\
\text{s.t.} & a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & \leq & b_1 \\
& a_{21}x_1 & + & a_{22}x_2 & + & a_{23}x_3 & \leq & b_2 \\
& \vdots & & \vdots & & \vdots & & \vdots \\
& a_{m1}x_1 & + & a_{m2}x_2 & + & a_{m3}x_3 & \leq & b_m
\end{array}
$$

Following, the column generation framework, we solve a linear program solver with this subset of variables $\mathcal{X}'$. Then CG finds a promising variable and resolve again the smaller linear program with this added variable. CG keeps adding new variables until it has the guarantee that there is no other variable that would improve the solution. The point is that there are variables that are never explored, and therefore some time is saved, despite the multiple linear program resolutions.

Identifying good new variables and when to stop is key to Column Generation. This is possible by using the strong duality property (valid for any linear program): each linear program (called the primal problem) has a dual form, called the dual problem. The set of optimal solutions to the primal problem is the same set of optimal solutions to the dual problem. The dual problem

and the conversion between optimal solutions of the dual and primal can be obtained following a systematic procedure.

$$
\begin{array}{lccccccccc}
LP-DUAL(\mathcal{Y}) & & & & & & & & & \\
\min & b_1y_1 & + & b_2y_2 & + & \dots & + & b_my_m & & \\
\text{s.t.} & a_{11}y_1 & + & a_{12}y_2 & + & \dots & + & a_{m1}y_m & \geq & c_1 \\
 & a_{12}y_1 & + & a_{22}y_2 & + & \dots & + & a_{m2}y_m & \geq & c_2 \\
 & \vdots & & \vdots & & & & \vdots & & \vdots \\
 & a_{1n}y_1 & + & a_{2n}y_2 & + & \dots & + & a_{mn}y_m & \geq & c_n
\end{array}
$$

Each constraint of the dual problem corresponds to a variable of the primal. Therefore, building the dual problem ($LP-DUAL'(\mathcal{Y})$) of $LP(\mathcal{X}')$ is made by ignoring the dual constraints tied to the primal variables.

$$
\begin{array}{lccccccccc}
LP-DUAL'(\mathcal{Y}) & & & & & & & & & \\
\min & b_1y_1 & + & b_2y_2 & + & \dots & + & b_my_m & & \\
\text{s.t.} & a_{11}y_1 & + & a_{12}y_2 & + & \dots & + & a_{m1}y_m & \geq & c_1 \\
 & a_{12}y_1 & + & a_{22}y_2 & + & \dots & + & a_{m2}y_m & \geq & c_2 \\
 & a_{13}y_1 & + & a_{23}y_2 & + & \dots & + & a_{m3}y_m & \geq & c_3
\end{array}
$$

Let's assume that we found the optimal solution $x*$ for $LP(\mathcal{X}')$. Then we can derive the optimal solution $y*$ for $LP-DUAL'(\mathcal{Y})$. Moreover, $y*$ is a solution for the unrestricted dual problem, $LP-DUAL(\mathcal{Y})$, if and only if $y*$ does not violate any additional constraints. Since, each dual's constraint corresponds to a primal's variable, finding a constraint that is not respected, means that we need to add the corresponding variable to X' before resolving $LP(\mathcal{X}')$.

Checking every constraint of the dual problem would defeat the purpose of CG since there are as numerous as the primal's variables. Instead, we focus on finding the index i of the constraint that is the most violated:

$$\sum_{j=1}^{m} a_{ji}y_j \geq c_i \Leftrightarrow \sum_{j=1}^{m} a_{ji}y_j - c_i \geq 0$$

Therefore, we want to find the index i that minimizes $\sum_{j=1}^{m} a_{ji}y_j - c_i$. This minimization is the pricing problem of the Column Generation framework. If the optimum is negative, the constraint i is violated, and we need to add the corresponding primal variable. If the optimum is positive, no constraint is violated, and we stop adding new variables. $y*$ is a solution for $LP-DUAL(\mathcal{Y})$ and its corresponding primal solution, $x*$ is a solution for $LP(\mathcal{X})$ by the strong duality. CG idea works if solving this pricing problem repeatedly is faster than solving the initial problem with all of its variables.

$x*$ is a solution to a continuous linear programming. To obtain a optimal integer program, we would need to add another layer to the algorithm, and perform, e.g., a branch-and-price [Bar+98]. However, in this thesis, we opted for a heuristics approach detailed in the rest of the chapter. It would be interesting to compare our heuristics approach to a classic branch-and-price above column generation.

3.4 Formalization

We model the network as a directed graph $G = (V, E, igp, c)$ where $igp : E \to \mathbb{Z}^+$ represents the IGP weights configured on the links and $c : E \to \mathbb{Z}^+$ represents the link capacities. We define the size of G as $|G| = |V| + |E|$.

The demands are simply modeled as a set $\mathcal{D}$ of triples (s, t, v) meaning that we have a volume v of traffic to route from s to t. For a given demand $d \in \mathcal{D}$ we write $s(d) = s$, $t(d) = t$ and $v(d) = v$.

We model sr-paths as sequences $\langle x_1, x_2, \ldots, x_n \rangle$ where each x_i is either an element of V, representing a node segment, or an element of E, representing an adjacency segment. If $x_i = (u, v)$ is an adjacency segment we denote $x_i^1 = u$ and $x_i^2 = v$. It is convenient to have a uniform notation to avoid having to distinguish between node and adjacency segments later on. For this reason, we also define for a node segment x_i the notations $x_i^1 = x_i^2 = x_i$. We denote the set of indexes of node segments by $node(p)$ and the set of indices of adjacency segments by $adj(p)$. Sr-paths actually represent a subgraph of the network since several shortest paths may exist between consecutive node segments. More specifically, $SP(x_{i-1}^2, x_i^1)$ represents the set of edges on a shortest path between nodes x_{i-1}^2 and x_i^1. We denote the set of edges that belong to the subgraph represented by a sr-path p by $E(p) = adj(p) \cup \left(\bigcup_{i=2}^{n} SP(x_{i-1}^2, x_i^1)\right)$.

Some routers support only a limited number of segments in the packet header [Tan17]. Therefore, our TE algorithms should be able to limit the number of segments. Adjacency segments are local segments which can only be parsed by the link origin. They require storing both the node segment to reach the link origin and the actual adjacency segment identifying the router interface. Hence, they cost twice as much as node segments.

The segment cost of a sr-path is thus the number of node segments plus twice the number of adjacency segments. We denote the segment cost of a sr-path p from s to t by $seg(p) = |node(p) \setminus \{s\}| + 2 \cdot |adj(p)|$ and the set of all sr-paths with segment cost at most k by $\overline{\mathcal{P}}(k) = \{p \mid seg(p) \leq k\}$. Note that the source node is not counted as it does not need to be added to the segment stack.

We say that a sr-path $p = \langle x_1, \ldots, x_n \rangle$ is compatible with a demand d if and only if $x_1^1 = s(d)$ and $x_n^2 = t(d)$. That is, p starts and ends at the source and destination nodes of d, respectively. We denote the set of demands compatible

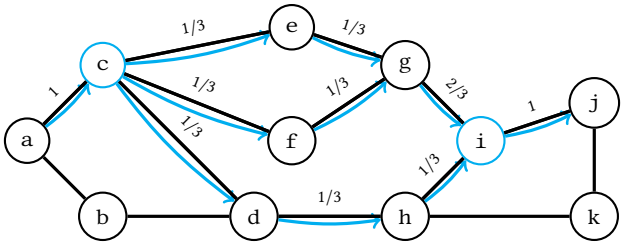

Figure 3.1: Illustration of the split ratio for sr-path $\langle a, c, i, j \rangle$. The figure assumes unit IGP weights.

with p by $\mathcal{D}(p)$.

When a demand is routed over a sr-path, it is split equally whenever there is ECMP between two consecutive segments. As illustrated in Figure 3.1, the traffic is first sent from node a to c without any split as there is a single shortest path. Then, between segments c and i there are three shortest paths, so the demand is split equally between them. The edge labels represent the ratio of the demand that will traverse each edge. This motivates the definition of $r(x, y, e)$ which represents the load generated on edge e per unit of traffic forwarded on the shortest paths between nodes x and y. If e does not belong to $SP(x, y)$ then $r(x, y, e) = 0$. More generally we define the utilization ratio on an edge for a given sr-path $p = \langle x_1, \ldots, x_n \rangle$ as

$$r(p, e) = \sum_{i=2}^{n} r(x_{i-1}^2, x_i^1, e) + |\{i \in adj(p) \mid x_i = e\}|.$$

The first part of the expression evaluates the ratio on e for the shortest path subgraphs between consecutive segments, while the second part counts how many times e is used as an adjacency segment on the sr-path p. This summation is valid because when an adjacency segment occurs, the full unit demand is routed through the edge without split.

3.4.1 Problem formulation

The Segment Routing Traffic Engineering (SRTE) problem is to discover a sr-path p for each demand such that the worst link utilization (total traffic/capacity) is minimized. This problem was demonstrated to be NP-hard [Har+15b].

We propose a scalable mathematical programming approach for solving the general SRTE problem with segments up to cost k also allowing adjacency segments. Our approach relies on the so-called Column Generation (CG) decomposition approach [DDS06] for solving huge linear programs. CG introduces one decision variable per column and defines the objective function

to maximize as a linear combination of those column variables. For SRTE, columns correspond to candidate sr-paths that can be selected for each demand. The minimization of maximum link utilization is not directly a good fit for a CG formulation as our experiments showed that it had difficulties to converge. Therefore, we instead solve a close variant of this problem seeking to assign demands to sr-paths so that the amount of demand volume routed is maximal without exceeding the link capacities above a given threshold and later adapt it to minimize the maximum link utilization. We call this problem SRTED.

Considering the whole set of sr-paths $\overline{\mathcal{P}}(k)$ upfront would not scale as it would require solving a linear program with a huge number of candidate paths and variables. Therefore, a restricted set of sr-paths $\mathcal{P} \subseteq \overline{\mathcal{P}}(k)$ is initially considered (called the columns in the CG framework) and then is iteratively grown by carefully selecting new candidate paths from $\overline{\mathcal{P}}(k)$ until we obtain a proof that no matter what new paths would be added, the solution would not be further improved. The linear program working on the restricted set of sr-paths $\mathcal{P}$ is called the *master problem*. Once the master problem is solved optimally, new interesting paths can be added incrementally to the restricted set $\mathcal{P}$ by solving a sub-problem called the *pricing problem*. This pricing problem amounts to discovering new columns (sr-paths) with negative reduced costs in the master problem, meaning that if added to the path set, they could potentially improve the current solution. The process of interleaving the master problem solving and the pricing problem continues until the pricing problem fails to discover negative reduced cost columns. At that point, the master problem, although working on a restricted set of sr-paths is proven optimal with respect to every possible path in $\overline{\mathcal{P}}(k)$. The advantage of CG is thus to solve tight linear program formulations with an exponential number of variables without even considering them all explicitly in practice.

The ILP formulation of the SRTED problem which maximizes the volume of the demands, that are routed through the set of sr-paths $\mathcal{P}$, depends on three parameters: the sr-path set $\mathcal{P}$, the demand set $\mathcal{D}$ and a capacity factor $\lambda \in \mathbb{R}^+$. This capacity factor is used to control by which factor we allow the edge capacities to be exceeded. A binary variable is introduced for each pair of compatible sr-path p and demand d:

$$\mathbf{x}_{dp} = \begin{cases} 1 & \text{if demand } d \text{ is routed over sr-path } p \in \mathcal{P} \\ 0 & \text{otherwise} \end{cases}$$

The ILP is then formally defined as:

$$\text{SRTED-ILP}(\mathcal{P},\mathcal{D},\lambda)$$

$$\max_{\mathbf{x}} \quad \sum_{p\in\mathcal{P}} \sum_{d\in\mathcal{D}(p)} v(d)\cdot \mathbf{x}_{dp}$$

$$\text{s.t.} \quad \sum_{p\in\mathcal{P}} \mathbf{x}_{dp} \leq 1 \qquad \forall d\in\mathcal{D}$$

$$\sum_{p\in\mathcal{P}} \sum_{d\in\mathcal{D}(p)} r(e,p)\cdot v(d)\cdot \mathbf{x}_{dp} \leq \lambda\cdot c(e) \qquad \forall e\in E$$

$$\mathbf{x}_{dp} \in \{0,1\} \qquad \forall p\in\mathcal{P}, \forall d\in\mathcal{D}(p)$$

The objective function seeks to route a maximum demand volume over the sr-paths $\mathcal{P}$. The first set of constraints ensures that at most one sr-path is selected for each demand and the second set of constraints makes sure that the capacity of an edge is not exceeded by a factor greater than λ. If $\lambda = 1$ then the problem routes the maximum amount of demand volume without exceeding any link capacities. Our model may thus discard some demands if the parameter λ is too restrictive. Discovering the minimal value for λ, such that all demands are routed, can be achieved with a dichotomic search on top of the CG framework as discussed later.

In order to be able to use LP duality theory and ultimately column generation, we consider the LP relaxation of SRTED-ILP$(\mathcal{P},\mathcal{D},\lambda)$ which we name SRTED-LP$(\mathcal{P},\mathcal{D},\lambda)$. It has the exact same formulation except that the binary variables $\mathbf{x}_{dp}$ are allowed to have fractional values in $[0,1]$.

It is known that any LP model has a dual problem which can be obtained by following a systematic procedure [Gas03]. Doing so, we obtain the following dual problem:

$$\text{SRTE-DUAL}(\mathcal{P},\mathcal{D},\lambda)$$

$$\min_{\mathbf{y},\mathbf{z}} \quad \lambda\cdot\sum_{e\in E} c(e)\cdot \mathbf{y}_e + \sum_{d\in\mathcal{D}} \mathbf{z}_d$$

$$\text{s.t.} \quad \mathbf{z}_d + \sum_{e\in E(p)} r(e,p)\cdot v(d)\cdot \mathbf{y}_e \geq v(d) \qquad \forall p\in\mathcal{P}, \forall d\in\mathcal{D}(p)$$

$$\mathbf{y}_e,\mathbf{z}_d \geq 0 \qquad \forall p\in\mathcal{P}, \forall d\in\mathcal{D}(p)$$

To simplify the notations we use the names of the problems defined above to refer both to the problem itself and to the value of their optimal solution.

As a consequence of the strong duality theorem for linear programming we have that

$$\text{SRTED-LP}(\mathcal{P}, \mathcal{D}, \lambda) = \text{SRTE-DUAL}(\mathcal{P}, \mathcal{D}, \lambda).$$

and that whenever we solve SRTED-LP$(\mathcal{P}, \mathcal{D}, \lambda)$ we obtain a solution $\mathbf{x}$ together with an optimal solution of SRTE-DUAL$(\mathcal{P}, \mathcal{D}, \lambda)$ that we denote by $(\mathbf{y}, \mathbf{z})_{\mathbf{x}}$.

The next theorem is at the heart of the pricing problem. It gives the necessary condition for a sr-path to be considered as interesting for inclusion in the restricted set $\mathcal{P}$.

Theorem 1 *Let $\mathcal{P} \subseteq \overline{\mathcal{P}}(k)$, $\mathbf{x}$ be an optimal solution of SRTED-LP$(\mathcal{P}, \mathcal{D}, \lambda)$ and $(\mathbf{y}, \mathbf{z})_{\mathbf{x}}$ the corresponding solution of SRTE-DUAL$(\mathcal{P}, \mathcal{D}, \lambda)$. Then, $\mathbf{x}$ is optimal for SRTED-LP$(\overline{\mathcal{P}}(k), \mathcal{D}, \lambda)$ if and only if for all $p \in \overline{\mathcal{P}}(k), d \in \mathcal{D}(p)$,*

$$\mathbf{z}_d + \sum_{e \in E(p)} r(e,p) \cdot v(d) \cdot \mathbf{y}_e \geq v(d). \tag{3.1}$$

Suppose that for all $p \in \overline{\mathcal{P}}(k), d \in \mathcal{D}$ inequality (3.1) holds. Then this means that $(\mathbf{y}, \mathbf{z})_{\mathbf{x}}$ is also an admissible solution of SRTE-DUAL$(\overline{\mathcal{P}}(k), \mathcal{D}, \lambda)$. Since $\mathcal{P} \subseteq \overline{\mathcal{P}}(k)$ we have that SRTE-DUAL$(\mathcal{P}, \mathcal{D}, \lambda)$ is a relaxation of SRTE-DUAL $(\overline{\mathcal{P}}(k), \mathcal{D}, \lambda)$. Therefore SRTE-DUAL$(\mathcal{P}, \mathcal{D}, \lambda) \leq$ SRTE-DUAL$(\overline{\mathcal{P}}(k), \mathcal{D}, \lambda)$. Hence, since $(\mathbf{y}, \mathbf{z})$ is an admissible solution of both problems, we have that SRTE-DUAL $(\mathcal{P}, \mathcal{D}, \lambda) =$ SRTE-DUAL$(\overline{\mathcal{P}}(k), \mathcal{D}, \lambda)$. Hence, by strong duality,

$$\begin{aligned} \textit{SRTED-LP}(\mathcal{P}, \mathcal{D}, \lambda) &= \textit{SRTE-DUAL}(\mathcal{P}, \mathcal{D}, \lambda) \\ &= \textit{SRTE-DUAL}(\overline{\mathcal{P}}(k), \mathcal{D}, \lambda) \\ &= \textit{SRTED-LP}(\overline{\mathcal{P}}(k), \mathcal{D}, \lambda). \end{aligned}$$

In practice, what this means is that if we solve SRTED-LP to optimality for a given subset of sr-paths $\mathcal{P}$, demands $\mathcal{D}$ and λ, then we can decide whether this solution is optimal over the set of *all* sr-paths $\overline{\mathcal{P}}(k)$ by checking whether there exist some sr-path p and demand $d \in \mathcal{D}(p)$ that violate inequality (3.1). This amounts to finding a pair (p, d) such that

$$\mathbf{z}_d + \sum_{e \in E(p)} r(e,p) \cdot v(d) \cdot \mathbf{y}_e < v(d). \tag{3.2}$$

Given $(\mathbf{y}, \mathbf{z})$, the Pricing problem is to find a sr-path $p \in \overline{\mathcal{P}}(k)$ and a demand $d \in \mathcal{D}$ such that (3.2) holds. Note that solving this problem by iterating over all sr-paths $p \in \overline{\mathcal{P}}(k)$ is unrealistic since $|\overline{\mathcal{P}}(k)| = O(|G|^k)$.

Next section introduces an efficient algorithm to solve the Pricing problem. We show how to integrate it into an algorithm that solves SRTED-LP $(\overline{\mathcal{P}}(k), \mathcal{D}, \lambda)$.

3.5 Algorithm

3.5.1 Solving the Pricing problem

In order to solve the Pricing problem, we developed a general Dynamic Programming (DP) algorithm for solving the following, more general, Segment Routing path finding problem.

Problem 1 ***(SR-Min cost path problem)***[1] *Compute a sr-path of minimum cost between two nodes with a given segment budget. Let $G = (V, E)$ be a graph. Given two cost functions $c_{sp} : V \times V \to \mathbb{R}^+$ and $c_{adj} : E \to \mathbb{R}^+$, a constant k and $s, t \in V$. Find a sr-path $p = \langle x_1, \ldots, x_n \rangle$ from s to t such that $seg(p) \leq k$ and*

$$c(p) = \sum_{i=2}^{n} c_{sp}(x_{i-1}^2, x_i^1) + \sum_{i \in adj(p)} c_{adj}(x_i)$$

is minimum. The cost function c_{sp} represents the cost of using the shortest path DAG between any two given nodes and the c_{adj} represents the cost on the edges used to evaluate adjacency segments.

In order to solve this problem, let us define the following DP state space

$$dp(i, x) = \text{the minimum weight (w.r.t. } c\text{) sr-path from } s \text{ to } x \text{ with segment cost at most } i.$$

The solution to the SR-min cost path problem is $dp(k, t)$. For $i = 0$ the only possible sr-path of segment cost 0 from s to any node is $\langle s \rangle$. Thus, we have that $dp(0, s) = 0$ and $dp(0, x) = \infty$ for $x \neq s$. For $i \geq 1$ it can be shown that

$$dp(i, x) = \min \begin{cases} dp(i-1, x) & & & & \\ dp(i-1, y) & + & c_{sp}(y, x) & s.t & y \in V \\ dp(i-2, r) & + & c_{sp}(r, z) & s.t & r \in V, \\ & + & c_{adj}(z, x) & & (z, x) \in E \end{cases}$$

where the third value is only defined for $i \geq 2$ (set to ∞) otherwise.

The intuition behind this recurrence is that there are three possibilities to reach a node x with a sr-path of segment cost at most i. These three possibilities are illustrated in Figure 3.2. The first way is to simply not use the extra segment and reach x in the best way using a sr-path of segment cost at most $i-1$. The second possibility is to reach some node $y \in V$ using a sr-path

[1] In addition to solving our Pricing problem, this general problem could also be used for instance to compute a minimum latency sr-path between two nodes in a network.

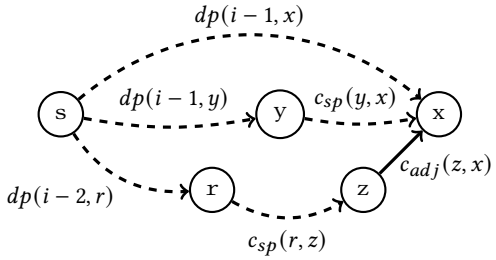

Figure 3.2: Illustration of the dp recurrence.

of segment cost at most $i-1$ and then use one extra node segment to reach x incurring a cost of $c_{sp}(y,x)$. Finally, we can use a sr-path of cost at most $i-2$ to any node r and then append to it an adjacency segment (z,x) where z is a neighbor of x. Note that nothing prevents r from being equal to z. In this case it simply means that we append the adjacency (z,x) to a path that already ends at node z.

Since the values of $dp(i,*)$ depend only on $dp(i-1,*)$, we can compute all these values by iterating in increasing order of i. There are $O(k\cdot|V|)$ states to compute and evaluating each of them takes $O(|V|)$, so the total complexity of the algorithm is $O(k\cdot|V|^2)$.

We now show how to cast the Pricing problem as the one of solving one minimum cost sr-path problem per demand.

Let $d\in\mathcal{D}$ be a demand and $(\mathbf{y},\mathbf{z})_{\mathbf{x}}$ be the dual values corresponding to a primal solution $\mathbf{x}$. Define $c_{sp}(x,y)=\sum_{e\in E} r(e,x,y)\cdot \mathbf{y}_e$ (if $x=y$ we set $c_{sp}(x,y)=0$) and $c_{adj}(e)=\mathbf{y}_e$. The above dynamic program computes a minimum cost sr-path $p=\langle x_1,\dots,x_n\rangle\in\overline{\mathcal{P}}(k)$ from $s(d)$ to $t(d)$ with respect to c_{sp} and c_{adj} that minimizes

$$
\begin{aligned}
c(p) &= \sum_{i=2}^{n} c_{sp}(x_{i-1}^2, x_i^1) + \sum_{i\in adj(p)} c_{adj}(x_i)\\
&= \sum_{i=2}^{n}\sum_{e\in E} r(e,x_{i-1}^2,x_i^1)\cdot \mathbf{y}_e + \sum_{i\in adj(p)} \mathbf{y}_{x_i}\\
&= \sum_{e\in E}\sum_{i=2}^{n} r(e,x_{i-1}^2,x_i^1)\cdot \mathbf{y}_e + \sum_{e\in E}\sum_{i\in adj(p):x_i=e}^{n} \mathbf{y}_e\\
&= \sum_{e\in E}\left(\sum_{i=2}^{n} r(e,x_{i-1}^2,x_i^1) + |\{i\in adj(p)\mid x_i=e\}|\right)\mathbf{y}_e\\
&= \sum_{e\in E} r(e,p)\cdot \mathbf{y}_e.
\end{aligned}
$$

Thus, there exists a sr-path p that satisfies inequality (3.2) with respect to demand d if and only if $c(p) < \frac{v(d)-\mathbf{z}_d}{v(d)}$. This means that we can solve the Pricing problem by solving the minimum cost sr-path problem for each demand d. The total time complexity of this algorithm is $O(k \cdot |\mathcal{D}| \cdot |V|^2 + |V|^2 \cdot |E|)$ where the second term represents the cost of precomputing the cost function c_{sp}. This shows that we can solve the pricing problem in polynomial time.

3.5.2 The column generation algorithm

Algorithm 1 describes the column generation algorithm solving optimally SRTED-LP$(\mathcal{P}, \mathcal{D}, \lambda)$. The set $\mathcal{P}$ contains any set of initial paths. We use $\mathcal{P} = \{\langle s(d), t(d)\rangle \mid d \in \mathcal{D}\}$ so that the algorithm preferably uses the IGP's shortest paths whenever possible. The main loop of the algorithm consists in calling iterate (Algorithm 2) in charge of the path-generation process until no new interesting path can be generated. By Theorem 1, the stopping criterion ensures that our solution is optimal for the global sr-path set $\overline{\mathcal{P}}(k)$. The iterate algorithm performs two main steps:

1. Optimizing SRTED-LP$(\mathcal{P}, \mathcal{D}, \lambda)$ with an LP solver and collecting the final dual values $(\mathbf{y}, \mathbf{z})_{\mathbf{x}}$ (a by-product of the simplex algorithm).

2. Considering each demand d and computing the minimum cost sr-path p from $s(d)$ to $t(d)$ as described above. Every minimum cost sr-path p such that $c(p) < (v - \mathbf{z}_d)/v$ is added to P' that eventually is added to the restricted path set $\mathcal{P}$.

To ensure that SRTED-LP is solved optimally, we can choose in step 2 to add any number of paths satisfying $c(p) < (v - \mathbf{z}_d)/v$. At one extreme, one could stop as soon as one such path is found. The path-finding process is then fast but with the drawback of possibly increasing substantially the number of iterations needed to converge. At the opposite extreme, we could generate for each demand every possible path satisfying the property (and not only the optimal one). Doing so would probably reduce the number of iterations but would increase the computation cost of each. We chose a trade-off by adding up to *maxp* paths generated at step 2. Those paths are generated by considering the demands by decreasing the value of $(v(d) - \mathbf{z}_d)/v(d)$ in order to maximize our chances of finding a path satisfying inequality (3.2).

In our implementation, we use a parallel execution for the last loop of Algorithm 2. One main worker appends to a bounded buffer the demands by decreasing value of $(v(d) - \mathbf{z}_d)/v(d)$. Consumer workers execute the mincost-srpath algorithm and add the path to P' if inequality (3.2) is satisfied. This provides a significant speedup because no communication between

Algorithm 1 colgen ($\mathcal{P}, \mathcal{D}, \lambda$, *maxp*)

while $P' \leftarrow$ iterate($\mathcal{P}, \mathcal{D}, \lambda$, *maxp*) $\neq \emptyset$ **do**
    $\mathcal{P} \leftarrow \mathcal{P} \cup P'$
return LP-solve(SRTED-LP($\mathcal{P}, \mathcal{D}, \lambda$))

Algorithm 2 iterate ($\mathcal{P}, \mathcal{D}, \lambda$, *maxp*)

$\mathbf{x}, (\mathbf{y}, \mathbf{z})_{\mathbf{x}}, vol \leftarrow$ LP-solve(SRTED-LP($\mathcal{P}, \mathcal{D}, \lambda$))
for $x, y \in V$ **do**
    $c_{sp}(x, y) \leftarrow \sum_{e \in E(\langle x, y \rangle)} r(e, x, y) \cdot \mathbf{y}_e$
for $e \in E$ **do**
    $c_{adj}(e) \leftarrow \mathbf{y}_e$
$P' \leftarrow \emptyset$
for $d \in \mathcal{D}$ **in decreasing order of** $(v(d) - \mathbf{z}_d)/v(d)$ **do**
    $p \leftarrow$ mincost-srpath($s(d), t(d), c_{sp}, c_{adj}$)
    if $c(p) < (v(d) - \delta_d)/v$ **then**
        $P' \leftarrow P' \cup \{p\}$
    if $|P'| \geq$ *maxp* **then**
        break
return P'

the workers is required during the mincost-srpath algorithm. We assume that the function LP-solve solves a linear program and returns a tuple containing the values for the primal variables, the values of the dual variables and the value of the objective function.

3.5.3 Minimizing the worst link utilization

Traditionally, TE aims at routing a whole set of demands while minimizing the worst link utilization. There are two reasons why Algorithm 1 cannot be used directly for solving this problem:

1. It will select a subset of demands to route under the constraint of a given maximum overload λ.

2. The decision variables may take fractional values in the optimal solution to SRTED-LP.

Algorithm 3 CG4SR addresses these two issues by making use of colgen (Algorithm 1) as a subroutine.

The first issue is addressed by performing a binary search for the parameter $\lambda \in [0, \lambda_M]$ (where λ_M is a large enough value) to discover the minimum value for λ such that the full set of demands is routed. At each step of the binary search, we check whether all the volume is routed. If so, we decrease λ. Otherwise, we increase it. From one step to the next, the initial path set

provided to Algorithm 1 is the one generated at the previous step so that Algorithm 1 only needs to call iterate (Algorithm 2) a few times in practice.

The second issue is addressed by heuristically solving an ILP problem denoted SRTE-UTIL-ILP to select for each demand one path $\mathcal{P}$ while minimizing the worst link utilization.

$$
\begin{aligned}
&\text{SRTE-UTIL-ILP} \\
&\min_{\lambda} && \lambda \\
&\text{s.t.} && \sum_{p \in \mathcal{P}} \mathbf{x}_{dp} && = 1 && \forall d \in \mathcal{D} \\
& && \sum_{p \in \mathcal{P}} \sum_{d \in \mathcal{D}(p)} r(e,p) \cdot v(d) \cdot \mathbf{x}_{dp} && \leq \lambda \cdot c(e) && \forall e \in E \\
& && \mathbf{x}_{dp} && \in \{0,1\} && \forall p \in \mathcal{P}, \\
& && && && \forall d \in \mathcal{D}(p)
\end{aligned}
$$

This model is very similar to the SRTED-ILP model. It changes the objective function to minimize the capacity factor λ and replaces the inequality in the first set of constraints with an equality to guarantee that each demand is satisfied. It is guaranteed to be feasible since by initialization of $\mathcal{P}$, it contains at least one path for each demand. Unfortunately it has no guarantee to solve optimally the worst link utilization problem. Indeed, $\mathcal{P}$ may not contain all the necessary paths to reach optimality. Forcing discrete decisions starting heuristically from the set of columns produced by a column generation process is a standard approach (see for instance [DL05]). We can also report the optimality $gap = \text{ILP-solve}(\text{SRTE-UTIL-ILP}(\mathcal{P}, \mathcal{D})) - lb$ providing a guarantee of how suboptimal the integer solution might be. As shown in the experimental section, the solutions computed by Algorithm 3 are very close to optimality in practice.

As before, we assume that ILP-solve is a function that solves an integer linear program.

Our framework, similarly to [Har+15b], is flexible enough to accommodate other constraints on the paths by only adapting the mincost-srpath algorithm. For example, by computing only paths of segment cost at most k, we implicitly limit the maximum segment cost of any path in the solution. Other types of constraints could for instance be a limit on the latency or forbidden/-mandatory intermediate nodes (for service chaining constraints [Zha+13]).

3.6 Evaluation

This section describes the results using the column generation approach described in Algorithm 3 CG4SR. All the experiments were conducted on avail-

Algorithm 3 GC4SR ($\mathcal{D}, \lambda_M, k, \epsilon, maxp$)

$\mathcal{P} \leftarrow \{\langle s,t\rangle \mid (s,t,d) \in \mathcal{D}\}$
_, _, *target-vol* $\leftarrow$ colgen($\mathcal{P}, \mathcal{D}, \lambda_M, maxp$)
$lb \leftarrow 0$
$ub \leftarrow \lambda_M$
while $|lb - ub| > \epsilon$ **do**
    $\lambda \leftarrow (lb + ub)/2$
    $\mathbf{x}, (\mathbf{y}, \mathbf{z})_{\mathbf{x}}$, *vol* $\leftarrow$ colgen($\mathcal{P}, \mathcal{D}, \lambda, maxp$)
    if *vol* $\geq$ *target-vol* **then**
        $ub \leftarrow \lambda$
    else
        $lb \leftarrow \lambda$
return ILP-solve(SRTE-UTIL-ILP($\mathcal{P}, \mathcal{D}$))

Table 3.1: Dataset summary.

ID	# nodes	# links	# demands
AS 1221	104	302	10712
AS 1239	315	1944	98910
AS 1755	87	322	7482
AS 3257	161	656	25760
AS 3967	79	294	6162
AS 6461	138	744	18906

able RocketFuel topologies [SMW02] for reproducibility. Their characteristics are described in Table 3.1. We use Repetita [GSV17] to run all the other solvers: DEFO [Har+15b; Har+15a], Bhatia [Bha+15] and SRLS [GHV17]. We reuse the demand matrices generated for Repetita [GSV17]. For each topology, they generated 5 demand matrices through the gravity model described in [Rou05]. Demands were normalized so that MCF can merely force all link loads to be below or equal to 90%. The number of nodes ranges from 80 to 315 and the number of demands from 7482 to 98910.

All our experiments are easily reproducible[2]. Our experiments were run on a computer with 32 CPUs at 2.60 GHz, 128 GB of RAM and Java 1.8 JVM. CG4SR does not actually need 128 GB of RAM, but it is able to take advantage of the 32 CPUs thanks to the multithreading of the pricing computation. We used the same version of Gurobi [LLC18], v8.0, for all the solvers.

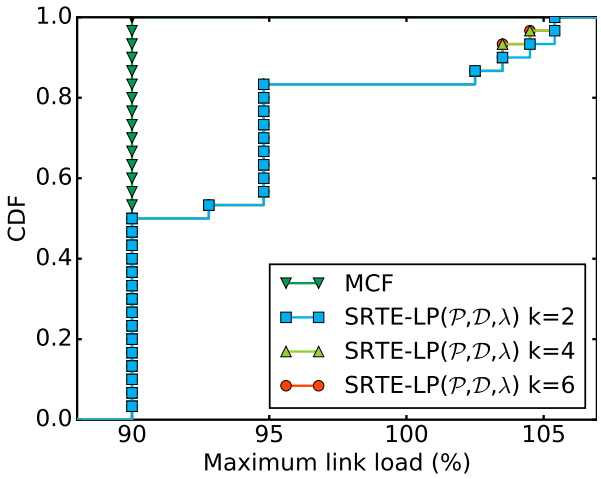

Figure 3.3: Lower bound of MCF and CG4SR without time limit.

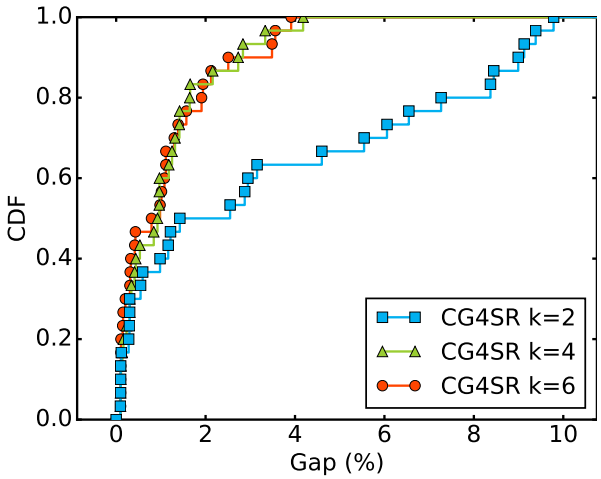

Figure 3.4: Gap of CG4SR without time limit.

3.6.1 Near optimum evaluation

CG4SR provides a better lower bound for TE over SR than MCF. Traditionally, the value of an optimal MCF solution is used as a lower bound for the minimum maximum link utilization that one can achieve for routing a traffic matrix. However, as mentioned above, this bound is unrealistic as MCF is oblivious to SR. Figure 3.3 studies the quality of the lower bound provided by CG4SR and SRTED-LP, compared to MCF. The load predicted by MCF is

[2]The code of our solver is integrated to Repetita [GSV17] and is available at `https://github.com/jadinm/Repetita`

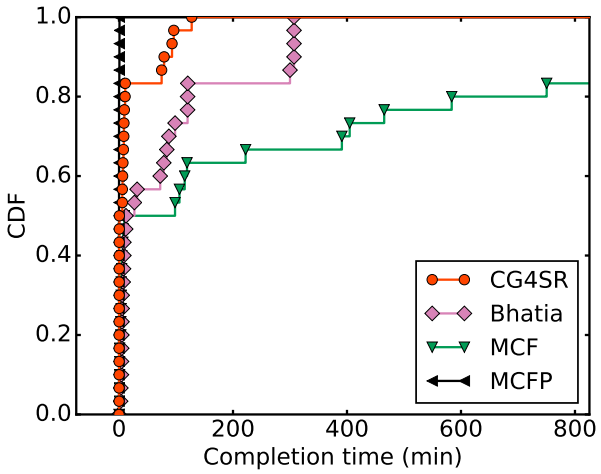

Figure 3.5: Execution time (CG4SR is limited to two segments of MCF, MCFP, Bhatia and CG4SR without time limit).

always of about 90% because the Repetita's demand matrices [GSV17] were generated artificially to be at this value. However, CG4SR shows that it is strictly impossible to escape network congestion for 5 of these demand matrices. Moreover, the difference between the CG4SR and MCF lower bounds can be as high as 15% in the predicted maximal load. Increasing the number of segments does not bring the CG4SR lower bound much closer to the MCF.

CG4SR provides solutions whose maximum load is at most 4% more than the optimal solution. We ran CG4SR without enabling adjacency segments and without time limit. The experiment was repeated with limits of 2, 4 and 6 segments to observe the impact of the segment limit on the quality of the solution. Figure 3.4 shows CDFs of the gap (in percentage) between the CG4SR upper and lower bounds on all the 30 instances (i.e., 5 demand matrices for each of the 6 topologies) and increasing the limit on the number of segments. This gap is the maximum distance to the actual optimum. We can see that increasing the limit from 2 to 4 segments impacts the quality of the solution while increasing the limit from 4 to 6 has little impact. Paths with 4 or 6 segments add more flexibility to SRTE-UTIL-ILP than paths with 2 segments. We can see that this gap is most of the time below 1% of the load and at worst 4% of the load if 4 segments are allowed.

CG4SR is more efficient than Bhatia and MCF. We compared the execution time of CG4SR to Bhatia, MCF and MCFP. MCFP is an efficient variant of MCF (described in [GSV17]) that is only able to compute the optimal objective value of MCF, not the actual paths comprising the solution. Figure 3.5 describes how fast the different solvers can find their best solution. During these runs, the limit on the number of generated paths at each column gener-

ation iteration, *maxp* in Algorithm 2, is fixed to 10. This figure shows a CDF of the execution time on the different topologies and demand matrices for CG4SR, Bhatia and MCF. Because Bhatia only allows two segments, CG4SR is also limited to two segments in this figure. MCF is the slowest one, and it runs out of memory for all the demand matrices on the largest topology despite the 120-GB memory. Bhatia only considers paths that can be expressed with two segments. This significantly reduces the problem size and Bhatia can always find an answer. CG4SR can run with any number of segments because of the lazy generation of the paths. And this is so effective than we are actually faster than Bhatia with a two-segment limit.

Figure 3.5 also shows that the MCFP variant of MCF can actually compute the optimal value of the MCF quicker than CG4SR. But, as mentioned above, MCFP only provides the maximum link utilization of the MCF formulation but not a set of paths satisfying it. Hence, in practice, MCFP can only be used to provide lower bounds which, as we showed in Figure 3.3, are worse than the ones provided by our algorithm.

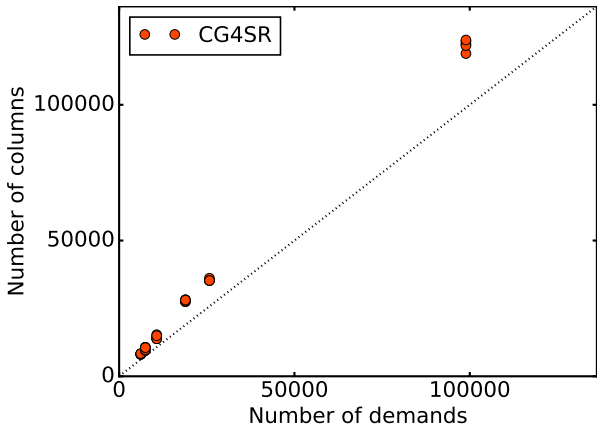

Figure 3.6: Number of generated columns over the size of the demand matrices.

CG4SR scales better than MCF and Bhatia. Our model has fewer variables than the MCF and Bhatia formulations. The size of our model is the number of paths that were generated. The size of MCF considers how much of each demand can be placed on each edge. Therefore, the number of variables is the number of edges multiplied by the number of demands. Bhatia considers for each demand, one detour (i.e., two segments) through a single node of the graph. Its number of variables is thus the number of nodes multiplied by the number of demands. We observe that CG4SR is more scalable because it considers at worst 60 times fewer possibilities than Bhatia and 200 times fewer than MCF. This explains why CG4SR is faster than Bhatia and

MCF. This difference does not change significantly when varying the limit on the number of segments. As can be seen in Figure 3.6, the number of generated columns seems to grow linearly with the number of demands. Given that the restricted path set is initialized with all the direct paths for every demand, the path-finding process (Algorithm 2) only creates a limited number of additional paths to reach optimality. This also explains why the column generation approach is so efficient in practice, as it only needs to solve the linear program with a number of variables only slightly above the number of demands.

3.6.2 Any-time behavior

The previous section shows that we can produce quality solutions with unlimited time budget. This section evaluates the evolution of the quality of the CG4SR solutions over time. We compare CG4SR to the heuristic approaches DEFO, SRLS and also to Bhatia.

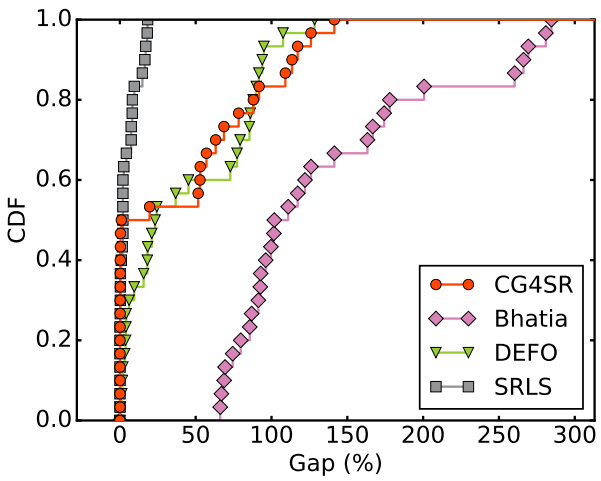

Figure 3.7: Timeout at 1 minute: gaps between SRTED-LP$(\mathcal{P}, \mathcal{D}, \lambda)$ and Bhatia, DEFO, SRLS or CG4SR. All solvers are limited to 6 segments.

CG4SR finds a good solution even if only allowed to run for a short amount of time. Figures 3.7, 3.8 and 3.9 show, for each of the cited solvers, a CDF of the gap to the SRTED-LP solution for the cited solvers after, respectively 1 minute, 5 minutes and 10 minutes. During these runs, the *maxp* parameter (see Algorithm 2) of CG4SR is fixed to 10. The limit of segments is set to 6, except for Bhatia which limits itself to 2. The quality of a solution is the difference between this current solution and the CG4SR lower bound that was computed without time limit and with the same limit on the number of segments.

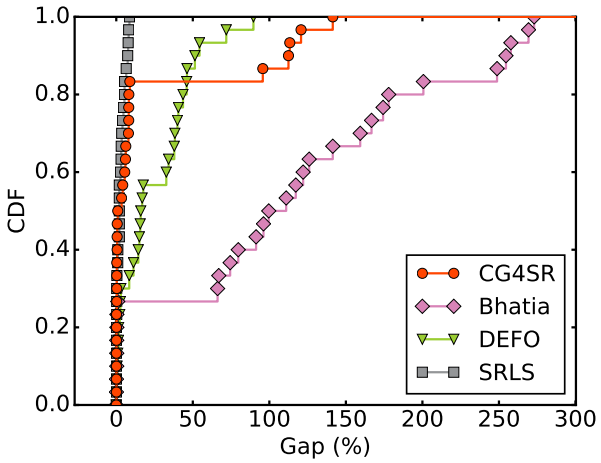

Figure 3.8: Timeout at 5 minutes: gaps between SRTED-LP$(\mathcal{P}, \mathcal{D}, \lambda)$ and Bhatia, DEFO, SRLS or CG4SR. All solvers are limited to 6 segments.

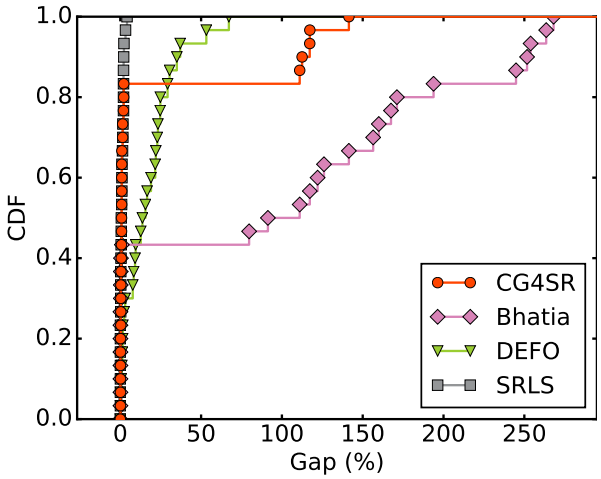

Figure 3.9: Timeout at 10 minutes: gaps between SRTED-LP$(\mathcal{P}, \mathcal{D}, \lambda)$ and Bhatia, DEFO, SRLS or CG4SR. All solvers are limited to 6 segments.

We see that we are always faster than Bhatia even with limited time spans. SRLS and DEFO are heuristic approaches and therefore are able to quickly find good solutions. Figure 3.7 shows that CG4SR is already comparable to SRLS and better than DEFO for half of the instances after 1 minute. We see that DEFO initially finds better solutions but CG4SR catches up for most instances by increasing the timeout in Figure 3.8 and Figure 3.9. The largest instance is not yet solved after 10 minutes and that explains why DEFO is still better.

CG4SR is more robust than SRLS over different sets of demands. SRLS produces good results, but this solution is based on local search and

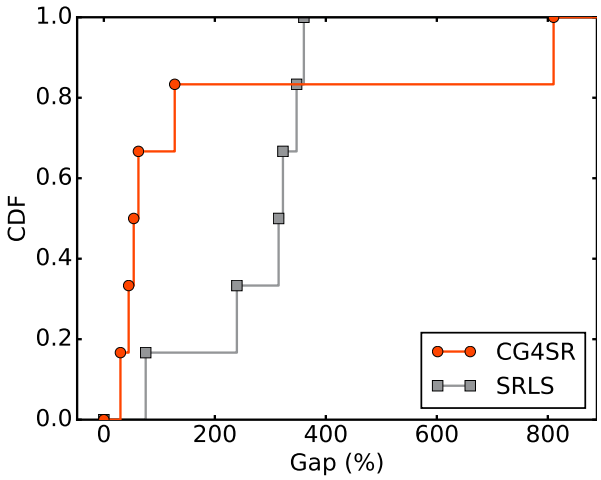

Figure 3.10: Gaps between SRTED-LP and SRLS or CG4SR after 10 min with a limit of 6 segments.

can be stuck in a local optimum. We did not observe it on the demand matrices generated by the gravity model. Indeed, the demand volumes are generally much lower than link capacities. This also means that there are many possible ways to reach good solutions, even if the best solution is hard to find. The gravity model is a good match to the traffic engineering problem in ISPs but demand volumes are likely higher in inter-datacenter communication [Jai+13]. This also means that there are fewer good solutions. We generated one additional demand matrix for each RocketFuel topology with a low number of large demands requiring 95% of the bandwidth available between their source and destination. Figure 3.10 shows a CDF of the quality of the solution with a time limit of 10 minutes and a limit of six segments as in Figure 3.9. These results confirm that SRLS can be worse than CG4SR when fewer good solutions are available.

3.6.3 Adjacency segment benefits

Adjacency segments are important in TE and CG4SR is the first to use them. CG4SR is the first SR traffic engineering model to allow adjacency segments. We evaluate the benefits of adjacency segments on the inter-datacenter network topology of OVH in Europe (described in [Aub+16]). This topology has more parallel links than RocketFuel topologies and thus, the OVH topology can really benefit from adjacency segments. We do not have access to the IGP weights and capacities of OVH. Therefore, for each link bundle we set the capacity of half of the links to 10Gbps and the other half to 2.5Gbps. This simulates link upgrades on the network. For pairs of nodes

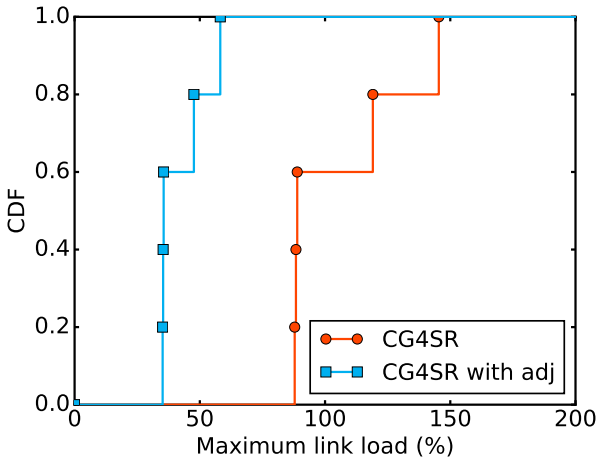

Figure 3.11: The CG4SR solutions with or without adjacency segments. The limit of segments is fixed to 4.

with a single link between them, we set the capacity to be ten times bigger. We fix all IGP weights to 1. Five demand matrices were generated for the OVH topology with the gravity model [Rou05].

Figure 3.11 shows CDFs of the gap (in percentage) between the CG4SR upper and lower bounds over the demand matrices of the OVH topology. We do not limit the execution time, and we limit the number of segments to 4. This means that we allow at most one detour through a specific link because one segment is needed for the destination and an adjacency segment has a cost of 2. Even allowing only one link detour halves the load of the maximally loaded link because it utilizes better the parallel links of this topology.

3.7 Conclusion

The goal of this thesis is to propose and explore the use cases of a deployable architecture allowing endhosts to control their paths. In this chapter, we present CG4SR, a scalable approach that leverages column generation to efficiently solve the traffic engineering problem in networks using Segment Routing that can be used in the SRN controller presented in Chapter 2 to choose relevant paths to send to the applications. CG4SR enables network operators to exploit both node and adjacency segments with full control over the maximum number of segments while existing techniques only support node segments. Our experiments demonstrate that CG4SR is faster than previously proposed ILP approaches. Compared to heuristics, CG4SR provides a tight bound that provides the near optimality of its solution in the tested topologies while heuristics are fast but can achieve worst results if there are

not many good available solutions.

Towards a minimal TCP extended through eBPF

4

TCP is a well established protocol whose core was specified in [Pos81b]. Nevertheless, more than 42 RFC update this protocol since its first RFC [Pos81b]. TCP is implemented in the kernel of every modern OS. However, protocol development is slow, even more in the Linux kernel because kernel updates take time to reach some distributions. For instance, Selective Acknowledgments, defined in [Flo+00], took a decade to be deployed according to [Fuk11]. Another example is the implementation of MPTCP [For+13] whose RFC was published in 2013. It took 7 years to be merged into the Linux kernel.

Through kernel updates, applications were able to configure more and more the TCP behavior to their needs, through the socket API or through system-wide configuration (i.e., sysctl in Linux). However, this did not address the main issue: the kernel development is a slow process and the aggregation of all the TCP extensions makes it even harder to include more changes.

To allow more modularity, Brakmo [Bra17] introduced eBPF in the Linux kernel to dynamically program the TCP stack. For instance, users can inject custom congestion controls or use custom TCP headers [TB20] without modifying the kernel. This addition of eBPF was the first step towards a truly modular implementation of TCP and many steps were taken afterwards. The overhead induced by eBPF extensions is low, even when modifying every packet in eBPF [TB20]: the throughput decreases by less than 1% but the CPU overhead is higher, up to 10% of CPU usage at worst.

The goal of this chapter is to design strategies to allow the network and the endhosts to collaborate and use this collaboration to reap path diversity benefits while using single-path protocols. In Chapter 2, we propose Software Resolved Networks. The main challenge to the deployment this architecture is its deployment is the need for explicit support from every enhanced application. The ability of eBPF to dynamically inject code into the TCP stack without application support is a promising lead to overcome this issue.

In this chapter, we survey the progress of the Linux stack towards a truly extensible TCP, where most of its optional features would be injected from user space through eBPF. This would accelerate the development of future extensions of TCP such as those proposed in this thesis. We start by defining

what a minimal and modular TCP stack, that we call xTCP, would look like. In Section 4.1, we present the invariants of xTCP and its minimal data structures. Then, in Section 4.2, we present locations in the packet workflows that can be dynamically programmed. In Section 4.3, we survey the current modularity that eBPF brings to the Linux kernel as of version 5.14. We compare it to xTCP. In Section 4.4, we propose three extensions to the eBPF API, namely custom timers, the parsing of incoming ICMPv6 messages and the parsing of IPv6 extension headers in packets. We demonstrate their correctness by developing eBPF programs for concrete use cases.

4.1 A minimal TCP

Minimal TCP stacks such as uIP [Dun06] and lwIP [Dun01] also implement a minimal TCP/IP stack for IoT low-end equipment. However, these implementations target performance under low RAM or slow CPU constraints. This section presents invariants for a minimal TCP stack that is meant to be extended in the next section to achieve a modular TCP stack.

A minimal TCP remains a reliable bytestream transfer of data between two endpoints defined by their IP addresses and source ports. It also explicitly establishes and tears down connections, unlike UDP. To do so, it uses the state machine described in [Pos81b]. In the rest of the section, we present the required elements to match these invariants.

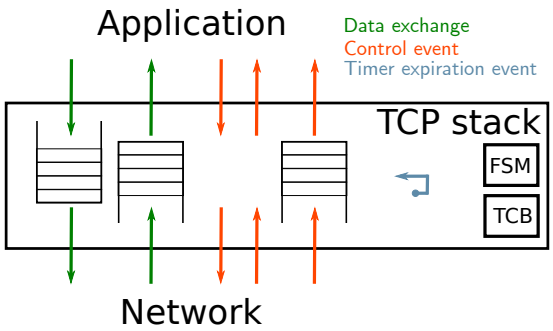

Figure 4.1: Minimal TCP building blocks and communication.

Since data needs to be transmitted reliably, we need one buffer storing the sending bytestream to be able to retransmit it. Another buffer stores received data that is waiting for being read by the application. These buffers interact with the application and the network by transmitting data from one to the other (see Figure 4.1).

Along with the stream of data, TCP endpoints exchange control events. These signals negotiate the start or the end of a connection as well as acknowledging the data received in sequence. They also advertise the size of

the TCP window that they can accept in their respective receiving buffers.

The network can communicate with the network layer to receive ICMP or ICMPv6 messages, e.g., Destination Unreachable Message, Packet Too Big Message, Parameter Problem Message. A minimal implementation of TCP has to parse these signals to guarantee a reliable transport protocol. Some of these messages can be read by the application. We require an error buffer to queue these errors waiting to be read by the application.

The application can interact with the stack by sending control messages as well: either through a socket API, an interface like netlink or event sysctls for system-wide options.

The TCP stack can call itself back after a given time through a Timer expiration event. This last signal type is used to trigger retransmissions when packets are not acknowledged fast enough.

The establishment or the closing of a TCP connection follows a Finite State Machine (FSM) whose current state and transitions dictate the different behaviors of the connection. All the TCP states and transitions defined in [Pos81b] are required in a minimal implementation.

Besides the buffers and the FSM, the TCP stack maintains a TCP Control Block (or TCB). We need the four-tuple identifying the connection: the source and destination IP addresses as well as the source and destination ports. To handle the sending bytestream, we need to keep the sequence number of the next byte to send from this bytestream as well as the first byte in flight that is not acknowledged in order to know which byte range to retransmit. If the buffer is empty, the next byte to send is also the next byte to queue. For the receiving side, the TCB contains the next expected byte to distinguish out-of-sequence packets It also contains the next byte to be read by the application if the bytestream is not released immediately. Extensions like a congestion control algorithm need to keep variables in the connections state, instead of an extension-specific state. These variables have to be inserted in the TCB. Upon using the insertion API call, an extension can allow the variable to be modified by other extensions. By default, every variable is read-only. The default variables are always read-only for the extensions. Each variable has a name defined by the extension. In case of conflict, the variable insertion fails. When unloading the extension, inserted variables are removed, but they can be removed only by the extension that inserted it.

4.2 A modular TCP

In this section, we first cover the necessary code injection spots and then discuss the API calls that are required to obtain a full-fledged TCP implementation as available in Linux or BSD implementations and to allow future extensions.

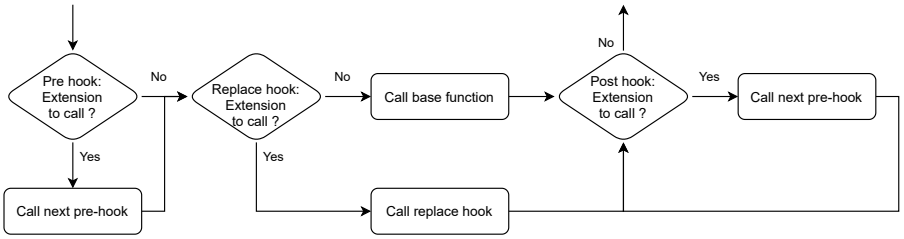

Figure 4.2: Workflow of an injection spot.

We call our minimal TCP xTCP and we plug it in extensions at runtime. These extensions implement the optional features found in a complete TCP implementation, for instance Explicit Congestion Notifications (ECN) [RFB01]. To support them, xTCP should allow code injections at specific injections spots in its workflow. For each injection spot, we can use three types of injections or hooks, as defined in [De +19]. An extension can hook itself to extend or replace a given default behavior. Figure 4.2 describes the workflow of a generic injection spot. Extensions hooking at the `pre hook` are called before executing the default behavior (e.g., a monitoring extension). If an extension returns a value, it is passed to the next extension hooked. The default behavior can be replaced by hooking to the `replace hook` (e.g., to change the congestion control algorithm). There can be only one `replace hook`. Afterwards, extensions hooking to `post hook` are called in order of injection. As in the `pre hook`, if an extension returns a value, it is passed to the next extension hooked (e.g., a monitoring extension). For each `pre hook` and `post hook`, the execution order is determined by a priority number set by the extension when hooking. This allows extensions to have different priorities for different hooks. xTCP should be able to call them in order, potentially through the use of priority queues.

4.2.1 Output workflow

Figure 4.3 shows xTCP's output workflow. The blue steps are injection spots skipped by the minimal default implementation, but they are necessary to implement some extensions and attain the features of a complete TCP implementation. The yellow steps are injection spots required even without extensions.

The sending of data can be triggered by the application sending its data, by a timer expiration (e.g., the retransmission timer) or from an extension (e.g., Nagle algorithm [Nag84]). The first step of sending data is to serialize the TCP header based on information of the trigger request (i.e., retransmissions require a specific byte range), the TCB and the FSM. This includes the

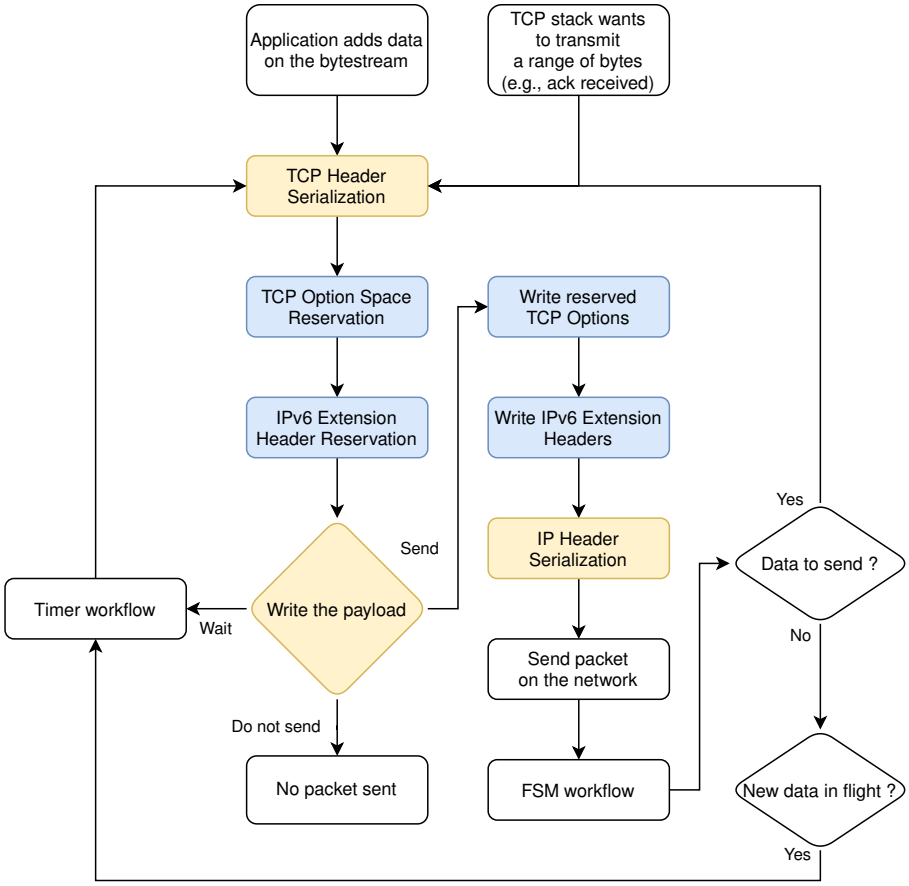

Figure 4.3: Output workflow.

sequence number and the TCP flags. Some extensions, like Explicit Congestion Notifications (ECN) or Windows Scaling, need a post hook to edit the TCP header. In ECN, the ECE and the CWR flags need to be set

By default, xTCP does not add any TCP option. However, it lets hooked extensions write the options. This is done in two injection spots. The first one allows to reserve space to write its option and the second one actually writes the option. Separating these two steps is needed because options, such as TCP Authentication Option [TMB10] (TCP-AO), need to know the payload before being used. Each extension in the first injection spot must announce the maximum amount of bytes it wishes to write as well as a callback function to write the option. Based on this information, xTCP will call the callback that required option space. Since the TCP option space is very limited, the extensions that set a high priority on their hook are served first.

The same reasoning applies to IPv4 Options or IPv6 Extension Headers.

As of this writing, only the Routing Header makes sense to change at the transport-layer. This allows eBPF to modify the Segment Routing Header [Fil+20]. The other headers can be transparently handled at the network layer. Depending on the API of the network layer, this writing can be either directly in the packet or indirect through a change to the IP Control Block.

When writing the payload, by default, xTCP writes as many bytes as available while keeping the constraint of the receiving window and of the MSS. If this is a retransmission, it sends the appropriate byte range, otherwise it uses the unsent part of the bytestream. This behavior can be replaced by an extension that would integrate a congestion control algorithm [APB09], Nagle [Nag84] or delayed acknowledgments [APB09]. In this case, each extension, hooking itself in a pre, post or replace hook, has to send a byte range or cancel writing. xTCP calls every hooked extension, giving the decision of the previous extension to the next one. The last extension has the final say and its decision is applied. The decision can be to wait for a timer expiration, to write a given range of bytes (e.g., zero for a pure ACK), or cancel the sending.

Because of some extensions like ECN, we need to change the IP header as well because of TCP-layer choices and therefore, we add an injection spot after the IP header serialization to set flags like ECN Capable in ECN.

Finally, we can send the packet on the interface. Some packets might change the Finite State Machine, e.g., a packet carrying the RST flag. This is checked in the FSM workflow that is detailed in a further section. If there is still data to send, we restart the process since the beginning. Otherwise, if xTCP sends new data, we start the retransmission timer that will eventually trigger the retransmission if no data was acknowledged.

4.2.2 Input workflow

Figure 4.4 shows the input workflow of xTCP. The green steps are hooks that are specific to a given type of option, header or type. They are only called if this element is present in the received packet. The yellow steps are required even in xTCP but need to accept extension pluginization.

As input stream, the TCP stack can receive either data or errors. Errors are encoded as ICMP or ICMPc6 messages. The network layer identifies the TCP connection that triggered the error by looking at the ICMP payload that contains the beginning of the faulty packet. By default, xTCP can parse and process the four messages of ICMP and ICMPv6 defined in [Pos81a; CDG06]. This behavior can be replaced, and new hooks can be defined for any unknown type of ICMP message. Some ICMP messages can be passed to the applications by placing them in an error buffer.

The network layer identifies the TCP connection and deserialize the IP

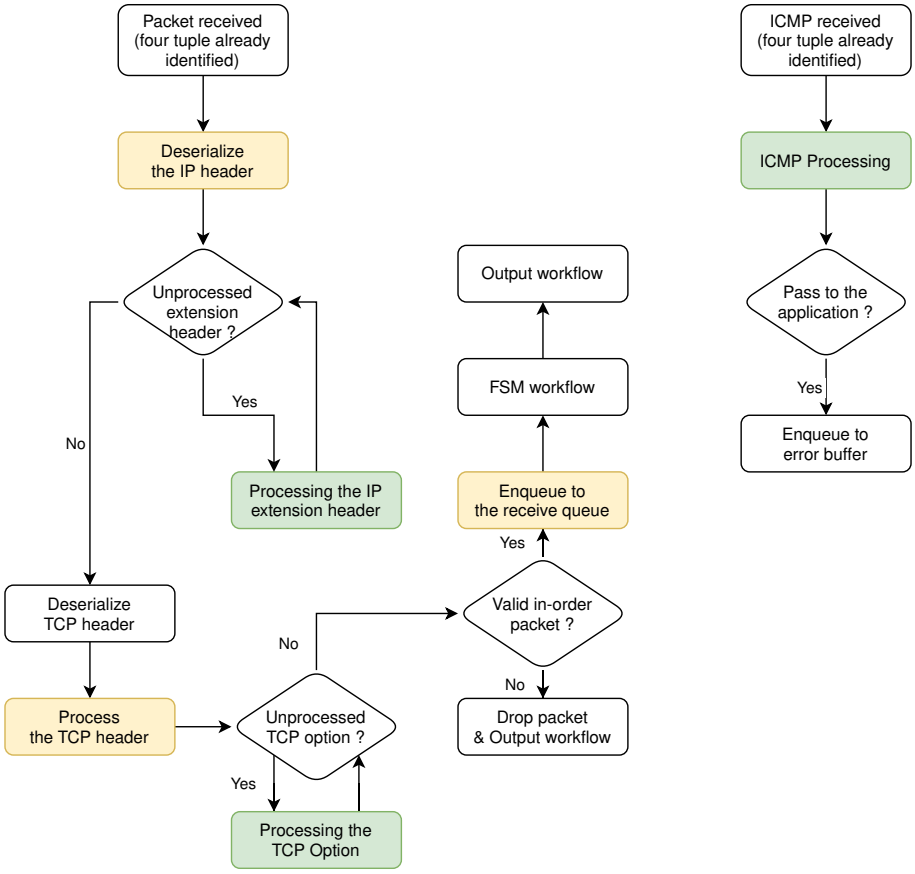

Figure 4.4: Input workflow.

header. For each extension header type, a hook can be created on demand. By default, they are ignored.

Then, the TCP header is read and deserialized. Each option of in-order packets is processed in a separate hook for each option. Each extension can register specifically to a given set of options. By default, xTCP only knows about NOOP and EOL options because these are the only options that don't include a length info in their second byte. The other options are skipped by default.

The next step is to check the sequence numbers. If the packet is in sequence, it can be processed. Otherwise, the packet is dropped and xTCP sends an acknowledgment. To allow out-of-order processing of packets, an extension plugs itself in a post hook of this step.

The byte range is then extracted and enqueued to the bytestream of received data. Some extensions can monitor its state of the receiving queue by plugging in a post hook there.

4.2.3 Finite State Machine workflow

TCP relies on an FSM to control the connection establishment and closing. In addition to that, xTCP allows extensions to hook to it. This workflow is triggered by both the input and output workflows, as well as extensions or the application through API calls. We define one insertion spot for each state. The default behavior of xTCP for each hook is fixed by [Pos81b]. The extension uses a `post hook` only to interpret this new signal and can leave the old signals to the default behavior.

Moreover, when xTCP changes the state, it calls the hooks at the extension spots for the transition between the original state and the new one. Extensions can hook themselves to the connection establishment or closing, to, respectively, allocate or free data structures.

4.2.4 Timer support

Besides the parameters and return values of each hook detailed before, extension need more capabilities that we give through an API.

Extensions such as delayed acknowledgments [APB09], Nagle [Nag84] need to start a custom timer to trigger the output workflow. To do that, we require three API functions. The first API call needs to start a timer for a given duration and call a given callback function upon termination. It returns a unique ID for the timer that ca be used in a second API call to cancel the timer if the timer is no more required. This timer is local to the extension and cannot be cancelled by another extension. In the callback function, an extension can call the third API call to trigger the start of the output workflow.

4.3 Linux TCP stack modularity

In this section, we survey the progress of the Linux kernel version 5.14 towards a modular TCP stack for each workflow presented in Section 4.2. It goes way beyond a minimal TCP implementation for historical reasons. Developers have little incentive to remove working features. However, the modularity is being improved since the addition of eBPF.

4.3.1 TCP header and its options

Looking back at Figure 4.3, we know that all the blue steps are not skipped by default since the TCP stack adds a lot of options by default (or through the setting of eBPF).

Additional TCP options can be defined since version 5.10. They proceed in two steps as in xTCP. First, they add a post hook to reserve option space to know the available space for data. And only then, they use another a post

hook to write TCP header in the option space. Note that in that hook, extensions can modify already written options. This is not needed in xTCP because it does not write any option but the Linux implementation does. Linux does not allow eBPF programs to read the packet payload in this hook. Therefore, implementing a TCP-AO [TMB10] option in eBPF is impossible We can however deduce if the packet is a pure ACK segment or not from the TCP data length field.

The entire TCP header is also available in this post hook and the extensions can read and modify the packet TCP header there.

On the input side (see Figure 4.4), an eBPF program can read any known or unknown TCP option. They do not propose one program by option: each program will have access to the whole option space as well as the TCP header.

4.3.2 TCP congestion control

The Linux TCP stack allows a control of the payload writing different from the other hooks presented so far. It allows eBPF programs to redefine congestion control functions since kernel 5.6 [Cor20]. This allows to implement custom congestion controls but not different strategies for Nagle or delayed acknowledgments. They are already implemented inside the kernel, but their behavior cannot be tuned using eBPF.

4.3.3 Finite State Machine

Since kernel 4.16, eBPF programs can monitor TCP state changes. They do not have a specific hook for each transition, but eBPF programs can infer the transition by remembering the previous state.

4.3.4 Missing features

The kernel does not allow reading or writing the IP layer (its header or extensions) from an eBPF program attached to the socket. They also do not provide a way to parse known or unknown ICMP messages from an eBPF program attached to the socket. While it does have a signal for retransmission timeouts, the kernel does not allow eBPF programs to define their own timers.

4.4 Extensions to the Linux TCP stack

In this section, we present three additions to the eBPF API and illustrate them through their use cases. First, we showcase an in-band traceroute based on our implementation of the Timer Workflow of eBPF (see Figure 4.3) and the parsing of incoming ICMPv6 messages (see Figure 4.4). Then, we implement

IPv6 SRH reversal in eBPF by adding the parsing of IPv6 extension headers. The SRH reversal is demonstrated through two use cases: a connection failover and a load balancer avoidance. (see Figure 4.4).

4.4.1 In-band traceroute

In some cases, failures concern a small portion of the connections. They are difficult to debug because in the global view of the network operator, everything is fine. Finding out which nodes are blocking a portion of connection cannot be identified from a regular traceroute because it won't use the same five-tuple. Therefore, ECMP might route it elsewhere. Performing a traceroute inside the connection is the best way to find the faulty nodes.

Listing 1 Pseudocode of one ACK probe for a traceroute.

```
void probe_traceroute(struct bpf_sock_ops *skops,
                      u32 hop_limit) {
  curr_hop_limit = get_hop_limit(skops);
  set_hop_limit(skops, hop_limit);
  send_ack(skops);
  set_hop_limit(skops, cur_hop_limit);
  start_timer(skops, 1000, // duration in ms
        BASE_OP + conn->n_hops);
}
```

Listing 1 presents the pseudocode of the routing that we use to perform a single probe of a traceroute. The `probe_traceroute` routine sends an ACK whose hop limit is parametrized. After sending the probe, a timer is started. After 1 second, it calls back the program with an opcode that depends on the hop number. If we don't receive the ICMP message before this opcode is triggered, we know that either hop failed to respond or we reached the final destination.

Figure 2 shows the main routing of the eBPF program. First, we retrieve the state of the connection, stored in an eBPF map before checking whether we were called by the timer expiration of an already identified hop. Then, we trigger the first step of the traceroute when the connection is established. But it could be triggered after a given number of retransmissions to debug a faulty connection. When we receive an ICMPv6 Time Exceeded from an intermediate router, we register the source address of the ICMPv6 and send a probe for the next hop. In the end, we construct the ordered list of addresses.

These probes can alternatively be implemented in eBPF hooks inside the NIC as shown by ELF [SD21].

Listing 2 Pseudocode of the eBPF program to perform a traceroute.

```
#define BASE_OP 50 // start opcode for timers

// probe_traceroute routine

int handle_sockop(struct bpf_sock_ops *skops) {
  conn = bpf_map_lookup_elem(&conn_map, &five_tuple);

  if (skops->op == BASE_OP + conn->n_hops - 1)
    // The probe was dropped or reached destination
    register_end(skops, conn->n_hops);

  switch (skops->op) {
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
    case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:
      conn->n_hops = 1;
      probe_traceroute(skops, conn->n_hops);
      conn->n_hops++;
      break;
    case BPF_SOCK_OPS_PARSE_ICMP_CB:
      // An ICMP is received
      ip6 = skops->skb_data;
      icmp = (struct icmp6hdr *) (ip6 + 1);
      if (icmp->icmp6_type == ICMPV6_TIME_EXCEEDED) {
        register_hop(skops, conn->n_hops - 1,
                     ip6->saddr);
        probe_traceroute(skops, flow_info->n_hops);
        conn->n_hops++;
      }
  }
  return 0;
}
```

4.4.2 IPv6 Segment Routing Header Reversal

We showcase our hook to parse incoming IPv6 extension headers with an SRH reversal program implemented in eBPF. Such a program reverses the last SRH received to route its own packets.

Listings 3 presents the pseudocode of the eBPF program that we use to reverse an SRH. An eBPF program controls the IPv6 Extension Headers that trigger it. We set this option upon the end of the three-way handshake on

Listing 3 Pseudocode of the eBPF program to reverse an SRH.

```
int handle_sockop(struct bpf_sock_ops *skops) {
  conn = bpf_map_lookup_elem(&conn_map, &five_tuple);

  switch (skops->op) {
    case BPF_SOCK_OPS_TCP_CONNECT_CB:
      // Before sending a SYN segment on client-side
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
      // End of three-way handshake on server-side
      val = 1;
      // Trigger the eBPF program on received SRHs
      rv = bpf_setsockopt(skops, SOL_IPV6,
                          IPV6_RECVRTHDR,
                          &val, sizeof(int));
      break;
    case BPF_SOCK_OPS_PARSE_EXT_HDR_CB:
      // An SRH was received in a packet in sequence
      reverse_srh(skops->skb_data, skops->skb_data_end,
                  &reversed_srh);
      move_to_path(skops, &reversed_srh);
      break;
  }
  return 0;
}
```

server-side and at the connection start on client-side. Indeed, the Linux kernel uses a lightweight structure to represent a non-established TCP connection on server-side, called a request socket. This structure is more lightweight to prevent Denial of Service attacks using SYN flooding [Edd07]. However, it does not support setting a SRH on it. It is replaced by a regular socket at the end of the TCP three-way handshake.

Then, upon receiving a packet with an extension header, the IP header and its extension headers are stored between `skb_data` and `skb_data_end`. The kernel verifier checks that the program do not access memory outside the range delimited by these two pointers. Then we can reverse the SRH and set the SRH of the connection. Note that we change the SRH at each packet of the connection, but we could use a custom timer to periodically activate the parsing of the SRH.

One of the use cases of SRH reversal is a TCP connection failover when a client loses one of its two access points. Figure 4.5 illustrates it. Each link has a

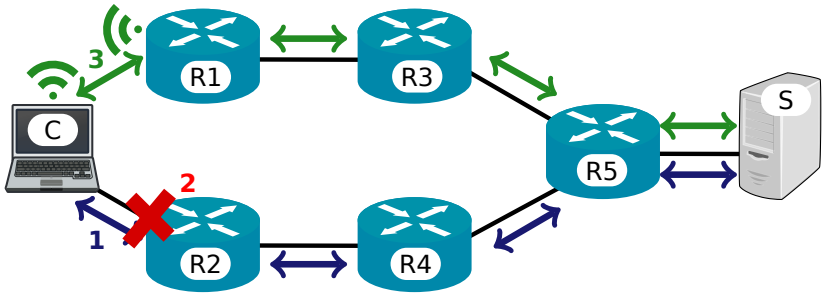

Figure 4.5: Topology to test TCP connection failover.

bandwidth of 100 Mbps and a delay of 1 ms. At step 1, Client C communicates with Server S through Ethernet in TCP. The connection is thus established with the Ethernet address of Client C. At step 2, the Ethernet access points of the client fails. Using a failure detection eBPF program (see Chapter 5), the client detects the failure and switches to the other path. Normally, the connection would fail because Server S cannot join the Ethernet address of Client C through the Wi-Fi access point. But, at step 3, we make Client C use the following segment list: <C_{Wi-Fi}, R1, S>. With the eBPF program in Listing 3, Server S uses the segment list <R1, C_{Wi-Fi}, $C_{Ethernet}$> which allows the return packets to join the Ethernet address of Client C through the Wi-Fi access point. The connection remains active even in plain TCP.

To demonstrate the use case feasibility we emulate the topology of Figure 4.5 in Mininet [Han+12]. Client C sends traffic to Server S through an IPerf connection capped at 10 Mbps. The Wi-Fi access point is emulated a regular cable connection. After 25 seconds, we make the current used path fail.

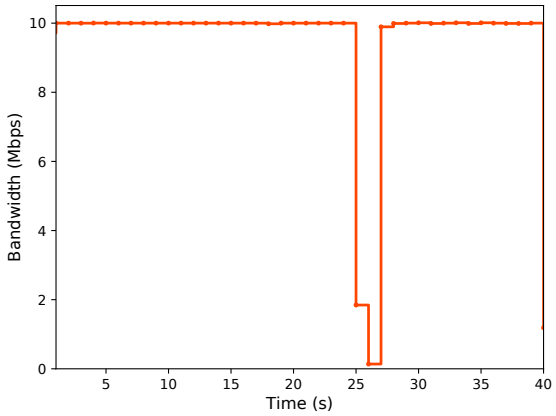

Figure 4.6: TCP connection failover maintains connectivity.

Figure 4.6 shows the bandwidth evolution through time. After 25 seconds,

the connection is interrupted but recovers even though only the reversal of the SRH is made on server-side.

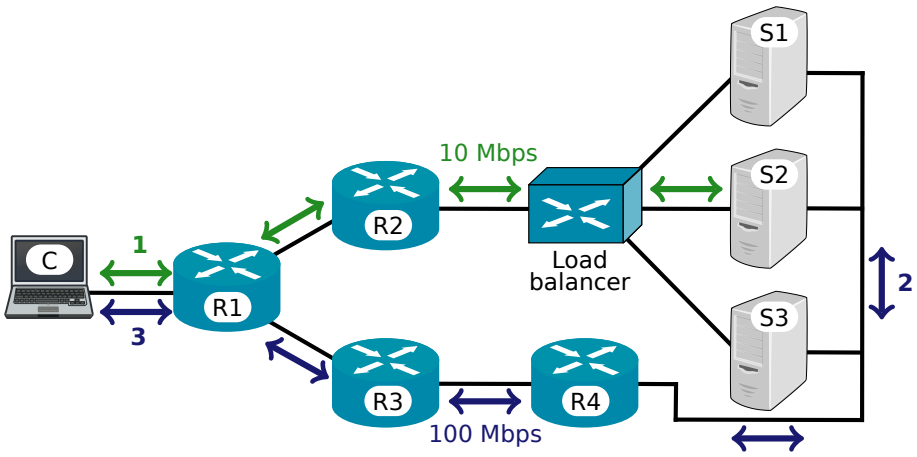

Figure 4.7: Topology to test load balancer bypass.

The client can also benefit from the SRH reversal as illustrated by Figure 4.7. In this example, a client reaches one of the servers that propose a service. The selection of the server is made through a load balancer. For performance reasons, we want to move the traffic away from the load balancer so that it can balance even more traffic and traffic does not depend on the survival of its load balancer. In this case, the client uses the reversal SRH program while the server has an eBPF program that switches to the path bypassing the load balancer after the connection establishment.

At step 1, Client C starts a connection through an anycast address identifying any of the servers. The load balancer chooses Server S2 as Client C's destination. At step 2, after that the session is established, Server S2 uses the segment list <$S2_{bypass}$, C>. At step 3, Client C reverses it to <$S2_{bypass}$, $S2_{loadbalancer}$>. In order to illustrate the path taken by the connection data, we use a different link delays of the links: 10 Mbps for the path with the load balancer or 100 Mbps otherwise.

To demonstrate the use case feasibility we emulate the topology of Figure 4.7 in Mininet [Han+12]. Client C sends traffic to Server S through an IPerf connection capped at 100 Mbps. Figure 4.6 shows the bandwidth evolution through time. We observe that the bypass works since the bandwidth is not capped at 10 Mbps.

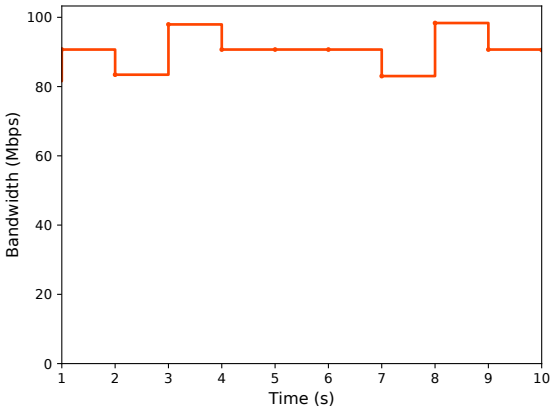

Figure 4.8: The TCP connection bypasses the load balancer.

4.5 Conclusion

In this chapter, we surveyed the progress of the Linux stack towards a truly extensible TCP, where most of its optional features would be injected from user space through eBPF. This would accelerate the development of future extensions of TCP such as those proposed in this thesis. We started by defining what a minimal and modular TCP stack, that we call xTCP, would look like. We implement three extensions to the eBPF API, namely custom timers, the parsing of incoming IPv6 extension headers in packets and the parsing of incoming ICMPv6 packets. These extensions to the kernel is about 100 lines of C code. We demonstrate their correctness by developing eBPF programs for concrete use cases. The eBPF programs are implemented in about 200 lines of C code.

Leveraging eBPF to make TCP path-aware

5

This chapter extends the Software Resolved Network solution, presented in Chapter 2, by using eBPF, whose capabilities were presented in Chapter 4. We empower the trusted (i.e., controlled by the IT) endhosts even more in the path selection process while allowing applications to stay agnostic to SRN and to benefit from our solution. We use the flow information of TCP to make an enlightened path selection. This chapter presents use cases for datacenter networks, but the architecture improvements are applicable to enterprise networks as well.

The Transmission Control Protocol (TCP) is one of the key Internet protocols. It is used by a broad range of applications. TCP was designed when there was typically a single path between a client and a server. Today's networks provide higher path diversity [AFT07], yet TCP still only uses the single path selected by the network layer. This limits the ability of TCP to react to events such as interdomain failures or highly congested peering links.

We propose the TCP Path Changer (TPC), a set of eBPF programs that are incorporated in the Linux TCP/IP stack to make it more agile. To illustrate the benefits of our approach, we first demonstrate that TPC can quickly reroute an ongoing TCP connection around a failure. We then show that TPC can also monitor the round-trip-time of active TCP connections and automatically reroute them if it becomes too high. Our evaluation of TPC in emulated networks evidences the significant performance benefits of a path-aware transport protocol.

5.1 Introduction

The Transmission Control Protocol (TCP) was designed in the 1970s. Yet, despite its age, TCP remains the dominant transport protocol in today's Internet, controlling more than 90% of the overall network traffic [Tre+20]. The protocol and its implementations have evolved during the last decades. While today's TCP implementations still use the original wire format [Pos81b], they include various improvements, including congestion control techniques [Jac88; Al-+19], support for larger windows [JBB92], selective acknowledgments [Mat+96], and more.

TCP reflects the layering principle and considers the underlying network layer as a black box that exposes IP addresses to the transport layer in which TCP operates. A TCP connection is always bound to the IP addresses of the client and the server. TCP is agnostic to the network path that the packets traverse between the communicating hosts and, in particular, neither selects nor influences this path in any way. As TCP performance can be adversely affected by packet reordering, most networks avoid spreading packets belonging to the same flow across different paths [Hop00] (even though recent advances such as RACK make TCP less sensitive to reordering [Che+21]). Two main types of routing events induce path changes that can impact TCP connections: (*i*) link or node failures [Mar+08; WJL03] and (*ii*) changes due to in-network traffic engineering [Red+20]. TCP reacts to path changes induced by such events by adjusting its round-trip time estimation, retransmitting lost packets, and possibly adjusting its congestion window.

When TCP was designed, networks supported one path for a given source-destination pair [Pax97] and so TCP's role was to adapt its transmission rate to the traffic conditions on this path. Today's networks support a higher number of paths that can potentially be used by a TCP connection between two hosts [AFT10]. In particular, many networks provide multiple equal cost paths towards internal destinations, and routers load-balance packets from different TCP connections across these paths [Hop00]. Measurement studies show that although equal cost paths are supposed, in principle, to be comparable, performance-wise, this is not always the case [Pel+13; Red+20]. The specific path traversed by the packets of a TCP connection in such networks mainly depends on the client's selected source port (since the IP addresses and server port are fixed). Researchers have proposed taking advantage of this to change the client's TCP port [Det+13; Kab+14a] to influence a TCP connection's path, but this solution has not been widely adopted. A similar approach was proposed for the context of Multipath TCP [For+13] in datacenters [Rai+11].

We leverage Segment Routing and the expressiveness afforded by eBPF programming [Bra17] to demonstrate how TCP can become path-aware, i.e., a transport protocol that has the capability of reacting to network-level events not only by modulating the transmission rate but also by selecting an alternate path. eBPF is a virtual machine included in the Linux kernel[1] that can be used to tune the Linux TCP/IP stack. We take advantage of this expressiveness to devise the TCP Path Changer (TPC) and implement it as a set of eBPF programs. Using eBPF provides extensive flexibility for an application to *customize* TPC to its specific requirements. For instance, some applications might require very low network latency while others might be more sensitive

[1] Although we focus on Linux in this chapter, we note that there are ongoing efforts to also port eBPF into Microsoft Windows [TG21] and FreeBSD [Hay18].

to route instability.

To demonstrate the usefulness and flexibility of the TPC, we evaluate it on two different use-cases: (1) recovery from distant link failures, and (2) dynamic selection of lowest-delay paths. Our results for the first use-case show that TPC can quickly react to link failures in a distant network, enabling TCP connections to continue operating without being severely affected. This should be contrasted with the time needed to converge to a new global routing configuration, which may require several seconds, if not much longer [Hol+19]. Our second use-case is motivated by highly interactive applications such as web services or request-response applications, whose performance is heavily impacted by network delays. We propose a TPC that monitors the round-trip-time of TCP connections and reroutes them to a shorter-delay path. This TPC employs an online learning algorithm to make good decisions despite uncertainty regarding its environment, and provides significant improvements in performance over path-oblivious transport.

This chapter is organized as follows. Section 5.2 presents the motivation for our work and some necessary background. In Section 5.3 we propose the TCP Path Changer (TPC) and demonstrate how it enables the Linux TCP/IP stack to automatically detect distant link failures and react by rerouting the affected connections. Section 5.4 demonstrates how our proposed TPC monitors the round-trip-time of connections and reacts by rerouting the flows that suffer from excessive delays. We position TPC with respect to related work in Section 5.5 and conclude in Section 5.6.

5.2 Motivation

Today's hyperscale datacenters, such as Amazon's AWS, Google Cloud and Microsoft's Azure, host hundreds of thousands of servers that are interconnected via a private globe-spanning network that is also connected to numerous Internet Service Providers. In addition to these hyperscalers, there are a myriad of smaller datacenter networks that host up to tens of thousands of servers. These smaller datacenters are frequently used by smaller companies to host internal servers and also provide public services such as websites, file and backup servers, game servers, etc. Little public information is available about these datacenters and their networks.

Interestingly, OVH, a French hosting company, provides publicly available information about the topology of its datacenters and backbone infrastructure[2]. At the time of writing this thesis, OVH maintains 28 datacenters that are located in 19 countries and host more than 300,000 servers. For redundancy purposes, most enterprises deploy servers in multiple geographic

[2]See `http://weathermap.ovh.net` and `http://peering.ovh.net`.

locations to cope with datacenter failures [Sar]. Content providers also often need to maintain multiple servers at different locations to offer low latency services.

Examining the topology of OVH's backbone and the locations of its datacenters Figure 5.1 reveals the number of *distinct* peering links between OVH and any external peer network. We filter parallel links and regard an external peer as having x *distinct* connections to OVH when that peer is directly connected to x different OVH routers. Figure 5.1 shows that roughly 40 % of the peers in OVH's backbone in Europe have two or more distinct connections with OVH. Note that the number of peering links is indicative of the importance of a peer, in terms of both traffic volume and business considerations.

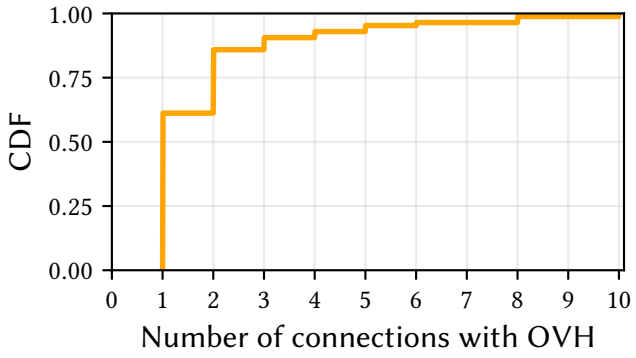

Figure 5.1: Distribution of the number of *distinct* connections between OVH and external peers in the Europe backbone.

Another interesting observation regarding the OVH network relates to the distance between a given datacenter and these multi-connected peers. We measured the difference in the number of hops between each datacenter i and external peer j, restricting our attention to external peers with at least two *distinct* peering links. We computed the shortest path from datacenter i to all *distinct* connections of peer j and quantified the length difference between the different paths. Figure 5.2 plots the CDF of these results when aggregating over all i's and j's. In almost 60 % of the samples, the *shortest* shortest-path and the *second shortest* shortest-path have the same number of hops. This suggests that there may be many multiple available paths with comparable performance interconnecting a datacenter and an external peer.

5.2.1 Path-aware datacenter servers

While datacenter networks provide multiple paths for reaching most Internet destinations, in existing deployments, most servers are oblivious to this

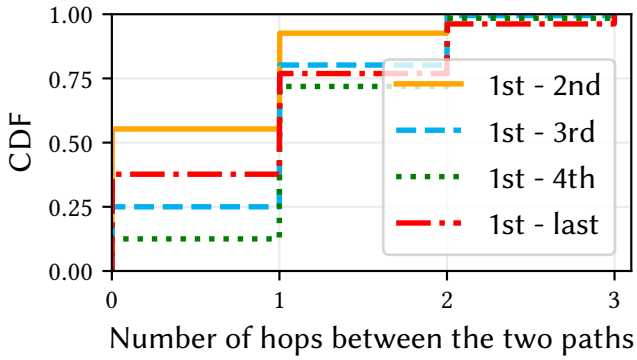

Figure 5.2: For each external peer - OVH datacenter pair, we compute all shortest paths from the datacenter to a *distinct* connection with the external peer. The figure presents the difference in the number of hops between all these paths for each pair in the Europe backbone. The paths are sorted according to the number of hops.

diversity and treat the underlying network as a black box. Specifically, each server uses the route selected for it by the first-hop router, entrusting the network with the responsibility for choosing the best path with respect to any destination.

We propose a new service that datacenter providers can offer to customers that may need higher performance (e.g., with respect to specific destinations) than the one provided by the default paths, as well as faster recovery from link failures. In our scheme, instead of blindly using the routes selected by first-hop routers, these servers learn additional routes themselves and actively select among them. To accomplish this, we install a BGP daemon on each of these servers and connect it to one or two BGP route reflectors. We configure these route reflectors to only send routes to these servers and never accept routes from these servers. To distribute alternate paths, we leverage the BGP Add-Path extension [Wal+16], which enables a BGP router to advertise multiple routes towards the same prefix over iBGP sessions. We point out that it is possible to configure BGP filters to only advertise multiple paths for the most important prefixes. We extend the SRN resolvers (see Chapter 2) to serve as route reflectors but any router reflector supporting the BGP Add-Path extension would do.

Figure 5.3 and Figure 5.4 illustrate this utilization of BGP Add-Path in a simple datacenter network connected to two different BGP peers. The server S uses BGP Add-Path on its iBGP session with a local route reflector. This route reflector learns the two routes from AS2 and AS3 through the R2 and R3 BGP next-hops. The server learns the two paths from the route reflector.

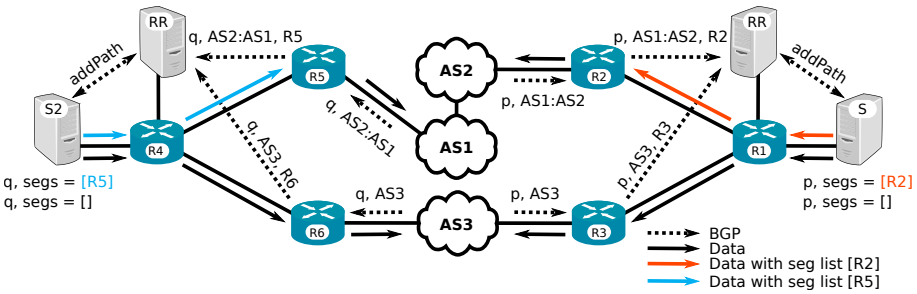

Figure 5.3: These servers learn several paths to each important prefix using BGP Add-path and use them to forward packets with the IPv6 Segment Routing Header (SRH).

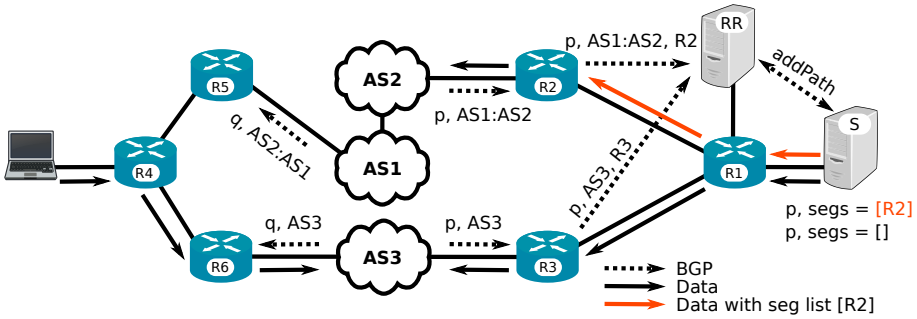

Figure 5.4: These servers learn several paths to each important prefix using BGP Add-path and use them to forward packets with the IPv6 Segment Routing Header (SRH), while clients only use one path.

If the server sends a regular IP packet, it will follow the default route, e.g., via R3. If the server prefers, for any reason, the path via R2, it simply needs to add an SRv6 header to forward those packets via R2. R2 removes the Segment Routing Header (SRH) and forwards it to its destination, through AS2. This is illustrated via the plain arrows in the figures.

To offer fast recovery from failures, the alternative routes are injected to both sides of the connection since a failure can affect both sides, e.g., in the context of inter-datacenter communication, as depicted in Figure 5.3. Importantly, however, to offer a best-path discovery service, we only need to control one end of the connection. We thus consider the asymmetric scenario depicted in Figure 5.4. As shown in the figure, the client does not enjoy path diversity while the server does.

5.2.2 eBPF makes the TCP/IP stack programmable

Historically, TCP implementations are structured as monolithic code that can be configured at the granularity of individual connections using system-wide

parameters [McK+96; FS11] and socket options. The Linux TCP/IP stack, which is dominant on servers, can also be tuned by leveraging eBPF [Bra17]. eBPF is a virtual machine included in the Linux kernel that supports a limited RISC-like assembly language. System administrators can attach eBPF programs using different hooks inside the kernel. A static verifier, packaged with the kernel, checks memory calls and program termination of the code before injection to guarantee that the code does not make the kernel crash. Different types of eBPF programs have been developed [Gre19]; some collect statistics about the utilization of the kernel data structures, while others monitor the operation of system calls, etc. Brakmo [Bra17] extended the Linux TCP stack to enable it to execute eBPF programs when specific events occur. Researchers have used this framework to support new TCP options [TB20]. Here, we implement eBPF programs that enable the Linux TCP stack to change the path of an established TCP connection in response to network failures or when the path is not providing the expected delay. We realize such path changes using the IPv6 Segment Routing implementation within the Linux kernel [LB17]. This scheme could also be realized via multi-topology routing [Pse+07] in IPv4 networks.

5.2.3 The TCP Path Changer (TPC)

By putting together the different ingredients described above, we demonstrate how TCP can become path-aware. We propose the TCP Path Changer (TPC) and implement it as a set of eBPF programs in the Linux TCP/IP stack. In the next two sections, we present two flavors of the TPC, each targeting a specific use-case. In Section 5.3 we show how the TPC enables the automatic detection of distant link failures and rerouting of affected connections. In Section 5.4, we present an online-learning-based TPC that monitors connections' round-trip-times and automatically reroutes flows that suffer from excessive delays.

5.3 Recovering from distant failures

Datacenter networks must preserve connectivity with peering networks in the presence of various types of link and router failures, which have been shown by measurement studies to be frequent in ISP [Mar+08] and datacenter networks [GJN11]. To address this, network operators have deployed various fast reroute techniques [CRS16], typically enabling transition to new routes within less than a few tens of milliseconds for wide area links. When the failure affects a peering link [Wan+07] or a distant network, affected prefixes might be unreachable for several seconds or more. To improve responsiveness to failures, researchers have proposed techniques such as Blink [Hol+19],

where routers monitor the TCP flows and automatically reroute the flows when their destination suddenly appears unresponsive.

In this section, we explore a simpler approach for coping with these distant failures: when such a failure occurs, the servers that exchange data with the affected destinations quickly detect that these became unreachable and leverage knowledge of alternate BGP next-hops to reroute the affected TCP connections.

5.3.1 Triggering rerouting upon path failure

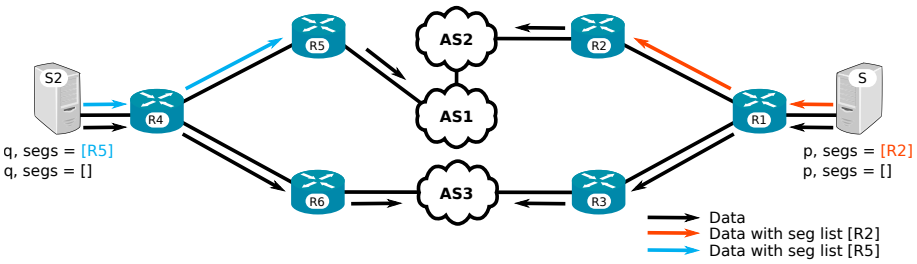

Figure 5.5: Example network for rerouting upon path failure.

Consider the network shown in Figure 5.5. The server sends packets towards a destination that belongs to prefix p learned from both AS2 and AS3. How can the server detect a transient failure on this path? A first approach would be to wait until the reception of an ICMP destination unreachable message from an intermediate router. This is unlikely to succeed since routers strictly limit the rate of transmission of ICMP messages. Instead, we leverage the fact that our server will stop receiving acknowledgments for the data sent towards this prefix. Its retransmission timer will expire, and the unacknowledged data will be retransmitted. We use these retransmissions as triggers for executing the eBPF code that selects an alternate path. We present two strategies for detecting path failures: NRTOChanger(N,M) and TimeoutChanger(T,M).

NRTOChanger(N,M) selects a path during the handshake (i.e., when sending the SYN or the SYN+ACK). Then, NRTOChanger(N,M) monitors the retransmission timer and the reception of duplicated data for each TCP connection, and selects an alternate path after N successive expirations or M retransmissions from the other end.

Failures can be asymmetric, that is, a failure can affect one direction of the communication and not the other. If the failure occurs on the path from the sender to the receiver (see Figure 5.6), NRTOChanger(N,M) waits for N retransmission timeouts (RTO) before selecting another path. Assuming that only the current path fails, and that TCP uses the standard exponential backoff

[Pax+11], which doubles the RTO each time, it takes the sender $\sum_{i=0}^{N-1} 2^i \cdot RTO_{sender}$ to switch to a new path.

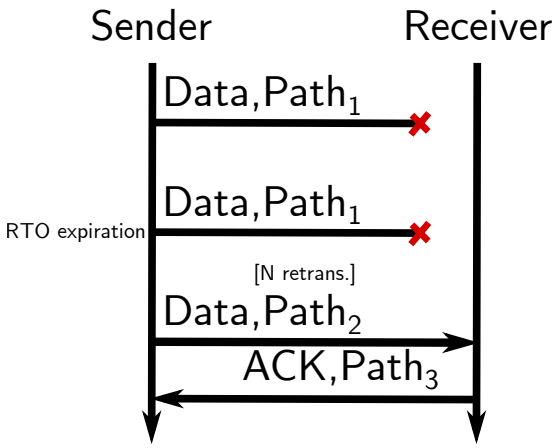

Figure 5.6: Detecting failures on the forward path.

The failure could also impact the return path. Figure 5.7 shows its detection. NRTOChanger(N,M) also monitors the sending of duplicated acknowledgments. These duplicated acknowledgments are sent upon receiving duplicated data. Since the return path fails, the duplicated acknowledgments are lost. The sender thus resend its data which will trigger other duplicated acknowledgments. NRTOChanger(N,M) waits for M duplicate acknowledgments sent and therefore the number of retransmissions received from the sender (see Figure 5.7) before selecting another path. Under the same assumptions as before, this will involve a time duration of $\sum_{i=0}^{M} 2^i \cdot RTO_{sender}$ from the reception of the data and the sending of an acknowledgment that will reach the sender.

Coordination between both entities would be useful from a performance point of view. Custom TCP options defined in eBPF [TB19] could carry these data. Indeed, we want M, the number of duplicated acknowledgments before changing the receiver's path, to be lower than N, the number of retransmission timer expirations, before changing the sender's path. However, the NRTOChanger(N,M) on both endhosts will eventually converge to a working path if they have access to at least one path that reaches the other end. The receiver's path only changes when the sender is sending on the right path. So, eventually, the receiver will use the right path. The sender might loop over its paths several times to make the receiver switch to a correct path.

TimeoutChanger(T,M) works differently. When encountering retransmissions, it selects an alternate path when some data remains unacknowledged for more than a predetermined time interval of length T. This time value could be selected by the application developer, e.g., 200 ms for interac-

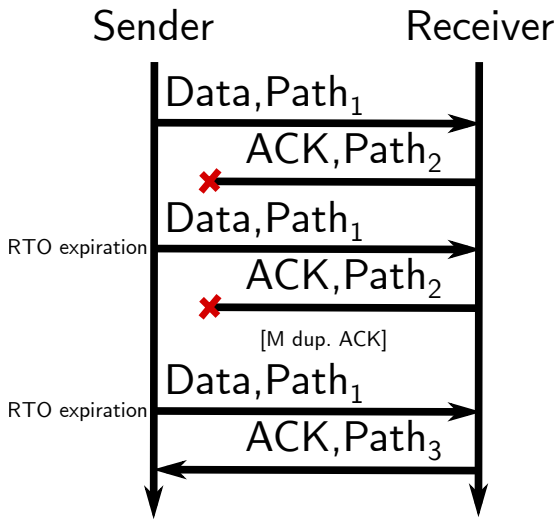

Figure 5.7: Detecting failures on the return path.

tive applications. TimeoutChanger(T,M) also changes the path if TPC has to send M duplicated acknowledgments.

5.3.2 Implementation

We implement TPC as a set of eBPF programs attached to the socket operations because such programs can access the state of the TCP socket. Our eBPF programs are triggered upon a series of events like the establishment of a connection, a retransmission,... This is more efficient from a performance viewpoint than attaching eBPF to every packet transmission [TB20].

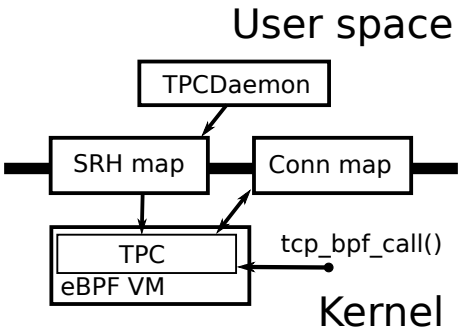

Figure 5.8: Path management code talks to the Path Daemon through eBPF maps.

Figure 5.8 shows the interactions between the different components inside the endhost. The TPC code runs in an isolated eBPF VM within the kernel. In

the TCP stack, the injected code receives a structure containing the opcode of the hook that triggered it, as well as the five-tuple and some additional variables of the TCP connection, such as the minimum RTT or the congestion window. Moreover, it can read and write to memory chunks, called eBPF maps, which can be accessed by a user-space application. In this architecture, the TPC daemon fills a first eBPF map, the SRH map storing the IPv6 Segment Routing Headers (SRH) towards the different BGP next-hops. Since eBPF injected code does not have global variables or heap, we use a second eBPF map, also organized as a hashmap, the connection map. The key is the five tuple, and it maps to a structure containing data about the connection.

Listing 4 TPC: initialization.

```
int handle_sockop(struct bpf_sock_ops *skops) {
  struct connection new_conn;
  struct conn *conn = bpf_map_lookup_elem(&conn_map,
                      &five_tuple);
  if (!conn) { // New connection
    new_conn = new_connection_state();
    new_srh = get_srh(0);
    move_to_path(skops, new_srh);
    bpf_map_update_elem(&conn_map, &five_tuple,
                        &new_conn);
    conn = &new_conn;
  }

  switch (skops->op) {
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
      // End of three-way handshake
      new_srh = get_srh(0);
      move_path(&dest_map, flow_id.remote_addr,
                flow_info->srh_id, skops);
      bpf_map_update_elem(&conn_map, &five_tuple,
                          conn);
      break;
    case BPF_SOCK_OPS_STATE_CB:
      // End of a connection
      if (skops->args[1] == BPF_TCP_CLOSE)
        bpf_map_delete_elem(conn_map, &five_tuple);
      break;
    // [...]
  }
  return 0;
}
```

Listing 4 shows the pseudocode for the path management initialization. Our TPC initializes its connection structure and sets the path upon actively starting the connection on the client-side. This way, the SYN follows the chosen path. However, we cannot do the same for the SYN+ACK. This is because the Linux kernel uses a lightweight structure to represent a non-established TCP connection, called a request socket. This structure is more lightweight than regular sockets to prevent SYN flooding attacks [Edd07]. In

particular, it does not support the socket options that are required to force a specific path for the SYN+ACK. The kernel copies the contents of the request socket in a regular socket at the end of the TCP three-way handshake (BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB hook). To clean up the state of the connection, we use the BPF_SOCK_OPS_STATE_CB hook.

Listing 5 TPC: reacting to failure.

```
int handle_sockop(struct bpf_sock_ops *skops) {
  // [Initialisation presented before]

  int new_id = (conn->srh_id + 1) % nbr_srhs;
  switch (skops->op) {
    // [Other cases presented before]

    case BPF_SOCK_OPS_DUPACK:
      // Duplicated Acknowledgement is going to be sent
      if (conn->last_rcv_nxt != skops->rcv_nxt) {
        flow_info->last_rcv_nxt = skops->rcv_nxt;
        conn->remote_retransmission_count = 1;
        return 0;
      }
      conn->remote_retransmission_count += 1;
      if (flow_info->remote_retransmission_count < M)
        return 0;
      struct srh new_srh = get_srh(new_id);
      move_to_path(skops, new_srh);
      break;
    case BPF_SOCK_OPS_RTO_CB:
      // Retransmission timer expiration
      if (conn->last_snd_una != skops->snd_una) {
        flow_info->last_snd_una = skops->snd_una;
        conn->local_retransmission_count = 1;
        return 0;
      }
      conn->local_retransmission_count += 1;
      if (flow_info->local_retransmission_count < N)
        return 0; // Not enough retransmissions

      struct srh *new_srh = get_srh(new_id);
      move_to_path(skops, new_srh);

      break;
  }
  return 0;
}
```

As shown in Listing 5, during the transfer, the eBPF program is triggered by expirations of the local retransmission timer (BFP_SOCKS_OPS_RTO_CB) or the transmission of duplicate acknowledgments (BPF_SOCKS_OPS_DUPACK), the consequence of a retransmission from the other endhost.

TPC allows detecting complete failures or extreme congestion. It maintains two counters: `local_retransmission_count` and `remote_retrans-`

`mission_count`. This counter is reset when the data transmission advances. If the local counter reaches N or the remote one reaches M, TPC calls `get_srh` to find a new path. Note that the value of the N and M threshold can be specified when the eBPF program is loaded.

Listing 6 Path migration.

```
void move_to_path(struct bpf_sock_ops *skops,
                  struct srh *new_srh) {

  // Actual move
  bpf_setsockopt(skops, SOL_IPV6, IPV6_RTHDR, new_srh,
                 sizeof(*new_srh));

  // Reset TCP metrics
  bpf_setsockopt(skops, SOL_TCP, TCP_PATH_CHANGED,
                 &val, sizeof(val));
}
```

TPC changes the path of the TCP connection by calling two new socket options with `bpf_setsockopt` as shown in Listing 6. The first one sets the SRH on the path. As TCP does not have any information about the level of congestion on the new path, the second socket option resets the retransmission timer. Indeed, the retransmission timeout is doubled at each retransmission [Pax+11] to prevent the connection from worsening an already congested path. When TPC moves from one path to another disjoint of the first one, it is pointless to use high retransmission timeout values.

5.3.3 Evaluation

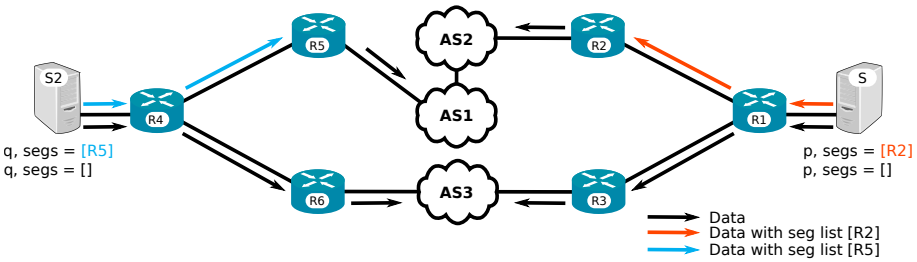

Figure 5.9: Evaluation topology.

We experiment on networks emulated using IPMininet [Jad+20] on a Linux kernel 5.3 running on a server equipped with 20 CPUs and 16 GB of RAM. To emulate links, we use `tc-netem` to set their delays and `tc-htb` to set their bandwidth. The other parameters of these tools are the default ones. For our evaluation, we use the topology shown in Figure 5.9. This network consists

of 100 Mbps paths with 6 ms RTT. In the figure, the bottom interdomain path corresponds to the preferred next-hop and the top to the alternate one.

The endhost S initiates a given number of connections throttled at 10 Mbps using `iperf3` towards the endhost S2. The connections send traffic across the emulated network for 30 seconds. We fix N (i.e., successive timer expirations) to 3 and M (i.e., the maximum number of retransmissions from the other end) to 2. After 10 seconds, we emulate a failure of the primary path by introducing 100 % packet loss with `tc-netem` so that no ICMP messages are generated. Then, we observe the time needed to send traffic to the working path for each side of the connection. We repeat this experiment over 100 runs and provide a cumulative distribution function (CDF) of the recovery latencies.

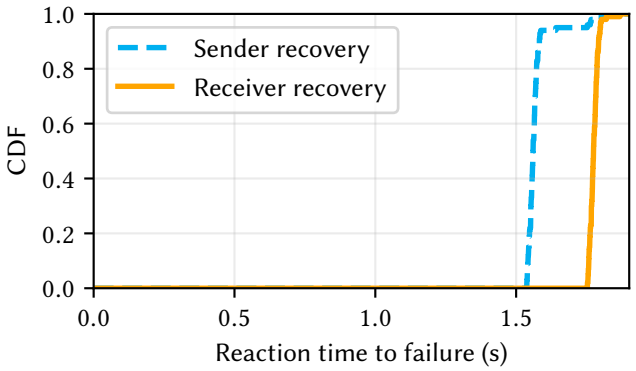

Figure 5.10: CDF of the recovery time with one connection with NRTOChanger(N=3,M=2).

NRTOChanger(N=3,M=2) can quickly reroute TCP connections after failures. Figure 5.10 shows a CDF of the recovery times of the sender and the receiver. In the median experiment, the sender recovers first after 1600 ms. This is the time duration required by the sender to experience three successive retransmission timeouts.

The Linux kernel computes the RTO slightly differently from what is advised by the IETF [Pax+11]. The IETF advises using the following formula:

$$RTO = max(H, SRTT + max(G, K * RTT_{VAR}))$$

with H set to 1 second, K set to 4, G being the clock granularity, $SRTT$ being the smoothed RTT and RTT_{VAR} being an estimation of the RTT variance. However, the Linux kernel sets H to 0, K to 1, and G to 200 ms. The RTT being 6 milliseconds, having only one connection in the network, the base RTO will

always be close to 206 ms. This base RTO is doubled at each consecutive timeout. For this reason, the sender using TPC waits for 3 timeouts before changing its path. Therefore, it waits at least $206+2\cdot206+4\cdot206 = 7\cdot206 = 1442$ ms.

The difference between our lower bound, 1442 ms, and the experienced one, 1560 ms, can be explained by three factors. First, the smoothed RTT can be slightly inaccurate because of delayed acknowledgments, which increase the RTT by a few milliseconds. Second, the rate at which the retransmission timer is checked is at a maximum resolution of 4 ms. The difference of 116 ms (for the median) is the time required to resend the window. The retransmission timer is started when the last packet is sent. This last source of delay is the main reason behind the variation that we observe in the CDF.

The receiver takes an additional 225 ms (for the median) to change its path. Indeed, we configured TPC to change its path after 2 retransmissions of the same acknowledgment. The receiver sent the initial acknowledgment during the failure. The first retransmission is triggered by the retransmission of the data on the working path. The receiver will wait a last retransmission from the sender, at least 206 ms later. Because of the time needed to send the window, we observe an increase of around 225 ms.

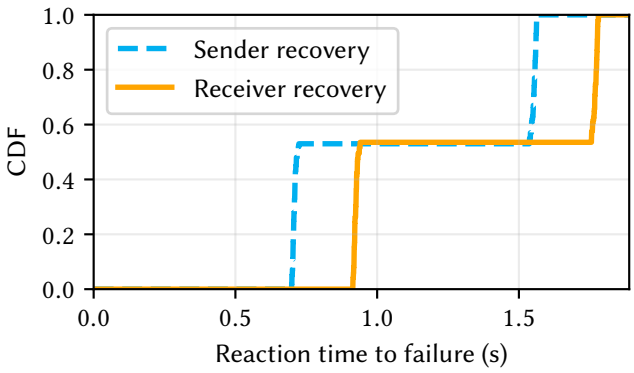

Figure 5.11: CDF of the recovery time with one connection with TimeoutChanger(T=700 ms,M=2).

TimeoutChanger(T,M) also detects failures. Instead of relying on a given number of retransmissions, TPC can redirect traffic if the same bytes are in flight for more than a given time that we arbitrarily fix at 700 milliseconds. Figure 5.11 shows the result with the same setup as before except that we use TimeoutChanger(T=700 ms,M=2) instead of NRTOChanger(N=3,M=2). TPC on the sender will change its path if the bytes are unacknowledged for more than 700 ms.

TimeoutChanger(T,M) measures this delay from the last call to a hook

called at each RTT estimation. It is called upon receiving an acknowledgment for the window, at least once per RTT. The number of times depends on the delayed acknowledgment implementation on the receiver. If there was no byte in flight during the failure, this time will be measured from the first retransmission timeout. 700 milliseconds is slightly above the time needed for two retransmissions, i.e., $206 + 2 \cdot 206 = 618$ ms. The change is triggered after 2 retransmissions (for more than half of the connections) or after 3 retransmissions at most. This depends on the state of the congestion window when the failure occurs. If the congestion window is almost full, the retransmission timeout occurs quickly and TimeoutChanger(T=700 ms,M=2) requires 2 retransmissions, otherwise, it needs 3 of them.

The receiver also waits for two retransmissions of the same acknowledgment before changing its path: hence the same delay difference of around 225 ms between the sender and receiver path changes.

Higher RTTs yield slower recovery. We use the NRTOChanger(N=3,M=2) and repeat the experiment by increasing the delays of all the links between routers. Increasing the RTT delay from 6 ms to 24 ms yields a recovery 400 ms slower. This is consistent between 24 ms and 84 ms: the recovery is 500 ms slower. This increase is explained by two factors: the retransmission timeout and the size of the congestion window. The retransmission timeout increases with the RTTs and this slows down the recovery. Moreover, a higher RTT increases the congestion window which increases the number of packets to send before it is filled and therefore, before starting the retransmission timer.

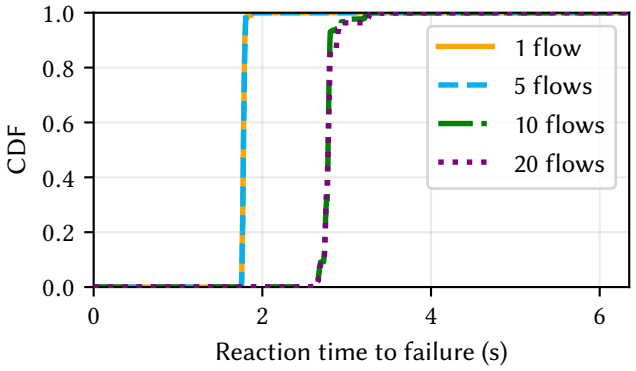

Figure 5.12: CDF of the recovery time with increasing numbers of parallel flows with NRTOChanger(N=3,M=2).

TPC failure detection works in presence of multiple flows competing for the network bandwidth. We start, on the same path, 1, 5, 10 or 20 connections in parallel, each using 10 Mbps. The bandwidth of the links being

100 Mbps, starting 10 or 20 flows triggers congestion and the data transfer becomes network limited. This explains the gap observed in Figure 5.12. With 10 or 20 flows, the window is larger because application-limited transfers do not increase their congestion window as much. A larger window means a longer time to send it on the wire. Other than that, the observed variance is slightly larger for 20 flows, but the difference is not significant.

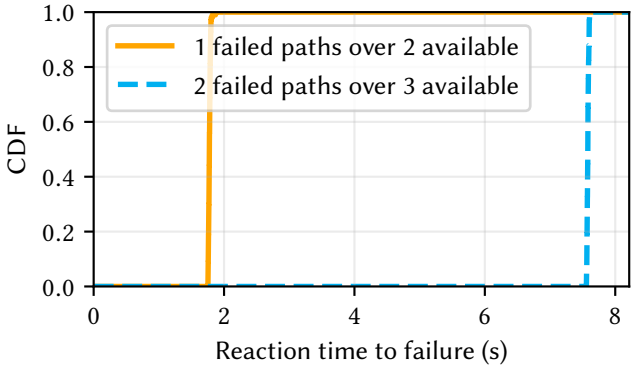

Figure 5.13: CDF of the recovery time with different number of failed paths with NRTOChanger(N=3,M=2).

TPC failure detection works even if more than one path has failed. We add one alternative interdomain path to the topology shown in Figure 5.9 and we introduce, after 10 seconds, a 100 % loss rate on two of the three paths. Even if this extreme scenario is unlikely to happen, it shows the resilience of TPC. Figure 5.13 shows that the impact of two failing paths out of three is larger than doubling the recovery time. Indeed, both the sender and the receiver need to converge on the correct path.

Figure 5.14 explains this increase. One line represents a retransmission from the sender or a duplicate acknowledgment from the receiver. The color of a line represents the path it is sent on. In our example, the first two paths (i.e., the red and blue ones) failed. Only the green one works. We see that the sender needs to change its path 6 times to force the receiver to move to the third path.

TPC failure detection converges faster on asymmetric failures. In the previous experiments, we considered that both endhosts had the same set of paths and that each path failed in both directions, but this might not always be the case. Either endhost might be on a valid path while the path used by the other endhost fails. To emulate asymmetric failures in our topology (see Figure 5.9), we introduce `tc-netem` on only one side of a link. In case of sender path failure, the time required to converge is similar to the sender

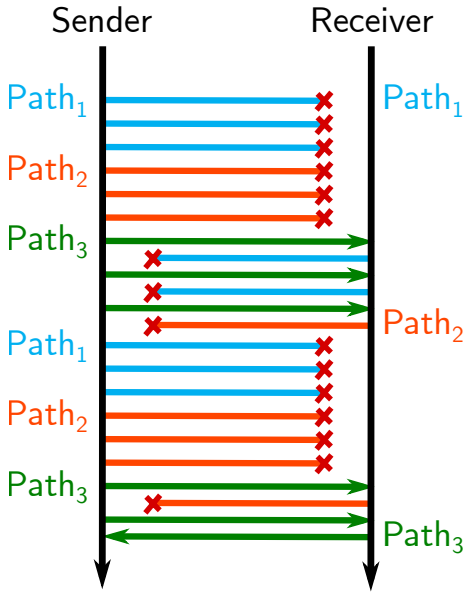

Figure 5.14: Recovery of a TCP connection when two of three paths failed with NRTOChanger(N=3,M=2).

reaction time on symmetric failures (see Figure 5.12). Indeed, upon symmetric failure, the sender is the first host to change its path. The receiver failure recovery in an asymmetric network takes two retransmissions and is thus, faster.

5.4 Dynamically selecting lowest-delay paths

The previous section has shown that by leveraging eBPF a TCP connection can react to network failures by changing its path. In practice, eBPF programs can be executed at different places in the TCP stack [Bra17]. In this section, we exploit this flexibility to enable TCP to autonomously find lowest-delay paths. Measurements have shown that different paths to the same destination can have different delay characteristics. Indeed, delays measured using `ping` vary even across equal-cost paths to a given destination [Pel+13]. In addition, measurements performed within a large cloud provider network [Red+20] show that the delays between servers located in different datacenters vary from one TCP connection to another. These two examples relate to the utilization of Equal Cost MultiPath (ECMP) for balancing packets across different paths on a per-flow basis. In Section 5.2, we discussed the existence of multiple peering links between a cloud provider and many of its external peers. Measurements on such peering links [Dha+18; FVD17] reveal that some of these links suffer from congestion. When the load on such a link increases, measurements

show that this first results in increased delays and then in packet losses (as link buffers become saturated). At any given time, different peering links will thus exhibit different delays, with the lowest delay path changing as the load on the peering links changes.

Our aim is to accommodate the mapping of transport-layer connections to low-delay paths. Doing so entails contending with three main challenges: (i) gaining host visibility into the delays of different available paths, (ii) determining which path each connection should be mapped to at any point in time, and (iii) enabling hosts to realize their choices of paths. This last challenge is solved by the architecture presented in Section 5.2. We next discuss the first two challenges, and how these are addressed by our schemes.

Selecting among available paths via online learning. A straightforward approach for choosing among different paths is to pick the path that exhibits the best performance at that point in time. Such a greedy approach, however, can have significant drawbacks, as discussed next. The performance of the chosen path might quickly deteriorate as additional connections are rerouted to that path. For instance, a path that currently exhibits low latency because its bandwidth is not fully utilized, can easily become congested once additional connections are mapped to it, leading to unacceptably long packet delays. This issue is further aggravated when decisions are made simultaneously and in an uncoordinated manner by different hosts, or for different connections from the same host. A too frequent rerouting of connections to (currently) better performing paths might also lead to traffic instability.

We observe that ideas from online learning theory (and, more specifically, multi-armed bandit theory [VBW15]) are naturally applicable to this challenge. In the multi-armed bandits (MAB) setting, a decision maker (*i.e.*, agent) repeatedly selects between different actions and observes, in hindsight, the implications of its chosen action on performance. In our context, the decision maker is the host and the actions correspond to different choices of paths to which a connection could be mapped (when initialized or while running). We show that utilizing a classical MAB algorithm with provable guarantees can yield both high performance and stability. Here, we focus on passive measurements because of technical limitations of the expressiveness of the eBPF interface in the Linux kernel. Section 5.4.2 details our solution for that challenge.

5.4.1 Path performance monitoring

To guide path selection, each host collects performance-related statistics for paths that carry its data traffic and stores these in a key-value cache. We next discuss our choices of the key of the cache, data to store, and data expiration time.

As in routing tables, data in the cache is aggregated by destination IP prefixes. To facilitate the aggregation of data by applications, the ports can also be included in the key or, alternatively, some applications can be assigned with unique identifiers, included in the key. The way data is aggregated has important implications; a very specific key will not be associated with sufficiently many connections to infer meaningful statistics while wasting memory, whereas a too broad key might encompass very different types of traffic, which would better be considered independently of each other.

The data itself consists of two elements: (i) the list of available paths (towards a certain destination IP prefix), and (ii) collected performance-related statistics for each available path. In SRv6, for instance, each path on the list will be represented as a sequence of segments, indicating the SRv6-capable routers the path traverses. Three natural path performance metrics to consider are throughput, packet loss ratio, and round-trip-time (RTT). To assign more weight to recent, and so more informative, measurements than to older ones, while not being overly sensitive to variance in measurements, we adopt the standard technique of keeping track of the exponential moving average of the monitored performance indicators (this approach is used, for instance, in the Linux TCP stack to compute the connection's smoothed RTT [PA00]).

5.4.2 Online Learning Path Selection

We now explain our approach for mapping a new connection to an available path. Recall that a greedy scheme, while simple, can induce undesirable effects for performance and stability, and so we adopt a different approach: employing an online learning algorithm, namely, EXP3 [BC12].

EXP3, described in Algorithm 4, is designed for the multi-armed bandits (MAB) setting. An *agent* is repeatedly faced with a choice between different *actions*. At each discrete time step $t = 1, 2, 3 \ldots$ the agent selects an action a_t from a fixed set of actions $\mathcal{A}$ and then learns, after the fact, the implications of selecting a_t, captured by an observed *utility value* u_t. Importantly, the agent only has visibility on the consequences of its chosen action a_t. Therefore, only its weight is updated. It cannot tell how *other* choices of actions would have impacted its derived utility value. The more rewarded the action is, the higher its weight increases, and the higher its selection probability increases. Another important feature of the MAB setting is that no assumptions whatsoever are made regarding how actions are associated with utility values. In fact, the utility values derived from selecting different actions can even be assumed to be picked *adversarially*. Despite these limitations on the feedback available to the algorithm and the unpredictability of the environment, algorithms like EXP3 provably provide meaningful benefits to the agent ("no regret" [BC12]) and convergence to equilibrium when multiple interacting

agents employ EXP3 [BC12].

Algorithm 4 EXP3 Algorithm.

Given $\Gamma \in [0, 1]$, set the weights: $w_a(1) = 1$ for all $a \in \mathcal{A}$
At each time step $t = 1, 2, 3 \ldots$,
  Set $p_a(t) = (1 - \Gamma)\frac{w_a(t)}{\sum_{b \in \mathcal{A}} w_b(t)} + \frac{\Gamma}{|\mathcal{A}|}$ for each action a
  Draw the action a_t randomly according to the $p_a(t)$ distribution
  Observe reward u_t at the end of time step t
  Define the estimated reward $\hat{u}_t$ to be $\frac{u_t}{p_{a_t}(t)}$
  Set $w_{a_t}(t + 1) = w_{a_t}(t)e^{\Gamma \hat{u}_t / |\mathcal{A}|}$
  Set $w_a(t + 1) = w_a(t)$ for each action $a \neq a_t$

MAB algorithms are natural to apply to our context. A host can be modeled as an agent that, for each connection destined for the same IP prefix, selects between different paths to determine which path the connection should be mapped to. Selecting a path constitutes the action. The empirical implications of a certain choice of path are only observable to the host in hindsight, and the performance that would have resulted from a different choice of path for the connection is unknown to the host. In addition, since path performance is often affected by many factors that are external to the host (such as the existence and behavior of connections originating on other hosts whose paths intersect this path), avoiding assumptions regarding path performance (the utility from selecting the path) is prudent. Γ captures the extent to which alternative paths are probed. If Γ is set to 1, no exploration is made. If Γ is set to 0, EXP3 degenerates to a uniform probability distribution over paths.

To demonstrate the feasibility of our proposed scheme, we implement a path selection mechanism that monitors TCP connections that require low delays. We aggregate path performance by destination IP prefix and monitor the smoothed RTT. We use IPv6 Segment Routing to select paths and the EXP3 weights, with respect to the paths towards a specific destination IP prefix, are updated once a connection towards that prefix terminates.

Listing 7 shows the pseudocode of this path management. We implement the cache using two eBPF maps: one for the list of paths and one for the connections, as shown in Figure 5.8. Our TPC path manager initializes its connection structure and sets the path upon the end of the three-way handshake on server-side as the TPC presented in Section 5.4. We use the EXP3 algorithm to select the path at the `BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB` hook. Every RTT, using the `BPF_SOCK_OPS_RTT_CB` hook, we check if we transmitted more than a given hook activation rate. If we did, we reward the path and restart a new path choice. If the chosen path changes, we update the SRH. The hook activation rate is an amount of sent data after which TPC rewards its path. For short-lived connections, we reward the path once, at the end of

Listing 7 Path management.

```
int handle_sockop(struct bpf_sock_ops *skops) {
  conn = bpf_map_lookup_elem(&conn_map, &five_tuple);

  switch (skops->op) {
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
      // End of three-way handshake
      new_conn = new_connection_state();
      flow_info->srh_id = choose_srh_exp3();
      new_srh = get_srh(flow_info->srh_id);
      move_path(skops, new_srh);
      break;
    case BPF_SOCK_OPS_RTT_CB:
      // Every RTT
      if (skops->snd_nxt - conn->last_seq >= HOOK_RATE) {
        conn->last_seq = skops->snd_nxt;
        reward_path_exp3(flow_info->srh_id, skops->srtt,
                         flow_info);
        flow_info->srh_id = choose_srh_exp3();
        new_srh = get_srh(flow_info->srh_id);
        move_path(skops, new_srh);
      }
    case BPF_SOCK_OPS_STATE_CB:
      // End of a connection
      if (skops->args[1] == BPF_TCP_CLOSE) {
  reward_path_exp3(flow_info->srh_id, skops->srtt,
                         flow_info);
        bpf_map_delete_elem(conn_map, &five_tuple);
      }
      break;
    // [...]
  }
  return 0;
}
```

a connection, using BPF_SOCK_OPS_STATE_CB before cleaning up the state of the connection.

Listing 8 Reward the path.

```
void reward_path_exp3(int srh_id, int srtt,
          struct flow_info *flow_info) {

  flow_info->max_srtt = max(flow_info->max_srtt, srtt);
  observed_reward = flow_info->max_srtt - srtt;
  estimated_reward =
    observed_reward / flow_info->path_probability;

  // Update weights
  exp = exponent(e, GAMMA * estimated_reward
        / nbr_paths);
  weight = get_weight(srh_id) * exp;
  set_weight(weight, srh_id);
  normalize_weights();
}
```

Our implementation of EXP3 requires us to define the reward function. We target the path with the smallest latency so our EXP3 reward needs to include it. Moreover, we need an expression that is bigger when a path needs a high reward. This reward needs to be positive. Otherwise, EXP3 would decrease the probability of using a path each time we use the path, even if it is the best path available. Therefore, we define the reward as $SRTT_{max} - SRTT$. $SRTT_{max}$ is the highest observed SRTT value for the connection so that the reward is guaranteed to be positive. The pseudocode of this reward computation is detailed in Listing 8.

Listing 9 Weights's normalization.

```
#define NBR_TOKENS 10000

void normalize_weights() {
  for (i = 0; i < weights_len; i++) {
    weight = get_weight(i) * NBR_TOKENS / sum_weights();
    if (weight < 1)
      weight = 1;
    set_weight(weight, i);
  }
}
```

By default, the EXP3 algorithm is designed to reach equilibrium. However, in networks, various events can disturb it. If we let EXP3's weights exponentially increase in stable times, they will quickly reach the maximum floating point number that we can represent. Moreover, the increase of the best path's weight will deepen the difference with the other weights even if it does not actually change much in the probabilities of using the path. When the best path becomes undesirable, e.g., because of a congestion or a BGP rerouting on the Internet, it will take much time to reduce this difference and start using the other paths. For instance, TPC identifies a path as the optimum with a probability strictly above 99.99% but its RTT changes. If its weight is 99.99 (and the other path has a weight of 0.01), it will only take a few iterations/measurements to increase the other path's weight enough for it to be used. If the current path's weight was instead 2^{64}, it would have required many more iterations to let the other path being used. Both weight distributions will yield similar probabilities during stable phases. Therefore, we normalize paths' weights to optimize TPC's reaction time. We chose to limit it to $[1, 10000]$ so that it is at most possible to reach 99.99% of connections on the best path (if there are two alternative paths). On servers with millions of connections, it is interesting to increase the upper limit in order to go above 99.99% of probability during stable phases. Listing 9 shows how this is implemented in practice. We normalize the paths' weights after rewarding any path, at the end of a connection.

We emulate networks with IPMininet [Jad+20], an abstraction layer above Mininet [Han+12], on a Linux kernel 5.3 running on a server equipped with 20 CPUs and 16 GB of RAM. To emulate links delays, we use `tc-netem`. We use a `tc-htb` to emulate the bandwidth. The other parameters of these tools are the default ones.

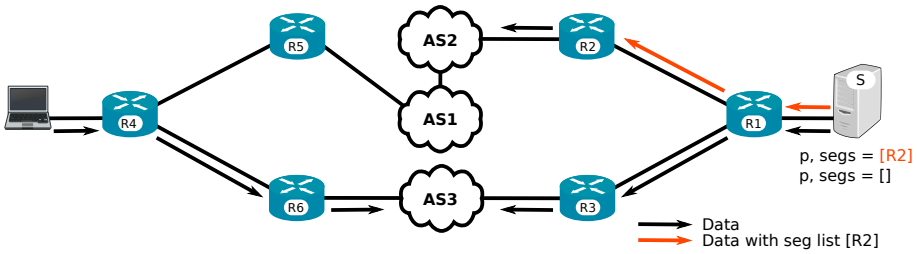

Figure 5.15: Evaluation topology.

We want to evaluate TPC for the interdomain use case shown in Figure 5.15. A CDN server is queried by endhosts in distant ASes. The AS of the server provides two paths to TPC. Both paths have different latencies that will change over time. We use our TPC in the server to find the lowest RTT path. The client does not run TPC and thus, only the server can change its path.

This network uses links with the same bandwidth (100 Mbps). The lower interdomain path has an RTT of 10 ms while the upper path has this RTT multiplied by a given factor (10 by default). The client always uses the lower path.

Our evaluation represents a set of clients that retrieve small files from a web server. The server includes our TPC with EXP3. The clients use Apache Benchmark to continuously download a 100 kB file from a `lighttpd` server during 50 seconds. We configured Apache Benchmark to use 4 parallel connections. Apache Benchmark creates one connection for each HTTP request and TPC is executed when the connection closes. After 10 seconds, we invert the delays of both paths in the direction from the server to the client. We use `tc-netem` to change the link delays. Then, we observe both the completion time of the requests and the time needed for EXP3 to converge to the working path at the initialization and after the inversion of the path delays. We consider that EXP3 has converged once the lowest-delay path carries over 90% of the weights' sum. We use a uniformly random selection of the paths as a baseline to compare the benefits of EXP3 in TPC. This baseline is implemented through static routes adding one of the SRH that are load balanced using ECMP.

TPC uses the quickest path as opposed to a uniformly random solution. Figure 5.16 shows the CDF of the completion time of each request

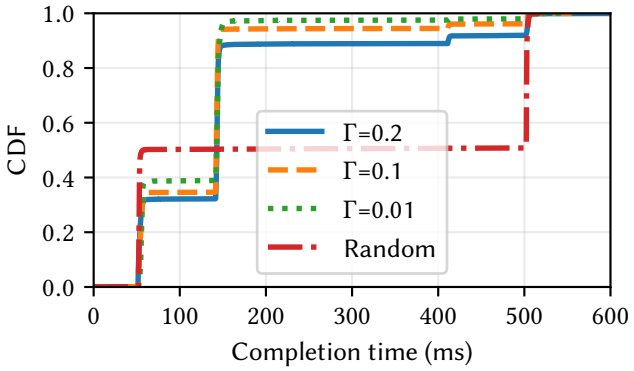

Figure 5.16: Completion times of HTTP requests. TPC outperforms a purely random selection and a low value of Γ provides better results.

with TPC either using a uniformly random choice or EXP3 with various Γ values. Without learning from experienced delays, the uniformly random heuristic cannot learn anything from the network and therefore half of the connections use the longest delay path. However, TPC using EXP3 with $\Gamma \neq 1$, learns and quickly adapts to delay changes.

We observe roughly 4 different completion times in each TPC curve, instead of 2 for a uniformly random heuristics. The fastest and longest of these four completion times are experienced by connections whose traffic goes through the lowest RTT path and the highest RTT path, respectively. The two completion times in between pertain to connections whose SYN+ACK is not sent along the path used to carry the data. Recall that since we do not control the path for the SYN+ACK packets in Linux, this path is always the lower (and default) path whose delay changes during the experiment. This accounts for the distance of the second completion time from the first, and of the third completion time from the fourth. In the uniformly random heuristics, the rest of the traffic always uses the same path as the SYN+ACK packet.

When the situation is stable, the lower the Γ, the better. In Figure 5.16, we also observe that the Γ value impacts the rate of connections using the quickest path over the 50 seconds of each experience. As described in Algorithm 4, every path has at least a probability of $\frac{\Gamma}{|Paths|}$ to be chosen. Intuitively, Γ is the percentage of uniformly random choices that TPC makes. In our example with two paths, at most $(1-\Gamma) + \frac{\Gamma}{2} = 1 - \frac{\Gamma}{2}$ of the traffic can be routed on the right path if Γ is fixed and the traffic is stable. Thus, with Γ set to 0.2, TPC cannot send more than $1 - \frac{0.2}{2} = 90\,\%$ of the traffic on the best path. There are only two convergences in this experiment and therefore, probing too much for only one change hurts more connections than converg-

ing slower. For instance, setting Γ to 0.01 is a better choice in the big picture while it converges 6 times slower than setting Γ to 0.1. A real deployment of the TPC would help to refine the choice of the Γ over time. This is left as future work.

A higher Γ enables a faster convergence. Indeed, with a high Γ, we often measure the path delay changes because this parameter gives the rate of connections that are routed randomly. With Γ set to 0.1, 90 % of the convergence times are lower than 1.25 s. If we set Γ to a tenth of that, 0.01 %, the convergence time increases to 5 s. Therefore, with unstable paths, when we expect frequent delay changes, Γ should be set higher.

A larger difference in path delays only slightly speeds up convergence. We change the delay factor between the quickest and the slowest paths. We use two, five and ten times the quickest path RTT of 10 ms. Increasing the delay factor beyond five times does not change the convergence time. Even with a delay factor of two, the increase of delay did not exceed 100 ms in 95 % of the runs. Indeed, the reward is more important in Listing 8 when the maximum smoothed RTT is higher. TPC works better in heterogeneous networks and this is in these networks that TPC's purpose is the most useful.

The higher the hook activation rate is, the quicker TPC converges. This hook activation rate is influenced by the size of the transfer since we use a hook at the end of the TCP connection. Logically, more information about the state of the paths yields faster convergence. With a hook activation rate of 1 kB, 90 of the 100 runs are below 0.75 seconds while with a hook activation rate of 1 MB, the 90 percentile is at 2 seconds.

If we do not change the hook activation rate of each connection but we increase the number of simultaneous connections, paths will be more rewarded. When TPC runs on 2 parallel connections instead of 4, paths are rewarded twice less. Therefore, we observe a decrease of the convergence time of 0.5 s. Only leaving one connection adds 1 second of delay in average.

These emulation-based evaluations have the advantage of being easily reproducible with little material. They demonstrate the behavior of TPC when tweaking its parameters. However, to fully demonstrate its performance, larger-scale performance tests over Internet are required. For instance, they would allow us to characterize TPC's behavior in unstable situations: each path is the best a given percentage of the time (e.g., 60% of time, the first path is good and the rest of the time, it is the other paths). We expect TPC to reach an equilibrium matching observed performance (e.g., weight of the first path is 60% of the weights' sum). These evaluations are left for future work.

5.5 Related work

5.5.1 Recovering from distant failures

CPR [Lan+21], Blink [Hol+19] and INFLEX [Ara+14] are the closest related work to our first use case, as they support recovery from remote failures. This section first presents these three solutions. Then, we compare each of them to TPC based on the genericity of the solutions, the deployability of the technologies, the endhosts's interactions if any, the network state overhead, the bandwidth overhead and the granularity of the remote failure detection.

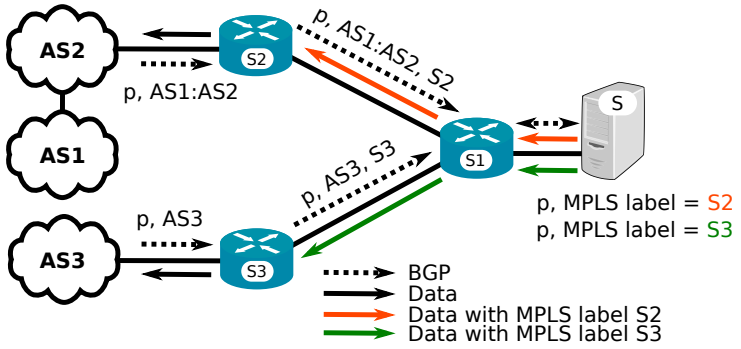

Figure 5.17: Example network using CPR architecture to recover from remote failures.

CPR can be installed in the servers in Edge Networks of Content Distribution Networks. Figure 5.17 shows an example network using CPR. CPR is instantiated a MPLS dataplane. Each exiting BGP peer has an assigned MPLS label that the other switches use to reach the edge of the network. Each border MPLS switch connects to each access MPLS switch to share all the BGP routes. Each access switch appends the MPLS label of the BGP peer to the BGP route before forwarding it to each of its endhosts.

Each endhost has a modified Linux kernel that detects failures similarly to the sender side of TimeoutChanger(T,M). If the server only retransmits data and does not send any new segments for T time, CPR increments a `reroute counter`. The current value of this counter is added as in the 4th most significant bit of the firewall mark. If these bits are set to zero, the default path is selected. Otherwise, the packets chooses the nexthop in another table using ECMP over the possible routes. Each route has an associated MPLS label representing a BGP border switch. The hash computation is tweaked by CPR's kernel patch to take the firewall mark into account. Therefore, incrementing the `reroute counter` of a socket reroutes the flows towards another BGP peer of the AS.

INFLEX [Ara+14] uses a classical OpenFlow dataplane and receives rerout-

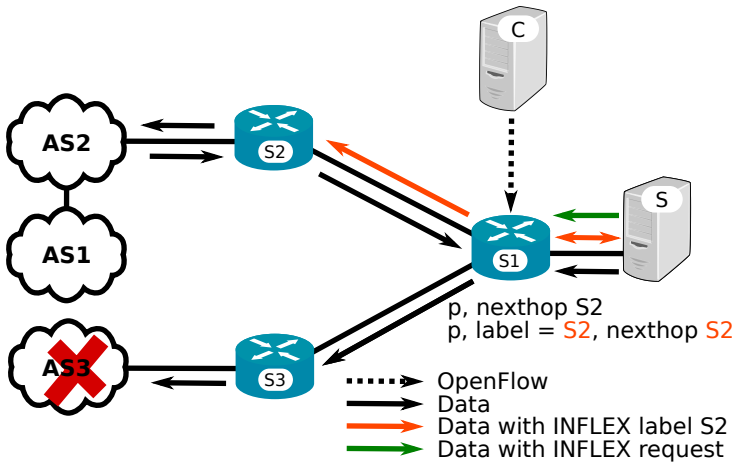

Figure 5.18: Example network using INFLEX architecture to recover from remote failures.

ing requests from servers when they lose connectivity. Figure 5.18 shows an example network using INFLEX. Several backup paths are prepared by the controller and installed in the OpenFlow switches: the lower path is the default choice but, with a label set in the IP header, the upper path can be taken. Server S has a patched kernel that tracks the retransmission timeouts. When a server retransmits data, it sets an INFLEX request bit in the DSCP field of its IP header to request a new path. The edge switch, S1, upon reception of this packet sets the INFLEX label S2 in the DSCP field of the IP header. This label is used by the core switches to forward the packet. In this example, there is no core switch but that could be the case. After sending this packet, if the edge switch receives traffic destined to this server, without the label, this means that the rerouting worked and the edge switch sets the packet INFLEX label S2 to the same value. This newly received packet let the server know that it should use the label received for its future packets.

Blink [Hol+19] is a remote failure detection mechanism deployed on the network nodes seeing a consequent share of traffic, e.g., the border router. Figure 5.19 shows an example of network using Blink on both its border routers. R1 and R2 receive the alternate routes from each other using BGP. When R1 detects a remote failure on the path through AS3, it reroutes this traffic to another border router, R2. This assumes that the Blink node can either tunnel the traffic to this border router, e.g., through the use of MPLS or IPv6 Segment Routing. In this example, both routers are directly connected so this is not necessary. Because of the limited capabilities of a network node, Blink is restricted to the detection of remote failures that affect the most active flows. Failures affecting a few TCP connections may not be detected if

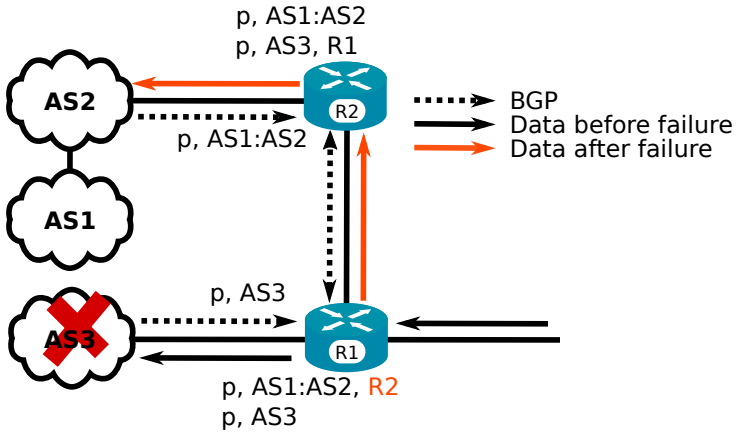

Figure 5.19: Example network using Blink on both border routers to recover from remote failures.

there are a lot of active TCP connections. However, by focusing on failures affecting multiple flows, it has a lower chance of false positives, even when reacting quickly. As NRTOChanger(N,M), Blink tracks the retransmissions of the monitored TCP flows. If this number is high enough, the next path alternative is selected. Because of its decentralized nature, loops of rerouting can exist for a short time. For instance, if there is a failure inside AS1 or AS2, R2 will detect it and decide to send traffic back to R1, effectively creating a traffic loop. Blink has the ability to detect such loops and to break them. In our example, R1 will switch back to AS3, even though the path is down.

Genericity TPC is more generic than CPR, INFLEX or Blink. TPC recovers from failures affecting the return path as well (see Section 5.3.1) while the other solutions cannot. They only monitor the retransmission of data and not the retransmission of duplicated acknowledgements.

Deployability The authors of CPR use a MPLS dataplane but the solution would work in a SRv6 dataplane, thus the choice of the tunneling protocol does not hurt the deployability of CPR. However, CPR does rely on kernel patches. This implies that the kernel of every protected server must be updated on each upgrade of CPR. While TPC requires some kernel modifications to extend the eBPF API, these modifications occur only once and will benefit other eBPF programs. INFLEX both requires an OpenFlow dataplane and kernel modifications which hurt its deployability since most of the networks do not use OpenFlow switches. Blink only requires P4 switches at the AS' border to analyze the traffic. This is easier than upgrading the kernels of the endhosts depending on the control of the network operators on the endhosts.

Endhosts' Interactions CPR, INFLEX and TPC are installed on the endhosts, and they rely on transport data to detect path failures. TPC adds the reaction to the sending of multiple duplicated acknowledgements of the same segment, which allows more robustness than CPR when a failure only affects the return traffic.

Network State Overhead Both TPC and CPR do not require state inside the network. Each endhost remembers the mapping for its TCP connections and the list of available paths. They have the lightest impact on the network state. INFLEX places the responsibility of the mapping between a flow and its path to the edge switch, but the edge switch only has to remember the mapping during a rerouting request. When the endhost has changed the DSCP field of its packets, the network state is not necessary anymore. Blink has the highest impact on the network state because it tracks the retransmission of the active connections by keeping track of the last sequence number sent for each monitored flow. To limit the network overhead, Blink bounds the number of monitored TCP connections by selecting the most active connections. These connections are the most likely to be the first to retransmit data in case of remote failures.

Bandwidth Overhead INFLEX is the only solution listed here that does not add a header to rerouted TCP packets. Indeed, it uses the DSCP field of the packets and an OpenFlow dataplane to switch to another path. CPR, Blink and TPC use a tunneling strategy, either MPLS or IPv6 Segment Routing, to use another path.

Granularity INFLEX, CPR and TPC are able to detect remote failures for every TCP connection because they tap into the endhosts' TCP state, either through kernel patches or eBPF. Blink only detects failures affecting multiple monitored flows. This limitation comes from the necessity of bounding its memory usage. The other solutions, being deployed on each endhost, do not have this issue.

5.5.2 Dynamically selecting the lowest-delay paths

The closest related work to our second use case are REPLEX [FKF06], CONGA [Ali+14] and Clove [Kat+17]. This section first presents these three solutions. All three of them are performing a dynamic rerouting portions of the traffic to achieve a given objective. Then, we compare each of them to TPC based on the genericity of the solutions, the deployability of the technologies, the endhosts's interactions if any, the network state overhead, the bandwidth overhead and the granularity of the dynamic path optimization.

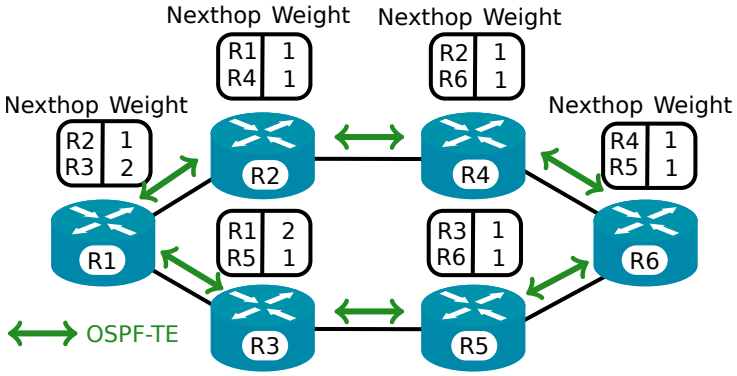

Figure 5.20: Example network using REPLEX to perform dynamic traffic engineering within the local network.

REPLEX [FKF06] performs traffic engineering by running its distributed algorithm on every router of an IGP (e.g., OSPF) network without controller as shown in Figure 5.20. This algorithm is based on game theory. Every router monitors its link usage and shares it through traffic engineering extensions of IGP protocols (e.g., OSPF-TE [Gia+15]). REPLEX routers then update IGP weights to change the routing, and thus, to minimize the load on each link utilization.

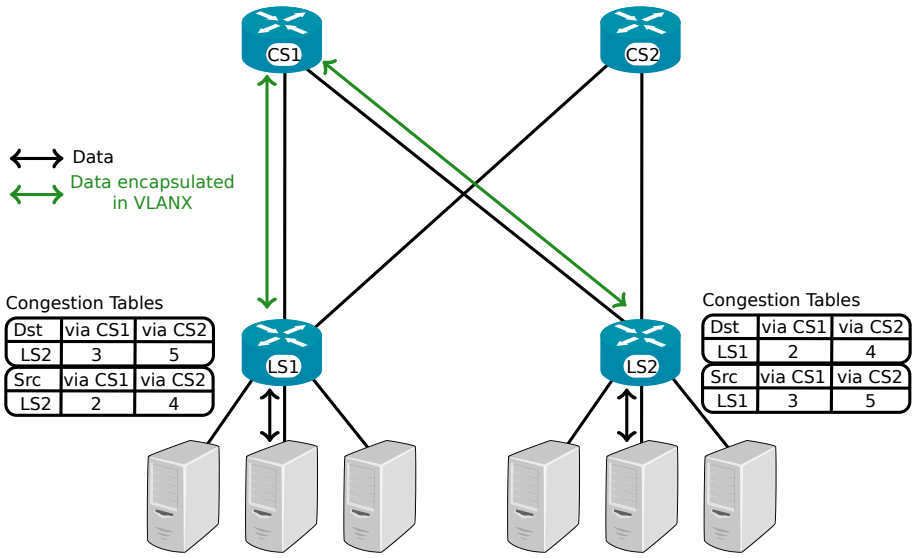

Figure 5.21: Example of datacenter network using CONGA to dynamically load balance flows within the local network.

CONGA [Ali+14] is a distributed load balancer for datacenters intended to minimize intradomain flow completion times. Figure 5.21 shows a small

example of datacenter network using CONGA. When a server connected to LS1 sends traffic to a server connected to LS2, LS1 first has to decide on which path to send the flow. For that, it looks up the first of its congestion table. For each next hop, CS1 and CS2, a value representing the current congestion is encoded. LS1 selects the path with the lowest congestion value. LS1 encapsulates packets in a VLANX [Mah+14] frame so that CS1 and CS2 can write their congestion value into the VLANX header when they forward packets. When receiving the encapsulated packet, LS2 writes down the new congestion value to its second congestion table. This congestion value is sent back to LS1 inside the VLANX header on the next packet that LS2 needs to send to LS1. Upon reception of this reverse traffic, LS1 updates its first congestion table.

Clove [Kat+17] tackles the same problem as CONGA, but it is running on the hypervisors' virtual switches present on the endhosts of a datacenter. CONGA only works in a fat-tree topology with only one intermediate level, Clove is more generic. It relies on encapsulating the packets in an outer IP/IPv6 header and choosing a specific source port that causes ECMP to take the desired path. Clove performs traceroutes with many source ports to discover the source ports that take different paths. Clove needs to perform this source port investigation periodically in case of topology changes that would cause ECMP updates. To monitor the congestion level of each path, Clove has two variants: Clove-ECN and Clove-INT.

Figure 5.22 shows an example of datacenter using Clove-ECN. In this example, S1's virtual switch forwards traffic to the hypervisor of S2. The traceroutes caused it to identify port 1001 as a source port to use CS1 and port 2002 to use CS2. At the beginning, each path has the same weight. In the example, it first tries CS1 by encapsulating the packets into an outer IP header with 1001 as the source port. CS1 being congested, it sets the ECT bit in the IP header of the forwarded packets. S2's virtual switch encodes the congestion into the Stateless Transport Tunneling (STT) header [DG16] added to the outer IP header of the reverse traffic. When the reverse traffic reaches S1's virtual switch, the weight of the first path is reduced because of the congestion data carried in it.

Figure 5.23 shows an example of datacenter using Clove-INT. As in the previous example, S1's virtual switch forwards traffic to the hypervisor of S2 and selects the path going through CS1. In addition to the outer IP encapsulation, S1's virtual switch adds an In-band Network Telemetry (INT) [Kim+15] header to request the current usage of the links along the path. LS1, CS1 and LS2 all modify this header to give the current usage of their links. S2's virtual switch encodes the actual usage state of the most congested link into the STT header added to the outer IP header of the reverse traffic. When the reverse traffic reaches S1's virtual switch, the weight of the first path is adapted in

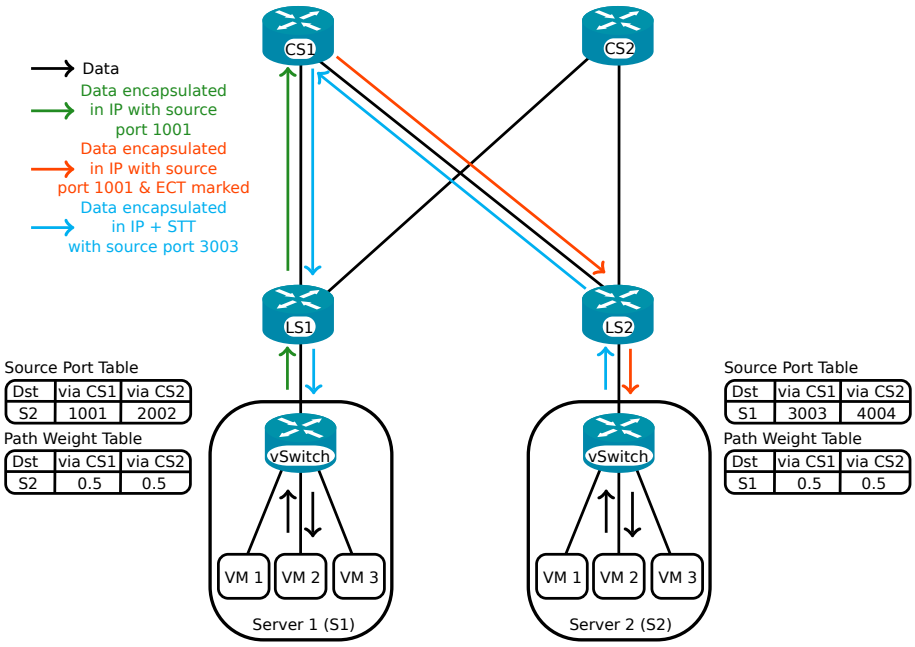

Figure 5.22: Example of datacenter network using Clove-ECN to dynamically load balance flows within the local network.

proportion to the congestion experienced by the most congested link.

Genericity CONGA only works in a fat-tree topology with only one intermediate level. Clove solves this issue by leveraging an exploration of the available paths using traceroute. However, this only works in symmetric topologies such as datacenters that have access to most of the topology with ECMP (such as in a fat-tree topology). Clove also requires both endpoints to be Virtual Machines with a virtual switch on the machine. REPLEX has limited flexibility when updating the link usage because it can only change the shortest path from one end to the other end of the network. CONGA, Clove and REPLEX can reroute any type of traffic, not only TCP. TPC does not natively support another protocol than TCP. However, TPC supports remote path optimizations. Clove-ECN could perhaps be extended to support remote path load balancing but Clove-ECN requires the other endpoint to support Clove-ECN while TPC does not.

Deployability REPLEX and CONGA require software updates on every router or switch of the network. Clove-INT also requires that to deploy INT protocol to measure the congestion level of the path. Clove-ECN requires ECN to be enabled on the network, which makes it the easiest variant to de-

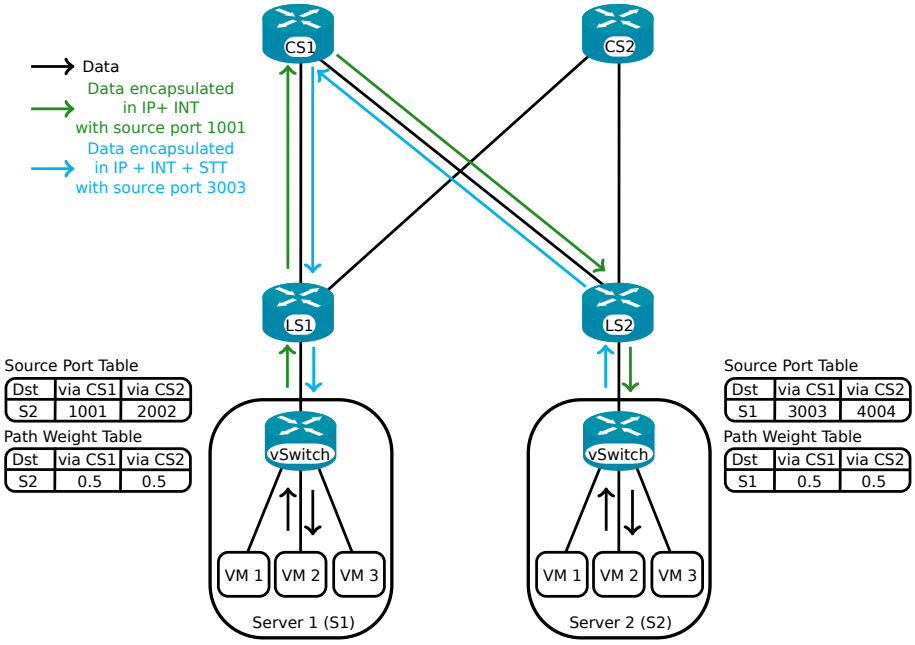

Figure 5.23: Example of datacenter network using Clove-INT to dynamically load balance flows within the local network.

ploy. For both Clove's variants, each virtual switch needs to be updated. TPC requires some kernel modifications to extend the eBPF API, but these modifications occur once and will benefit other eBPF programs. Any future upgrade of TPC's algorithm only requires a software update of the endhosts.

Endhosts' Interactions TPC is the only one of these four solutions to interact with endhosts.

Network State Overhead Because it optimizes the IGP weights, REPLEX only requires to keep the congestion state of each link. TPC keeps the state on the endhost which is the most scalable solution when needing per-connection data. Clove and CONGA keep the mapping between a connection and its path on, respectively, the ingress virtual switch or the leaf switch. They need to hold more state than an endhost because there are more than one endhost linked to them.

Bandwidth Overhead Clove, CONGA and TPC all have bandwidth overhead because of their encapsulation. Clove uses IP encapsulation, INT and/or STT to enforce a path and exchange congestion data, CONGA uses VLANX for the same purpose, and TPC uses SRv6 to enforce a path. REPLEX needs

to enable OSPF-TE extension to flood regularly the congestion of each link. This flooding induces a bandwidth overhead.

Granularity Clove, CONGA and TPC all allow rerouting each connection individually while REPLEX changes the IGP weights, thus, it cannot reroute differently two connections communicating with the same destination.

5.6 Conclusion and Discussion

TCP was designed to depend on the network layer for selecting the path used to reach any given destination. This separation of functions between the transport and network layer implies that TCP can only react in two ways to network-level events: the retransmission of lost packets and the adaptation of its transmission rate.

We have proposed the TCP Path Changer (TPC). TPC makes TCP more agile by enabling the TCP stack to change the current network-layer path in reaction to different types of events. Thanks to eBPF, each application can inject its own TPC into the underlying Linux TCP/IP stack. To illustrate the benefits of TPC, we have applied it to two very different use-cases. First, we have shown that TPC can detect distant link failures and react quickly by rerouting the affected connections, and also monitor the health of a connection and reroute it when needed. Second, we demonstrated how servers can use our TPC to find small delay paths.

The agility that we give to the endhost could be a problem if TPC eBPF programs misbehave due to implementation mistakes. For instance, all the traffic could be routed through the same path, worsening the end user experience in comparison to a simple ECMP rule. Network operators need to monitor their network and fall back to regular routing if they notice anomalies. This monitoring and fallback mechanisms could be implemented in a controller.

Our plans for future research include applying TPC to other use-cases (e.g., rerouting connections to better utilize network bandwidth), to collect measurements from production traffic, and to design ways to monitor and take back the control from compromised endhosts.

This chapter extends the Software Resolved Network solution, presented in Chapter 2, by using eBPF, whose capabilities were presented in Chapter 4. TPC empowers a set of trusted endhosts even more in the path selection process while allowing applications to stay agnostic to SRN and to benefit from our solution. We use the flow information of TCP to make an enlightened path selection.

Conclusion 6

In this thesis, we design and implement methods to expose path diversity to non-multipath transport protocols. We use IPv6 Segment Routing to give the trusted endhosts partial control of their paths. Trusted endhosts are IT-controlled endhosts: laptops given to employees without an admin access or servers. We also showcase the benefits of this exposition through several use cases. Using an SDN-based architecture, network operators can use a logically centralized controller to compute and filter the paths to expose to the trusted endhosts.

In doing so, we introduce Software Resolved Networks in Chapter 2, an SDN-like architecture that introduces collaboration with trusted endhosts. Like OpenFlow-based SDN solutions, SRNs enable the network operators to specify policies that control the network paths, represented as IPv6 Segment Routing Headers, that are used by applications. Applications use the DNS as a signalling protocol to request the creation of end-to-end network paths. Those paths are computed by a controller that interacts with the DNS resolver and returns the selected path to the requesting application as a segmented SRv6 path.

There are two important differences between SRNs and traditional SDNs. First, SRNs enable the applications to directly interact with the controller to specify path requirements like delay or bandwidth. Second, SRNs do not require per-flow state in all network nodes. The controller creates state on the access routers but not on the core routers. This improves the scalability of SRNs. We implement a complete Software Resolved Network on Linux hosts, routers and servers. Our performance evaluation demonstrates that our prototype meets the performance expectations of enterprise networks.

We explore in, Chapter 3 a new optimization algorithm, CG4SR, that can be used in our SRN controller to generate paths. CG4SR is a scalable approach that leverages column generation to efficiently solve the traffic engineering problem in networks using Segment Routing. CG4SR reaches full control over the maximum number of segments by supporting both node and adjacency segments while existing techniques only support node segments. Our experiments demonstrate that CG4SR is faster than previously proposed ILP approaches. Compared to heuristics, CG4SR provides a tight bound that evaluates the near optimality of its solution while heuristics are fast but cannot

provide any guarantee.

In parallel to the realization of the thesis, eBPF has been increasingly adopted in the Linux kernel. Chapter 4 takes a broader look at the capabilities of TCP customization in eBPF. We advocate for a truly extensible TCP, where most of its optional features would be injected from user space through eBPF. This would accelerate the development of future extensions of TCP such as the programs proposed in the other chapters of this thesis. We started by defining what a minimal and modular TCP stack, that we call xTCP, would look like. Then we surveyed the progress of the Linux TCP stack to xTCP's modularity. Finally, we implement three extensions to the eBPF API, namely custom timers and the parsing of incoming IPv6 extension headers in packets.

SRNs have the disadvantage to require applications to explicitly use our library to communicate with the controller and obtain the paths. While applications can explicitly pass requirements for the paths, this is not realistic to recompile every application on a trusted endhost. With eBPF, we can dynamically load programs that modify the behavior of the network stack. Therefore, we only need one daemon application on the endhost to receive the path information from the controller. This daemon loads an eBPF program to change paths of some or all the other connections on the endhost.

In Chapter 5, we have proposed the TCP Path Changer (TPC). TPC makes TCP more agile by enabling the TCP stack to change the current network-layer path in reaction to different types of events. To illustrate the benefits of TPC, we have applied it to two very different use-cases. First, we have shown that TPC can detect distant link failures and react quickly by rerouting the affected connections, and also monitor the health of a connection and reroute it when needed. Second, we demonstrated how servers can use our TPC to find small delay paths.

Software artifacts

To demonstrate and evaluate the proposed solutions of the thesis, we implemented and extended several programs. We release these implementations in open-source to help others reproducing the results in this thesis.

The emulations of SRN (see Chapter 2) and TPC (see Chapter 5) are made above IPMininet [Jad+20], an abstraction layer above Mininet [Han+12] that supports the emulation of complex IP networks. This Python library of over 9200 lines of code is written in collaboration with Olivier Tilmans, and is available at `https://github.com/cnp3/ipmininet`.

The Software Resolved Network components are implemented at `https://github.com/segment-routing/srn`. Since deploying an SRN is non-trivial, we provide a small library that adds the necessary abstractions to IPMininet. We provide in the same repository the scripts that we use for

SRN's evaluation. These 1100 lines of Python code can be found at `https://github.com/segment-routing/srnmininet`.

CG4SR (see Chapter 3) is integrated in a fork of Repetita [GSV17]. Repetita already includes several Traffic Engineering's algorithms, and thus, it allows the reproducibility of our experiments. The repository is available at `https://github.com/jadinm/Repetita`. Our patch consists of over 3600 lines of Java code.

The extensions to the TCP stack presented in Chapter 4 are available as well at `https://github.com/jadinm/tpc-kernel`. They consist of 100 lines of C code. The eBPF programs (about 200 lines of C code) are released at `https://github.com/jadinm/tpc-ebpf`. They can be run reusing the scripts at `https://github.com/jadinm/tpc`.

To build TPC (see Chapter 5), we wrote three eBPF programs of 1100 lines that are available at `https://github.com/jadinm/tpc-ebpf`. We also include our kernel patches of 470 lines at `https://github.com/jadinm/tpc-kernel`. Finally, as for SRN, we release the daemon code (1000 lines of C code), as well as the emulation and evaluation scripts (3700 lines of Python code) at `https://github.com/jadinm/tpc`.

Open challenges

In Chapter 3, we design CG4SR, an approach that leverages column generation to efficiently solve the traffic engineering problem in networks using Segment Routing. Because of the nature of the mincost-srpath algorithm (see Section 3.5.1), we can accommodate other constraints on the paths than the maximum number of segments, e.g., set a limit on the latency or forbidden/-mandatory intermediate nodes (for service chaining constraints [Zha+13]). It would be interesting to explore the extent of this modularity and evaluate these constraints.

In exploring eBPF capabilities in Chapter 5 and Chapter 4, we had to cope with the limitations of the Linux kernel's eBPF verifier. Even when performing checks in the C code for always valid accesses, the verifier can fail to understand that our code is valid. For instance, when accessing to an index in an array, we need to check that the index does not exceed its length. The verifier reads the compiled eBPF code and only checks that the register value is sufficiently constrained to never be out of array's boundaries when making a memory access inside the array. However, there are only 10 registers in the eBPF VM and if the register value is overwritten between the check and the memory access, the Linux kernel's verifier will not track it and it will fail to validate the program.

The verifier does not track boundaries outside the register for performance reasons. Its goal is to verify and load the eBPF program in a reason-

able amount of time. We advocate that an offline approach would allow a deeper and more accurate analysis. The authors of Pluginizing QUIC [De +19] present a secure way to offload verification the verification of eBPF programs. Their architecture could be applied for the Linux kernel as well to load larger and more complex programs, already verified offline.

In Chapter 5, we present different algorithms of path selection. They are based on retransmission timers and observed RTT. Their evaluations are emulation-based. This has the advantage of being easily reproducible with little material. However, to fully demonstrate its performance, larger-scale performance tests over Internet are required.

Future research could extend TPC to use attained network bandwidth as criteria to switch paths for TCP connections requiring more bandwidth. However, moving large flows could lead to network instabilities. To mitigate that, the hook activation rate should follow an exponential backoff strategy whenever a connection is moved. Moreover, we should prevent TCP connections from changing their path if they already have a sufficient amount of traffic. Applications such as videoconferencing applications typically know which amount of bandwidth they require to guarantee an acceptable quality.

The agility that we give to the endhost could be a problem if TPC misbehaves due to implementation mistakes. For instance, all the traffic could be routed through the same path, worsening the end user experience in comparison to a simple ECMP rule. Network operators need a monitoring tool to control misbehaving endhosts and fall back to regular routing if they notice anomalies.

In Chapter 2, applications use the DNS as a signalling protocol to request the creation of end-to-end network paths. They could use DNS to request TPC eBPF programs matching their objectives or the network operators' policies. For instance, our eBPF program to find the shortest delay path could be requested by interactive applications.

Other single-path transport protocols could be extended to support these different algorithms of path selection. For instance, videoconferencing applications typically use RTP and RTCP [Sch+03] instead of TCP. Another example is QUIC [IT21] which is increasingly used. QUIC supports sending multiple streams of data without end-of-line blocking. One of the streams could be used to actively probe the available paths. TCP on the other hand cannot actively probe paths without using application data. Using application data could cause retransmissions and additional delay, so we rely solely on passive measurements in TPC.

In this thesis, we enabled single-path protocols to switch these paths. However, multipath protocols, such as MPTCP [For+13; Hur+18], MPRTP [SAO13] or MPQUIC [DB17; RHB18], could also benefit from receiving different paths from the network. Several techniques exist to change the packet

header to influence ECMP and therefore find different paths (as shown in [Kab+14b]). However, it would be faster and cleaner to announce these paths explicitly to the endhosts whenever possible. Multipath protocols have a cost of testing a path that is much lower than for TCP. They can remember the performance obtained on a given path, and they can use all the paths at the same time for aggregating bandwidth, while single-path protocols have to choose only one of them.

Bibliography

[AFT07] B. Augustin, T. Friedman, and R. Teixeira. "Multipath tracing with Paris traceroute". In: *E2EMON 2007-5th IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*. IEEE. 2007, pp. 1–8.

[AFT10] B. Augustin, T. Friedman, and R. Teixeira. "Measuring multipath routing in the internet". In: *IEEE/ACM ToN* 19.3 (2010), pp. 830–840.

[Al-+19] R. Al-Saadi, G. Armitage, J. But, and P. Branch. "A survey of delay-based and hybrid TCP congestion control algorithms". In: *IEEE Communications Surveys & Tutorials* 21.4 (2019), pp. 3609–3638.

[Ali+14] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, et al. "CONGA: Distributed congestion-aware load balancing for datacenters". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. 2014, pp. 503–514.

[Ama+11] S. Amante, B. Carpenter, S. Jiang, and J. Rajahalme. *IPv6 Flow Label Specification*. RFC 6437. RFC Editor, Nov. 2011.

[AMO93] R. K. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[APB09] M. Allman, V. Paxson, and E. Blanton. *TCP Congestion Control*. RFC 5681. RFC Editor, Sept. 2009.

[Ara+14] J. T. Araújo, R. Landa, R. G. Clegg, and G. Pavlou. "Software-defined network support for transport resilience". In: *2014 IEEE NOMS*. IEEE. 2014, pp. 1–8.

[Aub+16] F. Aubry, D. Lebrun, S. Vissicchio, M. T. Khong, Y. Deville, and O. Bonaventure. "SCMon: Leveraging segment routing to improve network monitoring". In: *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*. 2016, pp. 1–9.

[Auk+00] P. Aukia, M. Kodialam, P. V. Koppol, T. Lakshman, H. Sarin, and B. Suter. "RATES: A server for MPLS traffic engineering". In: *IEEE Network* (2000).

[Awd+02] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao. *Overview and Principles of Internet Traffic Engineering*. RFC 3272. 2002.

[Bar+98] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. Savelsbergh, and P. H. Vance. "Branch-and-price: Column generation for solving huge integer programs". In: *Operations research* 46.3 (1998), pp. 316–329.

[BBC06] F. Baker, J. Babiarz, and K. H. Chan. *Configuration Guidelines for DiffServ Service Classes*. RFC 4594. Aug. 2006.

[BC12] S. Bubeck and N. Cesa-Bianchi. "Regret analysis of stochastic and nonstochastic multi-armed bandit problems". In: *arXiv preprint arXiv:1204.5721* (2012).

[BCS13] R. T. Braden, D. D. D. Clark, and S. Shenker. *Integrated Services in the Internet Architecture: an Overview*. RFC 1633. Mar. 2013.

[Ber+14] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. "ONOS: towards an open, distributed SDN OS". In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, pp. 1–6.

[Bha+15] R. Bhatia, F. Hao, M. Kodialam, and T. Lakshman. "Optimized network traffic engineering using segment routing." In: *INFOCOM*. IEEE, 2015, pp. 657–665.

[BM15] W. Braun and M. Menth. "Load-dependent flow splitting for traffic engineering in resilient OpenFlow networks". In: *2015 International Conference and Workshops on Networked Systems (NetSys)*. IEEE. 2015, pp. 1–5.

[BNC11] H. Babiker, I. Nikolova, and K. K. Chittimaneni. "Deploying IPv6 in the Google Enterprise Network. Lessons learned." In: *Proceedings of the 25th international conference on Large Installation System Administration, LISA*. 2011, p. 10.

[Bor+14] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. *TCP Extensions for High Performance*. RFC 7323. RFC Editor, Sept. 2014.

[Bra+13] B. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification*. RFC 2205. Mar. 2013.

[Bra17] L. Brakmo. "Tcp-bpf: Programmatically tuning tcp behavior through bpf". In: *NetDev 2.2* (2017).

[Brz+18] J. Brzozowski, J. Leddy, C. Filsfils, R. Maglione, and M. Townsley. *Use Cases for IPv6 Source Packet Routing in Networking (SPRING).* RFC 8354. 2018.

[C-A] C-Ares. https://c-ares.haxx.se.

[Cas+07] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. "Ethane: Taking Control of the Enterprise". In: *SIGCOMM '07*. Kyoto, Japan: ACM, 2007, pp. 1–12.

[CDG06] A. Conta, S. Deering, and M. Gupta. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification.* RFC 4443. RFC Editor, Mar. 2006.

[Che+21] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha. *The RACK-TLP Loss Detection Algorithm for TCP*. RFC 8985. RFC. Feb. 2021.

[Col+10] L. Colitti, S. H. Gunderson, E. Kline, and T. Refice. "Evaluating IPv6 adoption in the Internet". In: *Passive and active measurement*. Springer. 2010, pp. 141–150.

[Col+15] R. Coltun, D. Ferguson, J. Moy, and A. Lindem. *OSPF for IPv6.* RFC 5340. Oct. 2015.

[Col+16] L. Colitti, D. V. G. Cerf, S. Cheshire, and D. Schinazi. *Host Address Availability Recommendations.* RFC 7934. July 2016.

[Cor20] J. Corbet. *Kernel operations structures in BPF*. Feb. 2020.

[CRS16] M. Chiesa, G. Rétvári, and M. Schapira. "Lying Your Way to Better Traffic Engineering". In: *arXiv preprint arXiv:1610.02728* (2016).

[Czy+15] J. Czyz, M. Allman, J. Zhang, S. Iekel-Johnson, E. Osterweil, and M. Bailey. "Measuring IPv6 adoption". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 87–98.

[DAR81] DARPA. *Internet Protocol.* RFC 791. Sept. 1981.

[Dav+17] B. Davie, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. Gude, A. Padmanabhan, T. Petty, K. Duda, and A. Chanda. "A Database Approach to SDN Control Plane Design". In: *SIGCOMM Comput. Commun. Rev.* 47.1 (Jan. 2017), pp. 15–26.

[DB17] Q. De Coninck and O. Bonaventure. "Multipath quic: Design and evaluation". In: *Proceedings of the 13th international conference on emerging networking experiments and technologies*. 2017, pp. 160–166.

[DDS06] G. Desaulniers, J. Desrosiers, and M. M. Solomon. *Column generation*. Springer Science & Business Media, 2006.

[De +19] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure. "Pluginizing quic". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 59–74.

[Det+13] G. Detal, C. Paasch, S. Van Der Linden, P. Merindol, G. Avoine, and O. Bonaventure. "Revisiting flow-based load balancing: Stateless path selection in data center networks". In: *Computer Networks* 57.5 (2013), pp. 1204–1216.

[DG16] B. Davie and J. Gross. *A Stateless Transport Tunneling Protocol for Network Virtualization (STT)*. Internet-Draft draft-davie-stt-08. IETF Secretariat, Apr. 2016.

[DGV15] J. Damas, M. Graff, and P. Vixie. *Extension Mechanisms for DNS (EDNS(0))*. RFC 6891. Oct. 2015.

[DH17] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. STD 86. RFC Editor, July 2017.

[Dha+18] A. Dhamdhere, D. D. Clark, A. Gamero-Garrido, M. Luckie, R. K. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. C. Snoeren, and K. Claffy. "Inferring persistent interdomain congestion". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 1–15.

[Dic+16] J. Dickinson, S. Dickinson, R. Bellis, A. Mankin, and D. Wessels. *DNS Transport over TCP - Implementation Requirements*. RFC 7766. RFC Editor, Mar. 2016.

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[Dix+14] A. A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. "Elasticon: an elastic distributed sdn controller". In: *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM. 2014, pp. 17–28.

[DL05] J. Desrosiers and M. E. Lübbecke. "A Primer in Column Generation". In: *Column Generation*. Boston, MA: Springer US, 2005.

[Dun01] A. Dunkels. "Design and Implementation of the lwIP TCP/IP Stack". In: *Swedish Institute of Computer Science* 2.77 (2001).

[Dun06] A. Dunkels. "uIP-a TCP/IP stack for 8-and 16-bit microcontrollers". In: *Web page. Visited Nov* 13 (2006).

[Edd07] W. Eddy. *TCP SYN Flooding Attacks and Common Mitigations*. RFC 4987. RFC Editor, Aug. 2007.

[Egi+12] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp. "OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks". In: *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*. IEEE. 2012, pp. 1–8.

[Eri13] D. Erickson. "The beacon openflow controller". In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, pp. 13–18.

[FDC02] V. Firoiu, D. B. S. Davie, and A. Charny. *An Expedited Forwarding PHB (Per-Hop Behavior)*. RFC 3246. Mar. 2002.

[Fer+13] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. "Participatory Networking: An API for Application Control of SDNs". In: *SIGCOMM Comput. Commun. Rev.* 43.4 (Aug. 2013), pp. 327–338.

[Fil+18a] C. Filsfils, S. Previdi, B. Decraene, and R. Shakir. *Resiliency Use Cases in Source Packet Routing in Networking (SPRING) Networks*. RFC 8355. 2018.

[Fil+18b] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. *Segment Routing Architecture*. RFC 8402. 2018.

[Fil+20] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer. *IPv6 Segment Routing Header (SRH)*. RFC 8754. RFC Editor, Mar. 2020.

[Fil+21] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li. *Segment Routing over IPv6 (SRv6) Network Programming*. RFC 8986. RFC Editor, Feb. 2021.

[FKF06] S. Fischer, N. Kammenhuber, and A. Feldmann. "Replex: dynamic traffic engineering based on wardrop routing policies". In: *Proceedings of the 2006 ACM CoNEXT*. 2006, pp. 1–12.

[Flo+00] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. RFC 2883. RFC Editor, July 2000.

[For+13] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824 (Experimental). Internet Engineering Task Force, Jan. 2013.

[Fra+05] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure. "Achieving Sub-second IGP Convergence in Large IP Networks". In: *SIGCOMM Comput. Commun. Rev.* 35.3 (July 2005), pp. 35–44.

[FRT02] B. Fortz, J. Rexford, and M. Thorup. "Traffic engineering with traditional IP routing protocols". In: *IEEE communications Magazine* (2002).

[FRZ13] N. Feamster, J. Rexford, and E. Zegura. "The road to SDN". In: *Queue* 11.12 (2013), p. 20.

[FS11] K. R. Fall and W. R. Stevens. *TCP/IP illustrated, volume 1: The protocols*. Addison-Wesley, 2011.

[Fuk11] K. Fukuda. "An analysis of longitudinal TCP passive measurements (short paper)". In: *International Workshop on Traffic Monitoring and Analysis*. Springer. 2011, pp. 29–36.

[FVD17] R. Fanou, F. Valera, and A. Dhamdhere. "Investigating the Causes of Congestion on the African IXP substrate". In: *Proceedings of the 2017 Internet Measurement Conference*. 2017, pp. 57–63.

[Gas03] S. Gass. *Linear Programming: Methods and Applications*. Dover Books on Computer Science Series. Dover Publications, 2003.

[GHV17] S. Gay, R. Hartert, and S. Vissicchio. "Expect the unexpected: Sub-second optimization for segment routing". In: *IEEE INFOCOM*. 2017.

[Gia+15] S. Giacalone, D. Ward, J. Drake, A. Atlas, and S. Previdi. *OSPF Traffic Engineering (TE) Metric Extensions*. RFC 7471. Oct. 2015.

[GJN11] P. Gill, N. Jain, and N. Nagappan. "Understanding network failures in data centers: measurement, analysis, and implications". In: *Proceedings of the ACM SIGCOMM 2011*. 2011, pp. 350–361.

[Gre+16] H. Gredler, J. Medved, S. Previdi, A. Farrel, and S. Ray. *North-Bound Distribution of Link-State and Traffic Engineering (TE) Information Using BGP*. RFC 7752. Mar. 2016.

[Gre19] B. Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[GSV17] S. Gay, P. Schaus, and S. Vissicchio. "REPETITA: Repeatable Experiments for Performance Evaluation of Traffic-Engineering Algorithms". In: *arXiv preprint arXiv:1710.08665* (2017).

[Han+12] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. "Reproducible Network Experiments Using Container-based Emulation". In: *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. CoNEXT '12. Nice, France: ACM, 2012, pp. 253–264.

[Har+15a] R. Hartert, P. Schaus, S. Vissicchio, and O. Bonaventure. "Solving segment routing problems with hybrid constraint programming techniques". In: *CP*. Springer. 2015.

[Har+15b] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois. "A declarative and expressive approach to control forwarding paths in carrier-grade networks". In: *ACM SIGCOMM computer communication review* 45.4 (2015), pp. 15–28.

[Hay18] Y. Hayakawa. "eBPF Implementation for FreeBSD". In: *BSDCan 2018*. The BSD Conference, 2018.

[HLW15] A. Hari, T. Lakshman, and G. Wilfong. "Path switching: reduced-state flow handling in SDN using path information". In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM. 2015, p. 36.

[HM18] P. Hoffman and P. McManus. *DNS Queries over HTTPS (DoH)*. RFC 8484. RFC Editor, Oct. 2018.

[Hol+19] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever. "Blink: Fast connectivity recovery entirely in the data plane". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 161–176.

[Hol16] T. Hollmann. "A history of IPv6 challenges in Facebook data centers". Network @Scale conference. 2016.

[Hop00] C. Hopps. *Analysis of an Equal-Cost Multi-Path Algorithm*. RFC 2992. RFC Editor, Nov. 2000.

[Hu+16] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman. *Specification for DNS over Transport Layer Security (TLS)*. RFC 7858. May 2016.

[Hur+18] P. Hurtig, K.-J. Grinnemo, A. Brunstrom, S. Ferlin, Ö. Alay, and N. Kuhn. "Low-latency scheduling in MPTCP". In: *IEEE/ACM Transactions on Networking* 27.1 (2018), pp. 302–315.

[Int] Internet2. *Historical Abilene Data*. http://noc.net.internet2.edu/i2network/live-network-status/historical-abilene-data.html.

[ISO08] ISO/IEC. *Information technology – Intermediate System to Intermediate System intra-domain routeing*. ISO/IEC 10589:2002. Mar. 2008.

[IT21] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. RFC Editor, May 2021.

[Jac+09] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. "Networking named content". In: *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM. 2009, pp. 1–12.

[Jac88] V. Jacobson. "Congestion avoidance and control". In: *ACM SIGCOMM computer communication review* 18.4 (1988), pp. 314–329.

[Jad+20] M. Jadin, O. Tilmans, M. Mawait, and O. Bonaventure. "Educational Virtual Routing Labs with IPMininet". In: *ACM SIGCOMM Education Workshop*. 2020.

[Jai+13] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. "B4: Experience with a globally-deployed software defined WAN". In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 3–14.

[JBB92] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High Performance*. RFC 1323. RFC. May 1992.

[Jey+15] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. "Millions of little minions: Using packets for low latency network programming and visibility". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 3–14.

[Jun+02] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. "DNS Performance and the Effectiveness of Caching". In: *IEEE/ACM Transactions on networking* 10.5 (2002), pp. 589–603.

[Kab+14a] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks". In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 2014, pp. 149–160.

[Kab+14b] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks". In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM. 2014, pp. 149–160.

[Kat+17] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford. "Clove: Congestion-aware load balancing at the virtual edge". In: *Proceedings of the 13th International CoNEXT*. 2017.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-hashing for message authentication*. 1997.

[Kea17] M. Keane. *IPv6-only at Microsoft*. https://blog.apnic.net/2017/01/19/ipv6-only-at-microsoft/. Jan. 2017.

[Kim+15] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. "In-band network telemetry via programmable dataplanes". In: *ACM SIGCOMM*. Vol. 15. 2015.

[Kop+07] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. "A data-oriented (and beyond) network architecture". In: *ACM SIGCOMM Computer Communication Review*. Vol. 37. ACM. 2007, pp. 181–192.

[Kre+15] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76. ISSN: 0018-9219.

[Lan+21] R. Landa, L. Saino, L. Buytenhek, and J. T. Araújo. "Staying Alive: Connection Path Reselection at the Edge." In: *NSDI*. 2021, pp. 233–251.

[LB17] D. Lebrun and O. Bonaventure. "Implementing ipv6 segment routing in the linux kernel". In: *Proceedings of the Applied Networking Research Workshop*. 2017, pp. 35–41.

[LLC18] G. O. LLC. *Gurobi Optimizer Reference Manual*. 2018.

[LP16] P. Lapukhov and A. Premji. *Use of BGP for Routing in Large-Scale Data Centers*. RFC 7938. Aug. 2016.

[Mad+10] S. Madanapalli, J. Jeong, S. D. Park, and L. Beloeil. *IPv6 Router Advertisement Options for DNS Configuration*. RFC 6106. Nov. 2010.

[Mah+14] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. RFC Editor, Aug. 2014.

[Mal+04] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjálmtysson, and A. Greenberg. "Routing design in operational networks: A look from the inside". In: *SIGCOMM'04*. ACM. 2004, pp. 27–40.

[Mar+08] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot. "Characterization of failures in an operational IP backbone network". In: *IEEE/ACM transactions on networking* 16.4 (2008), pp. 749–762.

[Mar16] F. Martin. "IPv6 Inside LinkedIn Part II". https://engineering.linkedin.com/blog/2016/08/ipv6-inside-linkedin-part-ii–back-to-the-future. 2016.

[Mat+96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*. RFC. Oct. 1996.

[McK+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74.

[McK+96] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4 BSD operating system*. Vol. 2. Addison-Wesley Reading, MA, 1996.

[Moc87] P. Mockapetris. *Domain names - implementation and specification*. STD 13. RFC Editor, Nov. 1987.

[Moy13] J. T. Moy. *OSPF Version 2*. RFC 2328. Mar. 2013.

[Nad10] S. Nadas. *Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6*. RFC 5798. Mar. 2010.

[Nag84] J. Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. RFC Editor, Jan. 1984.

[NG16] M. Nikkhah and R. Guérin. "Migrating the Internet to IPv6: an exploration of the when and why". In: *IEEE/ACM Transactions on Networking* 24.4 (2016), pp. 2291–2304.

[PA00] V. Paxson and M. Allman. *Computing TCP's Retransmission Timer*. RFC 2988. RFC Editor, Nov. 2000.

[Pax+11] V. Paxson, M. Allman, J. Chu, and M. Sargent. *Computing TCP's Retransmission Timer*. RFC 6298. RFC Editor, June 2011.

[Pax97] V. Paxson. "End-to-end routing behavior in the Internet". In: *IEEE/ACM ToN* 5.5 (1997), pp. 601–615.

[PD13] B. Pfaff and B. Davie. *The Open vSwitch Database Management Protocol*. RFC 7047. Dec. 2013.

[Pel+13] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush. "From paris to tokyo: On the suitability of ping to measure latency". In: *Proceedings of the 2013 conference on Internet measurement conference*. 2013.

[Pos80] J. Postel. *User Datagram Protocol*. STD 6. RFC Editor, Aug. 1980.

[Pos81a] J. Postel. *Internet Control Message Protocol*. STD 5. RFC Editor, Sept. 1981.

[Pos81b] J. Postel. *Transmission Control Protocol*. STD 7. RFC Editor, Sept. 1981.

[Pre+17] S. Previdi, P. Psenak, C. Filsfils, H. Gredler, and M. Chen. "BGP Link-State extensions for Segment Routing". Internet draft, draft-ietf-idr-bgp-ls-segment-routing-ext-03, work in progress. July 2017.

[Pse+07] P. Psenak, S. Mirtorabi, A. Roy, L. Nguyen, and P. Pillay-Esnault. *Multi-Topology (MT) Routing in OSPF*. RFC 4915. RFC Editor, June 2007.

[Qaz+13] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. "SIMPLE-fying middlebox policy enforcement using SDN". In: *ACM SIGCOMM computer communication review* 43.4 (2013), pp. 27–38.

[Rai+11] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. "Improving datacenter performance and robustness with multipath TCP". In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 266–277.

[RB97] Y. Rekhter and J. Bound. *Dynamic Updates in the Domain Name System (DNS UPDATE)*. RFC 2136. Apr. 1997.

[Red+20] W. Reda, K. Bogdanov, A. Milolidakis, H. Ghasemirahni, M. Chiesa, G. Q. Maguire Jr, and D. Kostić. "Path persistence in the cloud: A study of the effects of inter-region traffic engineering in a large cloud provider's network". In: *ACM SIGCOMM Computer Communication Review* 50.2 (2020), pp. 11–23.

[RFB01] K. Ramakrishnan, S. Floyd, and D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. RFC Editor, Sept. 2001.

[RHB18] A. Rabitsch, P. Hurtig, and A. Brunstrom. "A stream-aware multipath quic scheduler for heterogeneous paths". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. 2018, pp. 29–35.

[RLH06] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. RFC Editor, Jan. 2006.

[Ros+02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. *SIP: Session Initiation Protocol*. RFC 3261. RFC Editor, June 2002.

[Rou05] M. Roughan. "Simplifying the synthesis of Internet traffic matrices". In: *ACM SIGCOMM CCR* (2005).

[Sai11] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. RFC 6120. Mar. 2011.

[Sal+06] S. Salsano, A. Botta, P. Iovanna, M. Intermite, and A. Polidoro. "Traffic engineering with OSPF-TE and RSVP-TE: Flooding reduction techniques and evaluation of processing cost". In: *Computer Communications* 29.11 (2006), pp. 2034–2045.

[SAO13] V. Singh, S. Ahsan, and J. Ott. "MPRTP: Multipath Considerations for Real-time Media". In: *Proceedings of the 4th ACM Multimedia Systems Conference*. MMSys '13. Oslo, Norway: ACM, 2013, pp. 190–201.

[Sar] S. Sarwood. "OVH data centre destroyed by fire in Strasbourg – all services unavailable". The Register, March 10, 2021.

[Sch+03] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. RFC Editor, July 2003.

[SD21] J. Sommers and R. Durairajan. "ELF: High-Performance In-band Network Measurement". In: *Proc. IFIP TMA* (2021).

[Sha+13] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. "Advanced study of SDN/OpenFlow controllers". In: *Proceedings of the 9th central & eastern european software engineering conference in russia*. ACM. 2013, p. 1.

[She+12] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. "Making middleboxes someone else's problem: network processing as a cloud service". In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 13–24.

[Sim+07] W. A. Simpson, D. T. Narten, E. Nordmark, and H. Soliman. *Neighbor Discovery for IP version 6 (IPv6)*. RFC 4861. Sept. 2007.

[SMW02] N. Spring, R. Mahajan, and D. Wetherall. "Measuring ISP topologies with Rocketfuel". In: *ACM SIGCOMM CCR* (2002).

[Sun+11] Y.-W. E. Sung, X. Sun, S. G. Rao, G. G. Xie, and D. A. Maltz. "Towards systematic design of enterprise networks". In: *IEEE/ACM Transactions on Networking (TON)* 19.3 (2011), pp. 695–708.

[Tan17] J. Tantsura. "The critical role of Maximum SID Depth (MSD) hardware limitations in Segment Routing ecosystem and how to work around those". In: *NANOG71*. 2017.

[TB19] V.-H. Tran and O. Bonaventure. "Beyond socket options: making the Linux TCP stack truly extensible". In: *2019 IFIP Networking Conference (IFIP Networking)*. IEEE. 2019, pp. 1–9.

[TB20] V.-H. Tran and O. Bonaventure. "Beyond socket options: Towards fully extensible Linux transport stacks". In: *Computer Communications* 162 (2020), pp. 118–138.

[TG21] D. Thaler and P. Gaddehosur. *Making eBPF work on Windows*. `https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/`. May 2021.

[TMB10] J. Touch, A. Mankin, and R. Bonica. *The TCP Authentication Option*. RFC 5925. RFC Editor, June 2010.

[Too+12] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. "On Controller Performance in Software-Defined Networks." In: *Hot-ICE* 12 (2012), pp. 1–6.

[Tre+20] M. Trevisan, D. Giordano, I. Drago, M. M. Munafò, and M. Mellia. "Five years at the edge: Watching internet from the ISP network". In: *IEEE/ACM ToN* 28.2 (2020), pp. 561–574.

[Tri+17] G. Trimponias, Y. Xiao, H. Xu, X. Wu, and Y. Geng. "On traffic engineering with segment routing in sdn based wans". In: *arXiv preprint arXiv:1703.05907* (2017).

[VBW15] S. S. Villar, J. Bowden, and J. Wason. "Multi-armed bandit models for the optimal design of clinical trials: benefits and challenges". In: *Statistical science: a review journal of the Institute of Mathematical Statistics* 30.2 (2015), p. 199.

[Wal+16] D. Walton, A. Retana, E. Chen, and J. Scudder. *Advertisement of Multiple Paths in BGP*. RFC 7911. RFC. July 2016.

[Wan+07] L. Wang, M. Saranu, J. M. Gottlieb, and D. Pei. "Understanding BGP session failures in a large ISP". In: *IEEE INFOCOM 2007*. IEEE. 2007.

[Wan+08] N. Wang, K. H. Ho, G. Pavlou, and M. Howarth. "An overview of routing optimization for internet traffic engineering". In: *IEEE Communications Surveys & Tutorials* (2008).

[Wan+13] Z. Wang, M. A. Carlson, W. Weiss, E. B. Davies, and S. L. Blake. *An Architecture for Differentiated Services*. RFC 2475. Mar. 2013.

[Wei+99] W. Weiss, D. J. Heinanen, F. Baker, and J. T. Wroclawski. *Assured Forwarding PHB Group*. RFC 2597. June 1999.

[Wel00] B. Wellington. *Secure Domain Name System (DNS) Dynamic Update*. RFC 3007. Nov. 2000.

[WJL03] D. Watson, F. Jahanian, and C. Labovitz. "Experiences with monitoring OSPF on a regional service provider network". In: *23rd International Conference on Distributed Computing Systems, 2003. Proceedings*. IEEE. 2003, pp. 204–213.

[Wu+13] P. Wu, Y. Cui, J. Wu, J. Liu, and C. Metz. "Transition from IPv4 to IPv6: A state-of-the-art survey". In: *Communications Surveys & Tutorials, IEEE* 15.3 (2013), pp. 1407–1424.

[Xia+00] X. Xiao, A. Hannan, B. Bailey, and L. M. Ni. "Traffic Engineering with MPLS in the Internet". In: *IEEE network* 14.2 (2000), pp. 28–33.

[Xia+15] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie. "A survey on software-defined networking". In: *IEEE Communications Surveys & Tutorials* 17.1 (2015), pp. 27–51.

[Zha+13] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan, et al. "Steering: A software-defined networking for inline service chaining". In: *IEEE ICNP*. IEEE. 2013.