Reaping the Benefits of IPv6 Segment Routing

David Lebrun

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

September 2017

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Olivier **Bonaventure** (Advisor)	UCLouvain, Belgium
Pr. Thomas **Clausen**	École polytechnique, France
Pr. Charles **Pecheur** (Chair)	UCLouvain, Belgium
Pr. Ramin **Sadre**	UCLouvain, Belgium
Pr. Stefano **Salsano**	Università degli Studi di Roma Tor Vergata, Italy

Reaping the Benefits of IPv6 Segment Routing

by David Lebrun

ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was partially supported by the ARC-SDN project funded by Communauté française de Belgique and by a grant from Cisco.

Meaning lies as much
in the mind of the reader
as in the Haiku.

— Douglas R. Hofstadter,
Gödel, Escher, Bach: An Eternal Golden Braid

Preamble

Since the early days of the Internet, the size of networks has grown by several orders of magnitude. Today, large networks may span an entire continent, and the largest of them operate on a planetary scale. Networks can be coarsely classified in a handful of categories, such as datacenter, enterprise, service providers, content delivery networks, etc. Each type of network yields different constraints, and each network instance brings its own set of requirements.

A few core principles underlie this plurality of requirements. For example, fault tolerance and scalability are essential properties of many networks. Efficient traffic engineering capabilities are also a common requirement. To meet operational expectations and fulfill business contracts, network operators must be able to control the traffic flows in a predictable and reliable manner.

Existing traffic engineering techniques address those problems in various ways. MPLS-based solutions are deployed in many ISP networks, but bear the consequences of design decisions made when networks were smaller. A major issue of those solutions is poor scalability. OpenFlow-based solutions propose paradigm shifting principles that leverage a logically centralized controller. While providing improved network management features, those solutions prove difficult to be deployed due to their fundamentally different nature.

The Segment Routing architecture emerged to address the operational issues of MPLS-based traffic engineering solutions. By leveraging the source routing paradigm, it provides fine-grained flow management capabilities without adding state in the core network. It relies on existing routing protocols to distribute reachability information. As such, it is more robust and easier to deploy than OpenFlow-based solutions.

The purpose of this thesis is to explore and reap the benefits of the IPv6 flavor of Segment Routing (SRv6). Its unique properties open the doors to a large, yet mostly unexplored, research area that can extend network management and traffic control up to the endhosts. The main contributions of this thesis are the following.

- *A reference open-source implementation of SRv6 in the mainline Linux kernel.*

  An open-source implementation of software systems increases their visibility and enables other researchers to use and explore them. Furthermore, the integration of a particular feature within the mainline Linux kernel ensures that such a feature will enjoy widespread availability, as the various Linux distributions integrate the recent kernel versions. In Chapter 3, we describe our Linux kernel implementation of SRv6, available in the mainline kernel since version 4.10. We detail our support for the SRv6 data plane and control plane, as well as the implementation of the HMAC security mechanism. We describe our custom-made testing environment that leverages the network namespaces feature of the Linux kernel and discuss the limits of such an environment. We evaluate the performance of our implementation on real hardware and show that it yields little overhead with respect to regular IPv6 forwarding, and is able to scale with the available CPU resources.

- *An exploration of SRv6 in its ability to solve various networking issues.*

  In Chapter 4, we explore the abilities of SRv6 in two aspects of networking. The first aspect is the support of low-latency real-time services. We show that by duplicating traffic across disjoint paths, SRv6 is able to significantly absorb the adverse effects of unexpected packet loss or jitter. Simulations in a virtualized network show that the Linux TCP stack is able to cope with the duplicated traffic without hurting performances. The second explored aspect is network monitoring. By leveraging the unique properties of Segment Routing, we propose and implement SCMon, a new network monitoring technique. SCMon computes a set of cycles covering the network and sends probes over these cycles. Our technique is able to deterministically explore the Equal-Cost Multi-Path components of a network as well as individual links in a bundle, while operating from a single vantage point. We implement a prototype of this solution and evaluate it in different emulated networks. The results show that SCMon is able to quickly and efficiently detect single-link failures.

- *The design, implementation and validation of Software Resolved Networks, a new architecture for IPv6 enterprise networks.*

  In the first part of Chapter 5, we propose *Software Resolved Networks* (SRNs). Designed for IPv6 enterprise networks, an SRN provides traffic

engineering capabilities through SRv6 and leverages a central controller. A particular aspect of SRNs is that the controller can interact with the applications through the DNS protocol. Such interactions enable the applications to participate in the management of their flows by providing hints about the nature and needs of their communications. We present the properties of enterprise networks and the building blocks necessary to run a Software Resolved Network. We describe how such an architecture can provide traffic management capabilities and present algorithms to realize this. We detail all the control plane components of an SRN and discuss its fault tolerance properties as well as various security implications.

- *An extensive implementation and evaluation of the control plane components running in Software Resolved Networks.*

  In the second part of Chapter 5, we provide a full implementation of the control plane components needed to operate an SRN. We discuss the changes needed to our SRv6 kernel implementation and thoroughly present the architecture and implementation of the SRN controller. We evaluate the performance of our implementation through benchmarks and simulations. We show that our solution meets the performance expectations of large enterprise networks.

Finally, even if not a scientific contribution *per se*, we would like to stress the fact that all the programs and tools that have been developed in the course of this thesis are open-source and freely available. Furthermore, we also provide the raw data that have been produced by the various experiments of this work, as well as the scripts that were developed and used to generate the plots visible in this document. [1]

Bibliographic notes

Conference publications

1. *Traffic duplication through segmentable disjoint paths*
   F. Aubry, D. Lebrun, Y. Deville and O. Bonaventure. IFIP Networking, 2015.

2. *SCMon: Leveraging Segment Routing to Improve Network Monitoring*
   F. Aubry, D. Lebrun, S. Vissicchio, M. Khong, Y. Deville and O. Bonaventure. IEEE INFOCOM, 2016.

[1] See https://github.com/target0/thesis-data

3. *Implementing IPv6 Segment Routing in the Linux Kernel*
   D. Lebrun and O. Bonaventure. ACM/ISOC ANRW, 2017.

4. *Software Resolved Networks: Rethinking Enterprise Networks with IPv6 Segment Routing*
   D. Lebrun, M. Jadin, F. Clad, C. Filsfils and O. Bonaventure. ACM CoNEXT, 2017 (submitted).

Posters and demos

1. *Leveraging IPv6 Segment Routing for Service Function Chaining*
   D. Lebrun. ACM CoNEXT student workshop, 2015.

2. *A Linux Kernel Implementation of Segment Routing with IPv6*
   D. Lebrun. IEEE INFOCOM student workshop, 2016.

3. *Demo: IPv6 Segment Routing to the End Host: A Linux Kernel Implementation*
   D. Lebrun. ACM SOSR, 2017.

IETF contributions

1. *IPv6 Segment Routing Header (SRH)*
   S. Previdi, C. Filsfils *et al.*. IETF Internet-Draft draft-ietf-6man-segment-routing-header-06, 2017.

2. *Insertion of IPv6 Segment Routing Headers in a Controlled Domain*
   D. Voyer, S. Previdi *et al.* IETF Internet-Draft draft-voyer-6man-extension-header-insertion-00, 2017.

3. *SRv6 Network Programming*
   P. Camarillo, C. Filsfils *et al.*. IETF Internet-Draft draft-filsfils-spring-srv6-network-programming-00, 2017.

Reading IETF draft names

All IETF draft names begin with `draft-`*`name`*`-`*`wg`*`-`. The *`wg`* part is the name of the working group relevant for the draft. When *`name`* is the last name of the draft's main editor, it means that the draft is in an early stage and not yet adopted by its working group. Instead, if *`name`* is equal to `ietf`, then the draft is adopted by its working group and will likely be promoted to RFC status once it reaches a sufficient level of maturity and stability.

Miscellaneous contributions

1. *Segment Routing: IPv6, Implementation and a Practical Use Case*
   D. Lebrun. RIPE 70, Amsterdam, 2015.

2. *Implementing IPv6 Segment Routing*
   D. Lebrun. Netdev 1.2, Tokyo, 2016.

3. *Reaping the Benefits of IPv6 Segment Routing*
   D. Lebrun. IIJ Innovation Institute seminar, Tokyo, 2016.

4. *IPv6 Segment Routing*
   D. Lebrun. LWN.net, 2017.

Acknowledgments

First and foremost, I would like to acknowledge my advisor, Olivier Bonaventure. He introduced me to my broad thesis topic, Segment Routing, when it was still in its infancy. He continuously presented me with challenging ideas, research topics, and goals to achieve. My successful journey through these challenges is greatly due to his advices, availability and support. The experience I built up during my time as a PhD student has shaped and sharpened my mind in a way that, I dare to hope, will positively and fundamentally influence my future career.

I would also like to thank my thesis jury, Thomas Clausen, Charles Pecheur, Ramin Sadre, and Stefano Salsano, for their insightful comments and the very interesting and in-depth discussions we had during my private defense.

For the work we jointly realised, I would like to thank my co-authors, François Aubry, François Clad, Yves Deville, Clarence Filsfils, Mathieu Jadin, Stefano Previdi, Stefano Vissicchio, and Eric Vyncke. I am also grateful to Daniel Bernier and Daniel Voyer for their invitation and hospitality in Bell Canada, Montréal. Special thanks go to Tazaki Hajime, who honoured and invited me as lecturer to the IIJ Innovation Institute seminar in Tokyo. It is worth mentioning that my time in IIJ-II during my master degree internship most certainly influenced my decision of pursuing a PhD degree.

For all the fun times we shared together, I am grateful to my former INL colleagues I appreciated most, Quentin de Coninck, Fabien Duchêne, Benjamin Hesmans, Christoph Paasch, and Olivier Tilmans.

Last but not least, I express my deep gratitude to all my friends and family who supported me during these past few years. A special mention goes to Charlotte, who helped me keep focus and accompanied me throughout the good and bad times.

Contents

Chapter 1

Introduction

Computer networks connect terminal nodes, or endhosts, and enable them to communicate by exchanging data. In IP networks, network nodes are assigned a unique, network-wide identifier. This identifier is called an *IP address*. The information transiting on IP networks is chunked into *packets*. Each packet contains several layers of information. The topmost layer, also known as the IP layer, provides basic information about the packet itself, such as the source and destination IP addresses and the length of the packet. Such information tells intermediate routers *where* the path of a given packet must eventually end, but not *how* to implement this path. Indeed, it is likely that multiple paths exist between a given source and a destination. Thus, intermediate routers need additional information to properly forward packets.

In this introduction chapter, we briefly present the fundamental concepts of packet forwarding deployed in current networks. Then, we describe the limits of classical traffic engineering techniques. We introduce how the Segment Routing architecture helps to overcome those limits and to provide better network management by leveraging the source routing paradigm. We briefly present the current trends of shifting network control towards a logically centralized controller. Finally, we introduce the IPv6 flavor of Segment Routing and how it open the doors to novel network management principles and research opportunities.

1.1 Networking principles and protocols

Classical IP networks [1, 2] implement a hop-by-hop, shortest-path, destination-based forwarding. Each network node contains a set of *routes*. Each of them maps an IP prefix (*i.e.*, a set of contiguous IP addresses) to a *nexthop*. A nexthop is another network node, directly attached to the current one (either virtually or physically). A route thus states which neighbor to use in order to reach

a given prefix. A *metric* is attached to each route. This metric represents a cost of using this particular route. Usually, it is a function of the topological distance from the current node to the node holding the destination prefix. In practice, route metrics are used to implement shortest-path forwarding. When a router receives a packet to be forwarded, it extracts the destination IP address of the packet and performs a lookup in its routing table. This lookup attempts to find a route whose destination prefix contains the destination IP address of the packet. If several routes match the destination, then the most precise route is selected, *i.e.*, the route whose prefix is longer. This operation is called the *longest prefix match*. If several routes with the same prefix length match the destination, then the route of least cost is selected. The packet is then forwarded to the nexthop attached to the route. If several routes with the same cost match the destination, then routers usually implement Equal-Cost Multi-Path (ECMP) [3]. The idea of ECMP is to apply a *hashing function* to some of the packet's headers and to select a route according to the result of this hashing function. The effect is that packets matching ECMP routes will be load-balanced across the available nexthops.

To fulfill their packet forwarding duties, routers must learn the routes needed to reach each other node in a given network. Such distribution of reachability information can be achieved in several ways. In the early days of the Internet, the number of nodes was so small that the routes could be configured statically. Whenever a new node or a new prefix would appear, the operators needed to manually reconfigure their routers to reflect the change. Obviously, such network management principles cannot withstand the scale of current networks. Routing protocols were developed to automatically distribute routing information between the routers of a network. On the one hand, distance-vector routing protocols such as RIP [4] propagate vectors of distances to other nodes in the network. Those vectors contain information very similar to the elementary routing tables described earlier. The difference is that vectors comprise only the routes of least cost. On the other hand, link-state protocols such as OSPF [5, 6] and IS-IS [7] distribute connectivity information among routers (*i.e.*, the state of their links). Using that information, each router constructs an internal graph of the network. Then, they apply a shortest path computation algorithm on the graph, such as Dijkstra. The resulting set of best (*i.e.*, shortest) paths forms their routing table. Those protocols (RIP, OSPF, IS-IS) are used to distribute reachability information within a single autonomous system. Hence, they are referred to as *Interior Gateway Protocols* (IGPs). The exchange of routing information between distinct autonomous systems is realized through the *Border Gateway Protocol* (BGP) [8]. BGP is considered as a path-vector routing protocol. It is conceptually close to distance-vector protocols. However, additional parameters are used in the best route selection process. For example, it is possible to define a *local preference* value that takes precedence over the announced topological distance.

Many networks need to implement traffic engineering to fulfill Service Level Agreements (SLAs). For example, a service provider may provide a low-latency path between two customer sites. Such a low-latency path is not necessarily the shortest one. The operator may tweak the link metrics to steer the traffic through the desired path. However, such tweaking will affect the entire network, while the intention was to specifically target the low-latency traffic. The *MultiProtocol Label Switching* protocol (MPLS) [9, 10] was designed to enable such per-flow traffic engineering. To realize this, MPLS implements virtual circuits that are reminiscent of ATM networks. The core idea of MPLS is to leverage a stack of 20-bit *labels* pushed on top of the packets' IP header. Each label represents a particular virtual circuit, or *label-switched path* (LSP). When an MPLS router receives a packet with an MPLS label stack, it examines the topmost label and performs an operation based on the contents of the label. This operation can be a *swap* (replace the topmost label by another label), *push* (add a new label on top of the stack) or *pop* (remove the topmost label from the stack). The packet is then forwarded to the appropriate neighbor. To distribute label reachability information, MPLS routers typically establish full-mesh sessions using the *Label Distribution Protocol* (LDP) [11]. LDP peers then build their label tables to implement LSPs. The RSVP-TE [12, 13] protocol enables IP routers to request the reservation of network resources. This reservation is carried by establishing LSPs constrained by the bandwidth, latency, or other parameter requested in the resource reservation. As a result, network operators are able to implement per-flow traffic engineering without modifying the IGP topology. Another gain is the ability to provide node and link protection by computing additional, backup LSPs to protect existing circuits. Upon failure, the routers can quickly switch to such a pre-computed backup path. This operation, faster than waiting for IGP convergence, is called *Fast Reroute* (FRR) and is expected to complete within 50 milliseconds of a failure.

Despite providing added-value services, the traffic steering model of MPLS exhibits several limitations [14, 15]. First, it has scalability issues. Indeed, each resource reservation creates state on all the routers part of the virtual circuit. Second, it fails to leverage ECMP routes available in a network. A workaround is to replicate circuits, but this introduces even more state in the routers. Third, the distributed nature of the routing protocols makes the flow management operations prone to induce transient states and unexpected side effects.

1.2 Source routing paradigm

In traditional networks, packet forwarding is performed in a hop-by-hop basis, *i.e.*, each intermediate router decides where to forward the packets. The source

routing paradigm proposes an alternative method, where the source of the packets may specify either partially or entirely the path packets must follow to reach their destination. The IPv4 protocol provides two header options specifying a loose or strict path for the packets. However, these options are generally blocked on the Internet due to security concerns. In practice, IPv4 source routing capabilities are seldom used in any network. The original specifications of IPv6 [2] also define a source routing header, the *Routing Header Type 0* (RH0). RH0 was later deprecated [16] due to similar security concerns.

The Segment Routing (SR) architecture [17, 15, 18] was originally proposed to improve the MPLS architecture by leveraging the source routing paradigm. Instead of using lookup tables to know which label must replace the current one, SR proposes to fully specify the path of the packets in the label stack. This specification is performed by the tunnel ingress, *i.e.*, the node that imposes an MPLS header on IP packets. Each label then refers to a particular network node or link that must be traversed by the packet. If two consecutive labels represent network nodes that are not direct neighbors, then regular shortest-path routing is used to transport packets from one label to the next. The packets are thus forwarded through shortest-path *segments*.

Such an architecture yields multiple benefits and addresses the main concerns of the MPLS traffic steering model. First, specifying the path of a packet in the packet itself has the direct effect of reducing the state to maintain in a network. Indeed, core routers processing only MPLS packets do not need to keep additional state besides reachability information. All per-flow state is moved to the edge of the network, *i.e.*, in the ingress routers that impose the MPLS label stack onto packets. Second, the SR architecture provisions *anycast* segments to leverage built-in ECMP paths. Third, thanks to the source routing paradigm, a packet's path can be entirely specified at single point of entry, instead of being distributed across multiple routers. These properties enable (i) a better scaling of the network, (ii) a more efficient utilisation of the network resources, and (iii) improved network management and troubleshooting operations. Furthermore, the Segment Routing architecture is intended to operate within the boundaries of a single autonomous system. As such, the security concerns that prevented the rise of other source routing implementations do not apply to SR.

In parallel, the Segment Routing architecture was also developed for the IPv6 dataplane [19] (SRv6). The SRv6 architecture extends and defines a new type of IPv6 Routing Header, namely the *Segment Routing Header* (SRH). One of the main benefits of SRv6 is that, unlike MPLS, IPv6 can be easily deployed up to the endhosts and supported by a wide range of applications. In SRv6, a segment is represented by an IPv6 address. The SRH specifies the list of segments (IPv6 addresses) that implements a particular path. A segment can represent various topological and service-based instructions. When entering an SRv6 path, instead

of receiving an MPLS label stack, the packets would simply receive a Segment Routing Header, either through encapsulation within an outer IPv6 header, or with the direct insertion of the SRH between the original IPv6 header and the rest of the packet. As SRv6 is simply an extension of IPv6 and does not bring brand new protocols, no additional signalling protocols are required in the network, besides the existing IGP.

1.3 Decoupling the control plane

Recently, a new paradigm has gained considerable momentum in networking vendors, operators and academia. Known as *Software-Defined Networking* (SDN), this paradigm first appeared as OpenFlow [20, 21]. The core idea of SDN is to decouple the control plane from the forwarding plane and to move network control into a logically centralized controller. This architecture enables to program the network through software and various implementations have been proposed [22]. OpenFlow, the first SDN implementation, gets rid of all distributed routing protocols and delegates forwarding decisions to a central controller. OpenFlow works on a switch level rather than a router level. Each OpenFlow switch maintains a flow table, as well as a control channel towards the controller. When the switch receives a packet and cannot find a matching entry in its flow table, it forwards the packet to the controller. Then, the controller makes a decision on how to handle the packet, inserts the appropriate flow entry into the switch and, if necessary, reinjects the packet into the switch forwarding path.

The SDN paradigm provides a large amount of flexibility through network programmability and centralized control. However, its OpenFlow implementation has drawbacks. For example, it creates per-flow state in all the switches participating in a flow's path. Fault tolerance is also an issue as reactions to, *e.g.*, link failures, require a specific action from the controller. Shifting network control away from distributed protocols towards a central controller improves flexibility but also reduces tolerance to failures by creating a Single Point Of Failure (SPOF).

Nevertheless, Software-Defined Networking is now an unavoidable cornerstone of modern network design.

1.4 Conclusion

Segment Routing is a powerful architecture enabling to steer packets through an arbitrary set of topological or service-based policies. It realizes this by leveraging the pre-existing IGP infrastructure and without adding state in core routers. In Chapter 2, we describe in details the Segment Routing architecture as well as

SRv6, its IPv6 flavor. In Chapter 3, we thoroughly explain our implementation of SRv6 in the Linux kernel and evaluate its performance on real hardware. In Chapter 4, we explore how SRv6 can help network operators to (i) provide reliable low-latency paths for real-time applications and to (ii) efficiently implement full-coverage monitoring in large networks, from a single vantage point. In Chapter 5, we design a novel architecture for IPv6 enterprise networks and present how SRv6 can be integrated with the SDN model. We also implement and evaluate an SDN-like controller for SRv6. Finally, we conclude in Chapter 6.

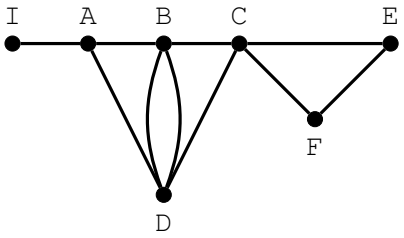
Chapter 2

Segment Routing

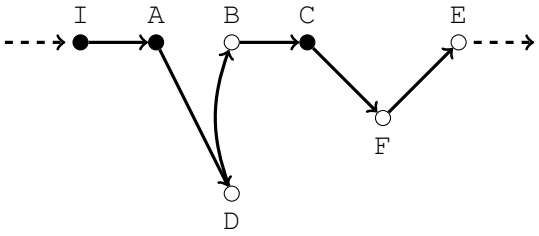
Segment Routing (SR) is a source routing paradigm. It originated from the Source Packet Routing in Networking (SPRING) IETF working group, created in 2013 [23, 17, 15]. The core idea of SR is to enable packet flows to leverage arbitrary paths in a network. These paths are defined by the source of the packets and are not necessarily the shortest ones. The source can be either the node that actually generated the packets, or any ingress node on the path of the packets. All the information needed to transport the packet along its specified path is encoded within the packet itself. This approach enables the intermediate nodes to be *stateless* with respect to the path of flows, at the cost of a small per-packet overhead. In this chapter, we explain the Segment Routing architecture in more details. First, we provide an abstract model of SR and illustrate its topological properties. Then, we present the IPv6 flavor of Segment Routing through its extension to the IPv6 protocol. We describe the basic operations of SRv6. Finally, we present the network programming concept introduced by the recent specifications of SRv6.

The path-related information is encoded in the packet as a sequence of shortest paths. Each element of the sequence (*i.e.*, each shortest path) is represented by a *segment*. These segments form an ordered *list of segments*. The terms *segment* and *segment identifier* (SID) can be used interchangeably. We define two main types of segments representing topological instructions. A *node* segment (or *Node-SID*) steers the packet through a particular network node. An *adjacency* segment (or *Adj-SID*) steers the packet through a particular link. In other words, an adjacency segment implies sending on an outgoing link, while a node segment implies a node traversal. Along with the list of segments is also encoded a pointer. This pointer indexes the current active segment. The corresponding network node that receives and processes SR-enabled packets is called a *segment endpoint*. The node that imposes a list of segments into packets is called an *SR ingress node*. This node can be a router as well as an endhost. Likewise, the node that removes the list of segments from packets is called an *SR egress node*. Typically, this node is

also the last segment of the path. The set of nodes that actively participate in the processing of SR-enabled packets is called an SR domain. This domain can be, *e.g.*, an entire Autonomous System or a subset of a larger network.



(a) Segment Routing domain.



(b) Forwarding path for segments `node(D)`, `adj(D, B, left)`, `node(F)`, `node(E)` imposed by SR ingress node `I`. Hollow nodes are segment endpoints.

Figure 2.1: Illustration of Segment Routing topological instructions.

Let us consider Figure 2.1a. It represents an SR domain composed of an SR ingress node `I`, an SR egress node `E`, and several intermediate nodes. Packets enter the network at `I` and the following list of segments is inserted: `node(D)`, `adj(D, B, left)`, `node(F)`, `node(E)`. The active segment pointer is initialized to the first segment of the list. Figure 2.1b shows the path followed by the packets within the SR domain. The first segment is `node(D)`. Thus, the packets follow the shortest path from `I` to `D`. The segment pointer is updated to the next segment, which is an adjacency segment. It instructs to follow a particular link between node `D` and `B` (the left link on the figure). The packets are then processed by `B` and forwarded over the shortest path to the next segment, which is node `F`. The node processes the packets and forwards them to the last segment, node `E`. This last node removes the list of segments from the packets and forwards them to their original destination.

Segment Routing was originally designed for the MPLS dataplane [24]. By using MPLS labels as segments, SR-MPLS does not require any dataplane modification. Several control plane protocols have been modified [25, 26] to support the signalling of SR labels. Inherently, SR-MPLS is intended to be used mainly in ISP networks. The IPv6 version of Segment Routing (SRv6) extends the use

of SR to environments where MPLS is not available. Furthermore, SRv6 enables to directly interact with endhosts. In this thesis, we focus on the IPv6 version of Segment Routing and explore its potential.

2.1 IPv6 Segment Routing

The IPv6 flavor of Segment Routing is implemented using an IPv6 extension header, the Routing Header [2]. The Routing Header (RH) is defined in the IPv6 specifications as a 4-byte header template and contains a Routing Type field, enabling the instantiation of RH subtypes. The IPv6 specifications originally defined RH type 0 (RH0) which provided rudimentary source routing capabilities [2]. RH0 was later deprecated [16] due to security concerns. The IPv6 Segment Routing Header (SRH) is defined as a new RH of type 4. It contains a list of segments encoded as IPv6 addresses. The SRH may also contain multiple optional Type-Length Values (TLVs). One such TLV is the HMAC TLV, which provides authenticity and integrity checks for the SRH. It addresses security concerns that led to the deprecation of the RH0. In this section, we present the SRH as specified by the version 5 of the IETF draft. At the time of writing, the version 6 has been released. However, the differences between the two versions are mostly technical precisions and synchronisation with other, more recent specifications of principles associated with SRv6. Those principles are further discussed in Section 2.2.

Figure 2.2 shows the Segment Routing Header wire format. The first four bytes are part of the generic Routing Header. The `Next Header` field encodes the type of the header immediately following the SRH (*e.g.*, another extension header, a transport protocol such as TCP or UDP, etc.). The `Hdr Ext Len` field encodes the length of the SRH as an 8-byte multiple. The `Routing Type` field is set to 4 [19]. The `Segments Left` field indicates the number of remaining segments. It is an index into the list of segments subsequently defined. The following four bytes are specific to the SRH. The `First Segment` field indexes into the list of segments the first segment of the path. The `Flags` field defines various flags. One such flag is the H-flag which is set to 1 when an HMAC TLV is present. The remaining two bytes are reserved for a future use. Then, the list of segments is encoded as 128-bit IPv6 addresses, in reverse order. The first segment of the list (at index 0) is the last segment of the path. The last segment of the list (indexed by the `First Segment` field) is the first segment of the path.

Multiple TLVs can be defined for an SRH. The HMAC TLV, as shown in Figure 2.3 has the role of ensuring the authenticity and integrity of the SRH. It contains a 32-bit `HMAC Key ID` field and a 256-bit `HMAC` field. The `HMAC Key ID` field is a non-zero, opaque, operator-defined value. It maps to at least a secret and a hashing algorithm. The input text of the HMAC function is the concate-

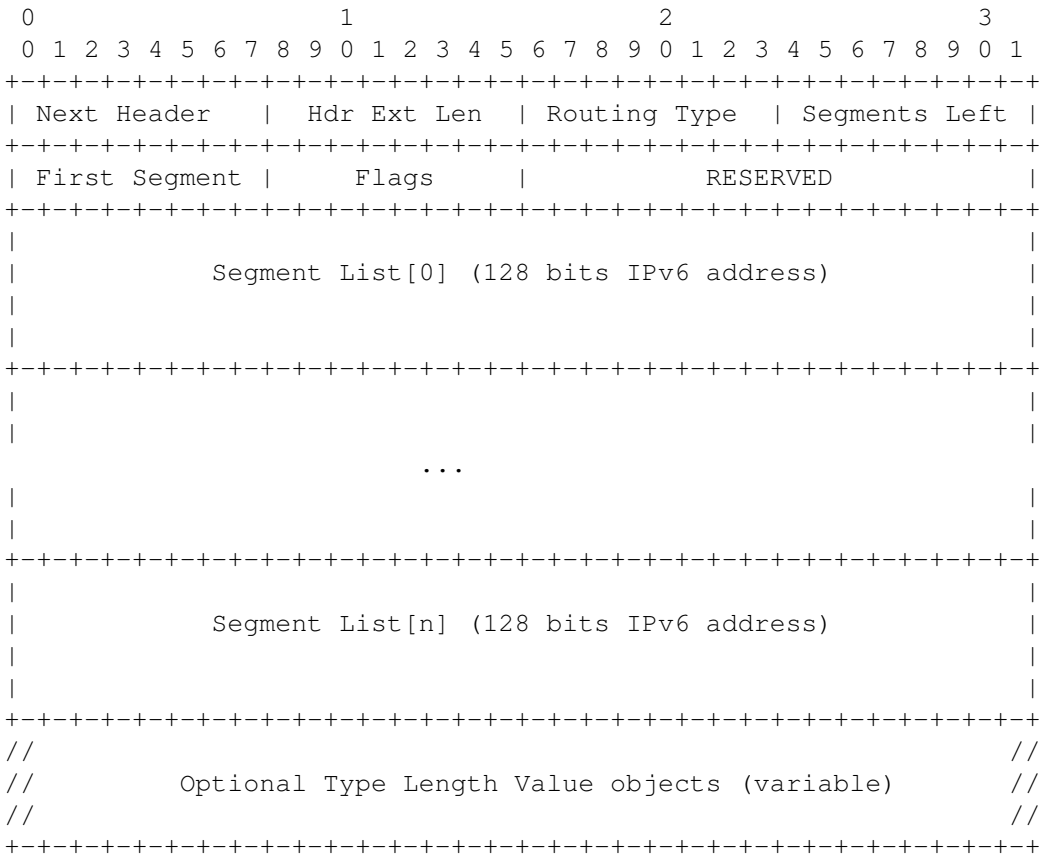


Figure 2.2: IPv6 Segment Routing Header.

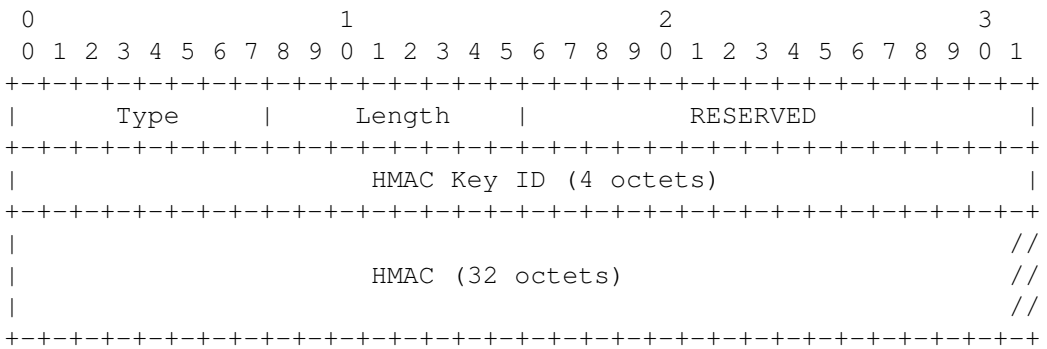


Figure 2.3: HMAC TLV.

nation of (i) the source IPv6 address, (ii) the `First Segment` field, (iii) the `Flags` field, (iv) the `HMAC Key ID` field, and (v) the full list of segments. The output of the HMAC function is stored in the `HMAC` field. If the output length of the hashing algorithm is less than 256 bits, then it is padded with zeroes. Otherwise, the output is trimmed to 256 bits.

2.1.1 Operations

The following basic operations are defined with respect to the Segment Routing Header: encapsulation, processing, and decapsulation.

Encapsulation

Encapsulation is used by an SR ingress node to impose an SRH onto packets. The original packet is encapsulated in an outer IPv6 header, which contains the SRH. The first three headers in the chain are thus `IPv6` → `SRH` → `IPv6`. Figure 2.4 shows an illustration. In earlier versions of the SRv6 IETF draft, another method of SRH insertion was described. The packet was not encapsulated. Rather, the SRH was directly inserted between the IPv6 header and the payload. The main issue with this technique is that it breaks ICMP messages triggered by such packets. Indeed, an ICMP message related to a packet contains part of its payload. When the ICMP reaches the source, it can recognize the original packet thanks to the attached payload. By modifying the packet en route, any generated ICMP message will no longer match the original packet. For this reason, direct insertion was removed from the specifications. However, a recent IETF draft [27] proposes to allow direct insertion at the condition that the original packet is not modified. Such scenario happens when the ingress node of an SR domain encapsulates packets within an outer IPv6 header, but without adding an SRH. The process of imposing an SRH is then realized by another node within the SR domain and thus decoupled from the IPv6 encapsulation itself. Encapsulation is preferred for persistent operations (*e.g.*, latency optimization, service chaining, etc.) while insertion is better suited for short-term failure mitigations. In both cases, any ICMP message generated for an encapsulated packet will reach the SR ingress node and not the original source of the packet. The SR ingress node is then able to react appropriately.

The SR ingress node should not blindly encapsulate packets. IPv6 headers contain a *flow label* field [28]. This field is taken into account by Equal-Cost Multi-Path (ECMP) hash functions, along with the source and destination address. As multiple flows may receive the same encapsulation, it is important to correctly set the flow label of the outer IPv6 header to ensure proper load balancing across the network resources, in case of ECMP. One solution is to inspect the original

packet and extract a flow label from its inner headers (*e.g.*, original source and destination, TCP ports, etc.). Another solution is to assume that the flow label of the inner packet is correctly defined and simply copy it into the outer IPv6 header. Each solution has its strengths and weaknesses. The former ensures that the flow label will actually depend on the inner packet's content, but require its inspection. Such an inspection may waste processing cycles. The latter can be achieved at no additional cost, but does not guarantee that the inner flow label was set to a meaningful value. At the time of writing and to the extent of our knowledge, there is no specification on how to handle flow labels for SR encapsulated packets. As such, it remains implementation-dependent.

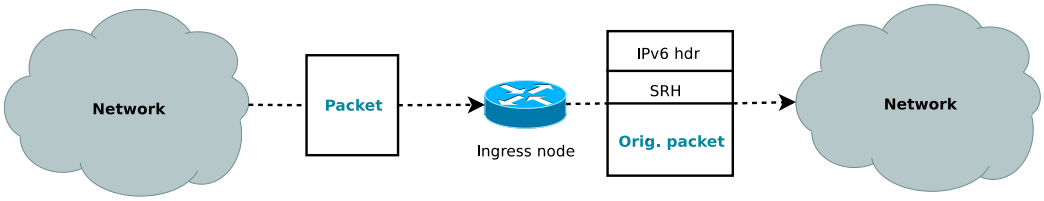


Figure 2.4: SRH encapsulation by intermediate node.

The SRH may also be inserted by the source of the packet. In this case, the packet is not encapsulated. The SRH is pushed on the packet right after the IPv6 header, like direct insertion mentioned above. Generated ICMP messages are not an issue in this case, as the source itself is aware of the SRH. Figure 2.5 illustrate this operation.

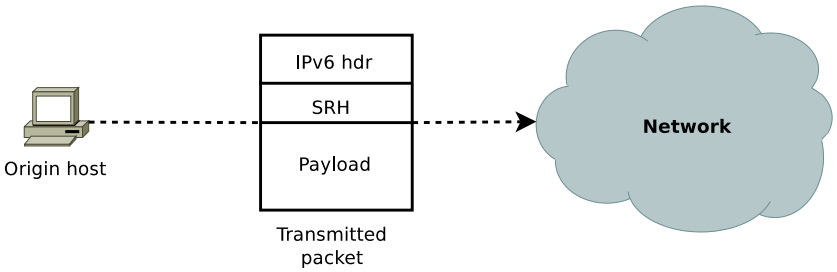


Figure 2.5: SRH insertion by source.

When an SRH has been inserted, the destination address of the packet is set to the first segment of the path (*i.e.*, the segment indexed by the `First Segment` field in the SRH). The packet is thus forwarded to the first segment following shortest path routing.

Algorithm 1 SRH processing.

```
1:  if DA = myself (segment endpoint) then
2:      if Segments Left > 0 then
3:          Decrement Segments Left
4:          Update DA with Segment List[Segments Left]
5:          Forward the packet out
6:      else
7:          Continue regular IPv6 processing of the packet
8:  e.g. decapsulate and forward
9:          End of processing
10:     end if
11: end if
```

Processing

When an SR-enabled packet reaches a segment endpoint, it must process the packet to forward it to the next segment. The active segment is represented by the current destination address of the packet. The SRH processing algorithm is defined by Algorithm 1. The basic case is when the segment endpoint is not the last segment of the path (*i.e.*, `Segments Left` > 0). The processing node decrements the `Segments Left` field and updates the destination address of the packet to the next segment, referenced by the decremented field. Then, the packet is forwarded to its new destination following shortest path routing. Figure 2.6 shows an illustration of this operation.

Between two segment endpoints, an SR-enabled packet may transit through several intermediate routers. Note that these routers do not need to support SRv6. Indeed, as the destination address of the packet is not local to those intermediate routers, they simply apply the standard shortest-path IPv6 forwarding.

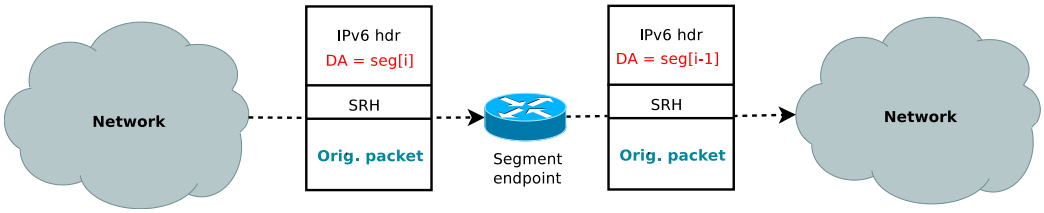


Figure 2.6: SRH processing.

Decapsulation

The last segment in the path of an encapsulated SR packet is the SR egress node. When a packet reaches that node, its `Segments Left` field is equal to zero. The node decapsulates the original packet by removing the outer IPv6 header and the SRH. The original packet is then forwarded to its destination, following the shortest path routing. Figure 2.7 shows an illustration of this operation.

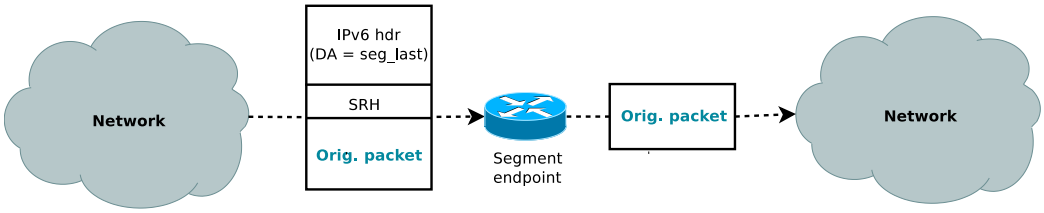


Figure 2.7: SRH decapsulation.

When the SRH is inserted by the source of the packet, there is no encapsulation. The last segment of the path is thus the final destination of the packet. In this case, when the last node receives such a packet, it simply continues the processing of the payload, ultimately delivering it to the related application.

2.1.2 HMAC validation

The SRv6 specifications [19] define how to generate and verify the HMAC field, but the decision to actually verify and enforce the HMAC is left to the operator. The HMAC TLV is meant to ensure the authenticity and integrity of the SRH when it is transmitted from beyond the trusted SR domain. As the HMAC computation induces a packet processing time overhead, HMAC validation should only be performed once at the edge of the SR domain.

2.2 Network programming

In the previous sections, we presented the basic operations of SRv6. The *SRv6 Network Programming* IETF draft [29] extends this definition and proposes a more advanced model where each segment can be represented as an arbitrary network function and conversely. In this section, we present the main ideas of this draft.

Each SRv6 node keeps a *My Local SID* table. This table lists the segments that are meaningful to the local node, which is referred to as the *parent node* of those segments. Each segment is composed of two parts. The first part is an IPv6 prefix

that is announced by the IGP. The second part, *i.e.*, the host part of the address, encodes the function associated to the SRv6 segment.

If an SRv6 node receives an SR-enabled packet whose active segment (*i.e.*, the IPv6 destination address of the packet) has no corresponding entry in the local SID table, then the packet should be dropped. Otherwise, the function associated with the segment is executed. Note that while the segment's IPv6 address is routed to its parent node, the address is not necessarily attached to a local interface. Several core functions are defined in [29]. The `End` function defines the basic segment endpoint processing as in Algorithm 1. It is equivalent to a node segment. The `End.X` function extends the endpoint behavior and specifies a given next hop to which the packet must be forwarded. It is equivalent to an adjacency segment. The `End.B6` function defines a *binding segment*. Such a segment maps to an *SRv6 policy*, *i.e.*, a list of segments. The behavior of such a segment is to add a new list of segments (*i.e.*, a new SRH) on top of the current one, either through direct insertion (`End.B6`) or encapsulation (`End.B6.Encaps`).

Other SRv6 functions exist, such as encapsulation of IPv4 packets or layer-2 frames, etc. In this thesis, we focus on node segments (`End`), adjacency segments (`End.X`), and binding segments (`End.B6`$\star$).

2.3 Related work

Multiple work on Segment Routing exist in the literature. In [15], Filsfils *et al.* describe the SR architecture in general as well as the main use cases that would benefit from SR. In the area of optimization of network utilization and requirements, Bhatia *et al.* discuss online and offline algorithms that leverage SR to optimize network utilization in [30]. In [31], Hartert *et al.* propose an SR-based framework to express and implement network requirements in a flexible and efficient manner. In [32], Hao *et al.* propose a linear programming model to optimize the restoration of bandwidth upon failures in segment routed networks. In the area related to properties of Segment Routing and its implementation in a network, Davoli *et al.* describe in [33] how Segment Routing can enable SDN-like per-flow control without adding state in core routers. In [34], Giorgetti *et al.* propose algorithms to minimize the number of segments required to compute segmented paths. Salsano *et al.* propose in [35] methods to leverage SR in a network without requiring extensions to routing protocols.

2.4 Conclusion

In this chapter, we presented the core principles of Segment Routing. We defined SRv6, the IPv6 flavor of Segment Routing, and enumerated its basic operations. We explained how SR enables to steer packets through arbitrary paths by attaching *segments* to the packets, following the principles of the source routing paradigm. Finally, we presented how SRv6 can provide flexible network programmability features by associating arbitrary functions to segments.

Chapter 3

Implementation of IPv6 Segment Routing in Linux

A key element in exploring and researching IPv6 Segment Routing (SRv6) is an open-source implementation. Our Linux kernel implementation has been merged into the main Linux tree and is part of Linux 4.10 released in February 2017 [36, 37]. We support the version 5 of the SRH specifications [19]. In this chapter, we first briefly explain the basic principles of the networking stack in the Linux kernel. Then, we present our SRv6 implementation [38], as well as supporting user space tools. Finally, we present our test framework and evaluate the performance of our implementation.

3.1 Networking in the Linux kernel

The Linux networking subsystem is very complex. As of Linux 4.10, it comprises more than 700,000 lines of code, without counting the drivers. In our SRv6 implementation, we mainly interact with the IPv6 packet processing and the routing engine. In this section, we explain how packets are internally represented by the kernel. Then, we briefly describe the path of a packet from the network interface through several stages of the routing engine. Finally, we explain the concept of network namespaces.

3.1.1 Socket buffers

Internally, packets are represented as socket buffers, or `skb`'s. An `skb` is a kernel structure of type `struct sk_buff` that represents a network packet, with metadata and payload. The metadata includes the ingress interface, checksum, header offsets, and other layer-specific data. The actual packet data can be

stored in two locations. The first location is a contiguous memory area, called the `skb` header. When all the packet data is contained in this area, the `skb` is said to be *linear*. The header structure is handled by four variables in the `skb`: `head`, `data`, `tail` and `end`. Figure 3.1 shows an illustration of the `skb` header structure. The `head` pointer references the start of the header. The `data` pointer references the start of the actual packet data. The `tail` pointer references the end of the packet data. Finally, the `end` pointer references the end of the `skb` header. The head room and tail room enable to prepend and append data to the packet. As packets are constructed from bottom to top, data is often prepended. If there is not enough head or tail room, the header is reallocated to a larger memory area. The `skb` also contains several offsets to easily access different parts of the packet, such as the network header, transport header, inner headers (in case of encapsulation), etc.

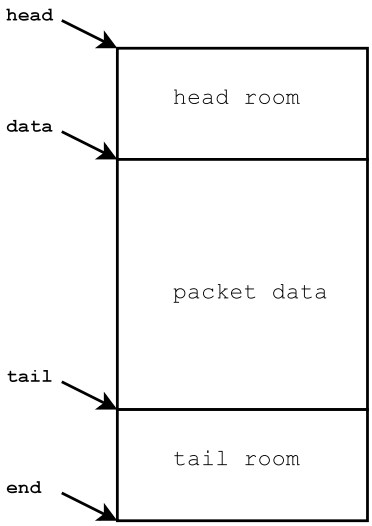


Figure 3.1: `skb` header structure. Memory addresses grow from top to bottom.

The second location for packet data storage is *paged fragments*. At the end of the `skb` header, there is a `struct skb_shared_info` containing an array of fragments. This structure contains the `frags` and `nr_frags` fields, which are resp. the array of fragments and its size.[1] Each fragment is identified by a `(page, offset, length)` tuple. When an `skb` contains fragments, the packet data is composed of the concatenation of the `skb` header and each fragment. In this case, it is no longer possible to append data in the tail room. Rather,

[1]The structure also contains a `frag_list` field, which is a list of `skb`'s. This list is used to store related `skb`'s such as IPv6 fragments. They are not related to the paged fragments we describe.

new data must be appended as a new fragment. Such an `skb` is said to be *nonlinear*. A nonlinear `skb` is mainly used for zero-copy packet transmission. Through the `sendmsg()` system call, a user space application can specify a vector of non-contiguous memory chunks to be transmitted. Instead of copying all the data into the linear `skb` header, the kernel simply stores references to those chunks using paged fragments. When the driver sends the packet to the NIC, it uses DMA to directly transfer the page chunks from the main memory to the network device, thus avoiding unnecessary copies. This technique is called *scatter-gather*. Another possible usage of nonlinear `skb`'s is when a network driver fetches Ethernet frames longer than the standard length of 1,500 (*jumbo frames*). For example, the `e1000` driver splits jumbo frames into multiple paged fragments.

3.1.2 Packet processing

The networking subsystem of the Linux kernel is divided into multiple layers. The lowest level is the network driver, which is closest to the hardware. When a packet arrives at the network interface card (NIC), it is copied into main memory and handed to the network driver. The driver transforms the raw packet data into a socket buffer. Once the `skb` is built, it is handed over to the upper layer, e.g. IPv4 or IPv6. The driver may pass several `skb`'s at a time using Generic Receive Offloading (GRO) [39]. GRO is a software mechanism that enables network drivers to group similar contiguous packets before handing them over to the upper layer, reducing the number of calls. The idea is to group packets that belong to the same flow. Each network protocol can define its `gro_receive` function that decides whether a given `skb` can be merged with previous ones. For example, the IPv6 GRO receive function verifies that all IPv6 header fields are the same except the length and traffic class. If the fields differ, then the `skb` is left as a standalone packet. If the fields match, then the decision to merge is delegated to the GRO receive function of the inner protocol. For example, if a TCP payload immediately follows the IPv6 header, then the GRO receive function of TCP is called. In turns, this function decides if the `skb` can be merged by checking the ports, sequence numbers, etc. When it is no longer possible to merge, the aggregated GRO `skb` is sent to the upper layer.

At the IP layer, the `skb`'s are processed by the routing engine. The kernel decides whether the packet should be locally delivered, or forwarded to another network node. Figure 3.2 shows an overview of the routing decision process. Each packet goes through several processing stages. Let us consider an IPv6 packet that was just delivered by the network driver. It enters the `ipv6_rcv()` function that corresponds to the `PREROUTING` stage. This function parses the IPv6 header, then decides whether the packet is bound for the local host or must be forwarded to the network. It does so by matching the destination of the packet

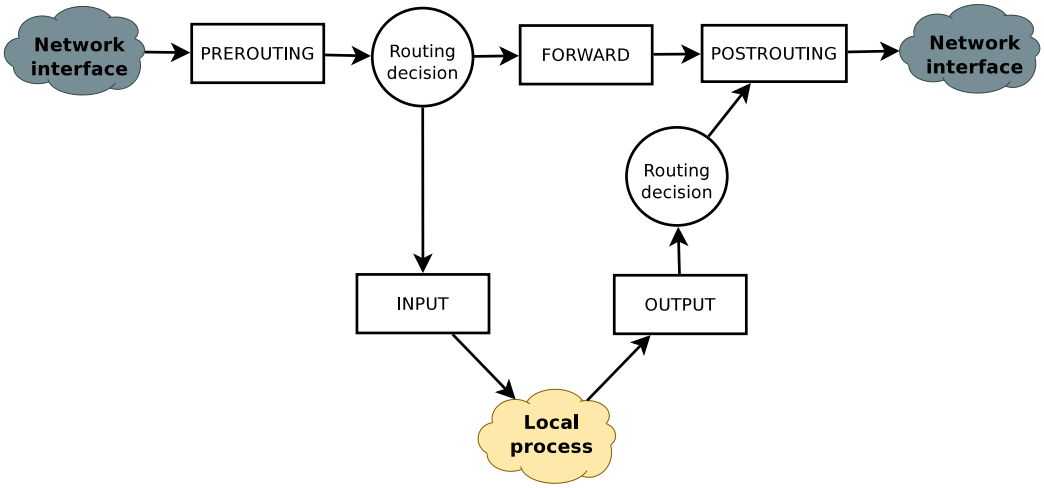


Figure 3.2: Linux routing decision process.

against the IPv6 routing table entries.[2] If the packet is to be forwarded, it enters the `ip6_forward()` function, corresponding to the `FORWARD` stage. This function is in charge of selecting the next hop, decrementing the hop limit, etc. Then, it enters the `ip6_output()` function, corresponding to the `POSTROUTING` stage. In this stage, the packet is ready to be transmitted and is handed over to the lower layers, ultimately being transmitted by the NIC. On the other hand, if the packet is to be locally delivered (*i.e.*, the destination address matches a local address), it enters the `ip6_input()` function, corresponding to the `INPUT` stage. This function iteratively processes each extension header in the IPv6 header chain until it reaches a final payload (*e.g.*, TCP, UDP), which is processed by its own input function.

When an application sends packets to the network, the packets are built from bottom to top. First, the transport header is pushed on top of the payload. Then, optional IPv6 extension headers are pushed on top of the transport header thanks to the `ipv6_push_nfrag_opts()` function. Finally, the IPv6 header is pushed on top and the `skb` enters the `OUTPUT` stage through, *e.g.*, the `ip6_xmit()` function for TCP or the `ip6_local_out()` function for other transport protocols. The routing decision is performed and the `skb` then enters the `POSTROUTING` stage and is transmitted to the network.

[2] In practice, other parameters can be used in the routing decision process, such as the source address. For the sake of simplicity, we only consider destination addresses.

3.1.3 Network namespaces

In Linux, namespaces [40] enable to isolate various parts of the system, as a kind of virtualization. For example, the PID namespace isolates the process ID number space and enables processes running in different PID namespaces to have the same PID number. The network namespaces [41] isolate routing tables and interfaces. When a new network namespace is created, it contains only a loopback interface. Virtual Ethernet (`veth`) interface pairs are used to interconnect namespaces. A `veth` pair consists of two connected virtual interfaces (packets transmitted on one interface will be received on the other and conversely). Each interface of the pair is then assigned to (different) namespaces, effectively interconnecting them. Other network parameters such as several sysctls are also isolated per namespace. It is the programmer's responsibility to ensure that a particular network feature is namespace-aware. Different types of namespaces are often used in combination by virtualization techniques such as Docker to virtualize PIDs, network, filesystems, etc., but can also be used independently to, *e.g.*, specifically emulate network topologies [42].

3.2 Data plane support

The core of the SRv6 implementation is the Segment Routing Header processing capability. It enables a Linux node to act as a segment endpoint and as an SR egress node as described in Section 2.1. When a segment endpoint receives an SR-enabled packet, the destination address of the packet is local to the segment endpoint. Thus, the packet enters the `INPUT` stage. We added a function `ipv6_srh_rcv()` called whenever the IPv6 input function encounters an SRH in the header chain. Listing 3.1 shows the C structure holding an SRH.

Listing 3.1: SRH structure.

```
struct ipv6_sr_hdr {
        __u8    nexthdr;
        __u8    hdrlen;
        __u8    type;
        __u8    segments_left;
        __u8    first_segment;
        __u8    flags;
        __u16   reserved;

        struct in6_addr segments[0];
};
```

The function performs the following operations. First, it checks that the node is allowed to act as a segment endpoint for SR-enabled packets coming from the ingress interface (`skb->dev`). This policy is configured through a per-interface sysctl boolean parameter `seg6_enabled`. If the boolean is set to false, then the `skb` is dropped. Otherwise, the processing continues. The packet then goes through an optional HMAC validation. This validation is controlled by a per-interface sysctl `seg6_hmac_require`. This parameter can take three different values: (i) a value of -1 means that the node must accept all SR-enabled packets, regardless of whether an HMAC TLV is absent, present, valid or invalid, (ii) a value of 0 means that the node must accept SR-enabled packets that do not have an HMAC TLV and must ensure the validity of the HMAC TLV when present, and (iii) a value of 1 means that the node must enforce a valid HMAC for all SR-enabled packets. Table 3.1 shows a summary of the possible combinations of `seg6_hmac_require` and the HMAC state.

Table 3.1: Possible scenarios for HMAC policies.

`seg6_hmac_require`	HMAC valid	HMAC absent	HMAC invalid
`-1`	Pass	Pass	Pass
`0`	Pass	Pass	Drop
`1`	Pass	Drop	Drop

Once those preliminary checks have been performed, the function handles two main cases: `Segments Left` being non-zero, and `Segments Left` being equal to zero. Let us first consider the latter case, where the node acts as an SR egress node. As the node is the last segment of the path, it has no choice but to inspect the inner header to decide the fate of the packet. If the next header is another IPv6 extension header or a final protocol (*e.g.*, TCP, UDP), then the header chain processing continues as normal, and the `skb` is eventually delivered to the corresponding local process. This would happen, for example, when the SRH was directly set by the source of the packet. Conversely, if the original packet was encapsulated by an SR ingress node, then the next header would typically be an IPv6 header (*i.e.*, IPv6-in-IPv6 encapsulation). However, nothing forbids, *e.g.*, that an IPv4 packet be encapsulated within the outer IPv6 header and the SRH. As such, it would make sense to let the kernel continue the default processing of the next header, as for the non-encapsulated case. However, the kernel handles IP headers differently, depending on whether it is the outermost header (the first header processed on the ingress interface, not counting the possible Ethernet header) or an inner, encapsulated header. When the kernel encounters an encapsulated inner header, it attempts to find an existing stateful tunnel interface, corresponding to the source and destination of the outer header. As no such interface exists, the `skb` is dropped. To avoid this issue, we bypass the default processing and rein-

ject the inner packet in the ingress interface. The side-effect is that the function explicitly checks for an inner IP header. As of Linux 4.10, only IPv6-in-IPv6 encapsulation is supported.

When the Linux node is an intermediate segment endpoint (*i.e.*, `Segments Left` > 0), it must forward the packet to the next segment. To realize this, the `ipv6_srh_rcv()` function decrements the number of segments left and updates the destination address of the packet to the next segment. Afterwards, a routing decision is applied to the `skb` thanks to the `ip6_route_input()` function. If the next segment is local to the node, then the `skb` is looped back to the beginning of the SRH receive function, after decrementing and checking the hop limit. This is an implementation choice that enables to skip a redundant re-entry into the `ip6_input()` function. In the future, this behavior might be controlled by a user-defined parameter. This parameter would enable to choose between fast-path loop-back and slow-path re-entry. The rationale is that the fast re-entry skips the `INPUT` netfilter hook, which may be needed in some usecases. If the next segment is non local, then the `skb` enters the `FORWARD` stage.

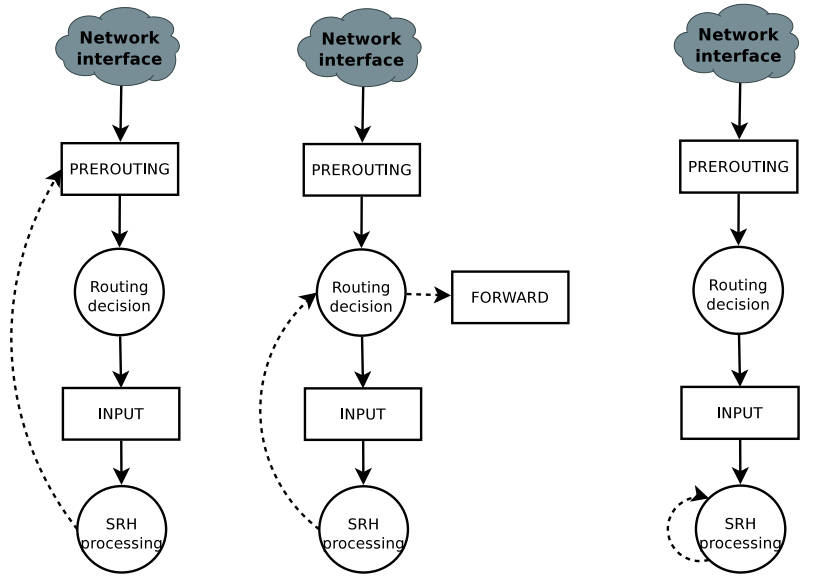
Figure 3.3 summarizes the flow of an SR-enabled packet through the networking subsystem. The codepath of the `skb` to the SRH processing function is shown with plain arrows. The dashed arrows show the codepath of the `skb` immediately after the SRH processing. Figure 3.3a shows the decapsulated `skb` being reinjected at the interface level. Figure 3.3b shows the `skb` being forwarded to the next segment. Figure 3.3c shows the `skb` being looped back within the SRH processing function to handle a local next segment.

3.3 Control plane support

To enable a Linux node to act as an SR ingress node, we implemented support for SRH insertion. This implementation evolved significantly before reaching the latest version available in Linux 4.10. We distinguish three main versions of the SRH insertion implementation. The first version, available in the very first out-of-tree, proof-of-concept, public release of SRv6[3] was very intrusive to the kernel. It had a hardcoded function call, at the end of `ip6_forward()`, to the SRH insertion function. This function tried to match the destination address of the packet against elements of an internal linked list. These elements were mapping an IPv6 prefix to an SRH. If the destination address matched the prefix, then the SRH was directly inserted into the packet, right after the IPv6 header.[4] The `skb` was then re-injected in the `ip6_forward()` function and forwarded to the first segment. This version had several problems. From a software engineering point of view,

[3]The kernel version was 3.14, released in April 2014.

[4]At that time, the IETF draft of SRv6 was not mentioning encapsulation, only direct insertion.



(a) SR egress node. (b) Remote next segment. (c) Local next segment.

Figure 3.3: Possible codepaths for SRH processing.

inserting such specific code in a highly generic function is a very bad idea. Additionally, performing SRH insertion only in the `FORWARD` stage prevented packets generated by a local application to receive an SRH, as they do not pass through this stage. The internal linked list was also a wrong choice of data structure, as it is neither efficient nor adapted, due to its linear time complexity for the lookup operation and due to the fact that it did not implement a longest prefix match (LPM) algorithm, which is more suited for prefix matching. The linked list was later replaced by an actual, more adapted LPM tree. The first in-depth refactoring of the SRv6 code took place at the same time as direct insertion was replaced by encapsulation in the IETF draft. To accommodate the new encapsulation mechanism, we attempted to leverage the existing IPv6 encapsulation mechanisms available in the Linux kernel. In particular, we modified the `ip6tnl` module. This module provides IPv6-in-IPv6 encapsulation. We decided to use and extend this module with the support of extension headers, and incidentally the SRH. However, two issues quickly came to light. The first issue is related to scaling. Such a solution would require one interface per SRH insertion rule, which could rapidly become an issue due to kernel memory consumption. The second issue is related to the differences between a plain IPv6-in-IPv6 tunnel ingress and an SRv6 ingress node. With `ip6tnl`, an IPv6 tunnel is configured using a pair of addresses: the local and the remote addresses. No other parameter is used to distinguish IPv6 tunnels. As such, it is not possible to create more than one tunnel from the same source to

the same destination. This is a severe limitation, as it is probable, even likely, that more than one SRH could have the same egress node.

The current, more elegant solution was brought thanks to a new technique, the *lightweight tunnels* (LWTs), that was merged in Linux 4.3. LWTs are a technique to implement interfaceless tunnels, which is exactly what we were trying to do. The idea of lightweight tunnels is the following. Each route in the kernel routing table is associated with two function pointers, `input` and `output`. Those pointers are initialized at the creation of the route. For an IPv6 route, the `output` function pointer references the `ip6_output()` function, that transmits the `skb` to the egress interface. The `input` function pointer depends on whether the route is local (packets matching this route must be delivered to a local process) or non-local (packets must be forwarded to a next hop). If the route is local, the function pointer references the `ip6_input()` function. Otherwise, it is `ip6_forward()` that is referenced. The idea of the lightweight tunnels is to override those function pointers with custom functions. To implement a lightweight tunnel, one needs to define their own input and/or output functions. Then, each route created to specifically use this LWT will have its `input` and/or `output` function pointers reference the custom functions. Per-tunnel stateful data (*tunnel state*) is also stored in the route. This technique has the advantage of using the existing routing table, thus avoiding the need to define a custom data structure for packet matching. It is also highly customizable, enabling differentiated treatment for forwarded packets and for locally generated packets. Lightweight tunnels are configured from user space through the `rtnetlink` protocol, which is commonly used by the `iproute2` tool to configure the routing tables.

Consequently, we implemented SRH insertion using the lightweight tunnels. When such a route is created, the full SRH is built in user space and transmitted to the kernel through `rtnetlink`. After checking the consistency of the SRH, it is stored as the tunnel state of the route. Along the SRH is stored an optional parameter stating whether the SRH should be directly inserted or encapsulated. Finally, a `dst_cache` entry stores the routing entry associated with the first segment of the SRH, enabling the route lookup for the first segment to be performed only once. The `dst_cache` kernel mechanism ensures that if the stored routing entry is deleted or obsolete, it reverts to a full route lookup. Both the `input` and `output` function pointers of the route ultimately call the `seg6_do_srh()` function. This function effectively inserts the SRH on the `skb`. The direct insertion function (`seg6_do_srh_inline()`) simply inserts the SRH between the IPv6 header and the rest of the packet. The encapsulation function (`seg6_do_srh_encap()`) needs to provision a new IPv6 header. The traffic class and the flow label are copied from the inner IPv6 header. In the future, this behavior should be made configurable, *e.g.*, by letting the kernel compute the

outer flow label based on the inner packet's 5-tuple. Such a behavior would ensure proper ECMP behavior when the inner flow label is null or unrelated to the packet's flow. The hop limit is also copied from the inner IPv6 header. The destination address is obviously set to the address of the first segment. The source address selection is less straightforward. It must reference the SR ingress node, but such a node may have multiple IPv6 addresses. At first, we decided to select the first address of the egress interface. However, during early benchmarking, we noted that the address selection function was time expensive. To avoid this, we created a per-namespace configurable parameter, holding the source address to use for SR encapsulations. In the future, this parameter could be directly attached to the routes for a more fine-grained configuration. Once an `skb` is augmented with an SRH, it is forwarded to the first segment according to the normal kernel routing mechanisms.

Listing 3.2: `iproute2` command to insert an SRv6 encapsulation route.

```
ip -6 route add fc42::/64 encap seg6 mode encap
   segs fc00::1,2001:db8::1,fc10::7 dev eth0
```

To enable user space control over SRv6, we extended the `iproute2` user space tool to insert, modify and read routes that use the SRH lightweight tunnels extension. Listing 3.2 shows an example of a route insertion command. This route matches packets whose destination belongs to the prefix `fc42::/64` and inserts an SRH on these packets (`encap seg6`). The original packet is encapsulated in an outer IPv6 header (`mode encap`). To directly insert the SRH in the original packet, one can use `mode inline`. The list of segments is given as a comma-separated list of IPv6 addresses. Finally, a non-loopback device must be specified. The choice of this device has no actual effect on the route, provided that it is not the loopback interface. Indeed, an IPv6 route attached to the loopback interface means that at least one address of the corresponding prefix is local to the host and not attached to a given physical interface. Such routes are marked as unreachable, to prevent the host from forwarding packets whose destination address is part of a local prefix, but not explicitly attached to the host. Furthermore, it is mandatory to specify a device for a route, either directly with the `dev` keyword, or indirectly by specifying the next hop with the `via` keyword. Thus, we need to specify a non-loopback interface, which has no effect on the route but enables the kernel to properly accept it.

Such routes are namespace-wide. To support a more fine-grained control over SRH insertion, we implemented a per-socket interface through the `setsockopt()` system call. Such an interface enables applications to specify the SRH to be inserted on a socket level. When building a packet generated by

Listing 3.3: Sample code to define a per-socket SRH.

```
struct ipv6_sr_hdr *srh;
int fd, srh_len;

srh_len = build_srh(&srh);
fd = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
setsockopt(fd, IPPROTO_IPV6, IPV6_RTHDR, srh, srh_len);
```

a local application, the kernel calls `ipv6_push_nfrag_opts()` before pushing the top IPv6 header. If an SRH is attached to the socket, the function will call `ipv6_push_rthdr4()` that effectively pushes the SRH on the packet. The last segment of the SRH is set to the original destination. The first segment is returned, to be set as the actual destination of the packet. Listing 3.3 shows how a C application can define an SRH for a TCP socket.

3.4 HMAC

In the early implementations of SRv6, the HMAC library was rudimentary. It supported only the SHA-1 hashing algorithm, through kernel library helpers, without dynamic memory allocation. The evolution from a proof-of-concept to a fully fledged implementation required a stronger hashing algorithm, and with reason [43]. We chose to support SHA-256. This upgrade introduced unexpected side effects. Indeed, the kernel implementation of SHA-256 was no longer available through library helpers, and required the use of the crypto API. The crypto API is a kernel subsystem that provides a generic interface to cryptographic features such as hashing and cipher algorithms, including HMAC algorithms with various hashing functions. However, the crypto API requires the allocation of per-algorithm descriptors, through the `crypto_alloc_shash()` function for symmetric hashing algorithms. This led to two main issues. The first issue is related to the internals of the function. When a required crypto algorithm is not readily available, the function will try to find and load the corresponding kernel module using `request_module()`. This function is not atomic, *i.e.*, it may sleep while waiting for the module to be loaded, causing another task to be scheduled. This cannot happen within an interrupt context, *i.e.*, during the execution of a function that was called due to a software or hardware interrupt, such as ingress packet handlers. Thus, the allocator function cannot be called in the SRH processing function, `ipv6_srh_rcv()`. The second issue is that, even if the allocator can be made atomic, doing per-packet allocations would be a waste of resources, as those allocations would be the same and could be reused from one packet to the

next.

The current implementation of HMAC for SRv6 leverages per-CPU allocations. The idea is to pre-allocate algorithm descriptors and to reuse them in the packet processing function, effectively improving from one allocation per packet to a single allocation at the initialization of SRv6. In SMP systems (multi-processor or multi-core), multiple threads can run the SRH receiving function simultaneously, thus potentially computing HMACs concurrently. Using a single descriptor would require locking, and thus might create lock contention. To prevent this, we allocate one descriptor per HMAC algorithm per CPU. Such a technique enables each thread to work on its local copy of the HMAC descriptor, without competing for a shared resource. In addition to per-CPU HMAC descriptors, we also allocate per-CPU ring buffers. Those buffers store the HMAC input (or *text*) that is derived from each packet's IPv6 header and SRH. Each supported HMAC algorithm is described by a `struct seg6_hmac_algo` structure, as shown in Listing 3.4. This structure contains a statically defined algorithm ID, a name describing the algorithm in the syntax of the crypto API. For example, HMAC with SHA-256 is written `hmac(sha256)`. Finally, the structure stores per-CPU pointers to instances of the algorithm (`tfms` field) and per-CPU pointers to auxiliary working buffers for the hashing algorithm (`shashs` field), that will be used during the hash computations. The ring buffers are statically defined in the code, as they are of small and fixed size.[5] For each supported HMAC algorithm, a `seg6_hmac_algo` structure is statically defined. The allocation of their per-CPU pointers is performed once at boot time. As defined in Section 2.1, an HMAC TLV contains a key ID, which is an opaque, operator-defined value that maps to at least a secret and a hashing algorithm. We implement this mapping with the `struct seg6_hmac_info` structure, as shown in Listing 3.5. This structure contains the key ID itself, the secret and its length, and the hashing algorithm identifier. Additionally, the structure is part of a hashtable through its `struct rhash_head` element, enabling fast lookups. The `struct rcu_head` element enables the structure to be asynchronously freed upon deletion, in the case where it is still in use when the mapping is removed.[6]

Figure 3.4 illustrates the memory layout of the HMAC structures. On top is shown the static `seg6_hmac_algo` structures containing per-CPU pointers to algorithm instances (`tfms`) and auxiliary working buffers (`shashs`). Below is shown each CPU, holding a local `hmac_ring` static buffer that contains HMAC texts. The local instances of `tfm` and `shash` are stored on the heap. Only the SHA-256 variant is represented. In practice, each CPU will hold an algorithm

[5] `static DEFINE_PER_CPU(char [SEG6_HMAC_RING_SIZE], hmac_ring);`

[6] The Read-copy update (RCU) mechanism implemented in the Linux kernel is a complicated synchronisation mechanism whose details are out of the scope of this thesis. For further information about RCU, the reader is invited to consult [44, 45, 46]

Listing 3.4: SR HMAC algorithm description.

```
struct seg6_hmac_algo {
        u8 alg_id;
        char name[64];
        struct crypto_shash * __percpu *tfms;
        struct shash_desc * __percpu *shashs;
};
```

Listing 3.5: SR HMAC mapping between key ID and secret, hashing algorithm.

```
struct seg6_hmac_info {
        struct rhash_head node;
        struct rcu_head rcu;

        u32 hmackeyid;
        char secret[SEG6_HMAC_SECRET_LEN];
        u8 slen;
        u8 alg_id;
};
```

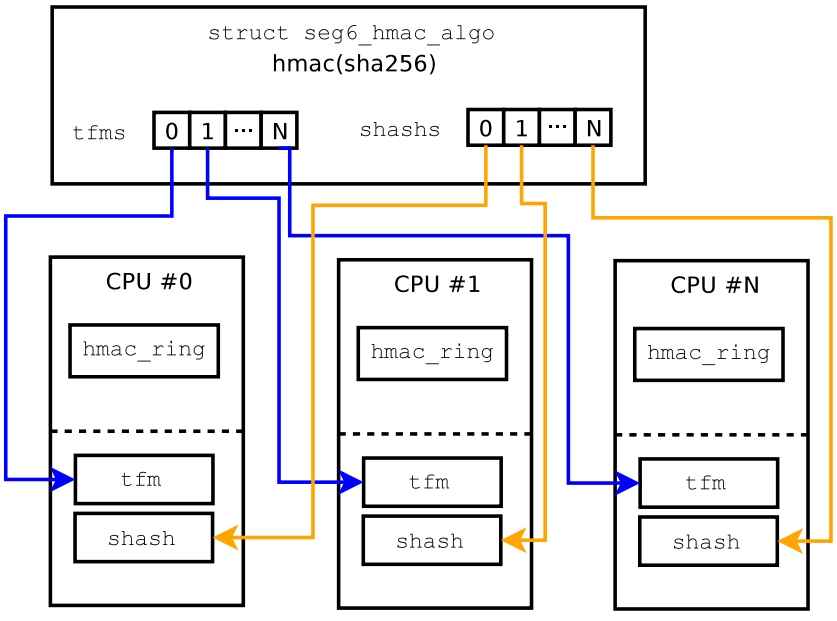


Figure 3.4: Memory layout of HMAC structures.

instance and an auxiliary working buffer for each hash variant.

The main function responsible for HMAC computation is `seg6_hmac_compute()`. It takes as input an HMAC descriptor `struct seg6_hmac_info`, an SRH, an IPv6 source address and a pointer to store the output of the HMAC algorithm. First, the function computes the length of the input text, which depends mainly on the number of segments present in the SRH. In Linux 4.10, the ring buffer is sized at 256 bytes, which allows for up to 14 segments in the SRH. Then, the function needs to ensure that no concurrent access will happen on its ring buffer. As the ring buffer is local to the CPU, there is no need to protect against parallel accesses even in SMP systems (as one CPU cannot simultaneously execute more than one task). However, in preemptible kernels[7], a kernel task may be preempted at almost any point in time. This means that the current task can be rescheduled to another CPU (thus changing of ring buffer in the middle of the computation), or it can be interleaved on the same CPU with another task running the same code (thus corrupting the ring buffer). To prevent this, we need to disable preemption through `preempt_disable()`. That would be sufficient if the HMAC computation function were called only from a process context, which is the case, *e.g.*, when processing packets generated by a local user application. However, when packets are available at a network interface, the NIC generates a hardware interrupt request (IRQ). The associated IRQ handler defined in the network driver fetches the packets, performs a quick preprocessing, enqueues the packets in a per-CPU backlog buffer and raises a software interrupt `NET_RX_SOFTIRQ`. As opposed to hardware interrupts, softirqs run with interrupts enabled and handle the most time-consuming part of the interrupt handlers. This prevents spending too much time with interrupts disabled. In the kernel, softirqs are also known as *bottom halves*. The raised softirq is handled by the `net_rx_action()` function which may ultimately call the HMAC computation function. If the IRQ was triggered in the middle of such a computation, then the ring buffer would get corrupted. We thus need to disable softirqs by calling `local_bh_disable()`. Note that this function also disable preemption, so that a call to `preempt_disable()` is redundant. This also protects the per-CPU algorithm descriptors and working buffers. Once the softirqs are disabled, the function computes the HMAC text and calls the crypto API functions that will effectively compute the HMAC. Immediately after the HMAC computation is done, the softirqs are re-enabled. Then, the result is copied into the output given as argument and the function returns.

Disabling preemption and softirqs has some consequences, mainly on real-time systems. The kernel will not be able to schedule another task on the CPU

[7] By default, only user space code is preemptible. Using `CONFIG_PREEMPT=y` enables kernel space code to also be preempted.

running the HMAC computation until it completes. As an HMAC is computed on the microsecond level, the overall impact is therefore limited, especially on multi-core systems where other CPUs are likely to be available.

Listings 3.6 and 3.7 illustrate the possible contexts for HMAC computation. They are call stacks generated by the GDB `backtrace` command. After setting a breakpoint on `seg6_push_hmac()`, an ICMP echo-request was generated through the `ping6` command between two network namespaces on the same host. The destination matched an SR encapsulation route with HMAC. The first listing shows the process context, when computing the HMAC for the generated ICMP packet. From bottom to top, we can see the entry into the `sendto()` system call (frames 15-17), the construction of the packet (frames 8-14), and its path through our hooked `output` route pointer (frames 0-4). The second listing shows the call stack in interrupt context, when the packet has been received on an ingress network interface. The `NET_RX` softirq is invoked from a hard IRQ return path[8] (frames 16-21). The packet passes through the `PREROUTING` stage (frames 8-10), then the `INPUT` stage (frames 4-7). It enters the `ipv6_srh_rcv()` function, finally calling the HMAC computation function.

Listing 3.6: Call stack for `seg6_hmac_compute` in process context.

```
#0  seg6_hmac_compute () at net/ipv6/seg6_hmac.c:167
#1  0xffffffff819150cc in seg6_push_hmac () at net/ipv6/seg6_hmac.c:348
#2  0xffffffff81913dc6 in seg6_do_srh_encap () at net/ipv6/seg6_iptunnel.c:138
#3  seg6_do_srh () at net/ipv6/seg6_iptunnel.c:223
#4  0xffffffff81914099 in seg6_output () at net/ipv6/seg6_iptunnel.c:261
#5  0xffffffff817e92ff in lwtunnel_output () at net/core/lwtunnel.c:310
#6  0xffffffff81917b18 in dst_output () at ./include/net/dst.h:501
#7  ip6_local_out () at net/ipv6/output_core.c:172
#8  0xffffffff818d444a in ip6_send_skb () at net/ipv6/ip6_output.c:1727
#9  0xffffffff818d4518 in ip6_push_pending_frames () at net/ipv6/ip6_output.c
    :1747
#10 0xffffffff818f619d in rawv6_push_pending_frames () at net/ipv6/raw.c:613
#11 rawv6_sendmsg () at net/ipv6/raw.c:927
#12 0xffffffff81882cb5 in inet_sendmsg () at net/ipv4/af_inet.c:744
#13 0xffffffff8179d1a3 in sock_sendmsg_nosec () at net/socket.c:635
#14 sock_sendmsg () at net/socket.c:645
#15 0xffffffff8179d9af in SYSC_sendto () at net/socket.c:1687
#16 0xffffffff8179e9d9 in SyS_sendto () at net/socket.c:1655
#17 0xffffffff8197e941 in entry_SYSCALL_64_fastpath () at arch/x86/entry/entry_64
    .S:204
#18 0x00005604a7944e21 in ?? ()
```

As explained above, the HMAC computation function has multiple call sites. One of them is in the `ipv6_srh_rcv()` function, to verify the validity of an SR-enabled packet with an HMAC TLV. Another one is through the `seg6_do_srh()`

[8]The acute reader will notice that the hard IRQ in question is the timer interrupt. This is due to the fact that the ingress interface is virtual and has no corresponding hardware interrupt.

Listing 3.7: Call stack for `seg6_hmac_compute` in interrupt context.

```
#0  seg6_hmac_compute () at net/ipv6/seg6_hmac.c:167
#1  0xffffffff81914a06 in seg6_hmac_validate_skb () at net/ipv6/seg6_hmac.c:274
#2  0xffffffff819068fc in ipv6_srh_rcv () at net/ipv6/exthdrs.c:346
#3  ipv6_rthdr_rcv () at net/ipv6/exthdrs.c:487
#4  0xffffffff818d49d1 in ip6_input_finish () at net/ipv6/ip6_input.c:279
#5  0xffffffff818d5166 in NF_HOOK () at ./include/linux/netfilter.h:257
#6  ip6_input () at net/ipv6/ip6_input.c:322
#7  0xffffffff818d4732 in dst_input () at ./include/net/dst.h:507
#8  ip6_rcv_finish () at net/ipv6/ip6_input.c:69
#9  0xffffffff818d57df in NF_HOOK () at ./include/linux/netfilter.h:257
#10 ipv6_rcv () at net/ipv6/ip6_input.c:203
#11 0xffffffff817bdd94 in __netif_receive_skb_core () at net/core/dev.c:4190
#12 0xffffffff817be8a8 in __netif_receive_skb () at net/core/dev.c:4228
#13 0xffffffff817bff8d in process_backlog () at net/core/dev.c:4839
#14 0xffffffff817bf165 in napi_poll () at net/core/dev.c:5202
#15 net_rx_action () at net/core/dev.c:5267
#16 0xffffffff81074dc1 in __do_softirq () at kernel/softirq.c:284
#17 0xffffffff8107525d in invoke_softirq () at kernel/softirq.c:364
#18 irq_exit () at kernel/softirq.c:405
#19 0xffffffff81040388 in exiting_irq () at ./arch/x86/include/asm/apic.h:658
#20 smp_apic_timer_interrupt () at arch/x86/kernel/apic/apic.c:961
#21 0xffffffff8197f5e3 in apic_timer_interrupt () at arch/x86/entry/entry_64.S
    :707
#22 0xffffc90000387df8 in ?? ()
#23 0x0000000000000000 in ?? ()
```

function, to augment packets with an SRH containing an HMAC TLV. Finally, an HMAC may also be computed for a packet generated through a particular local socket that has received an SRH through the `setsocktopt()` system call. To define an SRv6 encapsulation route with an HMAC, `iproute2` provides the `hmac` parameter that takes a key ID in argument. The SRH sent to the kernel contains a template HMAC TLV with the key ID set and the HMAC value zeroed. The same construction applies for a per-socket SRH. The configuration of key IDs is performed through the `genetlink` protocol. We modified the `iproute2` tool to support HMAC key ID configuration. Listing 3.8 shows an example of a key ID configuration. The `sr hmac set` command takes a key ID and a hashing algorithm name in parameter.

Listing 3.8: HMAC key ID configuration example.

```
ip sr hmac set 42 sha256
```

3.5 Testing

3.5.1 Nanonet framework

To evaluate the correct behavior of our implementation, we performed tests in virtualized environments. At first, we used the Mininet [42] framework. However, we required several features that Mininet was missing. For example, we wanted to use named namespaces instead of anonymous namespaces, the latter requiring a control process to run. With named namespaces, it is easier to move between the virtual nodes of the emulated network using regular shell commands. To automate the configuration of the network, we also wanted to automatically assign an IPv6 prefix to each node and link. Finally, we wanted to be able to instantiate very large networks (*e.g.*, Tier-1 networks with hundreds of nodes and thousands of links). The main issue with such networks is the generation and propagation of routes. Spawning a routing daemon such as Quagga for each virtual node would consume a lot of resources and will rapidly become difficult to maintain. As a result, we decided that the most resource-efficient solution would be to pre-compute reachability information and install it as static routes in each virtual node. Helpers would be provided to emulate topology changes and IGP propagation.

Hence, we created our own framework that we (creatively) called Nanonet. We used the same underlying technology as Mininet (*i.e.*, network namespaces and virtual Ethernet pairs) but we took a very different approach to the instantiation and management of virtual network topologies. The source code of Nanonet is available on http://www.segment-routing.org.

The input of Nanonet is a text file describing a network topology. Each line describes a link between two named nodes and specifies the weight, bandwidth and latency of the link. The output is a `bash` file containing all the commands necessary to spawn the namespaces (one per node), create and assign the virtual Ethernet pairs (one per link), configure the link parameters and populate the routing table of each namespace. The virtual topology is thus instantiated by executing the bash file. Afterwards, it is possible to enter and execute commands in an arbitrary namespace by executing the command `ip netns exec` *`nsname`* `bash`.

The topology generation process is composed of the following steps: (i) prefixes assignation, (ii) routes computation and (iii) commands generation. The first step consists in assigning a prefix for each node and for each link. Each of them is a `/64` taken from larger `/32` prefixes. By default, the node prefixes are taken from `fc00:2::/32` and the link prefixes from `fc00:42::/32`. Then, in the second step, Nanonet computes the shortest path DAG for each node, with the Dijkstra algorithm. From these DAGs are extracted, for each node, the routes needed to reach each node prefix. No route is computed for the link prefixes.

Finally, the third step consists in generating all the commands necessary to instantiate the virtual topology. First, each namespace is spawned and its node prefix is assigned to the loopback interface. Forwarding and SR processing are enabled. Then, for each link, the corresponding virtual Ethernet pair is spawned and assigned to their respective namespaces. Each interface receives its link prefix and is configured with the bandwidth and delay parameters through the `tc` command. The bandwidth setting is implemented using a Hierarchy Token Bucket (HTB) queueing discipline. The delay is implemented through the `netem` module. Finally, each route is created using the `iproute2` command. If a node has multiple paths of equal cost to reach another node, then all the routes corresponding to the available paths are created. This enables the node to be reachable through Equal-Cost Multi-Path (ECMP). The list of commands is then written in the output file.

3.5.2 Limits of same-kernel testing

The ability to create a virtual network on a single Linux kernel is extremely helpful to quickly deploy virtual topologies, test network features, etc. However, a virtual network cannot, by definition, exactly reproduce a real network. Two aspects in particular can be challenging to reproduce, due to how the Linux networking stack works. These aspects are (i) ordering of operations and (ii) jitter. To understand why those cannot be faithfully reproduced, we need to analyse the management of queuing disciplines, explore a part of the scheduling subsystem, and dig into the interrupt handling behavior.

On Linux, each network interface is associated with a queuing discipline (or *qdisc*). A qdisc provides packet scheduling and queue management algorithms. Each packet transmitted to such an interface is sent through its qdisc, which may then choose to drop, delay, or immediately send the packet. The default qdisc has been `pfifo_fast` for a long time. It is a simple FIFO queue that provides basic prioritization capabilities. It is now being replaced by `fq_codel` [47, 48, 49], which attempts to reduce bufferbloat by leveraging the packets queuing delays. When a user application attempts to send data to the network through a given syscall (*e.g.*, `send()`), the kernel handler will eventually call `dev_queue_xmit()`. It is the generic function to send a packet to an interface queue, which subsequently calls the qdisc-specific `enqueue()` function. If the packet must be dropped, then the `skb` is freed. If the packet must be sent immediately, then the function `sch_direct_xmit()` is called, which will subsequently call the NIC driver's `ndo_start_xmit()` to effectively send the packet on the wire. If the packet is delayed, then it is put in the qdisc buffer. A timer may be activated to ensure that the packet will be transmitted at the right time. This queueing process is a first source of unexpected jitter. Indeed, right after the call to `enqueue()`, the `__qdisc_run()` function is executed. This

function will attempt to dequeue and transmit packets as much as it can, bounded by either a configurable quota of packets (64 by default), the unavailability of subsequent dequeueable packets, or a scheduling preemption event. However, a given qdisc may contain packets from various user applications. As such, each user call to, *e.g.*, `send()` may potentially induce latency by spending time in the kernel, transmitting packets that belong to other processes. Note that this jitter happens for any kind of interface, real and virtual.

On the other side, when a NIC receives a packet from the network, an `skb` is built from the raw data. Then, the driver enqueues the packet in a per-CPU backlog queue.[9] To minimize the time spent with interrupts disabled, the driver raises the `NET_RX` softirq and returns. This is realized by the `netif_rx_internal()` function. This softirq will then be processed on the IRQ return path, with interrupts enabled. The `net_rx_action()` function is then executed and the `skb` is processed as explained in Section 3.1.1. However, with virtual Ethernet pairs, there is no hardware IRQ as the packet is generated from within the kernel. The packet transmission function that the `veth` driver registers as its `ndo_start_xmit()` function simply calls the `dev_forward_skb()` function. The purpose of the latter is to transfer an `skb` between two local interfaces. In practice, it calls the `netif_rx_internal()` function and returns. As mentioned earlier, this function enqueues the `skb` into a per-CPU backlog queue and raises the `NET_RX` softirq. This softirq is then processed by `do_softirq()` either when (i) returning from IRQ, (ii) re-enabling software interrupts if they were disabled, or (iii) returning to user space. The first option will not happen directly as we are not in an IRQ handler, but in a process context. However, the packet enqueueing function is always called with software interrupts disabled. As such, the pending softirq will be processed as soon as software interrupts are re-enabled, which is at the end of the `POSTROUTING` stage. If the corresponding `skb` is again transmitted through a `veth` interface, then another softirq will be raised. As we are now in software interrupt context, they will not be recursively processed again after the packet transmission. However, the `do_softirq()` function will continue to process softirqs as they are raised. The loop will stop when either (i) more than two milliseconds elapsed, (ii) more than 10 iterations occurred, or (iii) a scheduling preemption event happened. This is another source of jitter, as the user application that initially generated the packet to be sent is not executed while softirqs are processed in chain.

Perhaps more importantly, such chain processing of softirqs can change the expected order of events. Consider the virtual network shown in Figure 3.5. A user space application running in namespace `A` sends two consecutive packets to `D`. The first packet is routed through `B` and the second one through `C`. The expected

[9] The recipient CPU is selected using, *e.g.*, Receive Side Scaling or Receive Flow Steering [50].

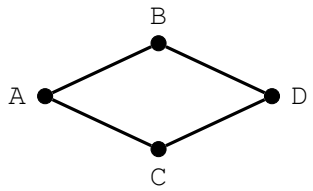


Figure 3.5: Virtual network composed of network namespaces and virtual Ethernet pairs.

order of events is the following, considering that the link delay and bandwidth are constant and identical for all the links in the network.

1. `A` sends packet P_1.
2. `A` sends packet P_2.
3. `B` forwards packet P_1.
4. `C` forwards packet P_2.
5. `D` receives packet P_1.
6. `D` receives packet P_2.

However, with chain processing of softirqs, the actual order of events is the following.

1. `A` sends packet P_1.
2. `B` forwards packet P_1.
3. `D` receives packet P_1.
4. `A` sends packet P_2.
5. `C` forwards packet P_2.
6. `D` receives packet P_2.

Currently, the only way to prevent the chain processing of softirqs is to enforce, on each link, a minimal queueing delay large enough so that the packets will not be transmitted on the first call of `__qdisc_run()`.[10] To address the

[10]Using, *e.g.*, the `netem` qdisc

jitter induced by softirqs processing, the Linux kernel includes a mitigation mechanism that is deployed when the condition to break from softirq processing happens (timeout, too much iterations, or preemption). This mechanism is a kernel thread called `ksoftirqd`. Its sole purpose is to process software interrupts. It is scheduled when the kernel estimates that too many softirqs are processed in a row. There is one `ksoftirqd` thread running for each CPU. While this mechanism enables to avoid spending the execution time of a user application in the processing of softirqs, the `ksoftirqd` thread will still be scheduled on the same CPU as the user application that initially triggered the softirq, eventually interrupting its execution and leading to unavoidable jitter. In the current state of the Linux kernel, there is no way around this issue. One solution is to use dedicated CPUs to handle the networking stack, thus avoiding interference with user applications scheduled on other CPUs. At the time of writing, this solution is currently being investigated as part of a larger effort to reduce latencies in the networking stack [51].

3.6 Performances

While virtualized environments can be used to assert the correct implementation of network features, they are less suited to performance measurements. For our SRv6 implementation to be deployed in production, we need to ensure that it yields good performances on real hardware.

3.6.1 Setup

Our setup consists of three identical machines, running Intel Xeon X3440 processors with 4 cores and 8 threads at 2.53 GHz. Each machine is equipped with 16 GB of RAM and two Intel 82599 10 Gbps network interface cards. The three machines are connected linearly. The first one is the packet generator (source). It is connected to another machine acting as an SR node, that will perform SR operations on packets. Itself is connected to the third one, the packet sink. Figure 3.6 shows an illustration of our setup, with interface names and link prefixes. The source has a route to `fc01::/64` via `fc00::5` and the sink has a route to `fc00::/64` via `fc01::5`.

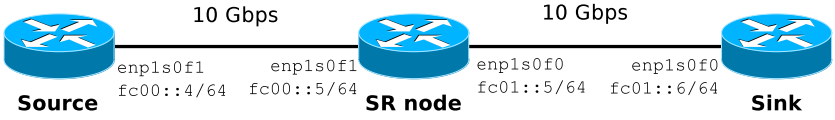


Figure 3.6: Network setup for performance measurements.

Each machine is running Linux 4.11-rc3.[11] All SR-related kernel options are enabled. The preemption model is voluntary and the clock ticks are set to 100 Hz periodic. All active measurement interfaces have GRO and GSO disabled, as well as all hardware transmit and receive offloading features. By default, each interface has one queue per CPU (*i.e.*, 8 queues in our setup). Each queue's IRQ is handled by the corresponding CPU. We change this setting to force all the queues of a network interface to be handled by a single CPU. The other interfaces parameters are left to their default setting.

Our source use the `pktgen` [52] packet generator. It has the particularity of being directly included in the kernel. As such, packets are directly handed over to the network driver, without further preprocessing. Consequently, `pktgen` is much faster than user space counterparts such as `iperf3`. A drawback is that `pktgen` is not able to handle streamed protocols such as TCP. It only sends UDP packets as fast as possible, with a configurable payload length. However, this is a suitable behavior for our tests. As our implementation adds per-packet processing overheads, we need to measure throughput in terms of packets per second rather than bytes per second.

As `pktgen` is able to generate packets an order of magnitude faster than what the kernel is able to process through its normal forwarding codepath, the SR node is always overloaded on its ingress interface. The actual throughput and performance of the SR node is measured on its egress interface.[12]

3.6.2 Measurements

We define the following types of measurements.

- **Plain:** regular IPv6 forwarding, without SRv6 interactions.
- **Encap:** SRH encapsulation (1 segment) with an outer IPv6 header
- **Inline:** direct SRH insertion (1 segment) in the original packet.
- **HMAC:** SRH encapsulation with an HMAC field (SHA-256).

The results shown further are represented as boxplots. Their format is defined as follows. At the core of each boxplot is a rectangle whose lower and upper bounds represent resp. the first and third quartiles (*i.e.*, the 25^{th} and 75^{th} percentiles). Within the rectangle is a horizontal line that marks the median value. Whiskers extend below and above the first and third quartiles. They are dashed vertical lines reaching down to the 5^{th} and up to the 95^{th} percentiles. The limits

[11] At commit `add641e7dee31b36aee83412c29e39dd1f5e0c9c`.

[12] By sampling `/sys/class/net/<iface>/statistics/tx_packets`

of the whiskers are marked by a horizontal line which is narrower than the median line. Outliers are marked by very short horizontal lines below and above the whiskers.

Single-CPU forwarding

Unless stated otherwise, for each measurement we run 100 batches of five millions IPv6 packets with a length of 64 bytes, which is the minimum length supported by `pktgen`. Each packet includes an IPv6 header (40 bytes), a UDP header (8 bytes) with source and destination port 9 (*discard* service) and a UDP payload of 16 bytes. The source IPv6 address is set to `fc00::44` (the source) and the destination address is set to `fc01::66` (the sink). The source MAC address is set to the `enp1s0f1` interface MAC on the source and the destination MAC address is set to the `enp1s0f1` interface MAC on the SR node. When the SR node inserts an SRH with one segment, the segment is defined to `fc01::6`.

In the first set of measurements, we compare the encapsulation and direct insertion cost with respect to the plain IPv6 forwarding performances. Figure 3.7 shows the result of those measurements. The baseline for plain IPv6 averages at 1,165 Kpps, while direct insertion and encapsulation average at resp. 776 and 784 Kpps. The standard deviation is about 7 Kpps for the three measurements. While those results are not catastrophic, they still could be improved by a factor of about 1.5 to reach the plain IPv6 forwarding performances.

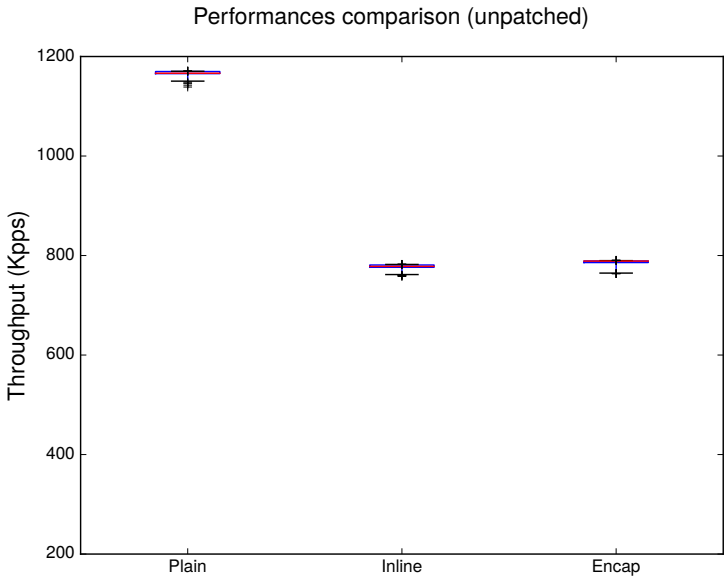


Figure 3.7: Initial performances for encapsulation and insertion.

To determine where the SRH insertion took time, we ran the `perf` tool on the SR node while transmitting packets and generated a differential analysis between plain IPv6 forwarding and SRH insertion. This analysis pinpointed two kernel functions, `fib6_lookup()` and `__slab_free()`. The former is the IPv6 route lookup function. We realized that it was called once too often per SR-processed packet. The reason was that we used the `dst_cache` mechanism for locally generated packets (in `seg6_output()`) but not for forwarded packets (in `seg6_input()`). We thus fixed the issue by leveraging the caching mechanism for both functions. The root cause of the `__slab_free()` increased usage was a little more difficult to determine. This function is called by `kfree()` (freeing kernel memory) when the data to free was not allocated by the same CPU as the one attempting to free the memory. In this case, a slowpath is taken which calls the `__slab_free()` function, itself taking a spinlock. This was likely due to some allocation we were performing in the SRH insertion codepath. Indeed, we were calling `pskb_expand_head()` for each packet. We used this function to increase the size of the `skb` headroom by the length of the IPv6 header and the SRH we were pushing onto the packet. However, the `skb` was originally allocated by the CPU that handled the hardware interrupt generated by the network card when receiving the packet. The CPU handling the `NET_RX` softirq and thus calling IPv6 and SR processing functions is not necessarily the same. As such, when the initial CPU attempted to free the `skb`, part of its data were reallocated by another CPU, and the freeing process took the slowpath through `__slab_free()`. To fix this issue, we took advantage of the already existing headroom in `skb`'s. We replaced `pskb_expand_head()` calls by `skb_cow_head()`, which reallocates the `skb` header only if the headroom is not large enough. After applying the two patches, we performed a second set of measurements, as shown in Figure 3.8. The performance gain is clearly noticeable, with direct insertion and encapsulation averaging resp. 1,019 Kpps and 1,001 Kpps (standard deviation resp. 11.1 Kpps and 7.8 Kpps).

In a third set of measurements, we analysed the effect of the packet size on SRH insertion performance. Instead of 64-byte packets, we generated 1000-byte packets. Figure 3.9 shows the results for plain, inline and encap measurements, with both packet sizes. As a general tendency, we can see that the performances for 1000-byte packets are slightly better, especially for peak throughput. The average is almost the same for 1000-byte and 64-byte packets, but the standard deviation is three times higher for large packets. In any case, we can say that the packet size does not significantly impact the SRH insertion performances.

A potentially harmful feature of SRv6, with respect to performances, is HMAC computation. In a fourth set of measurements, we analysed the impact of SRH encapsulation with an HMAC. The hashing algorithm is SHA-256. In our kernel version, there are three available implementations of SHA-256. A generic

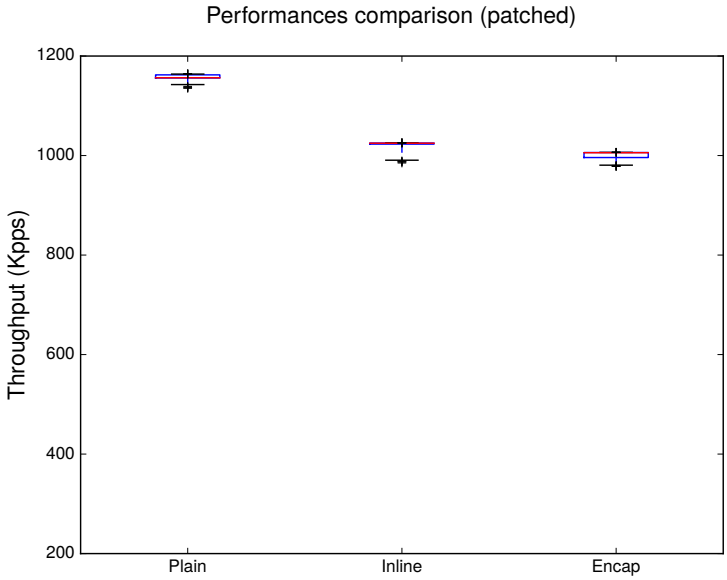


Figure 3.8: Performances for encapsulation and insertion after optimizations.

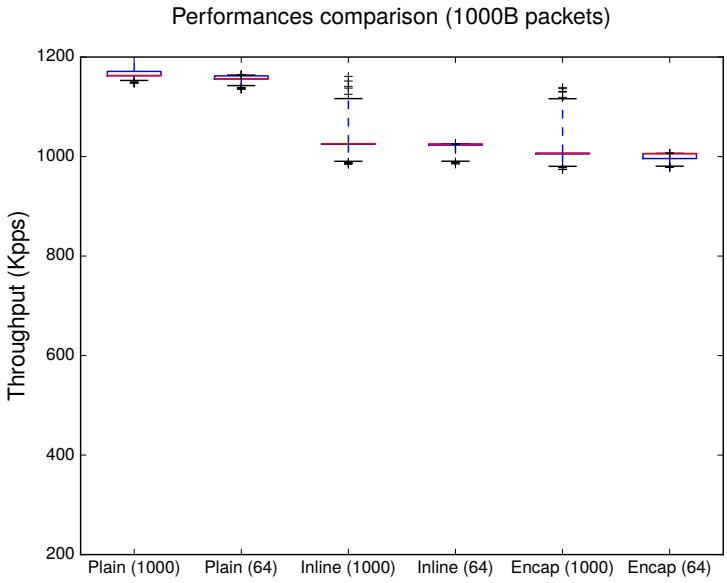


Figure 3.9: Performances for encapsulation and insertion using small and large packets.

one, a partially hardware-offloaded one and a experimental multi-buffer implementation. We could leverage the `ssse3` instruction set for the partial hardware-offloading implementation. We could not use the multi-buffer SHA-256 as the processor of our test machines was lacking the proper instruction set (`avx2`). The results are shown in Figure 3.10. The performance drop for HMAC is huge. Using the generic implementation of SHA-256, we average 240 Kpps. The `ssse3`-augmented version reaches 290 Kpps in average, which is a long way from the 1,001 Kpps baseline for encapsulation without HMAC. The standard deviation is below 1 for both SHA-256 implementations. This can be explained by the fact that the long computation time shadows the small random fluctuations in the packet processing time.

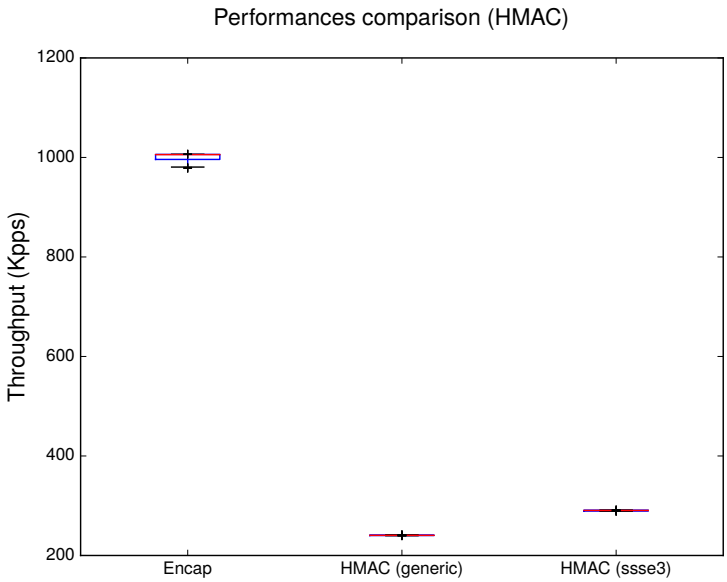


Figure 3.10: Performances for encapsulation with and without HMAC.

Figure 3.11 shows a summary of the performance measurements for SRH insertion with a single CPU. Table 3.2 describes the detailed statistic data for each performed measurement.

Forwarding at scale

To assess how our implementation performs when concurrently executed on multiple CPUs, we leveraged the Receive Side Scaling (RSS) feature of the Linux kernel. This feature enables the NIC to uniformly distribute incoming packets over its multiple receive queues, on a coarse per-flow basis (*i.e.*, at least the same

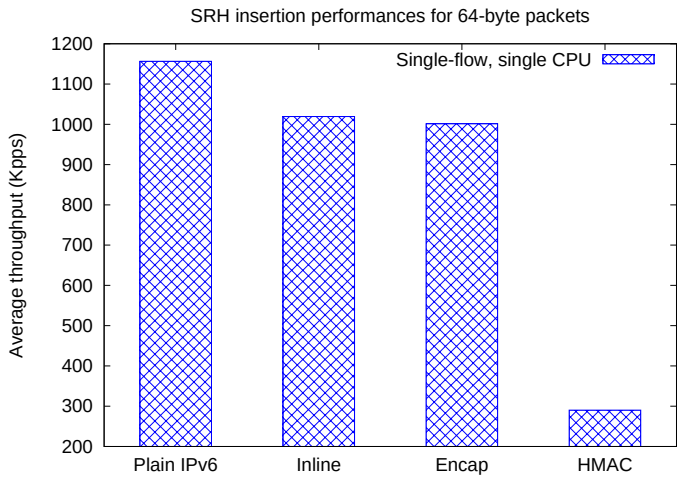


Figure 3.11: Performances summary for SRH encapsulation with a single CPU.

Table 3.2: Detailed statistics in thousands of packets per second (Kpps) for single-CPU measurements.

Measurement	Min	Max	Mean	Median	Stddev
plain-64	1135.547	1164.134	1156.462	1156.390	6.465
plain-1000	1147.429	1219.501	1169.196	1162.986	15.989
inline-64	984.974	1025.584	1019.391	1024.792	11.093
inline-1000	984.315	1161.050	1031.402	1025.601	33.422
encap-64	978.020	1006.785	1001.442	1005.676	7.807
encap-1000	973.969	1138.490	1017.841	1006.100	38.092
hmac-generic-64	239.557	241.657	240.497	240.474	0.512
hmac-ssse3-64	288.864	291.543	290.286	290.527	0.917

source and destination address). Each receive queue is associated with a dedicated IRQ. By assigning a different CPU for each IRQ of the queues, the packets can be processed in parallel. To realize this, each packet generated by `pktgen` was assigned a random destination address within the `fc01::/64` prefix, using a uniform distribution. With this modified setup, we replayed all four types of measurements. The results are shown in Figure 3.12. We can see that the performance scales sub-linearly with respect to the number of CPUs, even for plain IPv6 forwarding. This can be explained by the fact that the test machines have 4 physical cores, yielding 8 virtual CPUs with HyperThreading enabled. However, one core cannot execute more than one thread at the same time. Thus, the performance can improve at most by a factor equal to the number of cores. The performance gain is the same for each type of measurement, with inline and encap measurements reaching 4.2 Mpps. Table 3.3 shows the detailed statistics for the multi-CPUs measurements. Each figure has an improvement factor of about 4.2 with respect to the single-CPU measurements, which corresponds to the additional CPU cores. We conclude that the performance of our implementation scales linearly with respect to the available CPU power.

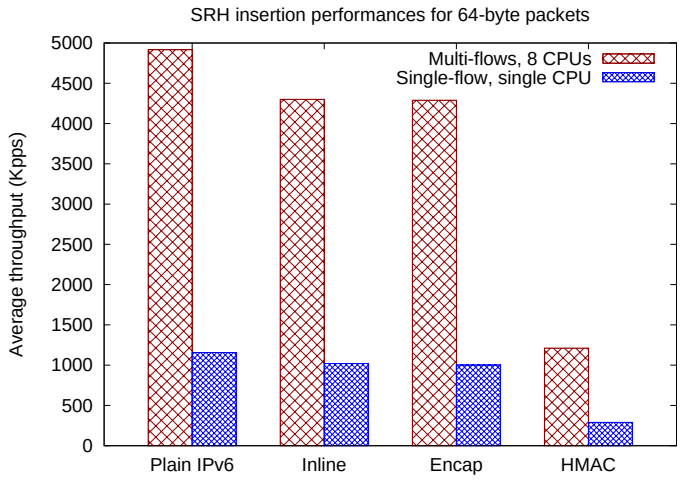


Figure 3.12: Performance summary of SRH encapsulation using one and multiple CPUs.

Table 3.3: Detailed statistics in Kpps for multiple CPUs measurements.

Measurement	Min	Max	Mean	Median	Stddev
mflows-plain-64	4883.166	4971.043	4917.450	4915.135	15.297
mflows-inline-64	4028.267	4320.840	4300.920	4305.048	28.667
mflows-encap-64	3731.864	4329.391	4289.581	4295.670	56.639
mflows-hmac-ssse3-64	1207.135	1215.387	1211.438	1211.453	1.816

3.7 Network programming support

After the writing of this manuscript, we implemented the latest SRv6 specifications as described in 2.2, which was subsequently merged into the mainline Linux tree.[13] In this section, we describe our implementation of SRv6 network programming in Linux. This is thus an update to the original manuscript.

The core idea of the SRv6 network programming specifications is that each SR node contains a *My Local SID* table. Each entry of this table is a segment mapped to a processing function. When a packet enters the SR node with an active segment matching an entry of the table, the associated function is applied to the packet. The specifications define a list of functions that each SR node must support.

Two key differences with the original implementation is that, in the network programming principles, (i) a packet may contain a valid active segment which is not attached to any interface of the corresponding SR node, and (ii) in some encapsulation cases, a packet may contain a valid active segment *without* an SRH. The consequence of those differences is that we must be able to match ingress packets based on their destination address (which contains the active segment), instead of catching them later in the `INPUT` stage. Such a requirement very much resembles a routing table. Furthermore, there already exists a kernel mechanism to associate custom functions to routing entries, the *lightweight tunnels*. We already use LWTs to support SRH insertion and encapsulation. To support SRv6 network programming, we define a new type of LWT, namely `seg6local`.

Each instance of a `seg6local` LWT is defined by an `action` parameter, defining the SR function to apply, and a set of arguments for the function. The type of arguments currently supported are routing table identifier, IPv6 next-hop, IPv4 next-hop, Segment Routing Header, ingress interface and egress interface. The meaning of those arguments depend on the SR function. We implemented the following functions, as described in [29].

- `End`. Regular processing of an SR-enabled packet as an intermediate segment endpoint..

[13] Those new features will be available for Linux 4.14, which will be released around November 2017.

- `End.X`. Apply the `End` function, but forward the processed packet to a specified IPv6 next-hop.
- `End.T`. Apply the `End` function, but lookup the IPv6 next-hop into the specified routing table.
- `End.B6`. Insert the specified SRH immediately after the IPv6 header of the packet. The destination address is set to the first segment of the list, and the original SRH is left unmodified.
- `End.B6.Encaps`. Apply the `End` function. Then, encapsulate the processed packet within an outer IPv6 header that contains the specified SRH.
- `End.DX2`. Remove the outer IPv6 header. The inner payload must be an Ethernet frame, which is forwarded on the specified egress interface.
- `End.DX6`. Remove the outer IPv6 header. The inner payload must be an IPv6 packet, which is forwarded to the specified IPv6 next-hop.
- `End.DX4`. Remove the outer IPv6 header. The inner payload must be an IPv4 packet, which is forwarded to the specified IPv4 next-hop.
- `End.DT6`. Remove the outer IPv6 header. The inner payload must be an IPv6 packet. The next-hop is selected by looking into the specified routing table.

Note that `End.D*` functions accept only packets that either contain an SRH whose `Segments Left` value is zero, or that do not contain an SRH. The rest of the functions accept only packets that contain an SRH whose `Segments Left` value is non-zero.

3.8 Related and future work

To the extent of our knowledge, this is the only open-source implementation of Segment Routing that supports both endhosts and router functionalities. The `fd.io` project has recently announced another implementation that focuses on router functionalities [53]. In the mainline Linux kernel, two other subsystems leverage the lightweight tunnels infrastructure. Those subsystems are the MPLS [54] and the Identifier Locator Addressing [55, 56] implementations. With respect to work related to Nanonet, multiple open-source network emulators have been developed. Prominently, Mininet [42] also leverages network namespaces and virtual Ethernet pairs to emulate network topologies. They

also leverage the OpenVSwitch implementation to emulate OpenFlow networks. Mininet-WiFi [57] is an extension to Mininet that enables to emulate wireless stations and access points. Other emulators that leverage the Linux networking stack also provides configuration helpers such as a graphical interface or Python bindings [58, 59]. Netkit [60] leverages User-Mode Linux [61] to emulate network nodes as instances of the Linux kernel running in user space. The well-known GNS3 [62] emulator focuses on emulating Cisco IOS images using Dynamips [63]. Regarding the limits of same-kernel testing, the idea of having CPUs dedicated to the network stack is perhaps similar to what is presented in [64]. Hruby *et al.* propose an architecture for MINIX where the network stack is independently and concurrently executed by multiple threads.

For future work, multiple aspects of the SRv6 implementation can be extended. For example, some of the network programming functions are, at the time of writing, not yet implemented. Those functions include, *e.g.*, MPLS encapsulation, forwarding to SR-unaware and SR-aware virtual functions, etc. The encapsulation process can be optimized by enabling a single copy of an SRH to be mapped to multiple routes. That would require to maintain an additional table containing the pre-defined SRHs, each associated to an index. This index could then be referenced in the `iproute2` command. The HMAC subsystem can benefit from the addition of new hashing algorithms. It was implemented specifically for SRv6. However, other network protocols may benefit from such an HMAC mechanism, such as Generic UDP Encapsulation [65]. To realize this, the current HMAC implementation must be generalized to support any type of network protocol. From an endhost point of view, new features such as the ability to extract the SRH attached to TCP or UDP packets could enable applications to better take advantage of per-flow SRHs.

Finally, the overall modularity and flexibility of our implementation can be improved by providing eBPF hooks at various places in the code. Such hooks would enable fine-grained configuration and alteration of the SR processing functions while requiring no changes in the kernel code.

3.9 Conclusion

In this chapter, we presented our implementation of IPv6 Segment Routing in the Linux kernel. We explained how the SRH processing, insertion, and HMAC computation were implemented, as well as the user space control mechanisms. We discussed some of the difficulties we encountered during the implementation. We presented Nanonet, our testing framework, and presented the limits of network testing on the same Linux kernel. Then, we measured the performances of our implementation through several sets of measurements. After a few optimizations,

we showed that SRH insertion reaches almost 90% of the baseline plain IPv6 forwarding performances. We also showed that HMAC computation was extremely costly, reaching only 25% of the baseline. A possible optimization would be to implement an HMAC cache to prevent recomputing the HMAC for each packet. Finally, we gave some guidelines to further extend our SRv6 implementation.

Chapter 4

Exploring IPv6 Segment Routing

Due to its ability to easily specify arbitrary paths for packets, the Segment Routing architecture can yield benefits in various aspects of networking. In this chapter, we explore two of those aspects. First, we explore how real-time applications can benefit from traffic duplication over disjoint paths to achieve low-latency communications in a reliable and robust manner [66]. Through simulations, we analyze how the Linux TCP stack reacts to duplicated traffic and how traffic duplication can absorb link delays and packet losses. Second, we leverage Segment Routing to improve network monitoring [67]. Using segments, we propose SCMon, a technique to send monitoring probes over cycles in the network, from a single vantage point. The monitoring cycles are computed by algorithms optimising the network coverage with trade-offs between the length of cycles and the number of segments. This technique enables to deterministically explore ECMP components and to pinpoint failures on individual links within a bundle. We implement SCMon and evaluate its performance on emulated topologies. In exploring those aspects, we also leverage our SRv6 implementation presented in Chapter 3.

The work presented in this chapter was realized jointly with the Belgian Constraints Group in our department. They focused on the algorithmic parts while we focused on the networking parts. As such, we only briefly present the optimization algorithms used in this chapter. The full details are available in the corresponding papers.

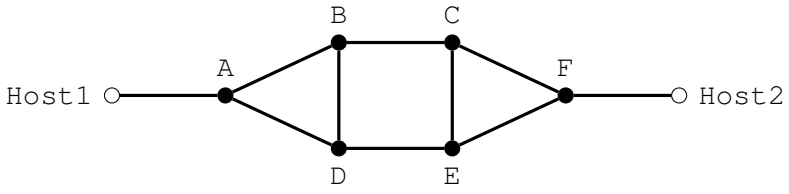
4.1 Traffic duplication for latency-critical applications

While low latency requirements are common for applications such as voice over IP, live streaming, online gaming, etc., ultra-low latency is a different world.

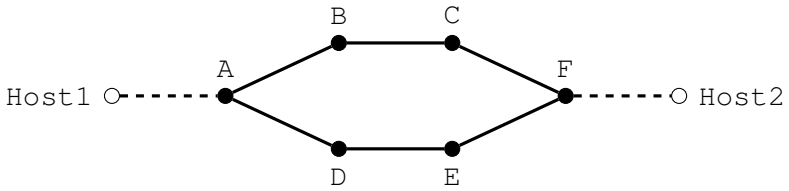
Safety-critical applications such as remote surgery cannot tolerate network events such as packet loss and high jitter, and commands must be delivered as soon as possible. In another area, high-frequency trading is also a heavy user of ultra-low latency [68]. Financial companies have a desperate need to exchange information and trading orders as fast as possible. Indeed, a 1-millisecond advantage for a major brokerage firm can yield a loss or a profit of $100 million a year [69]. Those applications require not only ultra-low latency communication channels but also transmissions that are reliable and robust to corruptions. Besides telesurgery and trading, multiple other applications also require low latencies and reliable deliveries [70, 71].

Some optical networks support two kinds of path protection schemes: $1:1$ and $1+1$ [72, 73]. Both schemes leverages disjoint paths. With $1:1$ protection, data is sent over a primary link, while a backup link stands ready to take over should the primary link fails. With $1+1$ protection, data is sent over both paths and the destination can immediately switch to another path in case of link failure. However, TCP/IP stacks do not natively support such path protection mechanisms. In this section, we present a solution that achieves $1+1$ protection by duplicating TCP traffic over multiple disjoint paths. First, we discuss how the multiple paths should be computed. Then, we describe our extension of the SRv6 Linux implementation to support traffic duplication. We evaluate the impact of traffic duplication on the Linux TCP stack. Finally, we briefly discuss existing related work.

4.1.1 Duplication over segmented disjoint paths



(a) Network topology connecting two hosts with ultra-low latency requirements.



(b) Two link disjoint paths exist, connecting the gateways of both hosts.

Figure 4.1: Illustration of disjoint paths.

An obvious requirement to achieve 1 + 1 path protection is path disjointness. Otherwise, there would be a Single Point Of Failure (SPOF) on the path. Also, the same data would be transmitted multiple times on the non-disjoint part of the paths, which would be a waste of resources. Figure 4.1 shows an illustration of two hosts connected with two disjoint paths. In this topology, the traffic would be duplicated along the north (`A-B-C-F`) and south (`A-D-E-F`) paths. The duplication can be performed either by the host or by its gateway. The duplicated traffic then converges at the remote gateway and is forwarded to the peer host. The networking stack of the host receiving duplicated traffic then needs to properly handle the duplicates. Such a setup would ensure that the fastest path is always used and provide robustness against adverse events such as packet loss, jitter, etc. on a single link.

A set of algorithms to compute disjoint paths and to realize them by using segments are presented in [66]. We provide a brief summary of those algorithms.

Their goal is to compute an arbitrary number of link disjoint paths between a given pair of nodes in a graph. Those paths must be expressed by a list of segments. The following parameters must be minimized. The latency of each path, the number of segments used to express them, and the difference of latencies between each computed disjoint path. For the path segmentation, only node segments are considered. As such, not all paths are segmentable. In this context, a path is segmentable if and only if all its edges belong to at least one shortest path Directed Acyclic Graph (SP-DAG) rooted at a node of the graph. For example, paths that contain a backup link which is not part of any SP-DAG are said to be *unsegmentable*. Traversing such links would require an adjacency segment. A path expressible with a list of k segments is said to be k-segmentable.

The first step of computing k-segmentable link disjoint paths is handled by the Build-Graph algorithm. This algorithm takes as input the graph $G = (V, E)$ of the network, the maximum number of segments K, a link latency function $l : E \to \mathbb{R}^+$, and the SP-DAGs $\mathcal{D}_v$ for all $v \in V$. The output is a graph $\overline{G}$ where each node is a tuple $(v, \mathcal{D}_r, k)$, with $v, r \in V$ and $k \in \{1, \ldots, K\}$. Two nodes $(v_1, \mathcal{D}_{r_1}, k_1)$ and $(v_2, \mathcal{D}_{r_2}, k_2)$ in $\overline{G}$ are connected by an edge of cost $l(v_1, v_2)$ if either, $(v_1, v_2) \in \mathcal{D}_{r_1}$ and $r_2 = r_1$ and $k_2 = k_1$, or $(v_1, v_2) \in \mathcal{D}_{v_1}$ and $r_2 = v_1$ and $k_2 = k_1 + 1 \leq K$. As shown in [66], it is possible to find a shortest-latency path on G that requires at most K segments by computing a shortest path in $\overline{G}$. This second step is realized by the SPS algorithm. It takes as input the graph $\overline{G}$, two nodes $s, t \in V(G)$, and the maximum number of segments K. Its output is the shortest-latency K-segmentable $s - t$ path.

The third step consists in iteratively applying the SPS algorithm. Each time a path is found, its edges are removed from the graph. The iterations stop when no more path can be found. To evaluate how these algorithms would perform on real topologies, we ran simulations on four Rocketfuel topologies [74] described

Table 4.1: Topologies used to evaluate link disjoint path computation.

Topology	Nodes	Links
RF AS1239	153	1010
RF AS1755	67	248
RF AS3257	103	484
RF AS3967	57	208

in Table 4.1. For each topology and for each possible pair of nodes, we generated an increasing number of link disjoint paths between the two nodes of the pair. The maximum number of segments was limited to $K = 3$. The results are shown in Figure 4.2. It was possible to find a single path between each pair of nodes for all topologies (*i.e.*, all nodes are part of a shortest path). Then, as the number of requested link disjoint paths increases, the number of pair of nodes satisfying the constraint decreases. The smaller is the network, the larger is the decrease. Indeed, it is more likely to find a higher number of link disjoint paths between two pair of nodes if the network has a large number of nodes and edges.

4.1.2 Implementation and evaluation

Implementation

To support traffic duplication, we modified the SRv6 lightweight tunnel state to support multiple SRHs instead of a single one. The SRHs are built by the userspace and transferred as a contiguous stack. The lightweight tunnel creation function parses this stack and saves the offsets of each SRH. In addition to the SRHs, the tunnel state also stores one `dst_cache` entry per SRH. Finally, we modified the `seg6_input` function to support duplication for forwarded packets. If more than one SRH is present in the SRH stack contained in the tunnel state, then we iterate over them, starting from the first one, and we apply the SRH processing function to a copy of the original `skb`, using the current SRH.

Evaluation

To evaluate the impact of traffic duplication on TCP, we designed a setup based on the network shown in Figure 4.1b. Nodes `Host1` and `Host2` exchange transactions over TCP. `Host1` acts as a client and sends 100-byte requests to `Host2`. Once `Host2` receives a request, it transmits a 100KB response. The content of each request and response is randomized. Both the client and server use the CUBIC congestion control algorithm with default parameters. The Nagle algorithm is disabled on both sides. The completion time of each transaction is measured on the client (`Host1`) side, starting from the emission of the request and until

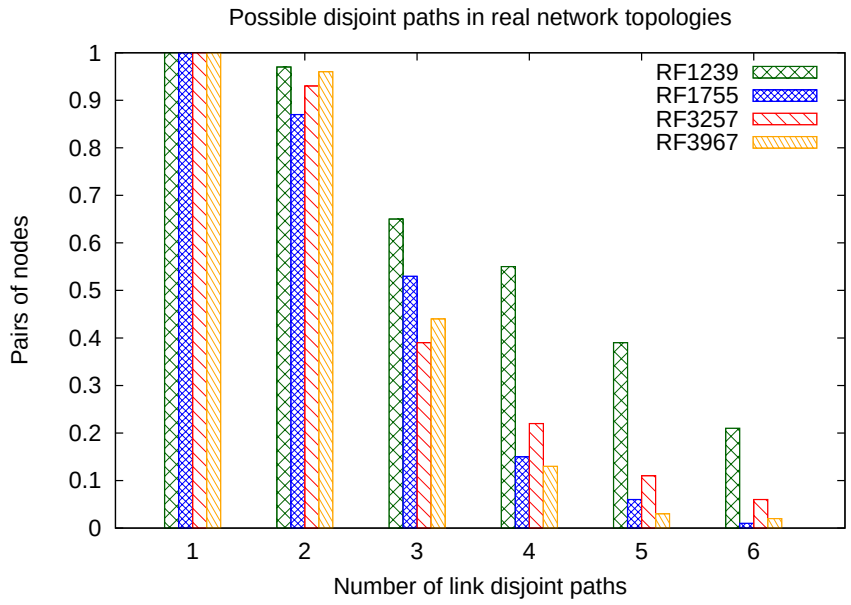


Figure 4.2: Number of link disjoint paths between each pair of nodes in several RocketFuel topologies, for $K = 3$.

the reception of the full response. As such, three-way handshakes and connection terminations are not included in the measurements. When duplication is enabled, nodes `A` and `F` act as traffic duplicators. Node `A` duplicates all packets sent to `Host2` over the `A - B` and `A - D` links. Conversely, node `F` duplicates all packets sent to `Host1` over the `C - F` and `E - F` links.

Due to hardware constraints, we deployed this network setup in a virtualized environment. However, we must be careful in the setup of such an environment. Due to the ordering issues with same-kernel testing explained in Section 3.5.2, we choose not to use Nanonet or other namespace-based environment to evaluate the impact of traffic duplication. Indeed, we assume that our duplicated packets will be transmitted and handled in a parallel, or at least interleaved, fashion. A straightforward way to realize this is to dedicate one CPU per networking stack (*i.e.*, virtual node). However, this cannot be achieved with same-kernel testing. Thus, we deployed a `qemu-kvm` based testbed [75, 76]. Each node in the network is a kvm virtual machine. The nodes are connected through emulated `e1000` network interface cards, bridged by the host, using one bridge for each virtual link. To maximize parallelism and minimize interference, each virtual machine

is pinned to a single CPU.[1] While this setup can still suffer from jitter generated on the host, it is more realistic than a namespace-based setup. To reduce the impact of jitter, links `Host1 -> A` and `Host2 -> F` are configured with a unidirectional delay of 5 ms, using the `netem` module. Thus, the round-trip time between `Host1` and `Host2` is 10 ms.

We performed multiple batches of measurements. Each batch is realized over 10,000 transactions with a particular network setup. The distribution of losses over lossy links is random and uncorrelated. We used five different setups.

1. **nodup-rtt10**: Regular transactions over one path with 10 ms RTT.
2. **dup-rtt10**: Duplicated transactions over two paths with 10 ms RTT each.
3. **nodup-rtt20**: Regular transactions over one path with 20 ms RTT.
4. **nodup-loss10**: Regular transactions over one path with 10 ms RTT and 10% losses.
5. **dup-loss10-rtt20**: Duplicated transactions over the two previous paths, *i.e.*, one with 20 ms RTT and without losses, and another one with 10 ms RTT and 10% losses.

The first two batches compare the impact of duplication over two homogeneous paths with respect to the non-duplicated transactions. The set of non-duplicated transactions over a path with 10 ms RTT yielded a median completion time of 56.009 ms, with a standard deviation of 3.414. The set of duplicated transactions over two homogeneous paths with 10 ms RTT each yielded a median completion time of 56.867 ms, with a standard deviation of 3.24. The results are shown in Figure 4.3. For most of the transactions, the completion time is about 0.8 milliseconds higher for duplicated transactions than non-duplicated transactions.

Then, we analyze whether traffic duplication over two faulty paths can yield better performances than the non-duplicated versions over each individual path. To realize this, we measure a first set of non-duplicated transactions over a path with 20 ms RTT, and a second set of non-duplicated transactions over another path with 10 ms RTT and 10% packet loss. In other words, the second path is twice as fast as the first one but suffers from high packet loss, while the first path is lossless but slower. In the final batch of measurements, we run duplicated transactions over both paths. The network setup for these three batches is shown in Figure 4.4. The results of those measurements are shown in Figure 4.5. We can see that while the lossless path yields stable transaction times, the lossy path is much more volatile. About 40% of the transactions performed over the lossy

[1] The CPU affinity of a process can be defined using the `taskset` command.

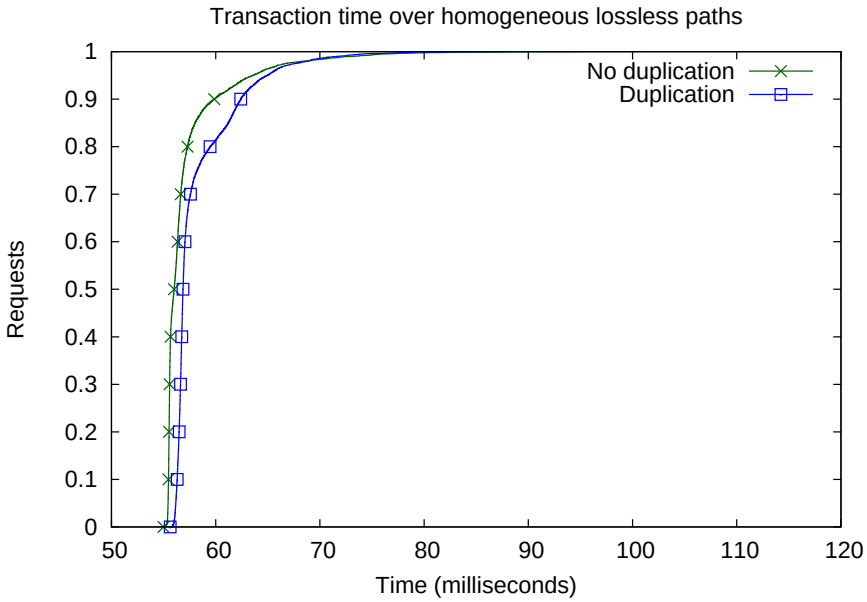


Figure 4.3: Measurements of non-duplicated and duplicated transactions over homogeneous lossless paths.

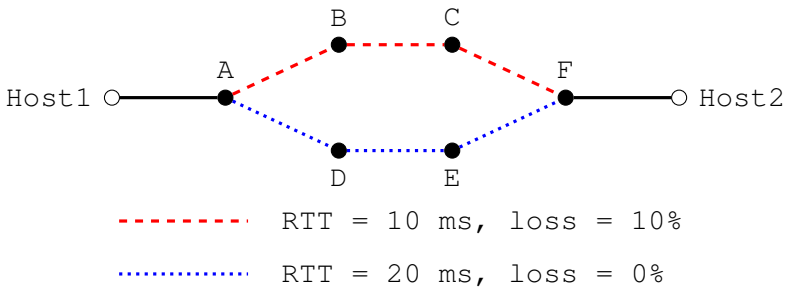


Figure 4.4: Network setup for transactions over heterogeneous paths.

path have faster completion time than the average transaction over the lossless (but slower) path. However, it escalates quickly, with 20% of the transactions taking more than one second to complete. In contrast, the duplication of the transactions over both paths yields completion times that are consistently better than when performed individually over each path. Table 4.2 shows the detailed statistic data for each measurement batch. The high maximum values shown for transactions over lossless paths are due to the unpredictable jitter generated on the host of the virtual machines.

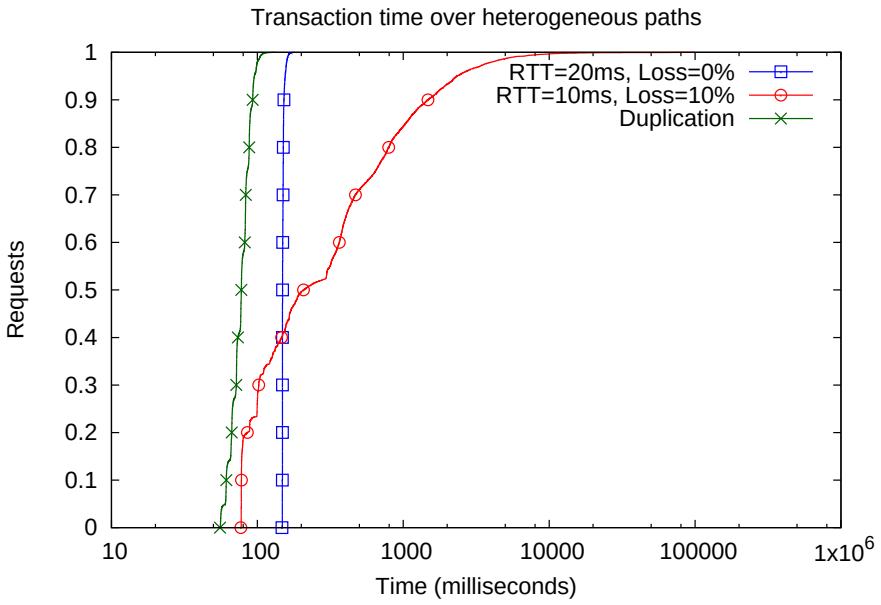
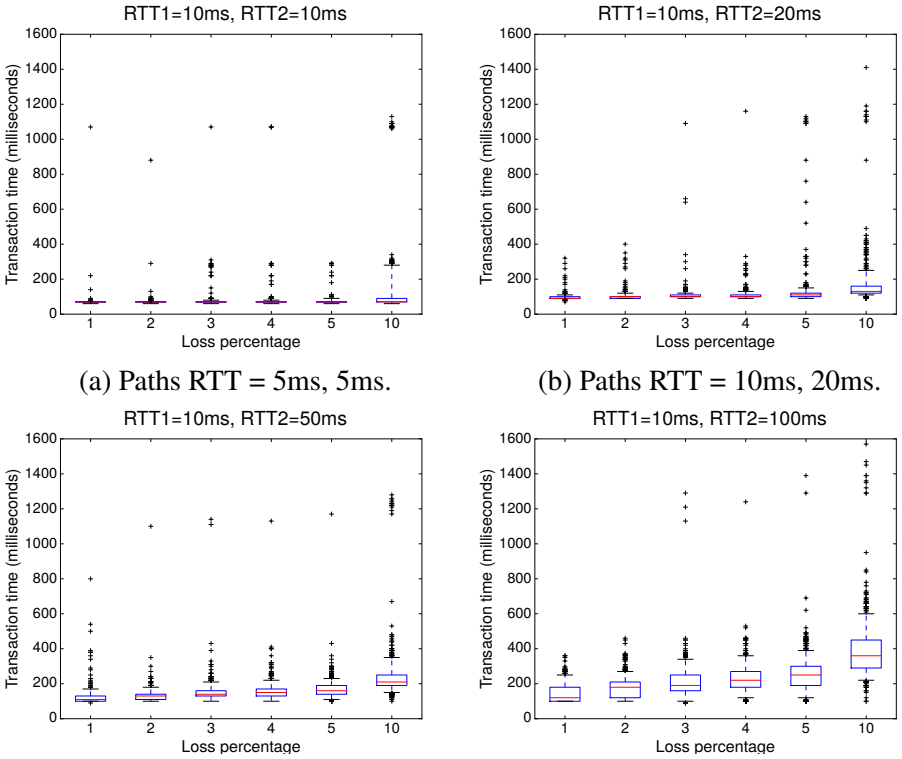


Figure 4.5: Measurements of non-duplicated and duplicated transactions over heterogeneous paths.

Table 4.2: Detailed measurement statistics in milliseconds (transaction time).

Batch	Min	Max	Mean	Median	Stddev
nodup-rtt10	54.998	119.327	57.103	56.009	3.414
dup-rtt10	55.638	87.023	58.239	56.867	3.240
nodup-rtt20	147.522	180.340	149.900	148.787	2.628
nodup-loss10	77.004	100344.465	665.269	207.210	2284.182
dup-loss10-rtt20	55.531	133.357	77.449	77.651	11.458



(a) Paths RTT = 5ms, 5ms.

(b) Paths RTT = 10ms, 20ms.

(c) Paths RTT = 10ms, 50ms.

(d) Paths RTT = 10ms, 100ms.

Figure 4.6: Distribution of the completion time for requests duplicated over two lossy links, for several loss percentages. Each subfigure has different delay parameters for the two paths.

In a second set of experiments, we analyzed how losses on both links, and delay differences, would impact traffic duplication. We use the network setup as shown in Figure 4.4 and we modify the loss and delay parameters of the two disjoint paths between A and F. We denote by $\texttt{RTT}_1$ the round-trip time of the northern path (A – B – C – F) and by $\texttt{RTT}_2$ the round-trip time of the southern path (A – D – E – F). We used the following four delay configurations.

1. $\texttt{RTT}_1$ = 10 ms, $\texttt{RTT}_2$ = 10 ms (ΔRTT = 0 ms)
2. $\texttt{RTT}_1$ = 10 ms, $\texttt{RTT}_2$ = 20 ms (ΔRTT = 10 ms)
3. $\texttt{RTT}_1$ = 10 ms, $\texttt{RTT}_2$ = 50 ms (ΔRTT = 40 ms)
4. $\texttt{RTT}_1$ = 10 ms, $\texttt{RTT}_2$ = 100 ms (ΔRTT = 90 ms)

Then, for each delay configuration, we run 10,000 transactions in six packet losses setups. For each packet loss setup, the loss rate is set to the same value on both paths, and the losses are random and uncorrelated. We used the following loss rate values: $1, 2, 3, 4, 5, 10$. The results are shown in Figure 4.6. The semantics of the boxplots are the same as in Section 3.6.2. When the RTT is the same on both paths (top left figure), small loss rates have little impact on the transaction time. Up to 5% of losses, the 95^{th} percentile of the transaction time is below 100 milliseconds. At 10% losses, the increase in the transaction time is more noticeable, with the 95^{th} percentile at 300 milliseconds. As the delay difference increases (top right and bottom left figures), the dispersion of the transaction times increases and the effect of packet losses is more visible. When the ΔRTT reaches 90 milliseconds, even small loss rates have a significant impact on the transaction time. At 10% losses, the 95^{th} percentile extends up to 600 milliseconds.

4.1.3 Related work

The *Low latency via Replication* proposal [70] is close to our work. Vulimiri *et al.* propose to replicate packets over diverse resources and evaluate how replication affects response under load. However, their approach differs in the sense that they use replication to query different systems such as DNS, while our solution uses multiple disjoint paths.

Another type of approach is to rely on Forward Error Correction to protect reliable services from the impact of losses. Such techniques have been used in a variety of networks including ATM [77], wireless [78], multicast [79] and inter-datacenter networks [80]. The closest to our work is Balakrishnan *et al.* who propose in [80] a Forward Error Correction (FEC) mechanism to recover from bursty losses. They propose to install this FEC mechanism on wide-area links

that interconnect datacenters. Compared to replication, a FEC approach has the benefit of a lower bandwidth consumption at the expense of a higher CPU load and possibly a higher latency.

4.1.4 Conclusion

In this section, we exposed a traffic duplication principle to ensure the fastest delivery of data to latency-critical applications. This duplication is performed over precomputed disjoint paths, enforced using the IPv6 Segment Routing data plane. We described our extension to the SRv6 implementation in the Linux kernel, enabling the support of traffic duplication. Then, we analyzed the impact of such a duplication on the Linux TCP stack for relatively small transactions. This analysis involved duplicating traffic over two heterogeneous paths with different delays and packet loss rates, in a virtualized environment. Our measurements showed that while duplication over homogeneous, lossless paths yields a small overhead over non-duplicated traffic, duplication over heterogeneous paths is highly beneficial, especially in the presence of packet loss on a single path. When both paths are lossy, duplicating the traffic enables to absorb the losses and yields transaction times that are close to what would yield a lossless path. As the difference of delays between the disjoint paths increases, the impact of losses is more significant.

4.2 Fine-grained network monitoring with SCMon

Monitoring is a crucial task for network operations. It is needed to ensure that all network elements operate correctly and behave according to the operator's intended configuration. Effective monitoring is also a valuable tool for maintenance and troubleshooting. Unfortunately, even basic monitoring tasks, like checking for hardware malfunctions, are practically hard, due to the complexity of current networks. In current networks, multi-path routing is widely used, both to spread the load on multiple paths and aggregate parallel links in bundles. While enabling better performance and robustness, multi-path routing also poses significant obstacles to monitoring [81]. For instance, assessing the exact path and performance of each packet becomes complex [82, 83] since such a path depends on (vendor-specific) hash functions used by routers for load-balancing. As a consequence, not only naive approaches (*e.g.*, based on `ping` or `traceroute`) are not sufficient, but also state-of-the-art monitoring techniques tend to be ineffective.

On the one hand, protocol-based approaches use control-plane messages to infer possible failures. For example, link-state routing protocols (like OSPF or IS-IS) or specialized ones (BFD [84]) rely on heartbeat-like mechanisms to check bi-directional connectivity among pairs of adjacent nodes. This approach only en-

sures detection of failures that affect control-plane messages. However, it cannot be used to detect failures that only affect data-plane traffic like: (i) corruption of an optical link that leads to framing errors and packet losses, (ii) dysfunction of a router interface that considers the link still up but discards all the received packets, and (iii) failure of only one link among a bundle between two routers.

On the other hand, probe-based techniques rely on sending data-plane monitoring packets, *i.e.*, *probes*, between fixed vantage points in the network. Vantage points typically run standard protocols (*e.g.*, IPSLA [85]) to send probes and extract measurements from them. Unfortunately, if the probes are sent over paths used to forward regular traffic, many vantage points may be needed to obtain high coverage, and links not used by current paths (*e.g.*, backup links) cannot be checked at all. Otherwise, probes can be sent over tunnels (*e.g.*, RSVP-TE [13] ones) to enforce specific paths, but this is not scalable. Indeed, even for detecting single-link failures and pinpointing their position, the number of needed tunnels tends to explode, and so does the control-plane overhead (to signal tunnels) [86].

In this section we present *SCMon*, a monitoring technique that ensures the full coverage of all network resources, from a single vantage point. This coverage is realized with data plane probes sent over precomputed cycles, which are enforced thanks to the Segment Routing data plane. Using cycles enables a single box to send and receive monitoring probes, removing the need to synchronize multiple vantage points. First, we explain how the cycles must be computed to yield an efficient coverage of the network. We will briefly describe some algorithms involved in the cycles computation. The details of those algorithm are described in [67]. Then, we present the design and implementation of a monitoring tool that leverages the SCMon principles. Finally, we evaluate the effectiveness of this tool by monitoring real topologies in a virtualized environment.

4.2.1 Network coverage with segmented cycles

Using cycles for probe-based monitoring has the main advantage of requiring a single vantage point. Consider the Abilene network shown in Figure 4.7. Using `KSCY` as the vantage point, we can use four partially overlapping cycles to cover all the edges and nodes in the network. Figure 4.8 illustrates this coverage. A fundamental trade-off in cycle coverage exists between the number of cycles and their average length. Fewer cycles implies longer cycles and conversely. The Segment Routing dataplane imposes an inherent limit to the length of each cycle. Indeed, each cycle will be represented by a list of segments. In most cases, the number of segments used to express a cycle will be proportional to its length. As each segment adds 16 bytes in the packet header, we impose an upper bound to the number of segments that can be used for each cycle. We call this upper bound the segment budget.

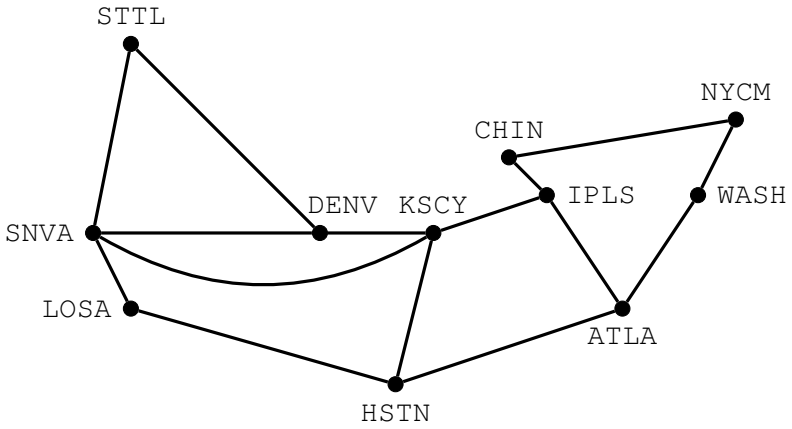
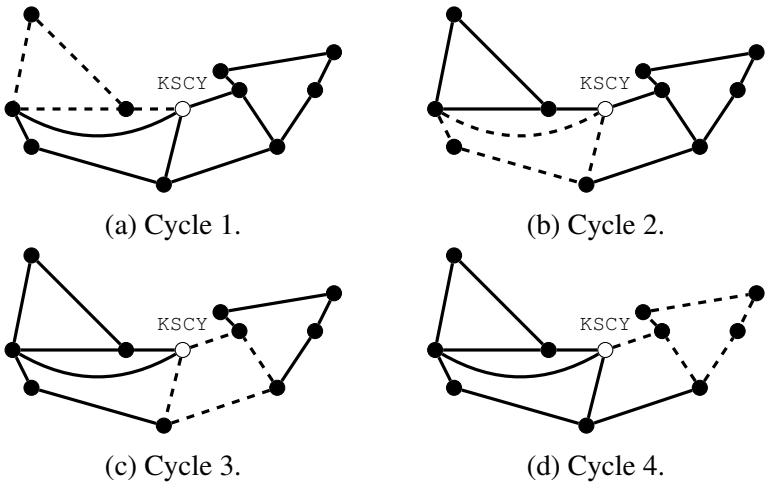


Figure 4.7: Abilene network.



(a) Cycle 1. (b) Cycle 2.

(c) Cycle 3. (d) Cycle 4.

Figure 4.8: Cycle coverage for the Abilene network.

With SR, a given path can be expressed with different equivalent segmentations. Let us consider the Abilene network in Figure 4.7, with unit weights. The path $p = (\texttt{LOSA}, \texttt{HSTN}, \texttt{ATLA})$ can be expressed with either $\langle \texttt{HSTN}, \texttt{ATLA} \rangle$ or $\langle \texttt{ATLA} \rangle$, assuming LOSA as the starting node. Both segmentations express the same path. The 2-segment expression contains redundant information that is already present in the shortest-path DAG. As each segment costs header space, we would like to use them only if necessary. As such, we require the segmentation of our cycles to be *minimal*.

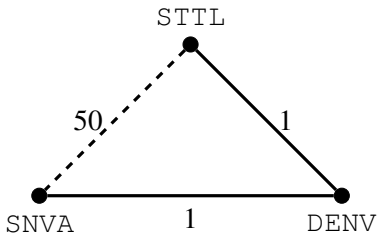


Figure 4.9: Backup link.

Some links may not be reachable from any shortest-path DAG. In Figure 4.9, consider the link between nodes SNVA and STTL. It is not part of any shortest path, as its cost (50) is larger than the cost of a detour through DENV (2). In this case, an adjacency segment must be used to force the traversal of the backup link. For example, to cover this network with a cycle starting at DENV, we would first need a node segment to SNVA, then an adjacency segment to force the traversal of the link to STTL, and finally a node segment to return to DENV. Formally, this would translate as $\langle \texttt{SNVA}, (\overrightarrow{\texttt{SNVA}, \texttt{STTL}}), \texttt{DENV} \rangle$. Using an adjacency segment is costly, as two segments are actually needed. First, a node segment to reach the node connected to the link to traverse, then the actual adjacency segment to traverse the link. As such, we require the segmentation of our cycles to be *simple*, *i.e.*, using only node segments, whenever possible.

Although equal-cost multi-path is widespread in current networks [82], monitoring cycles should avoid taking such paths in the same way as regular packets do, *i.e.*, by being load balanced by the routers. Indeed, if a failure occurs in an equal-cost section of the cycle, the failure will not be detected until a probe is sent over the failing path, as illustrated in Figure 4.10. Such a steering depends on the hashing function implemented on the routers. To ensure a full network coverage at all times, we want to avoid ECMP. One way to realize this is to use additional segments to break ECMP. However, such a solution can quickly consume a large amount of header space in topologies with a lot of ECMP, such as grid-like topologies. To ensure a minimal amount of ECMP, we propose to create a *monitoring topology* in addition to the one used for user traffic. Existing routers have already

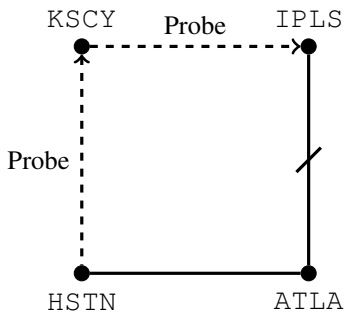


Figure 4.10: Link failure part of an equal-cost multi-path.

been shown to correctly support multiple topologies [87]. The algorithm to generate the monitoring topology assigns link weights by leveraging the properties of prime numbers and logarithms to ensure that the resulting topology has a minimum amount of ECMP. The full details of the algorithm are available in [67]. If ECMP is still present in the monitoring topology, then additional segments will be used to break it.

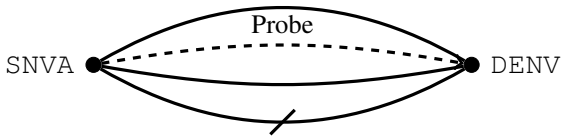


Figure 4.11: Link failure within a bundle.

A last network component that leverages parallel links and that cannot be dissociated by creating a monitoring topology is link bundles. Those are parallel physical links connecting two nodes, aggregated in a single logical link. By default, there is no guarantee that a probe will detect a link failure in such a bundle, as shown in Figure 4.11. To force the traversal of a particular link within a bundle, we have no choice but to use an appropriate adjacency segment. We assume that each link has a layer-3 identifier that can be used as a segment, such as an address in the `fe80::/64` space. If no such identifier is available, some local segments can be configured to steer traffic through given bundle links. This enables the monitoring of purely layer-2 links.

As a summary, we require the cycles to have the following properties.

- **Minimal**: the cycles should be constructed with the minimum number of segments, to save header space.
- **Simple**: the cycles segmentation should use only node segments whenever possible, as adjacency segments are costly.

- **ECMP-free**: to ensure a 1-to-1 mapping between a cycle and a network path, no cycle should use an ECMP path.

The necessary steps to construct the cycles are the following. First, a monitoring topology is created, with the goal of minimizing ECMP. Then, the cycles are created using the FindCycle algorithm, summarized as follows. Given a network topology, a source node and a segment budget (*i.e.*, the maximum number of segments allowed per cycle), this algorithm computes longest paths by iterating over successive DAGs while keeping track of already traversed edges and maintaining the segment budget. The longest paths are computed using the DAGLongestPath algorithm, that leverages a standard dynamic programming algorithm for longest paths computation [88], extended to also compute the number of segments required in a given path. Once all the cycles are computed, they are segmented, *i.e.*, transformed into a list of segments. This segmentation is performed using the MinSegECMP algorithm. This algorithm computes an ECMP-free segmentation of a path such that it is (i) minimal and (ii) simple whenever possible, *i.e.*, it uses an adjacency segment if and only if all possible segmentations of the path require an adjacency segment. The details of all these algorithms are available in [67].

4.2.2 Implementation and evaluation

Implementation

We developed an SCMon prototype in python [67] (henceforth referred to as simply *SCMon*). Our implementation takes as input the list of cycles computed by FindCycle and periodically sends UDP probes over each cycle. Each probe contains (i) a unique per-cycle token, (ii) the current transmission time in milliseconds and (iii) a monotonically increasing 16-bit probe identifier. A `Transmitter` thread sends those probes along each cycle. The destination address of the UDP probes is set to SCMon itself. A `Receiver` thread gathers the probes and updates the timers accordingly. A coordinating `Monitor` thread reacts on events and timeouts, and updates the cycles state when necessary. The state of each cycle is described by the Finite State Machine shown in Figure 4.12.

Each cycle receives a parameter T_1 that defines the delay between successive probes for this cycle. This parameter is user-defined and bounded by the maximum rate at which SCMon is able to send probes. When SCMon starts, it enters an initial state (`INIT`) which calibrates the RTT of each cycle. SCMon sends and receives probes over each cycle at the rate defined by T_1 and does not consider any late arrival as a cycle timeout (the state flips between `RTT` and `SEND2`). The number of calibrating probes is defined by a configuration variable P_k. We experimentally set P_k = 10 since it proved large enough to account for unexpected jitter in our experiments. Once SCMon has received P_k successive probes, it considers

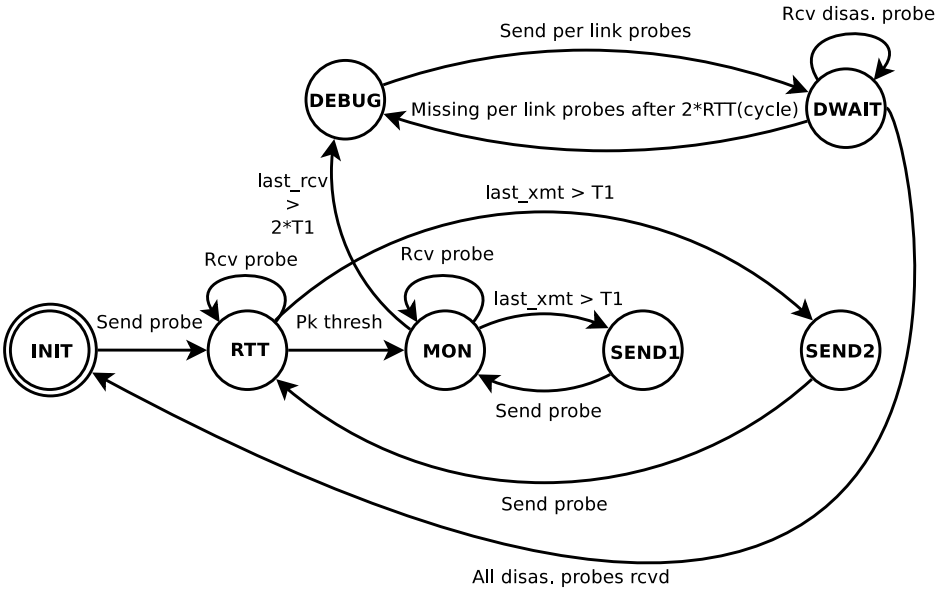


Figure 4.12: Per-cycle Finite State Machine.

that the cycle is up, set the cycle RTT as the average of the P_k measurements, and enters the actual monitoring state (MON). For a given cycle, if a probe is not received within $2 \times T_1$ milliseconds, then the cycle is considered as timed out and SCMon enters a debugging state for this cycle (DEBUG). In this state, SCMon sends one probe for each segment composing the cycle to determine the faulty one (for our prototype, we make the assumption of a single failure). If SCMon does not receive all the debugging probes within $2 \times \texttt{RTT}_{cycle}$ milliseconds, it outputs the first segment that has timed out and starts the debugging state over again. If all the debugging probes are received within $2 \times \texttt{RTT}_{cycle}$ milliseconds, the cycle is considered back up and SCMon re-enters the initial state for this cycle. Re-entering the initial state, and thus temporarily not reacting to timeout events allows the cycle RTT to get smoother if the detected fault was caused by a temporary jitter in the cycle RTT.

Evaluation

Our experiments consist in evaluating the effectiveness of SCMon to detect and correctly locate simulated link failures. We focus on single failures. Using Nanonet as presented in Section 3.5, we emulated the topologies of a large hosting provider and four RocketFuel topologies [89, 74]. The properties of each topology are described in Table 4.3. The OVH topology has a very high number of ECMP paths and thus is a good candidate to evaluate SCMon. As no link latency data was available for this topology, we arbitrarily set the delay of each link to

Table 4.3: Topologies used for SCMon evaluation.

Topology	Nodes	Links	Cycles	Avg RTT	Max RTT
OVH Europe	57	216	87	18 ms	28 ms
RF AS1239	153	1010	195	83 ms	360 ms
RF AS1755	67	248	34	49 ms	130 ms
RF AS3257	103	484	76	48 ms	127 ms
RF AS3967	57	208	24	109 ms	206 ms

one millisecond. Setting a minimal queueing delay for each link circumvents the events ordering issue as explained in Section 3.5.2. For each topology, the cycles were computed with a segment budget $k = 5$, except for the OVH topology where the budget was $k = 8$. This higher budget is due to the high amount of ECMP in this network. The cycle computation algorithm could not be completed with a smaller segment budget. Figure 4.13 shows the time needed to detect a link failure (simulated as a blackhole) for each test topology. We can see that the detection time mainly depends on (i) the number of cycles and (ii) the average cycle RTT. For the largest topology (RF1239), about 90% of link failures were detected in less than 100 milliseconds.

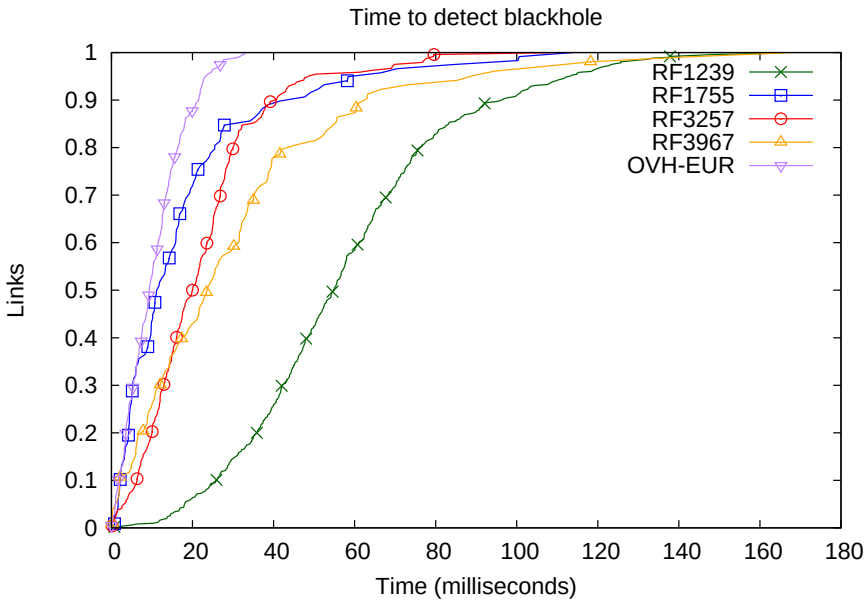


Figure 4.13: Link failure detection time for each topology.

4.2.3 Related work

There is a huge literature about monitoring and fault detection, including pioneering work published almost three decades ago [90]. Previous work typically start from the consideration that queries to devices (*e.g.*, through SNMP) cannot always be trusted [86], and analyses of control-plane messages (*e.g.*, OSPF or IS-IS hello packets) do not provide enough information on data-plane performance. These limitations are also faced by most commercial products (*e.g.*, Tivoli NetView) that aggregate basic tools, from IP SLA to SNMP traps and Syslog collection into a common framework.

Many prior work (see, *e.g.*, [91]) focus on data correlation and statistical techniques for detecting faults and service disruptions. For example, [92] studied how to detect silent (hardware) failures with active measurements and their post-elaboration using a greedy heuristic. Similarly, many contributions have been made in the area of network tomography where topology and link performance are inferred from end-to-end measurements (*e.g.*, [93]). However, all prior work overlooked the impact of multi-path routing, that can make failures much harder to detect and troubleshoot. SCMon tackles those scenarios using an additional topology and Segment Routing to avoid multi-path routing for monitoring probes.

In addition, previous contributions typically assumed multiple vantage points, and tried to optimize their position in order to minimize their number while guaranteeing high network coverage (*e.g.*, [94]). The presence of multiple vantage points is costly and requires coordination (time synchronisation, probe identification and so on). An exception is represented by [95], which is based on a single monitoring point. However, that methodology needs unreliable tools like SNMP or Netflow to collect information on the traversed routers. SCMon effectively used a single monitoring box both to send probes (over cycles) and extract measurements from them.

The approach closest in spirit to SCMon is [86], where monitoring paths are source-routed thanks to either explicit tunnels (*i.e.*, RSVP-TE) or static routes. However, [86] can explore only layer-3 paths (hence, failures on aggregated link bundles are impossible to detect) and tends to create a lot of state (especially if RSVP-TE is used). SCMon avoids those limitations: It relies on Segment Routing that requires no state on the routers and can pinpoint a layer-2 failure. In addition, SCMon improves debugging time from order of minutes (as taken by [86]) to milliseconds.

4.2.4 Conclusion

In this section, we presented SCMon, a new monitoring technique that relies on Segment Routing to send probes over cycles. SCMon allows any single box

to effectively monitor all the network resources, including single links in bundles. We described algorithms to compute probe-traversed cycles inducing a limited overhead and the corresponding SR configurations for the probes. Further, we implemented an SCMon prototype and evaluated its performance on publicly-available topologies and emulated networks. Our experiments show that SCMon works well in practice. By using a limited number of cycles, it takes milliseconds to pinpoint the location of failures, like packets silently discarded by router hardware, that cannot be detected by existing techniques.

4.3 Conclusion

In this chapter, we explored how Segment Routing can (i) improve real-time applications running in low-latency environments and (ii) provide efficient and scalable network monitoring.

In Section 4.1, we analyzed how traffic duplication over disjoint paths can be beneficial to low-latency, real-time applications. We described algorithms to compute disjoint paths using segments and we presented an extension to our SRv6 Linux implementation that provides packet duplication capabilities. Then, we performed simulations to analyze the impact of duplication on the Linux TCP stack. By duplicating traffic over homogeneous links of identical delays and without packet losses, we measured that traffic duplication incurs a small overhead over non-duplicated traffic. Afterwards, we showed that traffic duplication over heterogeneous paths is highly beneficial, especially in the presence of packet losses. Furthermore, we analyzed how the delay difference between the disjoint paths and the rate of packet loss can impact the benefits of duplication. We showed that the adverse impact of packet losses increases as the difference of path delays increases.

In Section 4.2, we presented SCMon, a network monitoring technique that leverages the Segment Routing architecture to send probes over cycles. Those cycles provide a full network coverage and enable to deterministically explore all the components of ECMP paths and link bundles. We implemented SCMon and evaluated its performance on emulated real topologies. The results show that SCMon is able to detect and pinpoint single-link failures in less than 100 milliseconds for large topologies.

Chapter 5

Rethinking IPv6 Enterprise Networks

For the last few years, IPv6 adoption 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 [96]. Today, a growing fraction of mobile and home users rely on IPv6 to access web-based services [97, 98, 99] and some mobile providers have deployed IPv6-only networks.

However, this IPv6 wave has not yet reached the majority of enterprise networks. While very large enterprises have already pledged to move to an IPv6-only architecture [100, 101, 102, 103], the vast majority of them still rely exclusively on IPv4. 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 entreprises are seduced by Software Defined Networks (SDN) [21, 20, 22] 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 [104, 105]. A symptom of this complexity is the large number of middleboxes that are deployed in many enterprise networks [106]. Several types of SDN networks have been proposed (see [22] for a detailed survey). They typically rely on a logically centralized 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 [21].

In this chapter, we propose a variant of the SDN architecture that we call *Software Resolved Networks* (SRN). An SRN is a network that is managed by

a centralized controller like an SDN. There are two major differences between OpenFlow-based SDN networks and SRNs. First, as already proposed in [107], applications can interact directly with the controller. This interaction is performed by extending the DNS protocol that applications already use and by enabling the controller to act as a DNS resolver. For this reason, we call the controller an *SDN Resolver*. We argue that, as the ultimate traffic sources and sinks, the applications are in the best position to provide hints about the nature and needs of their communications. This is especially true in enterprise networks, which are controlled environments. The enterprise chooses and maintains the applications that use its network. Second, we leverage SRv6 to enable the controller to enforce network paths without having to create state on all intermediate routers. Using SR, the hints transmitted by the applications about their traffic (*i.e.*, *policies*) are translated into a suitable network path. A unique identifier is associated to this path (*Path ID*) and handed to the given application. The application can use this identifier to ensure that its traffic will be forwarded along the corresponding path. This path is enforced using the SR data plane. *SDN Resolver* catches any event that would cause the computed path to no longer match the policies (*e.g.*, traffic congestion) or to become unavailable (*e.g.*, link failure). A new path is automatically computed and mapped to the corresponding *Path ID*. The *Path ID* remains unchanged. This abstraction enables such internal reconfiguration to be transparent for the applications.

This chapter is organized as follows. First, we describe an architecture implementing an application-centric SDN paradigm suitable for enterprise networks. This architecture supports *conversations* between applications, regulated by their interactions with the controller through the DNS protocol. Then, we present an implementation of this architecture through the *SDN Resolver* controller. This implementation leverages DNS extensions enabling (i) the applications to embed traffic or path requirements in their DNS requests and (ii) the controller to return the appropriate Path IDs to the applications. Afterwards, we demonstrate the feasibility of our approach through a prototype implementation running on Linux. This prototype includes a complete implementation of a modular controller, and extensions to the SRv6 implementation and to DNS libraries. Finally, we assess the flexibility of our prototype and demonstrate its performance through various microbenchmarks and experiments in an emulated network.

5.1 Software Resolved Networks

In this section, we present the general principles of Software Resolved Networks (SRN). SRNs are designed to support applications and thus interact with them. We focus on unicast flows and leave multicast support for future work.

When an application running on a client host communicates with a server, the packets carried from one endpoint to the other is 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 bi-directional 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 located inside the enterprise network is called an *internal application*. Likewise, an application located outside the enterprise network 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 given DNS naming plan. Furthermore, we leverage the large number of IPv6 addresses [108]. For example, each server application 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 DNS to interact with the controller. To facilitate this interaction, the default resolver configured for these applications is the controller itself (the *SDN Resolver*), acting transparently as a regular DNS resolver.

When a client application initiates a connection 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 [109]. 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

[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.

may re-issue 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.

In legacy IPv4 networks, server applications are usually associated to a static IPv4 address. However, IPv6 addresses are much more volatile than their IPv4 counterparts [110]. Additionally, with the current trends in virtualization, applications may migrate from one VM to another. For redundancy or load balancing purposes, multiple instances of an application may also exist, each having its own address.

As such, the *SDN Resolver* 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 [111, 112] that will update the DNS entry corresponding to this name.

When the server receives a connection from a client (*e.g.*, a TCP SYN packet), 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 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, independently of the transport protocol. This is realized by using a Segment Routing Header and encoding the Path ID as a segment. Instead of resolving an application name, the server issues a conversation request for the Path ID, which is an opaque string. This request may also include traffic requirements. The controller recognizes the Path ID and 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 5.1 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 re-

verse Path ID, with a bandwidth requirement of 6 Mbps. The controller replies with the relevant Path ID and the exchange of packets continues.

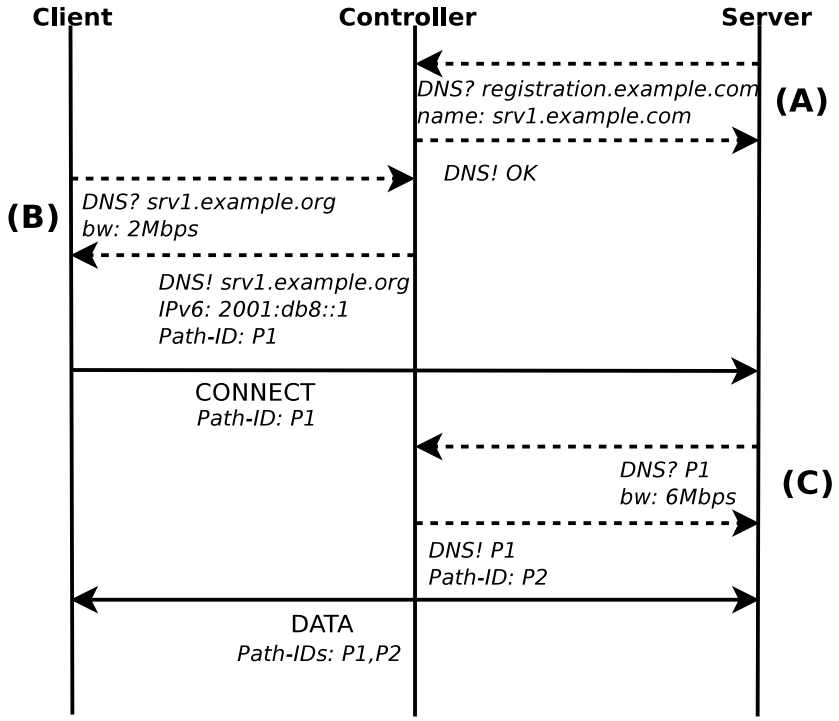


Figure 5.1: Workflow for server registration and connection establishment.

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 an SRH, which is 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

```
1: allow from LAN1 to LAN2 via FIREWALL maxidle 60s

2: allow from LAN1 to STREAMSERVER1 bw 5Mbps delay 10ms

3: allow from SERVER1 to EXTERNAL_BACKUP bw 100Mbps
```

Figure 5.2: Examples of controller rules.

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 particularity of this type of conversation is that the server resides outside the enterprise network. The controller can only control one direction of the conversation. 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.

Many SDN solutions allow the network operators to configure or program the controller with rules or specific languages [113, 114]. SRNs also support such operator-defined policies. We define those policies in a per-rule fashion. Each rule matches a set of conversation requests and defines the actions to apply and the properties to implement. Figure 5.2 illustrate these rules. Table 5.1 details the main keywords supported by our current rules syntax. The rules match conversation requests based on the source and destination application. When the controller receives a conversation request with a Path ID as name (*i.e.*, to request a Path ID for the opposite direction of a conversation), it simply performs a rule lookup by inverting the source and destination applications of the initial half-conversation.

Table 5.1: List of available keywords in rules syntax.

Keyword	Argument	Type	Role
allow	∅	Action	Accept the conv. req.
deny	∅	Action	Reject the conv. req.
from	*name*	Matching	Specify source app.
to	*name*	Matching	Specify destination app.
via	*list of nodes*	Property	Set loose path
last	*node*	Property	Set last node of path
bw	*integer*	Property	Bandwidth to reserve
delay	*integer*	Property	Maximum one-way delay
lifetime	*integer*	Property	Max conv. life time
maxidle	*integer*	Property	Max conv. idle time

Each rule can specify three classes of parameters for the matching conversations. The first one is topological with the `via` and `last` keywords. The former enables to specify a loose path (*i.e.*, a list of intermediate nodes which are not necessarily contiguous) for the conversations. The latter specifies a mandatory penultimate hop that must be traversed. This class of parameters enables to specify, *e.g.*, middleboxes to traverse or an egress router for outgoing external traffic. The second class of parameters relates to the characteristics of the network path with the `bw` and `delay` keywords. Those keywords specify resp. the minimum bandwidth and the maximum delay of the conversation path. Finally, the third class of parameters enables the operator to put time constraints on the conversations. The `lifetime` keyword forces a hard timeout for the matching conversation. It is automatically destroyed when the specified amount of time has passed since its creation. The `maxidle` keyword specifies a soft timeout. The conversation is destroyed when it has not exchanged packets for the given amount of time. This timer ensures that the conversations will not live forever in the network.

5.2 SDN Resolver

The *SDN Resolver* is the logical controller that manages an SRNs. 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 segments by the *SDN Resolver* and enforced into the network.

In this section, we describe in details the components, protocols and processes composing *SDN Resolver*. We start by listing basic assumptions about the underlying network. Then, we explain how we leverage the SR architecture and show how we use it as our data plane layer. Finally, we describe the inner workings of

the control plane components.

5.2.1 Enterprise network

Enterprise networks are composed of three main types of network devices: layer-2 switches, layer-3 routers and middleboxes [106]. We assume that the layer-2 switches support standard access control mechanisms such as IEEE 802.1x [115]. This access control either grants or blocks access to the layer-2 network. It does not provide fine-grained access control on a per destination basis as required in many enterprise networks [20].

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's routers.

We use Router Advertisements [116] (RAs) and StateLess Address AutoConfiguration (SLAAC) to assign one or more IPv6 addresses to the hosts. The DNS configuration of the hosts is achieved with the RDNSS extension to RAs [117]. For platforms that do not support RDNSS, stateless DHCPv6 can be deployed to push DNS configuration on the hosts. This is in line with [108].

Routers exchange routing information by using a link state routing protocol such as OSPFv3 [6]. Each router advertises a loopback address and possibly per-link addresses. We assume for simplicity that middleboxes also implement OSPFv3 and advertise one IPv6 prefix.[2] In particular, we assume that the link state protocol used in the enterprise supports traffic engineering extensions that distribute unidirectional link metrics such as bandwidth utilization and link delay [118].

Network paths are enforced using Segment Routing. In particular, we use the binding segments as described in Section 2.2. Binding segments map to an SRv6 policy (*i.e.*, a list of segments) and instruct the segment endpoint to augment the processed packet with a new SRH, either through encapsulation or direct insertion. We leverage binding segments to implement the Path IDs as described in Section 5.1. 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 traffic 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

[2] In practice, some operators prefer to avoid running routing protocols on non-router devices. In this case, the router attached to the middlebox can be configured to advertise the IPv6 prefix of the middlebox on its behalf and use a BGP session to verify that the middlebox remains up.

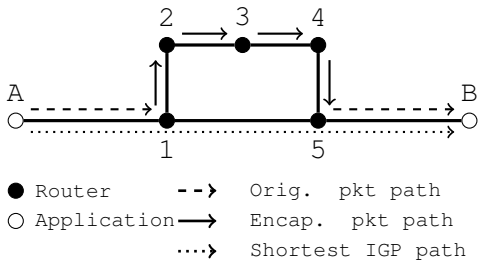


Figure 5.3: Illustration of Segment Routing operations.

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.

A short illustration of these operations is shown in Figure 5.3. On node 3, the segment `3::` is associated with the default endpoint function. This function only updates the IPv6 and SR headers to make the next segment active. On node 5, the segment `5::D` triggers a decapsulation function that removes the outer IPv6 and SR headers, then forwards the inner packet. On node 1, the SR policy $\langle$`3::`, `5::D`$\rangle$ is configured and bound to the segment `1::B1`. The IPv6 prefixes corresponding to these three segments are advertised in the IGP by their respective parent nodes. The client A is configured to send its traffic towards B with the segments list $\langle$`1::B1`, `B::`$\rangle$, where `B::` is a regular IP address configured on an interface of node B. When the packet reaches 1, (i) the IPv6 destination address is updated with the next segment of the list (`B::`), (ii) the packet is encapsulated within an outer IPv6 and SR header, and (iii) it is forwarded along the shortest path to the first segment of the new SRH (`3::`). At node 3, the default endpoint function replaces the outer IP destination address with the next segment (`5::D`). The packet is then forwarded again on the shortest path to 5, where the decapsulation function associated to the segment `5::D` removes the outer IPv6 and SR header. The inner, original packet is then forwarded towards its final destination B. The SR behaviors presented in this illustration are detailed in [29]. Note that no particular SR configuration is required on nodes 2 and 4, as they realize only plain IPv6 forwarding.

5.2.2 Traffic management principles

The ability to efficiently perform fine-grained traffic management is one of the founding principles of Segment Routing. In this subsection, we briefly discuss how SR can provide Quality of Service and how it stands among the existing traffic management principles.

Two Quality of Service (QoS) models are usually recognized: DiffServ [119, 120] and IntServ [121]. On the one hand, the DiffServ architecture aims to provide coarse-grained traffic management. It classifies packets into pre-configured traffic classes, each providing different levels of guarantees. Such classes may include best-effort forwarding, expedited forwarding (low latency) [122], assured forwarding (low drop probability) [123], etc. Packets are classified at the edge of the network and are marked with a Differentiated Service Code Point (DSCP) [124, 125] in the IP header. Core routers then recognize the DSCP and process packets according to the traffic class referenced by the DSCP.

On the other hand, the IntServ architecture proposes fine-grained traffic management capabilities, in a per-flow fashion. Instead of receiving a static, pre-configured classification as in DiffServ, packets are dynamically classified using a reservation system [121]. Using RSVP [126], network nodes that require traffic guarantees (*e.g.*, hosts) exchange `PATH` and `RESV` messages with routers. If the path reservation is successful, then all routers participating in the path maintain per-flow state. Subsequent traffic is then guaranteed to meet the requirements as long as it stays within the specifications.

Both architectures have their benefits and drawbacks. DiffServ offers few control over traffic but is highly scalable, as only the edge routers maintain classification rules. IntServ provides fine-grained per-flow traffic control but is not scalable as per-flow state must be kept on all routers along each path, although efforts have been made to aggregate path reservations [127]. In Software Resolved Networks, we extend the scalable, coarse-grained DiffServ architecture with the dynamic, per-flow reservation system of IntServ without adding state in the core. DiffServ-like functionality is implemented by the SR routes at the edge of the network. IntServ-like functionality is implemented by the conversation requests (akin to path reservations) and the controller's global view of the network. In contrary to IntServ, the intelligence is not present in the network but in the controller. Table 5.2 shows a comparison of the different QoS models.

5.2.3 Path segmentation

Network paths implementing application policies are enforced using Segment Routing. Using a loose path description and constraints about link properties, the controller must produce a list of segments that can be used as an SRH. To

Table 5.2: Comparison of QoS models characteristics.

	DiffServ	IntServ	SRN
Classification	Static	Dynamic	Dynamic
Granularity	Per-class	Per-flow	Per-flow
Flow state	Edge	Edge + core	Edge
Protocol	DSCP matching	RSVP	DNS
Decision	Local	Distributed	Centralized

realize this, we leverage the SR formalization described in [67] and summarized as follows.

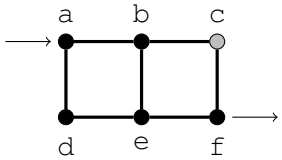
The network is modeled as a weighted directed graph $G = (V, E, w)$ where $w : E \to \mathbb{Z}^+$ is a function corresponding to IGP costs. A path is defined as a sequence of nodes $p = (x_1, x_2, \ldots, x_n)$ where $x_i \in V$ and $(x_i, x_{i+1}) \in E$. Given $p_1 = (x_1, \ldots, x_n)$ and $p_2 = (x_n, \ldots, x_{n+m})$, the concatenation of p_1 and p_2 is defined as $p_1 \oplus p_2 = (x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+m})$. Given a path $p = (x_1, \ldots, x_n)$, the first and last nodes of the path are defined as resp. $\mathit{first}(p) = x_1$ and $\mathit{last}(p) = x_n$. Given G, $\mathcal{D}_x$ is the shortest-path Directed Acyclic Graph (SP-DAG) rooted at node $x \in V$, with respect to the weights given by w. Thus, $\mathcal{D}_x$ is the sub-graph of G containing all the edges that belong to a shortest path starting at x. The set of all shortest paths in G is defined as $\mathit{Sp}(G)$ and the set of all adjacency segments as $\mathit{Adj}(G) = \{(\overrightarrow{x, y}) \mid (x, y) \in E(G)\}$. The set $\mathcal{S}(G) = \mathit{Sp}(G) \cup \mathit{Adj}(G)$ represents the set of all possible segments. In [67], the segmentation of a path $p = (x_1, \ldots, x_n)$ in G is defined as a list $s_1, \ldots, s_k \in \mathcal{S}(G)$ such that $p = s_1 \oplus s_2 \oplus \ldots \oplus s_k$. To transform a path segmentation into a list of segments, we use

$$seg(s_i) = \begin{cases} \mathit{last}(s_i) & \text{if } s_i \in \mathit{Sp}(G) \\ s_i & \text{if } s_i \in \mathit{Adj}(G) \end{cases}$$

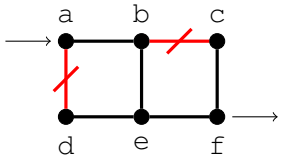
The list of segments that corresponds to $s_1 \oplus s_2 \oplus \ldots \oplus s_k$ is thus defined as $\langle \mathit{seg}(s_1), \mathit{seg}(s_2), \ldots, \mathit{seg}(s_k) \rangle$.

Figure 5.4 shows examples of possible types of constraints. The same graph $G = (V, E, w)$ is represented in the two subfigures. In Figure 5.4a, we search for a path p such that $\mathit{first}(p) = a$, $\mathit{last}(p) = f$ and $c \in p$. Such a path can be $p = (a, b, c, f)$ with a corresponding list of segments $\langle c, f \rangle$. Another valid list of segments is $\langle b, c, f \rangle$. The only difference is that the former is *minimal*. In Figure 5.4b, we search for a path p such that $\mathit{first}(p) = a$, $\mathit{last}(p) = f$, $(a, d) \notin p$ and $(b, c) \notin p$. Such a path is $p = (a, b, e, f)$ with a corresponding list of segments $\langle b, e, f \rangle$. Note that segment b is necessary to prevent from using the forbidden link (a, d).

Using this formalization, we define the buildSegpath function as described in Algorithm 2. From a graph G, a source node s, a destination node t, a list



(a) Node traversal constraint through c: *first*$(p) = a$, *last*$(p) = f$ and $c \in p$



(b) Link avoidance constraint: *first*$(p) = a$, *last*$(p) = f$, $(a, d) \notin p$ and $(b, c) \notin p$

Figure 5.4: Example of possible path constraints from `a` to `f`.

of nodes to traverse L, a set of forbidden edges F and a set of path policies $\mathcal{P}$, this algorithm computes the minimal list of segments to implement the path while satisfying the policies. First, we create a temporary graph where the forbidden edges are removed. Then, we set the source node as the current node and we iterate over each node in L. At each iteration, we select a shortest path going from the current node to the next node within the temporary graph. The selectPath function is generic and must be extended by an external algorithm to compute a path that matches the policies. Then, we leverage the MinSegECMP algorithm defined in [67]. This algorithm computes the minimal ECMP-free segmentation of a path. We append to S the list of segmentations produced by MinSegECMP, for the selected path and within the *original* graph. Note that it is necessary to use the original graph for the minimal segmentation. Let us consider Figure 5.4b and the graph G' where the edges (a, d) and (b, c) are removed. The minimal segmentation of the path (a, b, e, f) in G' is $\langle f \rangle$ because it is the unique shortest path from a to f in G'. However, in the actual graph G, multiple shortest paths are available, and several of them traverse forbidden edges. Using MinSegECMP in the original graph ensures that those paths are not inadvertently considered.

5.2.4 SRN Control plane

The control plane of an SRN consists of several components. At the core is the logically centralized controller (the *SDN Resolver*). It does not communicate directly with applications. Instead, the exchanges between the applications and the controller are mediated by DNS forwarders and a DNS proxy. A DNS forwarder is installed on each access router. Each forwarder receives DNS requests from applications and forwards them to the DNS proxy. The latter performs the

Algorithm 2 Segmented path construction

function BUILDSEGPATH($G(V, E, w), s, t, L, F, \mathcal{P}$)
  $G'(V', E', w') \leftarrow G(V, E, w)$
  $S \leftarrow \langle\rangle$
  for all $(x, y) \in F$ **do**
    $E' \leftarrow E' \setminus \{(x, y)\}$
  end for
  $L \leftarrow L + t$
  $cur \leftarrow s$
  for all $v \in L$ **do**
    $\mathcal{D}_{cur} \leftarrow Dijkstra(G', cur)$
    $p \leftarrow selectPath(\mathcal{D}_{cur}, v, \mathcal{P})$
    $S \leftarrow S + MinSegECMP(p, G)$
    $cur \leftarrow v$
  end for
  return S
end function

actual DNS resolution, using the enterprise's resolver, and forwards the conversation requests to the controller. The controller processes the conversation request, then instructs the router's routing daemon to insert the resulting SR policy into its Forwarding Information Base (FIB). Simultaneously, the controller returns the generated binding segment (*i.e.*, Path ID) to the DNS proxy. The proxy then crafts the corresponding DNS reply, using the resolved address and the binding segment. This reply is transferred to the DNS forwarder. Finally, the forwarder monitors the router's FIB and 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. Ultimately, the forwarder passes on the DNS reply to the application. Using forwarder-proxy pairs enables to move the actual DNS resolution process closer to the controller. Assuming that the controller, proxy, and resolver are in the same network vicinity, this setup allows to group most of the transactions within a low-delay network radius. If the DNS forwarder were to perform the DNS resolution and interact with the controller, then at least two RTTs would be wasted in transactions (one for the DNS resolution and one for the controller transactions). Delegating this task to the DNS proxy enables to perform the full conversation request within approximately one RTT between the application and the controller (modulo computations and transactions overhead).

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 5.5.

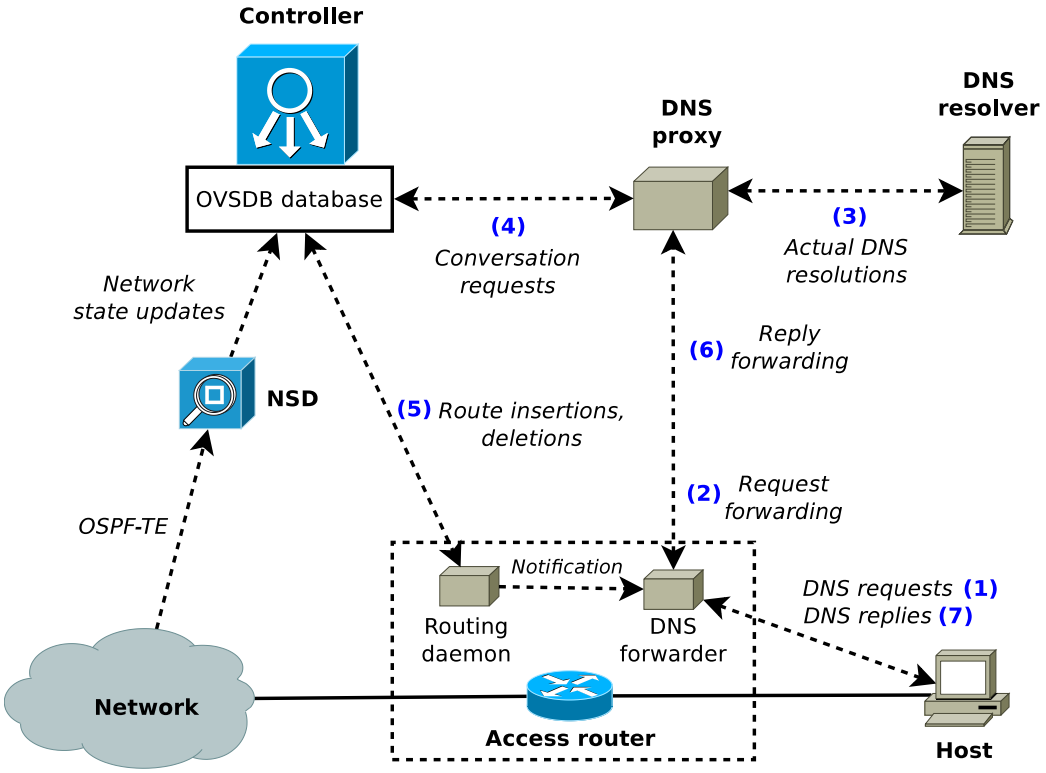


Figure 5.5: Illustration of the components of an SRN. The figure shows the exchanges involved in a conversation request.

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 [109] 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 the future.

In our prototype, the communications between the controller and its components (*i.e.*, southbound interface) are realized through the OVSDB protocol [128]. Originally designed for the Open vSwitch suite, OVSDB is a generic, JSON-RPC

based protocol, supporting transactional queries on NoSQL-like databases. In OpenFlow networks, OVSDB is used to configure the switches and to dump the flow tables. The actual per-flow configuration is performed through the OpenFlow protocol. However, OVSDB can be used as the per-flow configuration protocol for non-OpenFlow networks such as proposed in [129]. Although a database approach for state synchronization between network components is uncommon, OVSDB has the advantage of being generic and simple, as opposed to traditional network protocols such as BGP.

We define six OVSDB tables for *SDN Resolver*. 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, the identifier of the corresponding access router, and a status. The status is set to `REQ_STATUS_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 `REQ_STATUS_ALLOWED`. Otherwise, the status is set to a value that reflects the reason why the conversation is 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 application, 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. Although the controller could garbage collect conversations that hit the `lifetime` timer, it does not have sufficient information to remove conversations that reached the `maxidle` timer. The details of those timers are explained in Section 5.1. `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. Finally, `RouterConfig` stores various configuration parameters for SR-aware routers. It is written by the controller (or by an operator-specific management tool) and read by the routers. A summary is shown in Table 5.3.

Operations

To ensure the correct operation of the network according to the principles described in Section 5.1, our controller includes at least two processes: conversation requests and server registrations (see Figure 5.1). We also support a third opera-

Table 5.3: Summary of OVSDB tables and read/write access per component.

Table	Controller	NSD	DNS Proxy	DNS Forwarder	Routing daemon
`ConvReq`	R/W	-	R/W	-	-
`ConvState`	W	-	-	-	R/W
`ServReg`	R/W	-	R/W	-	-
`NodeState`	R	W	-	-	-
`LinkState`	R	W	-	-	-
`RouterConfig`	W	-	-	R	R

tion: reactions to network events. Those processes are handled 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 QoS 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. This transformation is described in Algorithm 2 (buildSegpath). Then, the controller generates a binding segment and create 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 OVSDB 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 [130]), 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 as long as needed (usually, until expiration). For resiliency purposes, the controller may run multiple iterations of buildSegpath by successively removing the edges of the previously computed paths. Each iteration would then produce a valid path (*i.e.*, matching the constraints) that is link-disjoint from paths computed at previous iterations. Alternatively, the controller may 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 [111, 112]. 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.[3] 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 [131]. 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 [132]. When the controller detects a network event, it scans its conversation state database and builds a list of adversely affected conversations. In case of link or node failure, all conversations that traverse the failed link or node are affected. 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 path exists and is not adversely affected by the network event, it is selected as replacement. Otherwise, the controller recomputes a new path that suits the conversation policies using the buildSegpath function. The controller updates the corresponding entries in the OVSDB `ConvState` table. The affected routers receive the OVSDB update notifications and carry the changes to their FIB. Note that the controller can change a path without interacting with the application, as Path IDs provide a level of indirection to the actual segmented path implemented on the access router. 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 [133, 134] could also be included in the controller.

5.2.5 Fault tolerance

Let us consider the resilience of *SDN Resolver*. 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 affected (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

[3] Multiple entries for a single DNS record implement a DNS round-robin.

consider that the network should be able to operate in *degraded mode* without the controller. To realize this, the DNS proxies can be configured to return a default binding segment if the controller is unavailable. The behavior of this default SR policy is operator-defined. For example, it may simply forward the packets to their destination, following the IGP 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 [135, 136]. In the latter case, a controller failure will affect only its designated network partition, avoiding network-wide degraded mode. Partitioning the network can have additional benefits, such as reducing the controller's load. For example, if several groups of access routers are known for receiving conversation requests at a very high frequency, then they may be assigned their own controller. 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.

5.2.6 Security implications

From a security viewpoint, an *SDN Resolver* is exposed to the same security risks as the enterprise's internal DNS resolver. As such, external traffic 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 not allowed to cross the border routers. Otherwise, it would enable an external attacker to gain unauthorized access to the enterprise network resources. Such access enables attacks such as amplification, reflection, and bypass [19]. The mitigation for this threat is simply to configure the border routers to drop all SR-enabled packets originating from an external interface [29].

Internal threats coming from, *e.g.*, malicious employees, are more difficult to address. However, it is possible to mitigate some of them. The controller and DNS proxy are not directly exposed to user traffic, as all their interactions

with users happen through DNS forwarders. Furthermore, network devices cannot interact with other DNS forwarders than the one located on their access router. This enables to contain any damage done on a DNS forwarder to its service area. To prevent misuses of network resources, only authorized equipment should be allowed to emit arbitrary SRHs on the network. The SR-enabled packets generated by applications should be subject to access control when reaching an access router, to prevent unauthorized usage of segments. Additionally, DNSSEC [112, 137, 138] can be leveraged to prevent an attacker from tampering with DNS responses or generating fake server registrations.

5.2.7 Comparison with OpenFlow

There exist several key differences between SRN and OpenFlow, which is the foremost SDN implementation. An OpenFlow controller works directly on top of switches and fills their flow table. If a flow table is empty, then the corresponding switch is unable to forward packets. If a switch reboots while the OpenFlow controller is down, it is unable to recover its flow table. On the other hand, an *SDN Resolver* is running on top of a network of IPv6 routers. Those routers already have IPv6 reachability information distributed through an IGP. As a result, an SRN is still able to forward traffic and recover from failures even in the situation where the controller is down. Indeed, the IGP will converge and new routes will be automatically defined.

To install a path in an OpenFlow network, the controller needs to configure every switch that is part of that path. Consequently, the state in core switches grows linearly with the number of active flows in the network. This can lead to resources exhaustion as memory is often limited in this type of devices. In an SRN, only the edge routers need to maintain per-flow state. The core routers are *stateless* in this respect. They do not need to maintain more state than classical reachability information.

Finally, as SRv6 is an overlay over IPv6 routing, it is straightforward to realize incremental deployment. Indeed, only the routers acting as segment endpoints must support SRv6. The rest of the routers only need to perform regular IPv6 forwarding. While incremental deployments are possible in OpenFlow networks [148], it is more tedious than in SRNs, as all switches in a flow's path are expected to be OpenFlow switches.

5.3 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.

5.3.1 Kernel modifications

The Linux kernel, since version 4.10, already includes basic support for SRv6 [38]. However, this release does not explicitly support binding segments, which are required for the Path IDs that are used in our architecture. To use Path IDs, routers must be able to encapsulate a packet when its active segment matches a given address. The active segment is defined as the destination address of the packet. We implement the binding segments by extending the in-kernel SRv6 processing mechanism. For this, we use a hashtable that maps a binding segment to a routing entry. This entry points to the SRv6 encapsulation function with the corresponding list of segments as parameters. We also add code in the SRH processing function `ipv6_srh_rcv()` to verify if the active segment of the current packet matches an entry in the hashtable. In this case, the packet is re-routed using the associated routing entry and automatically encapsulated with the proper SRH. This two-steps mapping has the advantage of (i) reusing the SRv6 encapsulation functions and (ii) avoiding the proliferation of routing entries when several binding segments are mapped to the same list of segments.

We also associate two timers to each mapping. The first one is set at the creation of the mapping to the value of the *lifetime* rule keyword defined in Section 5.1. The second one is reset to the value of the *maxidle* rule keyword whenever a packet matches the mapping. The route implementing the SR policy is removed when either timer expires. SR policies are configured using the standard NETLINK interface. We also modified the `iproute2` userspace tool to support our kernel modifications. Our kernel modifications are grouped in two patches with a total of 760 lines of code. The `iproute2` changes consist of about 200 lines of code.

5.3.2 Path ID propagation

In Figure 5.1, after the client has established the connection, the server requests a binding segment for the server-client 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.

Additionally, technical limitations of the Linux kernel prevent the server from dynamically setting an SRH for SYN/ACK packets sent in response to TCP SYN requests. Indeed, when a Linux host receives a SYN packet corresponding to a

listening application, it immediately responds with a SYN/ACK, without notifying the application. A call to `accept()` to fetch the file descriptor of a new connection will succeed and return only when the three-way handshake is fully completed. At this point, the server application has the opportunity to define an SRH for the newly established connection. However, for some use cases, it is important that *all* packets follow the same SRv6 policy (*i.e.*, have the same SRH), including the SYN/ACK packet.

We propose a technique that enables the server application to immediately attach a suitable SRH to all packets following the SYN request, thus allowing (i) faster connection establishment and (ii) enforcement of the SRv6 policy for all packets. This technique is described as follows.

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 5.2.6 should prevent misuses of the feature. Additionally, the HMAC feature of SRv6 can be leveraged to ensure the authenticity and integrity of the SRH.

5.3.3 Segment Routing Database

The Segment Routing Database (SRDB) is the set of OVSDB tables used by *SDN Resolver* to maintain state in an SRN. To support SRDB, we implemented an abstraction layer over OVSDB primitives. An OVSDB table is referenced by a name, and contains multiple fields (or columns), also referenced by name. Two built-in fields are always present in each entry (or row) of the tables. Namely, those fields are `row` and `_version`. They both contain a UUID. The former unambiguously references the row within the OVSDB database and never changes, unless the row is deleted. The latter references the current version of the row and is updated for each write operation performed on the row. Operations on an OVSDB table are realized using JSON transactions. Clients connect to the OVSDB server and can choose to either perform a *transaction* on the database (row insertion, update, or deletion) or request to *monitor* a given table. After each write oper-

Listing 5.1: Storage for an entry of the `ConvReq` table.

```
struct srdb_flowreq_entry {
        struct srdb_entry entry;

        char request_id[SLEN + 1];
        char destination[SLEN + 1];
        char dstaddr[SLEN + 1];
        char source[SLEN + 1];
        int bandwidth;
        int delay;
        char router[SLEN + 1];
        char proxy[SLEN + 1];
        int status;
};
```

ation, the OVSDB server propagates the changes to all monitoring clients. The modified rows are encoded in a JSON format. A virtual field `action` is added to each row, describing the operation that triggered update (*e.g.*, *insert*). On top of these OVSDB primitives, we implement an SRDB library enabling the *SDN Resolver* components to easily interact with the database, in a flexible and extensible manner.

Structures

We define a basic `struct srdb_entry` to represent a generic OVSDB row. This structure contains only the built-in fields. Each SRDB table entry is represented by an extension of this structure. For example, an entry of the `ConvReq` table is represented by a `struct srdb_flowreq_entry`, as shown in Listing 5.1.

To each field (or column) of an SRDB table is associated a template `struct srdb_descriptor`. This descriptor contains the name, type, and length of the field, as well as its offset within the corresponding SRDB entry structure. Given an actual row data, this mechanism enables to automatically fill the corresponding entry structure, when provided with a descriptor for each field. The SRDB descriptor is shown in Listing 5.2.

Each SRDB table is represented by a `struct srdb_table`. This structure contains various fields including the name of the table, a callback function for OVSDB events (row insertion, update, deletion), and an array of descriptors for each of the corresponding entry structure. A description of an SRDB table is shown in Listing 5.3.

Listing 5.2: SRDB field descriptor.

```
struct srdb_descriptor {
        const char *name;
        enum srdb_type type;
        int index;
        size_t maxlen;
        bool builtin;
        off_t offset;
};
```

Listing 5.3: SRDB table.

```
struct srdb_table {
        const char *name;
        const struct srdb_descriptor *desc_tmpl;
        struct srdb_descriptor *desc;
        size_t desc_size;
        size_t entry_size;
        table_insert_cb_t cb_insert;
        table_update_cb_t cb_update;
        table_delete_cb_t cb_delete;
        sem_t initial_read;
        bool delayed_free;
};
```

Architecture

Our SRDB library has been designed to be integrated within a multi-threaded environment. As such, the monitoring and transaction operations can operate as stand-alone threads. We leverage callback functions and thread-safe queues to exchange data with the *SDN Resolver* components that interact with the SRDB. Figure 5.6 shows the overall SRDB library architecture. A *Monitor* thread must be spawned for each table to be monitored. This type of thread receives a callback function pointer in argument and establishes a permanent TCP connection with the OVSDB server. Whenever data is available on the socket, the callback function is called, with the data as argument. In our case, this callback function is `srdb_read()`. Using the raw data and the table descriptors, the function calls `fill_srdb_entry()` to transform the raw data into the corresponding entry structure. Then, another callback function is called, according to the type of operation (insertion, update, deletion).

To issue transactions, the *SDN Resolver* components push a two-element structure into a thread-safe queue. This structure contains the SRDB entry to be

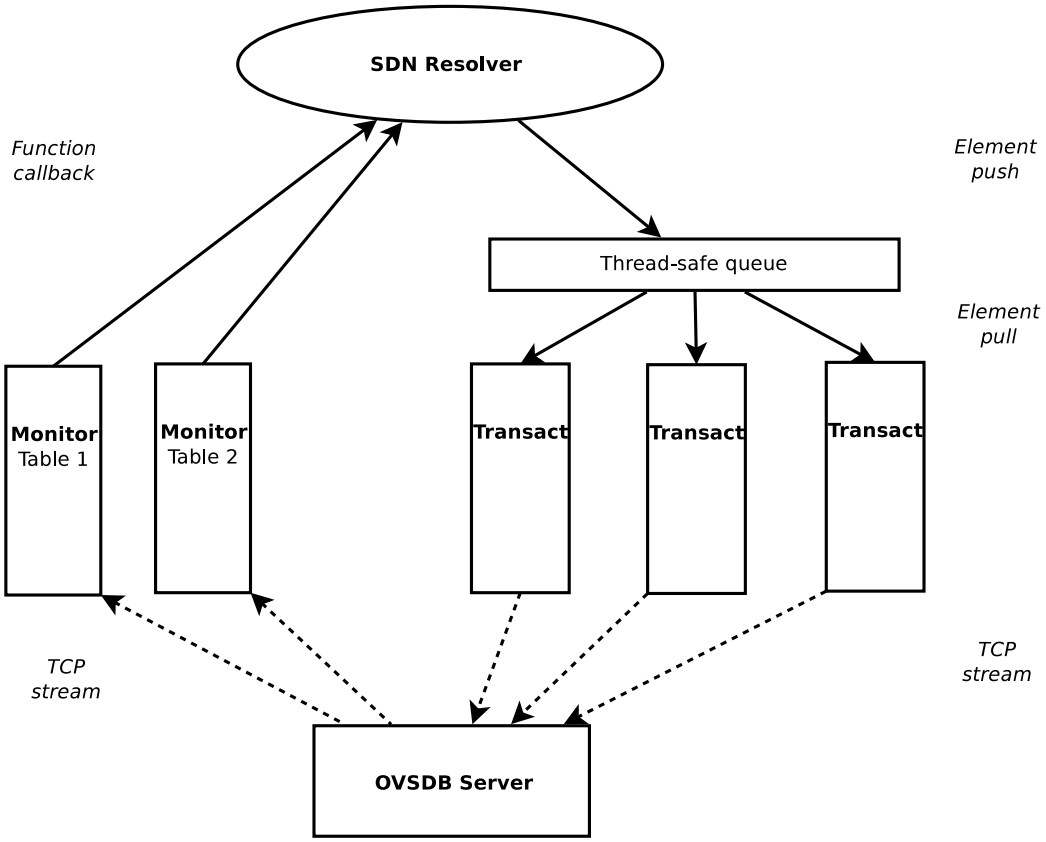


Figure 5.6: Architecture of the SRDB library.

transmitted to the OVSDB server, and a one-element thread-safe buffer, that will be used to store the transaction result. A pool of *Transact* threads fetch elements from the queue, transform the SRDB entry into a JSON transaction and transmit it to the server. The per-thread TCP connection to the server is established at the creation of the thread and is kept open during all the thread's runtime. This enables to skip the three-way handshake for each transaction. Finally, the transaction result is stored in the result buffer. Note that the OVSDB server issues keepalives over the transaction sockets. To be able to respond to them, the transact threads must continuously poll their socket to check if a keepalive has arrived. As such, they cannot use blocking calls to fetch elements from the thread-safe queue. Rather, we use the non-blocking variants of synchronization mechanisms, such as `sem_trywait()`.

Listing 5.4: SRDB instance.

```
struct srdb {
        struct ovsdb_config *conf;
        struct srdb_table *tables;
        struct sbuf *transactions;
        pthread_t *tr_workers;
        struct llist_node *monitors;
};
```

The description of a full SRDB instance is described in Listing 5.4. The structure contains various OVSDB configuration parameters (such as the server address and port), the list of tables, the shared thread-safe buffer used to store transactions, the pool of transaction threads, and the list of monitoring threads.

5.3.4 Graph library

To support the computation of constrained, segmented paths, we implement a lightweight, SR-aware graph library. This library supports a generic representation of nodes, edges, and segments. Using this, we implement the operations and algorithms required to compute and segment paths, such as the buildSegpath and MinSegECMP algorithms. We also ensure that the graph structures can be leveraged in multi-threaded environments by providing synchronization primitives such as read-write locks and reference counters.

Structures

Each node is represented by a numerical identifier and a generic pointer for user storage. Each edge represents a unidirectional link and contains a numerical

identifier, the two connected nodes, the link weight (or metric) and generic user storage. A segment is represented by either a node (for node segments) or an edge (for adjacency segments). Figure 5.7 shows the definition of those three basic structures.

Listing (5.5) Node.

```
struct node {
        unsigned int id;
        void *data;
        void (*destroy)(struct node *node);
        bool orphan;
        atomic_t refcount __refcount_aligned;
};
```

Listing (5.6) Edge.

```
struct edge {
        struct node *local;
        struct node *remote;
        unsigned int id;
        uint32_t metric;
        void *data;
        void (*destroy)(struct edge *edge);
        bool orphan;
        atomic_t refcount __refcount_aligned;
};
```

Listing (5.7) Segment.

```
struct segment {
    union {
            struct node *node;
            struct edge *edge;
    };
    bool adjacency;
};
```

Figure 5.7: Basic structures of the graph library.

A graph is represented by the `struct graph` structure. It contains the set of nodes and edges, stored in arraylists. Three helper hashtables enable fast access to certain types of data to speed up path computations. The first hashtable, `min_edges`, maps each connected pair of nodes to the edge(s) of minimal weight that connects them. This is useful for shortest path computation when two nodes are connected by multiple edges of possibly different weights. The second

hashtable, `neighs`, maps each node to a list of its neighbors. Again, such data is needed at each iteration of the shortest path algorithm and this hashtable enables to considerably accelerate the computation time. The third hashtable, `dcache`, maps each node to its precomputed shortest-path Directed Acyclic Graph (SP-DAG). The first two hashtables speed up the Dijkstra algorithm. The third one speeds up the MinSegECMP algorithm, which requires switching between multiple SP-DAGs. A per-graph `dirty` boolean flag is used as a signal when a node or edge is added or removed from the graph, but the three helper hashtables have not yet been recomputed.

A read-write lock prevents bogus path computations to happen while the graph is being updated. As multiple computations can be realized simultaneously, such a lock is more suitable than a mutex that would implement mutual exclusion to the graph. Furthermore, each node and edge structure contains a reference counter. This enables to hold references to them outside of critical sections (*i.e.*, when the per-graph lock is not taken). When a node or edge are removed from their parent graph but are still referenced, their `orphan` boolean parameter is set to indicate that they are no longer part of a graph. When the reference count of a node or edge reaches zero, then its `destroy` function pointer is called and the structure itself is freed. Besides the `destroy` operation, each graph is specified with a set of user-definable operations, implemented as function pointers. Those operations (*graph operations* or *graph-ops*) are related to the node and edge `data` pointer for generic user storage. They enable to customize node and edge comparison, copy, and destruction.

Operations

The fundamental algorithm to be applied to the graph is the shortest path computation algorithm. We thus implement the Dijkstra algorithm through the `graph_dijkstra()` function. It takes as parameter a graph structure, a source node, a result buffer, a set of user-definable operations and a generic `data` pointer. The set of operations (called *shortest path operations* or *sp-ops*) enables to arbitrarily interfere with and alter the shortest path computations. Four operations are defined, as function pointers: `init`, `destroy`, `cost` and `update`. The first operation is called at the very beginning of `graph_dijkstra()`, using the same arguments. It gives an opportunity to allocate a temporary state-holding buffer. Conversely, the second operation is called at the very end of the shortest path function to free this buffer. The `cost` function is called to compute the cost of an edge. Using a custom algorithm to determine the cost of edges enables to perform shortest path computation using other metrics than the default built-in edge weight (*e.g.*, link delay). The `update` function is called when Dijkstra finds a better path to a given node.

A key Segment Routing algorithm required for the implementation of buildSegpath as defined in Section 5.2.3 is the MinSegECMP algorithm [67]. Using a graph and a path defined within this graph, MinSegECMP computes the minimal list of segments needed to express this path. We implement this algorithm as the `graph_minseg()` function.

Listing 5.8: Path specification for buildSegpath

```
struct pathspec {
        struct node *src;
        struct node *dst;
        struct arraylist *via;
        void (*prune)(struct graph *g, struct pathspec *pspec);
        struct sp_ops *sp_ops;
        void *data;
};
```

On top of `graph_dijkstra()` and `graph_minseg()`, we implement the buildSegpath algorithm through the `build_segpath()` function. This function performs the actual constrained path computation. It takes as arguments a graph and a `struct pathspec` structure, which parametrizes the path to compute. This structure is illustrated in Listing 5.8. The path is described by the source and destination node, and a list of intermediate waypoints. The `prune` function enables to selectively remove edges from the original graph before performing the path computation. The `sp_ops` structure and the `data` pointer are passed on to the `graph_dijkstra` function. The former is the shortest path operations as previously explained, and the latter is a generic pointer given as argument to the sp-ops functions. Using this input, `build_segpath()` performs the following operations. First, the graph is cloned into a local writable copy, to which the prune function is applied. Then, Algorithm 2 is applied. The result is the minimal list of segments implementing the constrained path, or a null value if the path could not be computed.

5.3.5 Controller implementation

Using the *graph* and *SRDB* subsystems, the *core* of the controller 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 a given 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. An overview of the controller architecture is shown in Figure 5.8.

Structures

Two structures are defined to represent routers and links in an SRN, resp. `struct router` and `struct link`. Those structures are incorporated in the network graphs and referenced by the generic `data` pointer of nodes and edges. Figure 5.9 describes those structures.

Each router is defined by its name, a loopback address, an IPv6 prefix from which binding segments are generated, a list of prefixes announced by the router (through the IGP), the identifier of the corresponding node in the production graph, and a reference counter. Using the node identifier instead of a direct pointer enables more flexibility to, *e.g.*, regenerate (reallocate) the graph node using the same identifier, without having to update all existing pointers to the node.

Each link contains the loopback address of the two routers it connects, the bandwidth capacity of the link, the bandwidth currently available, the delay of the link, and a reference counter.

A third important structure used by the core controller is `struct flow`, representing a conversation state. It contains the named source and destination, the IPv6 address of the destination as resolved by the DNS proxy, references to the access routers of the source and the destination, path constraints such as bandwidth, latency, etc., and other parameters such as the creation timestamp and the conversation status.

The network state is maintained in a `struct netstate` as described in Listing 5.11. It contains the production and staging graphs, two timestamps used to keep track of when the production graph should be synchronized with the staging graph, the set of routers present in the network, and all the prefixes announced by the IGP. The prefixes are stored in a compressed prefix tree, to leverage the

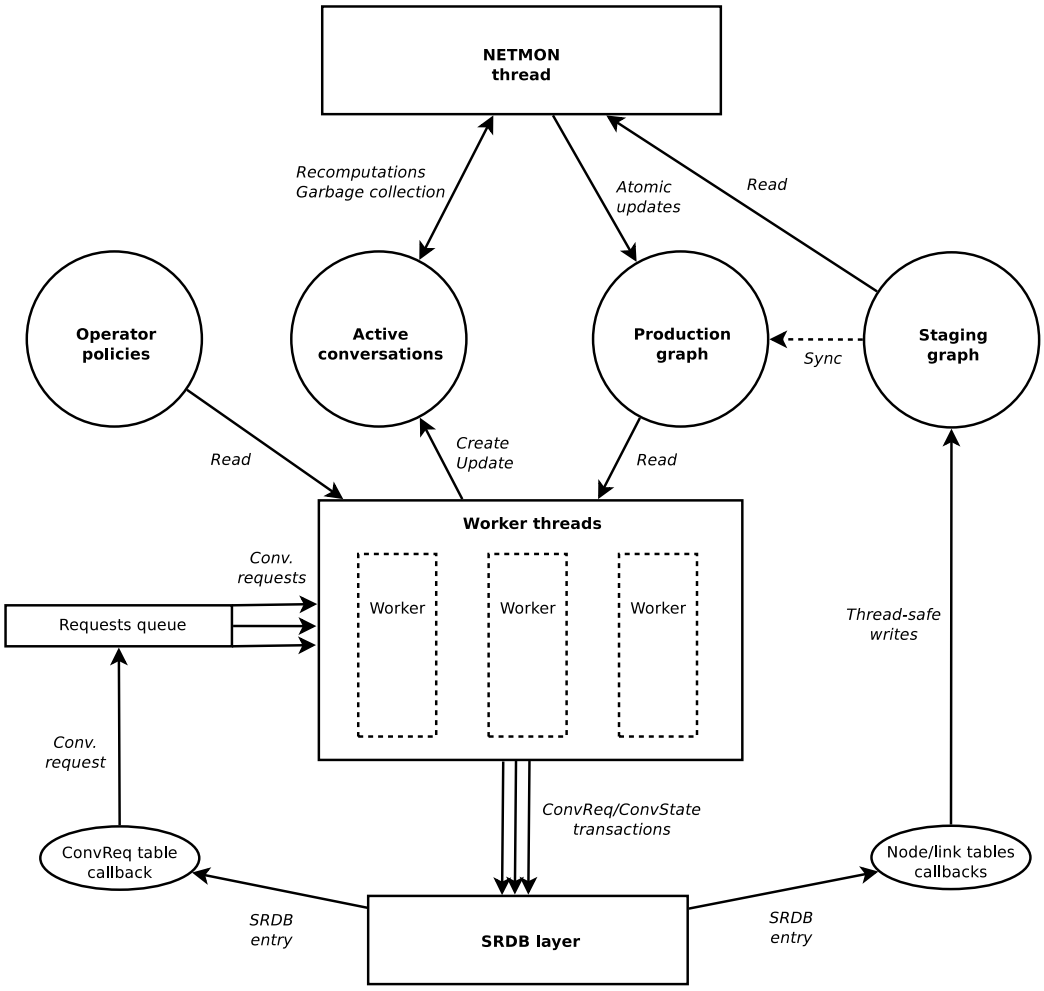


Figure 5.8: Controller architecture.

Listing (5.9) Router.

```
struct router {
        char name[SLEN + 1];
        struct in6_addr addr;
        struct prefix pbsid;
        struct llist_node *prefixes;
        unsigned int node_id;
        atomic_t refcount __refcount_aligned;
};
```

Listing (5.10) Link.

```
struct link {
        struct in6_addr local;
        struct in6_addr remote;
        uint32_t bw;
        uint32_t ava_bw;
        uint32_t delay;
        atomic_t refcount __refcount_aligned;
};
```

Figure 5.9: *SDN Resolver* extension of nodes and edges in an SRN.

Listing 5.11: Network state as maintained by *SDN Resolver*.

```
struct netstate {
        struct graph *graph;
        struct graph *graph_staging;
        struct timeval gs_mod;
        struct timeval gs_dirty;
        struct hashmap *routers;
        struct lpm_tree *prefixes;
        pthread_rwlock_t lock;
};
```

longest-prefix match algorithm. Additionally, a read-write lock protects all the structures in the network state, except for the staging graph that is handled separately.

Listing 5.12: *SDN Resolver* global state.

```
struct config {
        /* skipped various config parameters */

        struct srdb *srdb;
        struct llist_node *rules;
        struct rule *defrule;
        struct sbuf *req_buffer;
        struct netstate ns;
        struct hashmap *flows;
};
```

Finally, the *global state* is maintained in a controller-wide configuration structure as shown in Listing 5.12. This structure holds a reference to the SRDB subsystem, the set of operator-defined policies, the shared thread-safe buffer used to store conversation requests, the network state, and the set of active conversations in the network. Note that our implementation of hashtables (`struct hashmap`) has a built-in read-write lock. As such, it is possible to specifically lock the set of active conversations.

Request processing

The core function of the controller is the `process_request()` function, called by each worker thread when a conversation request is available in the shared buffer. The first step consists in matching the conversation request against the operator-defined policies. This matching is performed using the source application identifier and the named destination. If the policy lookup results in the denial of the request, then the controller updates the conversation request status to reflect this result and returns. Otherwise, the request processing continues. The controller allocates a new conversation state and fills some of its fields. The next step is to fetch the two graph nodes that correspond to (i) the source access router, from which the conversation request emanates, and (ii) the destination access router, attached to the targeted application. Then, the controller fills a `pathspec` structure according to the conversation requirements (*e.g.*, bandwidth, latency, etc.) and calls the `build_segpath()` function to compute the segmented path. If a valid path is found, the conversation state is set as active and inserted into the corresponding internal list. Finally, the conversation state is committed to the

ConvState table and the status of the corresponding entry in ConvReq is set to REQ_STATUS_ALLOWED.

Network monitoring

The role of the network monitoring thread (netmon) is threefold. It must (i) garbage collect expired conversations, (ii) synchronize the production graph with the staging graph if the network topology has changed and (iii) recompute conversations as needed.

To achieve this, netmon keeps three timers. The first one is gc_time and is triggered once every GC_FLOWS_TIMEOUT milliseconds. The default value is one second. The gc_flows() function is called each time this timer expires. Its role is to (i) build a list of expired conversations and (ii) iterate over this list and remove each conversation. The second timer is gs_mod as defined in the network state. It is reset whenever a change is performed on a node or edge in the staging graph. The third timer is gs_dirty, also defined in the network state. It is reset only on the first change to the staging graph since it was synchronized with the production graph. Those two timers expire after resp. GSYNC_SOFT_TIMEOUT (default at 5 milliseconds) and GSYNC_HARD_TIMEOUT (default at 50 milliseconds). When either of those timers expire, then netmon synchronizes the production and staging graphs, and triggers the recomputation of potentially affected conversations. The purpose of this pair of timers is to avoid synchronizing the graphs and recomputing the conversations at each topology change. Such a buffer helps to absorb bursts of changes. For example, if a network node fails, its connected links will be signalled as failed one after the other. Thus, it is more efficient to wait for all the links to come down, rather than trigger recomputations for each of them. This is the purpose of the gs_mod timer. The gs_dirty timer ensures that the controller will not wait indefinitely when topology changes happen at a frequency that always reset the gs_mod timer.

To synchronize the network graphs, netmon calls the netstate_graph_sync() function. This function (i) atomically copies the staging graph, (ii) computes the three helper hashtables, (iii) sets the processed copy as the production graph, and (iv) destroys the former production graph. If references to nodes and edges that were part of the former production graph still exist, the corresponding structures will be freed only when the reference count drops to zero.

The recomputation of conversations is realized by the recompute_flows(), recompute_flow(), and flow_affected() functions. The first function builds a list of affected conversations by iterating over the set of active conversations. Each conversation for which flow_affected() returns true is added to the list. Once the list is computed,

the `recompute_flow()` function is called on each of its elements. This last function is very similar to the `process_request()` function. If the list of segments that implements the network path of the conversation has changed, then the new list of segments is updated in the `ConvState` table.

5.3.6 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 [139]. 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.

5.4 Evaluation

In this section, we evaluate *SDN Resolver* through two angles. First, we use microbenchmarks to measure several aspects of the controller performances. Then, we evaluate how *SDN Resolver* behaves when all its components are integrated in an emulated network.[4]

5.4.1 Microbenchmarks

Conversation requests

The fundamental operations that the controller needs to support is handling conversation requests and generating the corresponding conversation state. We measure how the controller handles conversation requests under a high load. We proceed as follows. First, we initialize an empty OVSDB database with the SRDB schema. We populate the `NodeState` and `LinkState` tables using the Abilene topology [140], 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

[4]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.

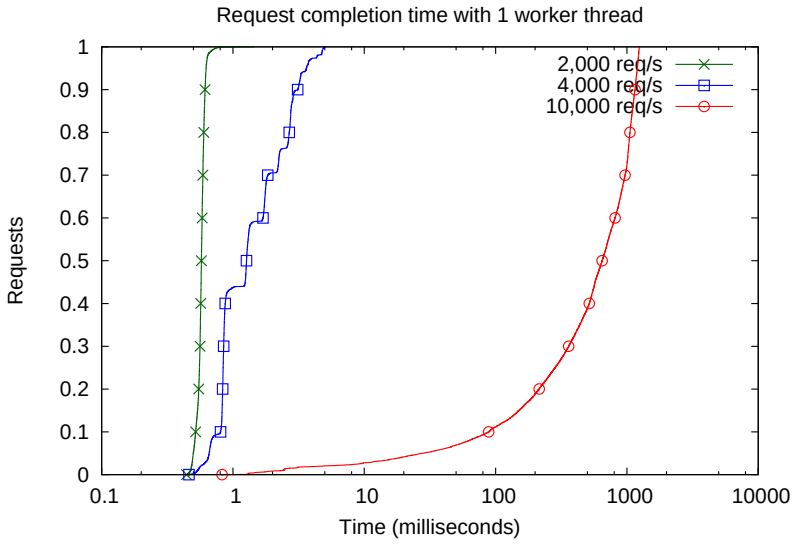


Figure 5.10: Request completion time with single worker thread for various loads.

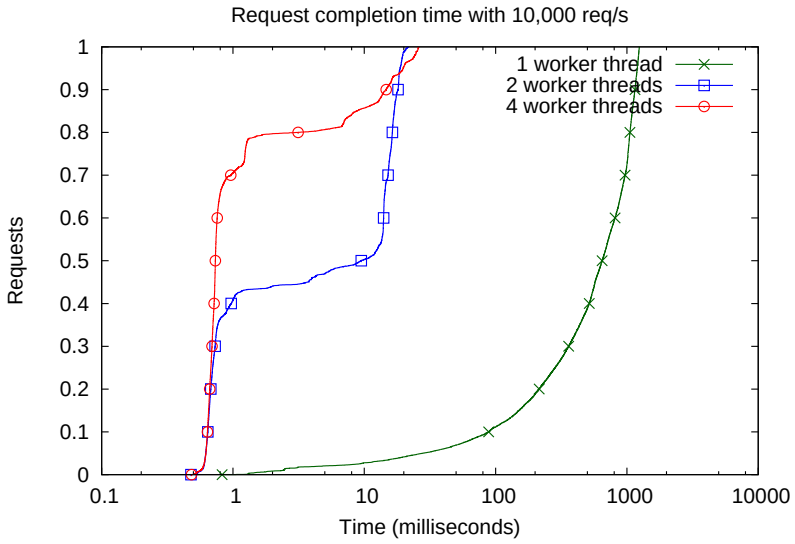


Figure 5.11: Request completion time with various worker threads for 10,000 reqs/s.

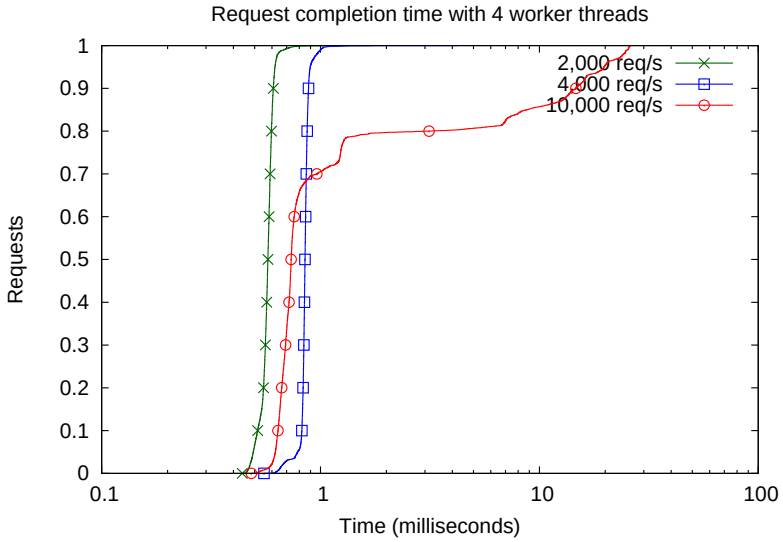


Figure 5.12: Request completion time with four worker threads for various loads.

a configurable uniform rate, and without path constraints. The source and destination of each request are selected at random. We measure the delay between the insertion of the request in the `ConvReq` table and the insertion of the conversation state in the `ConvState` table and plot this delay with the benchmarking tool. 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 the request and generation of the conversation state, including the path segmentation, (iv) the insertion of the conversation state into OVSDB, and (v) the reception of the `ConvState` entry by the benchmarking tool's monitor.

To estimate the load that an *SDN Resolver* 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 5.10.

At 2,000 requests per second, the requests are consistently processed at the sub-millisecond level. 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. We load this controller with 10,000 reqs/s. The results are shown in Figure 5.11. 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 5.12, 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 requests at 10,000 reqs/s are also completed within a millisecond.

Existing OpenFlow controllers such as Beacon and NOX have been shown to return responses at a higher rate [141, 142, 143]. However, *SDN Resolver* will receive several orders of magnitude less requests than a standard OpenFlow controller. Indeed, only conversation requests (*i.e.*, DNS requests) are handled by *SDN Resolver*. In contrast with an OpenFlow controller, an *SDN Resolver* does not need to act on a per-flow basis. Furthermore, DNS caching is known to perform very well [144] and directly applies to an *SDN Resolver*.

Those benchmarks are significant in a twofold manner. 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 power. 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 to be a signalling protocol, it can still be used as such without major performance issues, as proposed in [129].

Reaction to link failures

Link failures are inevitable in a network. When a link fails, the *SDN Resolver* potentially needs to recompute all the paths that were using the failed link. This recomputation is not required to preserve the connectivity since IGP convergence and SR-based fast reroute techniques cover this part of the recovery. We evaluate how quickly the controller handles the recomputation of conversa-

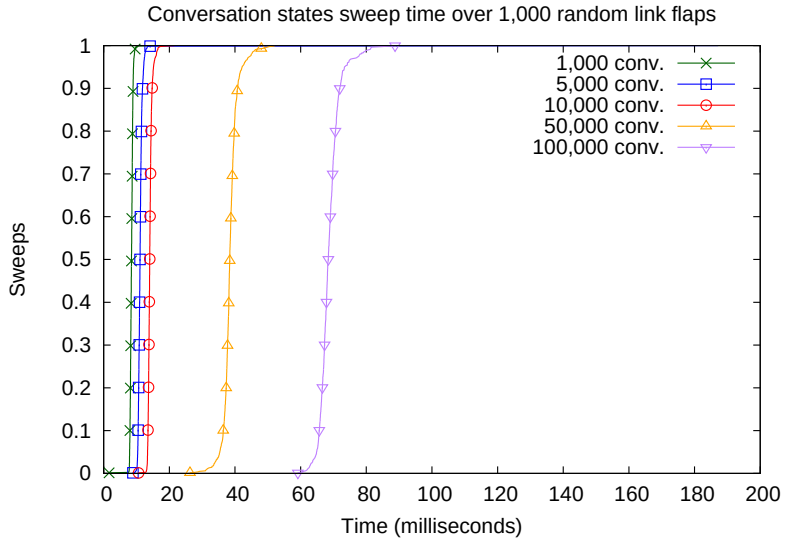


Figure 5.13: Time to sweep the full conversation states in a link failure event.

tions in case of link failures. Using the Abilene topology, we generated various numbers of conversation states with random sources and destinations, without path constraints. Then, we simulated random link flaps by updating entries in the `LinkState` table. When such an event happens, the controller sweeps the conversation states to decide which one must be recomputed. Without path constraints, the controller simply recomputes a new shortest path for conversations that were using the faulty link. The results are shown in Figure 5.13. We observe that there is a constant delay of approximately 5 milliseconds. This corresponds to the controller's recomputation timer, which enables to absorb bursts of link state changes. In average, the time spent on each conversation is approximately 0.6 microseconds. With a database of 100,000 active conversations, the average full sweep is completed within 65 milliseconds.

After the sweep, the controller recomputes the affected conversations. It takes about 0.2 milliseconds to recompute a single conversation. Considering 100,000 active conversations, in the extremely unlikely case where all the conversations are affected by a link failure, it would take 20 seconds to sequentially recompute all of them. This time can be reduced by distributing the recomputations over multiple threads.

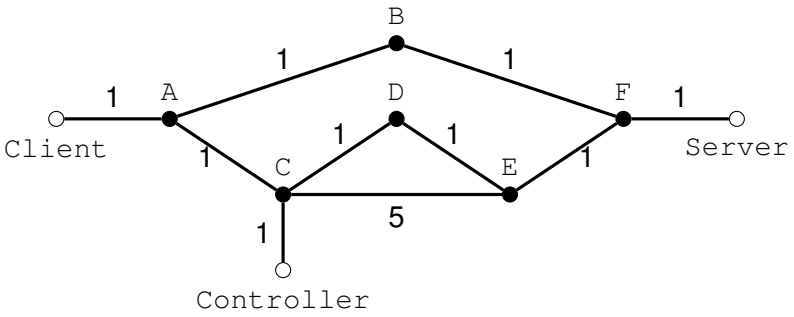


Figure 5.14: Emulated network topology. Each link has a unit weight. The numbers represent the link delays in milliseconds.

5.4.2 Emulated network

To evaluate our *SDN Resolver* architecture and implementation in an end-to-end setting, we instantiated a virtual network in a Mininet-like environment. The topology is shown in Figure 5.14. Each node is assigned an IPv6 prefix and contains pre-computed shortest-path routes towards each other node. The `Controller` node contains the main components of *SDN Resolver*, *i.e.*, the controller, DNS proxy, and OVSDB server. It also hosts a regular DNS server. 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 DNS forwarder and a routing daemon.

In a first experiment, 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 5.14. The round-trip time between the client and the controller is thus 6 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 30 milliseconds. The results are shown in Figure 5.15. We observe that the conversation request completion time consists of one RTT between the client and the controller, plus, on average, a constant overhead of 5 milliseconds. This overhead can be explained by the fluctuations of the virtualized environment. More importantly, the fact that conversation requests use one RTT shows that they will not be more affected by the network conditions than regular DNS requests.

In our setup, we did not explore the effect of DNS caching. The DNS proxy and forwarders could cache the controller-generated binding segments to speed up conversation requests for specific use cases. For example, if a client applica-

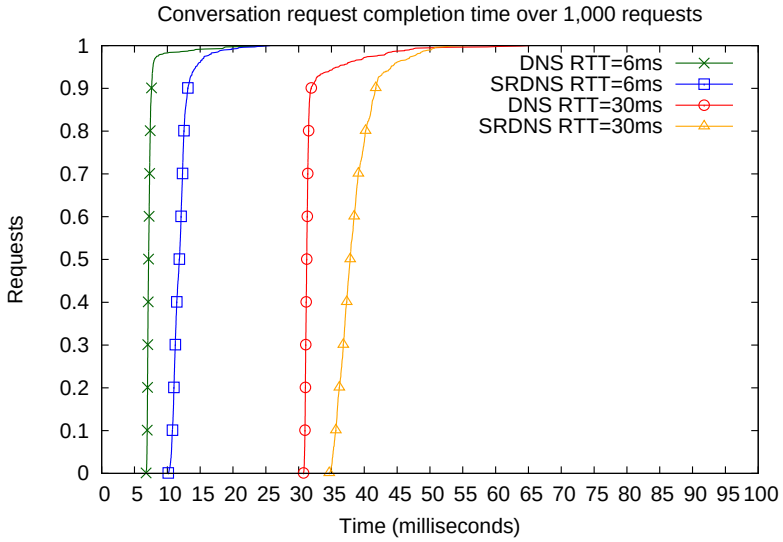


Figure 5.15: Conversation request completion time for various RTTs.

tion closes a conversation and quickly re-establishes it without changes in path constraints, then caching might be beneficial. The cache TTL should be low, to prevent from using obsolete network paths. In [144], the authors show that low TTL caching has no adverse effects on DNS resolvers hit rates.

In a second experiment, we observe the effects of link failures on active conversations. To realize this, we use the network topology as shown in Figure 5.14. First, we request a conversation between the client and the server, with a minimum-latency path constraint. The controller then configures router `A` with a binding segment B_1 mapping to the computed list of segments. In the initial network state, this list contains only the segment for node `F`. Indeed, the minimum latency path is also the shortest IGP path. Similarly, router `F` is configured with another binding segment B_2 mapping to a list of segments that consists of node `A`. The client node is configured to use the binding segment B_1 for all packets sent to the server node, and the server node is configured to use the binding segment B_2 for all packets sent to the client node. Then, from the client node, we run ICMPv6 echo-request measurements every 10 milliseconds. This precision was the best we could obtain with the `ping6` tool in our virtualized environment. Then, we repeatedly shut down the (`A` – `B`) link, wait for the controller reaction, then switch the link up again.

Figure 5.16 shows one cycle of this experiment. Between $t = 0$ and $t = 80$ ms,

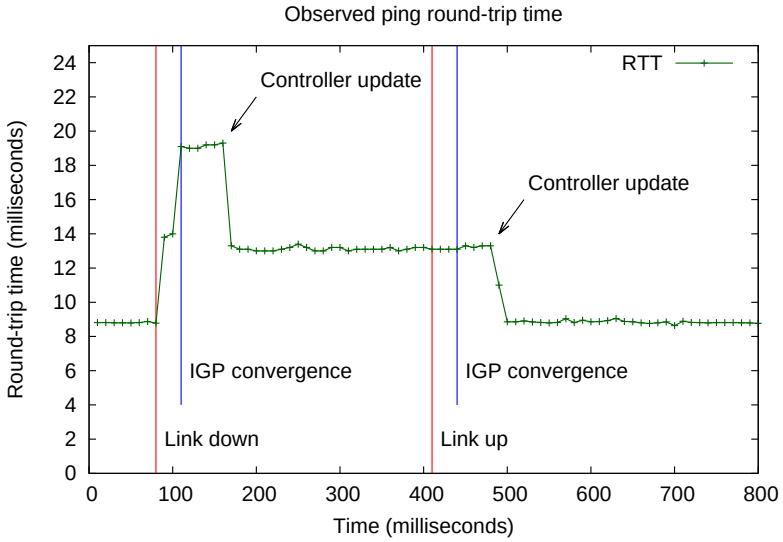


Figure 5.16: Controller reaction to link failure.

Table 5.4: Binding segments and paths evolution.

Event	B_1	B_2	Effective path	RTT
Initial	$\langle \mathtt{F} \rangle$	$\langle \mathtt{A} \rangle$	$(\mathtt{A}-\mathtt{B}-\mathtt{F})$	8 ms
Link down + IGP converg.	$\langle \mathtt{F} \rangle$	$\langle \mathtt{A} \rangle$	$(\mathtt{A}-\mathtt{C}-\mathtt{E}-\mathtt{F})$	18 ms
Controller update	$\langle \mathtt{D},\mathtt{F} \rangle$	$\langle \mathtt{D},\mathtt{A} \rangle$	$(\mathtt{A}-\mathtt{C}-\mathtt{D}-\mathtt{E}-\mathtt{F})$	12 ms
Link up + IGP converg.	$\langle \mathtt{D},\mathtt{F} \rangle$	$\langle \mathtt{D},\mathtt{A} \rangle$	$(\mathtt{A}-\mathtt{C}-\mathtt{D}-\mathtt{E}-\mathtt{F})$	12 ms
Controller update	$\langle \mathtt{F} \rangle$	$\langle \mathtt{A} \rangle$	$(\mathtt{A}-\mathtt{B}-\mathtt{F})$	8 ms

the path is the best possible one, *i.e.*, $(\mathtt{A}-\mathtt{B}-\mathtt{F})$ with a round-trip time of roughly 8 ms. At $t = 80$ ms, the link $(\mathtt{A}-\mathtt{B})$ is shut down. It takes about 30 milliseconds for the routes to change and the new path to be visible. The path converges to the now-shortest path, *i.e.*, $(\mathtt{A}-\mathtt{C}-\mathtt{E}-\mathtt{F})$. However, this path is also the longest-delay path with a round-trip time of about 18 ms. The controller is notified of the link failure and recomputes the affected paths. At $t = 170$ ms, the recomputed path is visible. During that time interval, (i) the controller was notified of the link failure, (ii) recomputed the new paths, (iii) it updated the `ConvState` table with the updated list of segments, (iv) the routing daemons on nodes `A` and `F` were notified of the update, and (v) they updated their respective routing table to reflect the new list of segments. The new lists of segments are now resp. $\langle \mathtt{D},\mathtt{F} \rangle$ and $\langle \mathtt{D},\mathtt{A} \rangle$ for nodes `A` and `F`, which correspond to the current minimum latency

path. The measured round-trip time is about 12 milliseconds. At t = 420 ms, the link (A – B) is brought back up and the new IGP routes converge at t = 450 ms. Note that the path does not change after the convergence. Indeed, the current lists of segments force the packets to transit through node D. However, the link state change triggers path recomputations and the controller's updates are visible at t = 480 ms. The packets then resume their original shortest and minimum latency path. Table 5.4 shows a summary of the binding segments and effective paths at each step.

5.5 Related and future work

Software Defined Networks have been a hot topic in the research community since the publication of [145]. Several survey papers have analyzed this vast literature in details [22, 146, 147]. In this section, we compare Software Resolved Networks (SRN) with several of the key related work. We structure the comparison according to the bottom-up approach adopted in section IV of [22].

The dataplane layer of SRNs differs from many proposed SDN solutions. SRNs operate in networks composed of IPv6 routers. To fully benefit from SRNs, the ingress routers need to support the binding segments that are part of SRv6 [19, 29]. Given that all IPv6 routers can forward packets with an SRH, it is possible to incrementally deploy SRNs starting with some upgraded hosts and ingress routers. Classical SDN solutions [20, 22] use flow tables installed on modified switches. Special solutions have been proposed to support legacy switches [148, 149].

A second difference between SRNs and OpenFlow-based SDNs is the southbound interface. SDNs rely on the OpenFlow protocol to configure flow tables on the switches. Future deployments could leverage more programmable switches [150]. 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 routers, 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. SRNs leverage Segment Routing to select and enforce network paths. Several SDN solutions also encode paths inside packets. Hari et al. propose in [151] 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 [152] 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. SRNs can also force specific conversations through middleboxes and pass parameters using TLVs inside the SRH [29].

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 [107] use a related approach. PANE proposes 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 [153, 154].

As future work, the next step should be to experiment with SRNs in real networks. This can be incrementally realized by implementing partial support for SRv6 in networks where IPv6 is already deployed. For example, one could upgrade a few edge routers to support SRv6, and deploy an *SDN Resolver* that would serve a subset of the network. Such experiments would provide insight on this novel architecture and create a feedback loop to improve SRNs and *SDN Resolver*. We discussed a few aspects of *SDN Resolver*, however, a lot of them are still unexplored. For example, operating and synchronizing multiple active controllers in a master-master fashion within a single network could lead to unexpected side effects and is worth exploring.

From an SRv6 point of view, SRNs could greatly benefit from the network programming model proposed in [29] and explained in Section 2.2. By implementing binding segments in the Linux kernel, we already leveraged a subset of this model. If fully implemented, such a model would enable network operators with even more flexibility to control and program their network. For example, *SDN Resolver* could dynamically steer conversations through a chain of virtual functions. The controller would then be able to change the parameters of those functions by updating the relevant segments in the SRv6 policy, according to, *e.g.*, the state of the network.

5.6 Conclusion

In this chapter, we presented 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 applications. For this, SRNs leverage the IPv6 Segment Routing architecture and the DNS protocol. There are two important differences between SRNs and SDNs. First, SRNs enable the applications to directly interact with the controller to specify path requirements like delay or bandwidth. This interaction is performed by using extensions to the DNS protocol. 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. We implement a complete prototype of an SRN network on Linux hosts, routers and servers. Our performance evaluation shows

that our prototype meets the performance expectations of enterprise networks.

Chapter 6

Conclusion

The scale and complexity of current networks warrant appropriate technologies and tools to operate them. A plethora of protocols and systems for network management rose and fell over the last few dozen years. Today, the ones that were selected and currently deployed in most production networks are not necessarily the best fit for their purposes. Indeed, multiple factors play a role in the adoption of technologies. Those factors do not always include elements suitable for future evolution of networks, such as scalability. Segment Routing is an attempt to bridge the gap between the distributed protocols managing the majority of current networks, and the paradigm shifting principles of Software-Defined Networking, as prominently spearheaded by OpenFlow.

In this thesis, we explored the potential of Segment Routing in its IPv6 flavor. SRv6 is an efficient and fine-grained network management architecture. Its source routing paradigm enables network operators to easily implement traffic engineering without adding state in the core network. By providing an abstraction layer over classical distributed protocols, SRv6 enables to centralize the network control while still relying on the resilience of such protocols. Another important feature of SRv6 is its inherent ability to extend its reach up to the endhosts. By bringing together the best of the distributed protocols and the SDN world, SRv6 paves the way for research areas hitherto unexplored. The main contributions of this thesis are the following.

In Chapter 3, we presented our open-source implementation of SRv6 in the Linux kernel. This implementation was merged into the mainline tree and is available in the official Linux kernel since v4.10. As SRv6 is still considered an experimental feature, a reference open-source implementation enables the practical validation of the SRv6 specifications before advancing further in the standardization process. Furthermore, it also enables other researchers to use, explore, and extend SRv6. We presented in details the different components of our implemen-

tation and evaluated its performance on real hardware. Experiments showed that our implementation yields performances very close to regular IPv6 forwarding for the main operations of SRv6, and is able to scale linearly with the available CPU resources. We also provided guidelines for future implementers and suggested some potential next steps to improve the SRv6 Linux implementation.

In Chapter 4, we explored how Segment Routing can help to solve specific networking issues in novel and efficient ways. We focused on two aspects in particular. First, we proposed to leverage traffic duplication over disjoint paths to enable robust low-latency communications for real-time applications. We leveraged SRv6 to steer the duplicated traffic over pre-computed disjoint paths. Simulations showed that the Linux TCP stack is able to cope with the duplicated traffic and absorb sudden jitter or packet losses. In a second aspect, we developed SCMon, a network monitoring technique that leverages the Segment Routing architecture to send probes over cycles, from a single vantage point. Those cycles enable to fully cover the network and explore ECMP components as well as specific links in a bundle. The result is a scalable solution that does not requires per-router configuration. Simulations on real topologies showed that SCMon is able to detect single-link failures within milliseconds.

In Chapter 5, we designed Software Resolved Networks, a new architecture for IPv6 enterprise networks. SRNs leverage SRv6 to enforce network paths. An SDN-like central controller, called the SDN Resolver, interacts with applications through the DNS protocol. The applications can specify path requirements for their flows, such as bandwidth or latency, by sending DNS requests. The controller then configures the access router of the requesting application with the appropriate SRv6 policy. A path identifier, implemented as a binding segment, is attached to this policy. This PathID is returned to the application, which subsequently uses this identifier to steer its packets through the corresponding network path. We extended the SRv6 implementation in the Linux kernel and implemented a complete prototype of an SDN Resolver. We evaluated our prototype through benchmarks and simulations, and showed that it meets the performance expectations of enterprise networks.

In the future, we expect that SRv6 will gain wider adoption and deployment. Through its unique properties, coupled with the SDN paradigm, we envision that a growing number of actors of the networking community will be able to reap the benefits of IPv6 Segment Routing.

Bibliography

[1] Jon Postel. Internet Protocol. RFC 791, September 1981.

[2] Steve E. Deering. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, December 1998.

[3] Christian Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, November 2000.

[4] C. Hedrick. Routing Information Protocol. RFC 1058, June 1988.

[5] John Moy. OSPF Version 2. RFC 2328, April 1998.

[6] Dennis Ferguson et al. OSPF for IPv6. RFC 5340, October 2015.

[7] ISO/IEC. Information technology – Intermediate System to Intermediate System intra-domain routeing. ISO/IEC 10589:2002, March 2008.

[8] Yakov Rekhter, Susan Hares, and Dr. Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[9] Arun Viswanathan, Eric C. Rosen, and Ross Callon. Multiprotocol Label Switching Architecture. RFC 3031, January 2001.

[10] Dan Tappan, Yakov Rekhter, Alex Conta, Guy Fedorkow, Eric C. Rosen, Dino Farinacci, and Dr. Tony Li. MPLS Label Stack Encoding. RFC 3032, January 2001.

[11] Bob Thomas, Loa Andersson, and Ina Minei. LDP Specification. RFC 5036, October 2007.

[12] Robert T. Braden, Lixia Zhang, Steven Berson, Shai Herzog, and Sugih Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205, September 1997.

[13] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Dr. Tony Li, Dr. Vijay Srinivasan, and George Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209, December 2001.

[14] Adrian Farrel, Olufemi Komolafe, and Seisho Yasukawa. An Analysis of Scaling Issues in MPLS-TE Core Networks. RFC 5439, February 2009.

[15] Clarence Filsfils, Nagendra Kumar Nainar, Carlos Pignataro, Juan Camilo Cardona, and Pierre Francois. The segment routing architecture. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–6. IEEE, 2015.

[16] George Neville-Neil, Pekka Savola, and Joe Abley. Deprecation of Type 0 Routing Headers in IPv6. RFC 5095, December 2007.

[17] Clarence Filsfils, Stefano Previdi, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment Routing Architecture. Internet-Draft draft-ietf-spring-segment-routing-11, Internet Engineering Task Force, February 2017. Work in Progress.

[18] Clarence Filsfils, Pierre Francois, Stefano Previdi, Bruno Decraene, Stephane Litkowski, Martin Horneffer, Igor Milojevic, Rob Shakir, Saku Ytti, Wim Henderickx, Jeff Tantsura, Sriganesh Kini, and Edward Crabbe. Segment Routing Use Cases. Internet-Draft draft-filsfils-spring-segment-routing-use-cases-01, Internet Engineering Task Force, October 2014. Work in Progress.

[19] Stefano Previdi, Clarence Filsfils, Brian Field, Ida Leung, J. Linkova, Ebben Aries, Tomoya Kosugi, Eric Vyncke, and David Lebrun. IPv6 Segment Routing Header (SRH). Internet-Draft draft-ietf-6man-segment-routing-header-05, Internet Engineering Task Force, February 2017. Work in Progress.

[20] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM '07*, pages 1–12, New York, NY, USA, 2007. ACM.

[21] Nick McKeown et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[22] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[23] Stefano Previdi, Clarence Filsfils, Bruno Decraene, Stephane Litkowski, Martin Horneffer, and Rob Shakir. Source Packet Routing in Networking (SPRING) Problem Statement and Requirements. RFC 7855, May 2016.

[24] Clarence Filsfils, Stefano Previdi, Ahmed Bashandy, Bruno Decraene, Stephane Litkowski, Martin Horneffer, Edward Crabbe, Jeff Tantsura, and Rob Shakir. Segment Routing with MPLS data plane. Internet-Draft draft-ietf-spring-segment-routing-mpls-07, Internet Engineering Task Force, February 2017. Work in Progress.

[25] Stefano Previdi, Clarence Filsfils, Ahmed Bashandy, Stephane Litkowski, Bruno Decraene, Jeff Tantsura, and Hannes Gredler. IS-IS Extensions for Segment Routing. Internet-Draft draft-ietf-isis-segment-routing-extensions-10, Internet Engineering Task Force, February 2017. Work in Progress.

[26] Peter Psenak, Stefano Previdi, Clarence Filsfils, Jeff Tantsura, Wim Henderickx, Hannes Gredler, and Rob Shakir. OSPF Extensions for Segment Routing. Internet-Draft draft-ietf-ospf-segment-routing-extensions-11, Internet Engineering Task Force, February 2017. Work in Progress.

[27] Daniel Voyer, Daniel Bernier, John Leddy, Clarence Filsfils, Stefano Previdi, Stefano Salsano, Antonio Cianfrani, David Lebrun, Olivier Bonaventure, Prem Jonnalagadda, Milad Sharif, Hani Elmalky, Ahmed Abdelsalam, Robert Raszuk, Arthi Ayyangar, Dirk Steinberg, and Wim Henderickx. Insertion of IPv6 Segment Routing Headers in a Controlled Domain. Internet-Draft draft-voyer-6man-extension-header-insertion-00, Internet Engineering Task Force, March 2017. Work in Progress.

[28] Shane Amante, Jarno Rajahalme, Sheng Jiang, and Brian E. Carpenter. IPv6 Flow Label Specification. RFC 6437, November 2011.

[29] Clarence Filsfils, John Leddy, Daniel Voyer, Daniel Bernier, Dirk Steinberg, Robert Raszuk, Satoru Matsushima, David Lebrun, Bruno Decraene, Bart Peirens, Stefano Salsano, Gaurav Naik, Hani Elmalky, Prem Jonnalagadda, Milad Sharif, Arthi Ayyangar, Satish Mynam, Ahmed Bashandy, Kamran Raza, Darren Dukes, Francois Clad, and Pablo Camarillo Garvia. SRv6 Network Programming. Internet-Draft draft-filsfils-spring-srv6-network-programming-00, Internet Engineering Task Force, March 2017. Work in Progress.

[30] Randeep Bhatia, Fang Hao, Murali Kodialam, and TV Lakshman. Optimized network traffic engineering using segment routing. In *Computer*

Communications (INFOCOM), 2015 IEEE Conference on, pages 657–665. IEEE, 2015.

[31] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *SIGCOMM '15*, pages 15–28, New York, NY, USA, 2015. ACM.

[32] Fang Hao, Murali Kodialam, and TV Lakshman. Optimizing restoration with segment routing. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[33] Luca Davoli, Luca Veltri, Pier Luigi Ventre, Giuseppe Siracusano, and Stefano Salsano. Traffic engineering with segment routing: Sdn-based architectural design and open source implementation. In *Software Defined Networks (EWSDN), 2015 Fourth European Workshop on*, pages 111–112. IEEE, 2015.

[34] Alessio Giorgetti, Piero Castoldi, Filippo Cugini, Jeroen Nijhof, Francesco Lazzeri, and Gianmarco Bruno. Path encoding in segment routing. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–6. IEEE, 2015.

[35] Stefano Salsano, Luca Veltri, Luca Davoli, Pier Luigi Ventre, and Giuseppe Siracusano. Pmsrpoor man's segment routing, a minimalistic approach to segment routing and a traffic engineering use case. In *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*, pages 598–604. IEEE, 2016.

[36] David Miller. IPv6 SR merge commit into net-next. https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/commit/?id=5db5b395150186d4a177ebfa563894af302ab3ad, November 2016.

[37] Linux community. Linux 4.10 ChangeLog. https://kernelnewbies.org/Linux_4.10, February 2017.

[38] David Lebrun and Olivier Bonaventure. Implementing IPv6 Segment Routing in the Linux Kernel. In *Proceedings of the 2017 Applied Networking Research Workshop*. ACM, 2017.

[39] Jonathan Corbet. JLS2009: Generic receive offload. https://lwn.net/Articles/358910/.

[40] Michael Kerrisk. Namespaces in operation, part 1: namespaces overview. https://lwn.net/Articles/531114/.

[41] Jonathan Corbet. Network namespaces. https://lwn.net/Articles/219794/.

[42] N. Handigol et al. Reproducible network experiments using container-based emulation. In *CoNEXT*, 2012.

[43] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. *IACR Cryptology ePrint Archive*, 2017:190, 2017.

[44] Paul McKenney and Jonathan Walpole. What is RCU, Fundamentally? https://lwn.net/Articles/262464/, December 2007.

[45] Paul McKenney and Jonathan Walpole. What is RCU? Part 2: Usage. https://lwn.net/Articles/263130/, January 2008.

[46] Paul McKenney and Jonathan Walpole. RCU part 3: the RCU API. https://lwn.net/Articles/264090/, January 2008.

[47] Kathleen Nichols and Van Jacobson. Controlling queue delay. *Commun. ACM*, 55(7):42–50, July 2012.

[48] Jonathan Corbet. TSO sizing and the FQ scheduler. https://lwn.net/Articles/564978/, August 2013.

[49] Dave Taht, Toke Hoeiland-Joergensen, Paul McKenney, Jim Gettys, and Eric Dumazet. The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm. Internet-Draft draft-ietf-aqm-fq-codel-06, Internet Engineering Task Force, March 2016. Work in Progress.

[50] Tom Herbert and Willem de Bruijn. Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[51] Eric Dumazet. Busypolling next generation. http://netdevconf.org/2.1/slides/apr6/dumazet-BUSY-POLLING-Netdev-2.1.pdf.

[52] Robert Olsson. Pktgen the linux packet generator. In *Proceedings of the Linux Symposium, Ottawa, Canada*, volume 2, pages 11–24, 2005.

[53] Cisco Systems. Segment Routing - The Fast Data Project (FD.io). http://www.segment-routing.net/open-software/vpp/.

[54] Roopa Prabhu. MPLS tutorial. http://www.netdevconf.org/1.1/proceedings/slides/prabhu-mpls-tutorial.pdf.

[55] Tom Herbert and Petr Lapukhov. Identifier-locator addressing for IPv6. Internet-Draft draft-herbert-nvo3-ila-04, Internet Engineering Task Force, March 2017. Work in Progress.

[56] Jonathan Corbet. Identifier Locator addressing. https://lwn.net/Articles/657012/.

[57] Ramon R Fontes, Samira Afzal, Samuel HB Brito, Mateus AS Santos, and Christian Esteve Rothenberg. Mininet-wifi: Emulating software-defined wireless networks. In *Network and Service Management (CNSM), 2015 11th International Conference on*, pages 384–389. IEEE, 2015.

[58] Jeff Ahrenholz. Comparison of core network emulation platforms. In *Military Communications Conference, 2010-MILCOM 2010*, pages 166–171. IEEE, 2010.

[59] University of Zagreb. Integrated Multiprotocol Network Emulator/Simulator. http://imunes.net/.

[60] Maurizio Pizzonia and Massimo Rimondini. Netkit: network emulation for education. *Software: Practice and Experience*, 46(2):133–165, 2016.

[61] Jeff Dike et al. User-mode linux. In *Annual Linux Showcase & Conference*, 2001.

[62] Jeremy Grossmann. Graphical Network Simulator-3. https://www.gns3.com/.

[63] Dynamips. https://github.com/GNS3/dynamips/.

[64] Tomas Hruby, Cristiano Giuffrida, Lionel Sambuc, Herbert Bos, and Andrew S. Tanenbaum. A neat design for reliable and scalable network stacks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 359–373, New York, NY, USA, 2016. ACM.

[65] Tom Herbert, Lucy Yong, and Osama Zia. Generic UDP Encapsulation. Internet-Draft draft-ietf-intarea-gue-01, Internet Engineering Task Force, March 2017. Work in Progress.

[66] François Aubry, David Lebrun, Yves Deville, and Olivier Bonaventure. Traffic duplication through segmentable disjoint paths. In *IFIP Networking Conference (IFIP Networking), 2015*, pages 1–9. IEEE, 2015.

[67] Francois Aubry, David Lebrun, Stefano Vissicchio, Minh Thanh Khong, Yves Deville, and Olivier 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*, pages 1–9, 2016.

[68] J. Adler. Raging bulls: How wall street got addicted to light-speed trading. *Wired*, Aug. 2012.

[69] R. Martin. Wall street's quest to process data at the speed of light. *Information Week*, 2007.

[70] A. Vulimiri, P. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *CoNEXT '13*, 2013.

[71] B. Briscoe et al. Reducing internet latency: A survey of techniques and their merits. *Communications Surveys Tutorials, IEEE*, 2014.

[72] Debasis Mandal and Bivas Mitra. Shared path protection in dwdm mesh networks. In *8th International Conference on the Information Technology, Bhubaneswar, India*, 2005.

[73] Wen-Ping Chen, Fu-Hung Shih, Wen-Shyang Hwang, et al. The multiple path protection of dwdm backbone optimal networks. *J. Inf. Sci. Eng.*, 25(3):733–745, 2009.

[74] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, February 2004.

[75] QEMU, the FAST! processor emulator. http://www.qemu.org.

[76] Kernel Virtual Machine. http://linux-kvm.org.

[77] Ernst W Biersack. Performance evaluation of forward error correction in an atm environment. *Selected Areas in Communications, IEEE Journal on*, 11(4):631–640, 1993.

[78] Jay Kumar Sundararajan, Devavrat Shah, Muriel Médard, Szymon Jakubczak, Michael Mitzenmacher, and Joao Barros. Network coding meets tcp: Theory and implementation. *Proceedings of the IEEE*, 99(3):490–512, 2011.

[79] Colin Perkins, Orion Hodson, and Vicky Hardman. A survey of packet loss recovery techniques for streaming audio. *Network, IEEE*, 12(5):40–48, 1998.

[80] M. Balakirshnan et al. Maelstrom: Transparent error correction for communication between data centers. In *IEEE/ACM Trans. on Networking*, 2011.

[81] Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clémence Magnien, and Renata Teixeira. Avoiding traceroute anomalies with paris traceroute. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 153–158. ACM, 2006.

[82] Brice Augustin, Timur Friedman, and Renata Teixeira. Measuring Load-balanced Paths in the Internet. In *IMC*, pages 149–160, 2007.

[83] Cristel Pelsser, Luca Cittadini, Stefano Vissicchio, and Randy Bush. From Paris to Tokyo: On the Suitability of ping to Measure Latency. In *IMC*, 2013.

[84] D. Katz and D. Ward. Bidirectional forwarding detection (bfd). RFC 5880, 2010.

[85] M. Chiba, A. Clemm, S. Medley, J. Saloway, S. Thombare, and E. Yedavalli. Cisco service-level assurance protocol. RFC 6812, 2013.

[86] Ross Cartlidge and Nicolas Guilbaud. Topology Aware Blackbox Monitoring. NANOG presentation, 2013.

[87] A. Kvalbein et al. Fast IP Network Recovery Using Multiple Routing Configurations. In *INFOCOM*, 2006.

[88] R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 2011.

[89] OVH. Ovh network weathermap. http://weathermap.ovh.net.

[90] Peng Wu et al. Alarm correlation engine (ACE). In *NOMS*, 1998.

[91] He Yan et al. G-RCA: A Generic Root Cause Analysis Platform for Service Quality Management in Large IP Networks. In *CoNEXT*, 2010.

[92] Ramana Rao Kompella et al. Detection and Localization of Network Black Holes. In *INFOCOM*, 2007.

[93] N. Duffield, F. Lo Presti, V. Paxson, and D. Towsley. Network loss tomography using striped unicast probes. *IEEE/ACM Transactions on Networking*, 14(4):697–710, Aug 2006.

[94] Liang Ma et al. On optimal monitor placement for localizing node failures via network tomography. *Performance Evaluation*, 2015.

[95] Y. Breitbart et al. Efficiently monitoring bandwidth and latency in IP networks. In *INFOCOM*, 2001.

[96] Mehdi Nikkhah and Roch Guérin. Migrating the internet to ipv6: an exploration of the when and why. *IEEE/ACM Transactions on Networking*, 24(4):2291–2304, 2016.

[97] Peng Wu, Yong Cui, Jianping Wu, Jiangchuan Liu, and Coert Metz. Transition from ipv4 to ipv6: A state-of-the-art survey. *Communications Surveys & Tutorials, IEEE*, 15(3):1407–1424, 2013.

[98] Lorenzo Colitti, Steinar H Gunderson, Erik Kline, and Tiziana Refice. Evaluating ipv6 adoption in the internet. In *Passive and active measurement*, pages 141–150. Springer, 2010.

[99] Jakub Czyz, Mark Allman, Jing Zhang, Scott Iekel-Johnson, Eric Osterweil, and Michael Bailey. Measuring ipv6 adoption. *ACM SIGCOMM Computer Communication Review*, 44(4):87–98, 2015.

[100] Haythum Babiker, Irena Nikolova, and Kiran Kumar Chittimaneni. Deploying ipv6 in the google enterprise network. lessons learned. In *Proceedings of the 25th international conference on Large Installation System Administration, LISA*, page 10, 2011.

[101] T. Hollmann. A history of IPv6 challenges in facebook data centers. Network @Scale conference, 2016.

[102] F. Martin. Ipv6 inside linkedin part ii. https://engineering.linkedin.com/blog/2016/08/ipv6-inside-linkedin-part-ii–back-to-the-future, 2016.

[103] Marcus Keane. IPv6-only at Microsoft. https://blog.apnic.net/2017/01/19/ipv6-only-at-microsoft/, January 2017.

[104] David A Maltz, Geoffrey Xie, Jibin Zhan, Hui Zhang, Gísli Hjálmtỳsson, and Albert Greenberg. Routing design in operational networks: A look from the inside. In *SIGCOMM'04*, pages 27–40. ACM, 2004.

[105] Yu-Wei Eric Sung, Xin Sun, Sanjay G Rao, Geoffrey G Xie, and David A Maltz. Towards systematic design of enterprise networks. *IEEE/ACM Transactions on Networking (TON)*, 19(3):695–708, 2011.

[106] Justine Sherry et al. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[107] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An api for application control of sdns. *SIGCOMM Comput. Commun. Rev.*, 43(4):327–338, August 2013.

[108] Lorenzo Colitti, Dr. Vinton G. Cerf, Stuart Cheshire, and David Schinazi. Host Address Availability Recommendations. RFC 7934, July 2016.

[109] Paul A. Vixie et al. Extension Mechanisms for DNS (EDNS(0)). RFC 6891, October 2015.

[110] Dr. Thomas Narten, Richard P. Draves, and Suresh Krishnan. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 4941, September 2007.

[111] Yakov Rekhter and Jim Bound. Dynamic Updates in the Domain Name System (DNS UPDATE). RFC 2136, April 1997.

[112] Brian Wellington. Secure Domain Name System (DNS) Dynamic Update. RFC 3007, November 2000.

[113] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 213–226. ACM, 2014.

[114] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 109–114. ACM, 2013.

[115] IEEE. 802.1X - Port Based Network Access Control. http://www.ieee802.org/1/pages/802.1x.html.

[116] William A. Simpson, Dr. Thomas Narten, Erik Nordmark, and Hesham Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861, September 2007.

[117] Syam Madanapalli, Jaehoon Jeong, Soohong Daniel Park, and Luc Beloeil. IPv6 Router Advertisement Options for DNS Configuration. RFC 6106, November 2010.

[118] Alia Atlas et al. OSPF Traffic Engineering (TE) Metric Extensions. RFC 7471, October 2015.

[119] Zheng Wang, Mark A. Carlson, Walter Weiss, Elwyn B. Davies, and Steven L. Blake. An Architecture for Differentiated Services. RFC 2475, March 2013.

[120] Daniel B. Grossman. New Terminology and Clarifications for Diffserv. RFC 3260, April 2002.

[121] Robert T. Braden, Dr. David D. Clark, and Scott Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633, March 2013.

[122] Victor Firoiu, Dr. Bruce S. Davie, and Anna Charny. An Expedited Forwarding PHB (Per-Hop Behavior). RFC 3246, March 2002.

[123] Walter Weiss, Dr. Juha Heinanen, Fred Baker, and John T. Wroclawski. Assured Forwarding PHB Group. RFC 2597, June 1999.

[124] Fred Baker, David L. Black, Dr. Kathleen M. Nichols, and Steven L. Blake. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, December 1998.

[125] Fred Baker, Jozef Babiarz, and Kwok Ho Chan. Configuration Guidelines for DiffServ Service Classes. RFC 4594, August 2006.

[126] Dr. Lixia Zhang et al. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205, March 2013.

[127] Fred Baker, Dr. Bruce S. Davie, and Carol Iturralde. Aggregation of RSVP for IPv4 and IPv6 Reservations. RFC 3175, September 2001.

[128] Ben Pfaff and Bruce Davie. The Open vSwitch Database Management Protocol. RFC 7047, December 2013.

[129] Bruce Davie, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Natasha Gude, Amar Padmanabhan, Tim Petty, Kenneth Duda, and Anupam Chanda. A database approach to sdn control plane design. *SIGCOMM Comput. Commun. Rev.*, 47(1):15–26, January 2017.

[130] Stephen Nadas. Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. RFC 5798, March 2010.

[131] Pierre Francois, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving sub-second igp convergence in large ip networks. *SIGCOMM Comput. Commun. Rev.*, 35(3):35–44, July 2005.

[132] Stefano Salsano, Alessio Botta, Paola Iovanna, Marco Intermite, and Andrea Polidoro. Traffic engineering with ospf-te and rsvp-te: Flooding reduction techniques and evaluation of processing cost. *Computer Communications*, 29(11):2034 – 2045, 2006.

[133] Hilmi E Egilmez, S Tahsin Dane, K Tolga Bagci, and A Murat 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*, pages 1–8. IEEE, 2012.

[134] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. *ACM SIGCOMM computer communication review*, 43(4):27–38, 2013.

[135] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[136] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. Elasticon: an elastic distributed sdn controller. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 17–28. ACM, 2014.

[137] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. DNS Security Introduction and Requirements. RFC 4033, March 2005.

[138] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. Protocol Modifications for the DNS Security Extensions. RFC 4035, March 2005.

[139] D. Sternberg and others. C-Ares. https://c-ares.haxx.se.

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

[141] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. *Hot-ICE*, 12:1–6, 2012.

[142] David Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2013.

[143] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. Advanced study of sdn/openflow controllers. In *Proceedings of the 9th central & eastern european software engineering conference in russia*, page 1. ACM, 2013.

[144] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. Dns performance and the effectiveness of caching. *IEEE/ACM Transactions on networking*, 10(5):589–603, 2002.

[145] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[146] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn. *Queue*, 11(12):20, 2013.

[147] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials*, 17(1):27–51, 2015.

[148] Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, and Anja Feldmann. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 333–345, Philadelphia, PA, June 2014. USENIX Association.

[149] Cheng Jin, Cristian Lumezanu, Qiang Xu, Hesham Mekky, Zhi-Li Zhang, and Guofei Jiang. Magneto: Unified fine-grained path control in legacy and openflow hybrid networks. In *Proceedings of the Symposium on SDN Research*, pages 75–87. ACM, 2017.

[150] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[151] Adiseshu Hari, TV Lakshman, and Gordon 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*, page 36. ACM, 2015.

[152] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. *ACM SIGCOMM Computer Communication Review*, 44(4):3–14, 2015.

[153] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 181–192. ACM, 2007.

[154] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.