UCL
Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Service Function Chaining with Segment Routing

Dissertation presented by
Thibault GÉRONDAL , Nicolas HOUTAIN

for obtaining the Master's degree in
Computer Science
Options: Networking & Development

Supervisors
Olivier BONAVENTURE , David LEBRUN

Readers
Olivier BONAVENTURE, Marco CANINI , David LEBRUN

Academic year 2015-2016

ABSTRACT

This Master Thesis covers multiple aspects for implementing Service Function Chaining (SFC) with Segment Routing (SR). As a great majority of networks perform a hop-by-hop routing based on a routing protocol without any path-control, Segment Routing has been introduced to address this problem. It to allows network devices to steer a packet through a controlled set of instructions. A Segment Routing Header based service function chaining architecture develops the ability of creating a chain of connected network services but it requires an SR-domain where every service is able to deal with the Segment Routing Header, plus some ingress and egress nodes which are respectively responsible for encapsulating/decapsulating the traffic. SFC is useful to steer traffic through an ordered list of service functions which apply some processing on the traffic. Network operators frequently deploy services in their network such as packet filtering, loadbalancing, proxies or other on-demand services. They act like black boxes inside the network by acting on the traffic steered through them. Those services can be deployed based on Network Function Virtualization (NFV). Architecture based on SFC and Network function virtualization enables to implement, deploy and maintain services with a high flexibility. Currently, Service Function Chaining and Segment Routing are not widely developed but generate interest in the academic and industry.

We will explore and develop services that act on the user space, enabling development with ease. Moreover, we provided a development environment to create with ease new services, plus a development testing framework called Segment Routing Functional Testing which allows to easily test the proper functioning of Service Function Chaining with Segment Routing. We rely on the seg6ctl library which implements a memory-mapped NETLINK-based data transfer mechanism between kernel space and user space.

ACKNOWLEDGMENTS

Many thanks to David Lebrun, Olivier Bonaventure, Aude Gérondal, Nathalie Franssen, Sarah Houtain, Newton Von Pemak, Mark Penny, Jordan Giovanola and Denis Vermylen.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LISTINGS

Part I

INTRODUCTION

1 INTRODUCTION

A great majority of networks perform a hop-by-hop routing based on a routing protocol. Each router, based on the destination address and its own knowledge of the network, decides the next hop that a packet will take. This means that it is difficult to accurately predict beforehand the path of a packet, which depends on the forwarding state of each traversed router. However, being able to steer a packet through a defined list of nodes can be useful. Segment Routing (SR) has been introduced to address this issue by offering a way to control the path followed by a packet.

1.1 SEGMENT ROUTING

Segment Routing is a new technology that allows network devices to steer a packet through a controlled set of instructions, called *segments*, by prepending a Segment Routing Header (SRH) to the packet. A segment can represent any instruction, topological or service-based. Figure 1 displays a packet prepended with a SRH containing two segments (B and C). When a packet enters an Segment Routing domain (SR-domain), the ingress node can add an SRH to the packet. Thus, the nodes in the core can be stateless, since forwarding instructions are carried by the packet to enforce a flow through any path (topological, or application/service based) while maintaining per-flow state only at the ingress node of the SR-domain [IETF SR 08].

Segment routing introduces the concept of a Segment Routing domain (SR-domain) as the set of nodes participating in the source based routing model. These nodes may be connected to the same physical infrastructure (e.g. a Service Provider's network) as well as nodes remotely connected to each other (e.g. an enterprise VPN or an overlay).

Segment Routing can be used with MPLS or applied to the IPv6 data plane with the addition of a new type of Routing Extension Header [IETF SR 08].

We can distinguish several kinds of segments. In this section, we discuss four of them : *Node segment*, *Adjacency segment*, *Anycast segment*

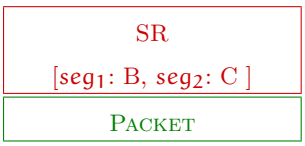


Figure 1: A packet prepend by an Segment Routing Header with two segments : B and C.

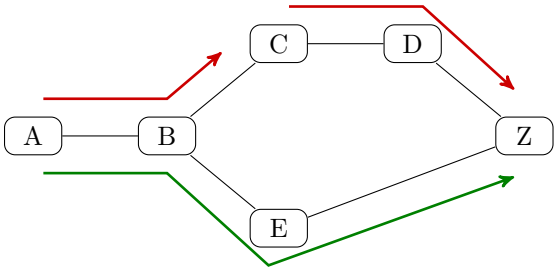


Figure 2: Green flow shows the shortest path used without Segment Routing. Red flow shows the path used by a flow prepend by a Segment Routing Header containing C.

and *Service segment*. In the following explanations, each node forwards packets using the shortest path assuming that each edge has the same cost.

1.1.1 *Node segment*

When a *Node Segment* is being used on a packet, this packet will have to go through the specified node in the Segment Routing Header (SRH). Figure 2 shows an example where packets go from node A to node Z. Green arrows depict the path used by packets from A to Z without Segment Routing. As it consists in a hop-by-hop routing, A has no control over the packet's path.

If the path between B and E is frequently congested, we might want to indicate to packets that they should use another path. If we prepend packets with a SRH containing the segment C, we can enforce a flow through this particular segment. The Red arrows depict the path that packets will use in this case. Indeed, they will reach C by using the shortest path from A and then Z using the shortest path from C.

1.1.2 *Adjacency Segments*

Figure 3 depicts in red the path taken by packets with a Segment Routing Header containing the segment C and D. Assuming link C-G is also frequently congested, adding this SRH does not fix the issue as the shortest path to reach Z from C uses the link C-G as the weight on C-D prevents the flow to use this link.

To address this problem, we can add an *Adjacency segment* in the SRH composed of C and Y. The *Adjacency segment* (Y) only has to be known by its predecessor segment (C); there is no obligation to be known by the whole network. In general, C would be set up such that it always prefers to reach Y through link C-D. To achieve this, D advertises the address Y only through link C-D, meaning that packets destined to Y must pass through C-D. This path is shown in green in the Figure 3.

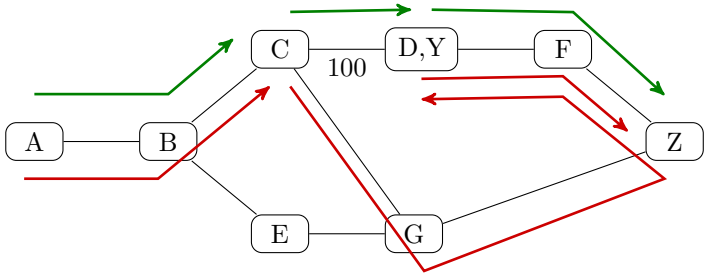


Figure 3: Green flow shows the shortest path with a Segment Routing Header (SRH) containing the segment C and the Adjancy segment Y. Red flow shows the path but with SRH segments C and D.

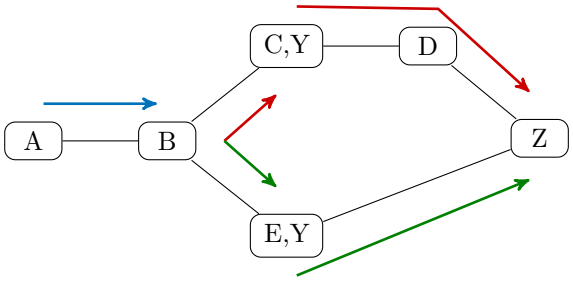


Figure 4: *Anycast segments* allows to do load-balancing between flows on a SR-domain using Equal-Cost Multi-Path (ECMP) routing.

1.1.3 *Anycast Segments*

Two or more nodes can have the same Node Segment label. In Figure 4, nodes C and E are also labelled Y. This makes it possible to have an *Anycast segments* to spread the traffic between two links of the network. Network operator can spread traffic over two links and can short-cut the control plane as shown in Figure 4 by prepending a SRH containing B and Y. Indeed, when the packet arrives at node B, the node will see that there are two paths with the same cost to reach the next destination (Y). This node will use Equal-Cost Multi-Path (ECMP) routing to load-balance the traffic across its two links reaching Y.

1.1.4 *Service Segments*

The purpose of *Service segments* is to provide a service when packets pass through them. We will detail those *Service segments* in the chapter 1.2 as those kinds constitute the main subject of this Master Thesis. To illustrate the use of a Service Segment, let us imagine a firewalling service. The aim of a firewall service is to decide if a traffic flow can cross a network or not. Figure 5 shows such a service. The green flow is not blocked by the firewall and can pass through the service whereas the red one is discarded. In this case, these flows must have a SRH in order to be processed by our service.

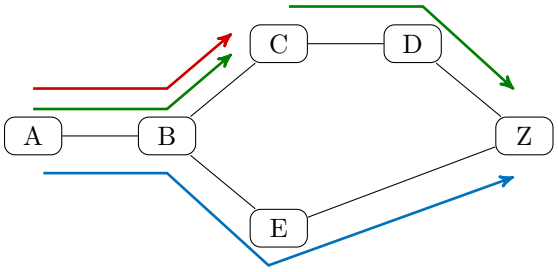


Figure 5: A service segment provides functional service on the network. This figure describes a firewall service.

1.2 SERVICE FUNCTION CHAINING

Network operators frequently deploy services in their network such as packet filtering, load-balancing, proxies or other on-demand services. They act like black boxes inside the network by acting on the traffic steered through them. Service Function Chaining (SFC) is the process of steering traffic through an ordered list of service functions. Such a steering can be performed in multiple ways. In this thesis, we leverage the Segment Routing architecture.

In Segment Routing, each service is represented by a segment. The Segment Routing architectural design naturally enables the implementation of SFC. Indeed, packets in an SR-domain visit the list of segments according to their SRH. SFC can provide a lot of services such as content caches, firewalling or DPI that can be chained according to a set of policies. The deployment of those services is gradually shifting from a hardware-based, vendor-specific appliance, to virtualized applications that run on commodity hardware. This kind of architecture is called Network Function Virtualization (NFV).

The combination of SFC and NFV enables the operators to implement, deploy and maintain services with a high flexibility, and therefore reducing capital and operational expenditures.

In this thesis, the Virtual Network Functions (VNFs) (also called services) are implemented with a Linux userland program. The packet steering mechanism from the kernel to the userland is explained in the chapter 4.

Figure 6 describes how SFC with Segment Routing is done. We imagine two hosts A and B that transmit traffic to their respective destinations C and D. In our example, A is a child and hence is subject to a *Parental Control* service (PC). The traffic is steered through an ingress node of a SR-domain. The ingress node will define which services will be applied to the packet by adding the appropriate segments to the SRH. A mapping can be defined on ingress nodes to define appropriate services for each flow.

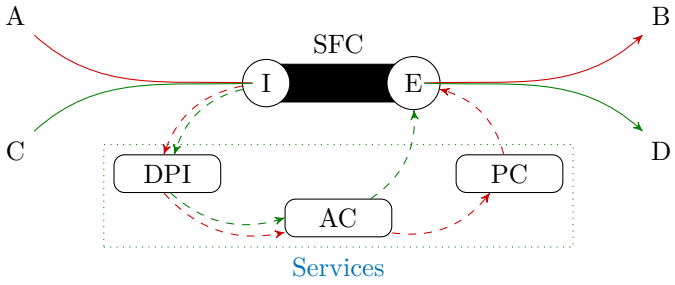


Figure 6: Service Function Chaining with Segment Routing leverage. Hosts A and C that transmit traffic to their respective destinations B and D. Those flows enter a Segment Routing domain via the ingress node I. These nodes choose which Virtual Network Functions of the SR-domain will be applied to them before they exit the SR-domain via egress node E.

1.3 STATE OF THE ART

In this section, we briefly present the state of the art of Service Function Chaining.

StEERING [stEERING] presents an SFC solution built on top of Software Defined Networking (SDN). SDN [SDN] enables to decouple the forwarding plane from the control plane which implies to put the network intelligence in a software-based controller instead of in the network devices. StEERING consists of a set of OpenFlow 1.1 [OpenFlow] switches which are programmed and managed by a central controller. It reuses metadata provided by OpenFlow 1.1 to encode the set of services where one bit is used to characterize the packet direction and the others bits to specify services. The direction indicates whether a packet goes from a subscriber to the core network or from the core network to a subscriber. It uses the control plane managed by the SDN of OpenFlow to support a flexible routing.

Other solutions rely on the Network Service Header (NSH) [IETF SFC-NSH] which defines a new service plane protocol. NSH is typically inserted between the original packet that must be treated by chained services and an outer network transport encapsulation such as MPLS [RFC 7746], VXLAN [RFC 7348] or GRE [Far+13]. NSH does not give a hop-by-hop path instruction in the NSH packet but a simple label which means that states must be held by devices in order to forward the packet to the next hop. Obviously, nodes participating in the SFC need to support it. OpenDaylight [opendaylight] presents an SFC solution built on top of Software Defined Networking (SDN) using NSH.

MPLS can also provide a way to build a Service Function Chain based on NSH [IETF SFC-MPLS]. The chain is represented by an MPLS label stack where each label corresponds to a service function. NSH [IETF

SFC-NSH] is mainly used to add metadata to each packet. This solution is similar to the SR solution because in both cases, the packet contains the path that it must follow. According to [SRA], MPLS suffers from a scalability point of view because each node in the domain has to maintain internal states. In addition, SR can benefit from the usage of ECMP in IP networks where pure MPLS has poor load-balancing characteristics.

Currently, Service Function Chaining is not widely developed but generates interest in the academic and industry. SFC with SR has been mentioned in [SRA], but we did not find any implementation of this solution even though it offers some advantages compared to pure MPLS. For example, SFC with SR takes advantage of ECMP, holds states only at the ingress node and is able to have the optimal path to reach intermediate node.

As we did not find any paper studying SFC with SR, this Master Thesis will explore this solution for the first time.

1.4 CONTRIBUTIONS

This master thesis covers multiple aspects of implementing Service Function Chaining with Segment Routing. We discuss the IPv6 flavour of Segment Routing in Chapter 2, then the global architecture of Service Function Chaining with SR in Chapter 3. This architecture is evaluated in Chapter 5 with different benchmarks to measure the performance of our SFC setup. Multiple services are presented and evaluated with benchmarks in Chapter 6, 7, 8 and 9. This thesis also presents a development environment in Chapter 10 and a new framework developed for SFC SR testing purpose in Chapter 11. Finally, Chapter 12 concludes and presents our contribution to other open-source project.

2 THE IPV6 ROUTING HEADER

IPv6 was developed in the 1990s by the Internet Engineering Task Force (IETF) to deal with the long-anticipated problem of the exhaustion of IPv4 addresses. It was first formally described in the Internet standards document [RFC 2460], published in December 1998. In addition to offering more addresses, IPv6 also implements features not present in IPv4. The IPv6 header format is shown in Figure 7.

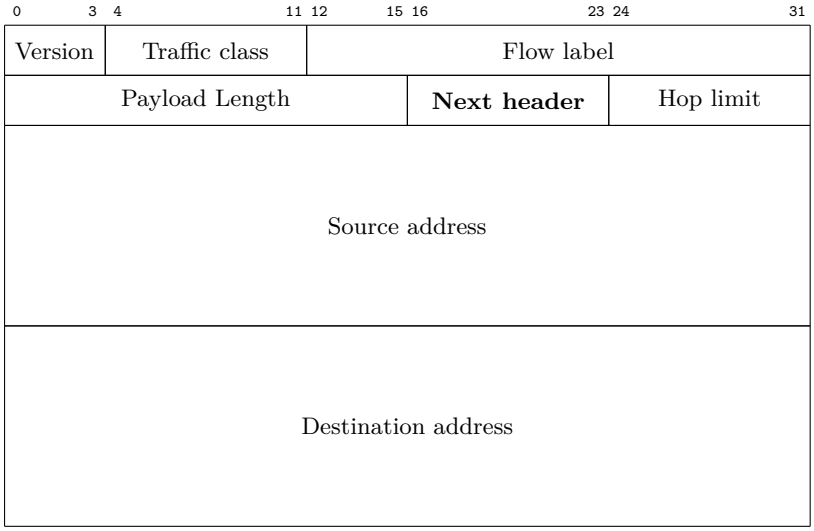


Figure 7: IPv6 Header Format

2.1 THE NEXT HEADER FIELD OF IPV6

One important feature of IPv6 is the *Next Header* field. This 8-bit field indicates the type of header that immediately follows the IPv6 header. It can be a transport layer header (e.g. 6 for TCP or 17 for UDP) or an IPv6 extension[RFC 2460]. These values are defined in the IPv4 Protocol field[RFC 1700]. Figure 8 shows how the *Next Header* field is used in front of an ICMPv6 packet. A non-exhaustive list of protocol numbers can be found in the appendix A.2. Only the *Next Header* field in the IPv6 header is revelant at the moment, descriptions of the other fields can be found in the appendix A.1.

Chaining extension headers in IPv6 allows optional internet-layer information encoded in separate headers to be placed between the IPv6 header and the upper-layer header of the packet. One IPv6 extension header defined by [RFC 2460] is the Routing Header. Figure 9 shows a chain of extension headers in IPv6 with a Routing Header and a TCP

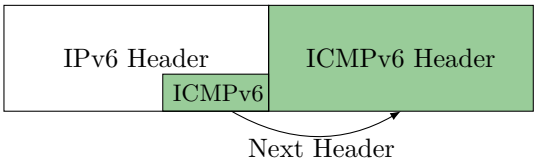


Figure 8: Chaining Extension Headers in IPv6 with an ICMPv6 packet

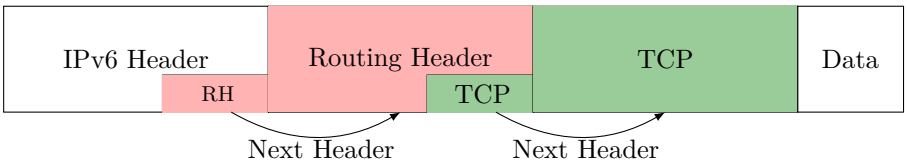


Figure 9: Chaining Extension Headers in IPv6 with a Routing Header and a TCP payload

payload. As we can see, the last header which is in fact the upper-layer does not have a *Next Header* field.

2.2 ROUTING HEADER

The Routing Header shown in Figure 10 is used to steer a packet through one or more intermediate nodes to its final destination. The Routing Header is identified by a *Next Header* value of 43 in the immediately preceding header, and has the following format defined by [RFC 2460]:

NEXT HEADER an 8-bit selector, it identifies the type of header that immediately follows the Routing Header, as explained earlier.

HDR EXT LEN an 8-bit unsigned integer, the length of the Routing Header in 8-octet units, not including the first 8 octets.

ROUTING TYPE an 8-bit identifier of a particular Routing Header variant.

SEGMENTS LEFT an 8-bit unsigned integer, the number of route segments that remain, that is, the number of explicitly listed intermediate nodes still to be visited before the final destination is reached.

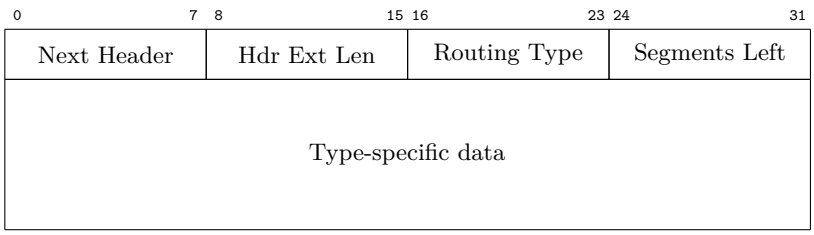


Figure 10: Routing Header Format

TYPE-SPECIFIC DATA a variable-length field of a format determined by the *Routing Type* field, and of such a length that the complete Routing Header is an integer multiple of 8 octets long.

[RFC 2460] also specifies that if a node, while processing a received packet, encounters a Routing Header with an unrecognized *Routing Type* value, the behavior of the node depends on the value of the *Segments Left* field, as follows:

- If *Segments Left* is zero, the node must ignore the Routing Header and process the *Next Header* in the packet, whose type is identified by the *Next Header* field in the Routing Header.
- If *Segments Left* is non-zero, the node must discard the packet and send an ICMP Parameter Problem, Code 0, message to the packet's *Source Address*, pointing to the unrecognized *Routing Type*.

This specification is useful in order to deploy incrementally a new *Routing Type* value. Indeed, end-hosts do not need to know the Routing Header type if they are the final destination. Plus, there is the possibility to have a network that recognizes this new *Routing Type* while other networks are absolutely agnostic to it.

2.2.1 *Routing Header, different routing types*

- Type 0 is defined by [RFC 2460] and is discussed in section 2.3.
- Type 1 is defined by Nimrod, an old project funded by DARPA. This type is unused and will not be described.
- Type 2 is used by MIPv6 (Mobile Internet Protocol version 6) defined by [RFC 6275]. This is understood only by MIPv6-compliant stacks.
- Type 4 is Segment Routing, defined in draft [IETF SR 08] and is discussed in chapter 2.5.

2.3 TYPE 0 ROUTING HEADER (RH0)

[RFC 2460] defines a way to achieve loose source routing by defining the type 0 Routing Header, also denoted RH0 in this thesis. This header extension is displayed on Figure 11 and contains six fields:

NEXT HEADER the same as described in section 2.2.

HDR EXT LEN an 8-bit unsigned integer, the length of the Routing Header in 8-octet units, not including the first 8 octets. For the SRH, *Hdr Ext Len* is equal to two times the number of addresses in this header.

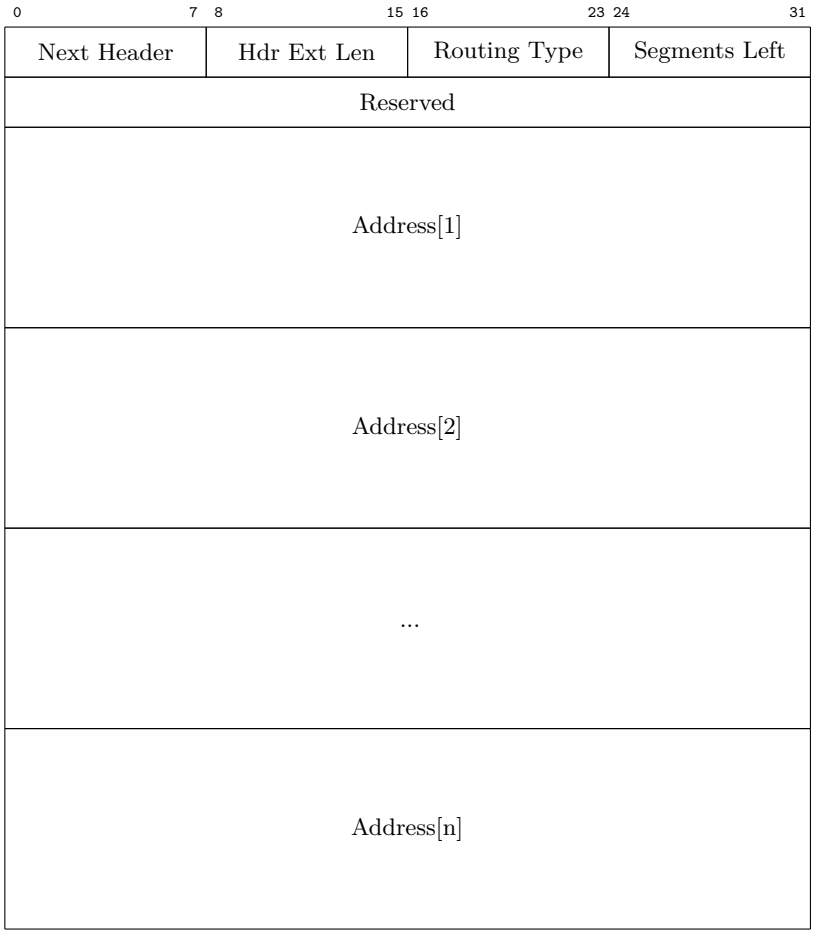


Figure 11: Type 0 Routing Header Format

ROUTING TYPE a constant, 0.

SEGMENTS LEFT the same as described in section 2.2.

RESERVED a 32-bit reserved field, initialized to zero for transmission; ignored on reception.

ADDRESS[1..n] a vector of 128-bit addresses, numbered from 1 to n.

Any good implementations of RH0 must perform safety checks on the fields. For example, you have to ascertain that the *Hdr Ext Len* field is not odd, that *Segments Left* field is not greater than the number of intermediate nodes, etc.

2.3.1 *Processing*

When the packet is crafted at the host, the *Destination Address* field of the IPv6 header is the first intermediate node, and so the first address in the vector list is the second intermediate node, and so on. The last address in the vector list is the final destination address. During the lifetime of the packet from one device to another, the vector list of addresses of the RH0 and the IPv6 *Destination Address* will be modified according to the algorithm described previously.

When the packet reaches the node identified in the Destination Address field of the IPv6 header, the node checks whether the packet is at its final destination or not by controlling the value in the *Segments Left* field. If the value of *Segments Left* is at 0, then the packet can be processed, otherwise the value of *Segments Left* is decremented by one. Then, the value of the IPv6 *Destination Address* and address at the offset $\frac{\text{"Hdr Ext Len"}}{2}$ − "Segments Left" are swapped.

2.3.2 *Checksum*

It is relevant to note that if the originating node has to compute a checksum with a pseudo header calculation that contains the destination address, the checksum is computed with the final destination address and not the first intermediate node. Thus, the checksum is incorrect during transit.

2.4 DEPRECATION OF RH0

Further work with RH0, including an entertaining demonstration with scapy [BIONDI, 2007], revealed severe security problems. For this reason, source routing with the type 0 Routing Header has been removed from the IPv6 specification through [RFC 5095] published in December 2007.

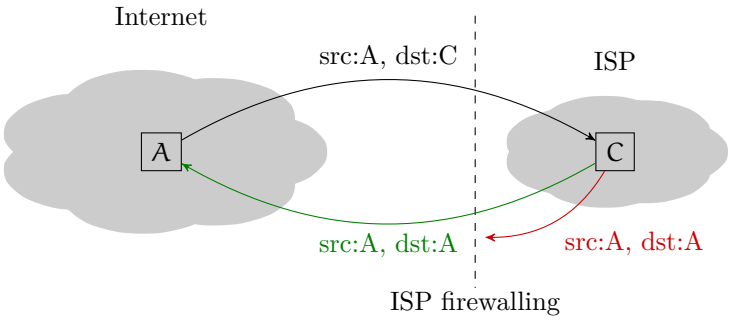


Figure 12: RH0 attack: checking if an ISP filters spoofed traffic from its clients

2.4.1 *Security implications*

To show the risks that RH0 poses to the security of a network, this section presents two different examples of attacks, examples taken from the document "IPv6 Routing Header Security" [BIONDI, 2007].

2.4.1.1 *Checking if an ISP filters spoofed traffic from its clients*

The idea behind the attack is to use a client that supports RH0 to send a boomerang packet. When the packet arrives at the client, the RH0 will set the IPv6 destination to the next segment, which is going to be the source of the packet. The packet will then go back to the source before returning to the host for processing. If the packet is effectively processed, that means that the ISP might not filter spoofed traffic from its clients. This attack can lead to more elaborate attacks against hosts.

Figure 12 shows an attacker A sending a packet to client C. This packet has an RH0 header with the address A. When the client processes the packet, it changes the destination address to A. The green arrow represents a successful attack in which the packet passes through the ISP Firewall. The red arrow renders an unsuccessful attack in which the ISP filters the spoofed traffic from its clients.

This attack can easily be done with a simple ICMP Echo Request as implemented in Listing 3 in Appendix C.1 in Python with Scapy[Scapy].

2.4.1.2 *Denial Of Service Amplification*

The idea behind this attack is to create a packet that will remain in the network between two hosts. This can be done simply by crafting a packet with a type 0 Routing Header that has a list of intermediate nodes which will contain the hosts that we want to target. Figure 13 shows how this attack can impact the bandwidth of a link and therefore the targeted host.

Again, this attack can easily be done with an ICMP Echo Request. The Scapy program shown in Listing 4 in Appendix C.1 is sufficient to

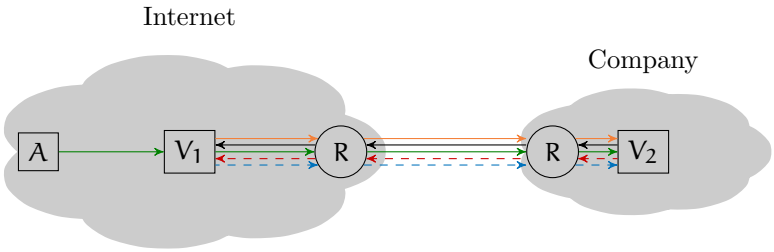


Figure 13: RH0 attack: Denial Of Service Amplification. The attacker sends a packet with an RH0. The aim is to saturate the link between two hosts. If a company has only one link to the internet, as shown on the figure, this can affect how easy it is to reach the company.

carry out this attack where the links between `addr1` and `addr2` will suffer from this attack. Also note that this attack also impacts hosts that have to compute a new RH0 each time the packet is received[BIONDI, 2007].

2.4.2 *Notes*

The Routing Header is a new feature of IPv6. However, [RFC 0791] defines two IPv4 options called "Strict Source Routing" and "Loose Source Routing". The first option provides a means for the source of an internet datagram to supply routing information to be used by gateways to forward the datagram to the destination, and to record the route information[RFC 791]. "Loose Source Routing" is similar to RH0, the packet has a list of intermediate nodes that must be visited. These options have the same security issues as the RH0 mentioned before. These have been deprecated in an IETF document called "Deprecation of Source Routing Options in IPv4"[IETF-Reitzel, 2015] and then in the [RFC 6274]. On this point, the designers of IPv6 did not learn from the weakness of IPv4.

2.5 SEGMENT ROUTING HEADER

As explained before, Segment Routing (SR) can be deployed over IPv6 with the addition of a new type of Routing Extension Header.

This new Routing Extension Header is the type 4 Routing Header and is being defined and developed within the SPRING working group of IETF[IETF SR 01]. The number 4 is suggested by the IETF draft but has still to be determined by IANA. This new Routing Header fixes the problems encountered with RH0. This chapter is dedicated to this Routing Extension Header since we shall use it to perform Services Function Chaining.

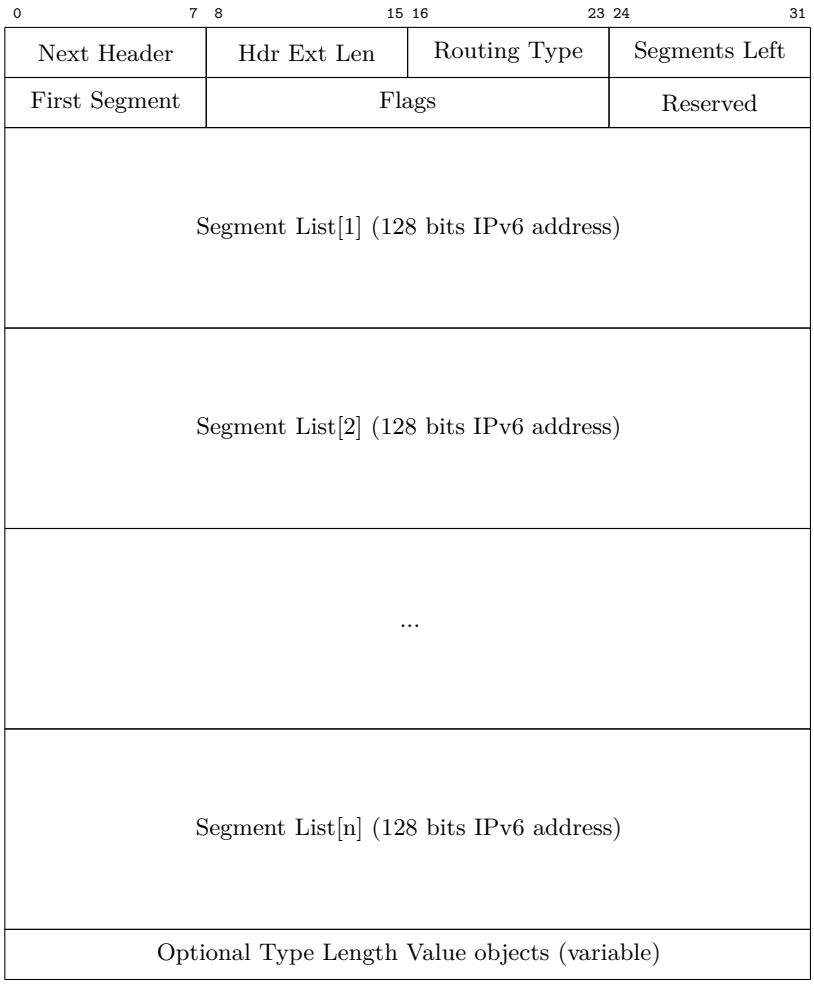


Figure 14: Segment Routing Header

2.5.1 *Header Format*

At first glance, Segment Routing Header (SRH) shown at figure 14 is similar to the type 0 Routing Header (RH0). However, there are differences.

NEXT HEADER an 8-bit selector that identifies the type of header immediately following the SRH.

HDR EXT LEN an 8-bit unsigned integer, the length of the SRH in 8-octet units, not including the first 8 octets.

ROUTING TYPE identifier to be determined by IANA, but every known Segment Routing implementation uses the value 4 as suggested in the IETF draft.

SEGMENTS LEFT defined by [RFC 2460], and described in section 2.2.

FIRST SEGMENT contains the index in the Segment List of the first segment of the path, which is in fact the last element of the Segment List.

FLAGS 16 bits of flags as described in the appendix A.3.3

RESERVED which should be unset on transmission and must be ignored on receipt.

SEGMENT LIST[0..n]128 bit IPv6 addresses representing the nth segment in the Segment List. The Segment List is encoded starting from the last segment of the path. I.e., the first element of the segment list (Segment List [0]) contains the last segment of the path while the last segment of the Segment List (Segment List[n]) contains the first segment of the path. The index contained in *Segments Left* identifies the current active segment.

TYPE LENGTH VALUE (TLV) an optional value that can be added to the segment routing header.

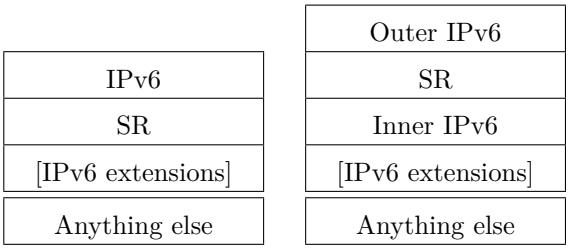
From the protocol's point of view, the order of IPv6 addresses in the Segment Routing Header (SRH) is reversed compared to the RH0. The first segment in SRH is in fact the final destination of the packet. This change allows to know the current segment without computation since the *Segments Left* field can be used as an index to the segment list. Also, the list in SRH is never modified, only the destination address field of the IPv6 is modified during the life of the packet. This reduces the cost of the operations needed for the router to be able to forward SR packets in IPv6.

2.5.2 *Encapsulation and inline mode*

There are two ways to use the Segment Routing Header in IPv6, as shown in Figure 15: **inline** mode and **encap** (encapsulation) mode . We shall work only with *encap* mode since the **inline** mode is allowed only for end hosts. Indeed, intermediate routers cannot add extension headers without encapsulation.

INLINE MODE The **inline** mode method shown in Subfigure 15a allows an end-host to define segments through which it wants packets to go. Like RH0, the SRH does not define a hop-by-hop path for the packet to follow, the control plane will be responsible for the path followed by the packet between segments.

ENCAP MODE The **encap** (encapsulation) mode method shown in Subfigure 15b allows a router to encapsulate a packet by adding a new (outer) IPv6 header followed by an SRH. This technique allows the incoming packets to be source-routed inside a Segment Routing



(a) Inline mode (b) Encap mode

Figure 15: Two ways to use the segment routing header in IPv6: encap mode and inline mode.

domain (SR-domain) as operators can enforce a specific path through segments. In this mode, there is an ingress node that will encapsulate packets and an egress node which will decapsulate them.

Note that in **inline** mode, we could do the same as in encap mode by adding and removing the SRH from the packet. But this is not a recommended practice. In such a situation, we should want to use the **encap** mode as we do not want to directly modify the packet.

2.5.3 *Security*

As explained in section 2.4.1, being able to modify the path of a packet with a routing header can create security issues. To address this problem, an SR-IPv6 packet must never be forwarded on a public network if the packet is not authenticated by an HMAC[IETF SR 08]. Authentication ensures that packets do not come from an unauthorized source. This prevents an attacker from using Segment Routing as a vector for attacks (see section 2.4.1).

2.5.3.1 *HMAC*

To authenticate an SR packet, the H-flag must be set and the HMAC must be encoded in the SRH as an optional TLV (Type-Length-Value) field. This HMAC TLV option shown in figure 16 contains the HMAC Key ID and the HMAC. If the packet is not signed, the packet must be discarded. For performance reasons, the HMAC field must only be checked and validated by the edge routers of the SR-domain, which are named *validating SR routers*.

The HMAC field allows an SR packet to move from one SR-domain to another. If a packet never goes outside its SR-domain and uses the **encap** mode, there is no need to authenticate it. An ingress node will encapsulate the packet and the last node that will process the packet will decapsulate it. However, we need to be sure that the ingress routers of the SR-domain drop every packet that already contains an SRH.

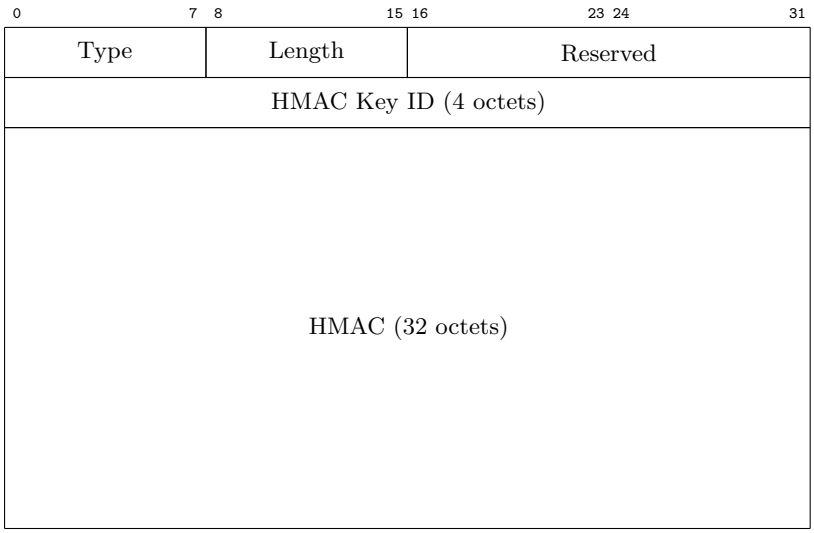


Figure 16: Segment Routing Header: HMAC TLV

2.5.4 *Segment Routing: specialized nodes*

In an SR-domain, we can distinguish multiple kinds of node and multiple kinds of segment: (1) ingress nodes, which encapsulate IPv6 packets, (2) egress nodes, which decapsulate IPv6 packets, and (3) service nodes, which apply a service function to IPv6 packets.

Figure 17 shows how packets enter an SR-domain to be forwarded to a specific service function. A is the source, B is the destination. I and E are respectively the ingress and the egress nodes of the SR-domain, S being the service nodes. Note that multiple ingress and egress nodes can exist inside an SR-domain. The following points, reported in figure 17, explain the different steps of processing for encapsulated packets:

1. An IPv6 packet issued by A arrives at an ingress node of an SR-domain.

2. The ingress node (I) encapsulates the IPv6 packet from A inside an IPv6 and an SRH. The *Segments Left* field of SRH is equal to one, referring to segment S.

3. The packet passes through the service node S. The *Segments Left* field is decremented by S and now equals 0. The destination address of the IPv6 header is changed by the new current segment (E).

4. The packet arrives at the egress node. The outer IPv6 and its SRH are removed.

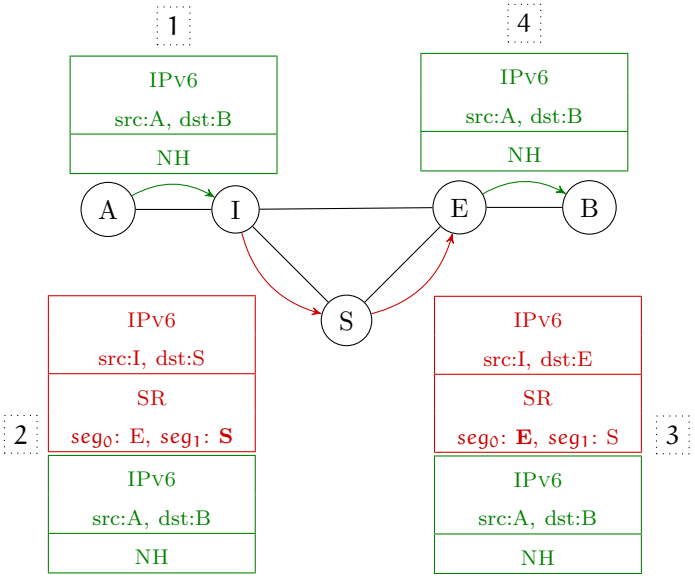


Figure 17: Example of the life cycle of a packet inside an SR-domain with services. First, the packet enters the SR-domain via an ingress node, then it can goes through some services and finally exits the domain via an egress node.

2.5.5 *Segment Routing Service*

The aim of this master thesis is to demonstrate that service function chaining with Segment routing is an interesting feature that gives operators new services in the network with ease. Our services are described in Chapter 3.

2.6 CONCLUSION

IPv6 provides an important feature which is the *Next Header* mechanism. This modular field allows new IPv6 Extension Headers, as Segment Routing, to be implemented with ease. Segment Routing provides an easy way to steer packets through intermediate nodes before reaching its final destination.

In the next chapter, the Service Function Chaining with Segment Routing will be presented.

Part II

SERVICES

3 SERVICES

As explained before, Service Function Chaining (SFC) is the capability to create a chain of connected network services. These service functions can be virtualized in order to create a Network Function Virtualization (NFV). Segment Routing (SR) is a technology that can be use to achieve this purpose.

3.1 ARCHITECTURE

In this chapter, we explain the global architecture of our services and how we use them. The figure 18 shows how a service (e.g. COUNTER described later in section 3.3) can be applied on a traffic from A to B. First, the traffic enters an Segment Routing domain (SR-domain), then goes through our service and finally exits the SR-domain and is forwarded to its original destination. In our setup, we have three important nodes:

THE INGRESS NODE OF THE SR-DOMAIN (I) Practically, this node performs an IPv6 encapsulation of the packet. The outer IPv6 is followed by a Segment Routing Header (SRH) which is used to define the service function chain through which the traffic must pass.

In other words, ingress nodes perform some packets classification as they decide for a packet which service function chain it will used.

THE EGRESS NODE OF THE SR-DOMAIN (E) is reached once the service function chain has been passed through and decapsulates the original packet, forwarding it to its original destination.

THE SERVICE NODES OF THE SR-DOMAIN (S) performs service functions on the traffic. In our case, we have only one service, our counter server.

The service function chaining architecture shown at figure 18 uses an IPv6-encapsulation technique to chain all service. The end-host is agnostic to the service function chain and does not need to support SR.

3.2 ASYNCHRONOUS OR SYNCHRONOUS SERVICES

As seen in chapter 4, when the kernel receives packets, it delivers them to the service function if appropriate. We can handle incoming pack-

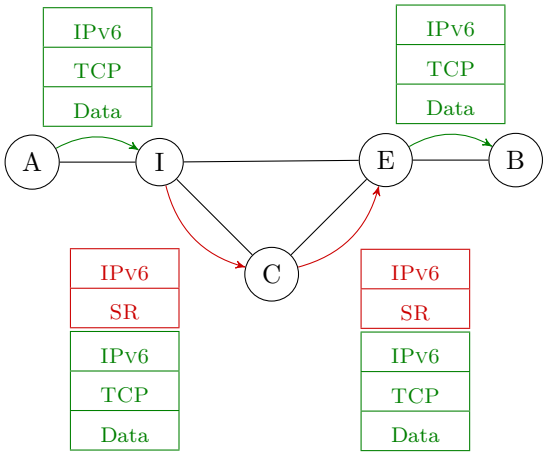
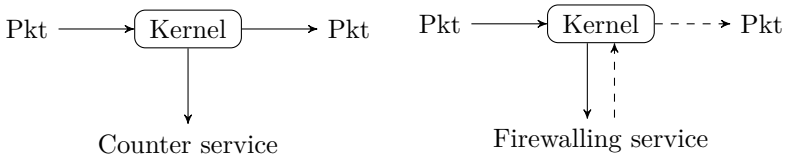


Figure 18: Traffic flows from a source A to a destination B where a service is applied



(a) Asynchronous service (b) Synchronous service

Figure 19: Comparison of synchronous and asynchronous service

ets asynchronously or synchronously. Figure 19 shows the difference between these two modes :

- **Asynchronous** service is not required to forward received packets. This is the case of our COUNTER service. Figure 19a details how it works: when a packet arrives, the kernel delivers it to the service and also forwards it in the network. Note that asynchronous service can also send new packets.
- **Synchronous** service will receive packets and will be responsible to send them back to the kernel if appropriate. In this case, the service can modify packets and send them whenever it wants. This kind of service permits to build firewalling service as represented in the figure 19b. For example, it can block all HTTP traffic based on port 80.

3.3 BASIC SERVICES

Before building complex services, we have developed two basic services: COUNTER and MEMLINK_TEST. The first one is asynchronous and counts the number of packets and the total bytes passed through the service. The second one is synchronous and its only purpose is to do

nothing with the packets except forwarding them. Those services are useful to benchmark performance of the network as they are simple. Obviously, there is no real use case for these services as the first one performs a really limited part of a COUNTING service and the second one has no effect at all.

3.4 CONCLUSION

The architecture for Service Function Chaining based on Segment Routing is quite common: some ingress and egress nodes respectively encapsulate and decapsulate traffic with SR, which acts as a service control plane. There are two main families of services: asynchrone and synchrone. The first one receives packets that have already been forwarded to the next destination while the second one is responsible for transmitting it if needed.

In the next chapter, we explain how packets are transmitted to our service function from the network interface. This solution is first presented for a service with only one thread, before we expose the multithreading solution.

4 FROM KERNEL SPACE

In UNIX systems, the kernel is responsible for receiving and sending packets through the right interface depending on its forwarding table. If the destination is recognised as itself, then the kernel is also responsible for delivering incoming packets to the corresponding user-space application. With Segment Routing (SR), we want to intercept traffic which is not related to an application running on the system, so we cannot use the standard kernel mechanism to transfer traffic between the kernel and our service (such as a TCP or UDP socket, for example).

In our thesis, we use a modified kernel that implements SR with IPv6[1]. Also, we use the SEG6CTL library [Seg6ctl] which implements a memory-mapped NETLINK-based [NETLINK] data transfer mechanism. NETLINK is a socket-based mechanism typically used by the kernel for user communications.

Since our services rely on this library to make the link with the kernel, this chapter explains how SEG6CTL works.

4.1 SEG6CTL

By default, NETLINK uses a message-based system through the `sendmsg()` and `recvmsg()` system calls. However, this method is not suited for high throughputs since the data must be copied from the packet to the kernel message, then to the user space message. In addition, the method needs one system call per message (packet). This leads to many memory copies, which are slow, and many context switches due to the frequent system calls.

This is why SEG6CTL does not use this mechanism but instead uses a frame-based memory mapped approach that enables the kernel and the application to share a common buffer, so that only one memory copy is necessary and fewer context switches are required.

The main goal of the SEG6CTL library is to build a bridge between the kernel and our launched service for traffic intended to it. To achieve this, the kernel must copy the received packet into the frame-based memory mapped buffer to enable it to be processed by a service. Obviously, some synchronization must be performed between the kernel and the service. The kernel needs to know when there is a new service bound to an IPv6 address. When a service binds itself to an IPv6 segment, it allocates memory shared with the kernel. This shared memory is called

1 http://www.segment-routing.org/

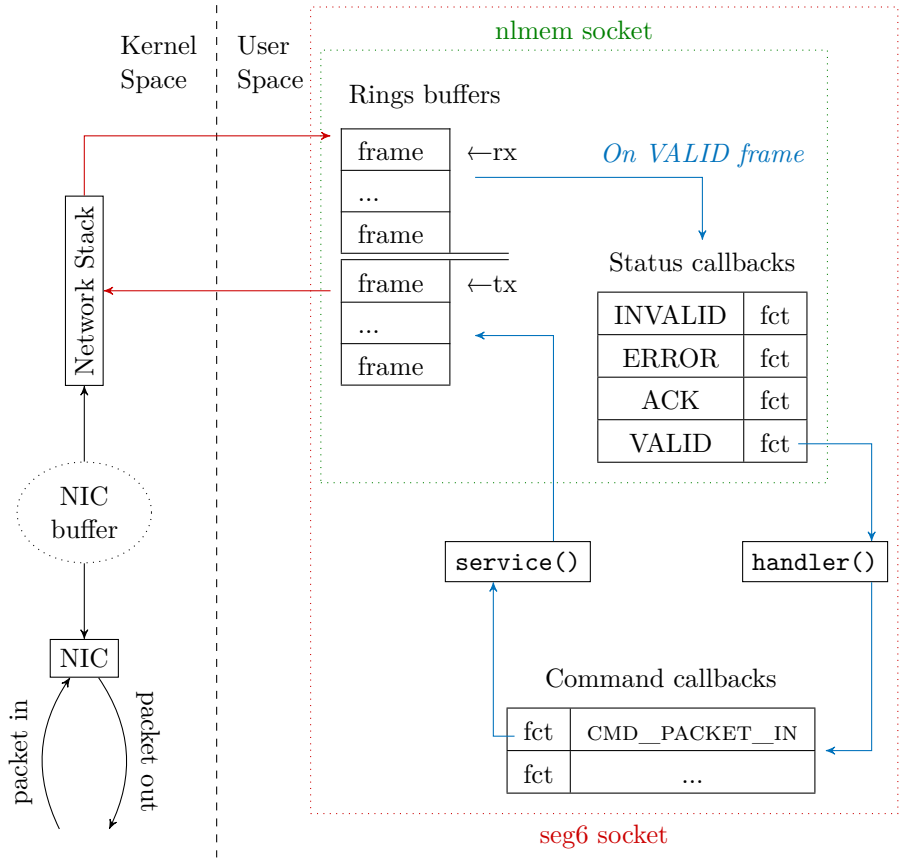


Figure 20: SEG6CTL: the glue between kernel and user space. The setup is divided between the kernel and the user space. The kernel space is responsible for transferring incoming packets from a NIC to an rx ring buffer and of transferring outgoing packets from the tx ring buffer to a NIC. This figure shows the interactions between the nlmem socket and the seg6 socket.

Legend
: Kernel interaction with tx and rx rings buffers
: Basic kernel interactions
: Seg6ctl interactions

FRAME STATUS	DESCRIPTION
NL_MMAP_STATUS_VALID	frame contains a pending message
NL_MMAP_STATUS_UNUSED	frame is currently unused
NL_MMAP_STATUS_SKIP	frame cannot be overwritten and must be skipped
NL_MMAP_STATUS_COPY	frame must be copied from NETLINK for some reasons outside the scope of this thesis.

Table 1: Frame status

a ring buffer and kernel uses it to write packets and to read packets that have been treated by the service.

Communication between kernel and user space is conducted through the ring buffer which is split into multiple frames. Each frame contains a message that has a status (*Message's Status*) and a related callback command (*Message's Command*). Also, each frame itself has a status (*Frame's Status*) which are listed in Table 1. The status of a frame is mainly used to determine if there is a message or if the slot is free. Typically, when a packet is coming, the kernel seeks an unused frame where the message can be written into.

4.1.1 *Structures*

The SEG6CTL library defines two structures : the *nlmem socket* and the *seg6 socket*. The *nlmem socket* is a generic way to handle incoming and outgoing packets. This method is used to transit information between the kernel and the user space. *seg6 socket* is an abstraction where we define functions associated with specific message commands added by the Segment Routing Linux implementation.

NLMEM SOCKET is composed of two ring buffers and a table of callbacks, represented in figure 20.

The rx ring buffer transfers data from the kernel to the service and the tx one from the service to the kernel. As explained before, each ring is split into multiple frames each containing a message depending on its *Frame's Status*. Frames have the same size, defined when their ring buffer was created. The total size of a ring buffer will determine how many frames it can hold.

There are four callbacks listed in the table 2, each one specifies a behaviour to have depending on the *Message's Status*. So, when a message is written on the rx ring buffer by the kernel, the callback related to the *Message's Status* is triggered if and only if the *Frame's Status* is VALID.

MSG STATUS CALLBACKS	DESCRIPTION
INVALID	Malformed packet. For example, packet too short.
ERR	Error returned by kernel
ACK	Acknowledgement from kernel with respect to a previous message
VALID	A valid message that can be treat

Table 2: Table of callbacks for nlmem socket

MSG COMMAND CALLBACK	DESCRIPTION
SEG6_CMD_PACKET_IN	Message contains an incoming packet
SEG6_CMD_PACKET_OUT	Message contains an outgoing packet

Table 3: Table of callbacks for seg6 socket

SEG6 SOCKET contains an nlmem socket as described above, and a set of callbacks. A seg6 socket extends the NETLINK protocol and defines its own NETLINK packet's commands types. These callbacks are shown at table 3 and have a defined behaviour for each of *Message's Command*. For example, we can setup the service called when a packet is coming with the SEG6_CMD_PACKET_IN type.

4.1.2 *Packet flow*

The figure 20 shows interactions between ring buffers and the interfaces of the host. Through these interactions, packets are delivered to and from the interfaces to the service. In this subsection, we explain how the kernel space and the user space interacts with the ring buffers.

4.1.2.1 *Kernel space*

To keep it simple, when a packet is received from an interface, the associated Network Interface Controller (NIC) transfers it into the NIC buffers and an interrupt is raised and handled by the kernel networking layer. The treatment of a packet will be decided according to its IPv6 header.

If the destination is not local, the packet will be forwarded or discarded depending on the system settings.

If the destination address is local, the Network Stack (NS) should send the packet to the corresponding application of the user space only if it reaches its final destination. However, with Segment Routing, being the destination address does not mean that the packet is at its final

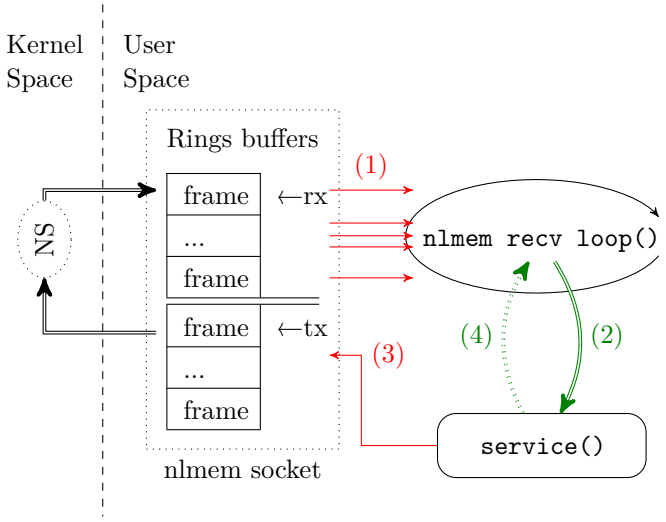


Figure 21: Interaction between the two ring buffers and the SEG6CTL library. There is a `nlmem_recv_loop()` used to launch a service for a packet written by the kernel inside the rx ring buffer. The service has direct access to the tx ring buffer.

destination. To determine that, in first place, the kernel inspects all the next header of incoming packets. If the kernel finds a Next Header corresponding to Segment Routing, the *Segment Left* field will be decremented and the current destination address of the IPv6 header will be changed with the new IPv6 address found in the *Segment List* at the index of the *Segment Left*.

So, if the destination address is an intermediate destination of the IPv6 packet with SR, it is sent to service in user space bound at this address if there is by using SEG6CTL.

4.1.2.2 *User space*

Figure 20 shows the global setup and how a packet passes through the two structures: *nlmem socket* and *seg6 socket*. In SEG6CTL, there is a loop function called *nlmem_recv_loop()* that is responsible for retrieving packets from the rx ring buffer and for calling our service. Figure 22 describes its decision process and a simplification of the four phases of its work is shown at Figure 21:

1. The function loops on the rx ring buffer to find a frame with a VALID status that contains a message that should be processed.

2. The function launches the corresponding service with regard to the decision process. The service function is called if and only if the message status is also VALID and the message command is SEG6_CMD_PACKET_IN.

3. While the service function handles the packet, the function is able to directly write packets into the tx ring buffer.

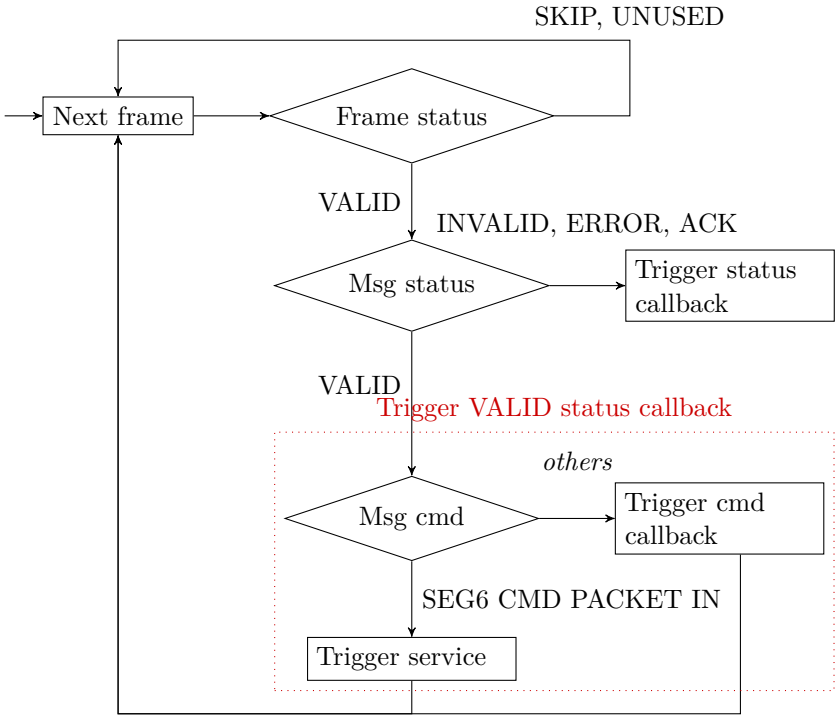


Figure 22: Decision process of `nlmem_recv_loop()`

4. The loop continues with the next VALID frame.

4.1.3 *Frame release*

Obviously, one goal is to operate as efficiently as possible. To increase efficiency, unnecessary memory copies must be avoided, because these are time-consuming. To minimize memory copies, the service receives a pointer to the packet which is stored inside the rx ring buffer. This packet will be available until the service finishes. The `nlmem_recv_loop()` function will release the frame by setting the *Frame's Status* to UNUSED.

Figure 23 shows the state diagram of a frame inside the rx ring buffer, and the transition of the *Frame's Status*. The frame must have status VALID before and during the service. Since `nlmem_recv_loop()` launches the service as a function, the frame can be released as soon as the function is complete because the service is done.

4.2 MULTI-THREADED SEG6CTL

Initially, the SEG6CTL library was implemented without any support for multi-threading. Figure 24 provides a schematic overview of that version. This design is not the most efficient one since it can process only one packet at a time. However, this system has the benefit of

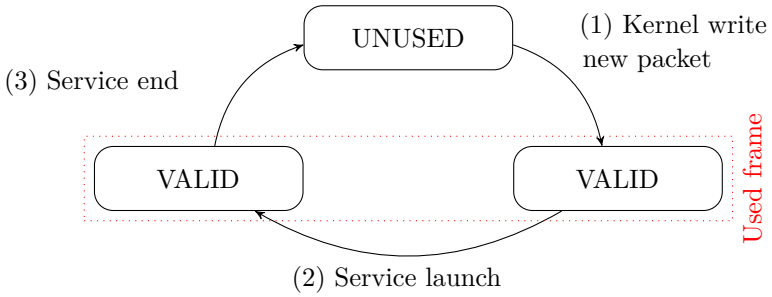


Figure 23: State diagram for a frame inside the read part of the ring buffer.

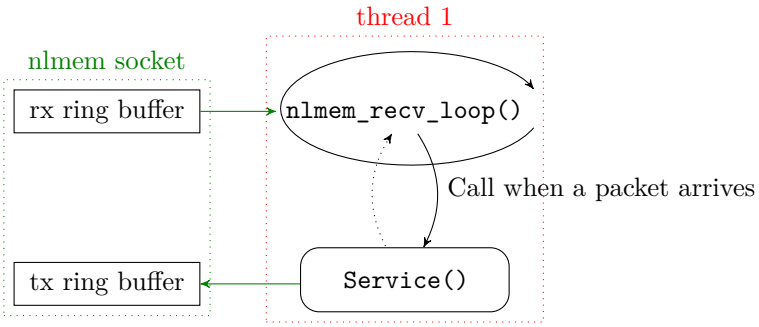


Figure 24: Single-threaded setup

being sequential, so frames can be released when the call to the service is done.

To be more efficient, the SEG6CTL framework should allow multiple packets to be processed in parallel. We choose to use a producer/consumer model to distribute the workload across multiple threads. This setup is shown in Figure 25. We have an asynchronous communication between one producer running the `nlmem_recv_loop()` to fill a queue and `n` consumer(s) that consume(s) it.

NOTE if the service is stateful, accesses to the shared states must be thread-safe.

4.2.1 *Frame release*

When we designed our multi-threaded architecture, we faced an issue with the release of the frames. Indeed, releasing the frame inside the `nlmem_recv_loop()` is no more suitable. If the frame is released when the producer finished its job, then the frame cannot be safely retrieved by a consumer. Indeed, there is no guarantee that the kernel has not overwritten the frame with a new incoming packet.

Building this solution without unnecessary `memcpy()` implies releasing the frame only when the service function has been executed. To achieve this goal, we added a new boolean variable called **delayed_release** inside the nlmem socket. It decides if `nlmem_recv_loop()` can release

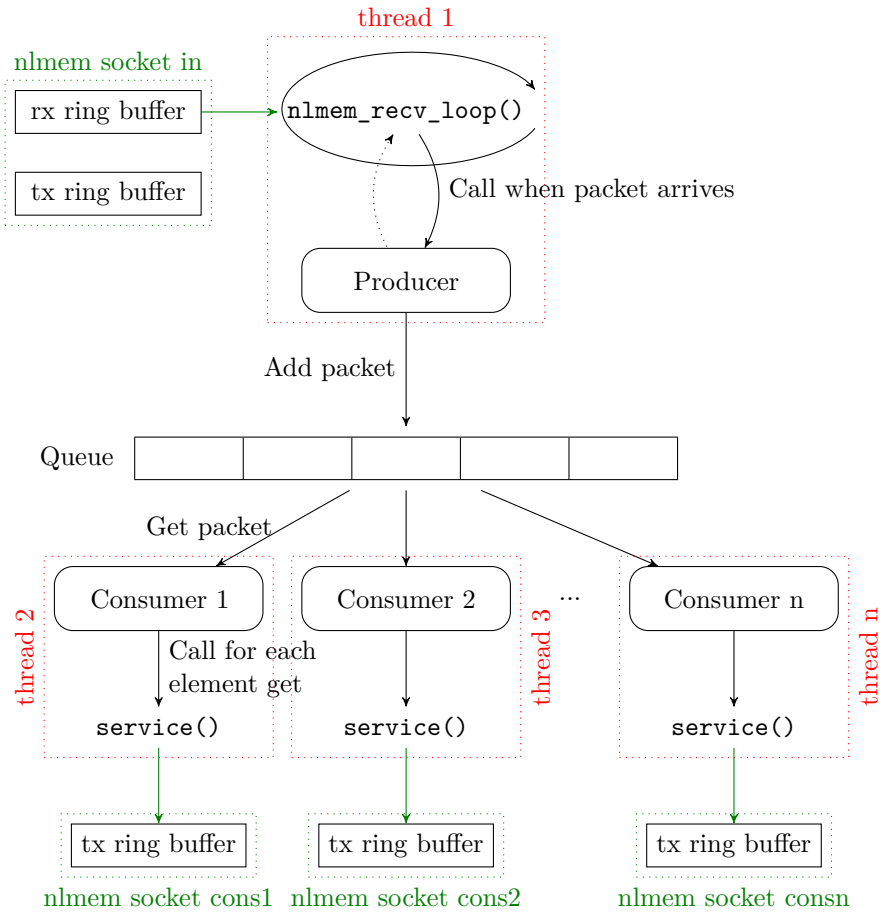


Figure 25: Multi-threaded setup

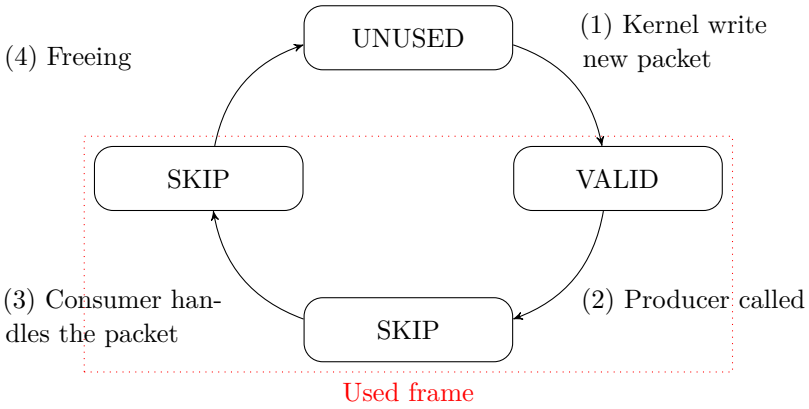


Figure 26: State diagram for a frame inside the rx ring buffer buffer.

the frame directly after the *Message's Status* callback. Listing 1 describes the behaviour of our new implementation with respect to the *Frame's Status*. If **delayed_release** is set, the consumer is then responsible for the release of the frame.

Algorithm 1 Behavior of nlmem_recv_loop()

```
1: if delayed_released then
2:     frame_status ← SKIP
3: call VALID status callback of nlmem
4: if not delayed_released then
5:     frame_status ← UNUSED
```

The new state diagram for a frame is shown in figure 26. In this diagram, we added a new status *SKIP* which represents the moment when the packet has been written into the rx ring buffer, read by the `nlmem_recv_loop()` and added by the producer. A consumer will later process the frame and release it.

4.2.2 *Sockets*

Also, note that each consumer has its own tx ring buffer in order to write outgoing packets. We have also tried to use a single tx ring buffer for all consumers. Unfortunately, the concurrency management to keep the *nlmem socket* thread safe is too costly and decreases the performance.

4.2.3 *Synchronized queue*

When playing with multiple threads, we need to synchronize their executions. The critical zone of our system is the queue distributing the workload across the threads. To implement a synchronized queue, we found two solutions:

1. The first one requires locks (mutex) around the critical area of code and semaphores. We used the well-known POSIX threads library[2].

2. The second is a *Lock Free Queue* (LFQ), this approach is less common; It uses atomic operations, memory barriers and cache coherency guarantees to implement a synchronized queue without relying on the kernel primitives, thus avoiding any context switches. This is suited for very high throughput. [3]

Performance tests are made in Chapter 5.

4.3 CONCLUSION

This chapter presented SEG6CTL library which implements a mechanism based on NETLINK to transfer data through ring buffers from the kernel to the user space. Initially, the library allowed only one thread for the service. We improved it to support multi-threaded services.

In the next chapter, we explain the global environment setup we used to perform our benchmarks. Then we present those for our two basic services: COUNT and MEMLINK_TEST services. Those will be used as references for the benchmarking of more advanced services.

2 http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html
3 http://github.com/target0/liblfq

5 BENCHMARKING

In this chapter, we describe the setup we have used in order to perform our benchmarks. Then, we inspect the performances of the COUNTER and MEMLINK_TEST services. Finally, we study variations in performance under varying running conditions of MEMLINK_TEST service.

5.1 ENVIRONMENT SETUP

Our environment setup, named "inlab", is composed of three servers H4, H5 and H6 with at least 8Go of RAM and Intel(R) Xeon(R) CPU with 4 cores. Those computers are all equipped with 10 Gbits Intel Ethernet Network Interface Card (NIC). Figure 27 depicts how our servers are linked between them and IPs assigned to them.

H5 is connected to H4 via its interface enp1s0f1 with the IPv6 `fc00::5`. H4 is also connected to H6 via its interface enp1s0f0 via the IPv6 `enp1s0f0`. H4 has two IPv6s on its interface enp1s0f1 :`fc00::4` and `fc00::44`. H6 has one IPv6 `fc01::6` on its interface enp1s0f0. The link H4-H5 has a MTU (Maximum Transfer Unit) of 9000 and the link H5-6 has a MTU of 1500. The MTU route from H6 to the destination address `fc00::44` (second IP of H4) is set to 1400. This allows the kernel to append an SR header of maximum 100 bytes.

H6 is the ingress node of our setup. It encapsulates outgoing packets to `fc00::44` by prepending an SR header containing H5 and H4 (`fc01::5,fc00::4`). H5 is used to run our services. Thus, H4 is the egress node in our setup.

We used `iperf3` to perform our performance tests. H4 runs `iperf3` in server mode [1]. H6 runs `iperf3` in client mode and connects to H4 via the IPv6 address `fc00::44`. H4 sends an XML file of 4.6Go stored in his Random Access Memory (RAM). The benchmark finishes once the file transfer is complete or after 60 seconds [2]. We then take the average throughput of the run.

We have to setup some network settings in order to increase the memory allocated to the NICs. We disable the Generic Receive Offload (GRO). GRO allows the NIC to combine received packets into a single large packet (up to 64k), so the Network Stack (NS) will handle fewer larger packets. If we do not disable GRO, the size of received packets is not correct and we cannot apply any service to them as we want to

1 iperf3 -s -B fc00::44
2 iperf3 -B fc01::6 -c fc00::44 -t 60 -F /mnt/ramfs/file.xml

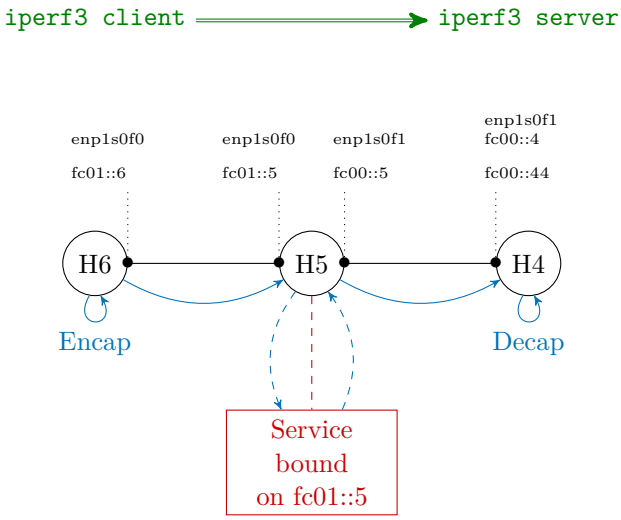


Figure 27: Our environment setup, named "inlab"

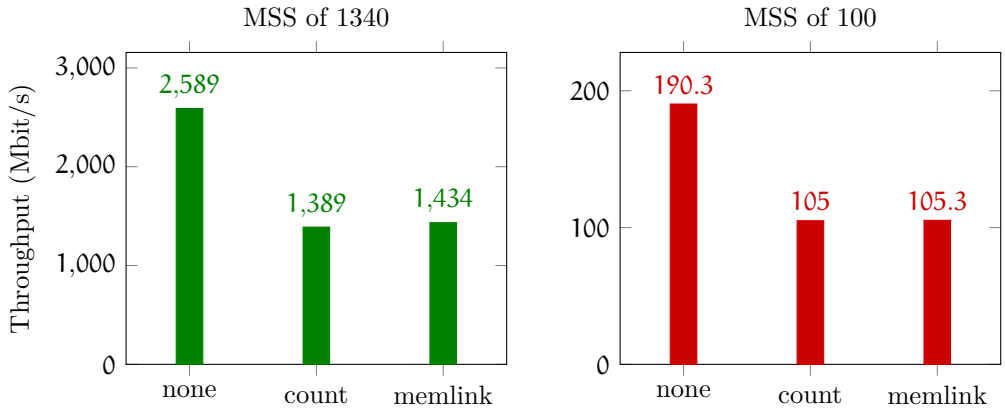


Figure 28: Performance with no service, COUNTER service and MEMLINK_TEST service

send back those packets in the network later. The Large Receive Offload (LRO) and Generic Segmentation Offload (GSO) are disabled, for the same reasons. We also disable NIC checksumming as the destination address changes within the transit with Segment Routing. Listing 2 in Appendix B.2 shows how those parameters have been changed on H5, a similar configuration is applied to H4 and H6.

5.2 SERVICE IMPACT

First, we need to set up a point of reference against which we can compare our measurements. For doing that, we have run our benchmarking setup without any service. Then, with our COUNTER and MEMLINK_TEST services.

MSS (BYTES)	MEAN (MBITS/S)	STD DEVIATION (MBITS/S)
100	190.3	1.56
1340	2589.0	14.49

Table 4: Performance without any service

5.2.1 *Point of reference (no service)*

Table 4 depicts results we got by running our benchmark test with no service on H5. In order to have a point of reference for the number of packets per seconds that our setup can handle, we used the option "M" of `iperf3` which specify the Maximum Segment Size. Normally, this kind of test is done by sending UDP packets instead of fixing the MSS of TCP. Unfortunately, UDP could not be used as a test because of some kernel issues with IPv6 Segment Routing. Note that the MSS without fixing it is equal to 1400 (MTU of the route) - 40 (outer IPv6 header) - 20 (TCP header) = 1340 bytes.

In effort to get a better idea of the performance of this setup, we ran each test 10 times and calculated the standard deviation. The formula used for the standard deviation is

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - m)^2}$$

where n is the size of the sample, x_i is the i-th element of the sample and m is the mean of the sample (the sum of the sample divided by the size of the sample).

We can notice that the MSS makes a huge difference in term of throughput. A MSS of 100 forces the kernel to craft a lot of packets. However, we can see that $1340/100 * 190.3 = 2550,02 \approx 2589$, meaning that the number of packets per second that our setup can handle does not depend on the size of transferred packets. As our NICs are able to handle 10 Gbps, we can say that the bottleneck relies on the number of packets that our kernel is able to send and receive per second. It is in order to achieve such outcomes that GRO and GSO are useful as these mechanisms alleviate the work of the kernel by allowing to handle fewer but larger packets.

5.2.2 *Asynchronous service (Counter service)*

We performed the two tests detailed beforehand with our COUNTER service; results are reported in the Table 5. Note that those tests were run with ring buffers of 128 MBytes without any thread. The service process was pinned to *CPU thread#0*. These choices will be explained later in subsection 5.3. Our throughput results are lower than without

MSS (BYTES)	MEAN (MBITS/S)	STD DEVIATION (MBITS/S)
100	105	0.06
1340	1389	5.67

Table 5: Performance with COUNTER service

MTU (BYTES)	MEAN (MBITS/S)	STD DEVIATION (MBITS/S)
100	105.3	0.823
1340	1434	8.43

Table 6: Performance with MEMLINK_TEST service

any service, meaning that our service is the bottleneck of this setup.

5.2.3 *Synchronous service (Memlink_test service)*

We performed the two same tests described in the previous section with the same choices with our MEMLINK_TEST service, results are reported in the Table 6. Results from this test and previous ones are aggregated in the summary, Figure 28. We can observe that synchronous and asynchronous services come with the same throughput. This means that we are limited by the number of packets per second that the kernel can handle.

5.3 VARYING RUNNING CONDITIONS

During our first benchmarks, we found that performance varied widely for our services. We explored several tracks in order to understand where these variations came from. Finally, we discovered the reason and found out a method to fix this issue. Following subsections explain some relevant experimentations that we have made in order to obtain the previously given results. As a synchronous service has more variable parameters, we performed all the following tests with our MEMLINK_TEST service.

5.3.1 *Scheduler*

During our first experiments, the throughput was not stable and varied during the tests. This behaviour triggered questions. Why was our service able to deliver 1Gbps but would then fall to inferior throughput before going back to 1Gbps. After a lot of measurements, we came to the conclusion that the scheduler was at fault.

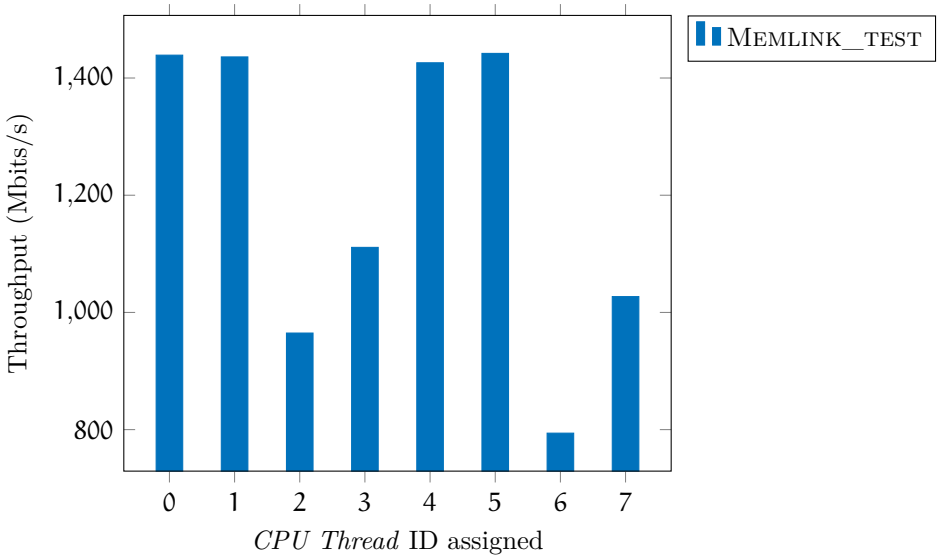


Figure 29: Performance with MEMLINK_TEST with no producer/consumer (1 thread) assigned on a specific *CPU thread* ID

To solve this problem, we pinned the cores used by the Network Interface Card (NIC). As a reminder, H5 is a quad-core xen processor with the Hyper-Threading technology. The Hyper-Threading technology allows concurrent scheduling of two processes per core. This means that we can have 8 simultaneous processes. In this master thesis, we call *core* a physical core of the processor and *CPU thread* a logical core enabled by the Hyper-threading technology. We count them from number 0. With this terminology, our servers have 4 *cores* (from 0 to 3) which give us 8 *CPU threads* (from 0 to 7). Both *CPU threads* $\#\mathtt{i}$ and $\#\mathtt{i}+4$ use the *Core* $\#\mathtt{i}$ and its L1 cache. As H5 have two links, we decided to pin a port to *CPU thread #6* and the other one to *CPU thread #7*, which means that the NIC uses the *core #2* and *#3*.

With the `taskset` linux program, we can launch a program and define which *CPU threads* can be used. In order to see the effect of the scheduler on our program, we launched ten times MEMLINK_TEST service (without any thread) with `taskset` for every *CPU thread* possible. This test was done without setting the MSS (hence, it is set to 1340). Results are reported in the Table 7 and Figure 29. We observe that the throughput is better on *CPU threads #0, #1, #4* and *#5*. This can easily be explained as the NIC uses the *core #6* and *#7* which means that *cores #3* and *#4* are intensively used by the NIC. With Hyper-Threading, *CPU threads #3, #4, #6* and *#7* are used by physical *cores #3* and *#4*. This explains that *cores* not used by the NIC provide higher throughput.

Previous tests were launched without our multi-threaded setup. In order to see how a multi-threaded system reacts, we decided to use

CPU THREAD ID	MEAN (MBITS/S)	STD DEVIATION (MBITS/S)
0	1439.00	16.63
1	1436.00	15.77
2	964.80	4.26
3	1111.00	7.37
4	1426.00	20.65
5	1442.00	14.75
6	794.70	6.00
7	1027.00	4.83

Table 7: Performance with MEMLINK_TEST with one thread assigned on a specific *CPU thread*

our threading system described in section 4 with one producer and one consumer. Without the help of `taskset`, the throughput of our tests varied widely. So, we tried to see the behaviour by setting a CPU affinity to our two threads (consumer and producer). We did this test with our two threading mechanisms : LFQ and Mutex. The results with Mutex and LFQ are depicted at the Table 8.

We can notice that what was true for one thread (no consumer/producer) is also true for two. But now, we have to find two physical cores which are not used by NICs in order to have the maximum throughput. Also, putting our producer and consumer on the same core gives really poor performance. We can also see that there is no configuration giving more throughput in this setup than in the previous one. We can conclude from this point that, for the MEMLINK_TEST which is not CPU greedy, adding a threading system deteriorates the performance. We can see that LFQ performs better than Mutex, except when everything is running on the same core. This can be explained by the fact that LFQ is doing busy-waiting, using a lot of CPU to check if there is something to process.

5.3.2 *Varying number of threads*

We have tested our system with one thread (no producer/consumer) and with one producer and one consumer. We now add more consumers in order to see how this setup reacts. Results are depicted in Figure 30. On the x-axis of this figure, 1 means no producer/consumer and otherwise it means the number of threads created for the consumer/producer (there is always one producer). For each situation, we forced our program to use the right number of threads with `taskset`, so the scheduler is not able to re-schedule anything on an unwanted *core*. As we can observe, adding our multi-threaded system only deteriorates the performance as there is an overhead inducted by the synchronization of the consumers.

Consumer \ Producer	0	1	2	3	4	5	6	7
0	441	1360	1330	1370	1130	1370	1350	1340
1	1360	420	1380	1370	1350	1130	1360	1330
2	944	943	370	949	938	951	911	945
3	1080	1070	1080	383	1080	1070	1090	640
4	1140	1350	1370	1370	435	1360	1370	1330
5	1360	1130	1360	1380	1370	493	1370	1350
6	774	770	630	774	773	784	316	775
7	971	980	999	787	986	989	987	343

(a) Lock Queue (Mutex)

Consumer \ Producer	0	1	2	3	4	5	6	7
0	135	1380	1390	1380	1150	1380	1380	1390
1	1350	153	1390	1380	1370	1160	1370	1370
2	950	949	128	955	958	952	918	949
3	1090	1090	1090	144	1130	1080	1090	1030
4	1130	1380	1380	1380	117	1380	1390	1360
5	1390	468	1370	1390	1380	143	1380	1380
6	775	774	358	780	773	774	106	779
7	999	992	1000	805	994	1000	1000	161

(b) Lock Free Queue (LFQ)

Table 8: Runs of MEMLINK_TEST with Lock Free Queue (LFQ) and Lock Queue (Mutex) by assigning consumer and producer threads to a determined *CPU thread*

Legend
- : < 500 Mbit/s
- : 500 - 800 Mbits/s
- : 800 - 1000 Mbits/s
- : 1000 - 1300 Mbits/s
- : > 1300 Mbits/s

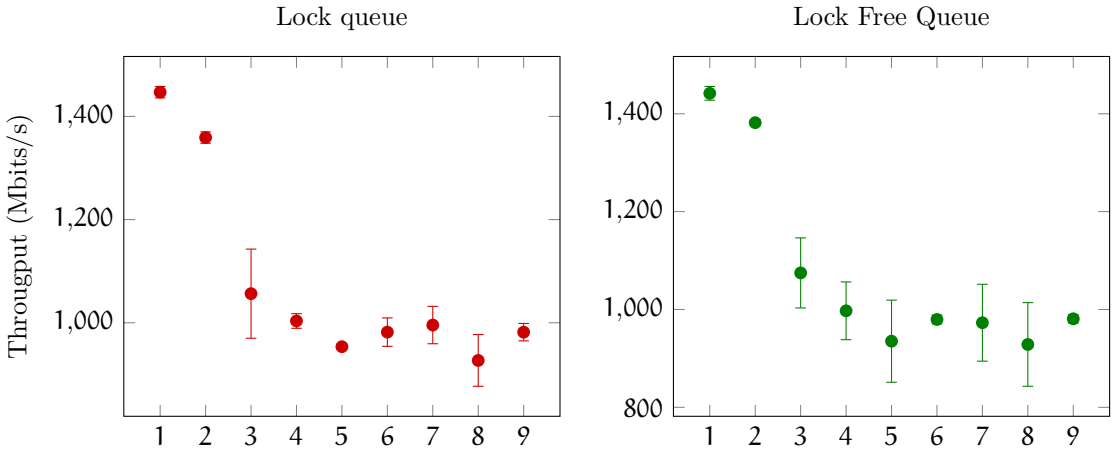


Figure 30: Comparison of Lock free queue (LFQ) and a Lock queue (Mutex)

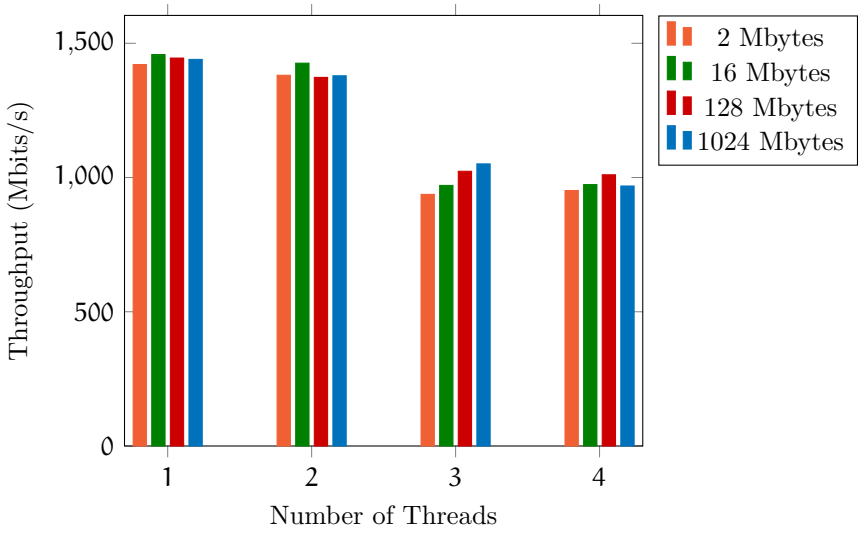


Figure 31: Comparison between multiple size of rx ring buffer memory

5.3.3 *Varying memory*

Incoming packets are written to the rx ring buffer before being handled by our service. In order to see if the size of the allocated memory makes a difference, we change it from 2 to 1024 MBytes with 1 to 4 threads with LFQ. Figure 52 depicts our results. We did not show the standard deviation to keep a clean figure, those are indeed similar to what we obtained in Figure 30. We cannot conclude that increasing the memory changes anything.

5.4 CONCLUSION

In this chapter, we presented the setup used for our benchmarks. We will use it for every other test in this Master Thesis.

From our tests, we can conclude that we need to pin the right number of *CPU threads* for each execution in order to get the best results. Also, the synchronization with a Lock Free Queue (LFQ) performs better than a synchronization with mutex. That is why we will not use the mutex synchronization in other tests. We have also seen that memory does not change performance. This chapter will be a point of reference for testing our services.

The next chapter will be dedicated to our first complex service which permits compression of network packets with Service Function Chaining.

6 COMPRESSION SERVICES

Compression is used widely in network transmission to reduce the volume of transmitted data and therefore reduce transmission time. In this chapter, we study the feasibility of a compression service with Segment Routing (SR). Why would it be interesting to use such a service? What would be the impact on the network?

6.1 STUDY OF THE PROBLEM

A compression service can be used to reduce networking costs by compressing data transmitted over a wire hold by an expensive Internet Service Provider (ISP). Two types of compression can be distinguished : lossless compression and lossy compression. With lossless compression, the original input can be recovered when the data is decompressed. In the contrary, a lossy compression loses information and the original data cannot be restored from its compressed version. For example, a JPEG compression is lossy and the final result is degraded.

6.1.1 *The TCP acknowledgement problem*

A commonly used compression service on the Internet is a compressor for the web traffic. The idea is that the service provides HTTP compression, helping the maintainers of a website to use less network bandwidth.

Our first idea was to try to apply the same idea with Segment Routing. For example, being able to compress images transferred by HTTP with a lossy compression service. However, we quickly noticed that such a service was impossible. Indeed, we cannot modify any parts of protocols that rely on TCP (Transmission Control Protocol). TCP is reliable: it is a stream-based protocol with checksum control in which clients acknowledge received data. Clients send back an acknowledgement based on the number of bytes received by the server. In fact, an acknowledgement is based on a sequence number negotiated during the initialization of the TCP connection. This number will be incremented during the connection by the volume of data correctly received by the client. Thus, if our service were to modify the payload of a packet in a TCP stream, we should disturb the operation of TCP.

To understand the problem, we simulated what would happen with such a service in the figure 32. The HTTP server sends data through TCP to a client. The service modifies the packets, changes the size of

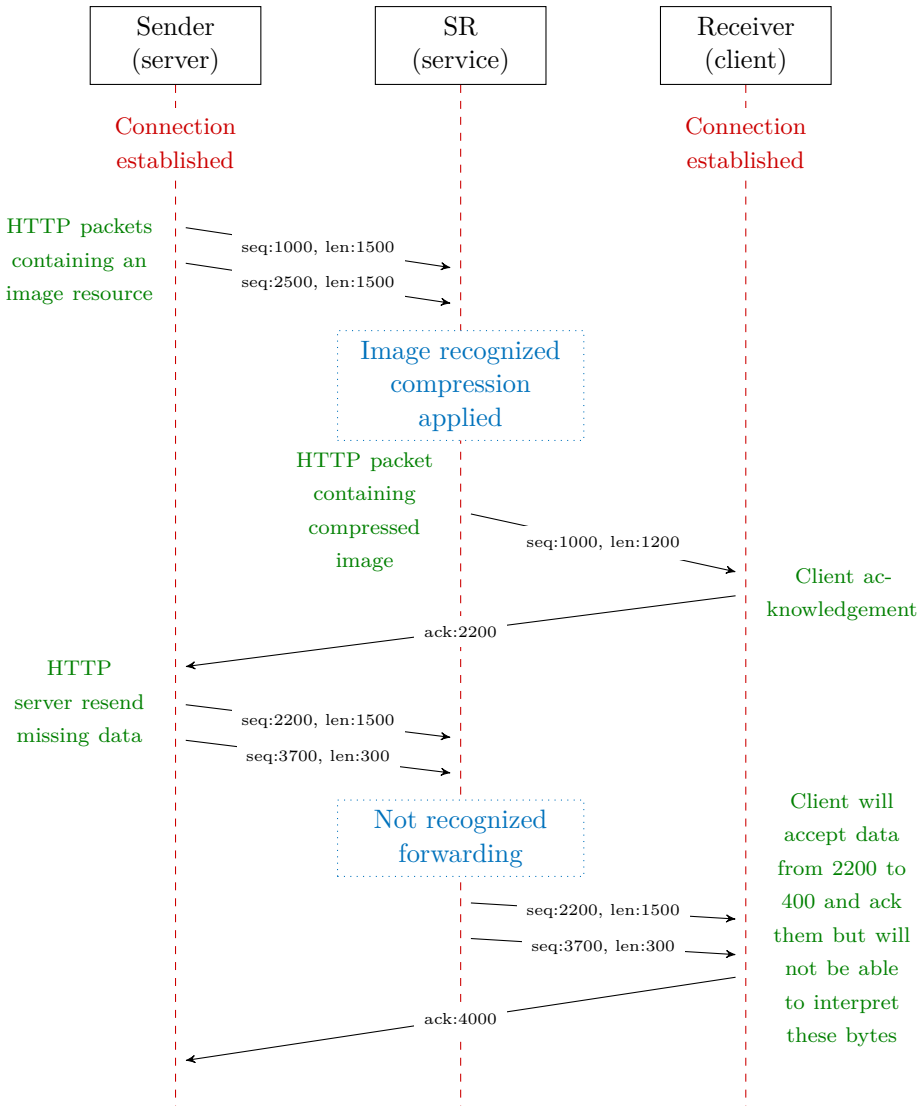


Figure 32: HTTP compression service with SR

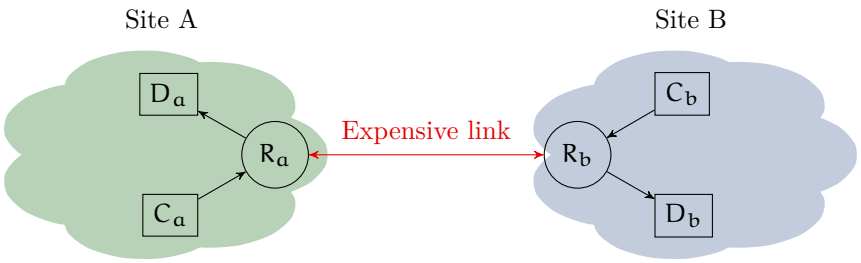


Figure 33: An use case of our compression service

the data and computes a new checksum. The client receives the data and sends an acknowledgement to the server. This acknowledgement is the sequence number of the first packet plus the size of the data. Since our service changes the size of the packet, the client would send to the server an acknowledgement that has a lower sequence number than expected by the server. The server would decide that the client did not receive the end of the message and would resend the incomplete part. Even if the client were to take into account the compressed resource by acknowledging it, the server would send more data than it should, making this compression service useless.

This problem could be circumvented if the acknowledgement from the client goes through our service on their way back. However, this is not possible as we cannot modify the behaviour of the HTTP client (browser). Plus, the Segment Routing Header (SRH) is dropped by the kernel, and the browser is not aware of it.

Solutions exists, as mentioned before, those act as a proxy server by creating two TCP connections. A connection from the client to the proxy server and another one from the proxy server to the web server. The query of the client goes to the proxy via the first TCP connection. Then, the service asks the web server for the requested resource of the client via the second connection. The service then compresses the resource and send it to the client. These kind of services are out of scope of this thesis as we want services that take advantages of SR.

6.2 OUR COMPRESSION SERVICE

Thus, to be able to create our compression service, we need two nodes: a first one to compress the data, and a second one to decompress it. The compression will be performed packet by packet. This enables our compression and decompression nodes to be stateless, removing the need for synchronization. The state must then be embedded within the packets. A stateless approach also has the advantage of being highly scalable.

Our compression service could be used by two sites of the same company who need to send uncompressed data over an expensive link con-

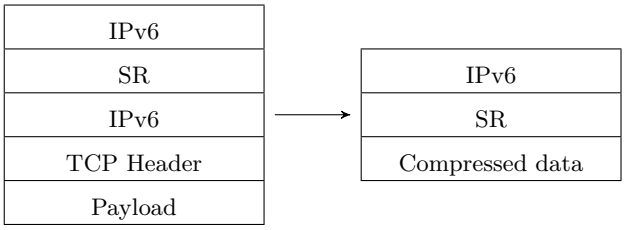


Figure 34: Example of a TCP packet with an SRH in encap mode compressed by our Compression Service

tracted via an ISP. Figure 33 shows this use-case. Both sites would need to have an SR-domain in which outgoing packets go through the compression service (C) to the decompression service (D). The SR-path of packets outgoing from site A would contain nodes C_a and D_b. The order matters, since the packets have to be compressed before going to the decompression service. And the SR-path of packets outgoing from site B would contain nodes C_b and D_a. Note that routers R_a, R_b and any routers between them do not necessarily need to be SR-capable for being able to forward this traffic.

6.2.1 *Operation of the Compression Service*

Since we want to have the best possible compression results, we use well-known libraries of compression utilities: lzo (Lempel-Ziv-Oberhumer), lz4 and zlib. The lzo and lz4 algorithms belong to the family of Lempel-Ziv algorithms ([LZ77]) known to have good throughput performance. The zlib (deflate algorithm) compression is known for being slow but also for providing excellent compression results. We have implemented it mainly for benchmarking.

Regardless of the compression library used, the behaviour of our service is always the same. First, we find the offset of the end of the Segment Routing Header (SRH). Then, we apply the compression (or decompression) function of the library (lz4, lzo or zlib) to the packet's data. We compress everything after the SRH, even if the next header is an extension header as shown in Figure 34.

6.3 COMPRESSION ALGORITHMS

To have a better idea of how a compression algorithm works, we briefly explain the Huffman coding and LZ77. lzo and lz4 are based on LZ77; zlib (deflate) uses both LZ77 and Huffman coding.

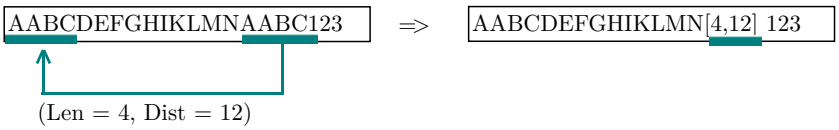


Figure 35: Compression with LZ77

6.3.1 *Huffman Compression*

Huffman compression, also known as Huffman coding, is used to reduce the number of bits needed to encode a message. To have a better idea of how the algorithm works, let us take an example. We have a plain-text message, whose characters are encoded in ASCII at 8 bits each. The idea behind the Huffman coding is that frequently-appearing characters should have shorter bit representations, and that less common characters should have longer representations. The algorithm creates a dictionary that giving new bit-representations for each character. This dictionary is included at the start of the encoded message, since it is needed to decode the data.

6.3.2 *LZ77 compression*

The LZ77 algorithm analyses a message and determines how to reduce the size of this message by replacing redundant sequences with metadata. It searches in a window (of the already read data) after the longest match with the beginning of the look-ahead buffer and outputs a pointer to that match by creating a special sequence of meta data. It is possible that there is no match at all, so the output cannot contain just pointers. The sequence of metadata is encoded in the form of a triplet $< \mathsf{o}, \mathsf{l}, \mathsf{c} >$, where "$\mathsf{o}$" stands for an offset to the match, "l" represents length of the match, and "c"" denotes the next symbol to be encoded. A null pointer (both offset and match-length equal to 0) is generated when there is no match. Fig 35 shows how an example sequence $AABC$ is replaced.

In the worst case, when no compression is possible, these compression algorithms will expand data a little bit. This means that the compressed packet will be larger than the original one. Thus, we need to ensure that the MTU of the outgoing interface is large enough to account for the expanded packets. Table 9 express the worst case for our compressor depending on the compression library expresses for a MTU of 1500 bytes.

6.4 BENCHMARKING: COMPRESSION RATIO

We define the compression ratio as $\frac{S_{in}}{S_{out}}$ where S_{in} is the volume of the data before the compression service, including a Segment Routing Header, and where S_{out} is the volume of the data after its compression.

LIBRARY	OVERHEAD	WORST FOR 1500 MTU
lzo	$\left\lceil \frac{\text{entry\_mtu}}{16} \right\rceil + 64 + 3$	1661
lz4	$\left\lceil \frac{\text{entry\_mtu}}{255} \right\rceil + 16$	1522
zlib	$\left\lceil \frac{\text{entry\_mtu}}{3200} \right\rceil + 6$	1513

Table 9: Overhead induced by lzo, lz4 and zlib for an entry MTU of 1500 bytes

A ratio less than 1 means that the data are incompressible, rendering the packet larger than before compression, because it now includes a compression overhead. Any ratio greater than 1 means that the data were compressible. For example, a compression ratio of 2 means that the size of the packet was halved.

6.4.1 *Compression ratio use-case*

For our first compression ratio benchmark, we chose to transfer an XML file that contains abstracts from Wikipedia [1] over SMB (Server Message Block). SMB is a protocol mainly used for providing shared access to files. It is used natively by Windows when Windows' users transfer files across a local network. We chose this transfer protocol since many companies run Windows and their employees are likely to use this file-sharing protocol.

In figure 33, we can image a company which has two sites at two different locations. The company pays an MPLS VPN (Multiprotocol Label Switching Virtual Private Network) [RFC 4364] to an Internet Service Provider (ISP) to connect their sites to communicate with each other. Computers from both sites share a single VLAN. For this VPN service, the Internet access provider charges them depending on their bandwidth usage. Our aim, in this scenario, is to show that our Segment Routing compression service could help this company in reducing its costs by using our service. Also, this kind of service could be offered directly by the ISP.

6.4.2 *Compression measurement on a compressible flow*

In order to benchmark the compression ratio of our service under realistic conditions, we have captured traffic from one of our computers with tcpdump. This traffic includes one hour's web-surfing. We then replayed the resulting dump through our compression service by encapsulating each packet of the dump with a Segment Routing Header containing three segments.

1 https://dumps.wikimedia.org/enwiki/latest/

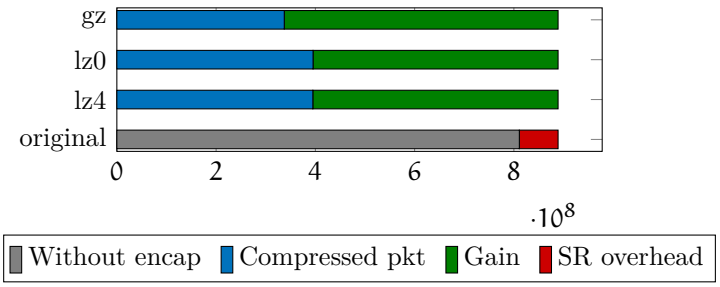


Figure 36: Comparison on compressible packets. Original packets size is depicted in grey, overhead induced by Segment Routing in red. Blue is the traffic size at the exit of our compression service for our different algorithms, spared traffic is depicted in green.

Figure 36 draws a comparison of our chosen algorithms. The overhead induced by the segment routing header equals 448 bits (64 bits for the compulsory fields of SRH + 3×128 bits for the three segments) times the number of packets sent in the flow. This overhead is compulsory, the first segment of the SRH will be the compression service, the second one the decompression service and the third will decapsulate the packet. The current implementation of SR in Linux does not allow us to perform a service on the last node. This current restriction should change in the future. The flow is represented in grey on in the Figure 36 and the red part is the total size of the overhead induced by Segment Routing. Spared bandwidth is depicted in green. Lz4 and lzo spares respectively 55.57% and 55.53% of the encapsulated traffic flow. The difference between these two algorithms is minor. The Gz algorithm spares 62.05% of the encapsulated traffic flow.

6.4.3 *Compression measurement by internet capture*

To benchmark the compression rate of our service in realistic conditions, we have captured some traffic from one of our computer with tcpdump. This dump is composed of different flows from a session of internet usage. We replayed this dump through our compression service by encapsulating each packet of our dump with a Segment Routing Header containing three segments.

In order to measure the compression rates, we tested our three compression libraries and compared the size of the packet before our service and after it. In order to have a better insight on these results, we classified packets by their protocols (TCP, UDP) and their ports. Our dump contains upload and download traffic, we managed to merge these two kind of traffic to only one flow as we want to have as much data possible. Also, to reduce noise, we decided to remove each flow with less than 100 packets.

Ports	Packets	SR Encap.	lzo	lz4	gz
TCP 80	188.27M	209.75M	204.33M	203.49M	203.30M
TCP 8000	10.28M	11.65M	9.59M	9.38M	9.57M
TCP 443	77.06M	87.30M	87.92M	87.60M	88.19M
UDP 443	86.40M	95.33M	95.52M	95.43M	95.94M

Figure 37: Comparaison between different states of the flows going through our compression service

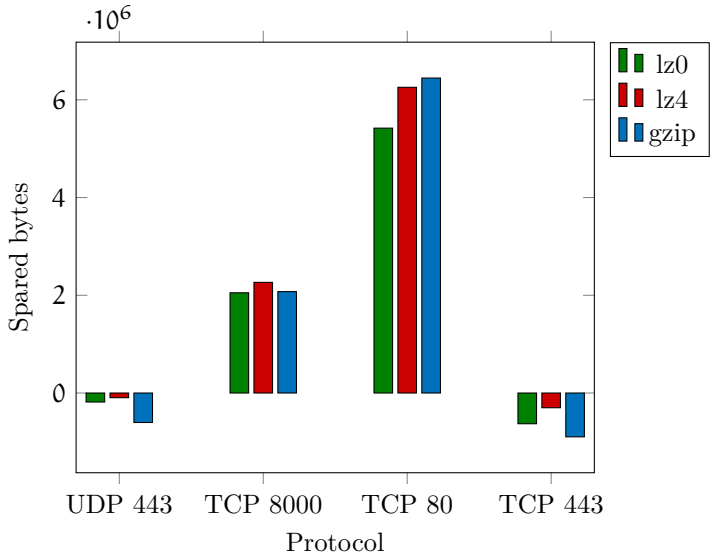


Figure 38: Spared bandwidth achieved on different kind of flows: comparison between the different algorithms. Overhead introduced by the compression algorithms can be greater than the savings from compression, resulting in negative gains

6.4.3.1 *Spared bandwidth by algorithm*

In the figure 63 in the appendix B.1, you can see the results of this experiment. The vertical scale is logarithmic for readability convenience. Conditions of this experiment are the same as described in section 6.4.2. We can observe that all flows of traffic are not equal with respect to their compression ratio. This depends on whether the protocols already comes with compression or encryption.

We chose to analyse in depth three kinds of flows: HTTP (TCP 80 and TCP 8000), HTTPs (TCP 443) and QUIC (UDP 443). We selected these because they are easily classifiable with respect to their protocol and port number, and they represent the majority of the traffic contain in our dump. Figure 37 shows the data collected from these flows, the cost of encapsulation with the Segment Routing Header (in encapsulation mode), and the results obtained by compressing these flows with our service using the three different compression algorithms.

Figure 38 shows the bandwidth saved for the flows from Figure 37. We notice two main things:

- The overhead introduced by the compression algorithms can be greater than the savings from compression, resulting in negative gains (packets are smaller uncompressed). This is the case with the HTTPS and QUIC protocols. QUIC is a protocol developed by Google to replace HTTPS in their services [QUIC]. This can be explained by the fact that both of these protocols use TLS/SSL encryption. Encrypted packets are impossible to compress since it is impossible to find repeated patterns. Therefore, the compressor will not reduce packet size, and worse, it will uses CPU and memory resources that serve only to enlarge the packet.

- With a plain-text protocol such as HTTP, our compressor can save some bits. We can see that it is hard to save as much as added by the overhead of the SR encapsulation (448 bits per packet in our example). This can be done on port 8000 but not on the standard port, 80. Also, we can see that we do not save many bits, this might be explained by the fact that a lot of HTTP traffic is already compressed in gzip. In December 2015, about 86.4% of website servers used the gzip compression [CheckGzip]. Our hypothesis is that HTTP packets sent on port 8000 are less susceptible to being compressed by gzip.

6.4.3.2 *Compression ratio by packet size*

To have a better overview of our HTTP traffic, we decided to graph the compression rate achieved with lz4, sorted by packet size, with a sample of 10000 packets. We merged traffic on ports 8000 and 80 for this experiment. Figure 39 shows that many packets have a bad compression ratio. The first subfigure shows the compression ratio for small packets (size $\leq$ 500 bytes), the second one for intermediate packets (size $>$ 500 and size $<$ 1400) and the last one for large packets (size $>$ 1400 bytes). We can observe that very small and large HTTP packets are higly incompressible. Meanwhile, intermediate packets are more susceptible to be compressed.

All sizes taken, 87% packets have a compression ratio less than or equal to 1 which means that 87% of the traffic of this HTTP dump is incompressible. This statistic is really close to the 86.4% from [CheckGzip], which means that gzip compression on HTTP is largely used. For the compressible traffic, we notice that larger packets are more susceptible to compression. This makes sense with regard to the lz4 algorithm: indeed, a larger payload allows the possibility of more recurrent patterns. Also, note that the maximum size of a packet is 1596 bytes, which comes from the maximum MTU of 1500 of the dump plus the 96 bytes of overhead from the IPv6 header and the SRH. We can

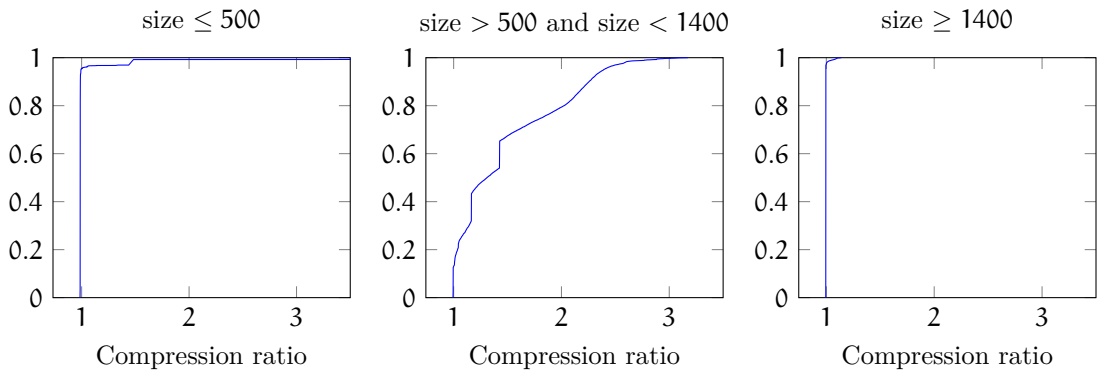


Figure 39: CDFs representing the compression ratio of incompressible packets in HTTP traffic varying in function of their size.

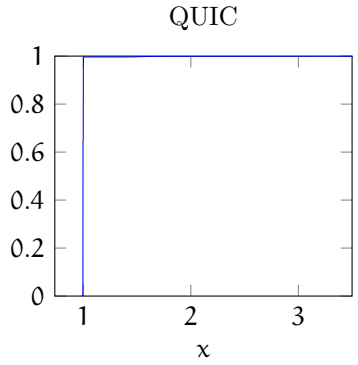


Figure 40: CDF representing the compression ratio of incompressible packets in QUIC traffic. We can see that no packets can be compressed at all.

see that we are able to compress packets of an intermediate size.

Figure 40 shows the compression rate obtained for the QUIC protocol, to confirm that an encrypted payload cannot be compressed. We were surprised that some rare large packets were compressed by a significant factor (> 1.4). Consequently, we inspected these packets to understand what happens. We discovered that these packets are "client hellos" that are padded at the end with many zeros. [IETF QUIC] defines a global minimum size for client hellos to limit amplification attacks. Since QUIC relies on UDP, it is easy to spoof an IPv6 address to create an amplification attack. By forcing the client to send a large packet, this attack is worthless, since the response from the server to the spoofed client will likely be smaller than the payload that the attacker has to send.

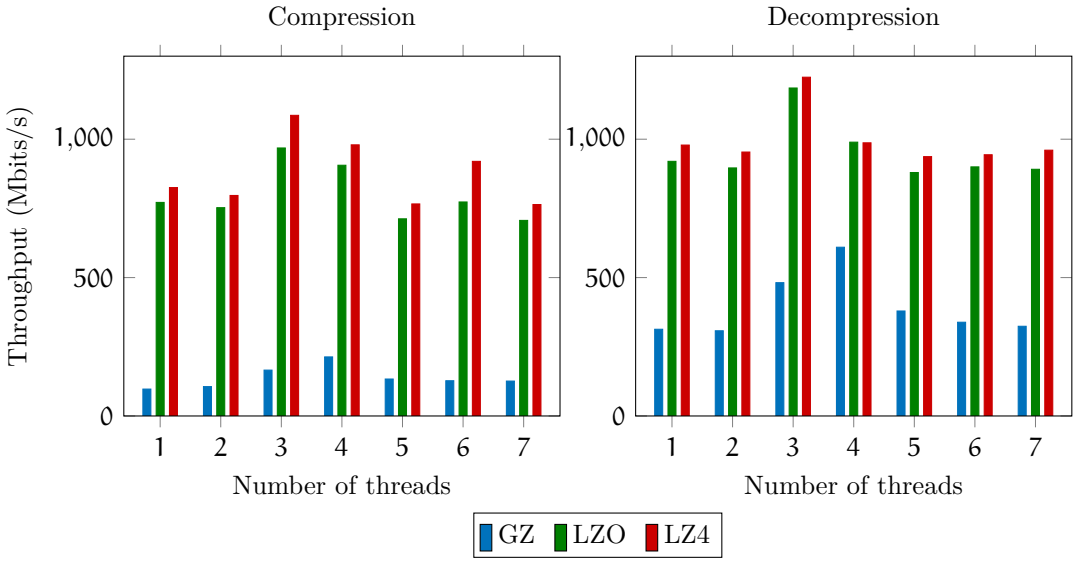


Figure 41: Throughput performance with 128 MBytes and multiple threads for our different algorithms. On the x-axis, we represent CPU THREADS where 1 thread corresponds to the single-threaded program and 2 or more rely on the multi-threaded program where there is one producer and $n-1$ consumer(s).

6.5 THROUGHPUT PERFORMANCE

In order to run this service, we need a node that compresses the flow and another one doing the reverse operation. As our benchmarking environment, inlab, only has one node to run our service, we have simulated our compression service to take a packet, applied the desired compression algorithm to the packet and then sending the original packet and not the compressed one. As we know that the size of a packet almost has no effect, this allows us to compare throughput obtained with benchmarks done on *memlink_test*. The same technique is used with the decompression service, except that we have a set of packets which are compressed instead of the original one.

Figure 41 depicts the throughput performance that our compression performs with different algorithms presented earlier. We ran this test with 128 MBytes of memory for the rx ring buffer. We varied the number of thread to see what happens.

First, we can observe that decompression is always faster than the compression. We can also observe that gz has really bad performances in comparison with lz4 and lzo. Lzo and lz4 better perform when running with two consumers, after that performance are even a little be degraded. However, with gz, we can notice that throughput increases with the number of threads used. When gz is using all possible threads (limited by taskset), the throughput is maximal. After what, performance also decreases. We can see that lz4 is always above the rest.

6.6 IMPLEMENTATION

You can add new compression algorithms with ease as the architecture of our compression service is made to be extended. To do so, you have to create a new C and H files implementing three functions : (1) a compression function which receives a packet in order to apply the compression on it; (2) a decompression function which also receives a packet in order to decompress it; (3) an initialization function which can be empty if no initialization is needed. Those three functions must be referenced into the compression.h file and the new files indicated into the Makefile.

6.7 CONCLUSION

This chapter explained the limitations of compression service we can make with Segment Routing and Service Function Chaining. Then, we described how we executed COMPRESSION of network packets with our solution and how compression algorithms work. We decided to implement three compression algorithms: lzo, lz4 and gz. Then we did some benchmarks on the compression rate and throughput. At the end, we can see that lz4 is the algorithm that fits the best our need as it provides a decent compression ratio with an intensive throughput.

In the next chapter, we introduce a new service that allows aggregating packets together.

7 BULK SERVICES

Inside a network, the main goal of a router is to route packets to their destinations based on a routing protocol that allows each router to discover the network topology. Routers decide the next hop that a packet will take depending on its destination. Obviously, hop-by-hop routing implies that each router inspects the IPv6 packet header in order to know the destination address. Even if this operation has been highly optimized, splitting data into a larger number of packets implies a higher workload for the network devices, which leads to a loss of throughput.

This is the entry point for our BULK service: aggregate packets into fewer bigger ones to reduce the number of packets treated by networking devices such as routers, DPI services, etc. We have designed two types of bulk service: (1) a generic bulk service called **packet bulk** in which packets are simply positioned behind a new header, and (2) a bulk service based on the Transmission Control Protocol (TCP) called **tcp bulk**. The PACKET BULK service needs another service that will execute the reverse operation. However, the TCP BULK service does not need this, since TCP is a stream-based protocol.

Bulking packets together results in building a bigger packet size which is still limited by the Maximum Transmission Unit (MTU). However, a packet cannot stay too long in our service. Otherwise, the transmission delay will increase, which is not desirable in any case. In TCP, the delay before acknowledgements received will have an impact on how fast the TCP congestion window increases (hence the throughput). When latency is high, it means that the sender spends more time idle (not sending any new packets), which reduces throughput.

As our service keeps packets in order to bulk them with others, when the sender stays idle for some time, packets are *stuck* in one of our buffer. To avoid this situation, we added a thread which is responsible for sending these packets periodically. This maximal waiting time, called *maximum delta time*, must be defined in arguments when launching the service.

7.1 PACKET BULK

PACKET BULK service aggregates packets that belong to the same path in order to reduce the workload of network devices (routers, DPI services, etc.). For example, this can be useful for transiting flows across Autonomous Systems (AS) which have a lot of traffic passing through.

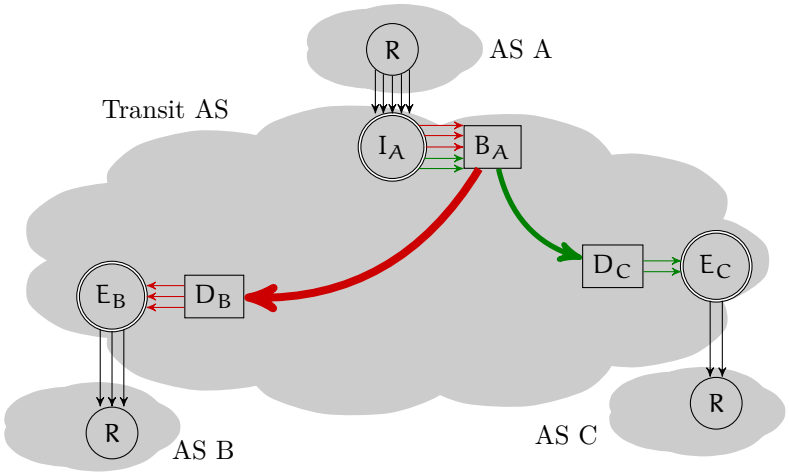


Figure 42: Use case of the PACKET BULK service for a Transit AS that is crossed by two kinds of traffic: from AS A to AS B in red and from AS A to AS C in green. B_A is a BULK PACKET service and D_C - D_B two DEBULK PACKET service.

This service must be coupled with a reverse service called DEBULK which recovers the original packets from the bulk packet.

Since SR is mainly meant to operate within a single AS, the Segment Routing encapsulation and de-encapsulation must be done inside the same AS.

7.1.1 *Operation*

Figure 42 describes an approach for reducing the impact of Transit traffic. In this example, a Transit AS is crossed by two kinds of traffic: from AS A to AS B, and from AS A to AS C. Packets are aggregated into two flows depending on their destination at node B_A: red or green for those which go to AS B or to AS C, respectively. Each flow has its own DEBULK service (D_B and D_C), which is responsible for recovering the original packets just before they leave the Transit AS. Obviously, the traffic bound for the AS itself does not need to be bulked.

To ensure that SR operates within a single AS (the Transit AS), device I_A acts as an ingress node by encapsulating traffic depending on its destination: (1) segments B_A, D_B and E_B for red flow and (2) segments B_A, D_C and E_C for green flow. Before leaving the Transit AS, E_B and E_C decapsulate traffic.

7.1.1.1 *Packet aggregation*

As shown in the example, Figure 42, all packets cannot be aggregated together because some of them should reach D_B, others D_C. Indeed, if all packets are aggregated together in order to reach for example E_B, those intended for AS C would take a longer path. In this case, the workload for Transit AS routers would increase rather than decrease.

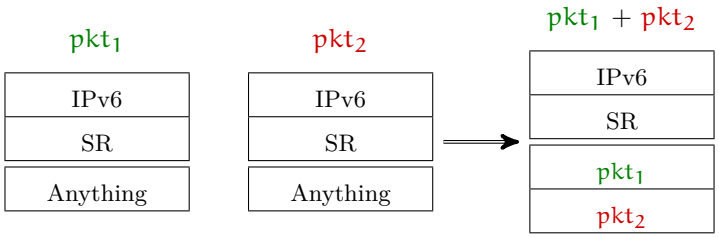


Figure 43: Aggregating together two packets : Two packet with same segments inside their SRH are aggregated together by the BULK PACKET service.

By taking this into account, we chose to bulk packets which have the same segments in the SRH. Indeed, packets with different segments mean that they follow distinct chains among service function. In our example, those which reach AS B or AS C contain respectively segments B_A, D_B and E_B or B_A, D_C and E_C. Differentiation based on segments is enough. To perform an efficient distinction, we compute an hash value for these segments. Figure 43 shows two packets, pkt_1 and pkt_2 with the same SRH segments bulk together behind a new IPv6 Header with SRH.

Therefore, the distinction between traffic which can be bulked together is accomplished by the ingress router when it performs the encapsulation.

7.1.1.2 *Packet recovery*

After aggregating packets as shown in Figure 42, we must recover the initial packets from the bigger bulk packet (pkt_{big}) with the opposite DEBULK service. Each packet in pkt_{big} starts with an IPv6 header and pkt_{big} contains only one IPv6 Header in front of all aggregated packets. So the offset in bytes for the first packet (pkt_1) inside pkt_{big} is found by searching the second IPv6 Header in pkt_{big}. The size of n^{th} packet (pkt_n) in pkt_{big} can be calculated by adding 40 to the length field of its first IPv6 Header (called *outer header*). So finally, the offset of a packet in pkt_{big} is computed as follow:

$$\begin{cases} offset(pkt_1) &= second\_IPv6(pkt_{big}) \\ offset(pkt_n) &= offset(pkt_{n-1}) + pkt_{n-1}.length + 40 \end{cases}$$

40 is added because the *length* field in the IPv6 header does not take its own size, 40bytes, into account.

7.1.2 *Header overhead*

Packet aggregation adds a new IPv6 and Segment Routing header. As shown in Figure 43, all packets are bulked behind a new header, the same as the headers of pkt_1 and pkt_2.

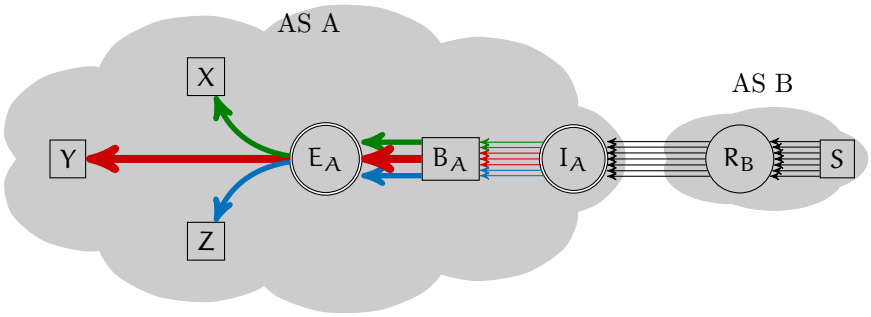


Figure 44: Use-case for the TCP BULK service with three different TCP connections: $S-X$, $S-Y$ and $S-Z$. B_a is a TCP BULK service.

7.2 TCP BULK

Transmission Control Protocol (TCP) supplements IP to reliably transport data from a source to a destination. TCP is stream oriented, so each TCP packet contains at least an IPv6 Header, and a part of the data stream characterized by a TCP Header.

Instead of aggregating the entire packet behind a new IPv6 header with Segment Routing, it is also possible to aggregate data from multiple TCP packets in only one. The purpose of the TCP BULK service is still to reduce the workload of the network router. The benefits over packet aggregation are twofold: (1) the proportion of header overhead is reduced because no additional header is added to the bulk packet, and (2) there is no requirement to have a *debulk service* since the bulked packet can still be understood by the destination. However, the second point is constrained by the MTU and the bulked packet must be lower than the MTU to reach that destination.

7.2.1 *Operation*

Figure 44 shows a typical use case for the TCP BULK service located at B_A. There are three TCP connections: $S-X$, $S-Y$, and $S-Z$. Since each TCP connection is independent from the others, our service must be able to distinguish each of them.

TCP uses a *sequence number* to identify the first data octet in a segment and all raw data positions are derived from this value. This means that only packets that contain consecutive data can be bulked.

Building this service raises some problems: how to disguish flows efficiently, what the bulking conditions are, and how to merge multiple TCP headers from the same TCP connection when packets are bulked.

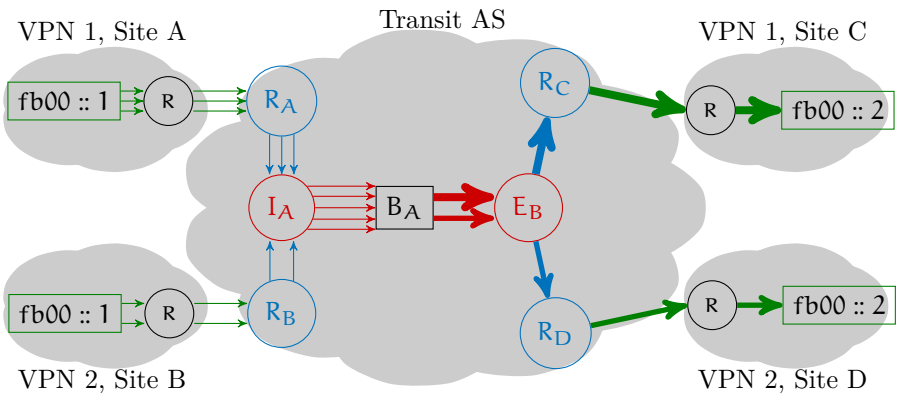


Figure 45: Bulking packets with with multiple IPv6 Header: case where our TCP BULK service receives packet with multiple IPv6 Header. Color red, green and blue describe which of IPv6 Header shown to Figure 46 are used to forward packet.

IPv6 (*src:* I_A, *dst:* B_A)
SR (*seg*$_0$*:* E_B, *seg*$_1$*:* B_A)
IPv6 (*src:* R_A, *dst:* R_C)
IPv6 (*src:* fb00 :: 1, *dst:* fb00 :: 2)
TCP
Data[x → y]

(a) Packet from site A to site C

IPv6 (*src:* I_A, *dst:* B_A)
SR (*seg*$_0$*:* E_B, *seg*$_1$*:* B_A)
IPv6 (*src:* R_B, *dst:* R_D)
IPv6 (*src:* fb00 :: 1, *dst:* fb00 :: 2)
TCP
Data[x → y]

(b) Packet from site B to site D

Figure 46: Example of a packet that might pass through our TCP BULK service

7.2.1.1 *Flow distinction*

Each TCP connection is identified by five pieces of information: the source and destination addresses, the source and destination ports, and a protocol set to TCP in our case. Addresses and ports can be retrieved from the IPv6 Header and TCP Header, respectively.

Unfortunately, taking only the nearest IPv6 header into account is not sufficient. Figure 45 shows a possible configuration thats leads packets with multiple IPv6 to our TCP BULK service. There are two VPN, each of which consisting of two site where the VPN isolation is based on IPv6 encapsulation. Figure 46 shows all IPv6 Headers added during a packet path:

1. Packet are crafted at private address *fb00::1* inside site A (resp. site B) and sent to address *fb00:2* inside site C (resp. site D) with green IPv6 as top header.

2. Router R_A (resp. R_B) performs an IPv6 encapsulation to be able to isolate each VPN. It sends this new packet to R_C (resp. R_D) in order to reach the site C (resp. site D). At this moment, blue IPv6 is the top header.

3. On the path, I_A performs a encapsulation with IPv6 and SR in order to add the TCP BULK service. Red IPv6 is the top header.

4. At the end, E_B and R_C (resp. R_D) remove IPv6 Headers that has been added respectively at I_A and R_A (resp. R_B).

The nearest IPv6 Header to TCP Header is the green IPv6 which contains for both VPN connection the same addresses: *fb00::1* and *fb00::2*. Assuming that the two connections use the same ports, we cannot perform our flow distinction only based on the last IPv6 Header and the TCP Header.

To deal with this special case, we need to take in account addresses of all IPv6 Headers inside a packet. In our case, the blue IPv6 contains different addresses.

For each incoming packet, we must efficiently find the data structure that contains previous packets for the same TCP connection. We store one data structure for each distinct TCP flow in a hash table, where the hash value is computed from two kinds of information (1) hash value of all source/destination IPv6 addresses found until the TCP Header and (2) source/destination ports. In order to avoid unnecessary memory to compute the hash value, we chose to build a hash table at two levels: the first one based on IPv6 addresses and the second one based on ports. Pseudo code shown at Algorithm 2 details how we perform our flow distinction. Note that we use the **Jenkin's** algorithm [**jenkins**] to compute the hash value.

Algorithm 2 Flow distinction

```
1: procedure GETFLOW(pkt)
2:     tcpoff ← FindTCPHdr(pkt)
3:     ipv6off ← FindIPv6Above(pkt, tcpoff)
4:     hashipv6 ← ComputeAllIPv6Hash(pkt, ipv6off)
5:
6:     addrflow ← GetOrCreateAddrFlow(pkt, hashipv6)
7:     flow ← GetOrCreatePortFlow(addrflow, pkt, tcpoff)
8:     return flow
```

7.2.1.2 *Merge header*

The TCP BULK service has the advantage of not adding additional headers but it does imply a way to merge headers. Indeed, bulking two TCP packets together shifts data from one to the other one, and merges the TCP header as shown in Figure 47. It goes without saying that we

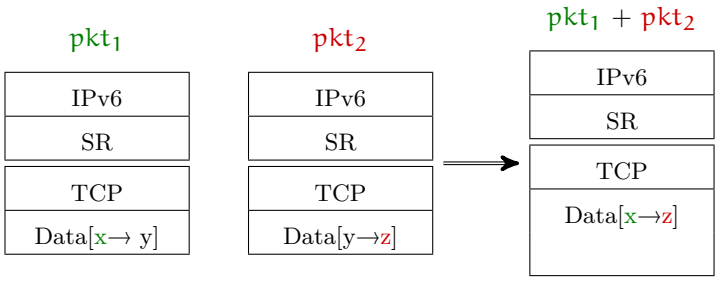


Figure 47: Two packets bulked together with the TCP BULK service.

update the length of the packet inside the IPv6 header. However we need to deal with the other TCP fields such as sequence number and acknowledgment number, the flags, and the checksum, or options.

The packet merging pseudo-code is summarized in the Algorithm 3.

DEALING WITH THE SEQUENCE NUMBER As explained earlier, only consecutive packets can be bulked together, so if packets are consecutive, the sequence number of the bulk packet can keep the sequence number associated with the first byte of data.

DEALING WITH TCP OPTIONS There are many TCP options available, and there are more to come, so our service must be able to deal with unknown options. Since TCP options can modify behavior of the protocol, packets should be bulked if and only if they strictly have the same options. It is not feasible to check each TCP option one by one in order to know if two packets have the same options. This is why we compute a hash value based on these TCP options and check if the values computed for two packets are the same.

PROCESS SPECIAL TCP FLAGS The special flags in the TCP header must have a special treatment: some of them prevent packets from being bulked and others may simply be aggregated.

- SYN, URG: These two flags prevent our service from bulking packets. packets. Indeed, in standard definition SYN flag inhibits packet to contains data and URG flag enables the *Urgent Pointer* field in the TCP header to be relevant, so they cannot be bulked without loss of information.

  At the end, when one of these two flags is set, the service sends the packet directly.

- CWR, ECE, RST, FIN, PSH: They can simply aggregate these flags with a simple **OR** operation.

  CWR and ECE contain some external information about congestion which need to be transmit to the end host. In the worst case

we reflect a lower congestion by aggregating multiple packets that encounter congestion in only one feedback to the destination.

RST and FIN reflect that connection must be closed when PSH informs the receiving host that the data should be pushed up to the receiving application immediately. So there is no information loss by performing a OR operation.

- ACK It can also aggregate with OR operation but, it implies some modifications at *Acknowledgment* field in the TCP header. As the acknowledgment is cumulative, we just take value from the latest incoming packet assuming that it is also the latest sending packet.

Algorithm 3 Merging two TCP packets $\mathtt{pkt}_1$ and $\mathtt{pkt}_2$ assuming that $\mathtt{pkt}_1.\mathtt{sequence} \leq \mathtt{pkt}_2.\mathtt{sequence}$

```
1:  procedure canBeBulked(pkt1, pkt2)
2:      hash1 ← hash(pkt1.options)
3:      hash2 ← hash(pkt2.options)
4:
5:      prevent ← pkt1.syn ∨ pkt1.rst ∨ pkt2.syn ∨ pkt2.rst
6:      consecutive ← pkt1.seq + pkt1.dataLen == pkt2.seq
7:
8:      return prevent ∧ consecutive ∧ hash1 == hash2
9:
10: procedure merge(pkt1, pkt2)
11:     if canBeBulked(pkt1, pkt2) then
12:         pkt1.ack ← pkt2.ack
13:         for flag ∈ [cwr, ece, rst, fin, psh] do
14:             pkt1.flag ← pkt1.flag ∨ pkt2.flag
15:         addData(pkt1, pkt2.data)
16:         return true
17:     else
18:         return false
```

7.2.2 *Mode*

There are two different modes which result from different implementations: Buffer mode and Queue mode. To represent the difference, let us say that $\mathtt{pkt}_1, \mathtt{pkt}_2$ and $\mathtt{pkt}_3$ are three packets that can be bulked together in this order and that $\mathtt{pkt}_2$ should arrive first to be processed by our service, then $\mathtt{pkt}_1$, and finally, $\mathtt{pkt}_3$.

7.2.2.1 *Buffer mode*

The first mode, called *buffer mode*, only bulks consecutive incoming packets if they respect the bulking conditions described in Algorithm 3.

If the TCP BULK service receives $\mathtt{pkt}_2$, $\mathtt{pkt}_1$ and then $\mathtt{pkt}_3$, it will output them without any aggregation. Indeed, our service will try to bulk (1) $\mathtt{pkt}_2$ with $\mathtt{pkt}_1$ in this order and then (2) $\mathtt{pkt}_1$ with $\mathtt{pkt}_3$.

This is implemented with a simple buffer containing the packets to be bulked. If the next packet can be aggregated, it is merged with the previous one in the buffer; otherwise, the current buffer is sent and the new packet takes its place in the buffer.

7.2.2.2 *Queue mode*

The second mode called *queue mode* allows a level of packet-reordering before it tries to bulk them. If the TCP BULK service receives $\mathtt{pkt}_2$, $\mathtt{pkt}_1$ and then $\mathtt{pkt}_3$, it will output the bigger packet, which is the aggregation of these three packets.

This is implemented by maintaining an ordered queue on the sequence number field of all incoming packets. When a packet comes in, it is added to the queue so that each packet before has a lower sequence number, and those after have a greater sequence number. When a sufficient number of packets can be bulked, a new packet is sent.

In addition, when a packet at position $\mathtt{i}$ is sent, we automatically send all packets before it, aggregated if possible, to avoid senders to trigger a fast retransmit packet.

7.3 BENCHMARKING

This section evaluates performance of BULK service through the TCP BULK service. Those evaluations are done via *iperf3*, generating perfectly ordered TCP traffic flows. It makes the service more efficient because it allows it to almost always aggregate packets together.

The PKT BULK is coupled with a reverse operation service which implies to have two services inside our test environment depicted in Section 5.1. However, we only have one available server to run our services. Running multiple services on the same machine would not be fair compared to the other benchmarks done before. That is why we do not include PKT BULK performance tests.

TCP BULK is evaluated by varying four parameters: (1) the implementation of the buffer, (2) the maximum delta time, (3) the multithreading and (4) the buffer size. We looked how the throughput and aggregation ratio behave.

We define this latter ratio for a flow as $\frac{N_{in}-N_{out}}{N_{in}}$ where N_{in} is the number of packets before the bulk service and N_{out} is the number of packets after it. This measure represents the proportion of the packets that has been aggregated to an existing packet. Hence, this number

represents the proportion of packets that have been removed from the network.

For the following tests, unless indicated, we use a maximum delta time of 20ms, a buffer size of 8000 and a MSS of 1340 bytes.

7.3.1 *Multi-threading*

Figure 49 depicts the throughput and ratio of TCP BULK in *Queue* and *Buffer mode*. As we can notice, the throughput is correlated with the ratio.

For TCP BULK in *Buffer mode*, increasing the number of threads decreases the aggregation ratio while this impact is not visible for the TCP BULK in *Queue mode*. This is due to the reordering of the packets and the implementation of those two modes. Figure 48 describes a mutli-threaded environment where the aggregation order can differ from the incoming order of packets. As depicted on the figure, (1) the packet $\texttt{pkt}_1$ is handled by a first blue thread, then this thread is "handed over" to the green thread; (2) this last one will handle the packet $\texttt{pkt}_2$ and try to aggregate it to (the) previous packet(s); (3) the blue takes over its execution and also executes the aggregation process.

Let us imagine that $\texttt{pkt}_1$ and $\texttt{pkt}_2$ may be aggregated together in respect to what has been described before in section 7.2.1.2. In this case, *Buffer mode* can aggregate them if and only if the aggregation process is performed on $\texttt{pkt}_1$ and then directly on $\texttt{pkt}_2$. Figure 48 shows a situation where the packets could be aggregated without multi-threading but is uncertain with two or more consumers as the scheduler can "hand over" a thread to another one. This is not the case with *Buffer mode* which is able to do reordering inside the aggregation process. This explains why the aggregation ratio for this last mode is steady while the *Buffer mode* decreases with the number of threads. Indeed, more threads implies having more rescheduling and the situation depicted before is increasingly frequent.

As shown in Figure 49, throughput seems linked with aggregation. As seen in Section 5.2.1, the throughput is more influenced by the number of packets rather than their size. This can be explained by the fact that a good aggregation ratio increases the throughput as there are fewer packets to send. This is why *Bulk buffer* can perform better than *Memlink_test*.

Bulk queue performs poorly as its reordering operation is more expensive than the gain obtained by the aggregation ratio.

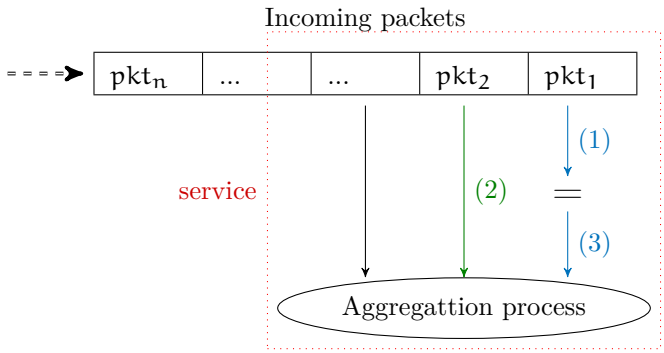


Figure 48: Scheduling problem with multi-thread implying that incoming packets can be reorder before executing aggregation process

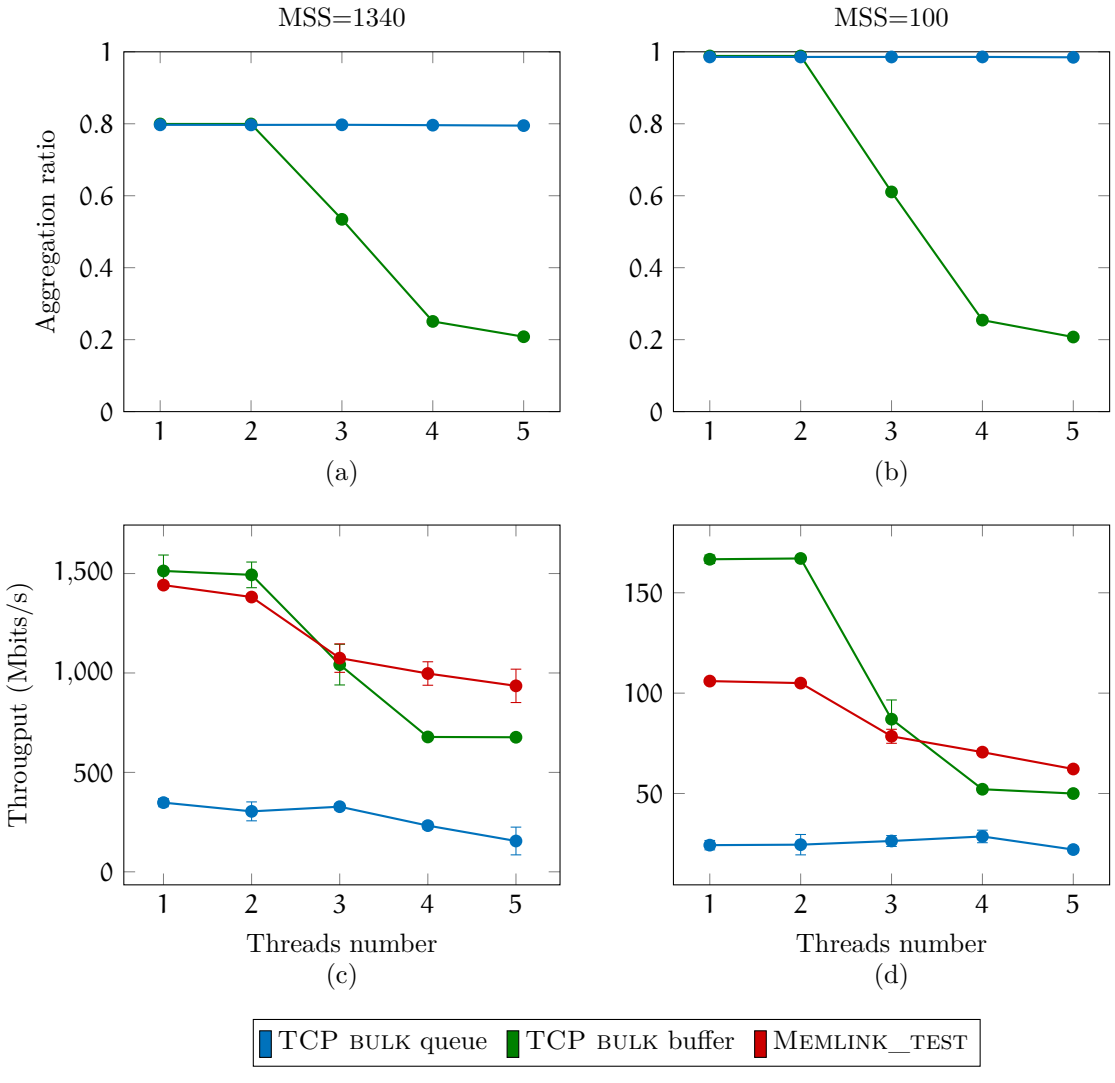


Figure 49: Bulk TCP performance depending on the number of thread

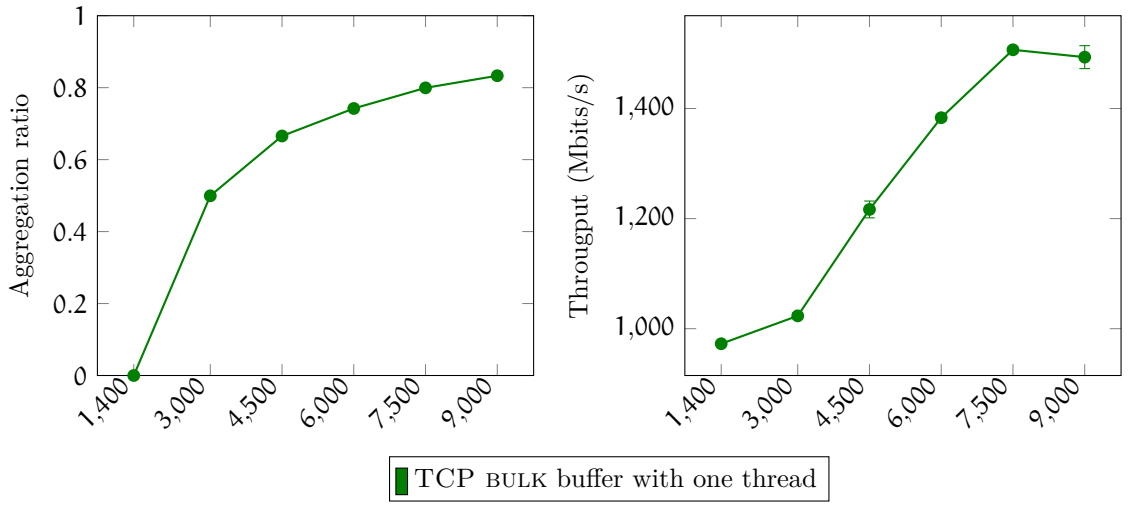


Figure 50: Performance of TCP Bulk buffer with a flow having an MSS of 1340

7.3.2 *Buffer size*

For a fixed MSS, we can calculate the maximum aggregation ratio for our service :

$$max(r) = \frac{N_{in} - min(N_{out})}{N_{in}} = \frac{N_{in} - N_{in} \times \frac{MSS}{\text{Buffer size}}}{N_{in}} = 1 - \frac{MSS}{\text{Buffer size}}$$

Because *iperf3* gives us a flow with ordered packets, our result at Figure 50 is close to an optimal. It confirms that the aggregation ratio mainly depends on the buffer size as described in the formula. For example, with a flow having a MSS of 1340 and a buffer size of 9000, the maximum aggregation ratio is $1 - \frac{1340}{9000} = 0.85$.

7.3.3 *Delta impact*

Figure 51 shows that *maximum delta time* seems to have no impact on the throughput for our TCP BULK buffer service. This appears logical as TCP BULK buffer keeps only a set of aggregated packets inside its buffer until a new packet impossible to aggregate arrives or until it becomes full. Hence, only a few number of packets can be hold during the *maximum delta time*.

7.3.4 *Parallel client impact*

We tested our TCP BULK buffer in a more realistic situation by having multiple simultaneous flows. Figure 52 shows the aggregation throughput of this flow depending on the number of threads. We observe higher throughput when there are more parallel flows than only one. This can

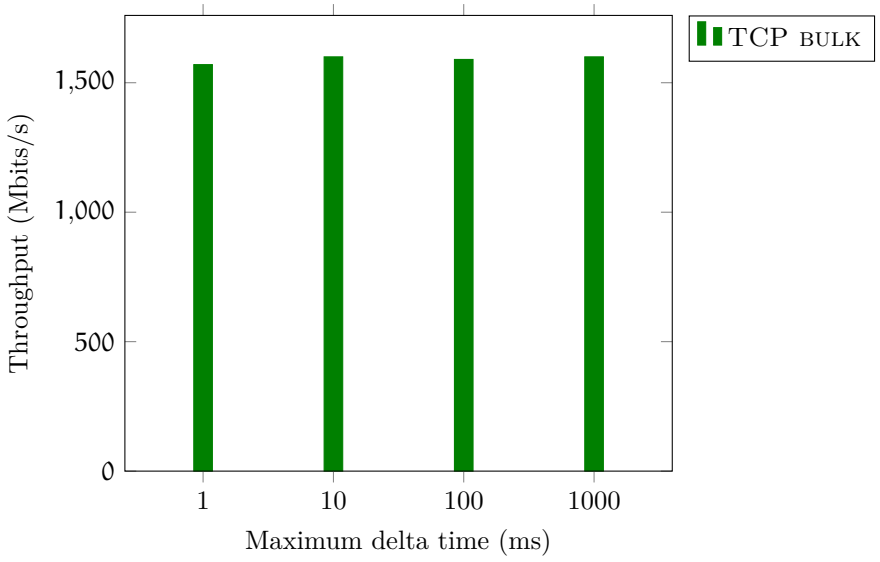


Figure 51: Performance of BULK TCP without multi-threading and a buffer of 8000bytes.

be explained as threads have a better probability to act on different flows, which reduces the probability to reach the case explained in the Figure 48.

7.4 CONCLUSION

This chapter presented the advantage of using a BULK service in order to decreases the number of packets transiting in a network. This service has two distinct bulk types: by packets and based on TCP flows. The first one has the disadvantage of adding a new overhead, while the second one obtains interesting results as it allows use to reduce the number of packets and improves the throughput of the network under certain conditions.

The next chapter reviews issues in the fragmentation header in IPv6 and addresses them with a Segment Routing service.

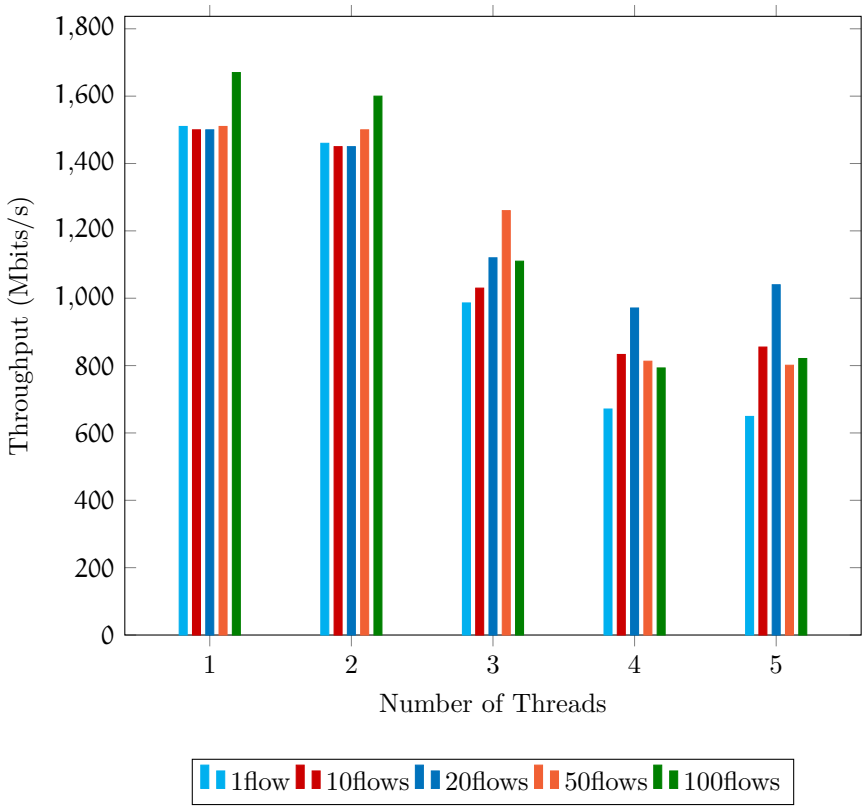


Figure 52: Impact of parallel client

8 RE-FRAGMENTATION SERVICES

The maximum transmission unit (MTU) is the size of the largest protocol data unit that a network layer can handle. In IPv4 and IPv6, if the data are too large for a network layer, the packet can be fragmented into multiple fragments that will be reassembled at the end-host. In IPv4, intermediate devices (such as routers) were able to break a datagram into an almost arbitrary number of pieces that can be reassembled later. Each of these pieces is small enough to pass over the single link for which the packet was fragmented using the MTU parameter configured for that interface.

In IPv6, the fragmentation can be performed only by the end-host that sends the packet. Therefore, an IPv6 host must perform an algorithm called "Path MTU Discovery" to determine the greatest MTU possible between two IP hosts. If the packet cannot go along its way because the MTU of the connection is inappropriate, the packet will be dropped and an ICMPv6 Packet Too Big (Type 2) message containing its MTU will be sent to the source host to allow it to reduce its Path MTU appropriately. Also, note that every IPv6 link must be able to deal with the minimum IPv6 MTU size of 1,280 bytes. If the source node does not perform the "Path MTU discovery", it limits itself to sending packets of this minimum size.

8.1 IPV6 FRAGMENTATION EXTENSION HEADER

IPv6 fragmentation is done by introducing a new extension header called "Fragment Header". The format of this header is shown in Figure 53.

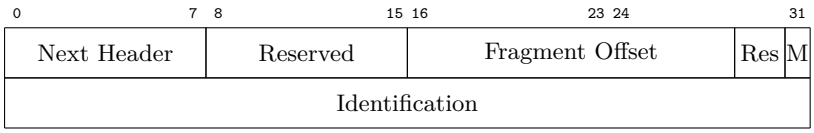


Figure 53: Fragment Header

NEXT HEADER 8-bit selector. Identifies the initial header type of the Fragmentable Part of the original packet (defined below). Uses the same values as the IPv4 Protocol field [RFC 1700].

RESERVED 8-bit reserved field. Initialized to zero for transmission; ignored on reception.

FRAGMENT OFFSET 13-bit unsigned integer. The offset, in 8-octet units, of the data following this header, relative to the start of the Fragmentable Part of the original packet.

RES 2-bit reserved field. Initialized to zero for transmission; ignored on reception.

M FLAG 1 = more fragments; 0 = last fragment.

IDENTIFICATION 32 bits. Identifies fragments that belong together.

8.2 STUDY OF THE PROBLEM

There are multiple ways to hide an attack by using fragmentation. This is not new to IPv6, so these attacks have been well-studied. The aim of such attacks is to send fake fragments in such a way that a deep packet inspection service does not reveal packets as dangerous. Those attacks, as summarised in [Newsham, 1998], are the following[Atlasis, 2012] :

- Disordered arrival of fragments (this may include reassembly of the packets before all the fragments arrive).
- IDS (Intrusion Detection System) flooding by partially fragmented datagrams (which may lead to IDS memory exhaustion and hence, to IDS DoS)
- Selective dropping of old and incomplete fragmented datagrams (if the dropping criterion used by the IDS is different than the one used by the end-systems).
- Overlapping fragments (and depending on whether the overlap favours old or new data). It can also result in attacks like the Teardrop attack.[RFC 1858].
- IP Options in Fragment Streams.

These attacks are nowadays hard to perform because they are known and good practices are defined to avoid them, see [RFC 1858]. However, some fragmentation attacks are still possible on some old operating systems. The IPv6 specification allows packets to contain a Fragment Header without the packet being actually fragmented into multiple pieces, we refer to these packets as "atomic fragments" [RFC 6946]. [RFC 6946] defines an attack in which an attacker can cause hosts to employ atomic fragments by forging ICMPv6 "Packet Too Big" error messages, and then it can launch a fragmentation-based attack against such traffic. [Shankar, 2003] describes an attack using the "Paxson/Shankar Model" that consists of a series of six fragments of varying

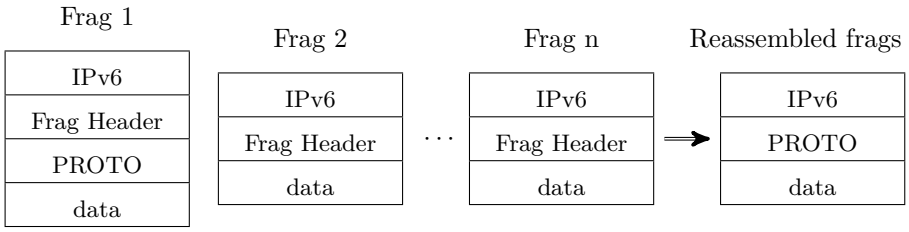


Figure 54: Re-Fragmentation Service aims to reassemble IPv6 fragments before reaching IDS and end-hosts

offsets, content, and length. The "Paxson/Shankar" attacks were not detected by any IDS in 2004. Other attacks based on fragmentation also allow attackers to gain the fingerprint of the targeted operating system (OS).

The purpose of the "re-fragmentation" service is to avoid attacks based on fragmentation by doing the reassembly with a Segment Routing service at the ingress of a SR domain. It reassembles fragments in one packet and removes the IPv6 fragmentation header. This avoids attacks based on fragmentation since we can be sure that the IDS will have the same view of the packets as the targeted OS.

Since legitimate reassembled packets are likely to be large, the MTU of the SR-domain should be large enough to allow packets to be re-fragmented. This service takes as a parameter the size of the buffer in order to forge the new packets. This argument must equal the MTU of the SR-domain. Since many devices can support jumbo frames, we recommend using an MTU of 9000 bytes for your SR-domain for the re-fragmentation service.

8.3 OPERATION

The Algorithm 4 gives an overview of how our algorithm works. The following paragraph explains this algorithm in depth.

When we receive an IPv6 packet with a fragment header, we create or retrieve the buffer identified by the identification number of the fragment header (Line 4 to 8). Then, we write the data of the packet at the offset referenced via the function `addToFrags`. This function also increments the value of the written data linked to its buffer. If the packet has the M flag equal to 0 (no more fragments), we save the total size of the reassembled packet (Line 14,15). If the packet has the offset 0 (first fragment), we save the number of the next header of the fragment (Line 16,17). Indeed, we will need to replace the next header from the upper header with this one when sending the packet without the fragment header. We send the reassembled packet if the buffer has timed

out after T seconds (where T is defined by an argument at the launch of the service) or if the written data are greater than or equal to the total size of the reassembled packet (Line 19). Any fragment that will create an overflow of the buffer is dropped. Thus, this service might create network connectivity problems (Line 11).

Algorithm 4 Refragmentation

```
1:  BUFFER_SIZE = argument for the maximal MTU
2:
3:  procedure FRAGS_IN(pkt)
4:      offset ← FindFragHdr(pkt)
5:      fhdr ← ParseFragHdr(pkt, offset)
6:      payload_offset ← FindPayload(pkt, fhdr)
7:
8:      frags ← FindOrCreate(fhdr.id)
9:      end_frags_offset ← fhdr.offset * 8 + hdr.size + offset
10:
11:     if end_frags_offset ≤ BUFFER_SIZE then
12:         addToFrags(frags, pkt, payload_offset, fhdr)
13:
14:         if fhdr.m == 0 then                          # Last fragment
15:             frags.end = end_frags_offset
16:         if fhdr.offset == 0 then                     # First fragment
17:             frags.next_header = fhdr.next_header
18:         if frags.wrote_bytes ≥ frags.end then        # All frags are
19:             send(frags)                              # reassembled
```

8.4 BENCHMARKS

We were not able to run any tests for this service due to a known issue in the Kernel. Indeed, the behaviour of the current Linux SR is to reassemble IPv6 fragments before encapsulating packets. This is not the desired behaviour and should be fixed in future versions. However, it is possible to verify that this service is working with our Segment Routing Functional Testing (SRFT) depicted in Chapter 11

8.5 CONCLUSION

This service shows that Segment Routing Service is able to address security issues. In SFC, this kind of service can be greatly appreciated. Indeed, network operators who want to protect their domains could easily define which services are needed depending on the operating systems for specific flows.

The next chapter introduces a service that enables other softwares to participate in our SFC even if those are not built to work with Segment Routing.

9 TUN SERVICE

Inspecting packets software such as DPI have already been implemented and optimized with a lot of functionalities. Creating new similar services for Segment Routing would be a waste of time. We wanted to find a way to be able to use already existing solutions as a service inside our Service Function Chain. To achieve this, we decided to offer a service that makes the bridge between our SFC, especially a service, and those existing softwares. This is the aim of our TUN service. As shown at Figure 55, TUN service provides a way to duplicate incoming traffic to existing softwares. This allows us to use available solutions with Segment Routing. This service works in the asynchronous mode (mode explained in Section 3.2).

9.1 OPERATION

The TUN Interface service creates a virtual network device called TUN interface. This service replicates every received packet on the newly created interface. This simulated device operates on layer 3 and all packets sent through it are delivered to user-space program attached to it. Note that packet sent by our TUN service must be IP packets.

Our service receives packets encapsulated in IPv6 within a Segment Routing Header. However, existing softwares might not understand the SR encapsulation needed in order to steer packets into our SFC. For this reason, when our service writes packets to the TUN interface, it removes the encapsulation.

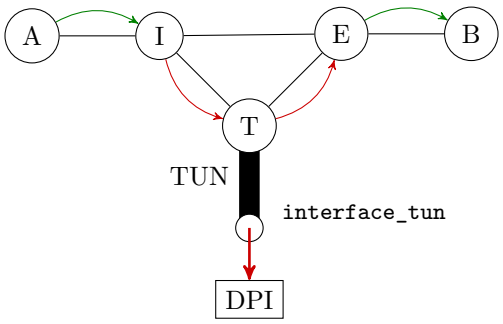


Figure 55: Tun setup

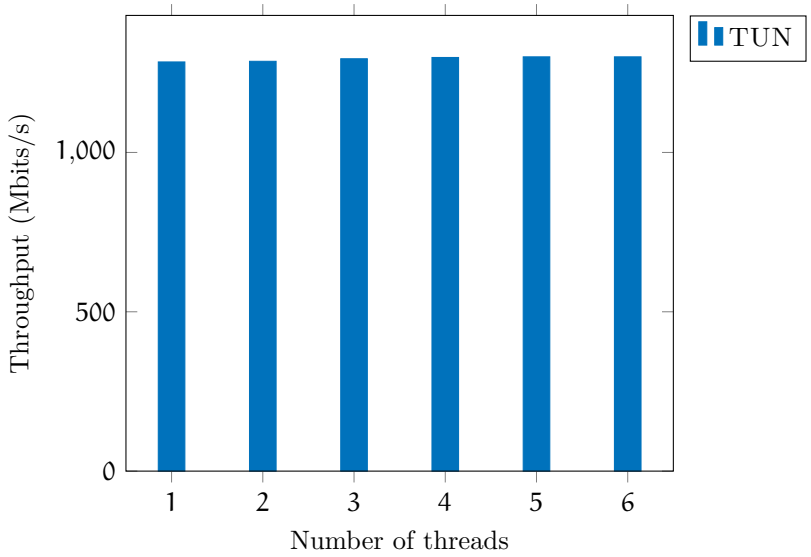


Figure 56: Performance with TUN service with multiple threads

9.2 SOME NOTES ON THE ENVIRONMENT

When launching our service, you have to specify the name of the tun interface you want to create. Let us imagine that this interface is called tunX. Given the fact that this interface is "down", you cannot attach any program on it as the interface is disabled. To enable the interface, you can simply use `iproute` or `ifconfig`. For example, `ifconfig tunX up`.

As the kernel reads the packets sent to TUN interfaces, we must implement a firewall rule in order to prevent packets from being treated by the kernel. Otherwise, packets would be forwarded if the preference of the kernel allows it. In order to prevent Kernel from forwarding incoming packets on this interface, we can apply the following ipatbles6 rule : `ip6tables -A FORWARD -i tunX -j DROP`. This command should be preferably ran before enabling the interface.

For our tests, we have used nDPI [nDPI] which is an Open and Extensible LGPLv3 Deep Packet Inspection Library. We can easily connect it with our service with the following command : `ndpiReader -i tunX`.

9.3 THROUGHPUT BENCHMARK

We have run our tests with multiple threads with 128 MBytes of ram for the rx ring buffer. We used `ndpiReader` from [nDPI] which is a lightweight DPI program that allows us to verify that our service works fine. Also, as this DPI does not perform CPU-intensive action on packets, we can be sure that it does not impact our benchmarking. Results are depicted in Figure 56.

We can observe that results are quite similar to those obtained in subsection 5.2.2 (Asynchronous service). Also, we can see that the

throughput does not drop when we have multiple threads. This can be explained as there is always a thread ready to process the incoming packet.

We can conclude that "copying" a packet on an virtual TUN interface is not resource expensive enough to alter the throughput delivered by an asynchrone service. This is quite good news in order to deploy existing solutions with Service Function Chaining with Segment Routing.

9.4 CONCLUSION

This service, easy to implement, enables to run a lot of existing solutions on our Service Function Chain with Segment Routing. This extends the field of application of our SFC Segment Routing to a new level.

The next chapter explains the environment that facilitates the development of our services.

Part III

ENVIRONMENT

10 DEVELOPMENT ENVIRONMENT

SEG6CTL enables to develop services in user-space which is developer-friendly. However, deploying locally a virtual Service Function Chaining in order to develop a service is a tedious task. In this chapter, we explain how we addressed this problem in order to facilitate the development of our services. We believe that this can be re-used in the future by anyone wishing to develop a new service with SR.

From our point of view, building a nice environment requires to address the following points:

- As our software is intended to run inside a network, we need to be able to simulate an entire virtual network, locally.

  Network are diverse and we must be able to test different topologies.

- Deploying a Service Function Chain requires to setup the topology which has been created before. The ingress node must be set up such that it performs the encapsulation on the right traffic and each service has to be launched on the simulated node.

- Debugging an SFC is mainly achieved by two way: (1) by using debugger such as *GDB* for the service itself and (2) by recording traces for network interactions bugs.

Network nodes are identified by one or more IPv6 addresses. We have to use them to reference our services in the Service Function Chain. However, this is not practical as the addresses change from one topology to another. To address this problem, we have a script that create labels which can be used in our environment for convenience.

In practice, the network nodes are identified by one or more IPv6 addresses. Those are usually used in order to define a topology. In our case, we can use them as a label for a service launched on a node.

The following sections of this chapter provide an overview of our development environment in order to break points explained before. Our environment is composed of a set of scripts that rely on two tools: (1) nanonet [Nanonet] which is a tool that allows to create a network topology locally based on configuration file and (2) tmuxinator [tmuxinator] which enables to automatically launch the SFC.

SOURCE	DESTINATION	WEIGHT	DELAY (MS)
A	B	1	2
B	C	2	1
B	E	1	1
C	D	1	1
D	E	1	1

Table 10: NTFL file example

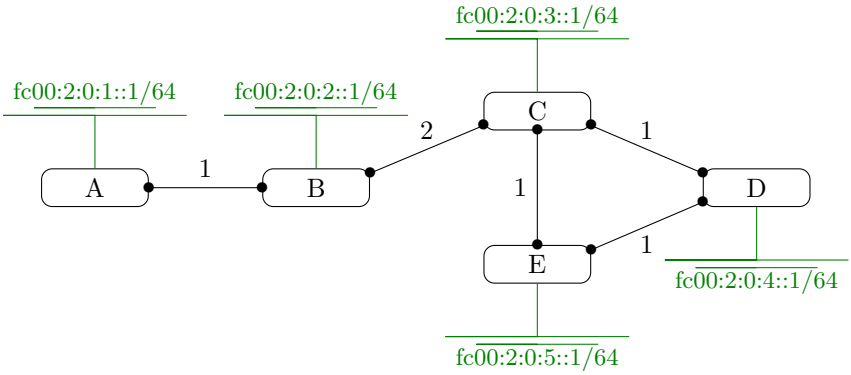


Figure 57: Topology generated by NANONET based on the NTFL file from table 10

10.1 NANONET

Nanonet [Nanonet] is a tool made by David Lebrun which allows to generate virtual networks from a topology description file. This tool uses network namespaces and virtual Ethernet pairs to build the virtual networks. Namespace is a Linux kernel feature that allows to isolate and virtualize resources of a collection of processes. In practice, processes are associated with a namespace and can only see the resources associated with that namespace. In this case, the networking aspects are isolated, i.e the network interfaces and routes. A virtual Ethernet pair enables communication between network namespaces through virtual interfaces. Moreover, NANONET automatically assigns loopback and interface addresses for each node. Then, it computes and installs all the routes needed for any node to reach any other node through their loopback addresses.

Table 10 shows an example file which contains a succession of links which are characterized with four fields: the node name of the source and the destination, the link weight and the transmission delay of the link.

Figure 57 shows the resulting virtual topology for the Network Topology File (NTFL) shown at Table 10. Each virtual node is linked with others by Virtual Ethernet (VETH) according to NTFL file. They have

interface addresses at each end of a VETH. In addition, each node has its own loopback address shown in green which is used as an IPv6 address to communicate with other nodes. Only loopback addresses are known by the entire network.

The virtual network creation from the NTFL file to the namespace generation is divided into three steps: (1) write a NTFL file, (2) run scripts on this file in order to generate a bash script which can be used to build namespace and (3) run this bash script.

10.2 LAUNCHER

After generating the virtual network, the services must be compiled and launched on their corresponding nodes (namespaces), as well as the traffic source and sink. This task can be either performed manually or be automated.

10.2.1 *Manually*

Let us consider a Service Function Chain composed of three different services: BULK, COMPRESSION and DECOMPRESSION. Imagine that you want to build the topology, apply it, compile each service used, launch the services on their own namespace, record input and output traffic and also launch a connection between two hosts by using for example `iperf3` [iperf3]. In this case, you would have to run *29* commands. It would be tedious.. We needed a way to automate this whole process.

10.2.2 *Tmuxinator*

Tmuxinator [tmuxinator] is a Ruby tool that allows to create and manage multiple terminals in only one, based on a configuration file. It relies on the terminal multiplexer tmux [tmux].

This tool enables to write a configuration file, called YML CONFIGURATION that will automatically perform all commands on distinct emulated terminals.

10.2.3 *Mapping*

To be launched, a service needs to know the address to which it will be bound. This address is in fact the loopback address associated with each namespace generated by NANONET. On topology modification, loopback addresses will change for a given virtual node from one topology generation to another, which imply the update of all configuration files.

To avoid this, configuration files should use labels to identify a node rather than its loopback address. We enable this feature by running

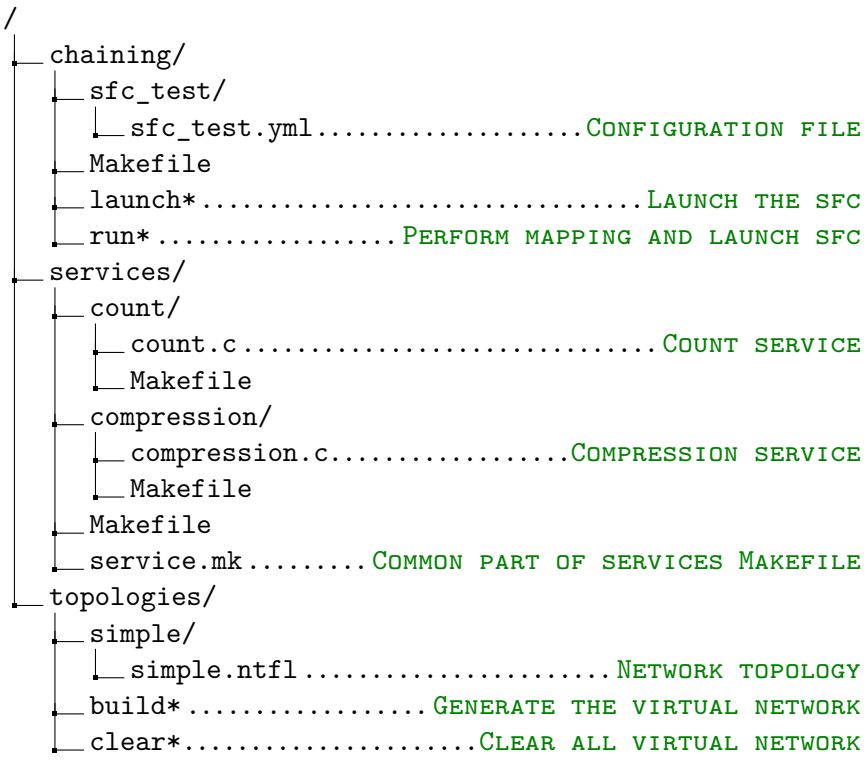


Figure 58: A directory structure.

a home-made script on the YML CONFIGURATION which performs the mapping between labels and their corresponding loopback addresses.

10.2.4 *Example*

The root directory of the development environment will be organized as shown in Figure 58. It is divided in three sub-directory: CHAINING which contains the SFC configuration files, SERVICES which contains services and Makefile to compile them and TOPOLOGIES which contains topologies files and script to build them.

Let us imagine a Service Function Chain (1) based on the topology described in Table 10, (2) composed of three different services: BULK, COMPRESSION and DECOMPRESSION and (3) that encapsulate traffic which passes by link B-C with our IPv6 Header and Segment Routing Header.

In this case, the Listing 6 in Appendix C shows how `sfc_test.yml` has to be written. When it is done, you just need to write one command from the root directory:

```
cd ../chaining/ && ./launch sfc_test
```

10.3 CONCLUSION

This chapter details our development environment based on *namespace*, which allows to test our services locally. In addition, it presents tmuxinator, a tools that permits to configure Service Function Chaining with a labelling system which allows to name and refer node.

The next chapter focuses on our SFC testing framework : Segment Routing Functional Testing. It enables us to verify that a Service Function Chain acts as expected.

11

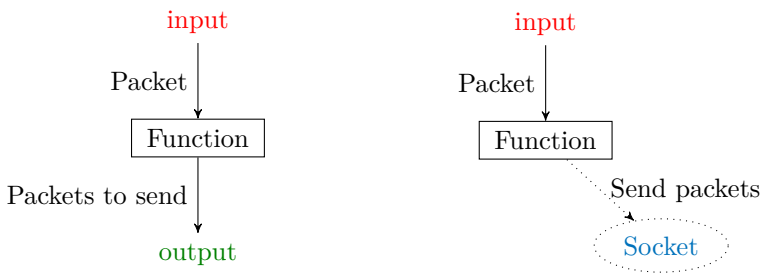
SEGMENT ROUTING FUNCTIONAL TESTING (SRFT)

We know that a test framework is useful for the services development. Typically we made sure that our library functions behaved as expected by using unit testing

We built our services and tried to test them, we faced a major issue: how to test our service function chaining inside a simulated network? For example let us consider the case of an Service Function Chaining composed by three of our services: BULK→ COMPRESSION → DECOMPRESSION. We should be able to test each service independently, and also test them together, as a chain.

By design, our service functions, such as BULK, are not simple functions that return an output resulting from a given input. When our service functions need to sent one or more packets, they communicate directly with the socket that is responsible for sending them. Figure 59 shows the main difference that explains why our service functions cannot be easily tested with a unit test such as CUnit[cUnit]. For the BULK service, when a packet arrives we do not necessarily have some packets to sent because they could be stored in the buffer. On the other hand, some packets can be sent by a timeout function without having an incoming packet.

We want to perform a unit test, we must collect an output that will be checked. It is possible to collect the packet sent to the socket, but it would take a lot of additional work. Nevertheless, if we can retrieve the output of a single services separately, it will be possible to chain them by giving the result of a service as the entry to the next service, just like a pipe. Obviously, this chaining also needs some additional work but, eventually, it will work.



(a) Expected function for unit test (b) Service function

Figure 59: Testing our service with a simple unit test.

However, there are other functional gaps in a basic unit test framework in addition to the difficulty of identifying, recovering, and composing output.

STACK It would be interesting to test the library on which our service relies. As explained before, all packets need to pass through the kernel and be retrieved by the seg6ctl library before they can be sent to our service.

Unfortunately, to test the stack with a basic unit test framework, we need to test the interactions between the kernel space and the user space, and that is not simple at all.

REMOTE Alternatively, it appears to be impossible to test an actual deployed setup. If you deploy your service on multiple hosts across the network, it is useful to be able to test it. Having a function where the test ensures that it has the expected behaviour is nice but it is not sufficient for us.

To test Service Function Chaining, we must take into account the global setup to ensure that the service has the right behaviour at all levels of operation and that packets follow the expected path.

11.1 SRFT

Faced with all these difficulties, we chose to build an improved unit test framework, called "Segment Routing Functional Testing" (SRFT). Built on top of a basic unit test framework, we wrote a specific way to perform a test case for our service based on Segment Routing on a network.

11.1.1 *Overview*

Our basic idea is to run our tests with a real topology in which packets are sent through a network. We do not worry about retrieval of the output of each service function because routing is responsible for forwarding packets from one service to the next. We only need to find a way to retrieve the packets sent by the last service. This way of testing is shown in Figure 60 for our BULK→ COMPRESSION → DECOMPRESSION example and contains four steps:

1. We crafted all packets needed in order to perform the test. For our SFC, the goal is to test that successive TCP packets are bulked and that the compression service retrieves the packets compressed. In practice, we just need to generate successive TCP packets for the same TCP connections.

   Note that they must have an adequate Segment Routing Header that corresponds to our chained service. For our example, we can craft the packets shown in Figure 61.

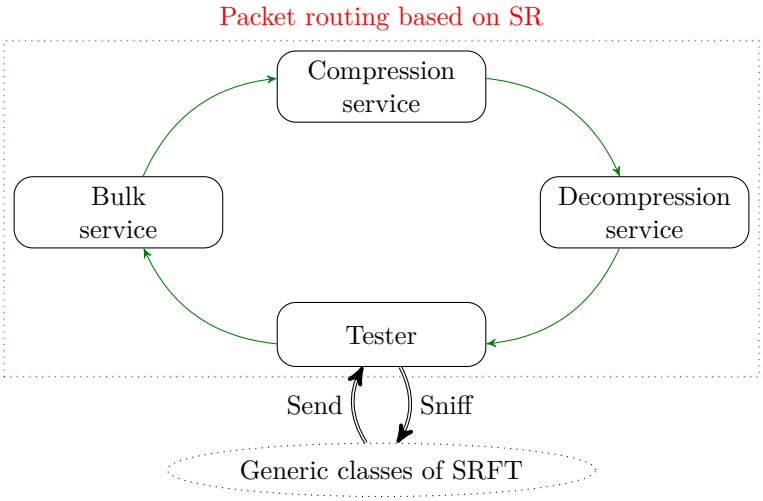


Figure 60: Basic environment setup for SRFT. Generic classes of SRFT are shown in Figure 62

pkt_1	pkt_2	pkt_3
IPv6	IPv6	IPv6
SR: Tester Decompression Compression Bulk	SR: Tester Decompression Compression Bulk	SR: Tester Decompression Compression Bulk
IPv6	IPv6	IPv6
TCP	TCP	TCP
Data[w → x]	Data[x → y]	Data[y → z]

Figure 61: Example of packets crafted to test BULK→ COMPRESSION → DECOMPRESSION

2. We send these packets inside the network and thanks to the routing, they follow the chaining service path described in the Segment Routing Header.

3. We retrieve all the resulting packets by sniffing them on the Tester's host. As Tester is the last segment in our Segment Routing Header, packets goes back to our Tester's host.

4. Now that we have sniffed all the resulting packets, we can perform a unit test to check if these packets match with the packets expected for the unit test.

   For our SFC and given the example crafted packet shown in Figure 61, the expected packet will contain `Data[w → z]` because the three packets must be bulked by the BULK service.

However, this raises at least one major problem: for a given test, how do we retrieve the packet that results from service function chaining without any doubt? It means that packets from two distinct tests cannot be mixed. This was not so easy to resolve because the retrieved packets are really similar.

11.1.2 *Flow label*

We chose to use the flow label fields inside the IPv6 header in order to make the distinction between flows from two distinct test. According to the IPv6 specification [RFC 2460], *"the 20-bit Flow Label field in the IPv6 header may be used by a source to label sequences of packets"*. The IPv6 Flow Label specification [RFC 6437] describe how flow labelling must be done. Globally they both explain that **flow label** can be used to distinguish different sequences of packets.

PACKET IDENTIFICATION As explained earlier, one of the main issues faced during the development of our test framework was to insure that the sniffed packets are legitimate packets. Indeed, if we craft a TCP packet, how can we identify without any doubt that the TCP packet sniffed is the result of the packet sent? To solve this, we chose to give a unique value to the **flow label fields** contained inside the IPv6 header. So each test case will retrieve its own flow label value used to identify all packets linked to that test case. In this way, there is no possibility of sniffing a 'foreign' packet because all packets related to a test case have the same, unique, flow label value.

MULTIPLE FLOWS By using the flow label field to identify which packet is linked to which test case, we also have the possibility of running multiple flows inside a test case by giving to each sub-test case a unique flow label value. We need to test how a service reacts when there are multiple simultaneous connections that go through it. With the flow label field, it is possible to write such a test without mixing packets from different flows inside the same test case. Thus, we can simulate that multiple senders using a service gets results that are individually those expected from each transmission, without interference.

11.1.3 *Generic classes*

Since a test case must be quite simple to write, we built some generic classes to provide abstraction when we write an SRFT test case. Indeed, these classes do a full job: they send crafted packets, they sniff the corresponding resulting packets, and they perform the comparison, the 2, 3 and 4 described in Section 11.1.1.

The interactions between the classes defined below are shown in Figure 62.

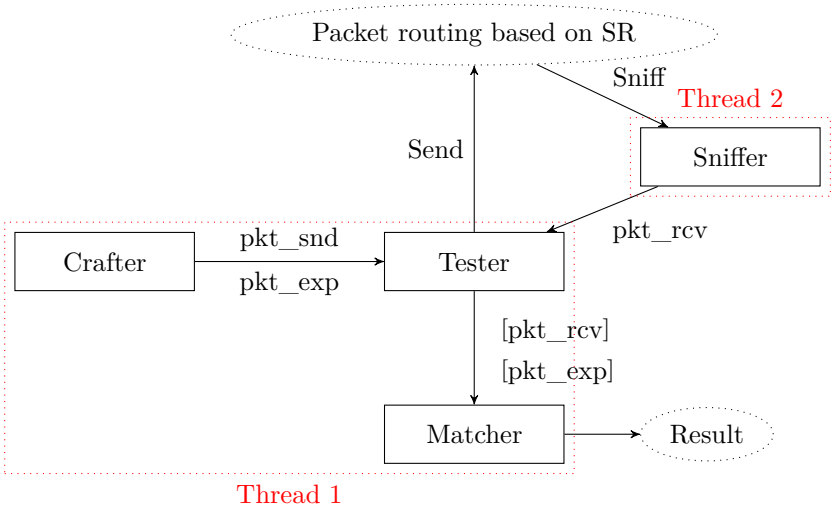


Figure 62: Generic class interactions. Packet routing based on Segment Routing shown in Figure 60

CRAFTER Used to generate packets that will be sent. A Crafter must receive:

- A craft function that returns all packets to be sent and all expected resulting packets.
- A Segment Routing Header added to each packet, to define the service function chaining used for this test.

MATCHER Used to perform the match between all packets received and all packets expected according to the flow label. It must receive a matching function that returns true if and only if two packets are similar, otherwise false.

Passing a function to MATCHER instead of using the default of full equality between packets gives us the possibility of removing information before performing the matching. For example, to avoid taking into account any specific field inside a header.

SNIFFER Used to sniff packets at a given interface according to a set of generic filter functions given in arguments.

As explained earlier, packets' owning test cases are distinguished by flow label, but you can add other filter functions if you want.

TEST Used to specify a test with a CRAFTER, a SNIFFER and a MATCHER.

Note that the filter based on flow label is automatically added when the test is created because at this moment we know which flow label number will be assigned.

TESTER Used to run the specified TEST class.

11.1.4 *Configuration file*

A problem which we have not talked yet concerning our SRFT, is the ability to know where (means, at which IPv6 address) each service is reachable in our network lab. The basic solution should have been to write this address directly inside the SRFT test case. But this is really not modular: if we were to change the location of our services, we would have to update all SRFT test cases in which we have been using the service. Obviously, this is not feasible.

In order to write an SRFT unit test without taking into account the exact IPv6 service addresses, we decided to use a configuration file to link a service with its address. This configuration file is given as an argument when a test is launched.

Listing 1: Example of configuration file

```
# Address when the test is launch
[Test]
Address= fc00:2:0:1::1

# Address for the compression service
[Compressor]
Address= fc00:2:0:5::1
```

LOCAL/REMOTE MODE In addition, it is possible to perform a test in local mode by using a configuration file called, for example, *local.conf*, and to perform the same test on a remote mode by using another configuration file, *remote.conf*. We actually tested two kinds of network topology:

1. Locally, with a virtual network built using our development environment based on NANONET, as explained in Chapter 10. In this case, only one device run all nodes, i.e the ingress and egress node(s), all services functions and the tester node.

2. Remotely, with a real network in a specific network lab. In this case, multiple devices can run one or multiple tasks. It means that our testing framework has the ability to test an SFC deployed inside a real network which can be useful to detect mistakes in configuration between services functions.

11.2 TOOLS

As explained at the start of this chapter, our SRFT framework will be an improvement of an existing unit test framework in which we have put some additional features in order to perform our specific test cases. To build this framework, we rely on two python tools:

UNITTEST MODULE [Unittest] It is the basic unit test framework used to perform our specific SRFT test case. It provides a ready ability to write a suite of tests with an understandable output.

SCAPY [Scapy] It is a powerful interactive packet manipulation program that allows the easy crafting, sending, sniffing, and inspection, of packets.

We chose to use python tools for two main reasons: it is a nice language that allows us to write code quickly; and we do not care much about run-time performance when we test.

11.2.1 *Unit test*

The first main tool used to build the framework is the unittest python module. This is a unit testing framework in python which supports some important concepts:

TEST RUNNER Executes tests and provides the outcomes to the user.

TEST CASE A unit test that compares the result obtained from a particular input with that expected.

TEST SUITE A collection of test cases.

TEST FIXTURE Represents some set-up and take-down operations to perform before and after a test case.

Typically, there is a test suite for each service that we need to test.

11.2.2 *Scapy*

Secondly, we chose to work with scapy to manage our packets. It is an open source toolkit and easy to use. In addition, scapy is well documented, with a lot of functionality already implemented. Furthermore, it is really easy to extend it, and to add additional functionality, such as our IPv6 extension header for segment routing. Our contribution to SCAPY is explained on chapter 12.

Of course, when we tried to design our test framework, we examined alternatives to Scapy. Initially, we tried to use Packetdrill, which is a tool that eases the writing of scripts to test the entire network stack. This tool is awesome for testing the functionality of a network protocol. Unfortunately, our goal was more to build a kind of unit test for the service that we had developed.

11.3 COMPARISON

We shall now make a brief comparison between our developed unit test framework (SRFT) and a standard unit test framework. The summary of this comparison is shown in Table 11.

	CUNIT	SRFT
Local mode	✓	✓
Remote mode	✗	✓
No addition work to retrieve function result	✗	✓
Allow chaining	✓	✓
Test the entire process	✗	✓

Table 11: Comparison between CUnit and SRFT

LOCAL-REMOTE SRFT allows testing of the setup in remote mode instead of in simple local mode. This is really useful, because instead of simply performing a test on the service, we test the entire stack. Note that in this case, routing across the service is also tested, and that is a good thing in remote mode.

RETRIEVE RESULT As our service is not a simple function in which a packet is the argument and the result, the return value, it was difficult to create a basic unit test case. Indeed, we had to find a way to retrieve the result which is the whole packet sent by the process. With our SRFT, there is no additional work surrounding the use of the service function because the packet follows regular routing inside the network topology. For local mode, this network must be created by using the development environment based on namespace.

STACK SRFT has the additional advantage of testing the operation in its entirety. We test the service function, but also the routing of packets across the service and the transfer of packets from kernel space to user space on the service host.

11.4 CONCLUSION

We have presented SRFT, our testing framework, which enables to test the complete Service Function Chaining instead of only the service function. It is written in Python with the unittest framework improved with our generic classes. Furthermore, it enables to test SFC in local or remote modes.

The next chapter concludes our thesis and develops our external contribution to open-source projects.

Part IV

CONCLUSION

12 CONCLUSION

In this chapter, we present the contributions made by our work. We start with what we did, what was already in service and how we have improved the existing systems.

12.1 OPEN-SOURCE PROJECT

Our whole Master Thesis is open-source and available on GitHub at `https://github.com/Tycale/SFC-SR`. In addition to our work, we have also contributed to open-source projects:

- Scapy[Scapy] is a powerful interactive packet manipulation program that allows you to craft, send, sniff and inspect packets. We contributed by adding support of Segment Routing Header, enabling crafting and inspecting Segment Routing packets.
- Wireshark[wireshark] is a packet analyzer used for network troubleshooting and for analysing communications protocol. We mainly used it to analyse unexpected behaviours of our SFC. We added the support of Segment Routing Header which allows analysis and inspection of Segment Routing packets.
- As shown in Chapter 4, our services rely on the SEG6CTL[Seg6ctl] created by David Lebrun. Initially, services were only able to handle one frame at a time. We added a multi-threading support which allows services to handle multiple frames at simultaneously.

12.2 OUR WORK

This Master Thesis covers multiple aspects for implementing Service Function Chaining with Segment Routing. SFC is useful to steer traffic through an ordered list of service functions which apply some processing on the traffic. Those services can be deployed based on Network Function Virtualization. This kind of architecture based on SFC and NFV enables to implement, deploy and maintain services with a high flexibility.

This Master Thesis began by explaining the concept of Segment Routing and its implementation through the IPv6 Extensions Routing Header. We provided an overview of the differences across routing type 0 (SR0) and 4 (Segment Routing), including design choices and security. SR0 has become deprecated due to security issues. We leveraged the possibility of performing SFC with Segment Routing which,

by design, can be used to steer packets into a sequence of instructions.

Our SRH-based service function chaining architecture offers the ability of creating a chain of connected network services. It requires an SR-domain where every service is able to deal with the Segment Routing Extension Header, plus some ingress and egress nodes which are respectively responsible for encapsulate/decapsulate the traffic.

Services functions run on the user space, which leads to easier service development. Those rely on the SEG6CTL library which implements a memory-mapped NETLINK-based data transfer mechanism between kernel space and user space. We contributed to explain its inner working in details, enabling future users to use it. We've also improved the library in order to support multi-threaded services without concurrency issues.

Moreover, we provided a development environment to create with ease new services, plus a development testing framework called Segment Routing Functional Testing which allows to easily test the proper functioning of Service Function Chaining with Segment Routing.

12.3 ACHIEVEMENT

We were successful on building multiple services with tests and benchmarks proving their operation.

We have provided a compression service coming with multiple algorithms of compression, and we have proved that this could be useful with protocol like Samba which does not include a built-in compression mechanism. We found that LZ4 algorithm is a perfect match for our service as it combines a good compression ratio while keeping high throughput.

Our TCP aggregation service allows to reduce the number of packets inside a network. We have measured its gains in multiple situations to determine its conditions for ideal operation. It exceeds the performance of our service that only forwards packets, called MEMLINK_TEST. This proves some of our assumptions about the limitation of the kernel concerning the number of packets per second it can handle. Indeed, this service allows the kernel to have less packets to send, and hence to perform higher throughput than our basic MEMLINK_TEST.

We have provided a fragmentation service that allows to reassemble IPv6 fragments in order to recreate the original packet. This service is useful for protecting some old Operating System having security issues with those kinds of packets.

Finally, we have created a service which enables programs agnostic to segment routing to plug on our Service Function Chaining without any modification. This provides backward compatibility, which facilitates the adoption of SFC with Segment Routing.

The throughput was decent in all the performed tests. We have also tried to gain performance on every of our service by enabling multi-threading support. However, results are not always conclusive when running on multiple threads as this mainly depends on the operation of the concerned service.

12.4 LIMITATION

Our Master Thesis relies on a Routing Header of IPv6, which should become the norm in the future for the datagram networks. However, currently, Network Operators are still quite cautious with respect to this change. Adoption is slow and operators may take time to adapt to new technologies that rely on Extension Header, such as Routing Header used by Segment Routing.

The Service Function Chaining with Segment Routing has some limitation due to its inherent work, as it rely on a SR-domain with ingress and egress nodes that perform IPv6-encapsulation with Segment Routing Header. The overhead should absolutely be taken into account as it will be quite substantial: $(24 + 8 \times \text{List size})$ bytes at least. This means that the MTU inside the SR-domain must be large enough to encapsulate incoming packets. The overhead can quickly become a limitation as it grows with the number of services inside a Service Function Chain. However, this brings the advantage of holding states only at the ingress nodes.

The solution described in our Master Thesis uses the Linux Kernel in order to offer services inside our SFC. However, we saw that throughput achieved by our services is far away from the throughput attainable with the Network Interface Controller (NIC) without any service. This limitation is due to the fact that we do not bypass the Linux kernel which is not suited for this kind of application, hence the insufficient performance.

Our ingress nodes perform classification based on a simple static routing table based on destination IP. This is sufficient for our purpose as we wanted to focus on service functions.

12.5 FURTHER WORK

It could be useful to work on a better traffic classification on ingress nodes. This would refine the paths taken by packets in our services. For example, this improvement would allow to compress only the HTTP traffic. However, this can be achieved by using Software Defined Network (SDN). In this case, the classification can be dynamically programmed instead of being static.

In this Master Thesis, we stuck to the use of the kernel IPv6 Segment Routing developed by David Lebrun in order to implement our services.

However, it would be possible to use other tools that bypass the kernel in order to achieve higher throughput, as DPDK[**DPDK**] for example.

12.6 CONCLUSION

During this Thesis Master, we had the chance to work with new technologies that are not widely developed but are starting to generate interests in the academic and industry fields.

We hope that our contributions will not be vain and will be useful to quickly develop new services with Segment Routing. Improving and documenting SEG6CTL should be useful to anyone who would wants to continue on this way. As IPv6 grows in importance, Segment Routing should spark interest of network operators. Being able to develop specific network services in user-land with Linux could be useful to prototype new network functionalities at lower cost.

We are delighted that we contributed to a new research field. Working with the new Kernel IPv6 Segment Routing also highlighted some issues that we hope will be fixed in the near future. We also hope that the network stack of the kernel will evolve to better support Segment Routing.

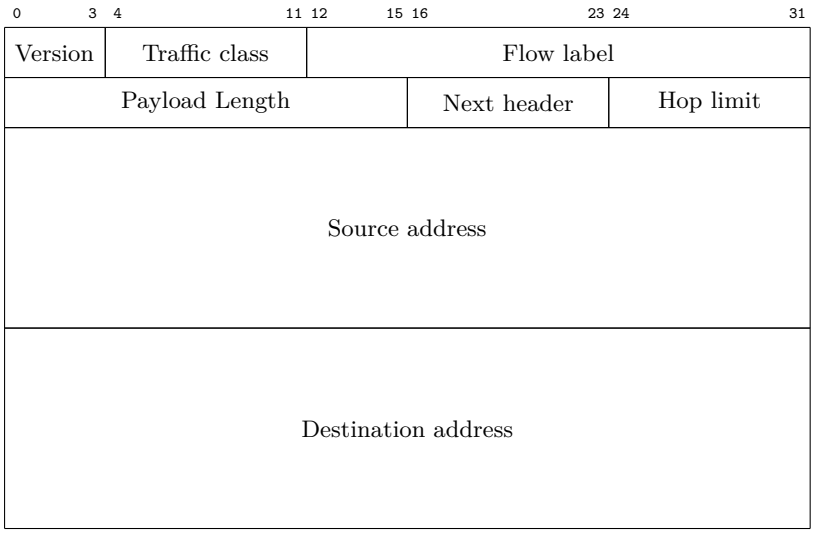
Part V

APPENDIX

A

SOME USEFUL REFERENCES

A.1 IPV6 HEADER FORMAT



Fields of the IPv6 explained below come from [Microsoft IPv6, 2008].

VERSION 4 bits are used to indicate the version of IP and is set to 6.

TRAFFIC CLASS Indicates the class or priority of the IPv6 packet. The size of this field is 8 bits. The Traffic Class field provides similar functionality to the IPv4 Type of Service field. In RFC 2460, the values of the Traffic Class field are not defined. However, an IPv6 implementation is required to provide a means for an application layer protocol to specify the value of the Traffic Class field for experimentation.

FLOW LABEL Indicates that this packet belongs to a specific sequence of packets between a source and destination, requiring special handling by intermediate IPv6 routers. The size of this field is 20 bits. The Flow Label is used for non-default quality of service connections, such as those needed by real-time data (voice and video). For default router handling, the Flow Label is set to 0. There can be multiple flows between a source and destination, as distinguished by separate non-zero Flow Labels.

PAYLOAD LENGTH Indicates the length of the IPv6 payload. The size of this field is 16 bits. The Payload Length field includes the extension headers and the upper layer PDU. With 16 bits, an IPv6 payload of up to 65,535 bytes can be indicated. For payload

lengths greater than 65,535 bytes, the Payload Length field is set to 0 and the Jumbo Payload option is used in the Hop-by-Hop Options extension header.

NEXT HEADER Indicates either the first extension header (if present) or the protocol in the upper layer PDU (such as TCP, UDP, or ICMPv6). The size of this field is 8 bits. When indicating an upper layer protocol above the Internet layer, the same values used in the IPv4 Protocol field are used here.

HOP LIMIT Indicates the maximum number of links over which the IPv6 packet can travel before being discarded. The size of this field is 8 bits. The Hop Limit is similar to the IPv4 TTL field except that there is no historical relation to the amount of time (in seconds) that the packet is queued at the router. When the Hop Limit equals 0, an ICMPv6 Time Exceeded message is sent to the source address and the packet is discarded.

SOURCE ADDRESS Stores the IPv6 address of the originating host. The size of this field is 128 bits.

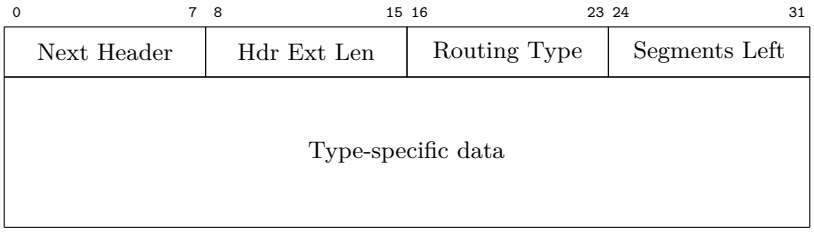
DESTINATION ADDRESS Stores the IPv6 address of the current destination host. The size of this field is 128 bits. In most cases the Destination Address is set to the final destination address. However, if a Routing extension header is present, the Destination Address might be set to the next router interface in the source route list.

A.2 PROTOCOL NUMBERS

An exhaustive list of protocol numbers is maintened by the IANA and can be found in [Protocol Numbers].

A.3 ROUTING HEADER FORMAT

A description of the fields can be found in the section 2.5.

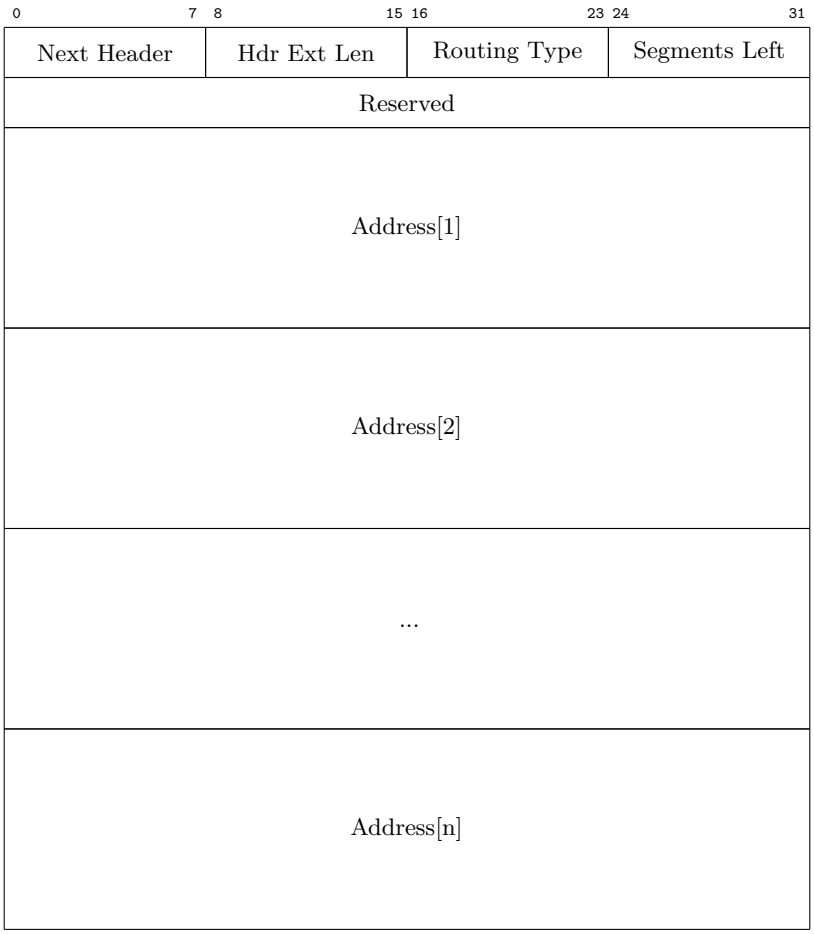


DECIMAL VALUE	HEADER
0	Hop-by-Hop Options Header
6	TCP
17	UDP
41	Encapsulated IPv6 Header
43	Routing Header
44	Fragment Header
46	Resource ReSerVation Protocol
50	Encapsulating Security Payload
51	Authentication Header
58	ICMPv6
59	No next header
60	Destination Options Header

Table 12: Protocol numbers (non-exhaustive list)

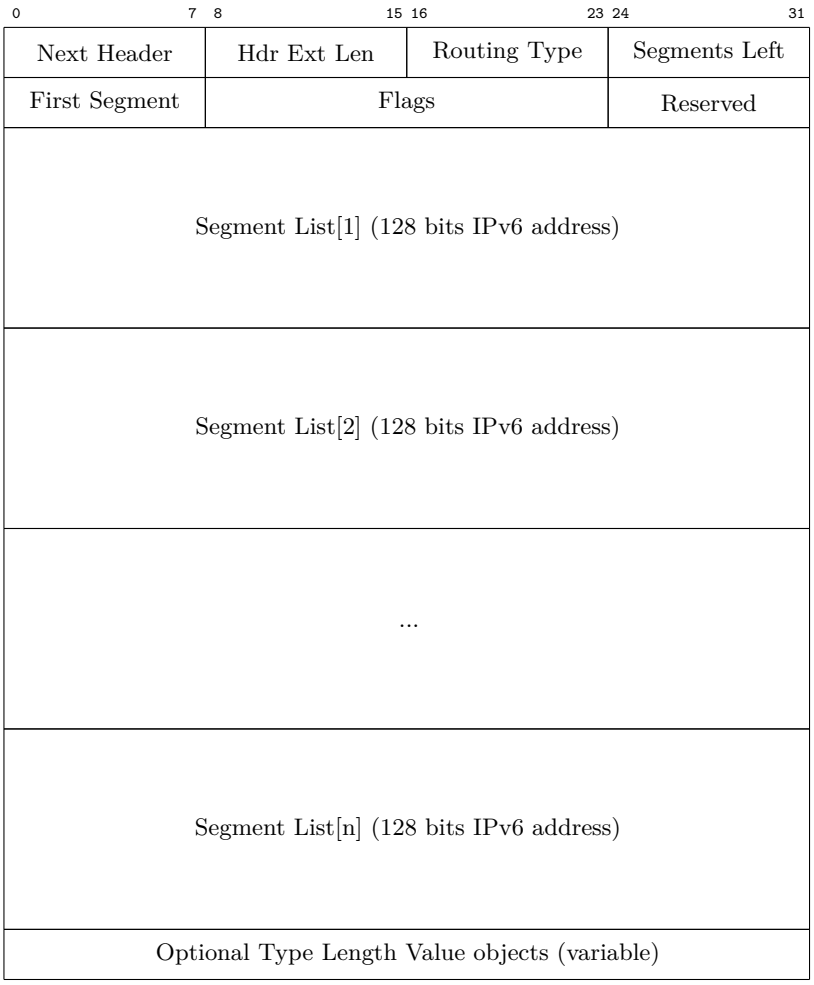
A.3.1 *Type 0 Routing Header Format*

Description of the fields can be found in the section 2.3

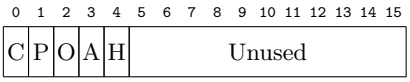


A.3.2 *Segment Routing (Type 4 Routing Header) Format*

Description of the fields can be found in the section 2.5



A.3.3 *Segment Routing Flags*



C-FLAG Clean-up flag. Set when the SRH has to be removed from the packet when the packet reaches the last segment.

P-FLAG Protected flag. Set when the packet has been rerouted through FRR (Fast Reroute) mechanism by an SR endpoint node.

O-FLAG OAM flag. When set, it indicates that this packet is an operations and management (OAM) packet.

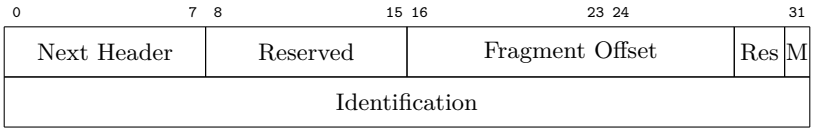
A-FLAG Alert flag. If present, it means important Type Length Value (TLV) objects are present. See Section 3.1 for details on TLVs (Type-Length-Value) objects.

H-FLAG HMAC flag. If set, the HMAC TLV is present and is encoded as the last TLV of the SRH. In other words, the last 36 octets of the SRH represent the HMAC information.

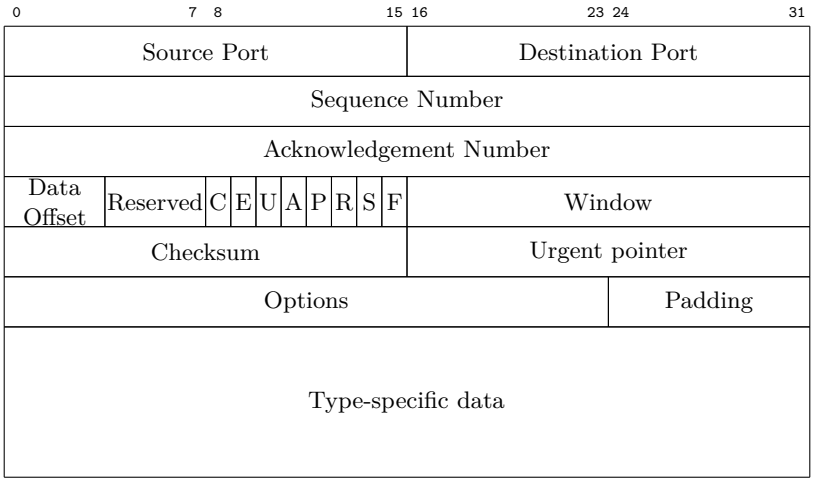
UNUSED Reserved and for future use. Should be unset on transmission and MUST be ignored on receipt.]

A.4 FRAGMENT HEADER

Description of the fields can be found in the section 8



A.5 TCP HEADER FORMAT



B

BENCHMARKING

B.1 COMPRESSION RATIO COMPARISON

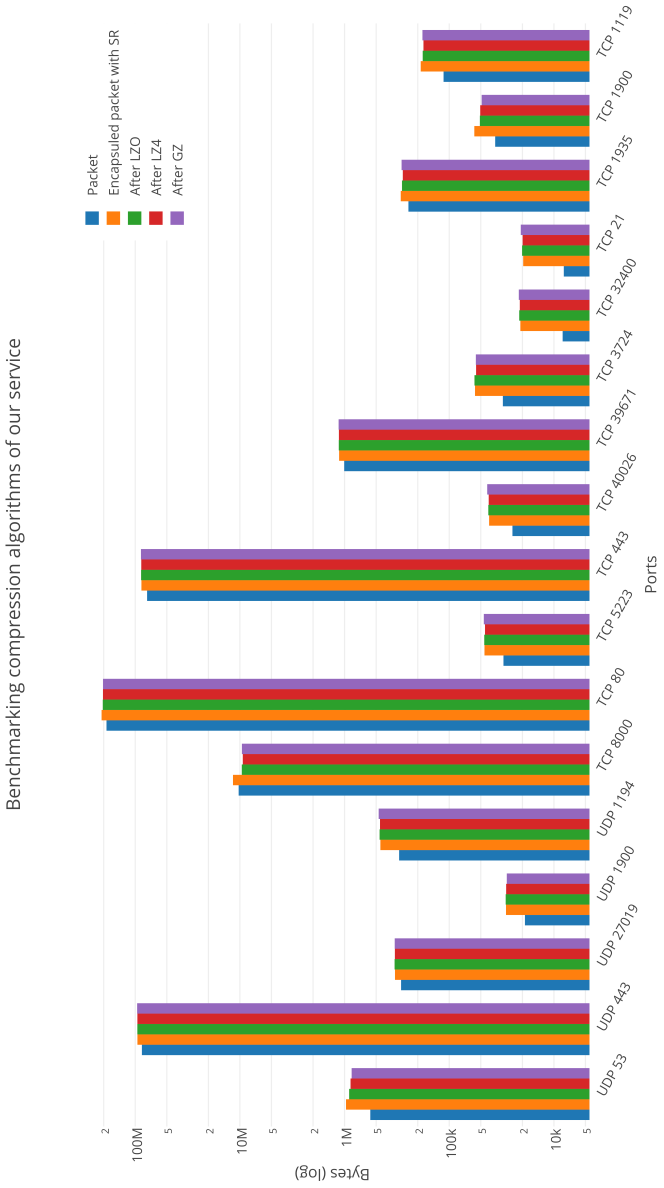


Figure 63: Captured flows compared with SRH overhead and our different kinds of compression

B.2 INLAB SETTINGS

Listing 2: Example of configuration for H5

```
sysctl -w net.core.rmem_default=524287
sysctl -w net.core.wmem_default=524287
sysctl -w net.core.rmem_max=524287
sysctl -w net.core.wmem_max=524287
sysctl -w net.core.optmem_max=524287
sysctl -w net.core.netdev_max_backlog=300000
sysctl -w net.ipv4.tcp_rmem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_wmem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_mem="10000000 10000000 10000000"
sysctl net.ipv6.conf.all.forwarding=1

ethtool -K enp1s0f0 gro off
ethtool -K enp1s0f1 gro off
ethtool -K enp1s0f0 lro off
ethtool -K enp1s0f1 lro off
ethtool -K enp1s0f0 gso off
ethtool -K enp1s0f1 gso off


for i in $(seq 29 36); do echo 6 > /proc/irq/$i/
    smp_affinity_list; done
for i in $(seq 38 45); do echo 7 > /proc/irq/$i/
    smp_affinity_list; done
```

C

OTHERS LISTINGS

C.1 SR0 ATTACKS

Listing 3: Checking if an ISP filters spoofed traffic from its clients [BIONDI, 2007]

```
sr1(IPv6(src=us, dst=tgt)/ \
    IPv6ExtHdrRouting(addresses=[us])/ \
    ICMPv6EchoRequest())
```

Listing 4: Denial Of Service Amplification[BIONDI, 2007]

```
sr1(IPv6(dst=addr2, hlim=255)/ \
    IPv6OptionHeaderRouting(addresses=[addr1, addr2]*43)/ \
    ICMPv6EchoRequest(data="staythere"), verbose=0, timeout
        =80)
```

C.2 BASIC SERVICE

Listing 5: Basic synchrone service with COUNT

```
void count_packet_in(struct seg6_sock *sk __unused, struct
    nlattr **attrs,
         struct nlmsghdr *nlh __unused) {

    static int cnt;
    static int bytes;
    int pkt_len;
    char *pkt_data;

    pkt_len = nla_get_u32(attrs[SEG6_ATTR_PACKET_LEN]);
    pkt_data = nla_data(attrs[SEG6_ATTR_PACKET_DATA]);

    cnt++;
    bytes += pkt_len;
}

int nl_recv_ack(struct nlmem_sock *nlm_sk __unused, struct
    nlmsghdr *hdr __unused,
         void *arg __unused) {

    return NL_SKIP;
}

int main(int ac, char **av) {
```

```
    struct seg6_sock *sk;
    struct in6_addr in6;
    int ret;

    if (ac != 2)
        usage(av[0]);

    inet_pton(AF_INET6, av[1], &in6);

    /*
     * seg6_socket_create(block_size, block_nr)
     * mem usage = block_size * block_nr * 2
     * default settings = 8MB usage for 4K pages
     * increase block_size and not block_nr if needed
     */
    sk = seg6_socket_create(128*getpagesize(), 64);

    ret = asynchrone_service_launch(sk, &in6, &nl_recv_ack,
        &count_packet_in, 0);
    if (ret)
        fprintf(stderr, "seg6_send_and_recv(): %s\n",
            strerror(ret));

    seg6_socket_destroy(sk);

    return 0;
}
```

C.3 CONFIGURATION FILE

Listing 6: Example of yml configuration file

```
name: sfc\_test
root: ~/sr6/

pre:
    - make -C services/compression/  || exit
    - make -C services/count/  || exit
    - ./topologies/clear || exit
    - bash topologies/simple/simple.topo.sh || exit

windows:
    - iperf3:
        layout: main-vertical
        panes:
            - A:
                - ip netns exec A bash
                - ethtool -K A-0 tx off
                - iperf3 -c $C -B $A
            - C:
                - ip netns exec C bash
                - ethtool -K C-0 tx off
                - ethtool -K C-1 tx off
                - iperf3 -s -B $C
    - encap:
```

```
        layout: main-horizontal
        panes:
            - SRencap:
                - ip netns exec B bash
                - ip -6 route del \@C
                - ip -6 route add \@C via C encap seg6 mode
                    encap segs $C,$D,$E,$D
    - services:
        layout: main-horizontal
        panes:
            - count:
                - ip netns exec C bash
                - ./services/count/count $C
            - compression:
                - ip netns exec D bash
                - ip link set dev D-0 mtu 9000
                - ip link set dev D-1 mtu 9000
                - ./services/compression/compression -c --
                    type lz4  $D
            - decompression:
                - ip netns exec E bash
                - ip link set dev E-0 mtu 9000
                - ip link set dev E-1 mtu 9000
                - ./services/compression/compression -d --
                    type lz4  $E
    - dumps:
        layout: main-horizontal
        panes:
            - count:
                - ip netns exec C bash
                - tcpdump -i C-0 -w dump/before-c.pcap
            - compression:
                - ip netns exec D bash
                - tcpdump -i D-0 -w dump/before-d.pcap
            - decompression:
                - ip netns exec E bash
                - tcpdump -i E-0 -w dump/before-d.pcap
```

C.4 SRFT APPENDIX

C.4.1 *Scapy*

After modification of Scapy, it takes only few lines of code to craft an IPv6 packet with a Segment Routing Header.

Listing 7: Forge a IPv6 with Segment Routing Header in scapy

```
# List of segment where the first one is the destination
# and the last one the source
sfc = ['fc00:2:0:1::1', 'fc00:2:0:5::1', 'fc00:2:0:6::1']

ip6 = IPv6(src=self.sfc[-1], dst=self.sfc[0])
sr  = IPv6ExtHdrSegment(addresses=self.sfc[::-1])
```

C.4.2 *Test example*

We shall now describe how a SRFT test case must be written by combining in a python file the generic classes, unittest, scapy and the configuration file.

To write a new test, we must define a few details, such as: the Segment Routing Header, a list of services to reach, a craft function, a matching function, and the list of filters for the sniffer. (Note that the sniffer also has a filter based on the flow label, as explained earlier.)

Listing 8: Example of test write with SRFT

```
class PacketPassTestCase(unittest.TestCase):

    # Define the service function chaining
    services = [helpers.BULK, helpers.TEST]

    # Done before all test
    def setUp(self) :
        self.conf = helpers.get_conf()
        self.sfc = helpers.addr_services(self.conf,
            PacketPassTestCase.services)

    def test_packet_pass(self) :
        conf.iface6="A-0"

        # Generate header
        ip6 = IPv6(src=self.sfc[-1], dst=self.sfc[0])
        sr  = IPv6ExtHdrSegment(addresses=self.sfc[::-1])
        hdr = (ip6/sr)

        # Define matching function
        matching = lambda a, b: tt.Utils.inner_IPv6(a) ==
            tt.Utils.inner_IPv6(b)

        # Define craft function
        def craft(hdr, fl):
            tt.Utils.set_fl(hdr, fl)
            pkt = hdr/IPv6()/ICMPv6EchoRequest()

            pkt = pkt.__class__(str(pkt))

            return ([pkt], {fl: [pkt]})

        # Create test
        sns = tt.Tester(tt.Test(
            ct.Crafter(hdr, craft),
            [sn.filter_ICMP, sn.filter_SEG],
            mt.Matcher(matching))
        )

        # Launch test
        self.assertTrue(sns.perform_test())
```

BIBLIOGRAPHY

[BIONDI, 2007] Philippe BIONDI and Arnaud EBALARD. *IPv6 Routing Header Security*. Apr. 2007. URL: http://www.secdev.org/conf/IPv6_RH_security-csw07.pdf.

[IETF SR 08] Ebben Aries, Stefano Previdi, Clarence Filsfils, Brian Field, Ida Leung, Eric Vyncke, David Lebrun, J. Linkova, and Tomoya Kosugi. *IPv6 Segment Routing Header (SRH)*. Internet-Draft draft-previdi-6man-segment-routing-header-08. Work in Progress. Internet Engineering Task Force, Nov. 2015. 30 pp. URL: https://tools.ietf.org/html/draft-previdi-6man-segment-routing-header-08.

[stEERING] Ying Zhang et al. "StEERING: A software-defined networking for inline service chaining." In: *2013 21st IEEE International Conference on Network Protocols (ICNP)*. 2013, pp. 1–10. DOI: 10.1109/ICNP.2013.6733615.

[SDN] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti. "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks." In: *IEEE Communications Surveys Tutorials* 16.3 (2014), pp. 1617–1634. ISSN: 1553-877X. DOI: 10.1109/SURV.2014.012214.00180.

[OpenFlow] Open Networking Foundation. *OpenFlow*. http://archive.openflow.org/index.php/OpenFloW_v1.1. [Online; accessed 1-June-2016].

[IETF SFC-NSH] Paul Quinn and Uri Elzur. *Network Service Header*. Internet-Draft draft-ietf-sfc-nsh-05. Work in Progress. Internet Engineering Task Force, May 2016. 38 pp. URL: https://tools.ietf.org/html/draft-ietf-sfc-nsh-05.

[RFC 7746] Ron Bonica, Ina Minei, Michael Conn, Dante Pacella, and Luis Tomotaki. *Label Switched Path (LSP) Self-Ping*. RFC 7746. Jan. 2016. DOI: 10.17487/rfc7746. URL: https://rfc-editor.org/rfc/rfc7746.txt.

[RFC 7348] Mallik Mahalingam, T. Sridhar, Mike Bursell, Lawrence Kreeger, Chris Wright, Kenneth Duda, Puneet Agarwal, and Dinesh Dutt. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. Oct. 2015. DOI: 10.17487/rfc7348. URL: https://rfc-editor.org/rfc/rfc7348.txt.

[Far+13] Dino Farinacci, Stanley P. Hanks, David Meyer, and Paul S. Traina. *Generic Routing Encapsulation (GRE)*. RFC 2784. Mar. 2013. DOI: 10.17487/rfc2784. URL: https://rfc-editor.org/rfc/rfc2784.txt.

[opendaylight] Linux foundation. *OpenDaylight*. https://www.opendaylight.org/service-function-chaining-reference-use-case. [Online; accessed 1-June-2016].

[IETF SFC-MPLS] Xiaohu Xu, Zhenbin Li, Himanshu C. Shah, and Luis M. Contreras. *Service Function Chaining Using MPLS-SPRING*. Internet-Draft draft-xu-sfc-using-mpls-spring-05. Work in Progress. Internet Engineering Task Force, Mar. 2016. 7 pp. URL: https://tools.ietf.org/html/draft-xu-sfc-using-mpls-spring-05.

[SRA] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois. "The Segment Routing Architecture." In: *2015 IEEE Global Communications Conference (GLOBECOM)*. 2015, pp. 1–6. DOI: 10.1109/GLOCOM.2015.7417124.

[RFC 2460] S. Deering and R. Hinden. *RFC 2460 Internet Protocol, Version 6 (IPv6) Specification*. Internet Engineering Task Force, Dec. 1998. URL: http://tools.ietf.org/html/rfc2460.

[RFC 1700] J. Reynolds, ed. *Assigned Numbers: RFC 1700 is Replaced by an On-line Database*. United States, 2002.

[RFC 6275] Dr. Dave B. Johnson, Jari Arkko, and Charles E. Perkins. *Mobility Support in IPv6*. RFC 6275. Oct. 2015. DOI: 10.17487/rfc6275. URL: https://rfc-editor.org/rfc/rfc6275.txt.

[RFC 5095] George Neville-Neil, Pekka Savola, and Joe Abley. *Deprecation of Type 0 Routing Headers in IPv6*. RFC 5095. Oct. 2015. DOI: 10.17487/

rfc5095. URL: https://rfc-editor.org/rfc/rfc5095.txt.

[Scapy] Philippe Biondi. *Scapy*. http://www.secdev.org/projects/scapy/. [Online; accessed 1-June-2016].

[RFC 0791] Jon Postel. *Internet Protocol*. STD 5. http://www.rfc-editor.org/rfc/rfc791.txt. RFC Editor, 1981. URL: http://www.rfc-editor.org/rfc/rfc791.txt.

[RFC 791] *Internet Protocol*. Legacy RFC 791. Mar. 2013. DOI: 10.17487/rfc791. URL: https://rfc-editor.org/rfc/rfc791.txt.

[IETF-Reitzel, 2015] Andrea Reitzel. *Deprecation of Source Routing Options in IPv4*. Internet-Draft draft-reitzel-ipv4-source-routing-is-evil-00. Work in Progress. Internet Engineering Task Force, Oct. 2015. 6 pp. URL: https://tools.ietf.org/html/draft-reitzel-ipv4-source-routing-is-evil-00.

[RFC 6274] Fernando Gont. *IETF RFC 6274: Security Assessment of the Internet Protocol Version 4*. Oct. 2015. DOI: 10.17487/rfc6274. URL: https://rfc-editor.org/rfc/rfc6274.txt.

[IETF SR 01] 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-01. Work in Progress. Internet Engineering Task Force, Mar. 2016. 29 pp. URL: https://tools.ietf.org/html/draft-ietf-6man-segment-routing-header-01.

[Seg6ctl] David Lebrun. *Seg6ctl library*. https://github.com/segment-routing/seg6ctl. [Online; accessed 1-June-2016].

[NETLINK] Linux foundation. *Netlink*. http://www.linuxfoundation.org/collaborate/workgroups/networking/netlink. [Online; accessed 1-June-2016].

[LZ77] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression." In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. ISSN: 0018-9448. DOI: 10.1109/TIT.1977.1055714.

[RFC 4364] Yakov Rekhter and Eric Rosen. *BGP/MPLS IP Virtual Private Networks (VPNs)*. RFC 4364. Oct. 2015. DOI: 10.17487/rfc4364. URL: https://rfc-editor.org/rfc/rfc4364.txt.

[QUIC] Janardhan Iyengar, Ian Swett, Ryan Hamilton, and Alyssa Wilk. *QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2*. Internet-Draft draft-tsvwg-quic-protocol-02. Work in Progress. Internet Engineering Task Force, Jan. 2016. 37 pp. URL: https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02.

[CheckGzip] *Monthly trends for GZIP compression usage - December 2015*. URL: http://checkgzipcompression.com/stats/.

[IETF QUIC] Janardhan Iyengar, Ian Swett, Ryan Hamilton, and Alyssa Wilk. *QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2*. Internet-Draft draft-tsvwg-quic-protocol-02. Work in Progress. Internet Engineering Task Force, Jan. 2016. 37 pp. URL: https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02.

[Newsham, 1998] Thomas H. Ptacek and Timothy N. Newsham. *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*. Tech. rep. Secure Networks, Inc.

[Atlasis, 2012] A. Atlasis. *Attacking IPv6 Implementation Using Fragmentation*. Tech. rep. BlackHat Europe 2012.

[RFC 1858] Darren Reed, Paul S. Traina, and Paul Ziemba. *Security Considerations for IP Fragment Filtering*. RFC 1858. Mar. 2013. DOI: 10.17487/rfc1858. URL: https://rfc-editor.org/rfc/rfc1858.txt.

[RFC 6946] Fernando Gont. *Processing of IPv6 Atomic Fragments*. RFC 6946. Oct. 2015. DOI: 10.17487/rfc6946. URL: https://rfc-editor.org/rfc/rfc6946.txt.

[Shankar, 2003] Umesh Shankar and Vern Paxson. "Active Mapping: Resisting NIDS Evasion Without Altering Traffic." In: *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. SP '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 44–. ISBN: 0-7695-1940-7. URL:

http://dl.acm.org/citation.cfm?id=829515.830553.

[nDPI] *nDPI, Open and Extensible LGPLv3 Deep Packet Inspection Library*. URL: http://www.ntop.org/products/deep-packet-inspection/ndpi/.

[Nanonet] David Lebrun. *Nanonet*. https://github.com/segment-routing/nanonet. [Online; accessed 1-June-2016].

[tmuxinator] Pete Doherty Christopher Chow Allen Bargi. *Tmuxinator, tmux session manager*. https://github.com/tmuxinator/tmuxinator. [Online; accessed 1-June-2016].

[iperf3] Jon Dugan et al. *iperf3 - The network bandwidth measurement tool*. https://iperf.fr/. [Online; accessed 1-June-2016].

[tmux] Nicholas Marriot. *Terminal multiplexer*. https://tmux.github.io/. [Online; accessed 1-June-2016].

[cUnit] A. Aksaharan. *cUnit framework*. http://cunit.sourceforge.net/. [Online; accessed 1-June-2016].

[RFC 6437] B. Carpenter J. Rajahalme S. Jiang and S. Amante. *IPv6 Flow Label Specification*. Internet Engineering Task Force. Dec. 2011. URL: http://tools.ietf.org/html/rfc6437.

[Unittest] Steve Purcell. *Python unit testing framework*. http://pyunit.sourceforge.net/pyunit.html. [Online; accessed 1-June-2016].

[wireshark] Gerald Combs and wireshark team. *Network protocol analyze*. https://www.wireshark.org/. [Online; accessed 1-June-2016].

[Microsoft IPv6, 2008] Microsoft Corporation. *Microsoft® Windows Server® 2008 White Paper: Introduction to IP Version 6*. Jan. 2008.

[Protocol Numbers] Internet Assigned Numbers Authority. *Protocol Numbers*. URL: http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**