SAPIENZA
UNIVERSITÀ DI ROMA

Sapienza University of Rome

Department of Information Engineering, Electronics and Telecommunications
PhD in Information and Communication Engineering

Thesis For The Degree Of Doctor Of Philosophy

Investigating Black Holes in Segment Routing Networks: Identification and Detection

Thesis Advisor
PHD Marco POLVERINI

Candidate
Carlo CAMPANILE

Academic Year MMXIX-MMXXII (XXXIV cycle)

All'amico Marco ed
ai miei figli.

Abstract

The new Segment Routing paradigm provides the network operator the possibility of highly increasing network performance exploiting advanced Traffic Engineering features and novel network programmability functions. Anyway, as any new solutions, SRv6 has a side effect: the introduction of unknown service disruption events. Network Black Holes (BHs) are logical failures that create a service disruption for a subset of traffic flows, generally due to device misconfiguration. Detection of a BH is a hard task due to its specific nature: the infrastructure is up and the disconnection affects a limited number of flows. An example of BH is the one caused by the failure of the Path MTU Discovery procedure in IPv6. The Segment Routing (SR) Architecture is an overlay infrastructure that realizes the source routing. SR exploits the connectivity service offered by the underlay IPv6 (SRv6). Thus SR inherits the problems related to BHs affecting IPv6. In SR this problem is even more stressed due to the encapsulation mechanism that is required to enforce the segment lists on packets. Even worse, existing active probing based tools to detect network BHs for IPv6 are not suitable in SR. In this paper we investigate the problem of detecting SR Black Holes in SR domains. First, we provide an experimental demonstration of the creation of an SR Black Holes. Then we show that existing tools based on active probing are not suitable to detect SR BHs. Then, a passive framework named Segment Routing Black Holes Detection (*SR-BHD*) is introduced. *SR-BHD* make use of specific traffic counters available in SR capable nodes to verify the validity of the flow conservation principle on each network element. Experimental evaluation carried out through simulation and emulation shows the effectiveness of *SR-BHD* in detecting the presence of SR BHs. The proposed framework, named *Segment Routing Black Holes Detection (SR-BHD)* uses a passive approach based on the observation of traffic counters available in SR capable nodes [9]. In particular, the main contributions of this thesis are:

- an experimental demonstration of the existence of *SR Black Holes* and of the possible failure detecting them through an active approach;
- the proposition of a passive detection system allowing a reliable identification of a black hole;
- a deep performance evaluation of the proposed method through simulation.
- a validation of the proposed framework over a real testbed.

Keywords: IPV6, SRv6, Segment Routing, Network Black Hole, failure detection, network monitoring.

Contents

List of Figures

List of Tables

Nomenclature

API: Application programming interface is a connection between computers or between computer programs.

Best effort: delivery describes a network service in which the network does not provide any guarantee that data is delivered or that delivery meets any quality of service.

Black hole: refers to a place in the network where incoming or outgoing traffic is silently discarded (or "dropped"), without informing the source that the data did not reach its intended recipient.

Connectionless: It does not include any connection establishment and connection termination.

Control plane: In network routing, the control plane is the part of the router architecture that is concerned with drawing the network topology, or the information in a routing table that defines what to do with incoming packets.

Data plane: The data plane is the part of the software that processes the data requests.

Decapsulation: Decapsulation is the process of opening up encapsulated data that are usually sent in the form of packets over a communication network.

Destination Routing: Destination routing is a sequential pathway that messages must pass through to reach a target destination.

Destination-based routing: In telecommunications, destination routing is a sequential pathway that messages must pass through to reach a target destination.

Encapsulation: In computer networking, encapsulation is a method of designing modular communication protocols in which logically separate functions in the network are abstracted from their underlying structures by inclusion or information hiding within higher-level objects.

IP Header: IP Header is meta information at the beginning of an IP packet.

iPerf: a tool for active measurements of the maximum achievable bandwidth on IP networks.

MPLS: Multiprotocol Label Switching is a routing technique in telecommunications networks that directs data from one node to the next based on short path labels rather than long network addresses, thus avoiding complex lookups in a routing table and speeding traffic flows.

Namespace: a set of signs (names) that are used to identify and refer to objects of various kinds.

Python: an interpreted high-level general-purpose programming language.

Routing: the process of selecting a path for traffic in a network or between or across multiple networks.

Segment Routing (SR): a protocol designed to forward data packets on a network based on source routes.

Source routing: in computer networking, source routing allows a sender of a packet to partially or completely specify the route the packet takes through the network.

Throughput: the rate of production or the rate at which something is processed.

Tracing: involves a specialized use of logging to record information about a program's execution.

Wireshark: a free and open-source packet analyzer.

Acronyms

API Application Programming Interface

BSID Binding Segment IDentifier

DPDK Data Plane Development Kit

FTP File Transfer Protocol

ICMP Internet Control Message Protocol

ICMPv6 Internet Control Message Protocol for IPv6

IETF Internet Engineering Task Force

IGP Segments Interior gateway protocol

INACL Input Access Control List

IPv4 Internet Protocol Version 4

IPv6 Internet Protocol Version 6

IS-IS Intermediate System - Intermediate System

LSRR loose source and record route

MPLS Multiprotocol Label Switching

MSS Maximum Segment Size

MTU Maximum Transmission Unit

NETNS Network Namespace

NFV Network Function Virtualization

OSPF Open Shortest Path First

PL Packetization Layer

PLPMTUD Packetization Layer Path MTU Discovery

PMTUD Path MTU Discovery

SDN Software Defined Networking

SID Segment Identifier

SPRING Source Packet Routing in Networking

SR Segment Routing

SR-BHD Segment Routing Black Holes Detection

SRH Segment Routing Header

SRv6 Segment Routing Version 6

SSRR strict source and record route

TCP TCP (Transmission Control Protocol)

TCP/IP TCP (Transmission Control Protocol) e IP (Internet Protocol)

VETH Virtual Ethernet

VPN virtual private network

VPP Vector Packet Processing

VPWS Virtual Private Wire Service

WDM Wavelength Division Multiplexing

Chapter 1

Introduction to network failure

Network infrastructures are becoming always more complex due to the ever increasing service requests from users [31]: multi layered architecture, including several different technologies and devices and eg. IP/MPLS over *Wavelength Division Multiplexing (WDM)*.

Network Failures (eg. fiber cut, router reboot, etc.) can happen in each layer of the architecture during failures, users can experience QoS degradation and/or service disruption Generally [**?**] devices raise alarms to notify a failure event Alarms coming from all over can make the failure localization harder.

Figure 1.1: Amazon error caused cloud outage
https://www.datacenterknowledge.com/archives/2011/04/29/amazon-networking-error-caused-cloud-outage

There is a broad range of causes of network failures: hardware faults (e.g., device failures, memory errors) and other multiple reasons, for interface errors such as faulty cable installation, faulty optical transceivers, for OS bugs, for a specially crafted *IPv6* packet that was found to crash the device and certain types of *Internet Protocol Version 4 (IPv4)* packets destined to a physical or virtual interface on the device that can caused a memory leak and finally for misconfigurations (e.g., ARP conflict).

During failure periods, the service would be available, but users may experience high latency or packet drops [16] e [33], for instance, due to interface errors *TCP (Transmission Control Protocol) (TCP)* may likely timeout and re-transmit in the slow-start phase thus degrading the service

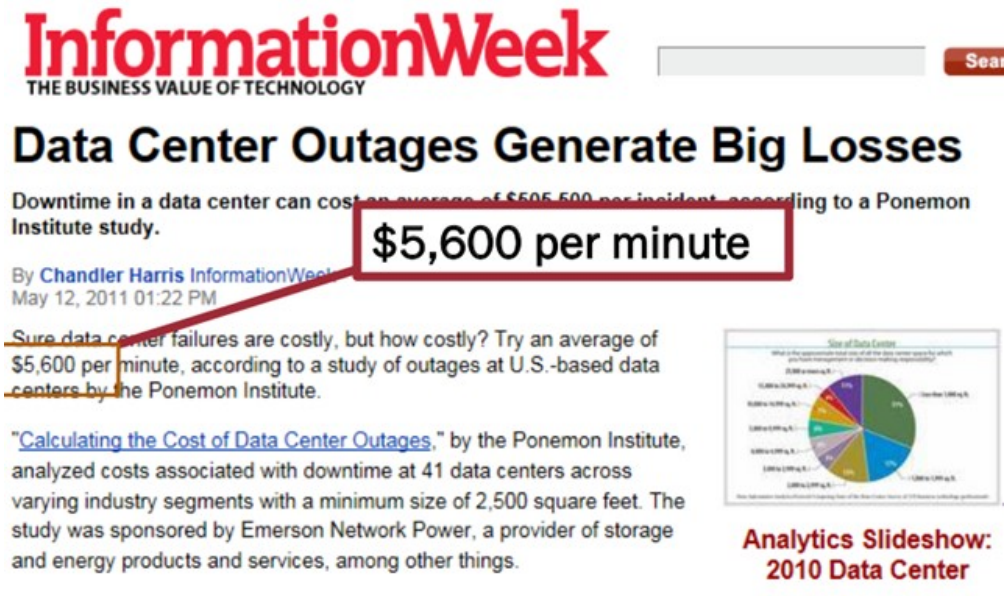


Figure 1.2: Data center outage generate big losses
https://www.informationweek.com/government/data-center-outages-generate-big-losses

performance.

The figure 1.3 shows the histogram of the top-k problems observed from trouble tickets associated with intra (a) and inter (b) datacenter failures;

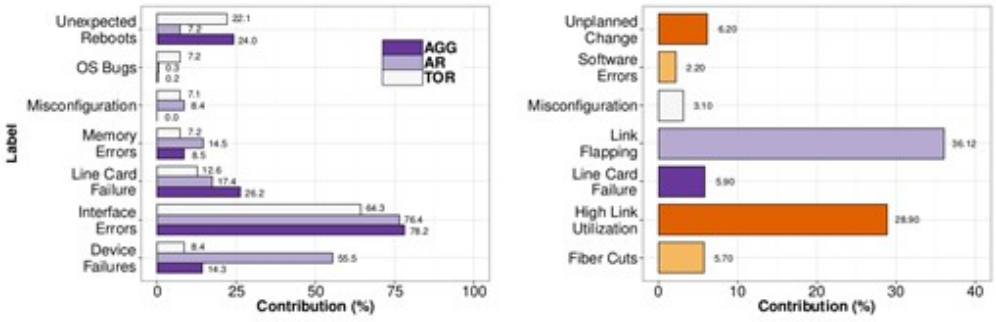


Figure 1.3: Top-k problems observed from trouble tickets for intra and inter datacenter [30]

There is a multitude of devices that can fail in a network [36] e [15]: from a user perspective, a network failure causes a loss of connectivity and when elements in the path between the source and destination hosts is not available. From now on we will look at a network failure as a loss of L3 connectivity recovery from higher layer allowing protection also at lower layers [26] figure 1.4 shows a possible L3 link disruption classification. In the figure 1.4 we have tried to summarize and build a taxonomic classification of the main failures.

L3 link failure is a frequent event into an ISP network and figures 1.6 and 1.8 show that the most of the L3 link failures are transient and single and short-lived 1.7.

IP restoration can be divided into three basic phases:

- Failure detection;
- Failure notification to the control plane and other network entities;

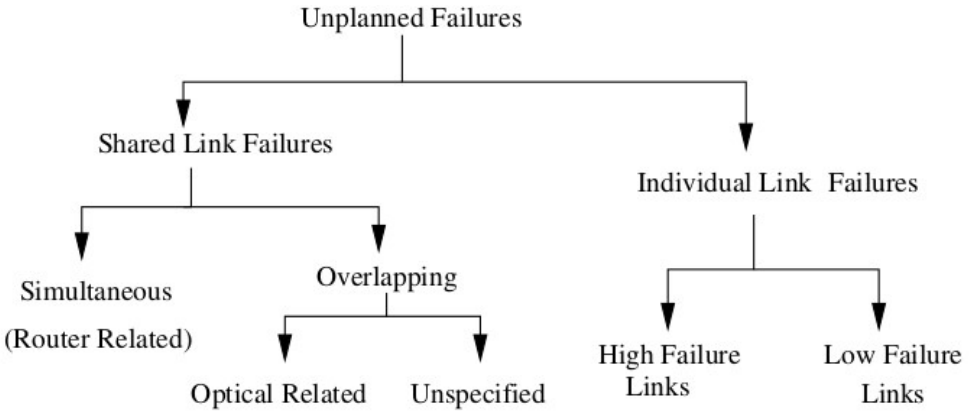


Figure 1.4: Disruption classification

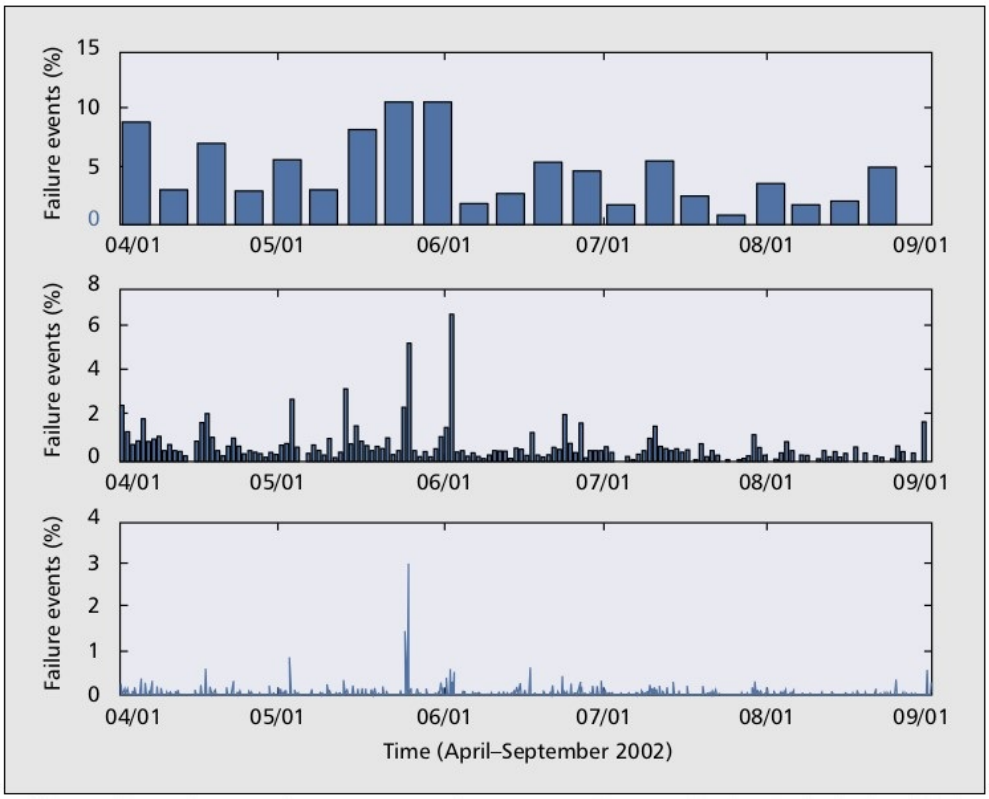


Figure 1.5: Failure notifications over three timescales: weekly (top), daily (middle), and hourly (bottom)

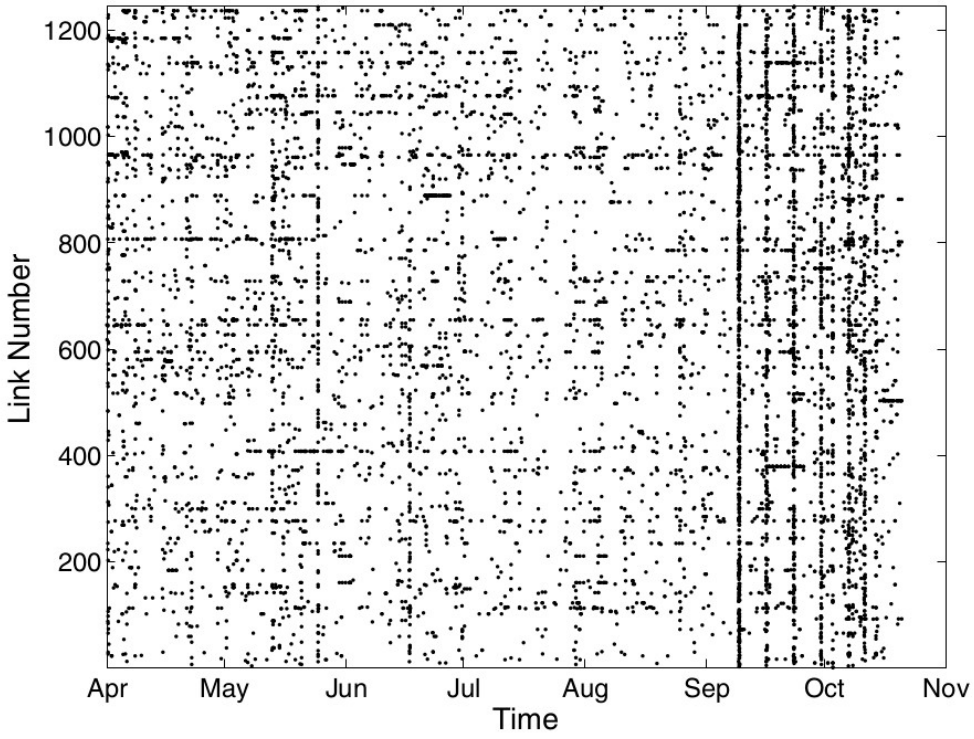


Figure 1.6: L3 failures transient and single

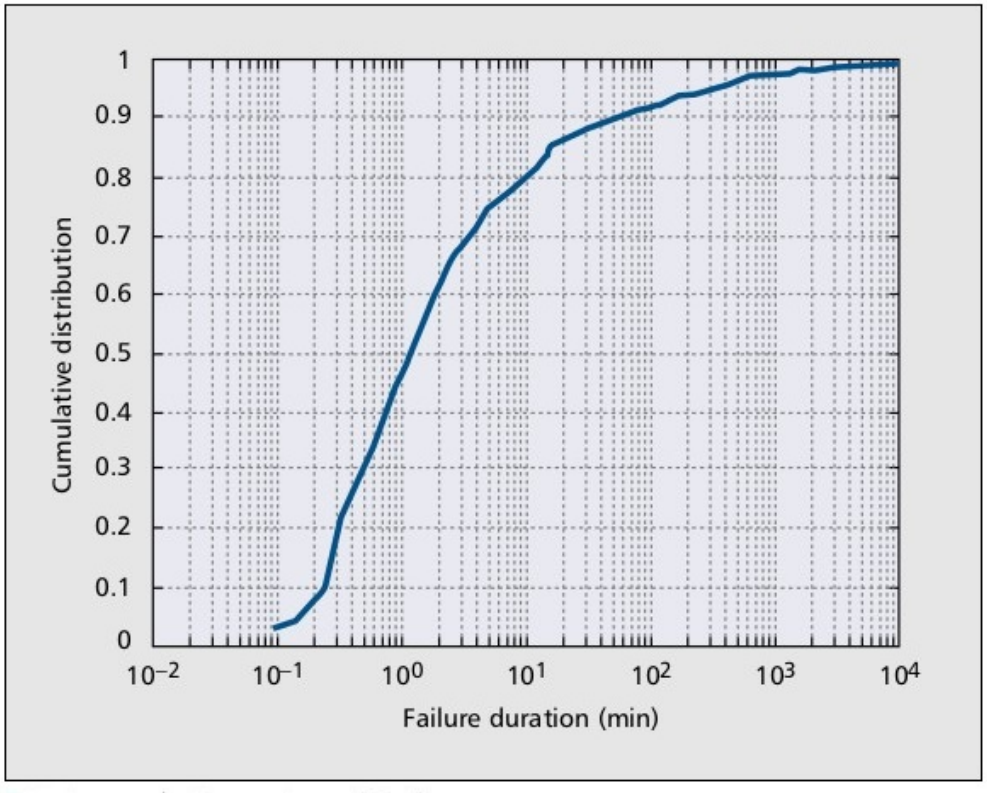


Figure 1.7: Duration of failure events

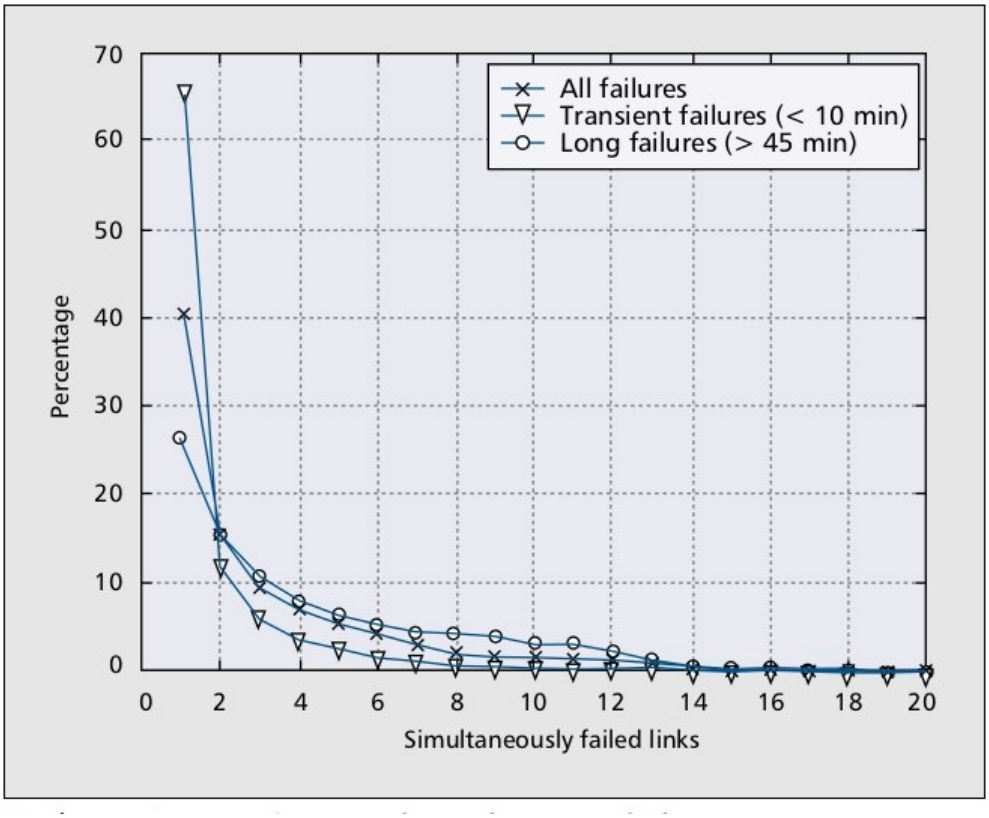


Figure 1.8: Distribution of simultaneous failures

- Failure detour.

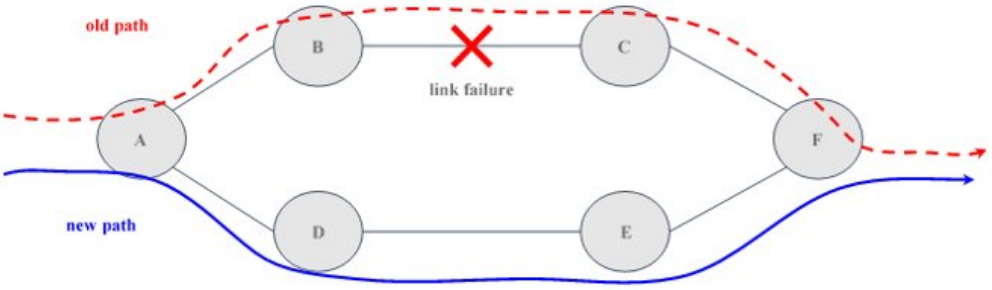


Figure 1.9: Failure detour

Bidirectional Forwarding Detection (BFD) [18] is a network protocol that is used to detect faults between two forwarding engines connected by a link:

- BFD is a simple Hello protocol;
- A pair of systems transmit BFD packets periodically over each path between the two systems.

If a system stops receiving BFD packets for long enough, the link with the neighboring system is assumed to have failed.

Once the failure has been detected, the event has to be notified to the interested parties:

- centralized control plane with *Software Defined Networking (SDN)* Controller;

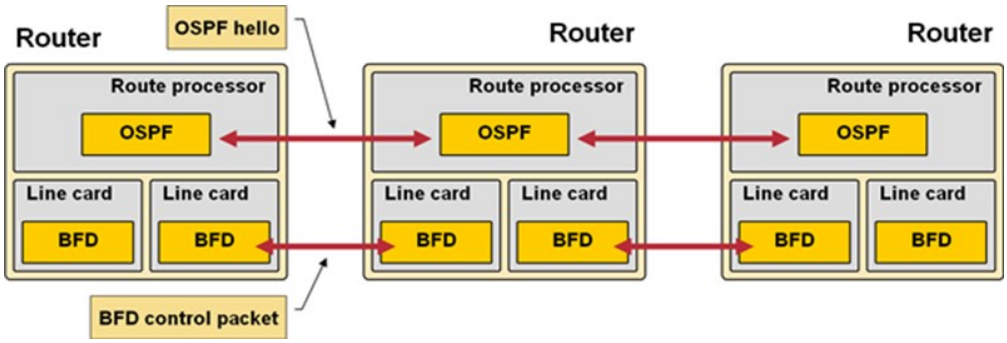


Figure 1.10: Bidirectional Forwarding Detection (BFD) mechanism

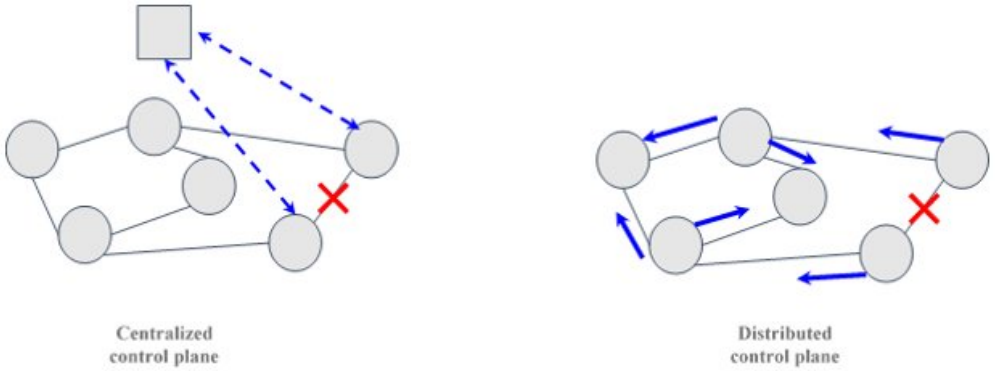


Figure 1.11: Centralized and distributed control plane

- distributed data plane in all the network routers.

The detour operation consists in changing the routing of the traffic flows currently forwarded over the failing device. There are two possibilities:

- protection: a backup path is configured in the network. It's fast (do not requires online configuration) but it's high resources consumer (capacity, memory, etc.);
- restoration: an alternative path is searched once the failure is detected. Low resource utilization is required but the restoration is slow.

Monitoring tools

- allow fast detection of failures;
- inaccurate;
- determine which L3 link has failed;
- hard to discover where the failure occurred (which device);

Data Driven Localization

- require up to several hours to detect a failure accurately;
- allow to exactly assess the failed device and the type of failure;

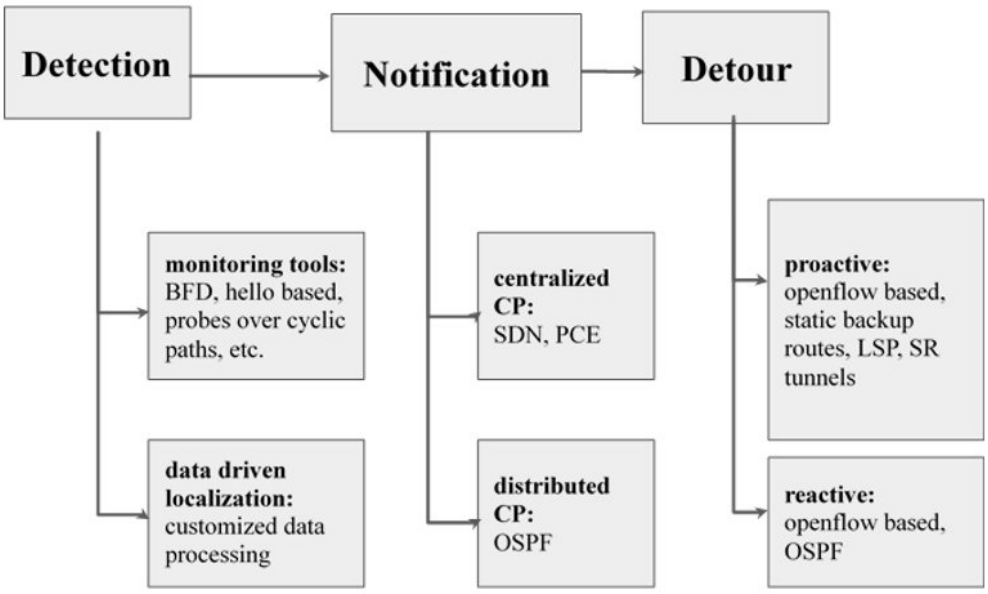


Figure 1.12: IP restoration mechanism

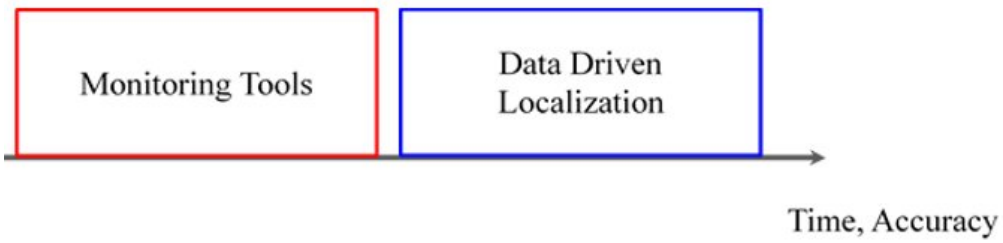


Figure 1.13: Monitoring tools and data driven localization

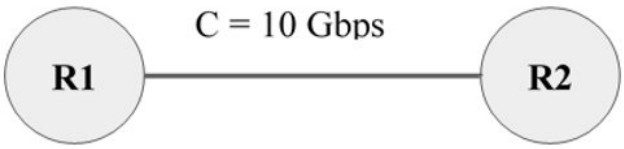


Figure 1.14: Evaluation for real network

Current technologies (BFD + *Segment Routing (SR)*) allow to react to a L3 link failure in 50 ms. With the always increasing data rate, 50 ms [2] [18] can easily turn into a severe packet loss.

Under the hypothesis of packets of length L = 1500 Byte, 50 ms of service disruption turns into 41.7 Kbps loss. In case of proactive configuration of the backup path, most the 50 ms needed to restore the service is due to detection (BFD). As shown in the introduction, L3 link failures exhibit a recurrent pattern: IDEA: if the failures could be predicted, it would be possible to reduce the packet loss PROS: reduce (or avoid) packet loss increase of QoS, CONS: trigger unneeded re-routing possible reduction of QoS.

Chapter 2

Source Routing and Segment Routing

SR architecture is based on the concept of source routing and has interesting scalability properties, as it dramatically reduces the amount of state information to be configured in the core nodes to support complex services. *SR* architecture was first implemented with the *Multiprotocol Label Switching (MPLS)* dataplane and then, quite recently, with the *IPv6* dataplane (*Segment Routing Version 6 (SRv6)*). *SRv6 SR* architecture (*SRv6*) has been extended from the simple steering of packets across nodes to a general network programming approach, making it very suitable for use cases such as Service Function Chaining and *Network Function Virtualization (NFV)*. The new Segment Routing paradigm provides the network operator the possibility of highly increasing network performance exploiting advanced Traffic Engineering features and novel network programmability functions. The network paradigm based on source routing, i.e., the source node decides the path that each packet has to go through. *SRv6* has a side effect: the introduction of unknown service disruption events and incorrect computation of the (*Maximum Transmission Unit (MTU)*) value of an end-to-end path in an *SRv6* network.

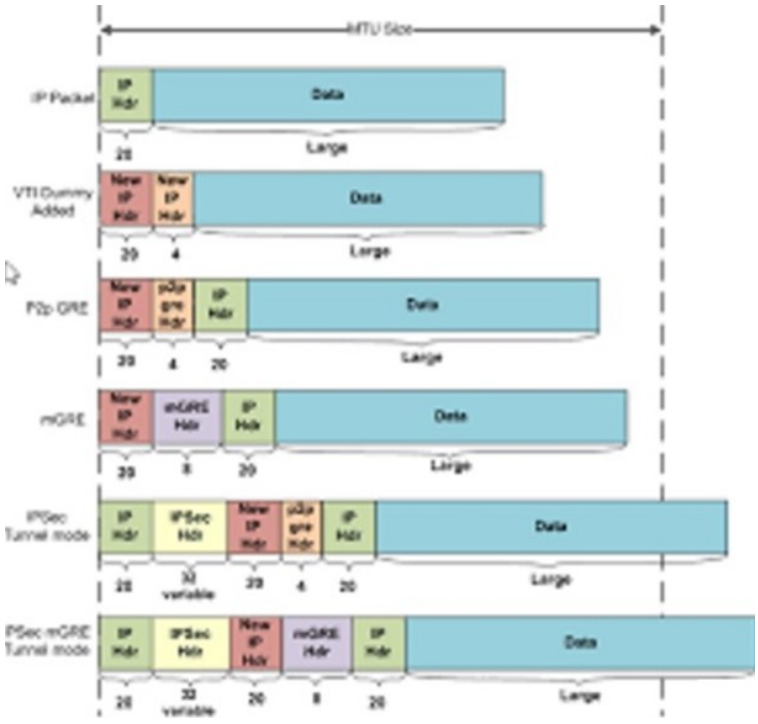


Figure 2.1: Exceeding the maximum transmission unit (MTU)

2.1 The Segment Routing

Architecture Segment Routing is a Cisco technology and is based on the paradigm of Source Routing. By calling intermediate devices, i.e. routers, with the name "nodes", they can have three distinct roles:

- Ingress node;
- Intermediate node;
- Egress node.

Segment Routing involves adding an ordered list of segments in the packet header, called Segment List. The ingress node is so called because it is the node that takes care of inserting this information into the packets and represents the entry point into the network SR. The segments represent instructions and, identified by theirs Segment Identifier (or *SID*), can be of two types:

- Reach node N by following the shortest path or by specifying the nodes to pass through (these will be reached by following the shortest path);
- Apply the S service.

The intermediate node is the node that performs the only routing function. The egress node has the task of eliminating all information related to SR including the Segment List, and sending the packet to its final destination. As for other features and functionality of the Segment Routing, a link failure protection is provided. In fact, if a connection between two nodes is interrupted,SR expects to redirect traffic by following another route, without any traffic loss occurring: it is called fast-reroute solution. In the end, SR supports all kinds of control plane whether it is centralized, distributed or hybrid, while, with reference to the data plane, can be used, among others, *IPv6*.

Segment routing, a form of computer networking, is a modern variant of source routing that is being developed within the *Source Packet Routing in Networking (SPRING)* and *IPv6* working groups of the *Internet Engineering Task Force (IETF)*. In a segment routed network, an ingress node may prepend a header to packets that contain a list of segments, which are instructions that are executed on subsequent nodes in the network. These instructions may be forwarding instructions, such as an instruction to forward a packet to a specific destination or interface. Segment routing works either on top of a *MPLS* network or on an *IPv6* network. In an *MPLS* network, segments are encoded as *MPLS* labels. Under *IPv6*, a new header called a Segment Routing Header (*SRH*) is used. Segments in a *SRH* are encoded in a list of *IPv6* addresses. Segment routing is a method of forwarding packets on the network based on the source routing paradigm. The source chooses a path and encodes it in the packet header as an ordered list of segments. Segments are an identifier for any type of instruction. For example, topology segments identify the next hop toward a destination. Each segment is identified by the segment ID (*SID*) consisting of a flat unsigned 20-bit integer. *Segments Interior gateway protocol (IGP)* distributes two types of segments: prefix segments and adjacency segments. Each router (node) and each link (adjacency) has an associated segment identifier (*SID*).

- A prefix *SID* is associated with an IP prefix. The prefix *SID* is manually configured from the *SPRING* range of labels, and is distributed by *Intermediate System - Intermediate System*

Figure 2.2: *SRH* packet header

(IS-IS) or *Open Shortest Path First (OSPF)*. The prefix segment steers the traffic along the shortest path to its destination. A node *SID* is a special type of prefix *SID* that identifies a specific node. It is configured under the loopback interface with the loopback address of the node as the prefix. A prefix segment is a global segment, so a prefix *SID* is globally unique within the segment routing domain;

- An adjacency segment is identified by a label called an adjacency *SID*, which represents a specific adjacency, such as egress interface, to a neighboring router. The adjacency *SID* is distributed by *IS-IS* or *OSPF*. The adjacency segment steers the traffic to a specific adjacency. An adjacency segment is a local segment, so the adjacency *SID* is locally unique relative to a specific router. By combining prefix (node) and adjacency segment IDs in an ordered list, any path within a network can be constructed. At each hop, the top segment is used to identify the next hop. Segments are stacked in order at About Segment Routing 1 the top of the packet header. When the top segment contains the identity of another node, the receiving node uses equal cost multipaths (ECMP) to move the packet to the next hop. When the identity is that of the receiving node, the node pops the top segment and performs the task required by the next segment. Dataplane Segment routing can be directly applied to the Multiprotocol Label Switching (*MPLS*) architecture with no change in the forwarding plane. A segment is encoded as an *MPLS* label. An ordered list of segments is encoded as a stack of labels. The segment to process is on the top of the stack. The related label is popped from the stack, after the completion of a segment. Services Segment Routing integrates with the rich multi-service capabilities of *MPLS*, including Layer 3 *virtual private network (VPN)* (L3VPN), *Virtual Private Wire Service (VPWS)*, Virtual Private LAN Service (VPLS), and Ethernet *VPN* (EVPN). Segment Routing for Traffic Engineering Segment routing for traffic engineering (SR-TE) takes place through a tunnel between a source and destination pair.

Segment routing for traffic engineering uses the concept of source routing, where the source calculates the path and encodes it in the packet header as a segment. Each segment is an end-to-end path from the source to the destination, and instructs the routers in the provider core network to follow the specified path instead of the shortest path calculated by the *IGP*. The destination is unaware of the presence of the tunnel. We need segment routing for traffic engineering (SR-TE), the network no longer needs to maintain a per-application and per-flow state. Instead, it simply obeys the forwarding instructions provided in the packet. SR-TE utilizes network bandwidth more effectively than traditional *MPLS*-TE networks by using ECMP at every segment level. It uses a single intelligent source and relieves remaining routers from the task of calculating the required path through the network;

- Ready for *SDN*: Segment routing was built for *SDN* and is the foundation for Application Engineered Routing (AER). SR prepares networks for business models, where applications can direct network behavior.

SR provides the right balance between distributed intelligence and centralized optimization and programming:

- Minimal configuration: Segment routing for TE requires minimal configuration on the source router;
- Load balancing: Unlike in RSVP-TE, load balancing for segment routing can take place in the presence of equal cost multiple paths (ECMPs);
- Supports Fast Reroute (FRR): Fast reroute enables the activation of a pre-configured backup path within 50 milliseconds of path failure.

2.2 Segment Routing version 6

Segment Routing can be applied to the protocol *IPv6*, adding a new type of header to packets, called Source Routing Header (or *SRH*). There Segment List, consisting of segments each coded in address *IPv6*, is inserted in the *SRH* where there is also a pointer indicating the next segment to be processed. The ingress node it then runs the encapsulation service by adding the *SRH*, while the egress node performs the decapsulation service by eliminating the *SRH*. The header of a packet that is encapsulated by the ingress node has the structure shown in Figure 2.2. As already mentioned, a segment represents an instruction and indicates the service to be performed on a given node. Consequently, a segment (or *SID*) is made up of 128 bits and has the following structure:

Figure 2.3: Structure of a segment

A Segment List, therefore, it looks like this:

- The Locator is the node to reach e Function is the function in it to perform.

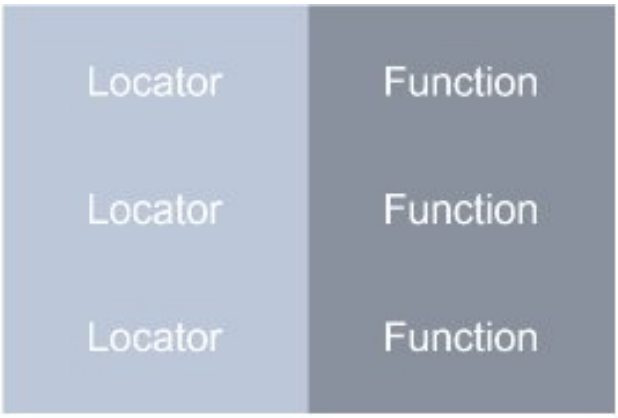


Figure 2.4: Segment List

- Some of the available functions are as follows:

Funzioni associabili ad un SID	Descrizione
End	Creazione di un prefisso SID
End.B6.Encaps	Endpoint legato ad una Policy di incapsulamento
End.DX6	Endpoint con decapsulamento e IPv6 cross-connect

Figure 2.5: Some of the functions identified by a *SID*

- Nodes have the ability to store several Segment lists, each of which is identified by a *Binding Segment IDentifier (BSID)* and it too, in this case, is an address *IPv6*.

A *BSID* is related to at least one Policy: this is entered dynamically or statically by an operator, and determines which one *BSID* must be added in the *SRH* of a package based on the matching of the established parameters. These parameters are defined when the Policy. To illustrate how a network works *SRv6*, an example topology is proposed below (Figure 2.6) and set *SID*, *BSID* And Policy. Let's imagine the following are set SID:

- On node E1 i SID e1 :: 1 with encap function and e1 :: 2 with decap function to host A
- On node E2 i SID e2 :: 1 with encap function and e2 :: 2 with decap function to host B
- On node C1 i SID c1 :: with function end node
- On node C2 i SID c2 :: with function end node

In order for hosts A and B to communicate, the *BSID* could be the following (the *BSID* and between the angle brackets le Segment list):

- On node E1, a1 :: / 128 <c1 ::, e2 :: 2>
- On node E2, a2 :: / 128 <c2 ::, e1 :: 2>

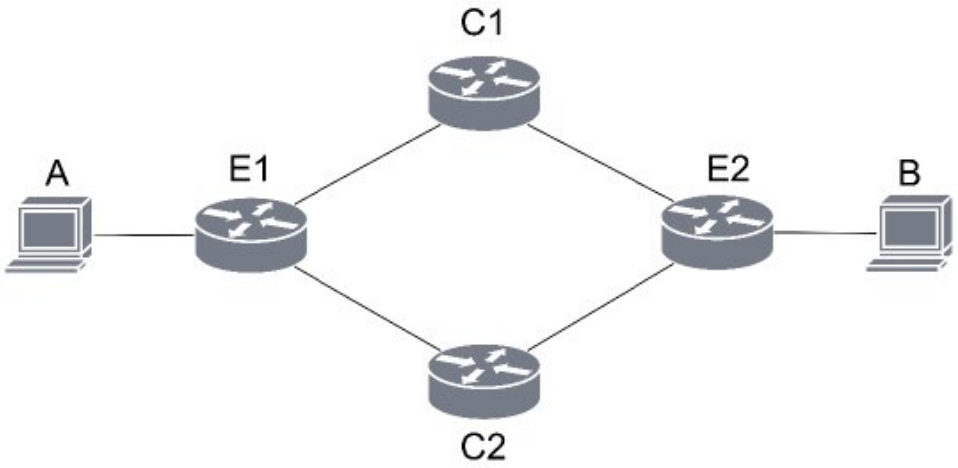


Figure 2.6: Example topology

Since the *BSID* there are two, as many will be policy:

- Policy 1 in node E1 associated with the BSID a1 :: / 128. The parameters to match are: ip-source = A; ip-destination = B;
- Policy 2 in node E2 associated with the BSID a2 :: / 128. The parameters to match are: ip-source = B; ip-destination = A.

That is, Policy 1, installed in E1 and which sees as an action to be taken that of sending traffic following the instructions of the BSID a1 :: / 128, is used for all those packets whose source IP address is the address of host A and the destination IP address the address of host B. Vice versa for Policy 2.

2.3 Vector Packet Processing

This framework is the open source version of Cisco *Vector Packet Processing (VPP)*: one stack high performance packet processing executable on commercial CPUs. Built using the *Data Plane Development Kit (DPDK)*, the *VPP* platform provides the functionality of dataplane in software.

The high performance is guaranteed by the packet processing mode: instead of taking place one at a time, i.e. in a scalar way, *VPP* processes packet vectors, hence its name. However, a further advantage of its use lies in its modularity: packets are processed following a graph that the user can modify or extend by operating on the nodes. Each time a new node is visited, *VPP* sorts the vector of packets by type and before continuing with the processing, divides it into further homogeneous vectors, i.e. containing all the same types of packets: in this way, since not all packets must be processed identically, they follow different paths on the graph. The operation is schematically presented in figure 2.7.

For this reason, three types of nodes are distinguished:

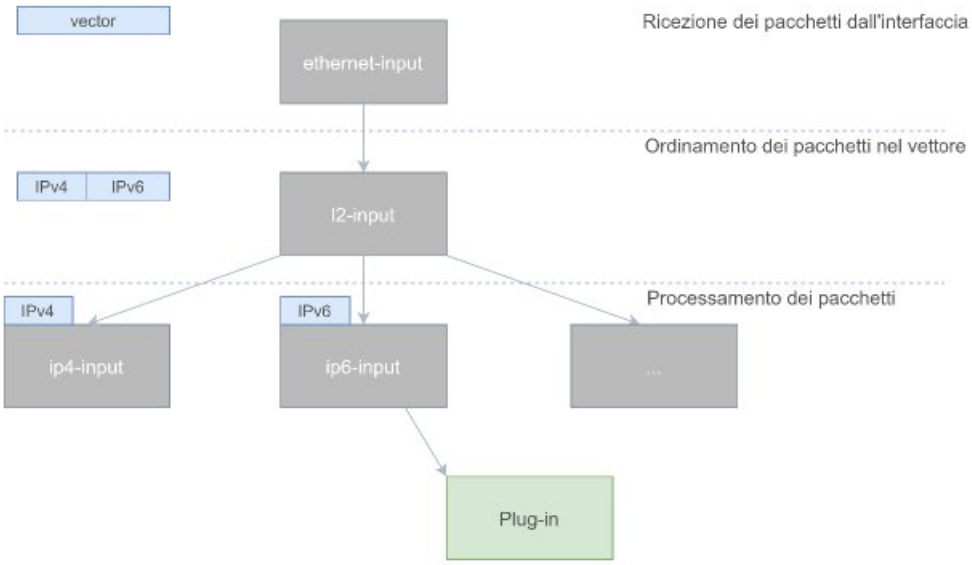


Figure 2.7: Example of a VPP processing graph

- Process node: recall simple functionalities that are part of the main core of *VPP* (for example by calling the libraries l2-input or ip4-lookup)
- Input nodes: manage the initial vector of packets
- Internal nodes: these types of nodes are crossed only under explicit request

For this study, the *VPP* version used is v20.01 and is installed in Ubuntu 20.04 where each instance of it represents a router. This way, once you have configured interfaces and set the parameters for Segment Routing, you will have a functioning network topology.

2.4 Linux namespace

Another technology used is the namespace. The namespace is a feature of the Linux kernel capable of reserving portions of global resources and isolating them, so as to make them for the exclusive use of specific processes. These resources do not affect the global ones and are not visible from other processes. There are several types of namespace depending on the resources they isolate, but the *Network Namespace (NETNS)*, or network namespace allows you to reserve instances of routing tables and network stacks that can be used by an interface assigned to it. Since the physical network interface has already been in use root namespace, you need to instantiate a *Virtual Ethernet (VETH)* pair: two connected virtual network interfaces will be created, one of which must be assigned to namespace of interest and the other remains in the root namespace and can therefore send / receive traffic from physical interfaces or interfaces connected to others *NETNS*. In the image below, the way in which communication takes place between *NETNS* and interfaces in the root namespace. Hereinafter, i *NETNS* will be used to simulate terminal hosts.

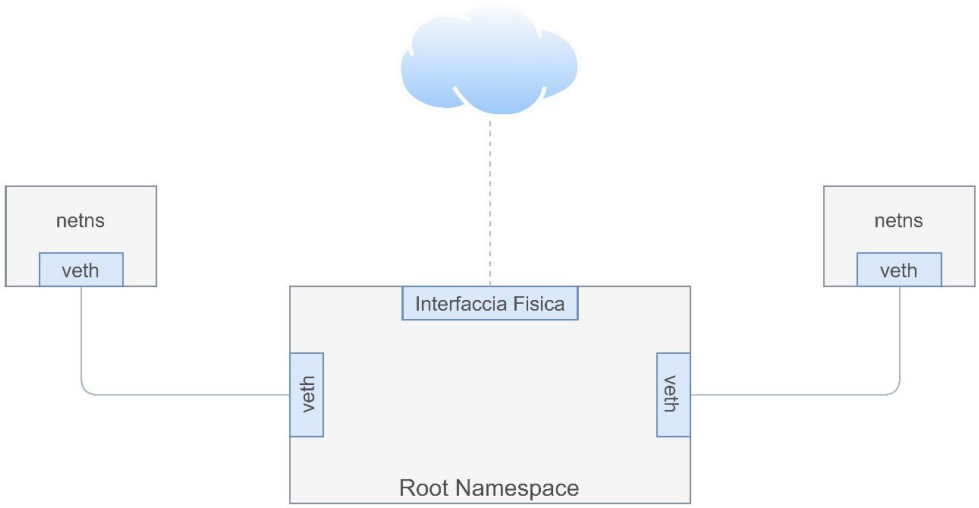


Figure 2.8: Namespace graphic illustration

Chapter 3

MTU Black Holes

3.1 MTU Black Hole and its effects

A Black Hole is defined as an area of the network where inbound and/or outbound traffic is discarded without the source being informed. In this case, the reason for assuming the occurrence of a Black Hole is due to a packet size exceeding with respect to the minimum *MTU* (§4.2.3). This event, referred to as *MTU* dependent *SR Black Hole*, under the SRv6 paradigm, cannot be detected by known monitoring solutions based on active probing: the reason is that in *SRv6* probe packets and user data do not follow the same paths and therefore detection is not possible.

In this work we propose a passive monitoring solution able to exploit the *SRv6* Traffic Counters to detect links where packets are lost due to *MTU* issues. The advent of the *NFV* paradigm and the need to create complex services by configuring the so called Service Function Chains (SFCs), have pushed forward routing technology enhancements. With reference to the control plane functionalities the introduction of the *SDN* has allowed the definition and implementation of efficient and flexible routing algorithms. Considering the data plane, this requirement is efficiently provided by the Segment Routing (SR) architecture [35]. SR leverages the concept of source routing to let the source node to declare the set of instructions (either topological or service based) to apply on each packet. In particular, SR defines a powerful network programming model [10] that offers an unprecedented expressiveness in the definition of network programs to be applied on traffic flows. An instruction, referred to as segment in the SR jargon, allows to specify the type of function to execute (and eventually to pass a set of arguments as input) and the node that has to perform it, also known as locator. A network program in then represented by a segment list, i.e., an ordered list of segments. To reduce the burden of network nodes and make the architecture scalable, network programs are directly inserted in the packet header. In particular, since SR uses the *IPv6* data plane (*SRv6*, [12]), the segment list is included in a new extension header named *SRH*. So, the flexibility of the SR architecture comes at the price of an overhead increase. If, on one hand, this cost is affordable considering the more and more growing of the backbone link capacities, on the other hand longer packets being transported over an *IPv6* data plane causes potential creations of anomalies in the packet forwarding. In fact, it is well known that *IPv6* has problems [4] with the correct handling of the *MTU*, since the fragmentation operation is allowed only at the source node. Communication failures due to silent discard of too big packets are known in *IPv6* to as black holes [24]. The working principle of *SRv6* further stresses the issues related on *MTU* constraint violation

in *IPv6*. As evidence of this fact, in the RFC 8574, that introduces *SRH* [12], suggests to deploy a greater *MTU* value within the SR Domain than at the ingress edges. Furthermore, efforts to reduce the overhead required to enforce a network program on packets have been recently made by defining the concept of micro*SID*s [34]. In a nutshell, a microSID is a special instruction able to encode a whole segment list into a single segment identifier. Nevertheless, while following the recommendation specified in [11] and using the microSIDs, the risk of the creation of a network black hole can be reduced, the problem is still present. There are several factors that contribute to the creation of a black hole. Among those, the most critical one is that network programs can potentially be enforced at every node in the network on the basis of the verification of a logical condition (e.g., re-routing policy to bypass a failed link [22], or redirect traffic in case of Virtual Machine migration [6]). To better explain this concept, let Δ be the difference between *MTU* of the bottleneck link in SR domain and the length a packet that is entering the domain. A black hole can happen if the overhead $\mathcal{O}$ due to the enforcement of network programs on the packet becomes greater than Δ. From this simple example it is evident the contribution of the recommendation specified in [12] and of the use of microSIDs: the first aims at increasing Δ, while the latter tries to reduce $\mathcal{O}$. In [28] we have conjectured the existence of a silent network failure in *SRv6* due to the *MTU* constraint violation, i.e., an *SR Black Hole*. Furthermore, we have discussed through an application example that classical detection tools using active approaches (i.e. the transmission of probes) fail in diagnosing the presence of an *SR Black Hole*. These methods work according to the fate sharing paradigm, i.e., they assume that probe and data packets share the same network "fate". Unfortunately, since SR architecture follows the policy routing approach, there is no guarantee that probes and data packets follow the same path. Consequently active detection tools are not suitable in case of *SR Black Holes*. In this paper we address the problem of detecting *MTU* related black holes in an *SRv6* network. In particular, we propose a passive monitoring framework that is able to accurately detect the presence of *SR Black Holes*, providing as output a short list of suspected link/flow pairs, i.e., the list of flows impacted by black holes and the links causing such black holes.

3.2 Related Works

We provide an overview of the research activities related to network black holes. In particular, we divide the literature in two categories: i) known types of network black holes and existing frameworks for their detection, and ii) performance measurements tools in the context of Segment Routing architecture.

3.2.1 Network Black Holes and Detection Frameworks

As defined in [19], network black holes are silent logical failures, often caused by events such as misconfiguration or software bugs. Among the different causes, the use of overlay architectures seems to be a common accelerator for the creation of black holes. A first example is reported in [8], where different failure modes that lead to the occurrence of a black hole are presented, in the context of an IP over MPLS infrastructure. In this scenario, failures affecting the Label Distribution Protocol (LDP) execution can create a black hole, due to the fact the underlying IGP domain is working correctly, while end to end reachability is compromised.

The present work is focused on black holes occurring in an SRv6 network due to the violation

of the MTU constraint caused by the failure of the Path MTU Discovery (PMTUD). The different types of failure modes for the PMTUD procedure are described in [24]. Among those, the most common one is represented by unresponsive routers, that are configured to not send ICMP Packet Too Big (PTB) messages back to the source node whenever a packet with a length exceeding the MTU is received. Many different detection systems have been proposed to detect the presence of network black holes in different contexts. All of them rely on the active test of the network status through the sending of probes. In the next we describe some of the most relevant detection tools.

In [19] an active probing detection mechanism is defined; it is able to detect network black holes occurring in an IP/MPLS backbone. A full mesh of probes are periodically exchanged among the edge routers. The method is based on the concept of *failure signature*, that represents the set of probes that are lost in case a black hole occurs in a specific link. Then, spatial correlation is exploited to identify a set of suspicious links. In particular, all the links whose failure signature is close to the actual set of failed probes are inserted into an hypothesis set.

One of the most reliable tools to discover PMTUD failures is Scamper [24]. Scamper is a two steps procedure to determine either the largest MTU that can be used on a end to end path, and to discover (in case of a failure) what is the router that is not participating to the PMTUD. Both the phases of Scamper are based on the enforcement of probes along the end to end path to check. These probes consist in a set of UDP segments destined to an unused port, when performing the first step, whereas a set of ICMP Echo messages destined to intermediate routers are used in the second phase.

Netalyzr is presented in [20], it is a network measurement and debugging tool to monitor the Internet. The architecture is provided with a set of pre-installed applets; one of these aims at determining the path MTU toward a destination server. This search is based on a process that emits a set of UDP probes to the target destination.

Ripe Atlas [32] is a worldwide monitoring infrastructure based on the use hardware probes placed in the so called vantage points. In [5] Ripe Atlas has been used to discover path MTU black holes in the Internet, with the specific focus of assessing the main causes and the most affected data plane protocol. The obtained results show that black holes due to failure in the PMTUD procedure affect both the IPv4 and the IPv6 data planes. Specifically, Ripe Atlas is able to detect the main causes and the location of these failures, such as PTB messages and fragmented packets filtered by firewalls.

In this paper we extend the seminal idea described in [28], i.e. a passive approach based on the elaboration of traffic-related data available in SRv6 network devices. This approach has already been successfully applied in the past to identify and detect network anomalies and failures. As an example, in [25] the network tomography is exploited to identify anomalous traffic flows, while [33] that defines a statistical analysis based on Signal Processing techniques and applies it over SNMP MIB data in order to detect different types of network anomalies, such as, file server failure due to abnormal user behavior or protocol implementation errors. Nonetheless, no passive monitoring tool has ever targeted the detection of *SR Black Holes*.

3.2.2 SR Performance Measurement tools

SR architecture provides a set of Operation And Maintenance (OAM) tools that Network Operators can use to measure the performance of their infrastructure, execute troubleshooting operations, and

so on. Here we report few examples. In [14] it is presented a scalable and topology-aware data-plane monitoring system for SR-MPLS. Ping and Traceroute for SR networks are defined in [21]. Bidirectional Forwarding Detection (BFD) to test the aliveness of Segment Routing Policies for Traffic Engineering is presented in [2].

SR capable nodes are able to collect statistics on the traffic by exploiting a set of traffic counters called SR Routing Traffic Counter (SRTCs), allowing to perform traffic measurements at different granularity. By means of SRTCs an SR node is able to collect statistics on the received traffic flows aggregating them according to the active segment. Three different types of SRTCs are used in the proposed framework: i) SR-INT, ii) PSID, and iii) POL. SR-INTs (also known to as link counts) account the traffic at link granularity, i.e., they measure the amount of SR traffic that is sent over a specific link. By means of the PSID counters, an SR capable node can account the amount of received traffic that is directed toward a specific node. Finally, POL counters keep track of the amount of traffic that has been steered through a specific SR policy. These counters are described in detail in [29], where the logical relations between them are captured by a mathematical model. Exploiting this model, the Authors show SRTCs can successfully used to improve the performance of existing algorithms in different networking problems (e.g, Traffic Matrix Assessment, Traffic Anomaly Detection, etc.). The present paper is strongly based on the findings reported in [29].

In [23] an SRv6 Performance Monitoring (SRv6-PM) framework is proposed, allowing to perform a deep performance monitoring on an SRv6 infrastructure. Three main components are defined: i) a set of data plane tools for performing traffic measurements (e.g, packet loss, delay) at line rate, ii) a control plane logic that requires to the network nodes to perform specific measurements (and a southbound interface for the data/control plane interaction), and iii) a Cloud Native Big Data Management system for data storage, processing and visualization. As use case for the validation of SRv6-PM, the fine grained measurement of the packet loss level affecting a single SR flow is considered. To measure the amount of packets that are lost for a specific flow, SRv6-PM exploits SR traffic counters instantiated at the ingress and egress nodes. Specifically, the difference among these two counters represents the overall number of lost packets for the target flow.

SCMon [3] is a network wide monitoring system that allows to check the health status of links. It exploits the source routing and the flexibility achieved by SR to create a set of monitoring cycles where to send probes to the aliveness of each them. By properly designing the different cycles it is possible to precisely localize a failed link. Furthermore, SCMon exploits adjacency SIDs of SR to test the status of IP links composed by bundle of connections at layer 2 (the inability to do that is a major drawback of the classical systems based on Bidirectional Forwarding Detection, BFD).

3.2.3 Segment Routing background

Segment Routing (SR) [13] is a novel network paradigm based on source routing , i.e., the source node decides the path that each packet has to go through. The end to end paths are encoded as an ordered list of instructions, also referred to as segments. Thus the full list of segments is named Segment List (SL). Segments are expressed as labels, named Segment IDentifiers or SIDs. In the case of SRv6, the underlay data plane is based on IPv6 and a SID is an IPv6 address.

In SRv6 packet forwarding works as follows. When the border router receives a packet, it has to steer it over a specific SL. This last is chosen according to a given *SR Policy*. After a packet has been processed according to a matched *SR Policy*, its most external IPv6 header is extended

by the inclusion of a SR Header (SRH). This last containing the SL and a pointer identifying the active segment, i.e. the current SID to be used for packet forwarding. In particular, the active segment is copied in the destination address field of the outer IPv6 header. Transit routers forward incoming packet by inspecting the IPv6 destination address of the outer header. Once the node having the same SID of the active segment is reached, the SID-related function must be applied (many functions can be defined). A common function is the END one, which implies that the active segment has to be updated, thus the pointer moves to the next SID in SL. A further action that can be performed is the enforcement of a SL by inserting a new policy. Finally, before the packet leaves the SR domain, the SRH has to be removed.

3.3 MTU Black Hole and SR Header

The traditional paradigm used in telecommunication networks is based on sending packets on a specific path, using the destination IP address as the only discriminating factor. In fact, the routers, which carry out the routing operation, when they process the packets consult their routing table: a set of records which, based on the destination address of the packet, communicates the output interface that allows you to reach the desired host. The method by which packets are placed on a specific path based on the destination IP address is called Destination Routing. However, for various reasons, it is useful for the source to indicate the path that the data flow must follow, using the other fields of the packet header as additional information. The normal routing tables are replaced with other data structures more suitable for this purpose or a solution Software Defined Networking (*SDN*) is used: in fact the matching, that is the correspondence between the fields of the packet and those of the data structure of the control plane of the router, this time must take place on several parameters and the entire path of the packets must be specified in advance. This paradigm is called Source Routing. In computer networking, source routing, also called path addressing, allows a sender of a packet to partially or completely specify the route the packet takes through the network. In contrast, in conventional routing, routers in the network determine the path incrementally based on the packet's destination. Another routing alternative, label switching, is used in connection-oriented networks such as X.25, Frame Relay, Asynchronous Transfer Mode and Multiprotocol Label Switching. Source routing allows easier troubleshooting, improved traceroute, and enables a node to discover all the possible routes to a host. It does not allow a source to directly manage network performance by forcing packets to travel over one path to prevent congestion on another. In the Internet Protocol, two header options are available which are rarely used: *strict source and record route (SSRR)* and *loose source and record route (LSRR)*. Because of security concerns, packets marked *LSRR* are frequently blocked on the Internet. If not blocked, *LSRR* can allow an attacker to spoof an address but still successfully receive response packets by forcing return traffic for spoofed packets to return through the attacker's device. In *IPv6*, two forms of source routing have been developed. The first approach was the Type 0 Routing header. This routing header was designed to support the same use cases as the *IPv4* header options. Unfortunately there were several significant attacks against this routing header and its utilisation was deprecated. A more secure form of source routing is being developed within the *IETF* to support the *IPv6* version of Segment Routing. Software-defined networking can also be enhanced when source routing is used in the forwarding plane. Studies have shown significant improvements in convergence times as a result of the reduced state that must be distributed by the controller into the network.

It is important to underline that in these types of Black Holes not all traffic is discarded, but only that which exceeds the minimum *MTU* in size. Let's consider the following network topology:

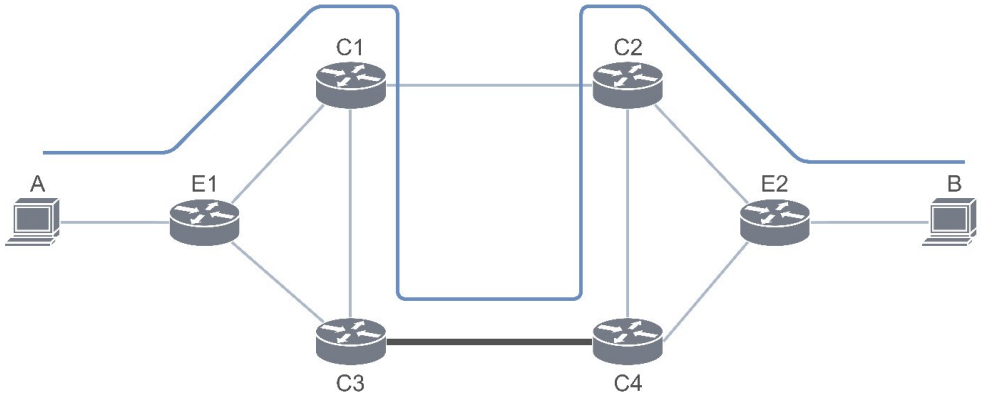


Figure 3.1: SRv6 network topology

Let's imagine that host A sends a large file to host B, following the blue path in the figure and using the protocol *File Transfer Protocol (FTP)* (in turn based on the protocol *TCP*) in the network *SRv6*. Handshake messages will be received and the connection will be established, however there will be problems sending the file: host A and host B, before starting the connection, they ran the algorithm *Path MTU Discovery (PMTUD)* or *Packetization Layer Path MTU Discovery (PLPMTUD)* (§4.2.5) to identify the minimum *MTU* along the path, which in the figure we assume is between nodes C3 and C4. However, for a reason related to *SRv6* and which will be analyzed later, the real amount of data that can be transmitted is less than that allowed by the calculated *MTU*. Therefore, host A in fragmenting the packets will use the *MTU* found as a reference value and for this reason when it tries to send the fragments, they will be discarded before being transmitted in the link highlighted in the figure. Routers, for security reasons, are usually not enabled to send and / or receive packets *ICMP* therefore, node C3 will discard the packets without notifying the source. Even if this does not happen, it would be likely that the source host is protected by a firewall which, for the same reason as the routers, blocks traffic *ICMP*. The consequences, since you are using a protocol based on *TCP*, are, in addition to the obvious total loss of data, also an increase in network congestion. Indeed, *TCP* provides for the postponement of packages for which no acknowledgment has been received. However, the biggest pitfalls that the *MTU* Black Hole would reserve, is in its detection:

- It is very difficult for it to be detected a priori.
- Its presence can be guessed once the traffic loss has been registered.
- Even if traffic loss is detected, as shown above it is possible that it is not total or that only part of the packets sent are lost. This can lead to researching the causes of the problem elsewhere.

3.4 Causes of MTU Black Hole

In §EncapIP-4.2.2 it was pointed out that in the C3-C4 link node C3 will discard packets with a size greater than that allowed by the link. This is because the network protocol in use is the *IPv6* which,

as reported in §4.2.5, does not allow further fragmentation of packets from intermediate devices. Therefore, the upstream source host must perform a correct fragmentation, that is, respecting the limitations imposed by the minimum *MTU* of the path. However, before instantiating a connection with host B, host A ran the algorithm *PLPMTUD*, so it is already aware of the minimum *MTU*. So what is the reason why, despite the fragmentation performed by the source host, the packets exceed the minimum *MTU*. The answer is to be found in Segment Routing. Segment Routing provides that the ingress node add the *SRH* (§2.2). The size introduced by this header is variable and depends on the length of the Segment List:

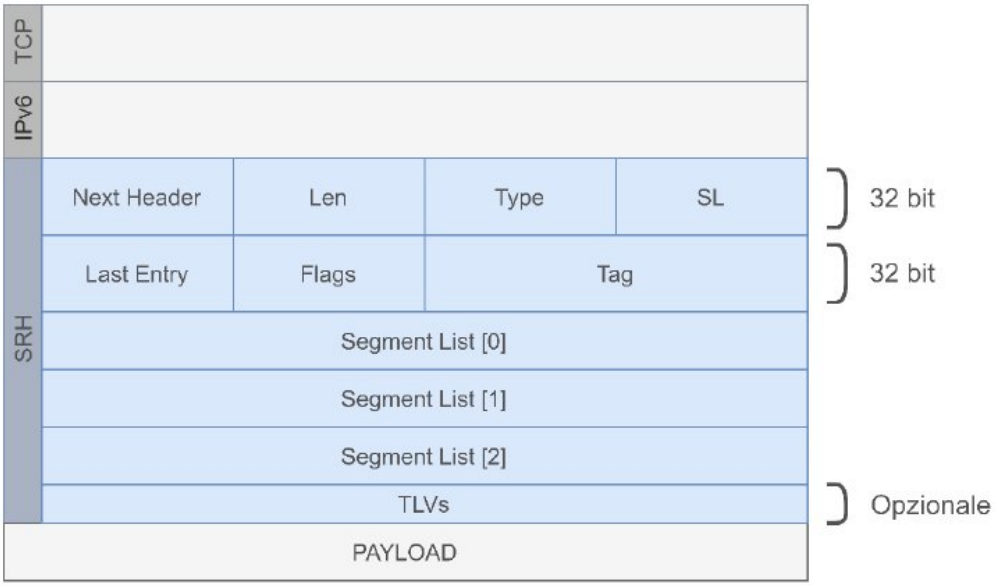


Figure 3.2: *SRH* dimensions

The minimum size introduced is the case where the Segment List contains two suns SID and, remembering that each SID being an address *IPv6* it has a length of 16 bytes, the increase in size amounts to 80 bytes. In general:

$$40bytes + 8bytes + 16 * nbytes \tag{3.1}$$

Where with 40 bytes the header is taken into account *IPv6*, dataplane used by SR. This increase in size cannot be predicted or known a priori by the source host, as it does not have a topological view of the network. Furthermore, the *SRv6* policy and the path to follow in order to reach the destination, are not elements of competence of the source host and of the destination host, but it is of the ingress node. The ingress node in phase of encapsulation must take into account the *MTU* of the domain SR and on the basis of this decide the most suitable path and such as to avoid exceeding this limit. However, network architecture environments *SRv6* are highly dynamic and thank to encapsulation, the path from one node to another can undergo changes to circumvent for example, a link failure. In fact, an intermediate node can itself become an encapsulator node and therefore use a recovery policy (§2.1), as shown in Figure 3.4.

This constitutes a further cause of the formation of *MTU* Black Hole: the input node cannot know in advance whether a link failure will occur with the consequent adoption of a recovery policy which, by providing for a further encapsulation, causes an uncontrolled increase in the size

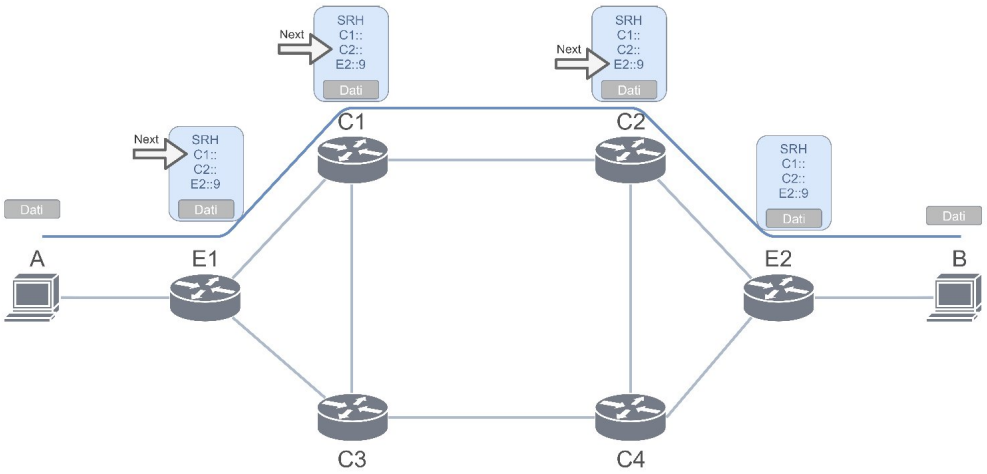


Figure 3.3: Example of Encapsulation with *SRH*

of the packet. If an ingress node must take into account the *MTU* minimum, due to repeated and unpredictable encapsulations *SRH* additives can cause an overrun of the *MTU* and consequent dropping in the intermediate nodes.

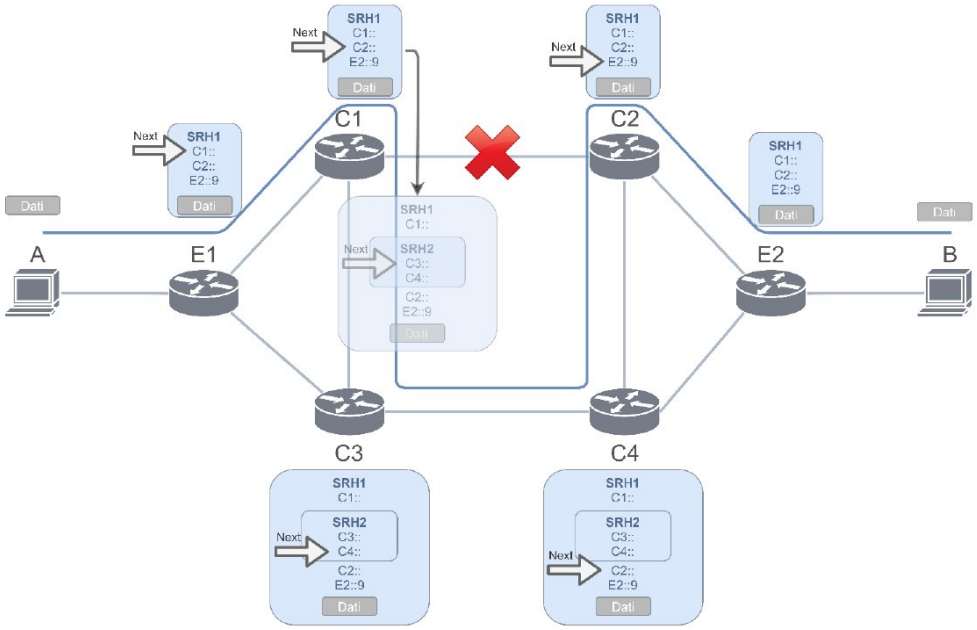


Figure 3.4: Repeated encapsulation following a link fail

In this scenario, host A sends packets to host B using a policy that provides for transit for nodes E1, C1, C2, E2. During communication, a link fail in C1-C2. C1, detecting the disservice, applies the recovery policy. The latter provides for a further encapsulation that makes the traffic pass through nodes C3 and C4. Therefore, the bytes of *SRH*2 of node C1 must be added to the bytes added by *SRH* of node E1, for a total indicated in figure 3.5:

In this case n is equal to 5, so 176 bytes are added. Now we assume that the *MTU* between connection C3-C4 is less than all the others and therefore constitutes the Minimum *MTU*. It may happen that at the first encapsulation, i.e. in node E1, the addition of the *SRH* still respects the minimum *MTU*, but at the second encapsulation, i.e. in C1, the addition of an additional *SRH*

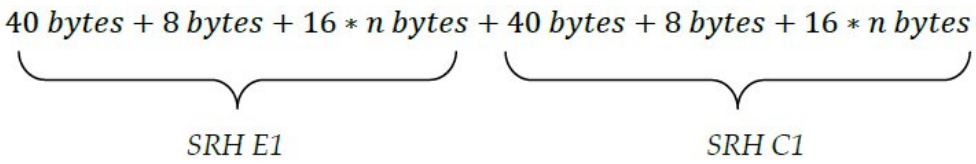


Figure 3.5: Repeated encapsulation following a link failure

satisfies the *MTU* of link C1-C3, but not the one between C3-C4. In this case, C3 does the drop packages and the source host will not be notified. We are therefore in the presence of a *MTU* Black Hole. In both figures 3.3 and 3.4 the same path is assumed to be valid also in the direction of host B host A. A further similar theoretical analysis is applied to a scenario experimentally replicated in §2.3.5.3.

3.5 Experimentation environment

To show the formation of the Black Hole, we proceed to set up an environment in which the network topology of Figure 3.6 is proposed and then various data flows are started between host A and host B.

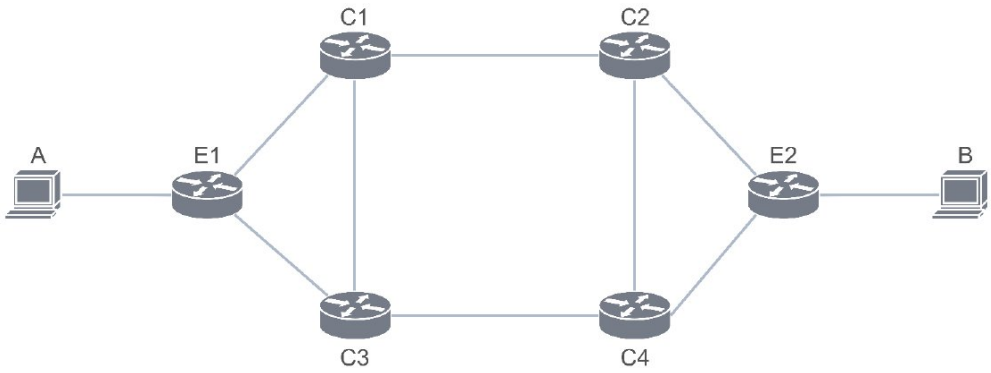


Figure 3.6: Network topology

The causes that lead to the formation of the Black Hole:

- Use of *IPv6* (in particular, the lack of support for fragmentation in intermediate nodes)
- Use of *SRv6* (in particular, the increase in size due to the encapsulation)
- Use of firewalls in terminal devices and / or blocking of *ICMP* traffic in nodes

The operating system in which the experimentation takes place is Ubuntu 20.04. As already mentioned in §2.4, the Linux kernel functionality will be used to simulate terminal hosts *NETNS* (network namespace), while the platform will be used to emulate the nodes *VPP* version 20.01 (§2.3), whose installation is described in Appendix A.

3.5.1 File structure

For the launch of *VPP* instances, installation of policies, start of traffic flows and more, a script has been created that is able to recall and execute files that allow these actions. The script was made

in the language Python, and the source code is shown in Appendix §A.4 The recalled files are with extension .sh, that is, content with language commands bash

- 0-0main, 0-1-linking-vpp-instances, 0-2-creating-hosts, 0-3-setting-SRv6, allow the creation of the topology
- tcp, runs a client on host A and a server on host B and initiates a *TCP* communication
- 2-1-ping, allows an *ICMP* message to be sent from host A to host B
- enable-flow, calls the 2-1-ping and tcp-client files and starts two types of traffic: *ICMP* and *TCP*.
- 1-1-enable-rec-policy, enable the recovery policy in C1 and C2
- 1-2-disable-rec-policy, disable the recovery policy in C1 and C2
- 6-MTU-Path-Discovery, executes the algorithm *PLPMTUD*
- 9-exit-kill-VPP, terminate all instances of VPP.

All the files are shown in the appendices A from §A.8 to §A.14.

3.5.2 Network topology set-up

The commands executed by the files proposed in §3.5.1 and which proceed to the correct network setting are described below.

3.5.3 IPv6 address assignment

The interfaces of each element of the network are assigned only *IPv6* addresses assigned in the following way:

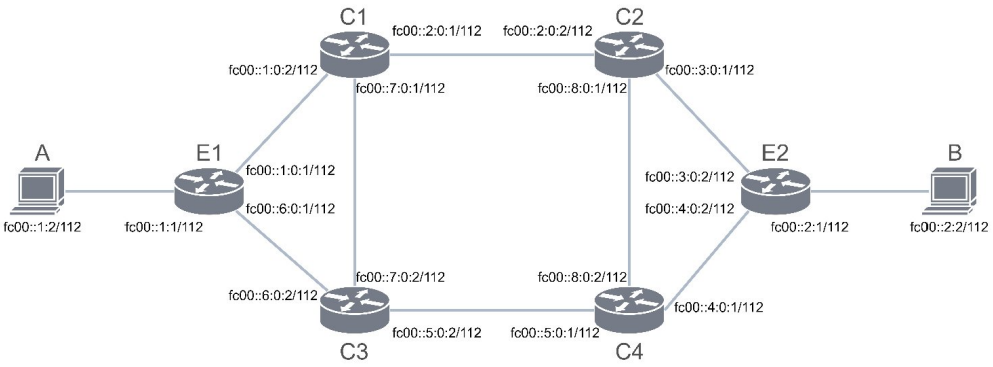


Figure 3.7: IPv6 addresses of the network interfaces

Each link has an *MTU* of 1500 bytes, with the exception of the host A - E1 and E2 - host B links which have an *MTU* of 1300 bytes and of the C3-C4 link which has an *MTU* of 1400 bytes in which a *MTU* Black Hole.

In fact, in networks in which the *SRv6* architecture is applied, it is good practice to have an *MTU* lower than the internal ones in the external connections, i.e. those that connect the terminal

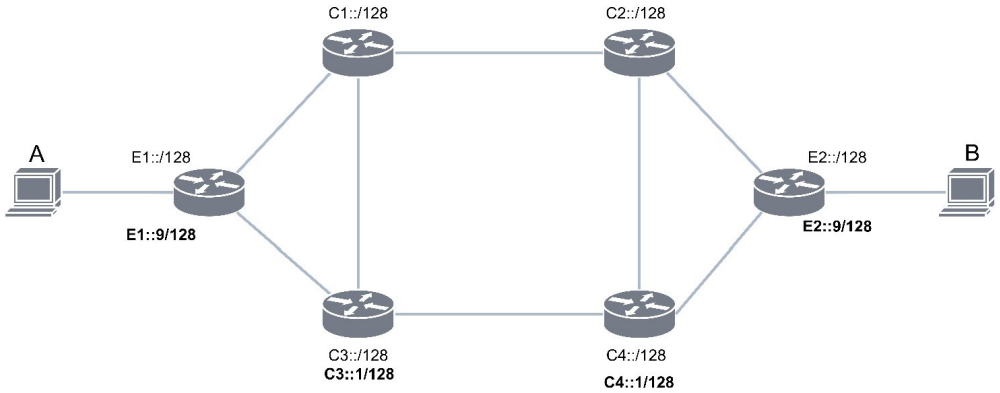


Figure 3.8: BSID addresses

hosts to the ingress / egress node, in order to avoid exceeding this last. THE *BSID* are assigned as shown in Figure 3.8 with the particularity that ingress and egress node (E1 and E2) and nodes C3 and C4 have two of them: one with the function of end point (indicates that the node is passing through and sends the packet to *BSID* next) and the other with the function of decapsulation (which, specifying the interface to which decapsulate, indicates the arrival node). All other nodes have only one *BSID* with the function of end point.

3.5.4 Policies and Routing Tables

In order for host A to reach host B, Policy 1 shown in red in Figure 3.9 is used link fail in C1-C2, to reach C2, Policy 2, indicated in blue in Figure 3.9, is also used together with Policy 1.

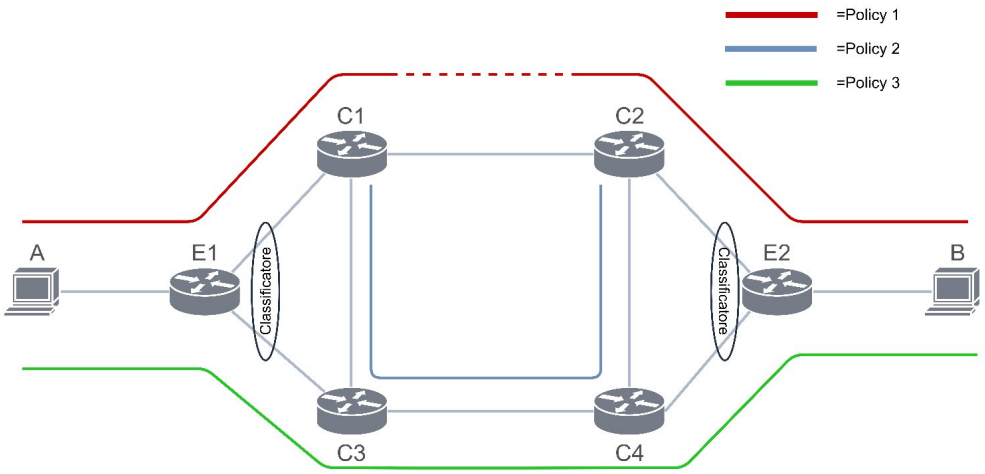


Figure 3.9: Policy

Both Policies are used for *TCP* traffic, while Policy 3, shown in green in Figure 3.9, is used for *ICMP* traffic. This traffic subdivision was possible by means of a traffic classifier in node E1 and E2: the processing graph of *VPP* was changed with the help of Python *Application Programming Interface (API)* and running the script in the Appendix §A.16. Indeed, after visiting the node ip6-inacl *Input Access Control List (INACL)* that separates one packet stream from another based on specific characteristics, the node has been added sr-pl-rewrite-encaps which encapsulates the packets in the correct Policy.

The following are listed *BSID* of each Policy and the corresponding expected path:

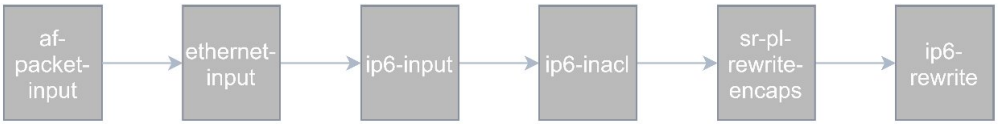


Figure 3.10: VPP Processing Graph

Policy 1 (BSID a1::)	C1::,c2::,e2::9 ; c2::,c1::,e1::9
Policy 2 (BSID a2::)	C3::,c4::1 ; c4::, c3::1
Policy 3 (BSID a3::)	C3::, c4::, e2::9 ; c4::,c3::,e1::9

Table 3.1: BSID

The outbound path, i.e. with direction host A-host B, is separated by a semicolon from the return path, i.e. with direction host B-host A. The routing tables of each node are listed below:

Destination	**Via**
fc00::1:0:2/112	fc00::1:0:1
fc00::6:0:2/112	fc00::6:0:1
fc00::2:2/112	fc00::6:0:2
fc00::1:2/112	fc00::1:1
c1::/128	fc00::1:0:2

Table 3.2: Routing Table E1

Destination	**Via**
fc00::1:0:2/112	fc00::1:0:2
fc00::1:2/112	fc00::1:0:1
fc00::2:0:2/112	fc00::2:0:1
fc00::7:0:2/112	fc00::7:0:1
c3::/128	fc00::7:0:2
e1::9/128	fc00::1:0:1

Table 3.3: Routing Table C1

Destination	**Via**
fc00::3:0:2/112	fc00::3:0:2
fc00::4:0:1/112	fc00::4:0:1
fc00::2:2/112	fc00::2:1
fc00::1:2/112	fc00::4:0:1
c2::/128	fc00::3:0:2
c4::/128	fc00::4:0:1

Table 3.4: Routing Table E2

However, the Routing tables undergo the following variations (the records that are deleted are shown in red) in order to simulate the link fail and to allow the Recovery Policy to function correctly:

- Installation of the Recovery Policy
- Uninstalling the Recovery Policy and restoring the C1 - C2 link

Destination	Via
fc00::2:0:1/112	fc00::2:0:2
fc00::3:0:1/112	fc00::3:0:1
fc00::2:2/112	fc00::3:0:2
fc00::8:0:2/112	fc00::8:0:1
c4::/128	fc00::8:0:2
e2::9/128	fc00::3:0:1

Table 3.5: Routing Table C2

Destination	Via
fc00::6:0:1/112	fc00::6:0:2
fc00::7:0:1/112	fc00::7:0:2
fc00::5:0:1/112	fc00::5:0:2
fc00::2:2/112	fc00::5:0:1
fc00::1:2/112	fc00::6:0:1
c4::/128	fc00::5:0:1
c1::/128	fc00::7:0:1

Table 3.6: Routing Table C3

Destination	Via
fc00::5:0:2/112	fc00::5:0:1
fc00::8:0:1/112	fc00::8:0:2
fc00::4:0:1/112	fc00::4:0:1
fc00::2:2/112	fc00::4:0:2
fc00::1:2/112	fc00::5:0:2
c2::/128	fc00::8:0:1
c3::/128	fc00::5:0:2

Table 3.7: Routing Table C4

Destination	Via
c2::/128	fc00::2:0:2
c2::/128	a2::

Table 3.8: Variation of Routing Table C1

Destination	Via
c2::/128	fc00::2:0:1
c2::/128	a2::

Table 3.9: Variation of Routing Table C2

Destination	Via
fc00::7:0:1/112	fc00::7:0:2
C4::1/128	fc00::5:0:1

Table 3.10: Variation of Routing Table C3

Destination	Via
fc00::8:0:1/112	fc00::8:0:2
C3::1/128	fc00::5:0:2

Table 3.11: Variation of Routing Table C4

Destination	Via
c2::/128	a2::
c2::/128	fc00::2:0:2

Table 3.12: Variation of Routing Table C1

Destination	Via
c1::/128	a2::
c1::/128	fc00::2:0:1

Table 3.13: Variation of Routing Table C2

Destination	Via
c4::1/128	fc00::5:0:1
fc00::7:0:1/112	fc00::7:0:1

Table 3.14: Variation of Routing Table C3

Destination	Via
c3::1/128	fc00::5:0:2
fc00::8:0:1/112	fc00::8:0:1

Table 3.15: Variation of Routing Table C4

3.5.5 Policy Analysis

This paragraph proposes an analysis of the dynamics of the Policy, paying attention to the size of the packages, their increase and allowed size.

3.5.6 Policy a1 ::

There Policy 1, identified by BSID a1 :: / 128, introduces an increase in packet size which, according to the formula in §3.4 , is equal to:

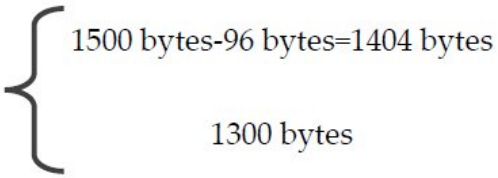
$$40bytes + 8bytes + 16 * 3bytes = 96bytes \tag{3.2}$$

Considering that the minimum MTU along the path after encapsulating the traffic in node E1 and, in the reverse path, of node E2 is equal to 1500 bytes:

$$1500bytes - 96bytes = 1404bytes \tag{3.3}$$

That is, the maximum size of the packets that respect the MTUs in the links inside the network is 1404 bytes. However, the MTU in host A - E1 and host B - E2 links is 1300 bytes:

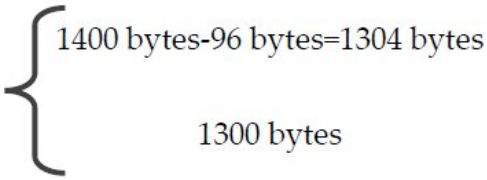
In other words, in the case of Policy 1, there can be no overrun of MTU in internal connections as hosts A and B, encapsulating the packets according to the MTU of their interface, automatically



respect the *MTU* limitations in intermediate connections despite the addition of the *SRH*.

3.5.7 Policy a3 ::

There Policy 3, identified by BSID a3 :: / 128, introduces the same size increase as Policy 1. The difference lies in the minimum *MTU* of the path, which is given by the link C3 - C4, that is 1400 bytes:



For the same reason in §3.5.6, hosts A and B respect the internal MTUs when encapsulating packets.

3.5.8 Policy a1 :: with recovery Policy a2 ::

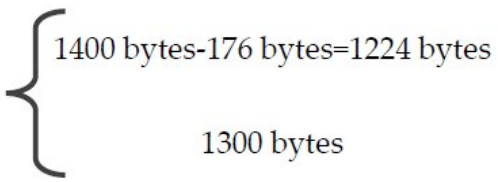
In §3.5.6 and in §3.5.7 it has been noted that the *MTU* of the interface of hosts A and B constitutes the minimum *MTU* of the path identified by BSIDs a1 :: / 128 and a3 :: / 128. In the case of joint use of the Recovery Policy with Policy a1 :: / 128, there are further considerations that modify the result compared to the previous cases. In fact, the minimum *MTU* of the path identified by Policies 1 and 2 after the double encapsulation, is given by the C3 - C4 link which is equal to 1400 bytes. With reference to the calculation of the dimension introduced by the double heading *SRH* in §2.2, the increase amounts to:

$$40bytes + 8bytes + 16 * 3bytes + 40bytes + 8bytes + 16 * 2bytes = 176bytes \tag{3.4}$$

Considering the MTU of connection C3 - C4:

$$1400bytes - 176bytes = 1224bytes \tag{3.5}$$

The amount of data that can be transmitted is 1224 bytes. However, the minimum *MTU* of the path is represented by the *MTU* of the interfaces of hosts A and B:



We note that although the minimum *MTU* is 1300 bytes, in reality the amount of data that can be transmitted over the entire path is, at most, 1224 bytes. This quantity is not detectable by currently available algorithms and can be found according to the above considerations. The main consequence is that, not being aware of the actual amount of data that can be transmitted due to repeated and unpredictable encapsulations, hosts A and B fragment packets according to the *MTU* equal to 1300 bytes. Therefore, in sending, for example, a large file on which fragmentation will necessarily occur, the host A will send packets of the maximum available size which, according to its parameters, is greater than 1224 bytes causing a Black Hole in connection C3 - C4. This theoretical discussion is flanked below by the experimental demonstration.

3.5.9 Starting the trial

As already mentioned in §3.5.1, the setup of the testbed takes place with the aid of a Python script which, by recalling the various files, applies the settings and parameters presented so far.

```
0) Create topology (running VPP instances, configuring link and SRv6).
1) TCP traffic (top-policy (a1::) ).
2) ICMPv6 traffic (down-policy (a3::) ).
3) Simultaneous TCP and ICMPv6 traffic.
4) Enable Recovery Policy.
5) Disable Recovery Policy.
6) MTU Path Discovery (TCP).
9) Exit and kill VPP instances & delete interfaces.
10) Exit without killing VPP instances & deleting interfaces.
Option:
```

Figure 3.11: Options available in the Python script

In the following paragraph, the development of the experiment will be presented, which will allow us to produce the tests that show the presence of a Black Hole. In §3.5.4 the presence of a traffic classifier has already been indicated: the latter has the task of separating the *TCP* traffic from the *ICMP* traffic, encapsulating the first in Policy 1 and the second in Policy 3. Recalling that Policy 1 the Recovery Policy can be added, it is noted that the paths involved by the latter and by Policy 3 have in common the passage on the C3 - C4 connection, as shown in Figure 3.9.

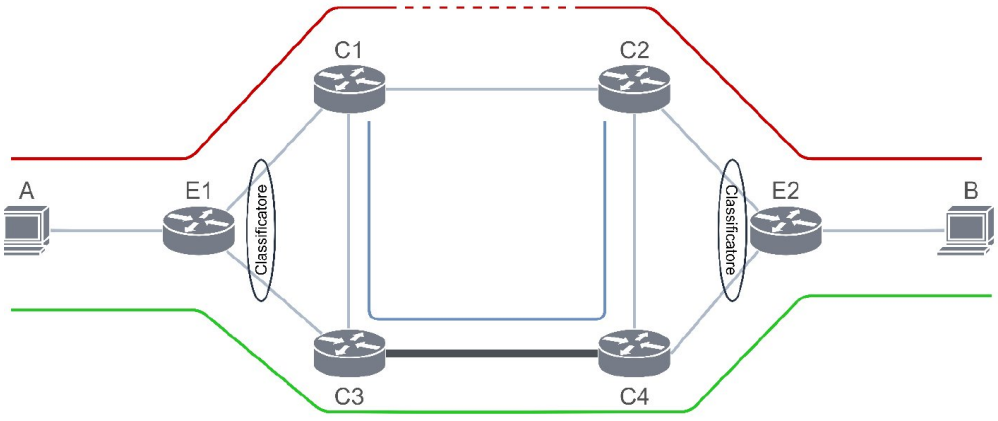


Figure 3.12: Link C3 - C4 in common in Policies 1 and 3

In fact, in this last link we want to show that, while the *ICMP* traffic is forwarded on the link

since the low size of the message encapsulated in Policy 3 does not exceed the *MTU*, the *TCP* traffic, whose packets exceed a certain size, will be discarded. In this way, the correct functioning of the C3 - C4 link is highlighted and therefore does not constitute an element of cause of the dropping of *TCP* packets. Analyzing subsequently the tracing of *TCP* traffic in node C3, we will see that the size of the packets received so far exceeds in size the *MTU* of the C3 - C4 link with consequent rejection of the same, without forwarding any notification to the source. As shown in Figure 3.11, two cases are distinguished: the case in which the link fail in C1 - C2 with consequent activation of the Recovery Policy, and the case in which all the connections are functional and the Recovery Policy is deactivated.

3.5.10 Recovery Policy not active

Referring to Figure 3.9, this is the case where *TCP* traffic follows Policy 1 (in red) and *ICMP* traffic follows Policy 3 (in green). Before starting any communication, it is necessary to execute the *MTU* Path Discovery algorithm (§4.2.5) with *TCP* protocol, by manually launching the appropriate command in the Python script. The traffic classifier will encapsulate the *TCP* packets in Policy 1 in which the minimum *MTU* of the path, according to the parameters presented in §3.5.3, is represented by that of the interface of host A and host B, i.e. 1300 bytes .

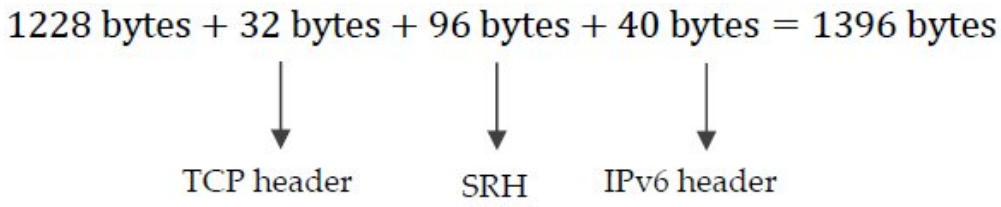
```
Choose option:
0) Create topology (running VPP instances, configuring link and SRv6).
1) TCP traffic (top-policy (a1::) ).
2) ICMPv6 traffic (down-policy (a3::) ).
3) Simultaneous TCP and ICMPv6 traffic.
4) Enable Recovery Policy.
5) Disable Recovery Policy.
6) MTU Path Discovery (TCP).
9) Exit and kill VPP instances & delete interfaces.
10) Exit without killing VPP instances & deleting interfaces.
Option: 6
traceroute to fc00::2:2 (fc00::2:2), 30 hops max, 80 byte packets
 1  fc00::3:0:2 (fc00::3:0:2)  0.708 ms  0.613 ms  1.144 ms
 2  fc00::3:0:2 (fc00::3:0:2)  4.395 ms  0.841 ms  0.584 ms
 3  fc00::2:2 (fc00::2:2)  0.576 ms  0.595 ms  0.385 ms
The MTU path discovery (TCP) result is:
fc00::2:2 from :: via fe80::fe:30ff:fec5:9bc9 dev vethns1 proto ra src fc00::1:2
 metric 1024 mtu 1300 hoplimit 64 pref medium
```

Figure 3.13: MTU Path Discovery Results

Now it is possible to initiate a *TCP* communication between host A and host B by launching the command from the Python script which, thanks to the utility iperf3, starts a server in host B and a client in host A. In §3.5.6 it was pointed out that, by fragmenting the packets according to *MTU* equal to 1300 bytes, the *MTU* of the internal connections cannot be exceeded. In fact, considering that the *TCP* header has a size of 32 bytes and the *IPv6* header has a size of 40 bytes, the maximum amount of useful data that can be transmitted in the host connection A - E1 with the *TCP* protocol is the following:

$$1300bytes - 40bytes - 32bytes = 1228bytes \tag{3.6}$$

That is, the *Maximum Segment Size (MSS)* is equal to 1228 bytes. Subsequently, the encapsulation in E1 takes place and the packets, without considering the 14 bytes of the ethernet protocol, will have the following size:



This size does not exceed the maximum packet size allowed along the Policy 1 path, i.e. 1404 bytes (§3.5.6). Even *ICMPv6* traffic, not having a particularly large size, will be forwarded in the host link A - E1, then encapsulated by node E1 in Policy 3. It will therefore reach the destination. Using the option in the Python script to simultaneously start *TCP* and *ICMPv6* traffic, the results are shown in Figures 3.14 and 3.15. To generate the *TCP* traffic, a 10.4 Kbyte file was sent. As can be seen from Figure 3.14 and as anticipated above, each *TCP* segment has a length of 1228 bytes. Figure 3.15 shows the correct sending and receiving of *ICMP* messages between host A and host B. To capture the traffic flows, Wireshark is the only interface of host B.

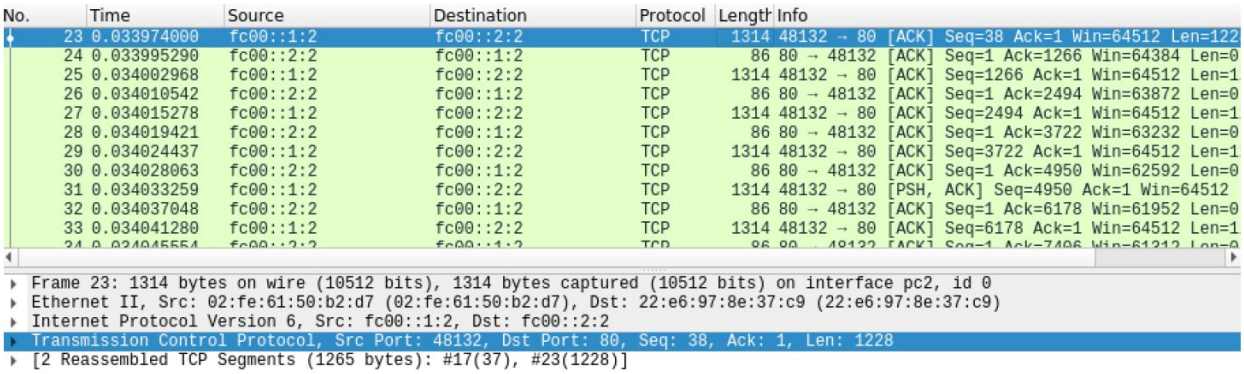


Figure 3.14: TCP Traffic Policy 1

3.5.11 Recovery Policy active

Referring to Figure 3.9, this is the case in which *TCP* traffic follows Policy 1 (in red) and Policy 2 (in blue), the latter due to a simulated link failure of C1 - C2, while *ICMP* traffic follows Policy 3 (in green). As in §3.5.10, it is necessary to manually launch the command in the Python script to execute the *MTU* Path Discovery algorithm, the result of which is the *MTU* of the interface of host A and host B: 1300 bytes. In §3.5.8 the amount of data that can be transmitted in the C3 - C4 link has been calculated considering also the double encapsulation, and it is less than the 1300 bytes calculated by the *MTU* Path Discovery and according to which the fragmentation will take place. L'*MSS*, therefore, is calculated as in §3.5.10 and is equal to 1228 bytes. Then, host A sends the *TCP* packets to E1 and their size is:

$$1228bytes + 40byte + 32bytes = 1300bytes \tag{3.7}$$

That is, it conforms to the MTU of host link A - E1. The moment packets enter the domain

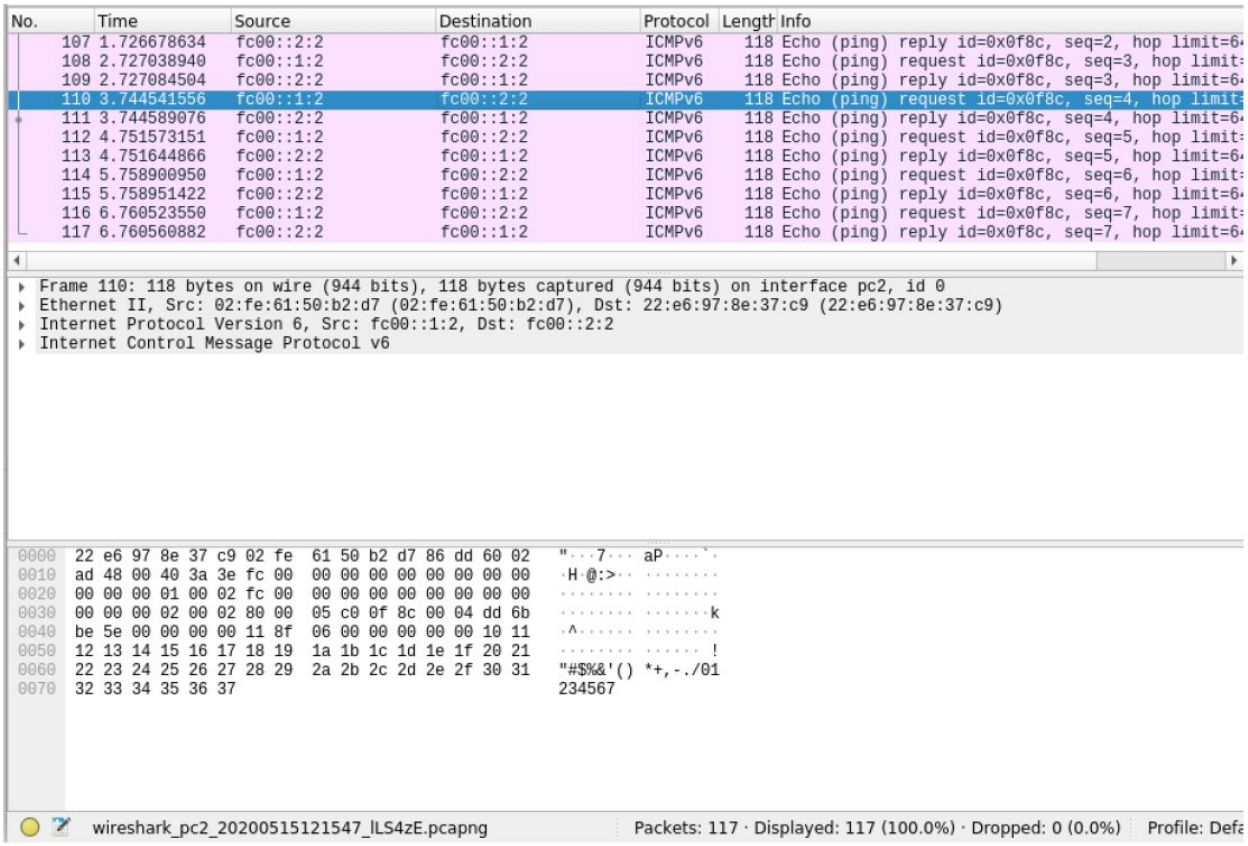


Figure 3.15: *ICMP* Traffic Policy 3

SRv6, the first encapsulation takes place which, already calculated in §3.5.6, amounts to 96 bytes:

$$1300bytes + 96bytes = 1396bytes \tag{3.8}$$

Being less than the MTU of the E1 - C1 connection which is equal to 1500 bytes, as shown in Figure 3.16 is correctly forwarded (to have the complete size of the packets at the value 1396 calculated above, 14 bytes of the ethernet header must be added).

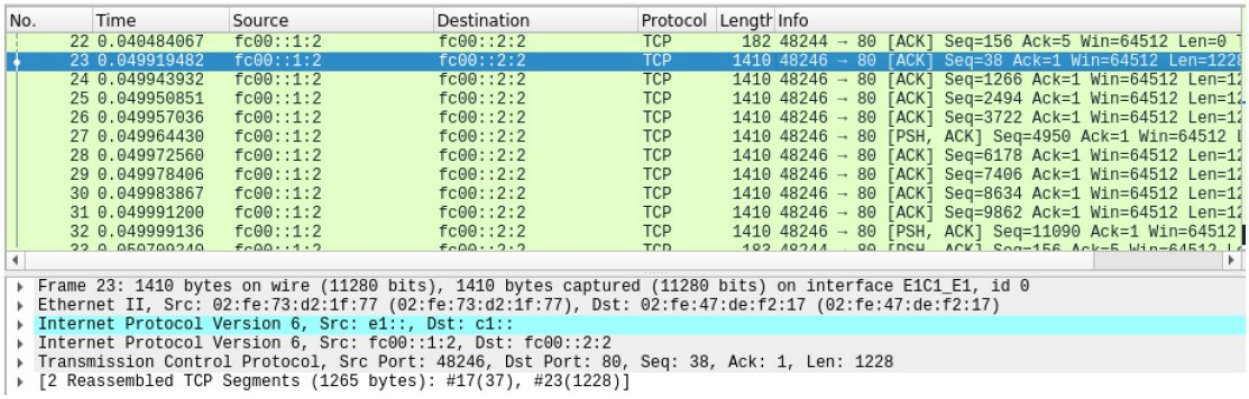


Figure 3.16: TCP packets received in the E1 interface of the E1-C1 link

Now, packages are encapsulated in Recovery Policy, or Policy 2, in order to bypass the link fail in C1 - C2. The increase in packet size, calculated with the formula in §3.4 , is equal to 80 bytes:

$$40bytes + 8bytes + 16 * 2 = 80bytes \tag{3.9}$$

Adding them to the 1396 bytes calculated above:

$$1396bytes + 80bytes = 1476bytes \tag{3.10}$$

Therefore, they respect the MTU of the C1 - C3 connection which is equal to 1500 bytes. (To calculate the total size, 14 bytes of the ethernet header must be added: 1476 bytes + 14 bytes = 1490 bytes).

No.	Time	Source	Destination	Protocol	Length	Info
21	0.027123094	fc00::2:2	fc00::1:2	TCP	183	80 → 48268 [PSH, ACK] Seq=4 Ack=156 Win=65152 L
22	0.027379440	fc00::1:2	fc00::2:2	TCP	262	48268 → 80 [ACK] Seq=156 Ack=5 Win=64512 Len=0
23	0.034815324	fc00::1:2	fc00::2:2	TCP	1490	48270 → 80 [ACK] Seq=38 Ack=1 Win=64512 Len=122
24	0.034827701	fc00::1:2	fc00::2:2	TCP	1490	48270 → 80 [ACK] Seq=1266 Ack=1 Win=64512 Len=1
25	0.034833184	fc00::1:2	fc00::2:2	TCP	1490	48270 → 80 [ACK] Seq=2494 Ack=1 Win=64512 Len=1
26	0.034839289	fc00::1:2	fc00::2:2	TCP	1490	48270 → 80 [ACK] Seq=3722 Ack=1 Win=64512 Len=1
27	0.034845364	fc00::1:2	fc00::2:2	TCP	1490	48270 → 80 [PSH, ACK] Seq=4950 Ack=1 Win=64512
28	0.034850810	fc00::1:2	fc00::2:2	TCP	1490	48270 → 80 [ACK] Seq=6178 Ack=1 Win=64512 Len=1
29	0.034856932	fc00::1:2	fc00::2:2	TCP	1490	48270 → 80 [ACK] Seq=7406 Ack=1 Win=64512 Len=1
30	0.034862741	fc00::1:2	fc00::2:2	TCP	1490	48270 → 80 [ACK] Seq=8634 Ack=1 Win=64512 Len=1
31	0.034868132	fc00::1:2	fc00::2:2	TCP	1490	48270 → 80 [ACK] Seq=9862 Ack=1 Win=64512 Len=1

Figure 3.17: TCP packets received in the C3 interface of the C1-C3 link

However, the MTU of the C3 - C4 link is 1400 bytes, less than the size of the packets arriving at node C3. Therefore, C3 discards the packets and, not being enabled to send messages *ICMPv6* Packet too big, it does not inform the host you arise (host A). In fact, by adding a packet tracing at node C3 the result is the following:

```
    tos 0x00, flow label 0x3cb92, hop limit 63, payload length 1436
04:11:30:360466: ip6-load-balance
  fib 0 dpo-idx 8 flow hash: 0x00000000
  IPV6_ROUTE: :: -> c4::1
    tos 0x00, flow label 0x3cb92, hop limit 63, payload length 1436
04:11:30:360468: ip6-rewrite
  tx_sw_if_index 0 adj-idx 8 : ipv6 via fc00::5:0:1 host-C3C4_C3: mtu:1400 02fed497330102fe
a4015e5286dd flow hash: 0x00000002
  00000000: 6003cb92059c2b3e0000000000000000000000000000000c4000000000000
  00000020: 0000000000000012904040001000000000c400000000000000000000000000001
  00000040: 00c3000000000000000000000000006003cb92054c2b3e00e1000000000000
  00000060: 0000000000000000c2000000000000000000000000029060401
04:11:30:360475: ip6-icmp-error
  IPV6_ROUTE: :: -> c4::1
    tos 0x00, flow label 0x3cb92, hop limit 62, payload length 1436
04:11:30:360491: error-drop
  rx:host-C1C3_C3
04:11:30:360495: drop
  ip6-input: ip6 MTU exceeded
```

Figure 3.18: Packet dropping of node C3

Further evidence of non-receipt is indicated by the statistics provided by the iperf tool, both by the client and by the server:

The Server summary indicates that a connection was accepted on port 80 and was requested by the address *IPv6* fc00 :: 1: 2, which is host A. At the time of the test, a *TCP* stream was started whose statistics highlighted in red indicate no data reception.

The Client summary indicates that a connection has been initiated with the host from the

```
-----------------------------------------------------------
Server listening on 80
-----------------------------------------------------------
Time: Fri, 15 May 2020 12:25:20 GMT
Accepted connection from fc00::1:2, port 48300
      Cookie: 6iv4fwamqo52yirrfxmwqzqvbaneujryoy3h
      TCP MSS: 0 (default)
[  5] local fc00::2:2 port 80 connected to fc00::1:2 port 483
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks,
 1 blocks to send, tos 0
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate
[  5] (sender statistics not available)
[  5]   0.00-0.01   sec  0.00 Bytes  0.00 bits/sec
rcv_tcp_congestion cubic
iperf 3.7
Linux giulio-VirtualBox 5.4.0-29-generic #33-Ubuntu SMP Wed A
2020 x86_64
-----------------------------------------------------------
Server listening on 80
-----------------------------------------------------------
```

Figure 3.19: TCP Server Statistics

```
Control connection MSS 1228
Time: Fri, 15 May 2020 12:25:20 GMT
Connecting to host fc00::2:2, port 80
      Cookie: 6iv4fwamqo52yirrfxmwqzqvbaneujryoy3h
      TCP MSS: 1228 (default)
[  5] local fc00::1:2 port 48302 connected to fc00::2:2 port 80
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
 1 blocks to send, tos 0
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-0.00   sec  43.2 KBytes   321 Mbits/sec    0   12.0 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-0.00   sec  43.2 KBytes   321 Mbits/sec    0             sender
      Sent 43.2 KByte / 10.2 KByte (424%) of /home/giulio/Desktop/BlackHolesFi
nal3/big_file
[  5]   0.00-0.01   sec  0.00 Bytes  0.00 bits/sec                  receiver
CPU Utilization: local/sender 26.8% (0.0%u/26.8%s), remote/receiver 0.7% (0.0%u/
0.7%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

Figure 3.20: TCP Client Statistics

addressIPv6 fc00 :: 2: 2, that is host B, on port 80. From the statistics it is observed that the client has tried to send the file, sending 43.2 Kbytes and setting the *MSS* to 1228 bytes. The data sent was all lost while the network experienced an increase in terms of congestion. In fact, as already explained in §3.1, *TCP* attempts to retransmit packets:

No.	Time	Source	Destination	Protocol	Length	Info
26	0.050980224	fc00::1:2	fc00::2:2	TCP	1314	48302 → 80 [ACK] Seq=3722 Ack=1 Win=64512 Len=1
27	0.050980405	fc00::1:2	fc00::2:2	TCP	1314	48302 → 80 [PSH, ACK] Seq=4950 Ack=1 Win=64512
28	0.051064539	fc00::1:2	fc00::2:2	TCP	1314	48302 → 80 [ACK] Seq=6178 Ack=1 Win=64512 Len=1
29	0.051064815	fc00::1:2	fc00::2:2	TCP	1314	48302 → 80 [ACK] Seq=7406 Ack=1 Win=64512 Len=1
30	0.051065019	fc00::1:2	fc00::2:2	TCP	1314	48302 → 80 [ACK] Seq=8634 Ack=1 Win=64512 Len=1
31	0.051065205	fc00::1:2	fc00::2:2	TCP	1314	48302 → 80 [ACK] Seq=9862 Ack=1 Win=64512 Len=1
32	0.051065390	fc00::1:2	fc00::2:2	TCP	1314	48302 → 80 [PSH, ACK] Seq=11090 Ack=1 Win=64512
33	0.051966020	fc00::1:2	fc00::2:2	TCP	87	48300 → 80 [PSH, ACK] Seq=156 Ack=5 Win=64512 L
34	0.057066750	fc00::2:2	fc00::1:2	TCP	86	80 → 48300 [ACK] Seq=5 Ack=157 Win=65152 Len=0
35	0.057096180	fc00::2:2	fc00::1:2	TCP	86	80 → 48302 [FIN, ACK] Seq=1 Ack=38 Win=65152 Le
36	0.059190688	fc00::2:2	fc00::1:2	TCP	87	80 → 48300 [PSH, ACK] Seq=5 Ack=157 Win=65152 L
37	0.059259748	fc00::1:2	fc00::2:2	TCP	86	48300 → 80 [ACK] Seq=157 Ack=6 Win=64512 Len=0
38	0.059713616	fc00::1:2	fc00::2:2	TCP	90	48300 → 80 [PSH, ACK] Seq=157 Ack=6 Win=64512 L
39	0.061410965	fc00::2:2	fc00::1:2	TCP	86	80 → 48300 [ACK] Seq=6 Ack=161 Win=65152 Len=0
40	0.061457802	fc00::1:2	fc00::2:2	TCP	350	48300 → 80 [PSH, ACK] Seq=161 Ack=6 Win=64512 L
41	0.064024382	fc00::1:2	fc00::2:2	TCP	86	48302 → 80 [ACK] Seq=12318 Ack=2 Win=64512 Len=
42	0.066785373	fc00::1:2	fc00::2:2	TCP	1314	48302 → 80 [ACK] Seq=12318 Ack=2 Win=64512 Len=
43	0.069180248	fc00::2:2	fc00::1:2	TCP	86	80 → 48300 [ACK] Seq=6 Ack=425 Win=64896 Len=0
44	0.069213982	fc00::2:2	fc00::1:2	TCP	90	80 → 48300 [PSH, ACK] Seq=6 Ack=425 Win=64896 L
45	0.114332455	fc00::1:2	fc00::2:2	TCP	86	48300 → 80 [ACK] Seq=425 Ack=10 Win=64512 Len=0
46	0.116380656	fc00::2:2	fc00::1:2	TCP	350	80 → 48300 [PSH, ACK] Seq=10 Ack=425 Win=64896
47	0.116410442	fc00::1:2	fc00::2:2	TCP	86	48300 → 80 [ACK] Seq=425 Ack=274 Win=64384 Len=
48	0.117010505	fc00::1:2	fc00::2:2	TCP	87	48300 → 80 [PSH, ACK] Seq=425 Ack=274 Win=64384
49	0.117050739	fc00::1:2	fc00::2:2	TCP	86	48300 → 80 [FIN, ACK] Seq=426 Ack=274 Win=64384
50	0.121583655	fc00::2:2	fc00::1:2	TCP	86	80 → 48300 [FIN, ACK] Seq=274 Ack=426 Win=64896
51	0.121636740	fc00::1:2	fc00::2:2	TCP	86	48300 → 80 [ACK] Seq=427 Ack=275 Win=64384 Len=
52	0.127074104	fc00::2:2	fc00::1:2	TCP	86	80 → 48300 [ACK] Seq=275 Ack=427 Win=64896 Len=
53	0.274710030	fc00::1:2	fc00::2:2	TCP	1314	[TCP Retransmission] 48302 → 80 [ACK] Seq=38 Ac
54	0.699749602	fc00::1:2	fc00::2:2	TCP	1314	[TCP Retransmission] 48302 → 80 [ACK] Seq=38 Ac
55	1.533672481	fc00::1:2	fc00::2:2	TCP	1314	[TCP Retransmission] 48302 → 80 [ACK] Seq=38 Ac
56	3.200505163	fc00::1:2	fc00::2:2	TCP	1314	[TCP Retransmission] 48302 → 80 [ACK] Seq=38 Ac
57	6.728045114	fc00::1:2	fc00::2:2	TCP	1314	[TCP Retransmission] 48302 → 80 [ACK] Seq=38 Ac

Figure 3.21: Host A TCP retransmissions

In §3.1 the arrival at the destination of the packets of small dimensions is foreseen in fact, the *TCP* connection between the two hosts has been correctly established, i.e. the *TCP* packets such as SYN and FIN have not suffered the effects of Black Hole. The same is also true for traffic *ICMPv6* encapsulated in Policy 3, as shown in Figure 3.22: Host B receives and responds to *ICMPv6* messages sent by host A.

22	0.040384393	fc00::1:2	fc00::2:2	TCP	86	43610 → 80 [ACK] Seq=156 Ack=5 Win=64512 Len=0
23	0.053505902	fc00::1:2	fc00::2:2	TCP	87	43610 → 80 [PSH, ACK] Seq=156 Ack=5 Win=64512 L
24	0.053541825	fc00::2:2	fc00::1:2	TCP	86	80 → 43610 [ACK] Seq=5 Ack=157 Win=65152 Len=0
25	0.054400062	fc00::2:2	fc00::1:2	TCP	86	80 → 43612 [FIN, ACK] Seq=1 Ack=38 Win=65152 Le
26	0.055134823	fc00::2:2	fc00::1:2	TCP	87	80 → 43610 [PSH, ACK] Seq=5 Ack=157 Win=65152 L
27	0.057178966	fc00::1:2	fc00::2:2	TCP	86	43610 → 80 [ACK] Seq=157 Ack=6 Win=64512 Len=0
28	0.057399322	fc00::1:2	fc00::2:2	TCP	86	[TCP Previous segment not captured] 43612 → 80
29	0.057891251	fc00::1:2	fc00::2:2	TCP	90	43610 → 80 [PSH, ACK] Seq=157 Ack=6 Win=64512 L
30	0.057912544	fc00::2:2	fc00::1:2	TCP	86	80 → 43610 [ACK] Seq=6 Ack=161 Win=65152 Len=0
31	0.058438582	fc00::1:2	fc00::2:2	TCP	351	43610 → 80 [PSH, ACK] Seq=161 Ack=6 Win=64512 L
32	0.058449771	fc00::2:2	fc00::1:2	TCP	86	80 → 43610 [ACK] Seq=6 Ack=426 Win=64896 Len=0
33	0.058852727	fc00::2:2	fc00::1:2	TCP	90	80 → 43610 [PSH, ACK] Seq=6 Ack=426 Win=64896 L
34	0.059516467	fc00::1:2	fc00::2:2	TCP	86	43610 → 80 [ACK] Seq=426 Ack=10 Win=64512 Len=0
35	0.059628779	fc00::2:2	fc00::1:2	TCP	351	80 → 43610 [PSH, ACK] Seq=10 Ack=426 Win=64896
36	0.063735722	fc00::1:2	fc00::2:2	TCP	86	43610 → 80 [ACK] Seq=426 Ack=275 Win=64384 Len=
37	0.063827638	fc00::2:2	fc00::1:2	TCP	87	80 → 43610 [PSH, ACK] Seq=275 Ack=426 Win=64896
38	0.065024847	fc00::1:2	fc00::2:2	TCP	86	43610 → 80 [ACK] Seq=426 Ack=276 Win=64384 Len=
39	0.068670182	fc00::1:2	fc00::2:2	TCP	87	43610 → 80 [PSH, ACK] Seq=426 Ack=276 Win=64384
40	0.068765522	fc00::2:2	fc00::1:2	TCP	86	80 → 43610 [FIN, ACK] Seq=276 Ack=427 Win=64896
41	0.071443210	fc00::1:2	fc00::2:2	TCP	86	43610 → 80 [FIN, ACK] Seq=427 Ack=277 Win=64384
42	0.071483836	fc00::2:2	fc00::1:2	TCP	86	80 → 43610 [ACK] Seq=277 Ack=428 Win=64896 Len=
43	0.844339914	fc00::1:2	fc00::2:2	ICMPv6	118	Echo (ping) request id=0x11fc, seq=1, hop limit
44	0.844383608	fc00::2:2	fc00::1:2	ICMPv6	118	Echo (ping) reply id=0x11fc, seq=1, hop limit=6
45	1.845933897	fc00::1:2	fc00::2:2	ICMPv6	118	Echo (ping) request id=0x11fc, seq=2, hop limit
46	1.845972053	fc00::2:2	fc00::1:2	ICMPv6	118	Echo (ping) reply id=0x11fc, seq=2, hop limit=6
47	2.847936548	fc00::1:2	fc00::2:2	ICMPv6	118	Echo (ping) request id=0x11fc, seq=3, hop limit
48	2.847992330	fc00::2:2	fc00::1:2	ICMPv6	118	Echo (ping) reply id=0x11fc, seq=3, hop limit=6

Figure 3.22: *ICMPv6* and TCP messages in Host B

After deducing the existence of *MTU* Black Holes by retracing the dynamics of an *SRv6* network, a scenario was proposed in which, by emulating the real mechanisms of these networks, it was

possible to grasp behaviors of essential relevance, provided by the study of traffic flows for specific protocols and from the reaction analysis of both the terminal hosts and the internal nodes of the SR network, with consequent formulation of hypotheses of formation of the Black Hole. Therefore, following empirical detection of traffic loss, data provided by the statistics of the iperf and wireshark tools, an in-depth look at the method of processing the packets in the concerned nodes allowed the production of the evidence that leads to the validation of the hypothesis of formation of Black Holes. The hypothesis of its existence was therefore validated. Although the phenomenon of the Black Hole may be statistically unlikely, its formation is a possibility that must be considered and that needs the treatment provided in this work. There are, in fact, many scenarios in which it can occur, especially if considered the high dynamism of Segment Routing networks. The cause of its formation is the joint use of different tools, widely used and which boast functions that cannot be renounced today. Its effects are obviously important, having repercussions on data consumption, *Network Status*, and above all traffic loss, without the possibility of recovery.

Chapter 4

Network Black Holes and Detection Frameworks

As defined in [19], network black holes are silent logical failures, often caused by events such as misconfiguration or software bugs. Among the different causes, the use of overlay architectures seems to be a common accelerator for the creation of black holes. A first example is reported in [8], where different failure modes that lead to the occurrence of a black hole are presented, in the context of an IP over *MPLS* infrastructure. In this scenario, failures affecting the Label Distribution Protocol (LDP) execution can create a black hole, due to the fact the underlying *IGP* domain is working correctly, while end to end reachability is compromised.

This work is focused on black holes occurring in an *IPv6* network due to the violation of the *MTU* constraint caused by the failure of the Path MTU Discovery (*PMTUD*). The different types of failure modes for the *PMTUD* procedure are described in [24]. Among those, the most common one is represented by unresponsive routers, that are configured to not send *ICMP* Packet Too Big (PTB) messages back to the source node whenever a packet with a length exceeding the *MTU* is received.

Many different detection systems have been proposed to detect the presence of network black holes in different contexts. All of them rely on the active test of the *Network Status* through the sending of probes. In the next we describe some of the most relevant detection tools.

In [19] an active probing detection mechanism is defined; it is able to detect network black holes occurring in an IP/MPLS backbone. A full mesh of probes are periodically exchanged among the edge routers. The method is based on the concept of *failure signature*, that represents the set of probes that are lost in case a black hole occurs in a specific link. Then, spatial correlation is exploited to identify a set of suspicious links. In particular, all the links whose failure signature is close to the actual set of failed probes are inserted into an hypothesis set.

One of the most reliable tools to discover *PMTUD* failures is Scamper [24]. Scamper is a two steps procedure to determine either the largest *MTU* that can be used on a end to end path, and to discover (in case of a failure) what is the router that is not participating in the *PMTUD*. Both the phases of Scamper are based on the enforcement of probes along the end to end path to check. These probes consist of a set of UDP segments destined to an unused port, when performing the first step, whereas a set of *ICMP* Echo messages destined to intermediate routers are used in the second phase.

Netalyzr is presented in [20], it is a network measurement and debugging tool to monitor the Internet. The architecture is provided with a set of pre-installed applets; one of these aims at determining the path *MTU* toward a destination server. This search is based on a process that emits a set of UDP probes to the target destination.

Ripe Atlas [32] is a worldwide monitoring infrastructure based on the use of hardware probes placed in the so called vantage points. In [5] Ripe Atlas has been used to discover path *MTU* Black Holes in the Internet, with the specific focus of assessing the main causes and the most affected data plane protocol. The obtained results show that black holes due to failure in the *PMTUD* procedure affect both the *IPv4* and the *IPv6* data planes. Specifically, Ripe Atlas is able to detect the main causes and the location of these failures, such as PTB messages and fragmented packets filtered by firewalls.

4.1 Data Set description

SR architecture provides a set of Operation And Maintenance (OAM) tools that Network Operators can use to measure the performance of their infrastructure, execute troubleshooting operations, and so on. Here we report a few examples. In [14] is presented a scalable and topology-aware data-plane monitoring system for SR-MPLS. Ping and Traceroute for *SR* networks are defined in [21]. Bidirectional Forwarding Detection (BFD) [18] to test the viability of Segment Routing Policies for Traffic Engineering is presented in.

SR capable nodes are able to collect statistics on the traffic by exploiting a set of traffic counters called *SR* Traffic Counter (*SRTCs*), allowing the performance of the traffic measurements at different granularity. By means of *SRTCs* an *SR* node is able to collect statistics on the received traffic flows aggregating them according to the active segment. Three different types of *SRTCs* are used in the proposed framework: i) SR-INT, ii) PSID, and iii) POL. SR-INTs (also known as link counts) account the traffic at link granularity, i.e., they measure the amount of *SR* traffic that is sent over a specific link. By means of the PSID counters, an *SR* capable node can count amount of received traffic that is directed toward a specific node. Finally, POL counters keep track of the amount of traffic that has been steered through a specific *SR Policy*. These counters are described in detail in [29], where the logical relations between them are captured by a mathematical model. Exploiting this model, the Authors show *SRTCs* can successfully be used to improve the performance of existing algorithms in different networking problems (e.g, Traffic Matrix Assessment, Traffic Anomaly Detection, etc.). The present paper is strongly based on the findings reported in [29].

In [23] an *SRv6* Performance Monitoring (SRv6-PM) framework is proposed, allowing deep performance monitoring on an *SRv6* infrastructure. Three main components are defined: i) a set of data plane tools for performing traffic measurements (e.g, packet loss, delay) at line rate, ii) a control plane logic that requires to the network nodes to perform specific measurements (and a southbound interface for the data/control plane interaction), and iii) a Cloud Native Big Data Management system for data storage, processing and visualization. As-use case for the validation of SRv6-PM, the fine grained measurement of the packet loss level affecting a single SR flow is considered. To measure the amount of packets that are lost for a specific flow, SRv6-PM exploits SR traffic counters instantiated at the ingress and egress nodes. Specifically, the difference among

these two counters represents the overall number of lost packets for the target flow.

SCMon [3] is a network wide monitoring system that allows users to check the health status of links. It exploits the source routing and the flexibility achieved by SR to create a set of monitoring cycles where to send probes to the viability each of them. By properly designing the different cycles it is possible to precisely localize a failed link. Furthermore, SCMon exploits adjacency SIDs of SR to test the status of IP links composed by bundle of connections at layer 2 (the inability to do that is a major drawback of the classical systems based on Bidirectional Forwarding Detection, BFD).

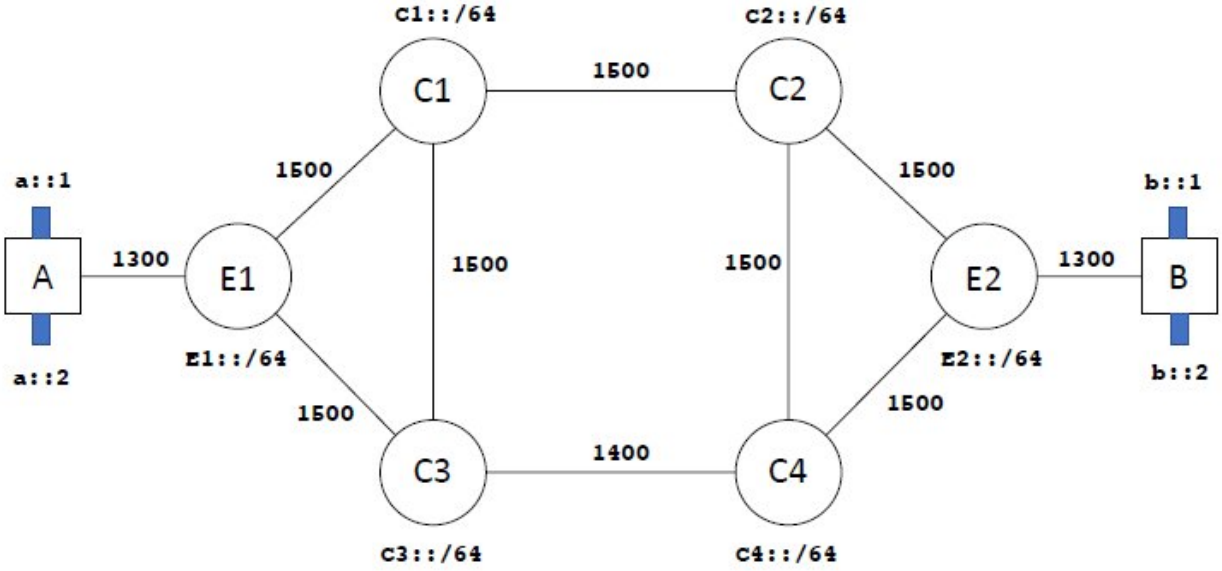


Figure 4.1: Reference scenario

Policy Name	Scope	Head end node	Segment List
pol 1	high reliability	E1	C1,C2,E2
pol 2	best effort	E1	C3,C4,E2
pol 2	link bypass	C1	C3,C4

Table 4.1: Main features of the policies configured in the emulated

4.2 IPv6 and MTU Path Discovery

In this paragraph the salient points of the Protocol will be highlighted IP (o Internet Protocol), in particular version 6 (briefly indicated with *IPv6*), useful for understanding the proof. *IPv6* is the successor of the protocol *IPv4*. Both belong to the network level (level 3) of the protocol stack *TCP (Transmission Control Protocol) e IP (Internet Protocol) (TCP/IP)*, shown in figure 4.2, and allow communication between terminal devices through one or more networks.

4.2.1 Level 3 services

To understand the protocol *IPv6*, a brief description of level three is required. The services it must guarantee are:

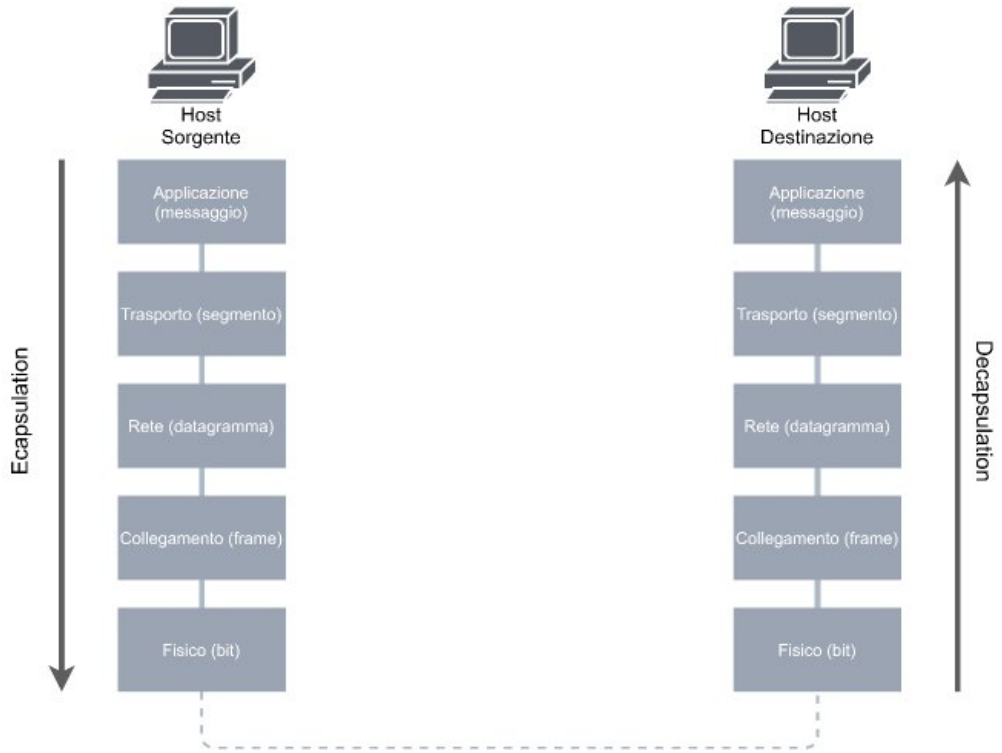


Figure 4.2: TCP/IP protocol stack with encapsulation and decapsulation direction

- the encapsulation. It is done from the source. This term indicates the encapsulation of data coming from the upper protocol level, that is transport, or level 4, in datagrams, defined as data units of level three. By "encapsulating," we mean adding additional data that enables the achievement of the objectives set by the level in question (in this case, level three allows communication between devices through one or more networks).
- the decapsulation. It is carried out by the destination. This term indicates the extraction of data from the datagram, that is, the elimination of the additional level three data inserted by the source. we then obtain the data unit of level 4, the segment.
- the routing. By adding data from the source, intermediate devices, or routers, are able to bring the packets to their destination. In fact, the data added in the encapsulation and then eliminated in the phase of decapsulation, provide information to routers about source, destination, packet size, and more.

4.2.2 Encapsulation of IP

IP encapsulates the transport layer segments by adding an IP Header. This header contains all the data needed to reach the destination and therefore is not altered or deleted until the destination host is reached. The encapsulation action allows independence between protocols of different levels. This way if as an alternative to the protocol *IPv4* the protocol is used *IPv6*, this does not affect, for example, layer 4 protocols. Consequently, in addition to modularity, the technique of encapsulation allows for scalability. This is reflected in the entire protocol stack: it too, in fact, has the property of scalability and the insertion of an additional level in the protocol stack adds a higher level

abstraction.

4.2.3 Characteristics of IP

To avoid an excessive increase in packet size, IP provides only the information necessary to reach the destination. The basic features offered by IP are three:

- Connectionless. The destination is not informed in advance when the source sends data. That is, an initial connection or exchange of information is not established between source and destination, which instead may be necessary for correct data transmission.
- Best effort. The IP protocol does not guarantee the arrival of packets, the order of receipt and their correctness. These services must be offered by other higher level protocols. IP is mainly concerned with identifying the source and destination.
- Medium independent. IP operates independently of the transmission medium. In fact, it is the task of the lower layer, the link layer, to prepare an IP packet for transmission. This means that, thanks to its scalability, IP can be used regardless of the type of transmission medium. However, one parameter that concerns it is taken into account by the network layer: it is the *MTU*, or maximum transmission unit. This value is provided by the link layer and indicates the maximum size limit of a single unit of data that can be transmitted on a specific link. Based on this value, the maximum size of a packet is determined.

It is plausible that not all links a packet crosses to reach its destination will have the same *MTU*. For this reason, in some cases the intermediate devices, usually routers, perform fragmentation: the packet entering the device, not respecting the *MTU* of the outgoing interface, is further divided so that it can then be forwarded. However, it will be pointed out later that this feature is disabled for some protocols.

4.2.4 Internet Protocol version 6

To locate a device on a network, the IP protocol requires the assignment of strings, called IP addresses. Based on the addressing capacity, i.e. the quantity of addressable devices, and therefore the length of the IP address, we distinguish two versions of the IP protocol: the *IPv4* and the protocol *IPv6*. However, the length of the IP address is not the only factor in distinguishing one protocol from another. The *IPv6* protocol arises from the need to increase the addressing capacity, while solving two other difficulties: the expansion of the routing tables and the elimination of the use of NAT. Focusing only on increasing addressing capacity, the *IPv4* protocol provides addresses with a length of 32 bits while the *IPv6* protocol provides addresses with a length of 128 bits, with the possibility of addressing respectively 232 and 2128 devices. In order to facilitate reading, *IPv4* addresses are converted from binary format to decimal format, while *IPv6* addresses are converted to hexadecimal format. An example of the two notations is shown below:

The number following the IP address specifies the portion reserved for identifying the network, while it is complementary with respect to the maximum length of the address identifies the device. The differences and the services offered by the two protocols are many, but it is necessary to pay attention to a functionality provided by the *IPv4* protocol but not by the *IPv6* protocol, and which makes the latter one of the causes of the creation of a MTU Black Hole: fragmentation.

Figure 4.3: Local *IPv6* and *IPv6* addresses

4.2.5 Packet Fragmentation and MTU Path Discovery

As already mentioned in §4.2.3, the fragmentation of the packets is performed by the routers so that the limitation on the *MTU* of the interfaces of each device along the path is respected. However, for security reasons, this functionality is not provided for the *IPv6* protocol, while it is up to the end devices to perform it. To do this, it is necessary to know in advance the minimum *MTU* of the entire path that will be followed by the data flow: the source host executes the MTU Path Discovery (indicated by the abbreviation *PMTUD*). This algorithm, in the specific case of *IPv6*, consists in sending probe packets assuming initially that the minimum *MTU* is the one set by the outgoing interface of the source host. If the *MTU* of a device is less than the packet size along the path, a type message is sent to the sourcePacket Too Big of the protocol *ICMPv6*, indicating its value. The above procedure will be repeated until the destination is reached correctly. There are two possible reasons that can invalidate the effectiveness of this algorithm, both related to security:

- The device running the algorithm is equipped with a firewall, instructed to block protocol packets *ICMP*.
- Routers are not allowed to send packets *ICMP* in order not to provide sensitive information.

To overcome these problems, a variant of *PMTUD*: *Packetization Layer (PL)* Path MTU Discovery (*PLPMTUD*). The *PL* indicates the level of the protocol stack which is responsible for specifying the initial size of the packet. The levels employed are transport or higher. That is, as opposed to *PMTUD*, *PLPMTUD* uses protocols that establish end-to-end connections with the target. Therefore, it sends a series of segments of increasing size: if received, the minimum *MTU* of the path is increased, otherwise if a packet loss is recorded, it is concluded that the last recorded *MTU* value is the minimum.

4.3 Experimental Demonstration Of Possible Existence Of SR Black Holes

The goal of this paragraph is twofold:

- to summarize the experimental demonstration about the existence of possible *SR Black Holes* in a *SRv6* networks, and
- to show that active probing based tools are not trustworthy for detecting such a type of failures.

The test was conducted over an emulated network, created through virtual routers supporting *SRv6* as data plane technology. *VPP* [1] with *SRv6* plugin has been used. The experimental topology is shown in figure 4.1. Two hosts (A and B) are connected through the *SRv6* domain. The two

policies are reported in Tab. 4.1. In the considered scenario, the two policies impose two levels of reliability of network paths: *pol_1* is used for high reliability traffic, while *pol_2* is associated to the regular traffic. As an example, *pol_3* reported in Tab. 4.1 is configured in node C1; in case of failure of the link between nodes C1 and C2, the policy imposes a re-routing of the packets over an alternative path (C1-C3-C4).

```
Control connection MSS 1228
Time: Fri, 14 May 2021 12:25:20 GMT
Connecting to host b::1, port 80
        Cookie: 6iv4fwamqo52yirrfxmwqzqvbaneujryoy3h
        TCP MSS: 1228 (default)
[ 5] local a::1 port 48300 connected to b::1 port 80
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
 1 blocks to send, tos 0
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-0.00  sec   43.2 KBytes  321 Mbits/sec    0    12.0 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-0.00  sec   43.2 KBytes  321 Mbits/sec    0    sender
        Sent 43.2 Kbyte / 10.2 Kbyte (424%) of /home/giulio/Desktop/BlackHolesFi
nal3/big_file
[  5]   0.00-0.01  sec   0.00 Bytes   0.00 bits/sec         receiver
CPU Utilization: local/sender 26.8% (0.0%u/26.8%s), remote/receiver 0.7% (0.0%u/
0.7%s)
snd_tcp_congestion       cubic
rcv_tcp_congestion       cubic

iperf Done.
```

Figure 4.4: Snapshot of the Iperf window on client

```
-----------------------------------------------------------
Server listening on 80
-----------------------------------------------------------
Time: Fri, 14 May 2021 12:25:20 GMT
Accepted connection from a::1, port 48300
        Cookie: 6iv4fwamqo52yirrfxmwqzqvbaneujryoy3h
        TCP MSS: 0 (default)
[ 5] local b::1 port 80 connected to a::1 port 48300
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks,
 1 blocks to send, tos 0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate
[  5] (sender statistics not available)
[  5]   0.00-0.01  sec   0.00 Bytes   0.00 bits/sec
rcv_tcp_congestion       cubic
iperf 3.7
```

Figure 4.5: Snapshot of the Iperf window on server

To verify the existence of the *SR Black Hole* two traffic flows are created between hosts A and B. The first one is a *TCP* connection that requires the reliable transfer through the *SRv6* domain. This requirement is satisfied by adding a traffic classifier in the node E1 that steers the packets belonging to the *TCP* connection along the path specified by the policy *pol_1*. The second one is a series of *ICMP* Echo Requests, that are sent over the regular path by means of policy *pol_2*. T

As expected, no black hole has been experienced in the case of no link failures. In this situation,

the lowest *MTU* of the links belonging to the path followed by the *TCP* connection is equal to 1300 Bytes, while the overhead imposed by the use of *SRv6* is equal to 96 BytesSince the *MTU* of links crossed by the *TCP* connection that are internal to the *SRv6* domain is equal to 1500 Bytes, then no violation happens and the flow is correctly delivered to the destination.

In this situation the packets sent through the link C3-C4 have an overall length of 1476 Bytes, which exceeds the *MTU*, thus leading to a large quantity of packets to be silently dropped. Figure 4.4 shows the *Iperf* window at the client side of the *TCP* connection, where it can be seen the amount of traffic sent and the considered *MSS*. As shown in figure 4.5, due to the *SR Black Hole* the server does not receive the traffic. It is worth noting that the *TCP* connection has been successfully established, due to the small size of the messages exchanged during the three way handshake. Furthermore, the *ICMP* traffic, which is also crossing the link C3-C4, is correctly delivered to the destination. Summarizing, the previous experiment has proven the existence of a *SR Black Hole*. Clearly, this is an anomalous event that could happen only in case the network is not correctly configured. For instance, in the proposed experiment the sending of *ICMP* PTB messages was disabled (default configuration in *VPP*). This suggests that the *SRv6* domain must be carefully configured in order to avoid the creation of *SR Black Holes*, either enabling the sending of *ICMP* PTB messages on the nodes, and to properly design the policy enforced on the incoming flows. Clearly, misconfiguration is an unplanned and unwanted event, and generally [7] it is extremely hard to be found and corrected. For this reason, in this work we define a framework to help Network Operator to detect *SR Black Holes*.

4.4 Applying Active Probing Tools to Detect the *SR Black Hole*

The experimental demonstration about the existence of the *SR Black Hole* has been carried out in a non collaborative network environment, i.e., nodes that do not sent *ICMP* PTB messages. In this subsection we use the Scamper tool [24] to determine its effectiveness in the correct detection of the path *MTU*. Scamper has been though to detect *MTU* related black holes in a non collaborative network. It exploits a traceroute-like mechanism to accomplish two different tasks: i) find the link where the black hole occurs, and ii) determine the bottleneck *MTU*. UDP probes are sent along the path between the source and the destination nodes. At first small UDP probes are sent to test whether the destination is actually reachable or not. Next, a *PMTUD* process is executed by emitting bigger and bigger probes. In order to deal with unresponsive nodes, Scamper uses a timeout mechanism: if an answer is not obtained at the expiration of the timer, it assumes that the packet has been silently dropped due to the *MTU* constraint violation. After two consecutive timeout events, Scamper starts determining the maximum size for packets which supported. In particular it exploits a table of well known *MTU* values to speed up the process: when a given packet length is detected to exceed the *MTU* (through the timeout mechanism), *MTU* is fixed at the smallest value of the previously tested packet lengths. Once the path *MTU* value is determined, a traceroute like procedure is used to determine the hop where the bottleneck link is located. Probes size is set bigger than the path *MTU* and the Time To Live (or Hop Limit in case of *IPv6*) is incrementally increased. As soon as an *ICMP* Time Exceeded message is not received, the bottleneck link is declared as found: it is the one connecting the last responding node with its next hop.

A first observation arising from using Scamper in our scenario is that, due to the IP in IP

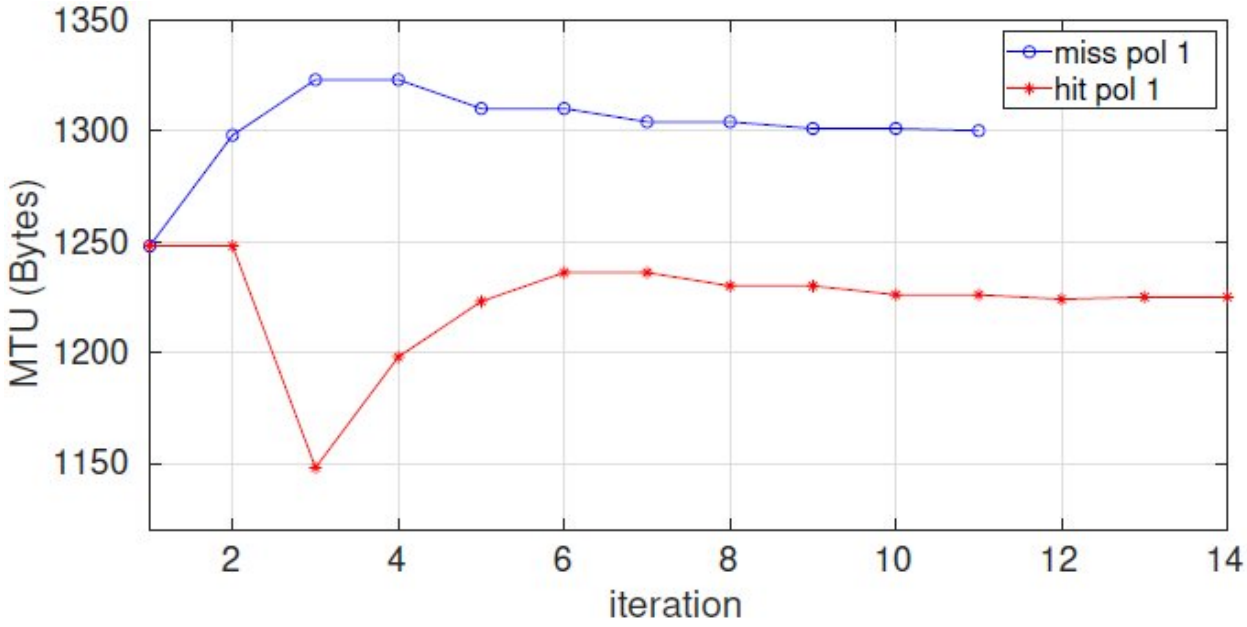


Figure 4.6: MTU assessment procedure followed by Scamper

encapsulation performed by the ingress node of the *SRv6* domain, the Hop Limit based mechanism for the detection of the location of the bottleneck link does not work whenever this node belongs to the *SRv6* domain. In fact, after the encapsulation the Hop Limit of the outer header is set to the default value, that is larger than the one reported in the inner header. Figure 4.6 shows *MTU* lengths that are tested by Scamper at each iteration of its execution. The value assumed in the last iteration is the one reported as output to the source host, which will generate packets accordingly. Two experiments are conducted. In the first one, we leave the network configuration unchanged with respect to the setting used in the previous experiment where the link between nodes C1 and C2 fails. In this scenario we have already commented that the *TCP* traffic is encapsulated twice, with an overall overhead due to *SRv6* processing of 176 Bytes. Despite of the fact that the link having the smallest *MTU* in the overall path followed by the *TCP* traffic is the one connecting the source node to E1, due to the *SRv6* overhead, the limit on the packet size is imposed by the link between nodes C3 and C4. As a consequence the maximum allowed size for packets injected in the *SRv6* domain is 1224 Bytes. Unfortunately, due to the policy based routing used in SR, the probes sent by Scamper follow a different path with respect to the data traffic; in particular, they are handled through the policy pol 2. The final *MTU* size obtained by Scamper in this case is the one reported by the blue curve in figure 4.6. The returned *MTU* size is equal to 1300 Bytes, as a consequence, the *TCP* source will generate packets having a size larger than the one that is supported, thus creating a black hole. The previous test has confirmed the intuition that active tools fail in detecting the *SR Black Hole* due to the policy based routing used in SR. To further stress this point, we have performed a second experiment by including in the node E1 a classification rule that forces Scamper probes to follow the same path of the *TCP* traffic. The outcome of the Scamper execution is represented by the red line reported in figure 4.6. As expected, under this setting Scamper correctly determines that the value of the maximum supported *MTU* is 1224. In fact, by injecting in the SR domain packets with such length, the enforcement of the *pol_1* and *pol_3* (that determine an SR related overhead equal to 176 Bytes) leads to the maximum size of

1400 Bytes, which is the *MTU* of the bottleneck link.

4.5 Segment Routing Black Holes Detection Algorithm (*SR-BHD*)

SR-BHD algorithm is the passive monitoring system that we have developed to detect *SR Black Holes* in SR networks. First the system model and the notation are presented , then the working principle of *SR-BHD* is introduced in two phases: the ideal model is firstly discussed, then, some modifications are introduced in order to make *SR-BHD* robust with respect to possible noise signals (e.g., packet loss due to congestion). Finally, an enhanced version of *SR-BHD*, called *SR-BHD*$^+$, based on the use of advanced traffic counters, is proposed to improve the precision of *SR-BHD*.

4.5.1 System model

Let $\mathcal{G}(\mathcal{N}, \mathcal{L})$ be the graph representing the topology of the considered SR domain, where $\mathcal{N}$ and $\mathcal{L}$ are the set of N nodes and L links respectively. Considering a link l, the head node is indicated with the notation $l.h$ and the tail node to as $l.t$ (the link leaves the tail and enters the head). In this scenario three different types of traffic counters are available. The first one is named *link count*, $y_L(l)$, and accounts the amount of traffic transmitted over the link l. The collection of all the link counts is represented by the vector $\mathbf{y}_L$. Furthermore, *PSID* and *POL* counters [29] are used to get statistics on the *SR* traffic. A PSID counter, instantiated at node i for the segment identifier a, accounts all the packets received at node i having a as active segment. This quantity is indicated with the symbol $y_B(i, a)$. The collection of all the PSID counters is represented by the vector $\mathbf{y}_B$.

SR capable nodes enforce the segment list on each incoming packet on the basis of an *SR Policy*. An *SR Policy* is represented by the tuple $< i, e, c >$, where i and e are the ingress and egress points of the *SR* tunnel and c is the color, that encodes the scope of the policy (i.e., low latency, best effort, high reliability, etc.). Each policy has an associated traffic counter (named POL counter) that accounts for the traffic that is steered through it. This quantity is indicated with the symbol $y_P(i, e, c)$. The collection of all the POL counters is represented by the vector $\mathbf{y}_P$.

Let us refer with the symbol $\mathcal{F}$ to the set of application flows (e.g, HTTP, SSH, etc.) that are injected in the *SR* domain. In order to transit the *SR* domain each application flow f is steered through an *SR Policy*. Considering the policy $< i, e, c >$, the set of application flows that are handled through it is referred to as $\Pi_{i,e,c}$. We define an SR *flow* as the aggregated traffic that is represented by the aggregate of all the application flows that are steered through the same policy. An *SR* flow is indicated with the symbol $x_{i,e,c}$. The collection of all the *SR* flows is represented by the vector $\mathbf{x}$.

In the *IPv6* underlay layer the shortest path policy is used to compute the path between every pair of nodes of the *SR* domain. The underlay path to go from node i to node a is referred to as $\mathbf{g}_{i,a}$ and is represented by a vector of length L, whose l-th component is equal to the percentage of traffic that goes over the link l if the node i sends one unit of traffic toward node a. With reference to the overlay, the routing is determined by the segment lists that are currently configured. The segment list associated whith the *SR Policy* $< i, e, c >$ is referred to with the symbol $\sigma_{i,e,c}$ and consists of an ordered sequence of node-SIDs. Then, the vector $\mathbf{r}_{i,e,c}$ representing the overlay path followed by the *SR* flow $x_{i,e,c}$ can be calculated using the following Equation 4.1:

$$\mathbf{r}_{i,e,c} = \sum_{k=1}^{|\sigma_{i,j,c}|-1} \mathbf{g}_{\sigma_{i,j,c}[k],\sigma_{i,j,c}[k+1]} \tag{4.1}$$

Equation 4.1 shows that the overlay routing of an *SR* flow is given by a linear combination of the vectors representing the paths in the underlay between nodes included in the segment list. The routing matrix $\mathbf{R}$ is the collection of the vectors $\mathbf{r}_{i,e,c}$ for all the existing policies. With reference to the PSID counter instantiated at node n for the active segment a, the variable $b^{n,a}_{i,j,c}$ represents the percentage of the *SR* flow $x_{i,e,c}$ that it accounts. The matrix $\mathbf{B}$ is given by the collection of the $b^{n,a}_{i,j,c}$ variables for all the PSID counters and *SR* flows. Then, the relation between the *SR* flows and the value assumed by each PSID counter can be expressed through the Equation 4.2.

$$\mathbf{y}_B = \mathbf{B} \cdot \mathbf{x} \tag{4.2}$$

4.5.2 SR-BHD Principle

SR-BHD is inspired by the flow conservation principle [17] which imposes that, considering a network node, the difference between the incoming and outgoing flow is equal to the local demand. Clearly, in case of the occurrence of an *SR Black Hole* on a link, the flow conservation principle is violated. In fact, a portion of the flow that should leave the node through the link where the *SR Black Hole* happens, is dropped, due to the *MTU* constraint. Consequently, the balance between the incoming and outgoing traffic is not more satisfied. Clearly, the occurrence of a black hole is not the only event that induces a violation of the flow conservation principle. Congestion, binary errors and destination unknown are examples of some common causes of packet loss leading to a violation of the equilibrium between traffic entering and leaving a node. In the rest of this subsection we present the equations that regulate the flow conservation principle. Then, in the remainder of the section we present a method to include the additional sources of packet loss into the mathematical models.

SR-BHD periodically monitors the traffic statistics collected by the network devices and verifies the validity of the flow conservation, and eventually raises an alarm for a possible *SR Black Hole*. Two different types of equations are defined in *SR-BHD* to check the validity of the flow conservation principle:

$$\sum_{a \in \mathcal{N}} g_{i,a}(l) \cdot y_B(i,a) = y_L(l) \quad \forall l \in \mathcal{L} \tag{4.3}$$

$$y_B(i,a) = \sum_{l \in \delta_i^+} g_{l.t,a}(l) \cdot y_B(l.t,a) \quad \forall i,a \in \mathcal{N} \tag{4.4}$$

Equation 4.5 imposes that the overall amount of traffic received at node i that has to be forwarded over the output link l is equal to the amount of traffic that is actually sent over the link l. In fact, each PSID counter accounts the traffic received at node i and having a as active segment. This traffic is routed according to the underlay path between the nodes i and a, that is represented by the vector $\mathbf{g}_{i,a}$. The multiplication of the l-th component of this vector with the value assumed by the PSID counter returns the portion of the traffic that is routed over the link l. Summing up the contribution of each possible active segment it is possible to calculate the amount of the received traffic that node i forwards over the output link l. In the ideal case (no losses) this quantity

coincides with the link count $y_L(l)$.

Equation 4.6, where the symbol δ_i^+ represents the set of links entering the node i, imposes the flow conservation principle at the active segment level. In particular, considering a node i and an active segment a, the counter $y_B(i,a)$ has to be equal to the overall traffic (having active segment a) that the neighbors of the node i send toward it.

Before presenting the details of *SR-BHD* implementation it is worth highlighting that, check the validity of Equation 4.6 is a critical task in a real network. In particular, it is due to the fact that traffic counters involved in the formula are instantiated at different network nodes. Consequently, lack of alignment of the traffic counters can compromise the validity of the flow conservation principle. On the contrary, the condition imposed by Equation 4.5 is robust with respect to alignment errors, since all the involved counters are instantiated in the same node.

$$\sum_{a \in \mathcal{N}} g_{i,a}(l) \cdot y_B(i,a) = y_L(l) \quad \forall l \in \mathcal{L} \tag{4.5}$$

$$y_B(i,a) = \sum_{l \in \delta_i^+} g_{l.t,a}(l) \cdot y_B(l.t,a) \quad \forall i,a \in \mathcal{N} \tag{4.6}$$

Equation 4.5 imposes that the overall amount of traffic received at node i that has to be forwarded over the output link l is equal to the amount of traffic that is actually sent over the link l. In fact, each PSID counter accounts the traffic received at node i and having a as active segment. This traffic is routed according to the underlay path between the nodes i and a, that is represented by the vector $\mathbf{g}_{i,a}$. The multiplication of the l-th component of this vector with the value assumed by the PSID counter returns the portion of the traffic that is routed over the link l. Summing up the contribution of each possible active segment it is possible to calculate the amount of the received traffic that node i forwards over the output link l. In the ideal case (no losses) this quantity coincides with the link count $y_L(l)$.

4.5.3 Framework Overview

Figure 4.7 shows the main block functions composing the proposed passive monitoring framework. It is thought to be integrated in a centralized monitoring system. Through a southbound interface the central monitoring system queries the network elements to collect the traffic statistics, which are stored in different databases. Every time new traffic measurements are available, the monitoring system triggers the execution of the *SR-BHD* block. It takes as input link and PSID traffic counters, the network topology and the current routing configuration (either in the *IPv6* underlay than the segment lists). *SR-BHD* block also receives as input the value of the margin, that consists in a vector containing the estimation of the packet loss due to different causes (e.g., congestion, transmission errors, etc.) for each network link. The margin estimation block requires to receive as input the current link utilization.

The output of *SR-BHD* block is a list containing a set of link/flow pairs that are suspected to be affected by an *SR Black Hole*. In order to improve the precision of the output, a refinement step can be performed through the adoption of the *SR-BHD*$^+$ block. Its execution requires the availability of a further set of traffic counters, i.e., the POL ones. Since these counters are not always available in *SR* capable nodes, the refinement performed by means of *SR-BHD*$^+$ is optional.

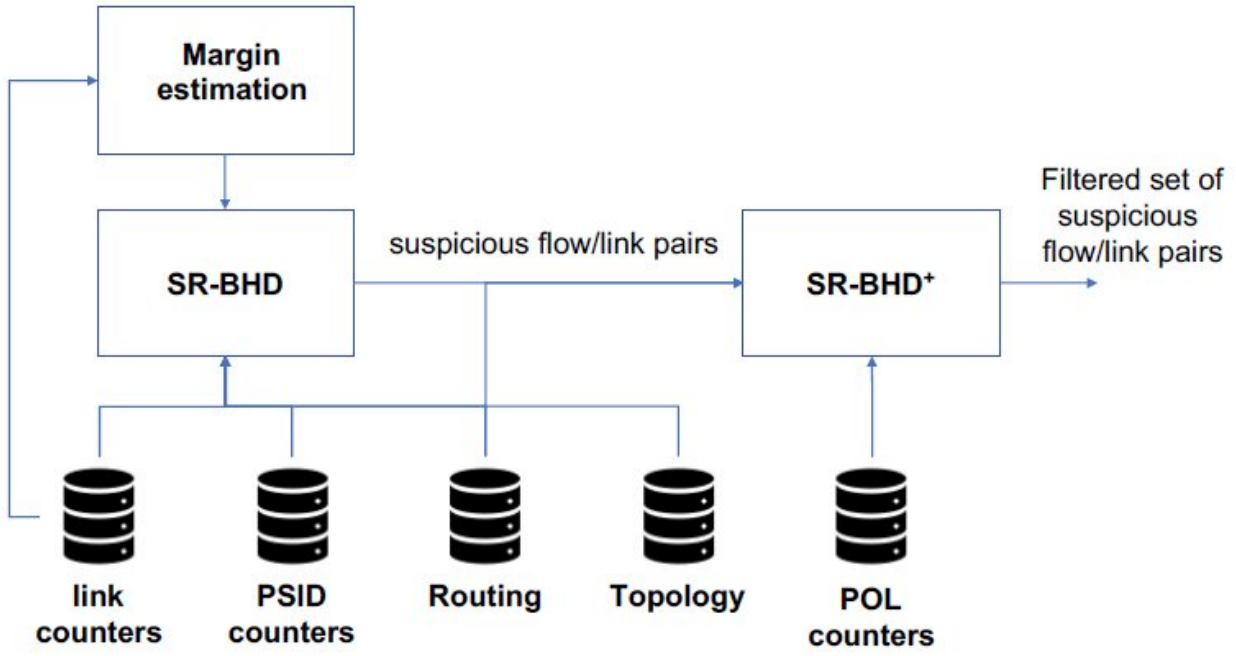


Figure 4.7: Scheme of the proposed monitoring framework

The next subparagraphs provide the insight of each of the functional blocks reported in Figure 4.7.

Algorithm 1 *SR-BHD* algorithm pseudo code.

Require: the network graph: $\mathcal{G} = (\mathcal{N}, \mathcal{L})$, the routing matrix: $\mathbf{R}$, the traffic counters: $\mathbf{y}_L$ and $\mathbf{y}_B$
1: initialize $\mathcal{L}_S \leftarrow \emptyset$, $\mathcal{F}_S \leftarrow \emptyset$ and $tmp \leftarrow \emptyset$
2: **for all** $l \in \mathcal{L}$ **do**
3: **if** Equation 4.5 for link l does not hold **then**
4: $\mathcal{L}_S \leftarrow l$
5: $tmp \leftarrow l.h$
6: **end if**
7: **end for**
8: **for all** $n \in tmp$ **do**
9: **for all** $a \in \mathcal{N}$ **do**
10: **if** Equation 4.6 for node n and active segment a does not hold **then**
11: $\mathcal{F}_S \leftarrow \{x_{i,e,c} : r_{i,e,c}(l) > 0, l \in \mathcal{L}_S \cap \delta_n^+ \wedge a \in \sigma_{i,e,c}\}$
12: **end if**
13: **end for**
14: **end for**
15: **return** the set of suspicious flows $\mathcal{F}_S$ and links $\mathcal{L}_S$.

The pseudo code of the *SR-BHD* is reported in Algorithm 1. It takes as input the network graph, the overlay routing matrix and the link count and PSID counter vectors. Then, in line 1 three data structures are initialized as empty: i) the two sets $\mathcal{L}_S$ and $\mathcal{F}_S$ that will contain the suspicious links and flows (i.e., those elements that could potentially be involved in a black hole), and ii) the set *tmp* that will store the head nodes of suspicious links. After that, one link at a time (lines $2-7$), the validity of Equation 4.5 is tested. In case the condition does not hold, then the current link is declared as suspicious and its head node is included in the *tmp* set (lines 4 and 5). Once the location of the potential *SR Black Hole* has been determined, the next step consists in the detection of the affected *SR* flow. This task is performed in lines 8-14. More precisely, taken a node in *tmp*, each

active segment is tested. The flow conservation law at node n for the current active segment a is checked (line 10). In case the condition does not hold, the set of suspicious flows is updated (line 11). In particular all the *SR* flows that pass through the suspicious link l having a as active segment are included in the set $\mathcal{F}_S$. Finally, the list of flows and links collected as suspicious black holes, are returned in line 15. It is clear that the previously presented implementation of *SR-BHD* works only in the ideal case that the only source of packet loss in the network is the presence of *SR Black Hole*. In an actual situation, the presence of other sources of packet loss would be interpreted by *SR-BHD* as the proof of the existence of a *SR Black Hole*, leading to the creation of a large number of false positives. To cope with the presence of other sources of packet loss we introduce a tolerance term in Equations 4.5 and 4.6, that is referred to as *margin*. Specifically, the margin of link l is indicated by the symbol $\mu(l)$. With this new logic, the flow conservation conditions are violated only in case the distance between the traffic that is supposed to be sent and the one that is actually transmitted is greater than the margin. The goal of the margin is to make *SR-BHD* robust with respect to sources of packet loss different from the targeted *SR Black Hole*. Considering the link l, the best value to use for the margin is to set it equal to the amount of traffic that is lost due to congestion, errors, etc. Unfortunately, this quantity is unknown in advance, so in the next we discuss a method for its estimation. The first step consists in building a dataset $\mathcal{D}$ of observations for each network link, considering a time horizon T. The hypothesis is that no *SR Black Hole* occurred during the creation of the dataset. The observation related to the link l at time t is represented by the tuple $< y_L(l)[t], C_l, y_E(l)[t] >$, which includes: i) the value of the link count on link l at time t, ii) the capacity of the link l, and iii) the amount of traffic dropped on link l at time t. The measurement of this quantity is performed by error counters widely deployed in current network cards, and that accounts for packets discarded due to different reasons (e.g., congestion, checksum errors, TTL expired, etc.). The first two parameters represent the features of the observation while the third one is the label. Once the dataset is built, it is used to train a Neural Network (NN) having the goal of learning the relation between the link utilization and the amount of traffic lost. The input layer of the NN is composed of two nodes, one receiving the value of the link count for the link l and the other one for its capacity, while the output layer contains a single node, reporting the targeted amount of traffic lost. This last quantity is used as the value for the margin.

4.5.4 Exploit POL Counters to improve the precision

In the next we introduce an enhancement of the proposed algorithm, named *SR-BHD*$^+$, that allows for a consistent improvement of the precision. In particular, the idea is to exploit the extra information provided by the POL counters to reduce the size of the set of suspicious flows. In fact, as previously explained, this type of traffic counter allows us to measure the volume of each *SR* flow. Starting from this information, by means of the Equation 4.7, it is possible to calculate the amount of traffic flowing over each network link in the ideal case ($\mathbf{y}_L^I$) of no packet loss occurred (either due to black holes or for other causes).

$$\mathbf{y}_L^I = \mathbf{R} \cdot \mathbf{y}_P \tag{4.7}$$

The goal is to determine the amount of traffic that is lost in the black hole, starting from the knowledge of the ideal link count vector. To do that, as first it is required to understand how the

packet loss events affect the traffic flows during their journey in the network. Specifically, the volume of a flow is influenced in two different ways by loss events: i) *localised* on a link, and ii) *cumulative* over the links that the flow has visited so far. Regarding the *localised* effect, it is due to the packet loss events happening on a single link, such as congestion, transmission errors, TTL expired, etc. With reference to the *cumulative* effect, it concerns the overall volume reduction that a given flow has experienced due to packet loss events *localised* in the links that the flow has already visited. To better explain the difference between these two contributions, we propose a simple example. Let us consider a flow having a volume of 100 that is routed over a cascade of four links l_1, l_2, l_3 and l_4. Furthermore, let us assume that 5 units of traffic are lost on each link due to congestion events. Then, with reference to the link count $y_L(l_3)$, its value is equal to 85, that is given by the initial volume minus the packet loss occurred at link $l3$ (that represents the *localised* contribution) and the sum of the packet loss happened in the already visited links (*cumulative* effect), i.e., l_1 and l_2.

To filter out the contribution of other sources of packet loss from the ideal link count of a generic link l, calculated through the Equation 4.7, we need to compensate for both the localized and the cumulative effects. With reference to the first one, it can be easily removed by means of the margin ($\mu(l)$), since it represents an estimation of the packet loss due to congestion occurring at link l. The latter effect can be cancelled by deleting from the ideal link count the value calculated according to the following Equation 4.8:

$$m(l) = \sum_{a \in \mathcal{N}} g_{i,a}(l) \cdot \left(\mathbf{y}_B^I(i,a) - \mathbf{y}_B(i,a)\right) \tag{4.8}$$

where the vector $\mathbf{y}_B^I$ contains the ideal values of the PSID counters, that can be calculated through the Equation 4.2. The intuition behind Equation 4.8 is that since PSID counters instantiated in a node allow to measure the overall amount of traffic that the node has received, then they can separate the *cumulative* contribution from the *localised* one. Then, the difference between the ideal value of the PSID counters and the actual one allows to estimate the amount of traffic that has been lost along the way (before reaching a given node).

Then, the vector representing the effects (in terms of packet loss) of the *SR Black Hole* over the network links is given by Equation 4.9.

$$\Delta_L = \mathbf{y}_L^I - \mu - \mathbf{m} - \mathbf{y}_L \tag{4.9}$$

The principle of the *SR-BHD*$^+$ algorithm is to reduce the set of suspicious flows by comparing the amount of traffic lost due to the black hole with the size of each *SR* flow. In case the two quantities are comparable, then the flow is suspected to be affected by an *SR Black Hole*. More in detail, considering an *SR* flow $x_{i,e,c}$ that is suspected to fall in a black hole located at link l, then *SR-BHD*$^+$ uses the condition reported in Equation 4.10 to determine if it is likely that the flow is actually affected by an *SR Black Hole*.

$$\frac{\Delta_L(l)}{y_P(i,e,c)} \in [s_{\min}, s_{\max}] \tag{4.10}$$

The numerator of the left term of the Equation 4.10 represents the estimated amount of traffic that is lost in a black hole, while the denominator is the volume of the suspected *SR* flow. The right side of the Equation 4.10 represents an interval that indicates the tolerance of *SR-BHD*$^+$. It

Algorithm 2 *SR-BHD*$^+$ algorithm pseudo code.

Require: $\mathbf{y}_L$, $\mathbf{y}_B$, $\mathbf{y}_P$, μ, $\mathbf{G}$, $\mathbf{B}$, $\mathbf{R}$, $s_{\min}$, $s_{\max}$, $\mathcal{F}_S$, $\mathcal{L}_S$
initialize $\mathcal{L}_S^+ \leftarrow \emptyset$ and $\mathcal{F}_S^+ \leftarrow \emptyset$
calculate $\mathbf{y}_B^I$ according to Equation 4.2
calculate $\mathbf{y}_L^I$ according to Equation 4.7
calculate $m(l)$ according to Equation 4.8
for all $l \in \mathcal{L}_S$ **do**
    calculate $\Delta(l)$ according to Equation 4.9
    for all $x_{i,e,c} \in \mathcal{F}_S$ **do**
        if $r_{i,e,c}(l) > 0$ **AND** Equation 4.10 is *true* **then**
            $\mathcal{F}_S^+ \leftarrow x_{i,e,c}$ and $\mathcal{L}_S^+ \leftarrow l$
        end if
    end for
end for
return the set of suspicious flows $\mathcal{F}_S^+$ and links $\mathcal{L}_S^+$.

is composed of two parameters, namely $s_{\min}$ and $s_{\max}$. In the performance evaluation it is shown how, by properly setting these two values, it is possible to make *SR-BHD*$^+$ robust also in critical situations (e.g., heavy congestion, margin estimation errors, etc.). The main steps regarding the *SR-BHD*$^+$ execution are summarized in Algorithm 2.

Before concluding it is important to highlight again that, the use of *SR-BHD*$^+$ requires the support of POL counters in network devices.

Chapter 5

Performance Evaluation

In this paragraph we conduct a performance evaluation based on simulations. As first the data set is described, then the results related to three different experiments are presented.

5.1 Data Set description

SR-BHD performance is tested over three different real networks taken from [27] : Abilene ($N = 12$, $L = 30$), Geant ($N = 22$, $L = 72$) and Germany ($N = 50$, $L = 176$). For these networks, real traffic matrices are available. The shortest path policy is considered for routing the traffic demands. It implies that the segment lists configured in the network contain only a single SID associated to the destination node. Each traffic flow defined in the traffic matrix is steered by means of an *SR Policy* installed at the source node. Capacity planning is performed as follows: i) the traffic matrix is routed according to the shortest path policy, and ii) each link is assigned with a capacity that is randomly (according to a uniform distribution) selected so that the utilization falls in the range $[50\%, 99\%]$. Packet loss due to congestion is simulated by dropping a percentage of the traffic flowing over a link. In particular the number of lost packets on each link l, indicated to as Q_l, is randomly selected according to a Normal distribution having mean proportional to the link utilization. Furthermore we introduce a congestion amplification factor (α_C) that allows to tune the level of packet loss due to congestion (by multiplying $Q(l)$ for the congestion amplification factor). Finally, the packet loss is shared among the flows routed over the link l, proportionally with respect to their intensity.

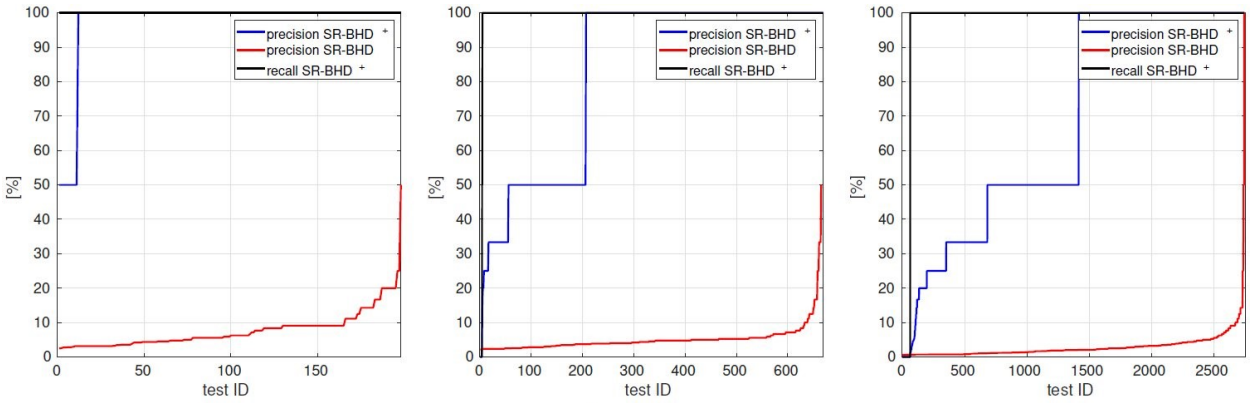
5.1.1 Experimental Evaluation

The first analysis we propose aims at evaluating the precision and the recall of the proposed algorithms in detecting the black holes. During the experiment we assume the presence of a single black hole. One at a time, all the possible couples link where the *MTU* violation occurs and the affected flow are tested. The congestion amplification factor is set equal to $\alpha_C = 10^{-3}$, and it is assumed that when a flow is affected by a black hole, all its packets are lost. Figure 5.1 shows the obtained results for different networks. In the figure, the x axis represents the different performed tests (i.e., a pair of link and flow involved in the black hole) sorted in increasing order with respect to the precision. In case of *SR-BHD*$^+$, the tolerance is set as follows: $s_{\min} = 0.97$ and $s_{\max} = 1.05$. We chose to use a narrow interval for the tolerance with the aim of maximizing the precision. Both the algorithms use the same neural network to assess the margin. Looking at the results reported in

Figure 5.1 it can be seen that the two methods (*SR-BHD* and *SR-BHD*$^+$) achieve a very high value of recall. In all the experiments *SR-BHD* obtained 100% of recall in all the networks for all possible configurations of link/flow pair involved in the black hole (for this reason we do not report it on the figures). This means that *SR-BHD* is always able to detect the presence of an *SR Black Hole*. Similar results are obtained for *SR-BHD*$^+$ in the Abilene network, whereas, for very few tests, in Geant and Germany networks, the algorithm did not correctly detect the black hole, i.e., the black hole does not appear among the suspected link/flow pairs returned as output. This is evidenced by the fact that, in these tests the recall is 0. This behavior is due to the estimation error in the evaluation of the amount of traffic lost in the black hole performed by the Equation 4.9. Anyway, the recall of *SR-BHD*$^+$ can be easily made equal to the one obtained by *SR-BHD* by tuning the width of the tolerance interval. Clearly this has a negative impact on the precision.

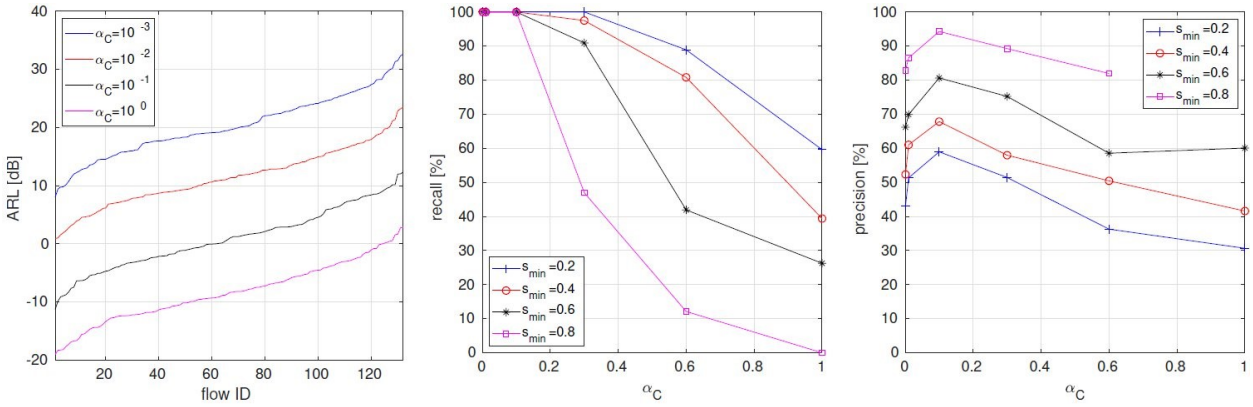
As far as the precision is concerned, results reported in figure 5.1 prove the huge performance improvement achieved by *SR-BHD*$^+$ due to the use of POL counters information. In particular, considering the Abilene network (Figure 5.1(a)) *SR-BHD*$^+$ achieves a mean value of the precision equal to 97.22, while the mean value obtained by *SR-BHD* is 7.71. The relevance of the improvement on the precision is better visible in the bigger networks (Geant and German). In fact, while in these situations *SR-BHD* basically suspects about all the flows passing over the link affected by the black hole (making impossible to fix the misconfiguration), *SR-BHD*$^+$ suspects at most 3 flows (33% of precision) in more than 90% of the cases. This improvement makes the proposed tool actually useful for troubleshooting in productive environments. The next two experiments represent a sensitivity analysis of *SR-BHD*$^+$. In the first one the aim is to test the ability of *SR-BHD*$^+$ to distinguish between regular packet loss events (caused by congestion, transmission errors, etc.) from anomalous packet loss events, due to the presence of the black hole. In the experiments, the amount of the regular packet loss events is tuned by changing the value of the congestion amplification factor (α_C). To give the intuition of the entity of the anomalous events with respect to the regular ones, we introduce the *Anomalous over Regular Losses* (ARL) ratio. In figure 5.2(a) it is shown the ARL value for the different flows, considering different values of α_C. In particular, for each flow we have calculated the ratio between its volume and the amount of packet loss due to congestion. Then the average value over all the links crossed by the flow is considered. Due to the large number of tests, only results for the Abilene network are shown. Figures 5.2(b) and 5.2(c) show the average recall and precision obtained by *SR-BHD*$^+$ as a function of the congestion amplification factor, for different values of the $s_{\min}$ tolerance parameter (the $s_{\max}$ is kept constant and equal to 1.1). The main outcomes of this analysis are three: i) the performance decreases as the α_C parameter increases, ii) by tuning the $s_{\min}$ tolerance parameter it is possible to make *SR-BHD*$^+$ be robust to the regular packet loss events, and iii) 100% of recall is feasible also for very low values of ARL (in the order of -10 dB). With reference to the first two points, the performed experiment has highlighted an interesting relation between the α_C parameter and the performance of the algorithm. On one hand, as expected the recall monotonically decreases as the congestion amplification factor increases 5.2(b)). In particular, by reducing $s_{\min}$ it is possible to keep the recall over the 90% also for high values of α_C, while keeping a good level of precision. For instance, the value $s_{\min} = 0.2$ allows for a 90% of recall when the congestion amplification factor is equal to 0.6, while having a precision of 38%. Consider that this level of precision implies that there are on average less than three suspected link/flow pairs. Then, although *SR-BHD*$^+$ does not provide the highest precision,

it drastically reduces the set of link/flow pairs that require further investigation.



(a) Abilene. (b) Geant. (c) Germany.

Figure 5.1: Precision and Recall analysis of the *SR-BHD* and *SR-BHD*$^+$ in different networks

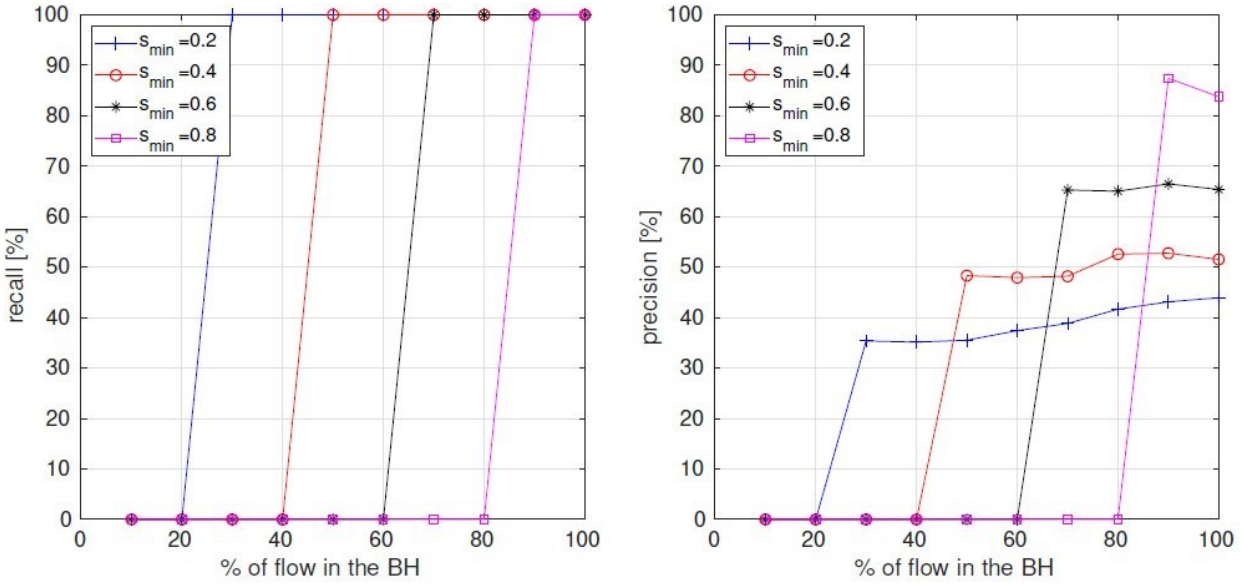


(a) ARL. (b) Recall. (c) Precision.

Figure 5.2: Sensitivity analysis of *SR-BHD*$^+$. Precision and recall as a function of the congestion level

A second interesting aspect that emerges from the performed experiment is the relation between the congestion amplification factor and the precision of *SR-BHD*$^+$ (Figure 5.2c). In particular, the precision does not monotonically decrease if α_C increases: i) initially the precision increases as the congestion level increases, then ii) once reached a maximum value it starts decreasing. This behavior is due to the working principle of *SR-BHD*$^+$, which selects the suspected flows by comparing their volume with the estimated amount of traffic that is lost in the black hole. If there are many flows having a comparable intensity, then the resulting precision of *SR-BHD*$^+$ is small. Also in the ideal case of no regular packet loss, if all the flows had the same volume, the resulting precision would be poor. In these situations, the presence of the regular packet loss creates a distance between the intensity of the different flows, helping *SR-BHD*$^+$ in correctly discriminating the flow that has actually fallen into the black hole. Clearly, when the congestion level overcomes a threshold value, the beneficial effect is lost and the performance starts to decrease as α_C increases. With reference to the third point (100% of recall is feasible also for very low values of ARL), *SR-BHD*$^+$ has shown a high tolerance to the presence of regular packet loss events on the traffic counters. In particular, by looking at figure 5.2(b) it is evident that the highest value of α_C at which *SR-BHD*$^+$ gets 100% of recall is 0.3. From Figure 5.2a it can be seen that, for such a value of congestion amplification factor, the lowest level of ARL reached is of the order of -10 dB. It means that *SR-BHD*$^+$ is able

to correctly determine what flow is lost in the black hole, also in theiscase its volume is 10 times smaller than the level of packet lost due to congestion. Furthermore, it is worth highlighting that, in these circumstances, *SR-BHD*$^+$ is still able to obtain an acceptable precision (see Figure 5.2(c)). In particular, for $\alpha_C = 0.3$ and $s_{\min} = 0.2$ the average precision is close to 50%, meaning that the suspicious link/flow pairs in output are 2. The last analysis we propose aims at evaluating the precision and the recall of *SR-BHD*$^+$ if only a portion of a flow is lost in the black hole. In fact an *SR* flow represents the aggregation of many application flows, then it can happen that only a subset of application flows falls into the black hole.



(a) Recall. (b) Precision.

Figure 5.3: Sensitivity analysis of *SR-BHD*$^+$. Precision and recall as a function of the percentage of flow that gets lost in the *SR Black Hole*

The result of this sensitivity analysis is reported in figures 5.3(a) and 5.3(b). In particular, the two figures show the precision and recall of *SR-BHD*$^+$ as a function of the percentage of flow that gets lost in the *SR Black Hole*, for different values of $s_{\min}$ parameter ($s_{\max}$ is kept constant and equal to 1.1). The main outcome of the performed experiment is that *SR-BHD*$^+$ is able to handle this situation. In fact, also in case only 30% of a flow is lost in the black hole, the obtained recall is 100% (meaning that it is always correctly detected) and the precision is approximately 33%, i.e., on average there are 3 link/flow suspicious pairs.

5.2 Performance evaluation

In this paragraph a description of the experimental prototype of *SR-BHD* that we have realized is provided. The goal of the prototype is to prove the effectiveness of the proposed approach. Three different aspects are discussed:

- the description of the design implementation of *SR-BHD* prototype,
- the methodology adopted to run the tests,
- the results obtained in the different types of performed tests.

5.2.1 Prototype description

The proposed *SR-BHD* algorithm is intended to be integrated on a centralized monitoring system, according to the *SDN* paradigm. In the prototype this task is accomplished by a bundle of scripts whose execution is triggered with a timer based mechanism. By a logical point of view, the prototype is composed by two main building blocks: i) the Stats Collector module, which is in charge of collecting the *SRTCs* in each node, and ii) the Black Hole Detection module, that implements the logic of *SR-BHD* on each link to detect potential black holes. Concerning the Stats Collector module, it is implemented as a bash script that automatically queries the network routers to get the required traffic statistics. On the contrary, the Black Hole Detection module is a Python script that takes as input the traffic measurements and verifies the validity of the equations that are at the basis of *SR-BHD* approach. This module requires the presence of a *configuration file* that specifies the routing, i.e., the segment lists that are enforced on packets in the data plane as well as the underlay paths, and the value of the margin.A *timer* component triggers the execution of the two blocks. In our prototype implementation, *VPP* virtual routers are used to instantiate an *SRv6* capable network. With respect to the functionality available in an ideal *SRv6* router, *VPP* lacks the presence of the full suite of *SRTCs*. In particular, the availability of traffic counters in *VPP* is as follows: i) INT counters, i.e., interface level traffic counters that account the number of packets TX/RX to/from each interface, ii) POL counters do not exist, and iii) PSID counters are updated only in case an *SR* operation is performed. With reference to the second point, it forced us to prototype *SR-BHD* instead of *SR-BHD*$^+$. Regarding the last point, it represents a critical difference for the actuation of the proposed *SR-BHD* framework. In fact, it relies on statistics coming from PSID counters that are collected at each node. On the contrary, in *VPP* this type of SRTC counters are maintained only at nodes that perform *SR* related operation (e.g., a router that accounts the packets on which it applies the END operation). In order to finalize our prototype, we have applied the strict source routing policy, i.e., for each flow the segment list contains the explicit set of intermediate nodes to go through. As a future step we aim at implementing the PSID counters on every node (regardless the application of *SR* functions).

5.2.2 Adopted Methodology

The reference topology used in the conducted experiment is shown in figure 4.1, with the only difference that in the performed experiments the initial *MTU* is set equal to 1500 Bytes in all the links. Nodes $E1$ and $E2$ represent the edge of the considered *SR* domain and implement *SR Policy* enforcement on the incoming traffic flows. The main features (source, destination, path) of the four traffic flows included in the scenario are reported in Tab. 5.1. These are generated by using the traffic generator included in *VPP*, that allows to create constant bit rate UDP flows, with the possibility to decide the packet size and the throughput. All the traffic flows have the same data rate of 100 Kbps

In each test a target flow and the link where the black hole occurs are selected. In order to create the black hole in the desired link and to hit the target flow, two actions are performed: i) packets of the target flow are generated with a higher size with respect to those of the other flows, and ii) the *MTU* of the link where the black hole happens is reduced compared with those associated to of the other links. Then, all the traffic flows are simultaneously started and ended 120

Source	Destination	Segment List
a::1	b::1	C1,C2,E2
a::2	b::2	C1,C3,C4,C2,E2
b::1	a::1	C4,C3,E1
b::2	a::2	C4,C2,C1,C3,E1

Table 5.1: Main features of the traffic flows included in the emulated

seconds later. The *Stats Collector module* is triggered every 60 seconds, meaning that in each run the *Black Hole Detection module* is executed 2 times. Furthermore, in order to account for possible discrepancy among traffic counters, an extra execution of *SR-BHD* is repeated after 130 seconds from the beginning of the test. The measurements obtained with the last collection phase allow to test the performance of *SR-BHD* in an ideal condition, i.e., when there is a perfect synchronization in the gathered statistics.

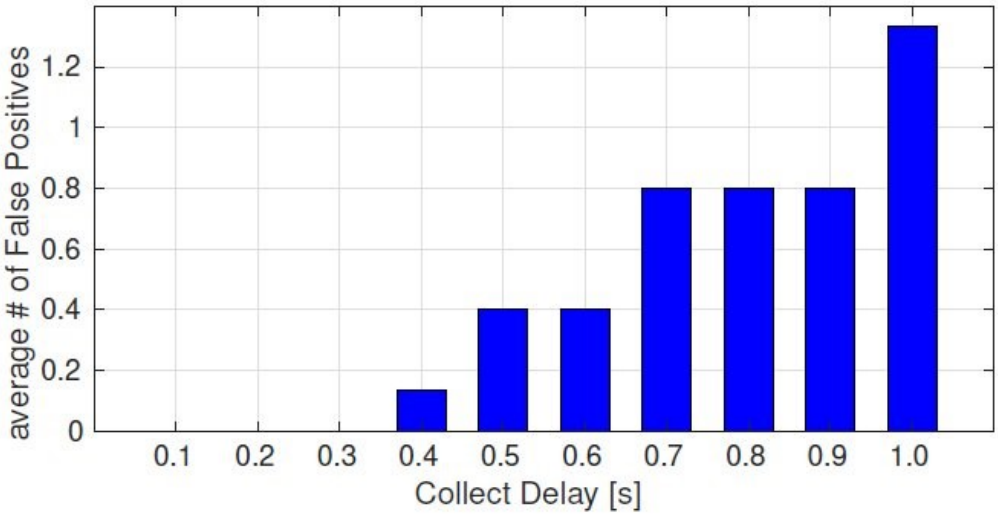


Figure 5.4: Average false positives as a function of the CD parameter

In order to carry out a sensitivity analysis, instead of considering congestion events, we have defined two tuning parameters to regulate the level of the misalignment introduced on the traffic counters. As a consequence of this, the value of the margin is manually set (no Neural Network is used to set its value). The first tuning parameter is referred to as *Target Rate Ratio* (TRR) being a scaling parameter that allows to reduce the intensity of the target flow with respect to the others. The TRR is selected in the range $(0, 1]$. As an example, assuming that all the generated flows have a throughput of 100 Kbps and the TRR= 0.5, then the target flow is generated with a lower throughput, equal to 50 Kbps. The second tuning parameter is the *Collect Delay* (CD) and represents a synthetic delay that is introduced between the counter gathering among different nodes. In particular, assuming that the monitoring system sequentially queries the nodes (one at a time) for the traffic statistics, the CD parameter represents the time interval between two consecutive queries. By tuning this parameter we are able to synthetically introduce a discrepancy on the counter values of different nodes. Since Equation 4.6 used by *SR-BHD* to enforce the flow conservation principle considers traffic counters instantiated in different nodes, the discrepancy introduced by the CD can lead to the violation of this condition, thus leading to the creation of false positives. As an example,

considering that the link capacity is set to 100 Kbps, a CD of 0.1 seconds creates a mismatch between the counters of neighbor nodes ranging between 1250 Bytes (in the best case that the counters of the two nodes are queried one after the other) and 7500 Bytes (in the worst case that one node is the first to be queried and the other is the last in the whole network). Regular packet loss events have, on the traffic counters, an effect similar to the one caused by the CD. Then, in the experiments we emulate the presence of different sources of packet loss by exploiting the CD, since it offers the possibility to precisely tune the regular loss level.

5.2.3 Conducted Experiments

Three different experiments have been performed to assess the effectiveness of the implemented prototype in the correct detection of the *SR Black Holes*. Before commenting the main obtained results, we mention that, as expected, when *SR-BHD* has been tested in ideal conditions (CD= 0, TRR= 1 and counters collected at $t = 130$ seconds), we have obtained as expected, the maximum values of precision and recall for all possible combination of target flow and affected link.

The aim of the first analysis is to show the impact of CD on the performance of *SR-BHD*. The CD introduces a discrepancy on the value of the traffic counters instantiated in different nodes, which could turn in large errors in the equations that regulate the logic of *SR-BHD*. For this reason the impact of the CD is evaluated in terms of false positive (a flow is erroneously detected as falling into a black hole) that it produces in the output of *SR-BHD*. In fact, the discrepancy between counters of neighbor nodes can be high enough to overcome the value of the margin, thus generating a false positive. The result of this analysis is reported in figure 5.4.

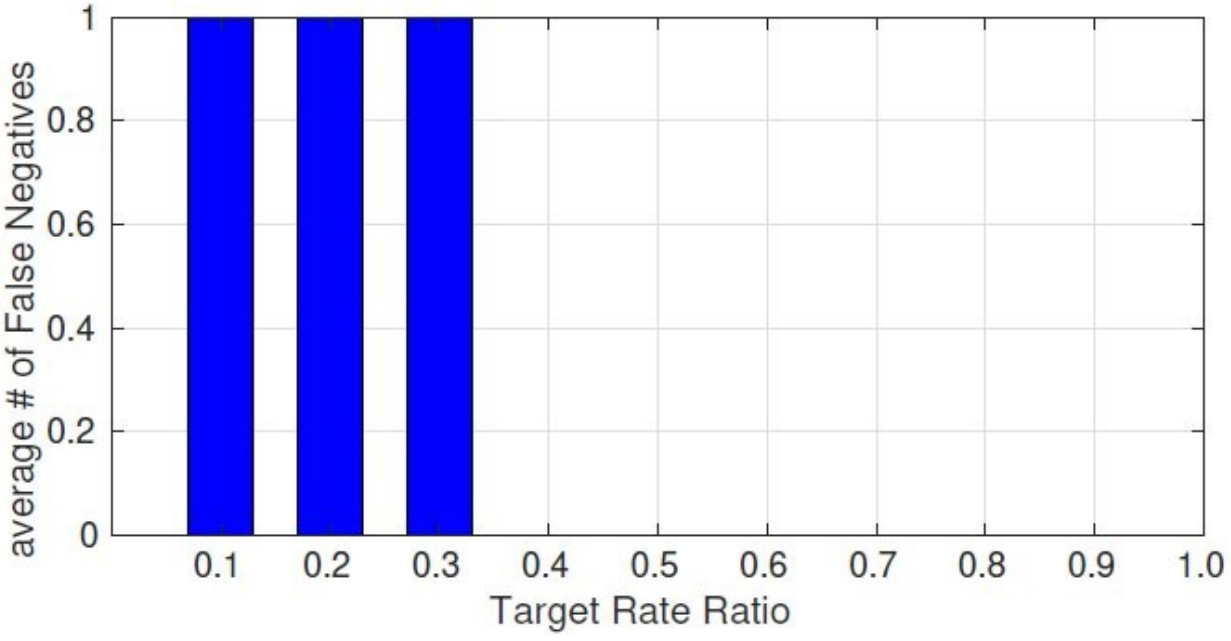


Figure 5.5: Average false negatives as a function of the TRR parameter

The setup for the experiment is as follows: i) results are averaged over all possible combinations of target flows and affected link, and over all the counters collection periods, ii) the TRR parameter is set to 1, iii) the measurements related to $t = 130$ seconds are neglected, and iv) the margin is equal to the 1% of the flow rate. As expected, when the value of the CD parameter increases *SR-BHD* erroneously detects some traffic flows to be affected by a black hole. In particular, in

case of CD= 1 there are on average 1.4 false positives over 16 possibilities. Clearly this situation is extreme, since it implies that in the worst case there are 6 seconds of difference among the time at which the first and the last nodes are queried for the traffic stats. We expect that possible query time interval is much shorter in actual networks. Finally, in the results shown in figure 5.4 the margin is kept constant during all the different experiments. As it is discussed later, false positives can be completely removed by properly adjusting the margin to the level of the regular packet loss.

The second analysis aims at determining the impact of the TRR parameter on the ability of *SR-BHD* to detect black holes. Specifically, the TRR has as main effect a reduction of the data rate of the target flow. Thus, for low values of this parameter, the intensity of the target flow could become comparable with the margin, consequently, *SR-BHD* could not be able to correctly detect Black Holes, generating a false negative. The result of this analysis is reported in figure 5.5, that has been obtained considering the CD= 0 and the margin equal to the 30% of the flow rate.

Looking at figure 5.5 it is evident that, when the TRR decreases below a threshold value, the volume of the target flow is low with respect to the margin. Consequently, *SR-BHD* neglects the discrepancy among the traffic counters caused by the black hole (since this last is smaller than the margin), thus leading to a false negative. As expected, this threshold value for the TRR is equal to 0.3, that is the value chosen to set the margin. Despite the obtained result could seem straightforward, it allows us to assess an important property of *SR-BHD*. In particular, the ability of detecting a target flow falling into a black hole is not related to the ratio between the intensity of the target flow with respect to the other ones, but only with the respect to the value of the margin. This last is a tuning parameter of *SR-BHD*, then it is always possible to set it to a value that completely avoids the presence of false negatives.

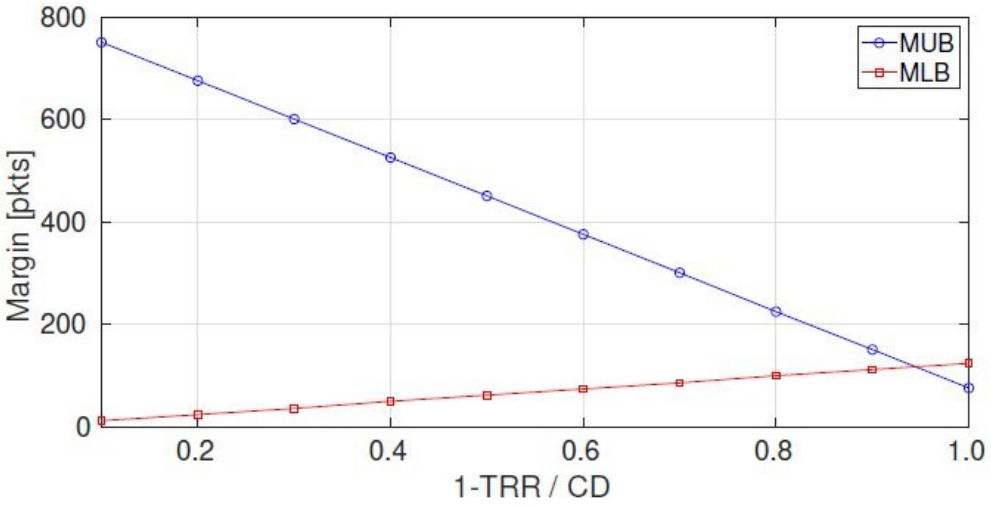


Figure 5.6: Analysis of the values of the margin that allows for the complete

On the basis of the last consideration, we investigate how to choose the proper value of the margin. The aim is to show that in most of the considered cases there exists a value of the margin such that it is possible to completely avoid false positives and false negatives, while correctly detecting the existing black hole. The obtained result is shown in figure 5.6. The blue line represents the Margin Upper Bound (MUB), which is the maximum value that the margin can assume while avoiding the presence of false negatives. This line is obtained by running the tests with the variable TRR and keeping the CD= 0. Considering the blue line, in figure 5.6 the x-axis represents the

parameter $(1.1 - \text{TRR})$, and the y-axis reports the value of the MUB. If the margin is set equal to the MUB, the black hole cannot be detected, thus leading to the creation of a false negative. All values of the margin that are below the blue line allow for a complete cancellation of false negatives.

On the contrary, the red line represents the Margin Lower Bound (MLB), i.e., the minimum value of the margin that allows for the complete cancellation of the effects due to regular packet loss, thus avoiding the presence of false positives. This line is obtained by running the tests with the variable CD and keeping the TRR$= 1$. Considering the red line, in Fig. 5.6 the x-axis represents the CD parameter, and the y-axis reports the value of the LBM. All values of the margin above the red line allow for a complete cancellation of the false positives.

In figure 5.6 by moving to the right on the x-axis the ARL decreases. As a consequence, in order to filter the contribution of regular packet loss, higher values of the margin are required (see the red line). Unfortunately, when the margin increases, the sensitivity of *SR-BHD* decreases, since the effect of a small flow falling into a black hole creates a discrepancy in the traffic counters that do not allow a correct detection. In figure 5.6, the area that is below the blue line and above the red line represents the setting for the margin that allows for a complete cancellation of the false positives and false negatives. Interestingly, in almost all the situations (TRR and CD) it is possible to find a value of the margin that makes *SR-BHD* to get the best possible performance in terms of recall and precision. Only in case of TRR$= 0.1$ the cancellation of the effects of the regular packet loss causes the creation of a false negative. Particularly, this happens in the case of CD $\in (0.7, 1)$. In fact, looking at figure 5.6 it can be seen that the value assumed by the blue line in TRR$= 0.1$ is smaller than the value assumed by the red line when CD $\in (0.7, 1)$. To conclude, the last presented analysis has highlighted that the range of margin values that allow for a perfect detection of the black holes is related to ARL value.

Chapter 6

Conclusion

In this paper we have addressed the problem of logical failures in Segment Routing networks due to the violation of the MTU constraint, named *SR Black Hole*. We have experimentally proven that: i) in misconfigured SR domains *SR Black Holes* can occur, and ii) classical detection methods based on active probing fail to detect the failure. Then, by exploiting specific SR Traffic Counters we have introduced a passive monitoring framework, named *SR-BHD*, that is able to detect the presence of black holes, also in presence of multiple sources of packet loss (e.g., congestion, transmission errors, etc.). In particular *SR-BHD* imposes the flow conservation principle at different levels in the network, being able to tolerate the presence of other sources of packet loss (e.g., congestion, transmission error, etc.) by the introduction of a safety margin. An estimation framework based on the use of Neural Networks is presented as a method to select the proper value for the margin. The framework has further been extended in case specific traffic counters (POL counters) are available in SR capable nodes. The enhanced version of *SR-BHD*, named *SR-BHD*$^+$, allows to greatly improve the precision in the detection of the black hole. Specifically, a tolerance interval can be set to make *SR-BHD*$^+$ robust against critical situations, such the case of high level of packet loss due to congestion, or the case in which only a portion of a traffic flow is lost in the black hole. The performance evaluation has shown how, by properly tuning the tolerance interval of *SR-BHD*$^+$, it is possible to find a good trade off between the precision of the algorithm and its robustness with respect to the aforementioned critical situations. Finally, a prototype of *SR-BHD* has been realized and tested over an emulated environment. The conducted experiments have confirmed the effectiveness of the proposed framework in detecting the presence of *SR Black Holes*. As future work we aim to design a tool to help Network Operators to configure their SR domain by avoiding the creation of *SR Black Holes*.

As future works we aim at design a tool to help Network Operators to configure their SR domain by avoiding the creation of SR Black Holes, and the integration of the full suite of SR traffic counters in VPP and others programmable data planes.

Bibliography

[1] CSIT REPORT - The Fast Data I/O Project (FD.io) Continuous System Integration and Testing (CSIT) project report for CSIT master system testing of VPP-18.04 release.

[2] Z. Ali, K. Talaulikar, C. Filsfils, N. K. Nainar, and C. Pignataro. Bidirectional Forwarding Detection (BFD) for Segment Routing Policies for Traffic Engineering. Internet-Draft draft-ali-spring-bfd-sr-policy-07, Internet Engineering Task Force, May 2021. Work in Progress.

[3] F. Aubry, D. Lebrun, S. Vissicchio, M. T. Khong, Y. Deville, and O. Bonaventure. Scmon: Leveraging segment routing to improve network monitoring. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[4] J. BOSMA, B. OVEREINDER, and W. TOOROP. Discovering path mtu black holes on the internet using ripe atlas. 2012.

[5] M. de Boer and J. Bosma. Discovering path mtu black holes in the internet using ripe atlas, 2012.

[6] Y. Desmouceaux, M. Townsley, and T. H. Clausen. Zero-loss virtual machine migration with ipv6 segment routing. In *2018 14th International Conference on Network and Service Management (CNSM)*, pages 420–425. IEEE, 2018.

[7] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proceedings of the 2007 ACM CoNEXT conference*, pages 1–12, 2007.

[8] L. Fang, A. Atlas, F. Chiussi, K. Kompella, and G. Swallow. Ldp failure detection and recovery. *IEEE Communications magazine*, 42(10):117–123, 2004.

[9] C. Filsfils, Z. Ali, M. Horneffer, D. Voyer, M. Durrani, and R. Raszuk. Segment Routing Traffic Accounting Counters. Internet-Draft draft-filsfils-spring-sr-traffic-counters-01, Internet Engineering Task Force, Apr. 2021. Work in Progress.

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

[11] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, and daniel.voyer@bell.ca. IPv6 Segment Routing Header (SRH). Internet-Draft draft-ietf-6man-segment-routing-header-21, Internet Engineering Task Force, June 2019. Work in Progress.

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

[13] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. Segment Routing Architecture. RFC 8402, July 2018.

[14] R. Geib, C. Filsfils, C. Pignataro, and N. K. Nainar. A Scalable and Topology-Aware MPLS Data-Plane Monitoring System. RFC 8403, July 2018.

[15] G. Iannaccone, C.-N. Chuah, S. Bhattacharyya, and C. Diot. Feasibility of ip restoration in a tier 1 backbone. *Ieee Network*, 18(2):13–19, 2004.

[16] G. Iannaccone, C.-n. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an ip backbone. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 237–242, 2002.

[17] A. Itai. Two-commodity flow. *Journal of the ACM (JACM)*, 25(4):596–611, 1978.

[18] D. Katz and D. Ward. Bidirectional forwarding detection (bfd). Technical report, 2010.

[19] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*, pages 2180–2188. IEEE, 2007.

[20] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: illuminating the edge network. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 246–259. ACM, 2010.

[21] N. Kumar, C. Pignataro, G. Swallow, N. Akiya, S. Kini, and M. Chen. Label Switched Path (LSP) Ping/Traceroute for Segment Routing (SR) IGP-Prefix and IGP-Adjacency Segment Identifiers (SIDs) with MPLS Data Planes. RFC 8287, Dec. 2017.

[22] S. Litkowski, A. Bashandy, C. Filsfils, P. Francois, B. Decraene, and D. Voyer. Topology Independent Fast Reroute using Segment Routing. Internet-Draft draft-ietf-rtgwg-segment-routing-ti-lfa-07, Internet Engineering Task Force, June 2021. Work in Progress.

[23] P. Loreti, A. Mayer, P. Lungaroni, F. Lombardo, C. Scarpitta, G. Sidoretti, L. Bracciale, M. Ferrari, S. Salsano, A. Abdelsalam, et al. Srv6-pm: A cloud-native architecture for performance monitoring of srv6 networks. *IEEE Transactions on Network and Service Management*, 18(1):611–626, 2021.

[24] M. Luckie, K. Cho, and B. Owens. Inferring and debugging path mtu discovery failures. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*. USENIX Association, 2005.

[25] M. Mardani and G. B. Giannakis. Estimating traffic and anomaly maps via network tomography. *IEEE/ACM transactions on networking*, 24(3):1533–1547, 2015.

[26] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, and C. Diot. Characterization of failures in an ip backbone. In *IEEE INFOCOM 2004*, volume 4, pages 2307–2317. IEEE, 2004.

[27] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly. SNDlib 1.0–Survivable Network Design Library. In *Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium*, April 2007.

[28] M. Polverini, A. Cianfrani, and M. Listanti. Snoop through traffic counters to detect black holes in segment routing networks. In *International Conference on Communications and Networking in China*, pages 337–350. Springer, 2020.

[29] M. Polverini, A. Cianfrani, and M. Listanti. A theoretical framework for network monitoring exploiting segment routing counters. *IEEE Transactions on Network and Service Management*, 17(3):1924–1940, 2020.

[30] R. Potharaju and N. Jain. An empirical analysis of intra-and inter-datacenter network failures for geo-distributed services. In *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*, pages 335–336, 2013.

[31] R. Potharaju and N. Jain. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–17, 2013.

[32] R. Staff. Ripe atlas: A global internet measurement network. *Internet Protocol Journal*, 18(3), 2015.

[33] M. Thottan and C. Ji. Anomaly detection in ip networks. *IEEE Transactions on signal processing*, 51(8):2191–2204, 2003.

[34] A. Tulumello, A. Mayer, M. Bonola, P. Lungaroni, C. Scarpina, S. Salsano, A. Abdelsalam, P. Camarillo, D. Dukes, F. Clad, et al. Micro sids: a solution for efficient representation of segment ids in srv6 networks. In *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1–10. IEEE, 2020.

[35] P. L. Ventre, S. Salsano, M. Polverini, A. Cianfrani, A. Abdelsalam, C. Filsfils, P. Camarillo, and F. Clad. Segment routing: A comprehensive survey of research activities, standardization efforts and implementation results. *IEEE Communications Surveys & Tutorials*, 2020.

[36] R. Zhang-Shen and N. McKeown. Designing a predictable internet backbone network. HotNets, 2004.

Appendix A

Appendix

A.1 VPP installation

apt-get update
cd /etc/apt/sources.list.d/
cat> 99fd.io.list
deb [trusted = yes] https://packagecloud.io/fdio/release/ubuntu bionic main deb [trusted = yes]
http://cz.archive.ubuntu.com/ubuntu bionic main universe deb [trusted = yes]
https : //nexus.fd.io/content/repositories/fd.io.ubuntu.xenial.main/. /
deb [trusted = yes]
https://nexus.fd.io/content/repositories/fd.io.stable.1804.ubuntu.xenial.main/ ./
deb [trusted = yes]
https://nexus.fd.io/content/repositories/fd.io.master.ubuntu.xenial.main/ ./ curl
-L https://packagecloud.io/fdio/release/gpgkey | sudo apt-key add - apt-get
update
apt-get install vpp vpp-plugin-core vpp-plugin-dpdk

A.2 Python script for executing commands

import subprocess
while True:
print ("Choose option:")
print ("0) Create topology (running VPP instances, configuring link and SRv6).")
print ("1) TCP traffic (top-policy (a1: :)).")
print ("2) ICMPv6 traffic (down-policy (a3: :)).") print
("3) Simultaneous TCP and ICMPv6 traffic.") print
("4) Enable Recovery Policy.")
print ("5) Disable Recovery Policy.") print
("6) MTU Path Discovery (TCP).")
print ("9) Exit and kill VPP instances & delete interfaces.") print ("10)
Exit without killing VPP instances & deleting interfaces.") option =

```
input ("Option:")
if str (option) == "0":
subprocess.call (["./ 0-create-topology / 0-0-main.sh"])
print ("Topology has been configured.")
print ("")
elif str (option) == "1":
subprocess.call (["./ 1-TCP-traffic / tcp.sh"])
print ("")
elif str (option) == "2":
subprocess.call (["./ 2-ICMPv6-traffic / 2-1-ping.sh"])
print ("")
elif str (option) == "3":
subprocess.call (["./ 3-TCP-ICMP / enable-flow.sh"])
print ("")
elif str (option) == "4":
subprocess.call (["./ 1-1-enable-rec-policy.sh"])
print ("")
elif str (option) == "5":
subprocess.call (["./ 1-2-disable-rec-policy.sh"])
print ("")
elif str (option) == "6":
subprocess.call (["./ 6-MTU-Path-Discovery.sh"])
print ("")
elif str (option) == "9":
subprocess.call (["./ 9-exit-kill-VPP.sh"])
print ("")
break
else:
print ("")
break
```

A.3 File '0-0-main.sh'

```
#! / usr / bin / sudo bash

    # Close VPP instances and delete links
sh ./9-exit-kill-VPP.sh

    function pause () {
read -p "$ *"
}
```

```
    # Starting VPP instances
sudo vpp unix {cli-listen /run/vpp/cli-vppE1.sock} api-segment {prefix vppE1}
# pause 'E1 instance starts. Press enter. '
sudo vpp unix {cli-listen /run/vpp/cli-vppC1.sock} api-segment {prefix vppC1}
# pause 'C1 instance starts. Press enter. '
sudo vpp unix {cli-listen /run/vpp/cli-vppC2.sock} api-segment {prefix vppC2}
# pause 'C2 instance starts. Press enter. '
sudo vpp unix {cli-listen /run/vpp/cli-vppE2.sock} api-segment {prefix vppE2}
# pause 'E2 instance starts. Press enter. '
sudo vpp unix {cli-listen /run/vpp/cli-vppC3.sock} api-segment {prefix vppC3}
# pause 'C3 instance starts. Press enter. '
sudo vpp unix {cli-listen /run/vpp/cli-vppC4.sock} api-segment {prefix vppC4}
# pause 'C4 instance starts. Press enter. '
pause 'E1, E2, C1, C2, C3, C4 instances starts. Press enter. '

    # Linking VPP instances
sh ./0-create-topology/0-1-linking-vpp-instances.sh
pause 'VPP instances linked. Press enter. '

    # Creating hosts instances
sh ./0-create-topology/0-2-creating-hosts.sh pause 'Hosts
created and linked to router. Press enter. '

    # Setting SRv6
sh ./0-create-topology/0-3-setting-SRv6.sh pause
'BSID and Policy configured. Press enter. '
```

A.4 File '0-1-linking-vpp-instances.sh'

```
#! / usr / bin / sudo bash
# This file is used to create connections between routers and set routes for adjacent
IPs.

    # Set up all connections
# E1C1 connection
ip link add name E1C1-E1 type veth peer name E1C1-C1
vppctl -s /run/vpp/cli-vppE1.sock create host-interface name E1C1-E1
vppctl -s /run/vpp/cli-vppC1.sock create host-interface name E1C1-C1

    vppctl -s /run/vpp/cli-vppE1.sock set int state host-E1C1-E1 up
vppctl -s /run/vpp/cli-vppC1.sock set int state host-E1C1-C1 up
vppctl -s /run/vpp/cli-vppE1.sock set int ip address host-E1C1-E1 fc00 :: 1: 0: 1/112 vppctl
```

```
-s /run/vpp/cli-vppC1.sock set int ip address host-E1C1-C1 fc00 :: 1: 0: 2/112

    vppctl -s /run/vpp/cli-vppE1.sock set interface mtu 1500 host-E1C1-E1
vppctl -s /run/vpp/cli-vppC1.sock set interface mtu 1500 host-E1C1-C1
# C1C2 connection
ip link add name C1C2-C1 type veth peer name C1C2-C2
vppctl -s /run/vpp/cli-vppC1.sock create host-interface name C1C2-C1
vppctl -s /run/vpp/cli-vppC2.sock create host-interface name C1C2-C2

    vppctl -s /run/vpp/cli-vppC1.sock set int state host-C1C2-C1 up
vppctl -s /run/vpp/cli-vppC2.sock set int state host-C1C2-C2 up
vppctl -s /run/vpp/cli-vppC1.sock set int ip address host-C1C2-C1 fc00 :: 2: 0: 1/112 vppctl
-s /run/vpp/cli-vppC2.sock set int ip address host-C1C2-C2 fc00 :: 2: 0: 2/112

    vppctl -s /run/vpp/cli-vppC1.sock set interface mtu 1500 host-C1C2-C1
vppctl -s /run/vpp/cli-vppC2.sock set interface mtu 1500 host-C1C2-C2
# C2E2 connection

    ip link add name C2E2-C2 type veth peer name C2E2-E2
vppctl -s /run/vpp/cli-vppC2.sock create host-interface name C2E2-C2
vppctl -s /run/vpp/cli-vppE2.sock create host-interface name C2E2-E2

    vppctl -s /run/vpp/cli-vppC2.sock set int state host-C2E2-C2 up
vppctl -s /run/vpp/cli-vppE2.sock set int state host-C2E2-E2 up
vppctl -s /run/vpp/cli-vppC2.sock set int ip address host-C2E2-C2 fc00 :: 3: 0: 1/112 vppctl
-s /run/vpp/cli-vppE2.sock set int ip address host-C2E2-E2 fc00 :: 3: 0: 2/112

    vppctl -s /run/vpp/cli-vppC2.sock set interface mtu 1500 host-C2E2-C2
vppctl -s /run/vpp/cli-vppE2.sock set interface mtu 1500 host-C2E2-E2
# E1C3 connection
ip link add name E1C3-E1 type veth peer name E1C3-C3
vppctl -s /run/vpp/cli-vppE1.sock create host-interface name E1C3-E1
vppctl -s /run/vpp/cli-vppC3.sock create host-interface name E1C3-C3

    vppctl -s /run/vpp/cli-vppE1.sock set int state host-E1C3-E1 up
vppctl -s /run/vpp/cli-vppC3.sock set int state host-E1C3-C3 up
vppctl -s /run/vpp/cli-vppE1.sock set int ip address host-E1C3-E1 fc00 :: 6: 0: 1/112 vppctl
-s /run/vpp/cli-vppC3.sock set int ip address host-E1C3-C3 fc00 :: 6: 0: 2/112

    vppctl -s /run/vpp/cli-vppE1.sock set interface mtu 1500 host-E1C3-E1
vppctl -s /run/vpp/cli-vppC3.sock set interface mtu 1500 host-E1C3-C3
# Connection C1C3
ip link add name C1C3-C1 type veth peer name C1C3-C3
```

```
vppctl -s /run/vpp/cli-vppC1.sock create host-interface name C1C3-C1
vppctl -s /run/vpp/cli-vppC3.sock create host-interface name C1C3-C3

    vppctl -s /run/vpp/cli-vppC1.sock set int state host-C1C3-C1 up
vppctl -s /run/vpp/cli-vppC3.sock set int state host-C1C3-C3 up
vppctl -s /run/vpp/cli-vppC1.sock set int ip address host-C1C3-C1 fc00 :: 7: 0: 1/112 vppctl
-s /run/vpp/cli-vppC3.sock set int ip address host-C1C3-C3 fc00 :: 7: 0: 2/112

    vppctl -s /run/vpp/cli-vppC1.sock set interface mtu 1500 host-C1C3-C1
vppctl -s /run/vpp/cli-vppC3.sock set interface mtu 1500 host-C1C3-C3
# C3C4 connection
ip link add name C3C4-C3 type veth peer name C3C4-C4
vppctl -s /run/vpp/cli-vppC3.sock create host-interface name C3C4-C3
vppctl -s /run/vpp/cli-vppC4.sock create host-interface name C3C4-C4

    vppctl -s /run/vpp/cli-vppC3.sock set int state host-C3C4-C3 up
vppctl -s /run/vpp/cli-vppC4.sock set int state host-C3C4-C4 up
vppctl -s /run/vpp/cli-vppC3.sock set int ip address host-C3C4-C3 fc00 :: 5: 0: 2/112 vppctl
-s /run/vpp/cli-vppC4.sock set int ip address host-C3C4-C4 fc00 :: 5: 0: 1/112

    vppctl -s /run/vpp/cli-vppC3.sock set interface mtu 1400 host-C3C4-C3
vppctl -s /run/vpp/cli-vppC4.sock set interface mtu 1400 host-C3C4-C4
# C2C4 connection

    ip link add name C2C4-C2 type veth peer name C2C4-C4
vppctl -s /run/vpp/cli-vppC2.sock create host-interface name C2C4-C2
vppctl -s /run/vpp/cli-vppC4.sock create host-interface name C2C4-C4

    vppctl -s /run/vpp/cli-vppC2.sock set int state host-C2C4-C2 up
vppctl -s /run/vpp/cli-vppC4.sock set int state host-C2C4-C4 up
vppctl -s /run/vpp/cli-vppC2.sock set int ip address host-C2C4-C2 fc00 :: 8: 0: 1/112 vppctl
-s /run/vpp/cli-vppC4.sock set int ip address host-C2C4-C4 fc00 :: 8: 0: 2/112

    vppctl -s /run/vpp/cli-vppC2.sock set interface mtu 1500 host-C2C4-C2
vppctl -s /run/vpp/cli-vppC4.sock set interface mtu 1500 host-C2C4-C4
# C4E2 connection
ip link add name C4E2-C4 type veth peer name C4E2-E2
vppctl -s /run/vpp/cli-vppC4.sock create host-interface name C4E2-C4
vppctl -s /run/vpp/cli-vppE2.sock create host-interface name C4E2-E2

    vppctl -s /run/vpp/cli-vppC4.sock set int state host-C4E2-C4 up
vppctl -s /run/vpp/cli-vppE2.sock set int state host-C4E2-E2 up
vppctl -s /run/vpp/cli-vppC4.sock set int ip address host-C4E2-C4 fc00 :: 4: 0: 1/112 vppctl
```

-s /run/vpp/cli-vppE2.sock set int ip address host-C4E2-E2 fc00 :: 4: 0: 2/112

vppctl -s /run/vpp/cli-vppC4.sock set interface mtu 1500 host-C4E2-C4
vppctl -s /run/vpp/cli-vppE2.sock set interface mtu 1500 host-C4E2-E2

Routing Table
Link E1C1
vppctl -s /run/vpp/cli-vppE1.sock ip route add fc00 :: 1: 0: 2/112 via fc00 :: 1: 0: 1
vppctl -s /run/vpp/cli-vppC1.sock ip route add fc00 :: 1: 0: 1/112 via fc00 :: 1: 0: 2
vppctl -s /run/vpp/cli-vppC1.sock ip route add fc00 :: 1: 2/112 via fc00 :: 1: 0: 1 ns1-C1
Link C1C2
vppctl -s /run/vpp/cli-vppC1.sock ip route add fc00 :: 2: 0: 2/112 via fc00 :: 2: 0: 1
vppctl -s /run/vpp/cli-vppC2.sock ip route add fc00 :: 2: 0: 1/112 via fc00 :: 2: 0: 2
Link C2E2
vppctl -s /run/vpp/cli-vppC2.sock ip route add fc00 :: 3: 0: 2/112 via fc00 :: 3: 0: 1
vppctl -s /run/vpp/cli-vppE2.sock ip route add fc00 :: 3: 0: 1/112 via fc00 :: 3: 0: 2
vppctl -s /run/vpp/cli-vppC2.sock ip route add fc00 :: 2: 2/112 via fc00 :: 3: 0: 2 ns2-C2
Link E1C3
vppctl -s /run/vpp/cli-vppE1.sock ip route add fc00 :: 6: 0: 2/112 via fc00 :: 6: 0: 1
vppctl -s /run/vpp/cli-vppC3.sock ip route add fc00 :: 6: 0: 1/112 via fc00 :: 6: 0: 2
Link C1C3
vppctl -s /run/vpp/cli-vppC1.sock ip route add fc00 :: 7: 0: 2/112 via fc00 :: 7: 0: 1
vppctl -s /run/vpp/cli-vppC3.sock ip route add fc00 :: 7: 0: 1/112 via fc00 :: 7: 0: 2
Link C3C4
vppctl -s /run/vpp/cli-vppC3.sock ip route add fc00 :: 5: 0: 1/112 via fc00 :: 5: 0: 2
vppctl -s /run/vpp/cli-vppC4.sock ip route add fc00 :: 5: 0: 2/112 via fc00 :: 5: 0: 1
Link C2C4
ping
ping

vppctl -s /run/vpp/cli-vppC2.sock ip route add fc00 :: 8: 0: 2/112 via fc00 :: 8: 0: 1
vppctl -s /run/vpp/cli-vppC4.sock ip route add fc00 :: 8: 0: 1/112 via fc00 :: 8: 0: 2
Link C4E2
vppctl -s /run/vpp/cli-vppC4.sock ip route add fc00 :: 4: 0: 2/112 via fc00 :: 4: 0: 1
vppctl -s /run/vpp/cli-vppE2.sock ip route add fc00 :: 4: 0: 1/112 via fc00 :: 4: 0: 2

A.5 File '0-2-creating-hosts.sh'

#! / usr / bin / sudo bash

```
    # This file is created to instantiate hosts and connect them to routers.

    # HOSTS
ip netns add ns1
ip link add pc1 type veth peer name vethns1 ip
link set vethns1 netns ns1
ip netns exec ns1 ip link set lo up ip
netns exec ns1 ip link set vethns1 up ip
link set pc1 up
ip netns exec ns1 ip route flush table main #delete the existing routing table ip netns
exec ns1 ip addr flush dev vethns1 #delete all ip addresses present ip netns exec ns1
ip addr add fc00 :: 1: 2/112 dev vethns1
ip netns exec ns1 ip route add default via fc00 :: 1: 1

    ip netns add ns2
ip link add pc2 type veth peer name vethns2 ip
link set vethns2 netns ns2
ip netns exec ns2 ip link set lo up ip
netns exec ns2 ip link set vethns2 up ip
link set pc2 up
ip netns exec ns2 ip route flush table main #delete the existing routing table ip netns
exec ns2 ip addr flush dev vethns2 #delete all ip addresses present ip netns exec ns2
ip addr add fc00 :: 2: 2/112 dev vethns2
ip netns exec ns2 ip route add default via fc00 :: 2: 1

    # LINK HOSTS TO VPP
sudo vppctl -s /run/vpp/cli-vppE1.sock create host-interface name pc1 sudo
vppctl -s /run/vpp/cli-vppE1.sock set interface mtu 1300 host-pc1 sudo vppctl
-s / run / vpp / cli-vppE1.sock set interface state host-pc1 up
sudo vppctl -s /run/vpp/cli-vppE1.sock set interface ip address host-pc1 fc00 :: 1: 1/112

    sudo vppctl -s /run/vpp/cli-vppE2.sock create host-interface name pc2 sudo
vppctl -s /run/vpp/cli-vppE2.sock set interface mtu 1300 host-pc2 sudo vppctl
-s / run / vpp / cli-vppE2.sock set interface state host-pc2 up
sudo vppctl -s /run/vpp/cli-vppE2.sock set interface ip address host-pc2 fc00 :: 2: 1/112

    # SEND IPv6 Neighbor Discovery Messages
ip netns exec ns1 ping -c 1 fc00 :: 1: 0: 2 ip
netns exec ns2 ping -c 1 fc00 :: 3: 0: 1
```

A.6 File '0-3-setting-SRv6.sh'

```
#! / usr / bin / sudo bash

    # This file is created to set SRv6

    # Setting BSID
# E1
sudo vppctl -s /run/vpp/cli-vppE1.sock create loopback interface instance 0
sudo vppctl -s /run/vpp/cli-vppE1.sock set interface state loop0 up
sudo vppctl -s /run/vpp/cli-vppE1.sock set interface ip address loop0 e1 :: / 128
# E2
sudo vppctl -s /run/vpp/cli-vppE2.sock create loopback interface instance 0
sudo vppctl -s /run/vpp/cli-vppE2.sock set interface state loop0 up
sudo vppctl -s /run/vpp/cli-vppE2.sock set interface ip address loop0 e2 :: / 128

    # BSID ROUTING TABLE
sudo vppctl -s /run/vpp/cli-vppE1.sock ip route add c1 :: / 128 via fc00 :: 1: 0: 2
sudo vppctl -s /run/vpp/cli-vppE1.sock ip route add c3: : / 128 via fc00 :: 6: 0: 2

    sudo vppctl -s /run/vpp/cli-vppC1.sock ip route add e1 :: 9/128 via fc00 :: 1: 0: 1
sudo vppctl -s /run/vpp/cli-vppC1.sock ip route add c3 :: / 128 via fc00 :: 7: 0: 2 sudo
vppctl -s /run/vpp/cli-vppC1.sock ip route add c2 :: / 128 via fc00 :: 2: 0: 2

    sudo vppctl -s /run/vpp/cli-vppC2.sock ip route add c1 :: / 128 via fc00 :: 2: 0: 1 sudo
vppctl -s /run/vpp/cli-vppC2.sock ip route add c4: : / 128 via fc00 :: 8: 0: 2 sudo
vppctl -s /run/vpp/cli-vppC2.sock ip route add e2 :: 9/128 via fc00 :: 3: 0: 2

    sudo vppctl -s /run/vpp/cli-vppC3.sock ip route add c1 :: / 128 via fc00 :: 7: 0: 1 sudo
vppctl -s /run/vpp/cli-vppC3.sock ip route add c4: : / 128 via fc00 :: 5: 0: 1 sudo
vppctl -s /run/vpp/cli-vppC3.sock ip route add e1 :: 9/128 via fc00 :: 6: 0: 1

    sudo vppctl -s /run/vpp/cli-vppC4.sock ip route add c3 :: / 128 via fc00 :: 5: 0: 2 sudo
vppctl -s /run/vpp/cli-vppC4.sock ip route add c2: : / 128 via fc00 :: 8: 0: 1 sudo
vppctl -s /run/vpp/cli-vppC4.sock ip route add e2 :: 9/128 via fc00 :: 4: 0: 2

    sudo vppctl -s /run/vpp/cli-vppE2.sock ip route add c2 :: / 128 via fc00 :: 3: 0: 1
sudo vppctl -s /run/vpp/cli-vppE2.sock ip route add c4: : / 128 via fc00 :: 4: 0: 1

    # configuring srv6
sudo vppctl -s /run/vpp/cli-vppE1.sock set sr encaps source addr e1 ::
sudo vppctl -s /run/vpp/cli-vppE1.sock sr policy add bsid a1 :: next c1 :: next c2 :: next e2 :: 9
encap
```

```
sudo vppctl -s /run/vpp/cli-vppE1.sock sr policy add bsid a3 :: next c3 :: next c4 :: next e2 :: 9
encap
sudo vppctl -s /run/vpp/cli-vppE1.sock sr steer l3 fc00 :: 2: 2/112 via bsid a3 ::
sudo vppctl -s /run/vpp/cli-vppE1.sock sr localsid address e1 :: 9 behavior end.dx6 host-pc1 fc00
::1: 2

    sudo vppctl -s /run/vpp/cli-vppE2.sock set sr encaps source addr e2 ::
sudo vppctl -s /run/vpp/cli-vppE2.sock sr policy add bsid a1 :: next c2 :: next c1 :: next e1 :: 9
encap

    sudo vppctl -s /run/vpp/cli-vppE2.sock sr policy add bsid a3 :: next c4 :: next c3 :: next e1 ::
9 encap

    sudo vppctl -s /run/vpp/cli-vppE2.sock sr steer l3 fc00 :: 1: 2/112 via bsid a3 ::
sudo vppctl -s /run/vpp/cli-vppE2.sock sr localsid address e2 :: 9 behavior end.dx6 host-pc2 fc00
::2: 2
# pause 'E2: SRV6 configured. Press enter. '

    sudo vppctl -s /run/vpp/cli-vppC1.sock sr localsid address c1 :: behavior end
sudo vppctl -s /run/vpp/cli-vppC2.sock sr localsid address c2 :: behavior end
sudo vppctl -s / run /vpp/cli-vppC3.sock sr localsid address c3 :: behavior end
sudo vppctl -s /run/vpp/cli-vppC4.sock sr localsid address c4 :: behavior end

    python3 moving-node-graph.py

    # All TCP traffic goes through policy1 (and if active, also through policy 2). The rest for the 3
sudo vppctl -s /run/vpp/cli-vppE1.sock classify table mask l3 ip6 src dst proto
# proto = next header, 58 = ICMPv6, 6 = TCP
sudo vppctl -s /run/vpp/cli-vppE1.sock classify session acl-hit-next 1 table-index 0 match l3 ip6 src
fc00 :: 1: 2 dst fc00 :: 2: 2 proto 6 action set-sr- policy-index 0
sudo vppctl -s /run/vpp/cli-vppE1.sock set interface input acl intfc host-pc1 ip6-table 0

    sudo vppctl -s /run/vpp/cli-vppE2.sock classify table mask l3 ip6 src dst proto
sudo vppctl -s /run/vpp/cli-vppE2.sock classify session acl-hit-next 1 table-index 0 match l3 ip6 src
fc00 :: 2: 2 dst fc00 :: 1: 2 proto 6 action set-sr- policy-index 0
sudo vppctl -s /run/vpp/cli-vppE2.sock set interface input acl intfc host-pc2 ip6-table 0

    # The next two lines are due to the fact that the linux kernel has problems in TCP-SRv6
in fully emulated environments (checksum).
ip netns exec ns1 ethtool -K vethns1 tx off ip
netns exec ns2 ethtool -K vethns2 tx off
```

A.7 File 'tcp.sh'

```
#! / usr / bin / sudo bash
gnome-terminal - bash -c "ip netns exec ns2 iperf3 -s -V -p 80; exec bash"
sleep 2
gnome-terminal - bash -c "ip netns exec ns1 iperf3 -c fc00 :: 2: 2 -V -p 80 -l 1460; exec bash"
```

A.8 File 'tcp.sh'

```
#! / usr / bin / sudo bash
# ping of 1176 bytes
sudo ip netns exec ns1 ping -c 5 -M do -s 1176 fc00 :: 2: 2
```

A.9 File 'enable-flow.sh'

```
#! / usr / bin / sudo bash
echo 'Start TCP communication: start Server on Host B and client on Host A' gnometerminal
- bash -c "ip netns exec ns2 iperf3 -s -V -p 80; exec sudo bash" # add to string;
exec sudo bash to keep the shell open after ctrl + c aborting the process

    sleep 2
gnome-terminal –workingdirectory = '/ home / giulio / Desktop / BlackHolesFinal3 /
3-TCP-ICMP' - bash -c "sh. /tcp-client.sh; exec sudo bash"

    echo 'Start ping'
gnome-terminal –workingdirectory = '/ home / giulio / Desktop / BlackHolesFinal3 /
3-TCP-ICMP' - bash -c "sh. /2-1-ping.sh; exec sudo bash"
```

A.10 File 'tcp-client.sh'

```
#! / usr / bin / sudo bash

    while true
do
ip netns exec ns1 iperf3 -6 -c fc00 :: 2: 2 -V -p 80 -k 1 -F '/ home / giulio / Desktop /
BlackHolesFinal3 / big-file' # send big-file once (-k 1) ( the mtu regulates it himself (perhaps following a mtu discovery))
exec sudo bash #to remove if you want it to loop while true
# sleep 1.5
Done
```

A.11 File '1-1-enable-rec-policy.sh'

```
#! / usr / bin / sudo bash

    # This file is created to set SRv6 policy
sudo vppctl -s /run/vpp/cli-vppC1.sock ip route of c2 :: / 128 via fc00 :: 2: 0: 2
sudo vppctl -s /run/vpp/cli-vppC1.sock sr policy add bsid a2 :: next c3 :: next c4 :: 1 encap
sudo vppctl -s /run/vpp/cli-vppC1.sock sr steer l3 c2 :: / 128 via bsid a2 ::

    sudo vppctl -s /run/vpp/cli-vppC4.sock ip route add c3 :: 1/128 via fc00 :: 5: 0: 2
sudo vppctl -s /run/vpp/cli-vppC4.sock sr localsid address c4 :: 1 behavior end.dx6 host-C2C4-C4
fc00 :: 8: 0: 1
sudo vppctl -s /run/vpp/cli-vppC4.sock ip route of fc00 :: 8: 0: 1/112 via fc00 :: 8: 0: 2

    sudo vppctl -s /run/vpp/cli-vppC2.sock ip route of c1 :: / 128 via fc00 :: 2: 0: 1
sudo vppctl -s /run/vpp/cli-vppC2.sock sr policy add bsid a2 :: next c4 :: next c3 :: 1 encap sudo
vppctl -s /run/vpp/cli-vppC2.sock sr steer l3 c1: : / 128 via bsid a2 ::

    sudo vppctl -s /run/vpp/cli-vppC3.sock ip route add c4 :: 1/128 via fc00 :: 5: 0: 1
sudo vppctl -s /run/vpp/cli-vppC3.sock sr localsid address c3 :: 1 behavior end.dx6 host-C1C3-C3
fc00 :: 7: 0: 1
sudo vppctl -s /run/vpp/cli-vppC3.sock ip route of fc00 :: 7: 0: 1/112 via fc00 :: 7: 0: 2
```

A.12 File '1-2-disable-rec-policy.sh'

```
#! / usr / bin / sudo bash

    # This file is created to set SRv6 policy

    sudo vppctl -s /run/vpp/cli-vppC1.sock ip route add c2 :: / 128 via fc00 :: 2: 0: 2
sudo vppctl -s /run/vpp/cli-vppC1.sock sr policy del bsid a2 ::
sudo vppctl -s /run/vpp/cli-vppC1.sock sr steer del l3 c2 :: / 128 via bsid a2 ::

    sudo vppctl -s /run/vpp/cli-vppC4.sock ip route del c3 :: 1/128 via fc00 :: 5: 0: 2
sudo vppctl -s /run/vpp/cli-vppC4.sock sr localsid del address c4 :: 1 behavior end.dx6 host-C2C4-
C4
fc00 :: 8: 0: 1
sudo vppctl -s /run/vpp/cli-vppC4.sock ip route add fc00 :: 8: 0: 1/112 via fc00 :: 8: 0: 2

    sudo vppctl -s /run/vpp/cli-vppC2.sock ip route add c1 :: / 128 via fc00 :: 2: 0: 1
sudo vppctl -s /run/vpp/cli-vppC2.sock sr policy del bsid a2 ::
sudo vppctl -s /run/vpp/cli-vppC2.sock sr steer del l3 c1 :: / 128 via bsid a2 ::
```

sudo vppctl -s /run/vpp/cli-vppC3.sock ip route of c4 :: 1/128 via fc00 :: 5: 0: 1
sudo vppctl -s /run/vpp/cli-vppC3.sock sr localsid del address c3 :: 1 behavior end.dx6 host-C1C3-C3 fc00 :: 7: 0: 1
sudo vppctl -s /run/vpp/cli-vppC3.sock ip route add fc00 :: 7: 0: 1/112 via fc00 :: 7: 0: 2

A.13 File '6-MTU-Path-Discovery.sh'

#! / usr / bin / sudo bash

ip netns exec ns1 traceroute -6 -T –mtu fc00 :: 2: 2 # to verify the happened mtu discovery path: ip
netns exec ns1 ip route get fc00 :: 2: 2
sleep 3
echo "The MTU path discovery (TCP) result is:" ip
netns exec ns1 ip route get fc00 :: 2: 2

A.14 File '9-exit-kill-VPP.sh'

#! / usr / bin / sudo bash

sudo pkill vpp
sudo ip netns of ns1 sudo
ip netns of ns2 sudo ip
link of E1C1-C1 sudo ip
link of C1C2-C2 sudo ip
link of C2E2-E2 sudo ip
link of E1C3-C3 sudo ip
link of C1C3-C3 sudo ip
link of C3C4-C4 sudo ip
link of C2C2

A.15 Python API installation

The following commands allow the installation of the Python API.

apt-get install vpp-api-python python3-vpp-api vpp-dbg vpp-dev curl
https://bootstrap.pypa.io/get-pip.py –output get-pip.py python getpip.
py
pip install vpp-papi

pip install cffi

A.16 Script 'moving-node-graph.py'

The following script allows adding the 'ip6-inacl' node to the processing graph of VPP- This node allows for traffic classification based on type, source address, destination address, and more.

#! / usr / bin / env python

from –future– import print-function

import os
import fnmatch

from vpp-papi import VPP

VPP-JSON-DIR = '/ usr / share / vpp / api / core /'
API-FILE-SUFFIX = '* .api.json'

def load-json-api-files (json-dir = VPP-JSON-DIR, suffix = API-FILE-SUFFIX):
jsonfiles = []
for root, dirnames, filenames in os.walk (json-dir):
for filename in fnmatch.filter (filenames, suffix):
jsonfiles.append (os.path.join (json-dir, filename))

if not jsonfiles:
print ('Error: no json api files found')
exit (-1)

return jsonfiles

def connect-vpp (jsonfiles, VPP-ID, prefix):
vpp = VPP (jsonfiles)
r = vpp.connect (VPP-ID, chroot-prefix = prefix)
print ("VPP api opened with code:% s"% r) return
vpp

# VppE1
vpp1 = connect-vpp (load-json-api-files (), 'vppE1', 'vppE1') vpp1.api.add-node-next
(node-name = 'ip6-inacl', next-name = 'sr-pl-rewrite-encaps')
# indicates that the next node in the graph of vpp and sr-pl-rewrite-encaps if that
just visited and ip6-inacl

```
vpp1.disconnect ()

    # VppE2
vpp2 = connect-vpp (load-json-api-files (), 'vppE2', 'vppE2') vpp2.api.add-node-next
(node-name = 'ip6-inacl', next-name = 'sr-pl-rewrite-encaps')
# indicates that the next node in the graph of vpp and sr-pl-rewrite-encaps if that
just visited and ip6-inacl
vpp2.disconnect ()
```

Appendix B

Appendix

B.1 Main

```
clear all
close all
load ('test3.mat')
statsSR = zeros(1, 4);
k = 1;
margine = 0;
scaling-factor = 0.6;
congestion-amplification = 3*10^3;
perc-flusso-perso = 1;
DP = 3;
    cammini, linkRete, flows, PSID, R, YbExp, G, BLUE, GREEN, B, ~, ~, SL, NH, NN

precalcola-dati(A, delay, scaling-factor * bandwidth, TM, congestion-amplification);
failures = zeros(length(linkRete), 2);
links-BH = zeros(length(linkRete));
flussi-BH = zeros(length(flows), length(linkRete));
flussi-BH1 = zeros(length(flows), length(linkRete));
for f = 1:length(flows)
for l = 1:length(linkRete)
if R(l, f) > 0
if linkRete(l, 2)  = flows(f, 2)
failures(k, :) = [l f];
    links-BH(:, k), flussi-BH(:, k), flussi-BH1(:, k)

start-failure-detection(failures(k, :), A, linkRete, flows, PSID, R, YbExp, G, BLUE, GREEN, B,
SL, cammini, NH, margine, congestion-amplification, perc-flusso-perso, DP, 0, NN);
k = k + 1
end
end
end
```

```
end
disegnaFigure
```

B.2 Pre-calculates data

```
function [cammini, linkRete, flows, PSID, R, YbExp, G, BLUE, GREEN, B, GREEN-eq, BLUE-eq,
SL, NH,NN] = precalcola-dati(A, delay, bandwidth, TM, congestion-amplification)

    split-color = 1;ECMP = 0;
num-color = length(split-color);
N = length(A);
L = sum(sum(A));

    linkRete = zeros(L, 5);
l = 1;

    for i = 1:N
for j = 1:N
if A(i, j) == 1
linkRete(l, :) = [l i j delay(i, j) bandwidth(i, j)];
l = l + 1;
end
end
end

    A(A == 0) = inf;
K = length(find(TM > 0)) * num-color;
flows = zeros(K, 5);
k = 1;

    for c = 1:num-color
for i = 1:N
for e = 1:N
if TM(i, e)  = 0
intensity = split-color(c) * TM(i, e);
flows(k, :) = [k i e c intensity];
k = k + 1;
end
end
end
end
```

```
    K = length(flows);

    [G, cammini, NH] = calcolaIGPpaths(L, N, A, linkRete, ECMP);

    R = zeros(L, K);

    [SL1, R] = calcolaSL-color1(N, R, G, linkRete, flows);
SL = SL1;

    [B, YbExp, PSID, GREEN, BLUE, GREEN-eq, BLUE-eq] = aggiornaPSID(SL, G, flows, N,
linkRete);

    NN = CalcolaMargine(1,1,congestion-amplification);
    return
```

B.3 Calculate IGP path

```
function [IGP-paths, cammini, NH] = calcolaIGPpaths(L, N, A, linkRete, ECMP)

    IGP-paths = zeros(L, N * (N - 1));
cammini = cell(N);
k = 1;

    if ECMP == 0
NH = zeros(N);
for s = 1:N
     , paths
= graphshortestpath(sparse(A), s);
for d = 1:N
if s  = d
NH(s, d) = pathsd(2);
end
end
end
end

    for s = 1:N
for d = 1:N
if s  = d
if ECMP == 1
shortestPaths = kShortestPath(A, s, d);
camminis, d = shortestPaths;
```

```
IGP-paths(:, k) = costruisciSupporto(shortestPaths, linkRete, L);
else
path = costruisciPathDaMatriceNH(NH, s, d, N);
camminis, d = path;
IGP-paths(:, k) = costruisciSupporto(camminis, d, linkRete, L);
end
k = k + 1;
end
end
end
```

```
    return
```

B.4 K shortestPath

```
function [shortestPaths, totalCosts] = kShortestPath(netCostMatrix, source, destination)

    k-paths = inf;

    k=1;
    path cost
= dijkstra(netCostMatrix, source, destination);

    path-number = 1;
Ppath-number,1 = path; Ppath-number,2 = cost;
current-P = path-number;

    size-X=1;
Xsize-X = path-number; path; cost;

    S(path-number) = path(1);
shortestPathsk = path;
totalCosts(k) = cost;

    if length(path) == 2
return
end

    while (k < k-paths && size-X  = 0 )
for i=1:length(X)
if Xi1 == current-P
size-X = size-X - 1;
```

```
X(i) = [];
break;
end
end

    P- = Pcurrent-P,1;
w = S(current-P);

    for i = 1: length(P-)
if w == P-(i)
w-index-in-path = i;
end
end

    for index-dev-vertex= w-index-in-path: length(P-) - 1
temp-netCostMatrix = netCostMatrix;
for i = 1: index-dev-vertex-1
v = P-(i);
temp-netCostMatrix(v,:)=inf;
temp-netCostMatrix(:,v)=inf;
end

    SP-sameSubPath=[];
index =1;
SP-sameSubPathindex=P-;
for i = 1: length(shortestPaths)
if length(shortestPathsi) >= index-dev-vertex
if P-(1:index-dev-vertex) == shortestPathsi(1:index-dev-vertex)
index = index+1;
SP-sameSubPathindex=shortestPathsi;
end
end
end

    v- = P-(index-dev-vertex);
for j = 1: length(SP-sameSubPath)
next = SP-sameSubPathj(index-dev-vertex+1);
temp-netCostMatrix(v-,next)=inf;
end

    sub-P = P-(1:index-dev-vertex);
cost-sub-P=0;
for i = 1: length(sub-P)-1
```

```
cost-sub-P = cost-sub-P + netCostMatrix(sub-P(i),sub-P(i+1));
end

    [dev-p c] = dijkstra(temp-netCostMatrix, P-(index-dev-vertex), destination);
if  isempty(dev-p)
path-number = path-number + 1;
Ppath-number,1 = [sub-P(1:end-1) dev-p];
Ppath-number,2 = cost-sub-P + c;
S(path-number) = P-(index-dev-vertex);
size-X = size-X + 1;
Xsize-X = path-number; Ppath-number,1 ;Ppath-number,2 ;
end
end

    if size-X > 0
shortestXCost= X13;
shortestX= X11;
for i = 2 : size-X
if Xi3 < shortestXCost
shortestX= Xi1;
shortestXCost= Xi3;
end
end
current-P = shortestX;

    k = k+1;
shortestPathsk = Pcurrent-P,1;
totalCosts(k) = Pcurrent-P,2;
end

    if totalCosts(k - 1) < totalCosts(k)
shortestPaths = shortestPaths(1:k - 1);
totalCosts = totalCosts(1:k - 1);
return
end
end

    return
```

B.5 Dijkstra

```
function [shortestPath, totalCost] = dijkstra(netCostMatrix, s, d)
==========================================================================
shortestPath: the list of nodes in the shortestPath from source to destination;
totalCost: the total cost of the shortestPath;
farthestNode: the farthest node to reach for each node after performing the routing;
n: the number of nodes in the network;
s: source node index;
d: destination node index;
==========================================================================
Code by:
++by Xiaodong Wang
++23 Jul 2004 (Updated 29 Jul 2004)
++http://www.mathworks.com/matlabcentral/fileexchange/5550-dijkstra-shortest-path-routing
Modifications (simplifications) by Meral Shirazipour 9 Dec 2009
==========================================================================
n = size(netCostMatrix,1);
for i = 1:n
initialize the farthest node to be itself;
farthestPrevHop(i) = i; farthestNextHop(i) = i;
end

    visited(1:n) = false;

    distance(1:n) = inf; parent(1:n) = 0;

    distance(s) = 0;
for i = 1:(n-1),
temp = [];
for h = 1:n,
if  visited(h) temp=[temp distance(h)];
else
temp=[temp inf];
end
end;
    t, u
= min(temp);
visited(u) = true;
for v = 1:n,
if ( ( netCostMatrix(u, v) + distance(u)) < distance(v) )
distance(v) = distance(u) + netCostMatrix(u, v); parent(v) = u; end;
end;
end;
```

```
    shortestPath = [];
if parent(d)  = 0 t = d;
shortestPath = [d];
while t  = s
p = parent(t);
shortestPath = [p shortestPath];

    if netCostMatrix(t, farthestPrevHop(t)) < netCostMatrix(t, p)
farthestPrevHop(t) = p;
end;
if netCostMatrix(p, farthestNextHop(p)) < netCostMatrix(p, t)
farthestNextHop(p) = t;
end;

    t = p;
end;
end;

    totalCost = distance(d);
```

B.6 Build support

```
function supporto = costruisciSupporto(shortestPaths, linkRete, L)

    supporto = zeros (L, 1);
P = length(shortestPaths);

    for p = 1:P
path = shortestPathsp;
for l = 2:length(path)
link = linkRete(:, 2) == path(l - 1) & linkRete(:, 3) == path(l);
supporto(link) = supporto(link) + (1 / P);
end
end

    return
```

B.7 Build path from matrix

```
function path = costruisciPathDaMatriceNH(NH, s, d, N)

    path = zeros(1, N);
path(1) = s;
nh = NH(s, d);
path(2) = nh;

    k = 3;

    while nh  = d
nh = NH(path(k - 1), d);
path(k) = nh;
k = k + 1;
end

    path = path(1:k - 1);

    return
```

B.8 Calculate SL color

```
function [SL, R] = calcolaSL-color1(N, R, IGP-paths, linkRete, flows)

    SL = cell(N);

    for s = 1:N
for d = 1:N
flusso = flows(:, 2) == s & flows(:, 3) == d & flows(:, 4) == 1;
if isempty(find(flusso == 1, 1))  = 1
SLs, d = [s d];
f = (s - 1) * (N - 1) + d - (d > s);
R(:, flusso) = IGP-paths(:, f);
end
end
end

    return
```

B.9 Update PSID

```
function [B, Yb, PSID, GREEN, BLUE, GREEN-eq, BLUE-eq] = aggiornaPSID(SL, G, flows, N,
linkRete)

    PSID = zeros(N * N, 3);
Yb = zeros(length(PSID), 1);
GREEN = zeros(N, N);
BLUE = zeros(N, N);
B = zeros(length(PSID), size(flows, 1));
GREEN-eq = zeros(length(PSID), size(flows, 1));
BLUE-eq = zeros(length(PSID), size(flows, 1));
p = 1;

    for i = 1:N
for a = 1:N
PSID(p, :) = [p i a];
p = p + 1;
end
end

    for f = 1:size(flows, 1)
sl = SLflows(f, 4)flows(f, 2), flows(f, 3);
contatore = PSID(:, 2) == flows(f, 2) & PSID(:, 3) == flows(f, 3);
B(contatore, f) = 1; Yb(contatore) = Yb(contatore) + flows(f, 5);
GREEN(flows(f, 2), flows(f, 3)) = 1;
GREEN-eq(contatore, f) = 1;
for a = 2:length(sl)
contatore = PSID(:, 2) == sl(a - 1) & PSID(:, 3) == sl(a);
if a < length(sl)
BLUE(sl(a - 1), sl(a)) = 1; BLUE-eq(contatore, f) = 1;
end
fid = (sl(a -1) - 1) * (N - 1) + sl(a) - (sl(a) > sl(a -1));
tmp = [linkRete(G(:, fid) > 0, :) G(G(:, fid) > 0, fid)];
for l = 1:size(tmp, 1)
psid = PSID(:, 2) == tmp(l, 3) & PSID(:, 3) == sl(a);
B(psid, f) = B(psid, f) + tmp(l, 6);
Yb(psid) = Yb(psid) + flows(f, 5) * tmp(l, 6);
end
end
end

    Yb = B * flows(:, 5);
```

return

B.10 Calculate Margin

function [NN] = CalcolaMargine(utilizzazione, CapacitaLink, congestion-amplification); L=10000;
P=[10,5];
utilizzazione= rand(1,L);
linkCap=[10$\hat{5}$, 10$\hat{6}$, 6*10$\hat{6}$,10$\hat{7}$, 10$\hat{8}$ 10$\hat{9}$];
indexCapacitaLink= randi(length(linkCap),L,1);
CapacitaLink=linkCap(1,indexCapacitaLink);

for cont = 1:length(utilizzazione);
t(cont)= probperdita(utilizzazione(cont),congestion-amplification);
t(cont)= round(t(cont)*CapacitaLink(cont)*utilizzazione(cont)); end
x=[utilizzazione;CapacitaLink];

net = feedforwardnet(P);
net = train (net,x,t);
NN = net;

B.11 Probability of loss

function [Qx] = probperdita(utilizazione,congestion-amplification) x = [0 0.6 0.85 0.95 1];
y= [0 0.2 0.45 0.7 1];
xx = [0:0.001:1];
yy = interp1(x,y,xx);

tmp==abs(utilizazione*ones(size(xx)) - xx);
index=find(tmp==min(tmp),1);
Qx= congestion-amplification * yy(index);

B.12 Draw figures

link-precision = zeros(1, length(failures));
link-recall = zeros(1, length(failures));

for f = 1:length(failures)
TP = links-BH(failures(f, 1), f);
FP = sum(links-BH(:, f)) - TP;

```
FN = 1 - TP;
link-precision(f) = 100 * TP / (TP + FP);
link-recall(f) = 100 * TP / (TP + FN);
end

    flow-precision = zeros(1, length(failures));
flow-precision1 = zeros(1, length(failures));
flow-recall = zeros(1, length(failures));

    for f = 1:length(failures)
TP = flussi-BH(failures(f, 2), f);
TP1 = flussi-BH1(failures(f, 2), f);
FP = sum(flussi-BH(:, f)) - TP;
FN = 1 - TP;
FP1 = sum(flussi-BH1(:, f)) - TP1;
flow-recall(f) = 100 * TP / (TP + FN);
if flow-recall(f)  = 0
flow-precision(f) = 100 * TP / (TP + FP);
flow-precision1(f) = 100 * TP1 / (TP1 + FP1);
end
end

    figure(1)
axis 'square'
grid on
box on
hold on
plot(sort(flow-precision), '-', 'LineWidth', 2)
plot(sort(flow-precision1), 'r-', 'LineWidth', 2)
plot(sort(flow-recall), 'k-', 'LineWidth', 2)
legend('precision 1','precision 2','recall')
xlabel('test ID')
ylabel('[
    tmp = sum(flussi-BH);
qualiLinks = zeros(length(linkRete), 1);
qualiLinks(failures(:, 1)) = 1;
qualiLinks = find(qualiLinks == 1);

    figura = zeros(length(qualiLinks), 2);

    for l = 1:length(qualiLinks)
figura(l, 2) = max(tmp(failures(:, 1) == qualiLinks(l)));
figura(l, 1) = min(tmp(failures(:, 1) == qualiLinks(l)));
```

```
end

    y = [0.5 * (figura(:, 1) + figura(:, 2)) abs(figura(:, 1) - figura(:, 2))];
y = sortrows(y, 1);

    y1 = [0.5 * (figura(:, 1) + figura(:, 2)) figura(:, 1) figura(:, 2)];
y1 = sortrows(y1, 1);

    figure(2)
axis 'square'
grid on
box on
hold on
plot(tmp, (100 / length(failures)) * tmp1, '-', 'LineWidth', 2)
ylabel('xlabel('
    figure(3)
axis 'square'
grid on
box on
hold on
plot(y1(:, 2), '-', 'LineWidth', 2)
plot(y1(:, 3), 'r–', 'LineWidth', 2)
ylabel('# of flows')
xlabel('link ID')
legend('min','max')

    tmp = sort((100 / length(flows)) * sum(flussi-BH));
tmp1 = zeros(length(tmp), 1);

    for i = 1:length(tmp)
tmp1(i) = sum(tmp <= tmp(i));
end
```