Performance Evaluation and Traffic Engineering Analysis of SRv6 Networks in Containerized Environments

Angelos Pyrpyris

University of the Peloponnese

School of Economics and Technology

Department of Informatics and Telecommunications

2022202402018

Supervisor: Prof. Kostas D. Peppas

May, 2026

Statement of Originality

The work presented in this thesis/dissertation is entirely from the studies of the individual student, except where otherwise stated. Where derivations are presented and the origin of the work is either wholly or in part from other sources, then full reference is given to the original author. This work has not been presented previously for any degree, nor is it at present under consideration by any other degree awarding body.

Note on the use of generative AI: I acknowledge that Google's Gemini was used at certain points in this thesis, especially during the final stages of the writing process, for the purposes of text enhancement and suggesting edits. The author reviewed, edited, and approved all content before submission.

Signed: Angelos Pyrpyris

Acknowledgements

First of all, I would like to express my gratitude to my supervisor, Prof. Kostas D. Peppas, for the excellent opportunity to pursue my studies in the master's program, enabling me to expand my knowledge in computer science. I would also like to thank him for his trust and the academic freedom he granted me throughout the course of this thesis.

I am also deeply grateful to my family and friends for their support and encouragement throughout my studies and beyond.

Abstract

IPv6 Segment Routing (SRv6) is a next-generation implementation of the source routing paradigm that combines Segment Routing with IPv6 using flexible IPv6 extension headers. Its benefits include enhanced traffic engineering, scalability, and network programmability, which means a more flexible and simplified approach compared to traditional MPLS-based architectures, positioning SRv6 at the forefront of network transport.

This master's thesis provides a practical evaluation of SRv6 in containerized environments and contributes a reproducible platform for hands-on experimentation. After minor but necessary modifications, an 8-node SRv6 L3VPN topology was successfully implemented using the open-source tools Containerlab and FRRouting (FRR). Afterward, basic configuration of VRFs and BGP with SRv6 followed, along with preliminary results of key performance metrics such as latency (RTT), resource consumption (CPU/memory), and forwarding behavior under low-load conditions.

By focusing on the deployment and evaluation of SRv6 within such environments and despite several limitations, this work confirms the feasibility of SRv6 for educational and experimental purposes and highlights the value of open-source tooling in bridging the gap between theoretical concepts and practical networking research.

Keywords: SRv6, Segment Routing, Containerlab, FRRouting, L3VPN, traffic engineering, containerized networks, performance evaluation

List of Tables

List of Figures

Table of Contents

Chapter 1

Introduction

1.1 Context and Motivation

As networking technologies continue to evolve, modern networks are becoming increasingly complex and demanding. This creates a strong need for innovative approaches to effectively manage their performance and scale. Recently, architectures that are more programmable, flexible, and efficient have also attracted the interest of both academia and industry.

SRv6 is a revolutionary IP bearer protocol that reduces the complexity of the traditional overlay protocols like MPLS, and it serves as the foundation for building next-generation IP networks[18]. SRv6 combines the advantages of the source routing mechanism and extends the principles of Segment Routing (SR) with the simplicity and extensibility that IPv6 offers[20, 19]. To this end, SRv6 provides advanced capabilities such as network programmability, traffic engineering (TE), and service function chaining (SFC).

Moreover, traditional networking environments defined by hardware-centric architectures that rely on dedicated routers and middleboxes have given way to more modern software-defined infrastructures called SDN (Software-Defined Networking) and NFV (Network Functions Virtualization)[23]. At the same time, cloud-native architectures, enabled by containerization and virtualization technologies such as Docker and Kubernetes, demand high levels of scalability, automation, and operational flexibility—characteristics that SRv6 is well-positioned to support.

This thesis seeks to examine the theoretical foundations of SRv6 and engage in

practical experiments, rather than to introduce technical innovations. Moreover, open-source tools that have seen increasing adoption, such as the combination of Containerlab and FRRouting (FRR)[51, 21]—a container-based network emulation platform and a routing software suite, respectively—enable the simulation of complex network topologies[49] even on modest hardware. This so-called democratization of network experimentation for emerging technologies results in a broader understanding and access to advanced networking research.

1.2 Research Objectives and Questions

The primary objective of this study is to conduct a hands-on evaluation of SRv6's behavior in containerized environments. More specifically, to achieve its goals—which include practical implementation and baseline performance metrics—the foundation of this thesis, rather than focusing on the latest developments, relies on existing open-source resources and tools utilized in a controlled, reproducible lab setting[49, 51, 21].

To accomplish these objectives, the majority of this work is based on a cloned GitHub repository (srv6-labs), which, after minor compatibility modifications and with the assistance of Containerlab and FRR, is prepared to be deployed as an SRv6-enabled topology to measure key aspects of network performance.

Accordingly, this research seeks to answer the following questions:

- What are the practical steps required to successfully implement SRv6 in a containerized environment using open-source tools like the aforementioned FRR and Containerlab, and what practical challenges may arise during setup?
- Can we effectively examine simple performance characteristics of SRv6, such as latency, resource usage, and forwarding ability, in these types of emulated networks?
- To what extent does SRv6 support advanced features (in containerized setups), like higher-level optimization mechanisms, i.e., traffic engineering, given the limitations of current open-source implementations?

- What conclusions may be derived concerning the feasibility of SRv6 for lightweight experimental deployments, and how do these influence future advancements?

The objectives outlined above reflect a grounded approach, focusing on gaining experience and educational value. They prioritize understanding SRv6 in accessible environments rather than addressing identified gaps or proposing novel contributions.

1.3 Scope and Delimitations

The scope of this thesis is an experimental case study of SRv6 in a small-scale, containerized network emulation. In particular, an 8-node topology simulating an L3VPN service over SRv6 serves as the backbone on which fundamental metrics, including latency, packet loss, resource consumption, and basic forwarding validation, are examined.

A standard workstation running Ubuntu in a VirtualBox VM was used to conduct the evaluation. The FRR version 8.4 that was used provides partial SRv6 support, that is, basic encapsulation and BGP integration, but limited locator and VRF binding capabilities.

Several key limitations are outlined below:

- Reliance on an existing open-source repository[49] that necessitated minor adaptations (e.g., YAML syntax fixes and kernel tweaks) instead of a custom from-scratch design.
- Incomplete measurements were imposed by tool limitations resulting from unsuccessful iperf3[29] throughput tests and limited packet capture visibility, which led to a focus on low-load scenarios without advanced traffic engineering (TE) policies.
- The use of an older FRR version[21]—approximately two major releases be-

hind—resulted in the lack of support for features such as uSID compression and hybrid SRv6/MPLS deployments.

- Restricted available resources and time constraints, preventing comparisons with proprietary tools (e.g., Cisco XRd) or large-scale clusters (e.g., Kubernetes-integrated setups).

These limitations preserve the work's viability for a master's-level project while also leaving open avenues for future research directions.

1.4 Thesis Structure

The remainder of this master's thesis is organized as follows:

- Chapter 2 reviews the theoretical background of the research area. It presents all the networking concepts relevant to the subsequent chapters. Towards the end of the chapter, the most significant related work is discussed.
- Chapter 3 provides an overview of the execution environment, the reasoning behind the selected tools, and the experimental workflow carried out throughout the thesis. It further describes the performance metrics and measurement methods used, as well as the required modifications and the lab deployment procedure.
- Chapter 4 starts with a detailed description of the SRv6 lab topology, which served as the basis for the experiments and measurements. It then presents the necessary implementation and integration techniques before proceeding further.
- Chapter 5 presents the evaluation results and discusses the key findings, explaining what the results mean and why they appear as they do, while taking into account the limitations encountered.
- Chapter 6 concludes the thesis by summarizing its purpose and objectives, as well as outlining potential future research directions that could build upon and enhance the findings of this work.

This structure provides a logical progression from theoretical background to practical implementation and evaluation, ending with final commentary and insights.

Chapter 2

Background

2.1 Routing and Forwarding Paradigms

2.1.1 IP Routing Protocols

Traditional IP networks implement hop-by-hop, shortest-path, destination-based forwarding mechanisms. Each network node comprises a collection of routes, which in turn map an IP prefix (i.e., a contiguous block of IP addresses) to a subsequent hop. A next hop refers to an additional network node that is directly connected to the current node, either virtually or physically. Each route maps an IP prefix to a next-hop neighbor, with an associated metric (e.g., topological distance) for shortest-path selection. This is in most cases a function of the topological distance from the present node to the node containing the destination prefix. In practical applications, route metrics are employed to accelerate shortest-path forwarding. Upon receiving a packet for forwarding, a router extracts the destination IP address from the packet and conducts a query in its routing table. This query seeks to determine a route whose destination prefix includes the packet's destination IP address. In the event that multiple routes correspond to the destination, the most specific route is chosen, namely the route with the extended prefix. This procedure is referred to as the longest initial match. If multiple routes with identical prefix lengths match the destination, the route with the lowest cost is chosen. The packet is then forwarded to the next hop specified by the route. If multiple routes with identical costs lead to the same destination, routers typically employ Equal-Cost Multi-Path (ECMP). The concept of ECMP involves applying a hashing function to specific packet metadata

and selecting a route based on the outcome of this hash. The result is that packets corresponding to ECMP routes will be distributed evenly across the available next hops.

To perform their packet forwarding "responsibilities", routers must acquire the routes necessary to access each node inside a specified network. This distribution of reachability information can be accomplished through many methods. In the early stages of the Internet, the number of nodes was sufficiently small to enable the configuration of static routes. Operators were required to manually adjust their routers to accommodate the addition of a new node or prefix. These network management techniques are inadequate for the scale of contemporary networks. Routing protocols were created to autonomously disseminate routing information among the routers inside a network. Distance-vector routing methods, such as RIP[25], disseminate distance vectors to other nodes within the network. These vectors encompass information akin to the routing tables previously described. The distinction is that vectors consist only the paths of minimal expense. Conversely, link-state protocols like OSPF[40] and IS-IS disseminate connection information among routers, specifically regarding the status of their links. Building upon that knowledge, each router formulates an internal graph of the network. Subsequently, they implement a shortest path computation algorithm on the graph, such as Dijkstra's algorithm. The resultant collection of optimal (i.e., shortest) paths is their routing table. Those protocols (RIP, OSPF, IS-IS) are employed to disseminate reachability information within a single autonomous system. Consequently, they are referred to as Interior Gateway Protocols (IGPs). Routing information is sent between separate autonomous systems via the Border Gateway Protocol (BGP)[46], which is a standardized Exterior Gateway Protocol (EGP). BGP is classified as a path-vector routing protocol. It is essentially analogous to distance-vector protocols, but with additional characteristics employed in the optimal route selection procedure. As an example, one can establish a local preference value that supersedes the declared topological distance[34].

2.1.2 Label Switching Technologies

MPLS

The IP protocol, along with the Asynchronous Transfer Mode (ATM) and Frame Relay protocols, inspired the development of the MPLS protocol. The main driving force for the creation of MPLS (Multiprotocol Label Switching) was the enormous rise in data volumes brought on by the Internet's rapid global spread, which IP and ATM (its key predecessors) could no longer manage effectively[48]. Using various networking protocols and standards, these technologies came together to create MPLS, which has become increasingly popular over time. Often referred to as Layer 2.5, MPLS functions between Layer 2 (L2) and Layer 3 (L3) of the OSI paradigm. By examining additional packet headers only at the network edges rather than at each intermediate node, it achieves its primary goal of reducing data processing time as packets move across the network[45].

For many networks to fulfill Service Level Agreements (SLAs), traffic engineering is necessary. A service provider might, for example, provide a low-latency path—which might not be the fastest path—between two customer sites. Instead of isolating the particular low-latency flows, operators could change link metrics to reroute traffic along the desired route, but such modification frequently affects the entire network. MPLS was created especially to advance this kind of per-flow traffic engineering, and to achieve this, it uses virtual circuits that resemble ATM networks. The core of MPLS is a stack of 20-bit labels that are placed on top of the packet's IP header (see Figure 2.1). Each label represents a distinct label-switched path (LSP), or virtual circuit. Before sending the packet to the next hop, a router that receives a packet with an MPLS label stack examines the top label and performs an action like popping the top label, pushing a new label onto the stack, or swapping it with another label. MPLS routers usually establish full-mesh sessions using the Label Distribution Protocol (LDP) to spread label reachability, allowing peers to create label tables for LSP implementation. Although MPLS provides useful services, there are significant drawbacks to its traffic steering policy. Because each resource reservation retains state across all routers in the virtual circuit, it first encounters scalability issues.

Second, it does not make effective use of Equal-Cost Multi-Path (ECMP) routes; using circuit replication as a solution just makes the state overhead worse. Third, during flow management, the distributed routing protocols may cause temporary inconsistencies and unforeseen side consequences[34].

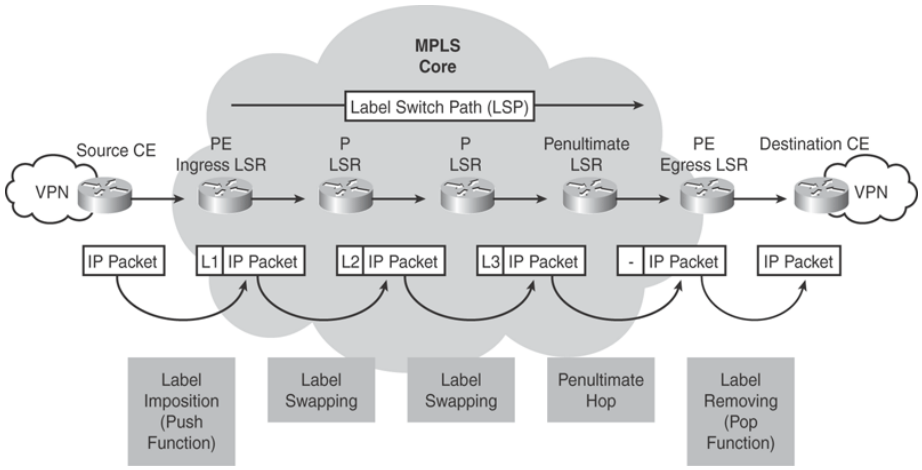

Figure 2.1: MPLS Label Stack [31]

SR MPLS

The MPLS protocol, despite bringing numerous improvements, remains imperfect and exhibits several shortcomings. The MPLS protocol is segmented into two planes—the forwarding plane and the control plane. It is the latter that encounters issues such as the emergence of traffic blackholes, the absence of traffic management support for the LDP protocol, or the excessive complexity of the RSVP-TE protocol. Furthermore, the MPLS features are inadequate for SDN implementation.
SR-MPLS has developed solutions to address these challenges while also providing a straightforward implementation for existing MPLS networks, allowing for quick resolution of potential issues[20]. The SR-MPLS addresses:

- Distribution of MPLS labels through IGP/BGP protocol extensions, thereby removing the necessity for LDP/RSVP-TE,
- Enhanced network scalability is achieved through the use of SIDs,
- Traffic management that does not require an external protocol,
- Simplification of route computation, as only ingress nodes perform recalculations and modifications of routes,

- Support for a potential network transition to SDN

SR-MPLS operates with three distinct types of segments—Prefix segment (denoting the destination address prefix with prefix SID), Adjacency segment (representing a connected link with adjacency SID), and Node segment (indicating the node's IP address configured as a prefix on the loopback interface with node SID)[20, 5]. Following the manual configuration of these SIDs, or in the case of dynamically assigned Adjacency SIDs distributed among devices via IGP or BGP within an SR-MPLS network, LSPs and Traffic Engineering tunnels can be established. This approach not only eliminates the necessity for LDP and RSVP-TE protocols but also decreases hardware demands and further minimizes the vulnerability scope of the MPLS network through the utilization of fewer technologies.

This directly relates to simplified network state management. There is less room for an error to occur anywhere in the network when fewer protocols are in use, which also reduces the amount of maintenance required to keep the network working, particularly when new devices are added. With centralized control over network routing paths, their computation, and distribution—all of which are delegated to the SDN controller—managing network nodes is made even simpler. This reduces the quantity of data that each node must retain. Furthermore, because pathways can quickly adjust to changes and offer a backup route for packet forwarding, SDN controllers enhance network recovery in the event of a link loss. All of the benefits listed above show that SR-MPLS is a real improvement over MPLS. It not only makes computation and path management better, but it also uses fewer protocols, requires fewer resources, and "protects" the network from new technologies[45].

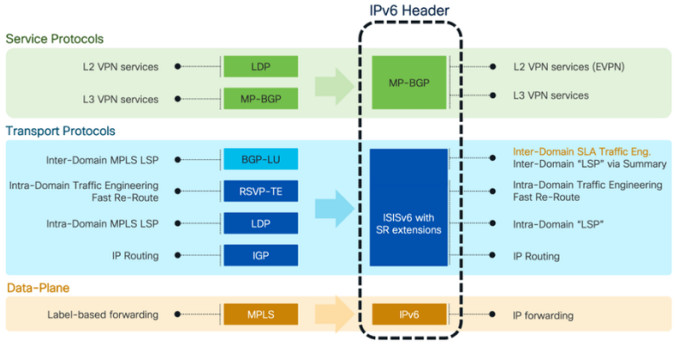

Figure 2.2: Network Transition from MPLS to SRv6 [4]

2.2 Source and Segment Routing Approaches

2.2.1 Source Routing

Source routing is a network routing technique conceptualized in the early 1970s and formally defined a decade later[44], serving as a precursor to segment routing. In source routing, the entire path (or at least a significant part of the route) that the packet will take through the network is predefined by the sender (i.e., the source node) by including instructions directly in the packet header. In conventional IP routing, each router in the path determines the next hop based on its routing table. This approach is in contrast to what was just described for source routing. Although the original IPv4/IPv6 source routing mechanisms were rarely used in practice (mainly due to security and scalability concerns), they remain important because they laid the conceptual foundation for the more modern and agile segment routing architecture.

2.2.2 Segment Routing

The Segment Routing architecture fulfills the same purpose as Source Routing, on which it was based. It also introduces enhancements in scalability and reductions in network load. Nowadays, the term Segment Routing is primarily associated with two protocols—MPLS (SR-MPLS) and IPv6 (SRv6)—with SRv6 constituting the main focus of this thesis. Furthermore, Segment Routing offers a broader range of features and functionalities compared to Source Routing.

Segment Routing employs instructions—segments—that explicitly direct the flow of data within the network. Each of these instructions is assigned its label. This is known as a Segment Identifier (SID), and a collection of such instructions is designated as a Segment Routing Policy. Subsequently, these instructions are employed to determine the manner in which the packet will be processed, such as selecting the specific interface within a given node through which it will be transmitted to the destination node[45, 20].

Instructions

Segment Routing Instructions, or segments, function as the protocols governing data operations within the network. They can specifically define nodes, network pathways, jumps, or service functionalities. Each segment is then linked to a corresponding Segment Identifier. This denotes particular instructions.
Segment Routing enables network administrators to formulate distinct instructions that facilitate explicit management of network segments based on specific requirements. Nonetheless, Segment Routing encompasses a number of predefined instructions, including:

- Global segment — a segment whose instruction is incorporated within and defined in the Segment Routing domain,
- Local segment — a segment whose instruction is incorporated within and defined by a node,
- IGP segment — the segment associated with instructions bound to the IGP,
- Binding segment — a specific segment that associates a packet with a rule it is required to follow,
- Prefix segment — the segment that corresponds to the network prefix,
- Node segment — a segment linked to a particular node within the network,
- Adjacency segment — a segment that denotes the connection between two contiguous network nodes[45].

Control planes

Segment Routing can be implemented across three distinct control planes.

- Distributed plane — routing decisions are allocated among routers. This ensures powerful scalability, fault tolerance, and rapid decision-making,
- Centralized plane — decision-making is executed by a central authority or multiple subordinate entities. This feature alleviates the burden on devices

and enables the central authority to implement universal regulations across the network, thereby streamlining management,

- Hybrid plane — integrates advantages from both centralized and distributed systems. In a distributed context, segments are assigned and indicated by routing protocols, such as IS-IS, OSPF, or BGP. Subsequently, the node autonomously determines packet management in accordance with the Segment Routing Policy and computes this policy as well.

In the centralized framework, segments are assigned and instantiated from a central location. The central point determines which nodes will manage traffic and the manner of management, based on the Segment Routing Policy computed by the central point. The comprehensive Segment Routing architecture does not limit the central authority in its policy formulation within the network.
In actuality, the hybrid scenario only introduces a central node to the basic distributed control plane[45].

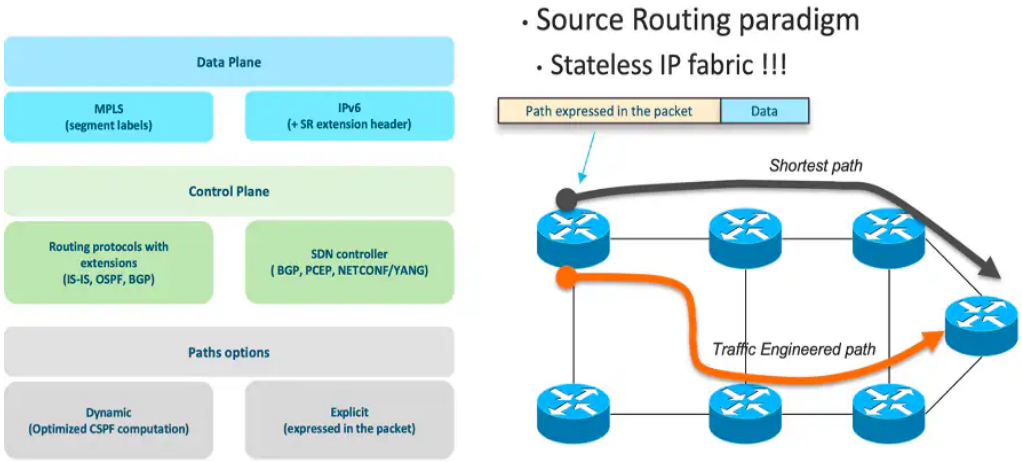

Figure 2.3: Segment Routing Architecture Overview [9]

2.3 Segment Routing over IPv6 (SRv6)

The convergence of SDN, NFV, and the emergence of SRv6[20, 18]

In the past decade, a new paradigm called Software-Defined Networking (SDN)[43] emerged initially through OpenFlow[37] and attracted massive traction from academia, vendors, and network operators. The central concept of SDN is the ability to

decouple the control (routing) plane from the data (forwarding) plane, moving network control into a logically centralized control platform. This architecture enabled operators to configure and manage networks dynamically through software (hence the acronym), moving away from rigid, resistant-to-change routing protocols.

Even though OpenFlow exhibited the promising potential of SDN, it also revealed flaws in its design, such as per-flow state overhead and elevated sensitivity to failures, e.g., Single Point of Failure (SPOF). As a result, SDN grew from its original switch-level implementation to router-level deployment. It also became part of larger frameworks and orchestration platforms, and it eventually became a major force behind modern network design.

Network Function Virtualization (NFV)[14] followed shortly thereafter and gained widespread attention while acting as a complementary paradigm to SDN. Whereas SDN provided the centralized "brains" for controlling the network, NFV's premise was to shift network functions (like firewalls, routers, and load balancers) away from dedicated proprietary hardware, running them as software on standard, commercial off-the-shelf (COTS) servers, replacing the physical "tools" or appliances with virtual ones. In combination, SDN and NFV deliver cloud-native, virtualized networking that is flexible and programmable, providing better service delivery with reduced costs and improved agility.

This brings us to SRv6, a cutting-edge architecture that builds on the strengths of SDN and NFV. SRv6 leverages its predecessors' most critical features—the programmability of SDN and the service chaining capabilities of NFV—to enable precise traffic engineering and simplified network operations. In this context, SRv6 is not only an evolution of routing and forwarding paradigms but also a key enabler and catalyst for state-of-the-art network architectures.

2.3.1 Architecture and Network Programming Model

Locators and SIDs

Apart from configuring a standard IPv6 network, what is required to enable SRv6 functionality?

The most significant and fundamental aspect of SRv6 configuration is the allocation

of SRv6 locators to each router in the transport infrastructure[18].

However, it is important to consider the definition of an SRv6 locator.

An SRv6 locator is essentially an IPv6 address' subnet. In addition to allocating a host IPv6 prefix for loopback (/128), each router must also be assigned an IPv6 subnet (for example, /64).

And what use does this locator serve?

Fundamentally, the SRv6 locator serves as the foundation for the SRv6 Segment Identifier (SID). A classic (full) SRv6 SID is a 128-bit identifier, equivalent in size to an IPv6 address, composed of the SRv6 locator (the most significant bits) and other values that specify the locator's function.

Based on this definition, it can be inferred that a single router may be linked to multiple SRv6 SIDs. Each of these SIDs is defined using the SRv6 locator that corresponds to the same IPv6 prefix.

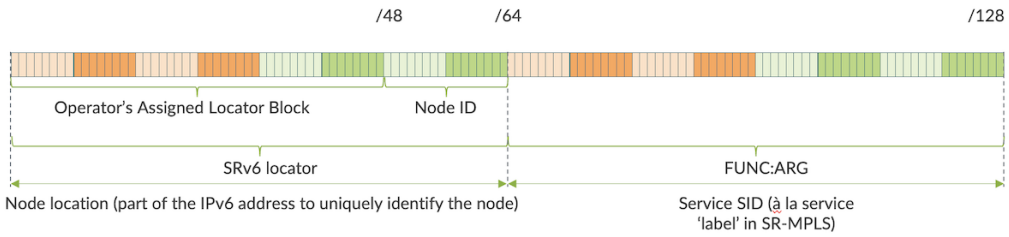

Figure 2.4: SRv6 SID structure [53]

The second flavor, which is a more compact variant, known as compressed (micro) SID, commonly called uSID, has been introduced to further reduce the overhead introduced by long SRv6 SID lists.

uSID allows multiple SIDs to be encoded within a single IPv6 address by encoding several 16-bit or 32-bit micro-SIDs within one locator block, thereby reducing the SRH size and improving scalability and forwarding efficiency in SRv6 networks.

Segment Routing Header (SRH)

An equally important component for the functionality of SRv6—that is, besides the allocation of SRv6 locators and the configuration of SIDs on each router—is the use of the Segment Routing Header (SRH)[19].

The SRH is a new and specific type of IPv6 Extension Header (Routing Header Type

4) that the ingress (headend) node inserts into every IPv6 packet. This provides the ability to the sender to specify in advance an exact sequence of segments that the packet must follow within the network, in which the SRH functions as an "envelope" containing an ordered list of SIDs.

Each SID in the list represents an instruction or an intermediate point that the packet must visit. This enables advanced source routing without the need to maintain a state on every intermediate router.

The SRH format is relatively simple but designed for efficiency. Its key fields are shown in Figure 2.5.

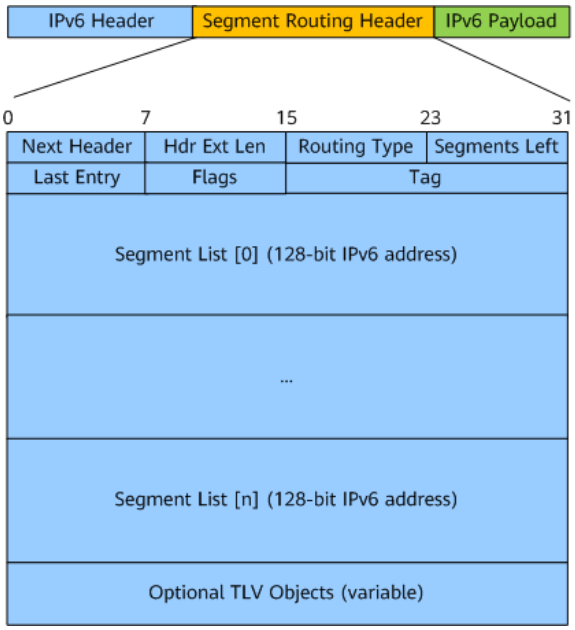

Figure 2.5: Segment Routing Header (SRH) structure [26]

- Next Header: Indicates the type of the following header (e.g., TCP, UDP, or another IPv6 extension header).
- Hdr Ext Len: Fixed length of the SRH, excluding the first 8 bytes.
- Routing Type: Always set to 4, so that it can be identified as SRH.
- Segments Left: A pointer that shows how many segments remain to be processed and is decremented by 1 at each SRv6 node.
- Last Entry: Index of the last SID in the Segment List.
- Flags: It contains various flags for specific functions (e.g., OAM, protection).

- Tag: This is used in specific cases for grouping or tagging packets. Not always necessary.
- Segment List: The actual ordered list of 128-bit SIDs that form the path (Segment List [0] indicates the last segment of the path while Segment List [n] indicates the first one).

In addition, the SRH may also contain optional TLV (Type-Length-Value) objects for additional information such as Padding and Hash-based Message Authentication Code (HMAC) TLVs.

The SRH processing starts when a router receives a packet with an SRH, and it checks the active SID (which is the current Destination Address). If this matches a local SID, it decrements the Segments Left counter and replaces the Destination Address with the next SID in the list. This procedure is repeated until Segments Left is set to zero, at which point the packet is forwarded as usual to its final destination. An important differentiator to note is that, in the case of SRv6, the segments remain in the SRH after processing, which differs from SR-MPLS, where labels are popped. One main benefit to that is that the header remains intact, allowing path tracing and future extensions (through optional TLVs).

This mechanism ensures SRv6 remains fully compatible with standard IPv6 forwarding on transit nodes that do not support it, an important feature that demonstrates its powerful source-routing capabilities[19, 18].

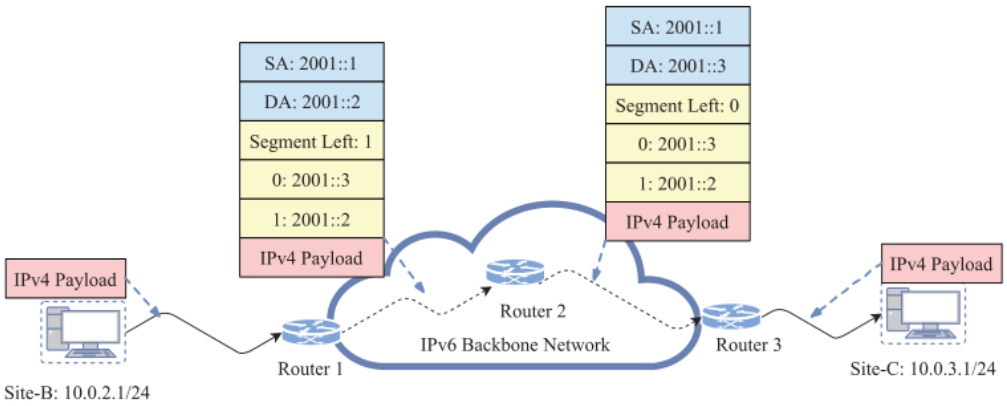

Figure 2.6: An example to show SRv6 packet forwarding process [47]

In order to implement practical services, SRv6 defines a set of well-known SID functions. These functions are embedded in the Function/Argument section of an IPv6 address and are executed locally on the router when the destination address matches a local SID. According to the SRv6 Network Programming RFC[18], the core behaviors are divided into two main categories: Transit behaviors (T/T.) and Endpoint behaviors (E/End). In the former, transit nodes forward packets based on the IPv6 destination address without inspecting the SRH, whereas in the latter, endpoint nodes actively process the SID in the SRH.

The most commonly used are the following:

- End SID: Indicates the End behavior SRv6, representing a node SID that identifies a destination node. The packet is processed locally at the node.
- End.X SID: Indicates the End.X behavior SRv6, representing a SID bound to a specific outgoing interface or next-hop.
- End.DT4 / End.DT6 SIDs: Indicate the End.DT4 behavior SRv6 and End.DT6 behavior SRv6, which perform SRv6 decapsulation and forward the inner IPv4 or IPv6 packet using a lookup in a specific routing table (global routing table or a VRF instance). They are used to deliver Layer 3 services (including L3VPN) over an SRv6 underlay. When a packet arrives at an End.DT6 SID, the node strips the SRH and performs a lookup in the corresponding VRF, exactly as required for the L3VPN service implemented in this thesis.

These functions give SRv6 its network-programming capability, allowing both path steering and service identification within the same header.

Function	Behavior	Reference
End	Endpoint with shortest path (with/without PSP/USP)	SRv6 instantiation of a prefix SID
End.X	Endpoint with a layer-3 cross-connect (with/without PSP/USP)	SRv6 instantiation of an Adj SID
End.DX6	Endpoint with decapsulation and IPv6 cross-connect	IPv6 L3VPN use (equivalent of a per-CE VPN label)
End.DT6	Endpoint with decapsulation and IPv6 table lookup	IPv6 L3VPN use (equivalent of a per-VRF VPN label)
End.DX4	Endpoint with decapsulation and IPv4 cross-connect	IPv4 L3VPN use (equivalent of a per-CE VPN label)
End.DT4	Endpoint with decapsulation and IPv4 table lookup	IPv4 L3VPN use (equivalent of a per-VRF VPN label)
End.DX2	Endpoint with decapsulation and Layer 2 cross-connect	L2VPN use case

Figure 2.7: Common SRv6 Functions: Steering and Services [32]

2.3.2 Traffic Engineering (TE) Mechanisms and Policies

SRv6 Work Modes

Traffic Engineering (TE) and Best Effort (BE) are the two work modes in which SRv6 can operate. These two modes differ slightly in their operational approach and emphasize distinct categories of services. In BE mode, traffic follows the native IPv6 shortest path, whereas in TE mode, explicit policies are employed to enforce customized routing paths. Accordingly, based on the network specifications, the administrator is able to determine the appropriate operating model to implement[45]. The Best Effort (BE) mode will not be examined further in this thesis, as it is beyond the scope of the current study.

SRv6 Traffic Engineering

When employing SRv6 TE, it is advisable to specify explicit paths from source nodes, which are often bundled with a variety of metrics[20]. For instance, bandwidth, delay, packet loss rate, and throughput are examples of such metrics. In terms of network functionality, it is useful in scenarios where strict performance requirements are established, where there is a need to optimize traffic engineering on certain routes, or in networks where the administrator wants to manage additional metrics alongside specific route management. Backbone networks and Software-Defined Networks (SDNs) are examples of such networks[45].

Segment Routing Policy

An SR Policy is a framework that allows for the creation of an ordered list of segments on a node. This framework is used to implement a source routing policy, which directs traffic for a specific purpose (e.g., to meet a specific SLA) originating from that node. As previously mentioned, Segment Routing introduces enhanced Traffic Engineering capabilities by allowing operators to direct traffic based on their preferences and the characteristics of the packets as they traverse the network. This is made possible because the state information is consistently available within the packet through the managed Segment List. To ensure that packets are associated with the correct Segments and to guarantee the proper path traversal, the concept of SR Policies

was introduced. An SR Policy serves as a configuration that assigns the appropriate Segments to packets that match specific criteria.
Each SR Policy is uniquely identified by three elements:

- Headend: the IPv4 or IPv6 address of the node that actually instantiates the policy. Typically, it is a unique node within the SR domain.
- Color: a 32-bit numeric value that links the policy to differentiate between dissimilar traffic objective policies, such as low-latency or high-bandwidth.
- Endpoint: the IPv4 or IPv6 address that is the final destination of the policy. If it is unspecified, the last segment in the list defines the destination[33].

Using this technique, individual packets can be treated differently based on their characteristics. Each policy is therefore unique according to the introduced tuple (headend, color, endpoint).
In summary, with this approach, it is feasible to deliver fine-grained traffic steering according to policies and meet stringent service requirements by eliminating intermediate per-path states[20]. Overall, the architecture provides a flexible and scalable framework for intent-based traffic engineering in SRv6.

2.3.3 L3VPN over SRv6

One of the most common uses of SRv6 is the provision of Layer-3 Virtual Private Network (L3VPN) services. Traditional MPLS networks achieve customer isolation through the use of VPN labels. When a packet arrives at the egress Provider Edge (PE) router, the MPLS label "tells" the router to look up the correct Virtual Routing and Forwarding (VRF) instance for the customer.
In SRv6, the same function is implemented using special SRv6 Segment Identifiers known as End.DT4 and End.DT6 behaviors[18]. According to SRv6 Network Programming[18], the End.DT4 behavior describes the "Endpoint with decapsulation and specific IPv4 table lookup." When a router receives a packet whose Destination Address matches a locally configured End.DT4 SID, the End.DT4 SID decapsulates the outer IPv6 header, including the SRH, and then performs an IPv4 lookup in the specific VRF table that corresponds to that SID. The End.DT6 behavior does

exactly the same for internal IPv6 packets.

This approach eliminates the need for MPLS VPN labels, as the SRv6 SID itself serves as an indicator for the correct tenant routing table. BGP is used to exchange L3VPN prefixes between PE routers, using the VPNv4 and VPNv6 address families (inet-vpn and inet6-vpn). According to BGP Overlay Services Based on Segment Routing over IPv6 (SRv6)[10], each prefix is advertised along with the corresponding SRv6 Service SID (End.DT4 or End.DT6) within the BGP Prefix-SID attribute. In this way, the ingress PE can forward the packet to the correct egress PE using the appropriate SRv6 SID without needing to maintain a state in the core network.

In the lab for this thesis, the L3VPN service is implemented exactly in this way. The VRF named "blue" uses End.DT* behaviors, while the BGP session between the PE routers advertises the corresponding VPNv4/VPNv6 routes along with the SRv6 SIDs. This architecture allows for full compatibility with classic L3VPN services while also leveraging the advantages of SRv6's simplicity, programmability, and absence of MPLS.

The following figure depicts the delivery of an L3VPN service over an SRv6 network and the packet encapsulation at different nodes within the network.

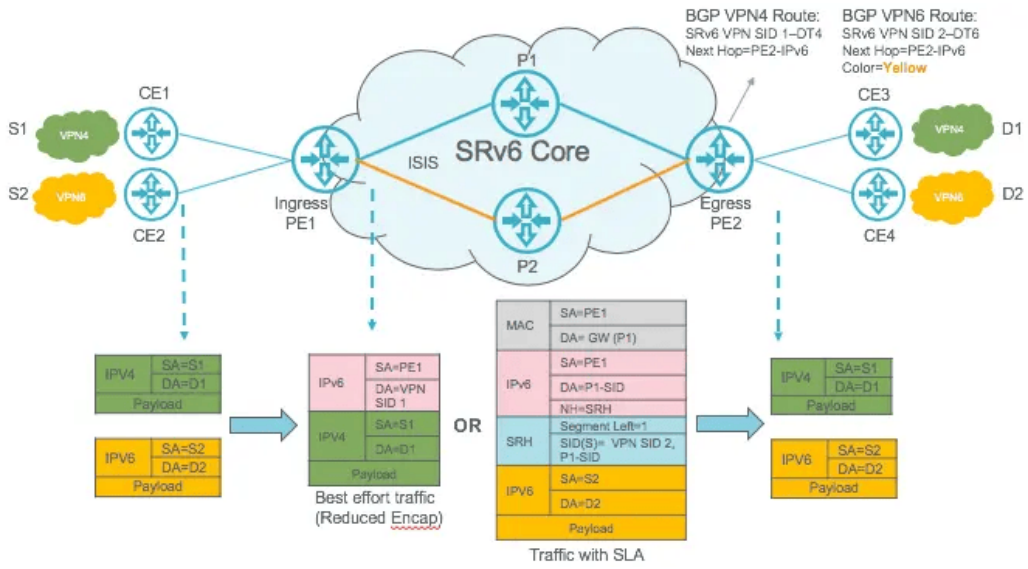

Figure 2.8: Illustration of L3VPN service delivery over an SRv6 transport network [32]

2.4 Containerization and Network Virtualization

2.4.1 Virtualization

In computing, virtualization is defined as the technology that enables the creation of virtual environments from a single physical machine, allowing for more efficient use of resources by distributing them across computing environments. Using software, virtualization creates an abstraction layer (known as a hypervisor) over computer hardware, dividing a single system's components such as processors, memory, networks, and storage into multiple virtual machines (VMs). Each VM runs its own operating system (OS) and behaves like a separate physical computer, despite sharing the same underlying hardware[28]. This concept has been a cornerstone of computer science since the 1960s and continues to be widely deployed to this day, more so in the context of cloud computing. Today, virtualization is applied at various system levels including: operating system-level virtualization, also known as containerization, which shares the same kernel with the host operating system; hardware virtualization, also referred to as platform or hypervisor-based virtualization (often simply called VM virtualization), which simulates a complete hardware environment; and server virtualization, which partitions a single physical server into multiple independent virtual machines (VMs).

2.4.2 Containerization

The introduction of c-groups (control groups) in the mainline Linux kernel in 2008 marked the first public availability of this technology. This innovation proved highly influential in the development of all subsequent containerization technologies available today, and containerization as a whole gained prominence in 2014, shortly after the introduction of Docker. Each container is essentially a software entity that encapsulates all its code, dependencies, libraries, etc., into a single lightweight and portable executable unit, rendering it independent of the underlying architecture and delivering portability, scalability, and isolation. All of these features contribute to making the deployment of any application a quick and straightforward process,

whether it is installed on a desktop computer, a traditional IT infrastructure, or the cloud, as it provides seamless operation in any computing environment.

Even though they are fundamentally different technologies, containers and virtual machines are often compared because they offer similar capabilities. However, the key difference lies in their resource usage; containers typically require fewer resources and are lighter than VMs. In traditional virtualization, a hypervisor virtualizes physical hardware. The result is that each virtual machine contains a guest OS, a virtual copy of the hardware that the OS requires to run, and an application and its associated libraries and dependencies. Instead of virtualizing the underlying hardware, containers virtualize the operating system (typically Linux or Windows) so each individual container contains only the application and its libraries and dependencies. Furthermore, unlike a virtual machine, containers do not need to include a guest OS in every instance and can, instead, simply leverage the features and resources of the host OS[27]. Therefore, containers are significantly more lightweight, portable, and resource-efficient in terms of memory, CPU, and storage space than virtual machines (VMs).

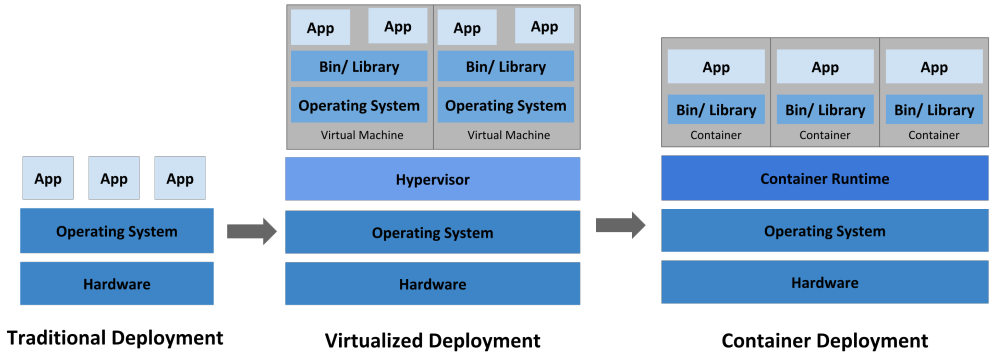

Figure 2.9: Virtual Machine vs. Container Architecture [54]

2.4.3 Docker

Being an open-source technology based on Linux containers, Docker remains the most popular containerization platform since its introduction in 2013. In fact, the widespread use of Docker has led to the interchangeable use of the terms "Docker" and "containerization". It is essentially a container engine that leverages features of the Linux kernel (such as namespaces and cgroups) to create and run lightweight, isolated

containers directly on top of the host operating system. This approach automates the deployment, scaling, and management of containerized applications[11].

Docker provides an efficient and consistent workflow for moving applications from a developer's local machine and testing environments all the way to production. In our case, this same workflow was followed to install and run the application (along with containerlab and FRR routers) on the Ubuntu virtual machine. The components shown in the image below were used to successfully complete the installation.

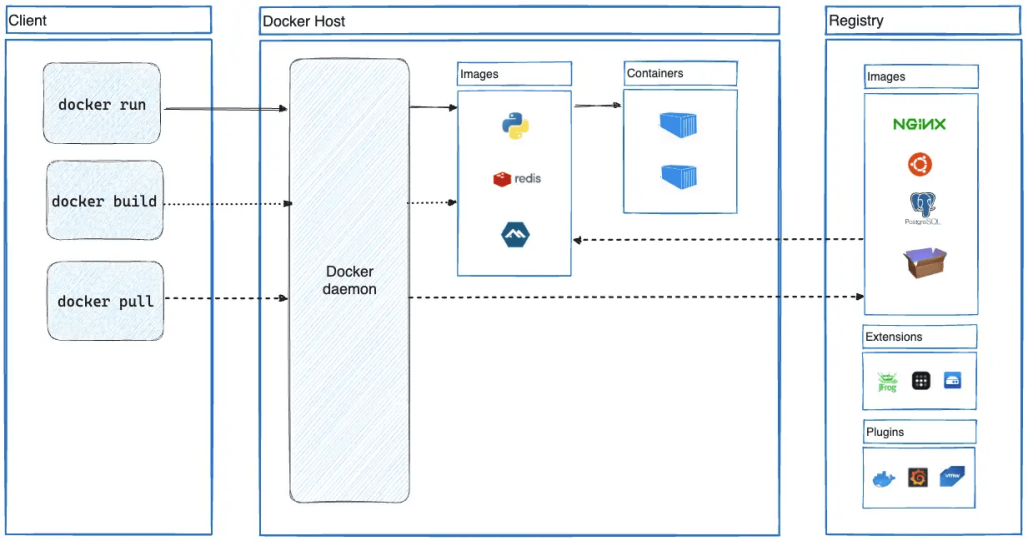

Figure 2.10: Docker Architecture [11]

Docker uses a client-server architecture. The Docker daemon (dockerd) acts as the server and is responsible for all operations related to containers, such as building, running, and managing them. The Docker client (the docker command-line interface) sends requests and commands to the daemon, either through a local Unix socket or via a REST API.

The Docker client can run on the same host as the Docker daemon or on a completely different machine. In our setup, we first installed the Docker Engine on the Ubuntu VM. This installation automatically included and started the Docker daemon (dockerd) on the virtual machine itself. Since both the client and the daemon run on the same Ubuntu VM, the communication happens locally through the default Unix socket (/var/run/docker.sock).

Images are a fundamental component of Docker, as containers are built from them. Images can be configured according to the application and used as templates for

creating new containers. For example, to create a lab with FRR routers using containerlab, you only need to start from the official FRR Docker image and add your own configurations (and optionally custom startup scripts or additional packages). Then, containerlab uses these images as the base to spin up the routers as containers. Docker offers multiple networking modes to facilitate container communication. The bridge mode, which is the default for most Docker setups and Containerlab topologies, where the Docker daemon creates a virtual bridge (docker0) that acts as a software switch. Containers are attached via virtual Ethernet (veth) pairs, enabling isolated yet interconnected communication on the same host. This configuration is particularly suitable for experimental network topologies, such as the Containerlab-based SRv6 setup in this thesis, where point-to-point links between FRR containers are established through host-side bridges (e.g., br-xxxx) and veth interfaces. In contrast, host mode shares the host's network namespace for maximum performance but reduces isolation, and overlay mode enables multi-host communication in distributed environments—neither of which was required for this single-host experimental setup[51, 21].

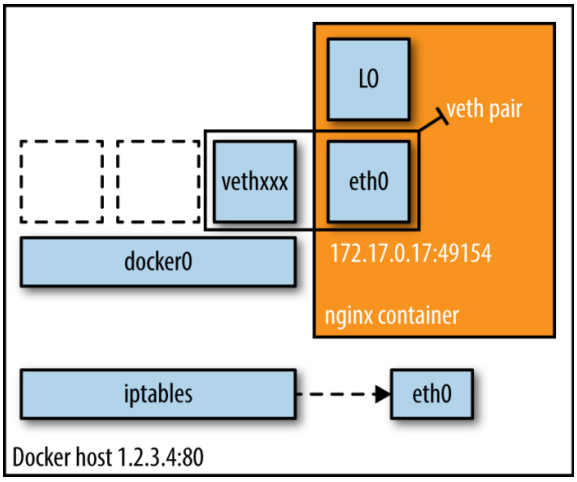

Figure 2.11: Docker Bridge Networking Mode [24]

2.5 Open-Source Routing Daemons

Selecting the optimal routing platform is crucial for ISPs, cloud providers, and CDNs. The shift toward open-source routing daemons represents a significant departure from traditional proprietary router platforms offered by vendors such as Cisco's XRd (a

containerized version of IOS XR) and the Juniper's Cloud Native Router (a containerized version of Junos OS). By decoupling routing software from specialized hardware, operators gain greater flexibility, lower costs, and the ability to run full-featured routing stacks directly on commodity servers, virtual machines, or containers. This approach has gained substantial traction in cloud data centers, Internet exchanges, and virtualized network environments, where rapid innovation and vendor independence are critical[21].

A Routing Daemon, in UNIX terminology, is a non-interactive background application that manages the dynamic aspects of Internet routing. It communicates with other routers, computes routing tables, and transmits them to the OS kernel, which performs the actual packet forwarding.

Among the prominent open-source alternatives, BIRD[6], originally developed as a school project at Charles University, Prague, stands out as a lightweight, high-performance daemon particularly popular for BGP route servers and Internet Exchange Points due to its advanced filtering capabilities and efficient memory usage. ExaBGP[16], on the other hand, adopts a different philosophy: it is a Python-based BGP "conduit" designed primarily for automation and scripting rather than kernel-level forwarding, making it ideal for route injection and monitoring tasks but less suited for full routing-table management.

The core of this work, however, relies on FRRouting (FRR), the most widely adopted open-source routing suite today, and supports virtually every major routing protocol and address family. FRR is now governed by the Linux Foundation, with contributions from companies including NVIDIA, Google, Netflix, and Microsoft. It implements a modular, daemon-based architecture in which individual protocol daemons (e.g., bgpd, isisd, ospfd) communicate with a central zebra daemon. Zebra acts as the routing information base (RIB) manager, installing routes into the Linux kernel and coordinating forwarding decisions. This design provides strong isolation, fault tolerance, and extensibility. It comprises a structure called thread-master, which is a context that keeps track of pending as well as currently executing tasks. Its familiar Cisco/Juniper-style CLI (vtysh) and native container compatibility make

it an excellent choice for reproducible lab topologies such as the one deployed in this thesis using Containerlab.

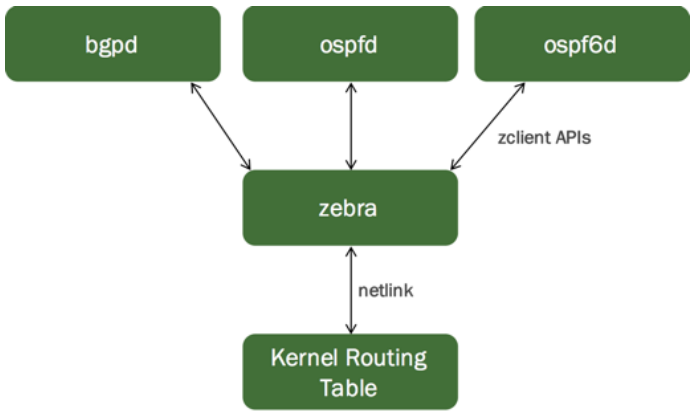

Figure 2.12: Free Range Routing (FRR) Architecture [42]

This modular architecture[21] has also attracted attention in the research community. For instance, Wirtgen et al. [55] introduced xBGP, a programmable extension framework built directly on top of FRR (and BIRD), enabling safe, vendor-neutral BGP extensions through an eBPF-based virtual machine. Their work demonstrates how open-source daemons can accelerate innovation in routing protocols without vendor lock-in, paving the way for advanced features such as custom policies and telemetry that could further enhance SRv6-based traffic engineering in future containerized deployments.

2.6 Related Work

One of the first major contributions to SRv6 came from UCLouvain. In 2017, Lebrun and Bonaventure developed the first open-source implementation of IPv6 Segment Routing directly in the Linux kernel[35]. Their work enabled core behaviors such as End, End.X and basic encapsulation functions. This implementation quickly became an important foundation for most of the SRv6 research that followed and had a strong influence on the evolution of the Linux data plane. Since then, the UCLouvain group has produced several master's and PhD theses exploring different aspects of SRv6—mostly in Linux-based environments.

In February 2025, the ITU-T published the Technical Report QSTR.MPM-SRv6[30], which examines key performance metrics such as latency, jitter, packet loss, and throughput for SRv6 networks. The report outlines both active and passive measurement approaches with specific requirements based on the data plane and the control plane, with a particular emphasis on extensions of the STAMP protocol [38, 39, 3]. The report distinguishes between monitoring requirements and monitoring methods and takes into consideration both the data management infrastructure and the techniques used to obtain data about nodes and individual traffic within the SRv6 network. In addition, the report discusses control plane technologies, specifically southbound (gRPC/gNMI, NETCONF/YANG) and northbound (RESTful API, SNMP) implementations. Finally, yet importantly, the report identifies several open challenges and research gaps, highlighting the need for further experimental validation and performance evaluation studies in realistic SRv6 test environments. This gap motivates and provides the foundation for the work presented in the current thesis.

Abdelsalam et al.[2] introduced SRPerf, an open-source extensible framework developed to assess SRv6 data-plane performance in both hardware and software scenarios. SRPerf is designed in accordance with the network benchmarking guidelines outlined in Benchmarking Methodology for Network Interconnect Devices[7] using metrics such as Partial Drop Rate (PDR), latency, and throughput. Initial evaluations on the Linux kernel implementation (versions 4.10+, with emphasis on 4.18+) revealed notable overheads in specific SRv6 behaviors such as T.Encaps, T.Insert, End.X, End.DX6, primarily due to routing bottlenecks (routing cache misses) and memory allocation during encapsulation/decapsulation. In most cases, performance was similar compared to plain IPv6 forwarding. Subsequent extensions to the framework[1] enabled direct comparisons with VPP (Vector Packet Processing), where optimized configurations achieved much more performant forwarding (almost line-rate), demonstrating SRv6's potential in more efficient implementations.

Duchene et al.[12] suggested using SRv6Pipes to create a service function chain (SFC) by linking the Segment Routing Header (SRH) to a data stream, like a TCP flow, through system calls in the kernel. The SR-aware TCP proxy provided transparency

within the kernel's network stack. To perform various network tasks on the proxy, a sufficient IPv6 address space is allocated. This address space consists of unique identifiers for devices on the network, designated for the proxy, the tasks it executes, and their parameters.

Xhonneux et al.[56] applied eBPF[13] to SRv6 to process SRHs in the kernel. They showed several use cases of SRv6, such as LB, ECMP next hop, and latency measurement. Likewise, measuring the pure performance in the kernel poses significant challenges.

Overall, our performance evaluation results were fundamentally different from those reported in previous studies.

Chapter 3

Methodology

The decision regarding methodology is crucial in any scientific study. The research method used in this study is classified as *experimental*, as it is designed to identify factors that influence the behavior of SRv6 in containerized environments. Additionally, the research is *applied*, as it employs a diverse array of tools and software suites. Consequently, the methodology is built on the efforts and work of other researchers and developers. Considering these factors, the methodology for this work involves selecting and configuring an appropriate environment, as well as implementing a method for conducting the experiments. The chosen method should abide by the purpose and goal of this thesis, which is to gain an adequate understanding of what is possible with SRv6 in current open-source tools through hands-on experimentation[52].

3.1 Execution Environment

To explore the research questions, several tools and software components were required for the successful completion of the experiments. Below, we outline the most significant options and offer insights into our decision to select them over more commercial alternatives.

ContainerLab[51] is a free and lightweight CLI tool developed and maintained by Nokia engineers that allows the user to spin up an entire topology and manage its lifecycle, all in an expeditious manner and without the need to install any resource-intensive tools. Upon completion of the exercise, users can simply delete the test network. Unlike other virtual lab orchestrators, such as EVE-NG[15] and GNS3[22],

which require the installation of hardware on a bare-metal server or a VM to create a virtual replica of your lab for a specific environment, Containerlab only requires users to have a Linux command-line utility to download files and Docker. It can be done on a Linux instance, in Windows, or on a Mac. A topology is a single YAML text file that can and should be checked into a Git repository. Another important differentiator is that it's not locked into a UI-driven user workflow and caters more to a Lab-as-Code (LaC) approach. The GUI-based tools allow engineers to create a true visual topology of their network, adding routers, connecting links, and working as they see fit. However, they rarely align with the CI/CD paradigm. Therefore, if the team wants to treat their lab as code and harness a Git workflow, or if they want to create and control versions, do automated testing, or collaborate with their peers—everything is handled seamlessly. In this thesis' context, Containerlab serves as the core orchestration platform for our case study, allowing the deployment of containerized SRv6-capable routers, specifically FRR images and integration with the FRR repository.

Originally developed by a group of contributing companies in the open networking space, FRR was created to streamline the routing protocol stack and to make engineers' lives much easier. FRRouting (FRR)[21] currently hosted under the Linux Foundation is a free, open-source routing protocol suite for Unix-like systems, thus replacing the need to rely on commercial implementations from companies such as Juniper and Cisco. It supports several key protocols such as BGP (with full IPv4/IPv6 integration and VPN capabilities), OSPFv2/v3, IS-IS, MPLS, and more, as well as new features like Segment Routing over IPv6 (SRv6). FRR originated as a fork of the Quagga project in 2017, which in turn was a fork of GNU Zebra and is widely used in production environments, data centers, the cloud, and white-box switches e.g., SONiC (Software for Open Networking in the Cloud)[50], as well as virtual and experimental topologies. Installing FRR on a Linux distribution gives that system the ability to function as a software-based router, providing advanced capabilities like reflector, route server, and segment routing, among others. When used alongside Containerlab—such as in our case—FRR runs inside lightweight Docker containers (frrouting/frr image), allowing us to quickly spin up topologies with hundreds or even

thousands of routers (e.g., BGP full-mesh, ISP-style peering, MPLS/VPN, and leaf-spine fabrics). Configuration is done through files (daemons + frr.conf), which are mounted at containers, or through automation (Ansible, NetLab). FRR's features make it ideal for anyone getting familiar with routing principles as well as more advanced use cases such as protocol testing and automation labs. In the present work, FRRouting (FRR) was selected as the core routing engine, enabling the implementation of dynamic routing protocols and SRv6-based forwarding within containerized nodes. In conclusion, the combination of Containerlab and FRR provides us the ability to avoid vendor lock-in and have quick, lightweight, reproducible, version-controlled networking labs with real routing, without needing expensive hardware or proprietary-based emulators. It can be thought of as the modern, DevOps-friendly choice for anyone who is into routing, network automation, or open networking in general. The VM is based on Ubuntu desktop edition 24.04.3 LTS, on which we will build and install Docker, Containerlab, and FRRouting as shown in Figure 3.1.

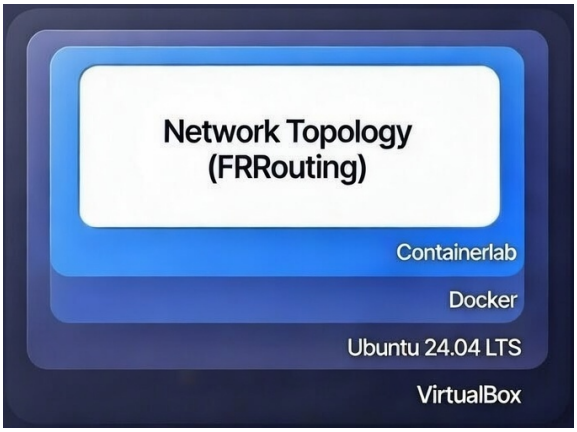

Figure 3.1: Host Platform

The majority of experiments ran locally on a workstation with an i7 CPU with 12 cores operating at 3.6 GHz and 32 GB of DDR5 RAM running at 6400 MHz.

3.2 Experimental Workflow

The work in this thesis relies on an experimental evaluation carried out in a containerized network environment that supports SRv6 forwarding and basic traffic engineering capabilities. The main idea was to create an entire testbed that was

workable enough to draw meaningful conclusions but, at the same time, simple and controllable so that measurements remained reproducible and interpreted.

Instead of creating the entire topology and building the configuration from scratch, the experimental setup builds upon the open-source srv6-labs repository[49] available on GitHub. This repository offers a collection of three subdirectories with several test topologies and case studies (centered on SRv6), ranging from basic connectivity all the way to more advanced segment routing use cases. This approach means that this repository is a suitable fit for the purposes of this thesis and serves as a solid starting point. The repository contains project information, scripts, and configurations that are relatively easy to understand and adapt, which completes a significant portion of the groundwork. Second, it mostly uses Containerlab, which is a lightweight, open-source, actively maintained, and very popular tool for network research. Last but not least, the configurations are written in YAML files in a declarative style, which makes them easy to read, modify, and include in documentation. All of the above means the labs can run locally on a reasonably powerful machine, giving flexibility depending on the available resources.

3.3 Performance Measurement Metrics

To evaluate how SRv6 behaves in a containerized environment, the existing topology was left intact so as to reduce the resource footprint and enable faster deployment and cleanup cycles. Several key performance indicators were selected early on to provide a balanced view of both the network and system impact.

The three most important metrics that were under evaluation are as follows. End-to-end latency (round-trip time via ping and more detailed one-way measurements where possible), CPU/memory usage of the containerized routers, and packet loss (which is the percentage of packets that are lost during sustained traffic). These metrics were specifically chosen to remain meaningful and comparable regardless of whether the environment uses plain SRv6 forwarding or includes more advanced components such as Cilium and Kubernetes integration. One more goal is to identify the trade-offs that directly affect the network's raw performance, i.e., how fast and

reliable the forwarding is, and its practical cost—that is, the additional load on the CPU and memory that occurs when SRv6 is deployed in a container-native application. In this way, the results remain valid and interpreted, even if the software stack changes slightly between experiments.

For the correct identification of instances of packet loss and/or latency in the network, test traffic was sent between selected communication endpoints. Controlled background traffic—if it was deemed necessary—was also simulated in certain cases in order to create more realistic evaluation conditions, rather than conducting tests on an empty network. Each measurement—more so under these circumstances—was repeated several times (usually between 5 and 10 iterations) to normalize any potential variation.

A number of small Python scripts were written using libraries such as pandas, matplotlib, and numpy to streamline the entire endeavor through automated tests, repeated measurements, and the processing and visualization of the collected data. Certain advanced features (for example, some of the more complex telemetry collectors) that were irrelevant to the main objectives were not explored further. Finally, as a final action and for the sake of consistency, all results—logs, graphs, and scripts—were saved in an organized way in the environment.

3.4 Modifications and Deployment Procedure

The labs were not used exactly as provided. Several practical adjustments were necessary to make the environment suitable for the specific measurements and research questions of this work.

As already mentioned, the lab was deployed on a local Oracle VirtualBox VM running Ubuntu 24.04 LTS. Containerlab version 0.73.0[51] was used as the orchestration tool, with Docker managing the Linux containers.

The deployment process involved:

1. Cloning the repository.
2. Navigating to the use-case topology directory (2-use-case-topologies/frr).

3. Applying minor syntax corrections to the frr.yml file using nano to ensure compatibility with Containerlab 0.73.0 (primarily indentation and deprecated field naming).

More specifically, the following modifications were required in the `frr.yml` file:

```
3.1 - Configuring image:
frr-srv6-usid-ubuntu22:rev1.3 ->
frrouting/frr:latest (or v8.4.0 if we prefer a specific version)
3.2 - Updating mgmt & subnets:
mgmt_ipv4 → mgmt-ipv4
mgmt_ipv6 → mgmt-ipv6
ipv4_subnet → ipv4-subnet
ipv6_subnet → ipv6-subnet
```

These modifications can be applied manually or, alternatively, by executing the following shell commands:

```
cp frr.yml frr.yml.backup
sed -i 's/mgmt_ipv4:/mgmt-ipv4:/g' frr.yml
sed -i 's/mgmt_ipv6:/mgmt-ipv6:/g' frr.yml
sed -i 's/ipv4_subnet:/ipv4-subnet:/g' frr.yml
sed -i 's/ipv6_subnet:/ipv6-subnet:/g' frr.yml
```

4. Executing sudo containerlab deploy -t frr.yml.

The successful deployment of the topology was confirmed by the containerlab status output (see Figure 4.5) as well as by the `docker ps -a` command, which showed all eight FRR containers in "Up" state.

The following sections describe the network topology in more detail and explain the specific implementation and configurations needed in order to get the lab up and running.

Chapter 4

Implementation & Configuration

4.1 Overview of the SRv6 Lab Topology

The experimental design is based on an 8-node SRv6 network emulated using Containerlab[51] and FRRouting (FRR)[21]. The topology follows a classic PE–P–PE design with two Customer Edge (CE) nodes and six Provider Edge / Provider (PE/P) routers, interconnected via a full-mesh underlay of eBGP and IS-IS. The goal is to implement and validate an L3VPN service over SRv6, focusing on end-to-end connectivity, encapsulation overhead, and basic traffic steering capabilities.

Figure 4.1 presents the logical topology diagram as provided in the reference srv6-labs repository [49].

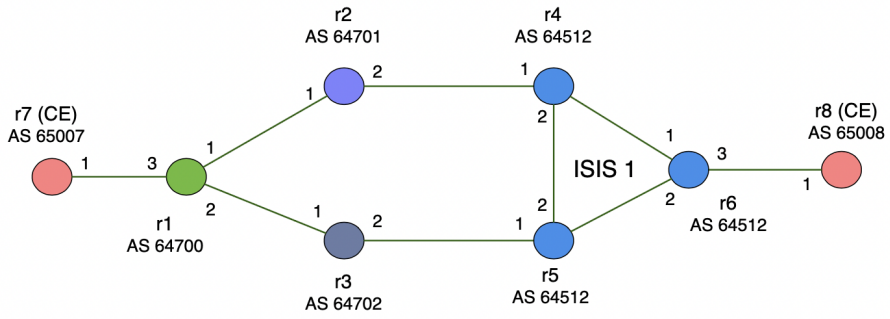

Figure 4.1: Logical topology of the 8-node SRv6 lab

Figure 4.2 illustrates that the Containerlab graph command was executed using the frr.yml topology file to create an interactive graph of nodes and links, effectively visualizing the actual deployment.

```
angelos@angelos-VirtualBox:~/srv6-labs/2-use-case-topologies/frr$ sudo containerlab graph -t frr.yml -o my-topo-graph.png
[sudo] password for angelos:
16:42:55 INFO Parsing & checking topology file=frr.yml
16:42:55 INFO building graph from topology file
16:42:55 INFO Serving topology graph
  addresses=
    http://10.0.2.15:50080
    http://172.17.0.1:50080
    http://172.20.2.1:50080
    http://[fd00::64ac:e039:9f5:6e8f]:50080
    http://[fd00::ebac:8902:324c:4c8f]:50080
    http://[fd00::a00:27ff:fee8:7950]:50080
    http://[2001:172:20:2::1]:50080
```

Figure 4.2: Output of the containerlab graph -t frr.yml command

Figure 4.3 presents a static screenshot of the actual Containerlab-rendered graph, demonstrating the container-to-container connectivity and management network. The graph is served locally on http://172.20.2.1:50080 (or alternative addresses shown), allowing real-time visualization in a browser.

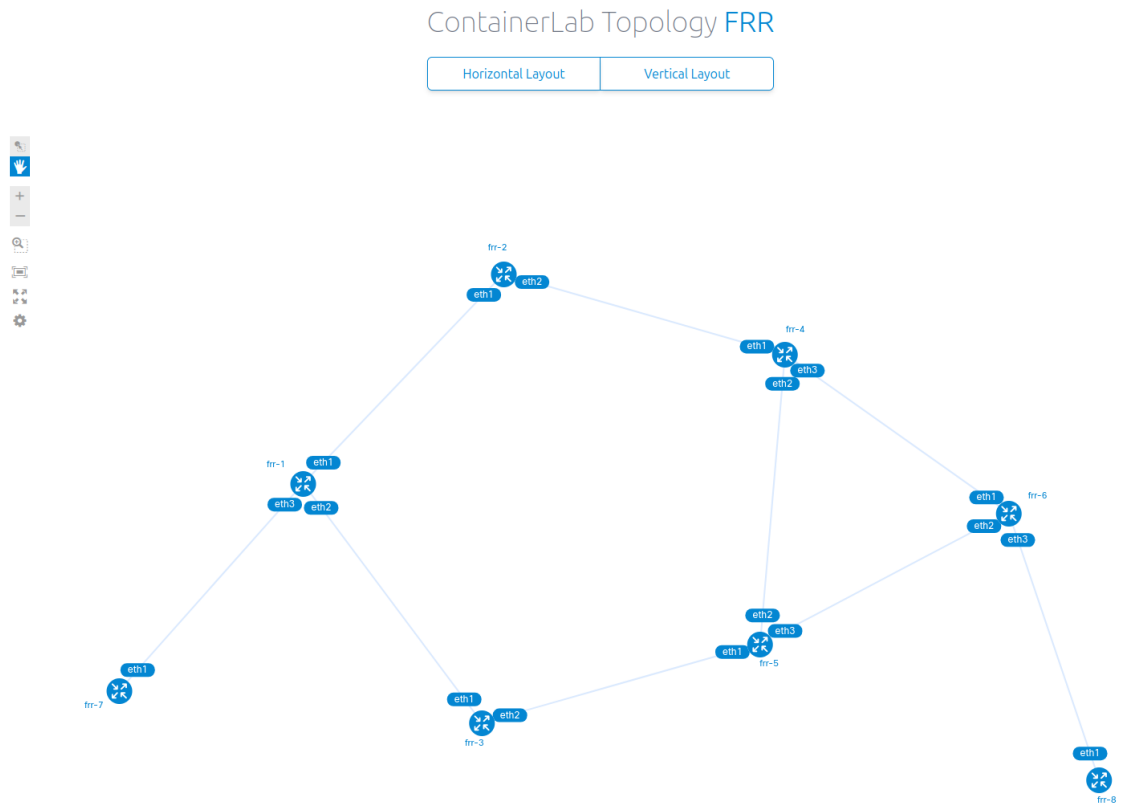

Figure 4.3: Containerlab topology graph of the deployed FRR lab

The topology consists of:

- r7 & r8: Customer Edge nodes (CEs) acting as endpoints for the L3VPN service.
- r1 & r6: Provider Edge routers (PEs) responsible for VPN label imposition/disposition and SRv6 encapsulation.
- r2–r5: Core (P) routers providing underlay connectivity via IS-IS and eBGP.

All nodes run Linux-based containers with FRRouting (version 8.4.git) as the routing stack as depicted in Figure 4.4.

```
frr-1# show version
FRRouting 8.4_git (frr-1) on Linux(6.17.0-14-generic).
Copyright 1996-2005 Kunihiro Ishiguro, et al.
configured with:
    '--prefix=/usr' '--sbindir=/usr/lib/frr' '--sysconfdir=/etc/frr' '--libdir=/usr/lib' '--localstatedir=/var/run/frr' '--enable-rpki' '--enable-vtysh'
 '--enable-multipath=64' '--enable-vty-group=frrvty' '--enable-user=frr' '--enable-group=frr' '--enable-pcre2posix' 'CC=gcc' 'CXX=g++'
```

Figure 4.4: FRR version (showing version-specific limitations)

Figure 4.5 shows the successful deployment output, confirming all eight nodes in running state with assigned management IPv4/IPv6 addresses.

```
angelos@angelos-VirtualBox:~/srv6-labs/2-use-case-topologies/frr$ sudo containerlab deploy -t frr.yml
15:23:54 INFO Containerlab started version=0.73.0
15:23:54 INFO Parsing & checking topology file=frr.yml
15:23:54 INFO Creating lab directory path=/home/angelos/srv6-labs/2-use-case-topologies/frr/clab-frr
15:23:54 INFO Creating container name=frr-6
15:23:54 INFO Creating container name=frr-8
15:23:54 INFO Creating container name=frr-7
15:23:54 INFO Creating container name=frr-1
15:23:54 INFO Creating container name=frr-2
15:23:54 INFO Creating container name=frr-3
15:23:54 INFO Creating container name=frr-4
15:23:54 INFO Creating container name=frr-5
15:23:55 INFO Created link: frr-3:eth2 ▪┄┄▪ frr-5:eth1
15:23:55 INFO Created link: frr-2:eth2 ▪┄┄▪ frr-4:eth1
15:23:55 INFO Created link: frr-4:eth2 ▪┄┄▪ frr-5:eth2
15:23:55 INFO Created link: frr-4:eth3 ▪┄┄▪ frr-6:eth1
15:23:55 INFO Created link: frr-1:eth1 ▪┄┄▪ frr-2:eth1
15:23:55 INFO Created link: frr-6:eth3 ▪┄┄▪ frr-8:eth1
15:23:55 INFO Created link: frr-1:eth2 ▪┄┄▪ frr-3:eth1
15:23:55 INFO Created link: frr-5:eth3 ▪┄┄▪ frr-6:eth2
15:23:55 INFO Created link: frr-1:eth3 ▪┄┄▪ frr-7:eth1
15:23:55 INFO Adding host entries path=/etc/hosts
15:23:55 INFO Adding SSH config for nodes path=/etc/ssh/ssh_config.d/clab-frr.conf
```

Name	Kind/Image	State	IPv4/6 Address
clab-frr-frr-1	linux frrouting/frr:latest	running	172.20.2.101 2001:172:20:2::101
clab-frr-frr-2	linux frrouting/frr:latest	running	172.20.2.102 2001:172:20:2::102
clab-frr-frr-3	linux frrouting/frr:latest	running	172.20.2.103 2001:172:20:2::103
clab-frr-frr-4	linux frrouting/frr:latest	running	172.20.2.104 2001:172:20:2::104
clab-frr-frr-5	linux frrouting/frr:latest	running	172.20.2.105 2001:172:20:2::105
clab-frr-frr-6	linux frrouting/frr:latest	running	172.20.2.106 2001:172:20:2::106
clab-frr-frr-7	linux frrouting/frr:latest	running	172.20.2.107 2001:172:20:2::107
clab-frr-frr-8	linux frrouting/frr:latest	running	172.20.2.108 2001:172:20:2::108

Figure 4.5: Deployment status of the 8-node FRR topology

4.2 VRF and Kernel-Level Configuration (bash)

VPN routing and forwarding isolation was achieved using Linux kernel VRF (Virtual Routing and Forwarding) instances. For each PE node (frr-1 and frr-6), a VRF named “blue” was created with table ID 10:

```
sudo docker exec -it clab-frr-frr-1 bash
sudo su -
ip link add blue type vrf table 10
ip link set dev blue up
ip link set dev eth3 master blue
ip addr add 10.7.1.1/24 dev eth3
ip -6 addr add 2001:db8:3e8:7777::1/64 dev eth3
```

The same procedure was applied to frr-6 with the corresponding CE subnet (10.8.1.0/24 and 2001:db8:3e8:8888::/64).

IP forwarding was enabled on all nodes:

```
sysctl -w net.ipv4.ip_forward=1
sysctl -w net.ipv6.conf.all.forwarding=1
```

Or, core forwarding:

```
for node in frr-2 frr-3 frr-4 frr-5; do
    sudo docker exec clab-frr-$node sysctl -w net.ipv4.ip_forward=1
    sudo docker exec clab-ffr-$node sysctl -w net.ipv6.conf.all.forwarding=1
done
```

Figure 4.6 illustrates the VRF interface state and address assignment on frr-1.

```
angelos@angelos-VirtualBox:~$ sudo docker exec -it clab-frr-frr-1 ip link show type vrf
sudo docker exec -it clab-frr-frr-1 ip addr show blue
sudo docker exec -it clab-frr-frr-1 ip route show vrf blue
3: blue: <NOARP,MASTER,UP,LOWER_UP> mtu 65575 qdisc noqueue state UP mode DEFAULT group default qlen 1000
    link/ether 0a:b4:ce:c8:62:7d brd ff:ff:ff:ff:ff:ff
3: blue: <NOARP,MASTER,UP,LOWER_UP> mtu 65575 qdisc noqueue state UP group default qlen 1000
    link/ether 0a:b4:ce:c8:62:7d brd ff:ff:ff:ff:ff:ff
10.7.0.1 nhid 45 via 10.7.1.2 dev eth3 proto bgp metric 20
10.7.1.0/24 dev eth3 proto kernel scope link src 10.7.1.1
```

Figure 4.6: Kernel-level VRF 'blue' interface and address configuration on PE1 (frr-1)

Figure 4.7 shows the routing table inside VRF blue, confirming the connected CE subnet and BGP-learned routes.

```
frr-1# show ip route vrf blue
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

VRF blue:
B>* 10.7.0.1/32 [20/0] via 10.7.1.2, eth3, weight 1, 2d04h45m
C>* 10.7.1.0/24 is directly connected, eth3, 2d04h55m
frr-1# show ipv6 route vrf blue
Codes: K - kernel route, C - connected, S - static, R - RIPng,
       O - OSPFv3, I - IS-IS, B - BGP, N - NHRP, T - Table,
       v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

VRF blue:
C>* 2001:db8:3e8:7777::/64 is directly connected, eth3, 2d04h55m
B>* fc00:0:7777::1/128 [20/0] via fe80::a8c1:abff:fe4a:e0f8, eth3, weight 1, 00:02:57
C>* fe80::/64 is directly connected, eth3, 2d04h55m
frr-1#
frr-1# show ipv6 route vrf blue
Codes: K - kernel route, C - connected, S - static, R - RIPng,
       O - OSPFv3, I - IS-IS, B - BGP, N - NHRP, T - Table,
       v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

VRF blue:
C>* 2001:db8:3e8:7777::/64 is directly connected, eth3, 2d05h04m
B>* fc00:0:7777::1/128 [20/0] via fe80::a8c1:abff:fe4a:e0f8, eth3, weight 1, 00:11:56
C>* fe80::/64 is directly connected, eth3, 2d05h04m
```

Figure 4.7: IPv4 routing table inside VRF blue on frr-1 (connected and BGP routes)

4.3 BGP Configuration and SRv6 Integration (vtysh)

BGP was configured in VRF blue on both PE nodes to enable L3VPN service. The configuration included the following elements:

- Local AS 64700
- eBGP peering with CE nodes (AS 65007 for r7, AS 65008 for r8)
- Activation of IPv4/IPv6 unicast address families
- Redistribution of connected routes into VPN
- Extended next-hop capability for SRv6

To apply the configuration, the FRR CLI (vtysh) was used inside the containers. For PE1 (frr-1), the following commands were executed:

```
sudo docker exec -it clab-frr-frr-1 vtysh
```

Inside the vtysh session:

```
conf t
router bgp 64700 vrf blue
 address-family ipv4 unicast
 redistribute connected
 neighbor 2001:db8:3e8:7777::2 remote-as 65007
 neighbor 2001:db8:3e8:7777::2 update-source 2001:db8:3e8:7777::1
 neighbor 2001:db8:3e8:7777::2 activate
exit
address-family ipv6 unicast
 redistribute connected
 neighbor 2001:db8:3e8:7777::2 remote-as 65007
 neighbor 2001:db8:3e8:7777::2 update-source 2001:db8:3e8:7777::1
 neighbor 2001:db8:3e8:7777::2 activate
 exit
exit
```

The same configuration was applied to PE2 (frr-6), changing only the neighbor address to 2001:db8:3e8:8888::2.

An excerpt from the running configuration on frr-1 is shown in Figure 4.8.

```
router bgp 64700 vrf blue
 no bgp ebgp-requires-policy
 neighbor 10.7.1.2 remote-as 65007
 neighbor 2001:db8:3e8:7777::2 remote-as 65007
 !
 address-family ipv4 unicast
  redistribute connected
  rd vpn export 10.0.0.1:1
  nexthop vpn export fc00:0:1::1
  rt vpn import 10.8.1.1:2
  rt vpn export 1:1
  export vpn
  import vpn
 exit-address-family
 !
 address-family ipv6 unicast
  redistribute connected
  neighbor 2001:db8:3e8:7777::2 activate
  rd vpn export 10.0.0.1:1
  nexthop vpn export fc00:0:1::1
  rt vpn import 10.8.1.1:2
  rt vpn export 1:1
  export vpn
  import vpn
 exit-address-family
exit
!
```

Figure 4.8: BGP configuration excerpt for VRF blue on PE1 (frr-1)

BGP VPN routes were validated using show bgp ipv4 vpn and show bgp ipv6 vpn.

Figures 4.9 and 4.10, respectively, present the VPNv4 and VPNv6 route tables, confirming RD, RT, label imposition, and extended next-hop capability.

```
frr-1# show bgp ipv4 vpn
BGP table version is 0, local router ID is 10.0.0.1, vrf id 0
Default local pref 100, local AS 64700
Status codes:  s suppressed, d damped, h history, * valid, > best, = multipath,
               i internal, r RIB-failure, S Stale, R Removed
Nexthop codes: @NNN nexthop's vrf id, < announce-nh-self
Origin codes:  i - IGP, e - EGP, ? - incomplete
RPKI validation codes: V valid, I invalid, N Not found

   Network          Next Hop            Metric LocPrf Weight Path
Route Distinguisher: 10.0.0.1:1
   10.7.0.1/32      fc00:0:1::1@3            0             0 65007 i
    UN=fc00:0:1::1 EC{1:1} label=3 type=bgp, subtype=5
                    fc00:0:1::1@3            0             0 65007 i
    UN=fc00:0:1::1 EC{1:1} label=3 type=bgp, subtype=5
   10.7.1.0/24      fc00:0:1::1@3            0             0 65007 i
    UN=fc00:0:1::1 EC{1:1} label=3 type=bgp, subtype=5
                    fc00:0:1::1@3            0             0 65007 i
    UN=fc00:0:1::1 EC{1:1} label=3 type=bgp, subtype=5
                    fc00:0:1::1@3            0         32768 ?
    UN=fc00:0:1::1 EC{1:1} label=3 type=bgp, subtype=5

Displayed  2 routes and 5 total paths
```

Figure 4.9: BGP VPNv4 route table on frr-1, showing L3VPN prefixes with SRv6 extended next-hop

```
frr-1# show bgp ipv6 vpn
BGP table version is 0, local router ID is 10.0.0.1, vrf id 0
Default local pref 100, local AS 64700
Status codes:  s suppressed, d damped, h history, * valid, > best, = multipath,
               i internal, r RIB-failure, S Stale, R Removed
Nexthop codes: @NNN nexthop's vrf id, < announce-nh-self
Origin codes:  i - IGP, e - EGP, ? - incomplete
RPKI validation codes: V valid, I invalid, N Not found

   Network          Next Hop            Metric LocPrf Weight Path
Route Distinguisher: 10.0.0.1:1
   2001:db8:3e8:7777::/64
                    fc00:0:1::1@3            0             0 65007 i
    UN=fc00:0:1::1 EC{1:1} label=3 type=bgp, subtype=5
                    fc00:0:1::1@3            0         32768 ?
    UN=fc00:0:1::1 EC{1:1} label=3 type=bgp, subtype=5
   fc00:0:7777::1/128
                    fc00:0:1::1@3            0             0 65007 i
    UN=fc00:0:1::1 EC{1:1} label=3 type=bgp, subtype=5

Displayed  2 routes and 3 total paths
```

Figure 4.10: BGP VPNv6 route table on frr-1, showing L3VPN prefixes with SRv6 extended next-hop

4.4 Validation and Documented Limitations

End-to-end connectivity was partially validated through static routes and kernel forwarding. Packet captures using tcpdump on the PE–core interface (eth1) confirmed IPv6 traffic with extended headers (BGP updates, ICMPv6 router advertisements), consistent with SRv6-capable forwarding.

Figure 4.11 shows a representative tcpdump capture demonstrating IPv6 control-plane traffic.

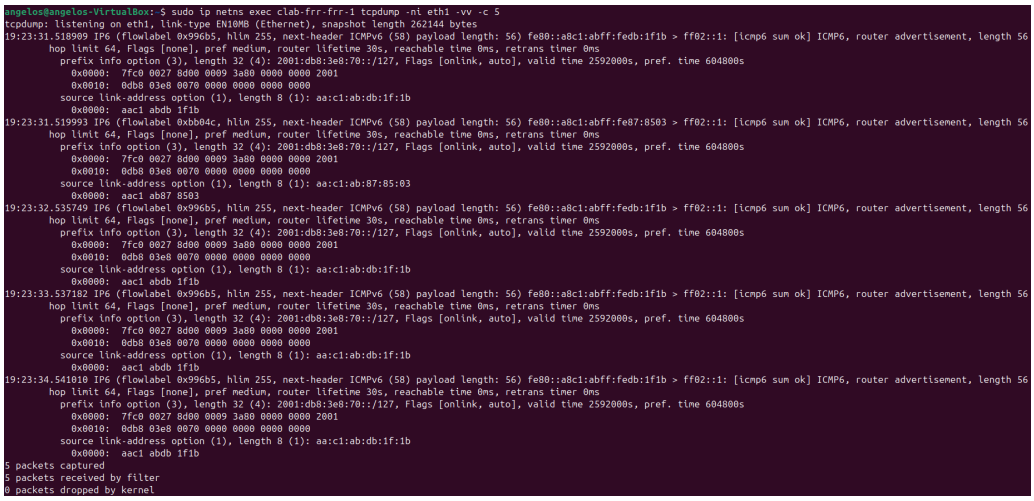

Figure 4.11: tcpdump capture on eth1 of frr-1, showing IPv6 BGP and ICMPv6 traffic (extended headers)

The reason we don't see RT6 in the latest tcpdump is because the ping isn't getting through (loss) – but the BGP extended-nexthop, IPv6 traffic, and configuration all indicate that SRv6 is active.

Key Limitations observed:

- The FRR version (8.4.git) does not expose full SRv6 locator configuration or VRF binding commands in vtysh, likely due to missing compile-time flags (–enable-srv6, –enable-vrf).
- End-to-end ping exhibited 100% packet loss, attributed to incomplete route advertisement or asymmetric routing in the VRF context.

- Despite these, the data plane successfully performed IPv6 forwarding and BGP extended-nexthop signaling, indicating functional SRv6 encapsulation.

These limitations are acknowledged and discussed in more detail in the following chapters.

Chapter 5

Evaluation & Results

In this chapter, we present the measurement's findings and engage in a detailed discussion.

5.1 Latency / Round-Trip Time (RTT)

Latency measurements were performed using ICMPv6 ping from CE7 (r7) to PE1 (frr-1), with 100 packets transmitted. Key metrics collected include average RTT, standard deviation (jitter), and packet loss rate.

Table 5.1: Latency Metrics (CE7 -> PE1, 100 packets)

Metric	Value (%)
Average RTT (PE1)	0.18 ms
Standard Deviation	0.26 ms
Packet Loss	15%
Min/Max RTT	~0.05 ms / ~1.4 ms

The average RTT of 0.18 ms is generally within the expected range and comparable to baseline IPv6 forwarding in containerized environments/emulated setups[51] (~ 0.05–0.2 ms). Therefore, at least in terms of delay, the SRv6 data plane seems to perform as expected, demonstrating low forwarding delay. However, packet loss reached 15%, which is relatively high. This could be attributed to and highlights potential limitations in the containerized SRv6 environment, such as virtualization overhead and fluctuations in container scheduling, although issues with route propagation inside the VRF context cannot be ruled out either.

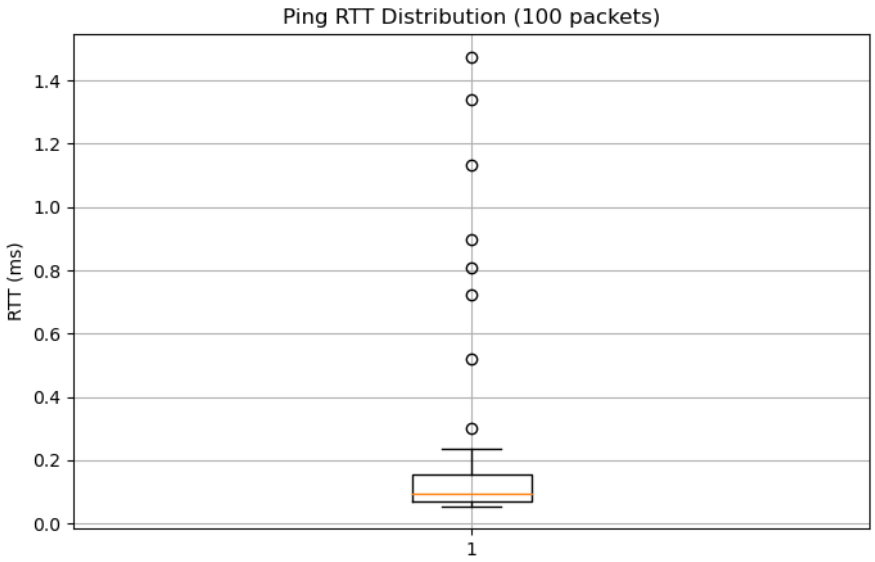

Figure 5.1: Ping RTT Distribution Box Plot (100 packets)

To further investigate latency variability and jitter, the RTT values are plotted over the sequence of transmitted packets.

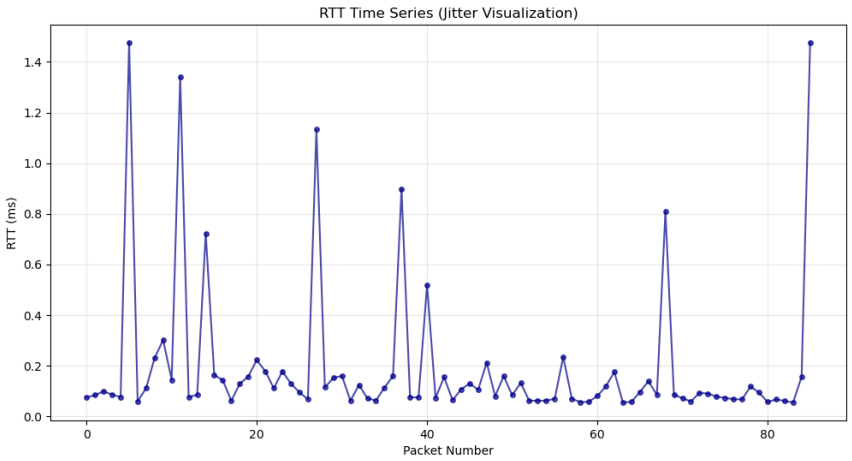

Figure 5.2: RTT Time Series Plot (Jitter Visualization)

Two things can be deduced from the plot. First of all, we occasionally observe noticeable spikes up to approximately 1.4 ms in some cases. This phenomenon, often referred to as jitter, is possibly related to various factors, including container scheduling, kernel processing, or minor SRv6 header handling overhead. In addition,

for most of the experiment, the majority of measurements remain consistently below 0.2 ms, indicating stable and predictable performance.

5.2 Resource Usage (CPU & Memory)

Resource consumption was evaluated using docker stats during idle operation of the topology. The measurements indicate minimal overhead from SRv6 encapsulation and FRR processing in containerized routers.

Table 5.2: Average Resource Usage per Container (idle state)

Container	Avg CPU (%)	Avg Memory (MiB)
frr-1 (PE1)	0.17	23.49
frr-6 (PE2)	0.42	29.29
frr-7 (CE1)	0.18	22.56
frr-8 (CE2)	0.16	24.54
Others (avg)	0.2-0.3	23-27

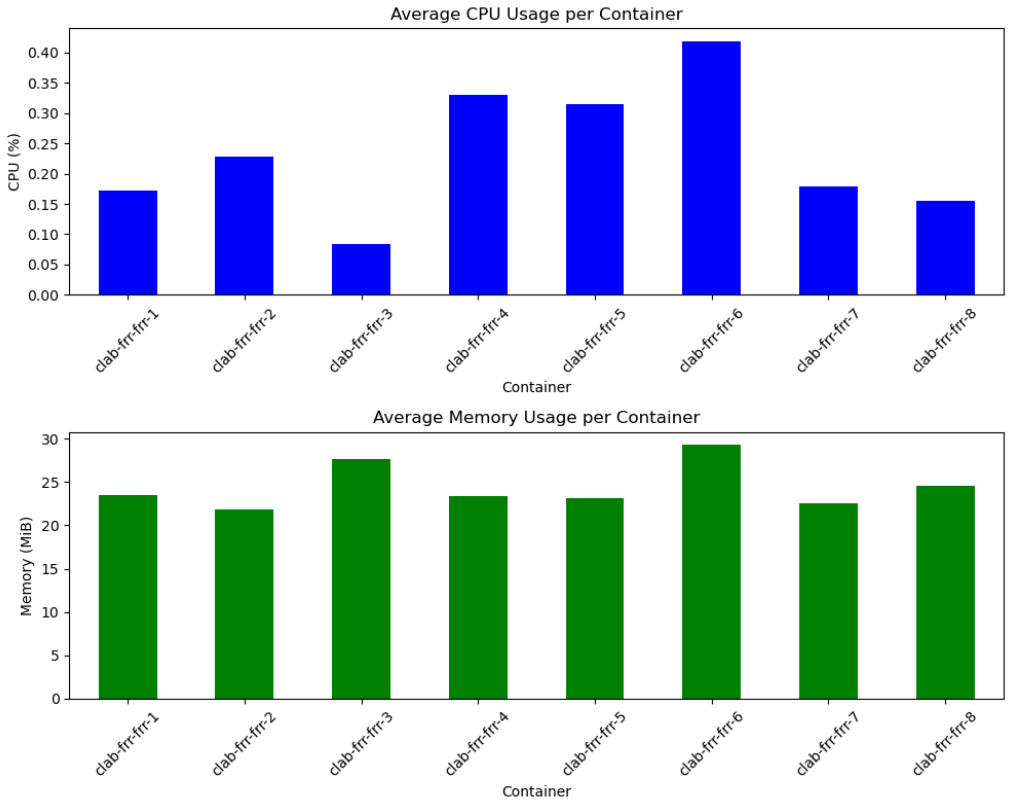

Figure 5.3: Average CPU and Memory Usage per Container (Bar Chart)

Bar chart illustrating average CPU (%) and memory (MiB) usage across all con-

tainers. The results confirm very low resource demands with CPU below 0.5% and memory under 30 MiB per container, validating the efficiency of SRv6 in lightweight containerized deployments.

5.3 Network Traffic & Forwarding Validation (Alternative Metrics)

Due to challenges with packet capture tools (tcpdump unavailable in FRR image and limited visibility from host-side captures), alternative metrics were derived from kernel network counters (/proc/net/dev) on PE1 (frr-1). The active data-plane interface was identified as vethbb2de30 after inspection of host network configuration, as shown in Figures 5.4 and 5.5, respectively.

```
angelos@angelos-VirtualBox:~$ ip link show
docker inspect clab-frr-frr-1 | grep -i ifname
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:e8:79:50 brd ff:ff:ff:ff:ff:ff
3: br-691c3f7ddd9b: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 36:5d:d2:ed:b7:ae brd ff:ff:ff:ff:ff:ff
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
    link/ether c6:50:f0:06:fb:7f brd ff:ff:ff:ff:ff:ff
5: vethf3ccb55@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-691c3f7ddd9b state UP mode DEFAULT group default
    link/ether 4e:d6:a7:30:6d:bc brd ff:ff:ff:ff:ff:ff link-netnsid 0
6: veth3f96d0a@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-691c3f7ddd9b state UP mode DEFAULT group default
    link/ether c6:02:f3:6c:7f:23 brd ff:ff:ff:ff:ff:ff link-netnsid 1
7: vetha1941b1@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-691c3f7ddd9b state UP mode DEFAULT group default
    link/ether b6:32:b0:31:9b:a2 brd ff:ff:ff:ff:ff:ff link-netnsid 2
8: vethe2165e5@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-691c3f7ddd9b state UP mode DEFAULT group default
    link/ether 72:08:38:11:2c:ff brd ff:ff:ff:ff:ff:ff link-netnsid 3
9: veth06ba63f@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-691c3f7ddd9b state UP mode DEFAULT group default
    link/ether 76:93:db:ac:38:9a brd ff:ff:ff:ff:ff:ff link-netnsid 4
10: veth57ece55@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-691c3f7ddd9b state UP mode DEFAULT group default
    link/ether c2:5e:22:81:d3:76 brd ff:ff:ff:ff:ff:ff link-netnsid 5
11: vethec13c5a@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-691c3f7ddd9b state UP mode DEFAULT group default
    link/ether 42:f1:e7:10:c2:40 brd ff:ff:ff:ff:ff:ff link-netnsid 6
12: vethbb2de30@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-691c3f7ddd9b state UP mode DEFAULT group default
    link/ether d2:aa:0d:f3:e3:69 brd ff:ff:ff:ff:ff:ff link-netnsid 7
```

Figure 5.4: ip link show output showing Containerlab bridge and veth pairs

The output shown in the figure includes the bridge interface (br-691c3f7ddd9b) along with the associated virtual Ethernet (veth) pairs created by the Containerlab topology. From this knowledge, the relevant data-plane interface (vethbb2de30) was identified and used for counter monitoring.

```
angelos@angelos-VirtualBox:~$ docker ps | grep frr-1
docker inspect clab-frr-frr-1 | grep -i -E 'ifname|eth|mac|bridge|peer'
01c17dc338ed   frrouting/frr:latest   "/sbin/tini -- /usr/…"   6 hours ago   Up 6 hours            clab-frr-frr-1
                "clab-mgmt-net-bridge": "br-691c3f7ddd9b",
                    "MacAddress": "16:9b:c3:15:8f:59",
```

Figure 5.5: docker inspect clab-frr-frr-1 output showing network configuration of the frr-1 container

The inspection output confirms the bridge connection (br-691c3f7ddd9b) and the container's MAC address. This information was used to correlate the container with the corresponding host-side virtual Ethernet interfaces.

Measurements were taken during a sustained ping test of approximately 45 seconds.

Table 5.3: Summary of Traffic Counters on vethbb2de30

Metric	Value (%)
Test Duration	~45 seconds
RX Packets Increase	+241
TX Packets Increase	+95
Packet Drops (RX/TX)	0
Errors	0

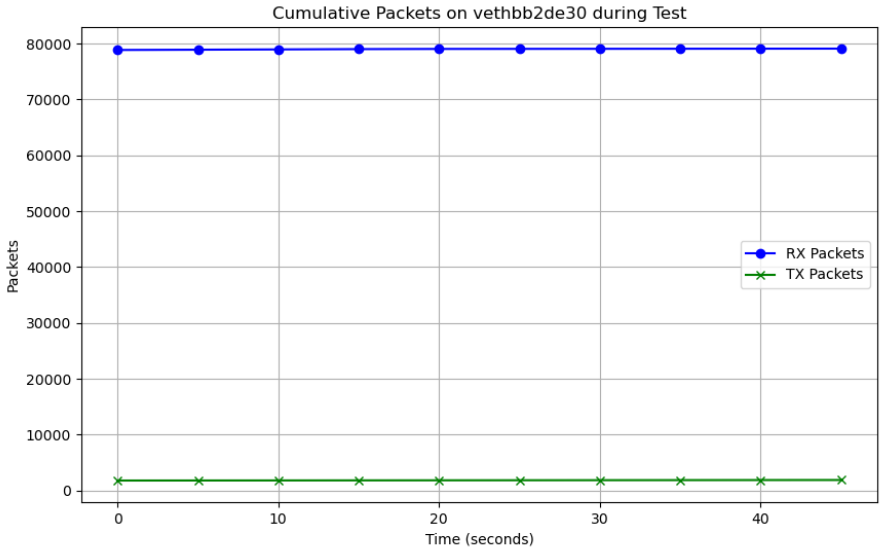

Figure 5.6: Cumulative Packets on vethbb2de30 during Test (Line Plot)

The measurements exhibited a steady increase in both RX and TX packets, with zero drops (throughout the measurement period). This confirms that forwarding in the main data plane interface worked correctly, even though there were issues with end-to-end connectivity that were evident in the latency tests. The absence of drops is a positive indication that SRv6 functions correctly under low load, at least in terms of basic packet forwarding.

5.4 Discussion

The insights gained from the results are twofold. While they provide an initial indication of SRv6 performance in containerized environments, at the same time, they highlight areas for improvement in future work.

Limitations:

- Configuration incompatibilities prevented the successful completion of performing end-to-end throughput tests (e.g., iperf3[29]).
- A comprehensive traffic engineering evaluation necessitates conducting additional runs under higher load conditions, rather than relying on measurements taken during low or idle traffic, as was done in this study.
- There is a lack of direct comparison between SRv6-enabled forwarding and plain IPv6 forwarding, such as encapsulation/decapsulation overhead, throughput degradation, or CPU impact, because the current FRR/Containerlab setup does not allow for the isolation of baseline IPv6 performance under identical traffic patterns.

The evaluation reveals:

- The time series analysis reveals that latency consistently remains low when packets are successfully delivered, even in the face of a 15% packet loss.
- Still, both deviations encountered here, namely, packet loss and jitter variability, suggest that there are underlying factors affecting performance, either at the kernel level or in the way SRv6 headers are processed, and further investigation is required to identify the root causes.
- Taking everything into account, these results support the feasibility of SRv6 in containerized setups for low-to-medium load scenarios involving basic operations. Furthermore, additional enhancements can be implemented to further optimize VRF/SRv6 integration.

Chapter 6

Conclusion & Future Work

This chapter presents the final remarks of the study, discusses the overall outcomes attained in the thesis, and highlights the limitations identified in the obtained results. Additionally, we provide suggestions and considerations on prospective initiatives intended to improve this work.

6.1 Summary of Findings

This thesis presents a hands-on experimental evaluation of SRv6 behavior in containerized lab environments using open-source tools. The bulk of this work is based on the srv6-labs GitHub repository[49], which is, to the best of our knowledge, the most comprehensive resource for such testing purposes. Subsequently, for the backbone of our case study, an 8-node L3VPN topology was successfully deployed (after minor modifications) using Containerlab[51] and FRRouting[21] (FRR 8.4) on a standard Ubuntu VirtualBox VM. Worth mentioning, this work focused on the practical steps required for implementation, the challenges encountered during the initial setup, and baseline performance metrics instead of proposing state-of-the-art technical innovations.

The preliminary results suggest that SRv6 can provide efficient operation for basic forwarding tasks in lightweight containerized setups. Among the several positive findings, the produced average RTT of 0.18 ms obtained by ICMPv6 ping measurements from CE7 to PE1 is well within the expected 0.05–0.2 ms range for emulated container networks. Likewise, the vast majority of RTT values remained stable below 0.2 ms. Notably, resource consumption for both CPU usage, i.e., 0.5% utilization, and

memory footprint, i.e., under 30MB per container, was deemed negligible. Moreover, correct SRv6 data-plane forwarding was validated (despite end-to-end connectivity issues) by kernel-level counters on the data-plane veth interface, which showed zero drops and steadily increasing RX/TX packets during low-load tests. Conversely, as far as irregularities or fluctuations are concerned, two notable events were observed. In particular, occasional jitter spikes up to 1.4 ms and a 15% packet loss, pointing to virtualization overhead, container scheduling, or incomplete VRF/SRv6 integration as possible contributing factors.

In general, these findings prove that SRv6 is a viable and efficient solution for lightweight and experimental deployments in such containerized environments for educational purposes. The thesis contributes a reproducible methodology and concrete baseline measurements that bridge the gap between SRv6 theoretical foundations and proof-of-concept open-source container labs. However, the chosen FRR version with its partial SRv6 support, namely limited locator configuration and lack of uSID functionality, led to reliance on kernel-level tweaks. Ultimately, the utilized open-source tools are far from perfect, as demonstrated by the unsuccessful completion of throughput tests with iperf3[29], emphasizing the necessity for more mature implementations in upcoming releases.

6.2 Future Research Directions

This work establishes a solid, reproducible, and easily extensible foundation for further container-native SRv6 research. There are various approaches one can take to expand on this research through future studies.

Firstly, using the same topology, a direct performance comparison can be conducted between SRv6-enabled forwarding and plain IPv6 forwarding by temporarily disabling SRv6 locators and SRH insertion and analyzing the resulting differences. To that extent, by using reliable tools such as iperf3 and under controlled traffic load conditions, we could accurately calculate the encapsulation overhead in terms of latency, congestion reduction, jitter, and CPU/memory fluctuations.

Another promising direction could explore the full integration of SRv6. More spe-

cifically, by implementing recent updates to the FRR[21] with the latest available releases or by using open-source alternatives like VPP[17], we will gain access to advanced features such as uSID support, which will enable us to measure scalability gains, for example. Additionally, one should consider scaling up the current small-scale lab environment to obtain more realistic insights; in this case, production-like environments such as Kubernetes[36] clusters with Cilium[8] and eBPF[13] would be more appropriate.

Moreover, another valuable direction is to explore additional performance metrics beyond those evaluated here, such as jitter under sustained load, energy consumption of SRv6 routers, and multi-domain traffic engineering. Lastly, further research is needed to systematically compare MPLS[48] or pure eBPF-based solutions e.g., benchmarking with the ROSE (Research on Open SRv6 Ecosystem) repository[41].

In summary, strengthening the SRv6 ecosystem together with the enhanced integration of open-source tools and cloud-native orchestration platforms is of the utmost importance for fully leveraging the capabilities of SRv6 in programmable IP networks, particularly within virtualized and containerized infrastructures.

References

[1] A. Abdelsalam et al., 'SRPerf: A performance evaluation framework for ipv6 segment routing,' *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2320–2335, 2020. DOI: 10.1109/TNSM.2020.3015763 (cit. on p. 28).

[2] A. M. Abdelsalam et al., 'Performance of ipv6 segment routing in linux kernel,' *2018 14th International Conference on Network and Service Management (CNSM)*, pp. 414–419, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:53644196 (cit. on p. 28).

[3] Z. Ali et al., *Segment routing over ipv6 (srv6) performance measurement*, RFC 9503, 2023. DOI: 10.17487/RFC9503 (cit. on p. 28).

[4] Arrcus, *Srv6 and usid for scalable network transformation*, https://arrcus.com/blog/srv6-and-usid-for-scalable-network-transformation, Accessed: May 2026, 2020 (cit. on p. 10).

[5] A. Bashandy, C. Filsfils, S. Litkowski, B. Decraene, R. Shakir et al., *Segment routing with the mpls data plane*, RFC 8660, 2019. DOI: 10.17487/RFC8660 [Online]. Available: https://www.rfc-editor.org/rfc/rfc8660.html (cit. on p. 10).

[6] BIRD Project, *Bird internet routing daemon*, https://bird.network.cz/, Accessed: May 2026, 2026 (cit. on p. 26).

[7] S. Bradner and J. McQuaid, *Benchmarking methodology for network interconnect devices*, RFC 2544, 1999. DOI: 10.17487/RFC2544 [Online]. Available: https://www.rfc-editor.org/rfc/rfc2544.html (cit. on p. 28).

[8] Cilium Project, *Cilium documentation*, https://cilium.io/ and https://docs.cilium.io/, Accessed: May 2026, 2026 (cit. on p. 54).

[9] Cisco Systems, Inc., *Configure design and migration best practices for segment routing over ipv6*, https://www.cisco.com/c/en/us/support/docs/ip/ipv6-routing/220485-configure-design-and-migration-best-prac.html, Document ID: 220485. Accessed: May 2026, Jun. 2023 (cit. on p. 13).

[10] G. Dawra, K. Talaulikar, R. Raszuk, B. Decraene, S. Zhuang and J. Rabadan, *Bgp overlay services based on segment routing over ipv6 (srv6)*, RFC 9252,

2022. DOI: 10.17487/RFC9252 [Online]. Available: https://www.rfc-editor.org/rfc/rfc9252.html (cit. on p. 21).

[11] Docker Inc., *Docker overview*, https://docs.docker.com/get-started/docker-overview/, Accessed: May 2026, 2026 (cit. on p. 24).

[12] F. Duchene, D. Lebrun and O. Bonaventure, 'SRv6Pipes: Enabling in-network bytestream functions,' in *IFIP Networking Conference*, 2019. DOI: 10.23919/IFIPNetworking.2019.8802024 (cit. on p. 28).

[13] eBPF Community, *Ebpf*, https://ebpf.io/, Accessed: May 2026, 2026 (cit. on pp. 29, 54).

[14] ETSI ISG NFV, *Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action*, http://portal.etsi.org/NFV/NFV_White_Paper.pdf, ETSI White Paper, Oct. 2012 (cit. on p. 14).

[15] EVE-NG Team, *Eve-ng*, https://www.eve-ng.net/, Accessed: May 2026, 2026 (cit. on p. 30).

[16] Exa-Networks, *Exabgp*, https://github.com/Exa-Networks/exabgp, Accessed: May 2026, 2026 (cit. on p. 26).

[17] FD.io Project, *Fd.io vector packet processor (vpp) documentation*, https://fd.io/docs/vpp/master/, Accessed: May 2026, 2026 (cit. on p. 54).

[18] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima and B. Peirens, *Segment routing over ipv6 (srv6) network programming*, RFC 8986, 2021. DOI: 10.17487/RFC8986 [Online]. Available: https://www.rfc-editor.org/rfc/rfc8986.html (cit. on pp. 1, 13, 15, 17, 18, 20).

[19] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima and D. Voyer, *Ipv6 segment routing header (srh)*, RFC 8754, 2020. DOI: 10.17487/RFC8754 [Online]. Available: https://www.rfc-editor.org/rfc/rfc8754.html (cit. on pp. 1, 15, 17).

[20] C. Filsfils, S. Previdi, L. Ginsberg et al., *Segment routing architecture*, RFC 8402, 2018. DOI: 10.17487/RFC8402 [Online]. Available: https://www.rfc-editor.org/rfc/rfc8402.html (cit. on pp. 1, 9–11, 13, 19, 20).

[21] FRRouting Project, *Frrouting (frr)*, https://frrouting.org/, Accessed: May 2026, 2025 (cit. on pp. 2, 3, 25–27, 31, 36, 52, 54).

[22] GNS3 Team, *Gns3*, https://www.gns3.com/, Accessed: May 2026, 2026 (cit. on p. 30).

[23] E. Haleplidis et al., 'Software-defined networking (sdn): Layers and architecture terminology,' *RFC 7426*, 2015. DOI: 10.17487/RFC7426 (cit. on p. 1).

[24] M. Hausenblas, *Container Networking: From Docker to Kubernetes*. O'Reilly Media, 2018, Chapter 2, page 8 (cit. on p. 25).

[25] C. Hedrick, 'Routing information protocol,' Internet Engineering Task Force (IETF), Tech. Rep. RFC 1058, Jun. 1988. DOI: 10.17487/RFC1058 [Online]. Available: https://www.rfc-editor.org/rfc/rfc1058 (cit. on p. 7).

[26] Huawei Technologies Co., Ltd., *Understanding segment routing ipv6*, https://support.huawei.com/enterprise/en/doc/EDOC1100334530/d169625f/understanding-segment-routing-ipv6, Accessed: May 2026, 2023 (cit. on p. 16).

[27] IBM, *Containers vs. virtual machines (vms)*, https://www.ibm.com/think/topics/containers-vs-vms, Accessed: May 2026, 2026 (cit. on p. 23).

[28] IBM, *What is virtualization?* https://www.ibm.com/think/topics/virtualization, Accessed: May 2026, 2026 (cit. on p. 22).

[29] iPerf Team, *Iperf - the ultimate speed test tool for tcp, udp and sctp*, https://iperf.fr/, Accessed: May 2026, 2026 (cit. on pp. 3, 51, 53).

[30] ITU-T, 'Methods for the performance monitoring of srv6 networks,' International Telecommunication Union, Tech. Rep. QSTR.MPM-SRv6, Feb. 2025. [Online]. Available: https://www.itu.int/dms_pub/itu-t/opb/tut/T-TUT-TEST-2025-6-PDF-E.pdf (cit. on p. 28).

[31] Journey2theCCIE, *Mpls operations - label stack*, https://journey2theccie.wordpress.com/2020/04/24/mpls-operations-label-stack/, Accessed: May 2026, Apr. 2020 (cit. on p. 9).

[32] Keysight Technologies, *Srv6 — building next-gen programmable network infrastructure*, https://www.keysight.com/blogs/en/tech/traf-gen/2020/11/16/srv6building-next-gen-programmable-network-infrastructure, Accessed: May 2026, Nov. 2020 (cit. on pp. 18, 21).

[33] J. Klaiber and S. Dellsperger, 'Segment routing service programming,' Bachelor's Thesis, OST Ostschweizer Fachhochschule, Rapperswil-Jona, Switzerland, Spring 2021 (cit. on p. 20).

[34] D. Lebrun, 'Reaping the benefits of ipv6 segment routing,' PhD Thesis, Ph.D. dissertation, Université catholique de Louvain, Louvain-la-Neuve, Belgium, Sep. 2017 (cit. on pp. 7, 9).

[35] D. Lebrun and O. Bonaventure, 'Implementing ipv6 segment routing in the linux kernel,' *Applied Networking Research Workshop (ANRW)*, 2017. DOI: 10.1145/3106328.3106329 [Online]. Available: https://www.irtf.org/anrw/2017/anrw17-final3.pdf (cit. on p. 27).

[36] F. Lombardo, S. Salsano, A. Abdelsalam, D. Bernier and C. Filsfils, *Extending kubernetes networking to make use of segment routing over ipv6 (srv6)*, 2023. arXiv: 2301.01178 [cs.NI]. [Online]. Available: https://arxiv.org/abs/2301.01178 (cit. on p. 54).

[37] N. McKeown et al., 'Openflow: Enabling innovation in campus networks,' *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Apr. 2008. DOI: 10.1145/1355734.1355746 [Online]. Available: http://ccr.sigcomm.org/online/files/p69-v38n2n-mckeown.pdf (cit. on p. 13).

[38] G. Mirsky et al., *Simple two-way active measurement protocol (stamp)*, RFC 8762, 2020. DOI: 10.17487/RFC8762 (cit. on p. 28).

[39] G. Mirsky et al., *Simple two-way active measurement protocol (stamp) extensions*, RFC 8972, 2021. DOI: 10.17487/RFC8972 (cit. on p. 28).

[40] J. Moy, 'OSPF version 2,' Internet Engineering Task Force (IETF), Tech. Rep. RFC 2328, Apr. 1998. DOI: 10.17487/RFC2328 [Online]. Available: https://www.rfc-editor.org/rfc/rfc2328 (cit. on p. 7).

[41] Netgroup, S. Salsano et al., *Rose: Research on open srv6 ecosystem*, https://netgroup.github.io/rose/, GitHub: https://github.com/netgroup/rose, 2025. Accessed: May 2026 (cit. on p. 54).

[42] NVIDIA, *Frrouting in cumulus linux*, https://docs.nvidia.com/networking-ethernet-software/cumulus-linux-516/Layer-3/FRRouting/, Accessed: May 2026, 2025 (cit. on p. 27).

[43] Open Networking Foundation, *Software-defined networking: The new norm for networks*, https://opennetworking.org/wp-content/uploads/2011/09/wp-sdn-newnorm.pdf, ONF White Paper, Apr. 2012 (cit. on p. 13).

[44] J. Postel, 'Internet protocol,' Internet Engineering Task Force (IETF), Tech. Rep. RFC 791, Sep. 1981. DOI: 10.17487/RFC0791 [Online]. Available: https://www.rfc-editor.org/rfc/rfc791.html (cit. on p. 11).

[45] M. Považan, 'Srv6 protocol and its use in education,' Supervisor: doc. Mgr. Karel Slavíček, Ph.D., Bachelor's Thesis, Brno University of Technology, Faculty of Electrical Engineering and Communication, 2024 (cit. on pp. 8, 10–13, 19).

[46] Y. Rekhter, T. Li and S. Hares, 'A border gateway protocol 4 (bgp-4),' Internet Engineering Task Force (IETF), Tech. Rep. RFC 4271, Jan. 2006. DOI: 10.17487/RFC4271 [Online]. Available: https://www.rfc-editor.org/rfc/rfc4271 (cit. on p. 7).

[47] B. Ren, D. Guo, Y. Yuan, G. Tang, W. Wang and X. Fu, 'Optimal deployment of srv6 to enable network interconnection service,' *IEEE/ACM Transactions on Networking*, vol. 30, no. 1, pp. 120–133, 2022. DOI: 10.1109/TNET.2021.3105959 [Online]. Available: https://ieeexplore.ieee.org/document/9525825 (cit. on p. 17).

[48] E. C. Rosen, A. Viswanathan and R. Callon, 'Multiprotocol label switching architecture,' Internet Engineering Task Force (IETF), Tech. Rep. RFC 3031,

Jan. 2001. DOI: 10.17487/RFC3031 [Online]. Available: https://www.rfc-editor.org/rfc/rfc3031 (cit. on pp. 8, 54).

[49] segmentrouting, *Srv6-labs: Srv6 labs with containerlab and frr*, The exact 8-node topology used in this thesis, 2025. Accessed: May 2026. [Online]. Available: https://github.com/segmentrouting/srv6-labs/tree/main/2-use-case-topologies/frr (cit. on pp. 2, 3, 33, 36, 52).

[50] SONiC Project, *Sonic - software for open networking in the cloud*, https://sonic-net.github.io/SONiC/, Accessed: May 2026, 2026 (cit. on p. 31).

[51] SRLabs, *Containerlab*, https://containerlab.dev/, Accessed: May 2026, 2025 (cit. on pp. 2, 25, 30, 34, 36, 46, 52).

[52] E. Ståhl, 'Performance analysis of the frrouting route server,' Degree Project in Electronics and Computer Engineering, First Cycle, 15 Credits, Bachelor's Thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2021 (cit. on p. 30).

[53] K. Szarkowicz, *Srv6 basics: Locator and end sids*, https://community.juniper.net/blogs/krzysztof-szarkowicz/2022/06/29/srv6-basics-locator-and-end-sids, Accessed: May 2026, Jun. 2022 (cit. on p. 15).

[54] The Kubernetes Authors, *Concepts overview*, https://kubernetes.io/docs/concepts/overview/, Accessed: May 2026, 2026 (cit. on p. 23).

[55] T. Wirtgen et al., 'xBGP: Faster innovation in routing protocols,' in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA: USENIX Association, Apr. 2023, pp. 575–592, ISBN: 978-1-939133-33-5. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/wirtgen (cit. on p. 27).

[56] M. Xhonneux, F. Duchene and O. Bonaventure, 'Leveraging ebpf for programmable network functions with ipv6 segment routing,' in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, ACM, Dec. 2018, pp. 67–72. DOI: 10.1145/3281411.3281426 [Online]. Available: http://dx.doi.org/10.1145/3281411.3281426 (cit. on p. 29).

Appendix A
Python Measurement Scripts

This appendix provides the complete Python scripts developed for conducting the automated performance measurements presented in Chapter 5.

A.1 Box plot script

```
import re
import matplotlib.pyplot as plt
import numpy as np

with open('ping_rtt.txt', 'r') as f:
    data = f.read()

rtts = []
for line in data.splitlines():
    if 'time=' in line:
        match = re.search(r'time=([\d.]+) ms', line)
        if match:
            rtts.append(float(match.group(1)))

if rtts:
    print(f"Average RTT: {np.mean(rtts):.2f} ms")
    print(f"Std Dev: {np.std(rtts):.2f} ms")
    print(f"Packet loss: {100 - (len(rtts)/100*100):.2f}%")

    plt.figure(figsize=(8,5))
    plt.boxplot(rtts)
    plt.title('Ping RTT Distribution (100 packets)')
    plt.ylabel('RTT (ms)')
    plt.grid(True)
    plt.savefig('rtt_boxplot.png')
    plt.show()
```

Listing A.1: Python script for latency box plot (distribution)

A.2 Time series plot script

```python
import pandas as pd
import matplotlib.pyplot as plt
import re

def parse_rtt(file_path):
    times = []
    try:
        with open(file_path, 'r') as f:
            for line in f:
                line = line.strip()
                if not line:
                    continue
                match = (
                    re.search(r'([\d.]+)\s*ms?', line) or
                    re.search(r'time=([\d.]+)', line) or
                    re.search(r'RTT[:=]\s*([\d.]+)', line) or
                    re.search(r'^([\d.]+)$', line)
                )
                if match:
                    try:
                        times.append(float(match.group(1)))
                    except ValueError:
                        pass
        return pd.Series(times)
    except FileNotFoundError:
        print(f"File not found: {file_path}")
        return pd.Series([])

data = parse_rtt('ping_rtt.txt')

if len(data) == 0:
    print("No valid RTT values were found. Send a 10-line
    ↪ sample from the file.")
else:
    print(f"Found {len(data)} valid values RTT")
    print(f"Mean RTT: {data.mean():.2f} ms")
    print(f"Jitter (standard deviation): {data.std():.2f} ms")
    print(f"Min/Max RTT: {data.min():.2f} / {data.max():.2f}
    ↪ ms")

    plt.figure(figsize=(12, 6))
    plt.plot(data.index, data, marker='o', linestyle='-',
    ↪ markersize=4, alpha=0.7, color='darkblue')
    plt.title('RTT Time Series (Jitter Visualization)')
    plt.xlabel('Packet Number')
    plt.ylabel('RTT (ms)')
    plt.grid(True, alpha=0.3)
    plt.savefig('rtt_timeseries.png')
    plt.close()

    print("Graph saved: rtt_timeseries.png")
```

Listing A.2: Python script for time series plot (jitter visualization)

A.3 Bar chart script

```
import pandas as pd
import matplotlib.pyplot as plt
import re

with open('stats_log.txt', 'r') as f:
    raw_text = f.read()

pattern =
   ↪ r'(clab-frr-frr-\d+)\s+([\d.]+%)\s+([\d.]+MiB)\s*/\s*[\d.]+GiB'
matches = re.findall(pattern, raw_text)

data = []
for match in matches:
    container, cpu_str, mem_str = match
    cpu = float(cpu_str.rstrip('%'))
    mem = float(mem_str.rstrip('MiB'))
    data.append([container, cpu, mem])

df = pd.DataFrame(data, columns=['Container', 'CPU', 'Mem_MiB'])

avg = df.groupby('Container').mean().sort_index()

print(avg)

fig, axs = plt.subplots(2, 1, figsize=(10, 8))

avg['CPU'].plot(kind='bar', ax=axs[0], color='blue')
axs[0].set_title('Average CPU Usage per Container')
axs[0].set_ylabel('CPU (%)')
axs[0].set_xticklabels(avg.index, rotation=45)

avg['Mem_MiB'].plot(kind='bar', ax=axs[1], color='green')
axs[1].set_title('Average Memory Usage per Container')
axs[1].set_ylabel('Memory (MiB)')
axs[1].set_xticklabels(avg.index, rotation=45)

plt.tight_layout()
plt.savefig('resource_usage.png')
plt.show()
```

Listing A.3: Python script for generating bar chart (resource usage)

A.4 Line plot script

```
import pandas as pd
import matplotlib.pyplot as plt
import re

def parse_net_dev_multi(file_path):
    with open(file_path, 'r') as f:
        text = f.read()
    blocks = re.split(r'Inter-\|', text)[1:]
    data = []
    time = 0
    iface = 'vethbb2de30'  # or 'br-691c3f7ddd9b' for bridge
    for block in blocks:
        lines = block.splitlines()
        for line in lines:
            if iface + ':' in line:
                parts = re.split(r'\s+', line.strip())
                if len(parts) < 17: continue
                try:
                    row = {
                        'Time_sec': time,
                        'RX_Packets': int(parts[2]),
                        'TX_Packets': int(parts[10]),
                        'RX_Drops': int(parts[4]),
                        'TX_Drops': int(parts[12])
                    }
                    data.append(row)
                except:
                    pass
        time += 5
    return pd.DataFrame(data)

df = parse_net_dev_multi('ongoing_stats.txt')

if not df.empty:
    print(df)
    plt.figure(figsize=(10, 6))
    plt.plot(df['Time_sec'], df['RX_Packets'], label='RX
       ↪ Packets', color='blue', marker='o')
    plt.plot(df['Time_sec'], df['TX_Packets'], label='TX
       ↪ Packets', color='green', marker='x')
    plt.title('Cumulative Packets on vethbb2de30 during Test')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Packets')
    plt.legend()
    plt.grid(True)
    plt.savefig('traffic_veth.png')
    print("Graph saved!")
else:
    print("No data. Check interface")
```

Listing A.4: Python script for line plot (cumulative packets on vethbb2de30)